Roberto Grossi
Fabrizio Sebastiani
Fabrizio Silvestri (Eds.)

LNCS 7024

String Processing and Information Retrieval

18th International Symposium, SPIRE 2011
Pisa, Italy, October 2011
Proceedings

Lecture Notes in Computer Science 7024

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Roberto Grossi Fabrizio Sebastiani
Fabrizio Silvestri (Eds.)

String Processing and Information Retrieval

18th International Symposium, SPIRE 2011
Pisa, Italy, October 17-21, 2011
Proceedings

Springer

Volume Editors

Roberto Grossi
Università di Pisa
Dipartimento di Informatica
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
E-mail: grossi@di.unipi.it

Fabrizio Sebastiani
Fabrizio Silvestri
Istituto di Scienza e Tecnologia dell'Informazione
"Alessandro Faedo"
Consiglio Nazionale delle Ricerche
Area della Ricerca di Pisa
Via Giuseppe Moruzzi 1
56124 Pisa, Italy
E-mail:{fabrizio.sebastiani; fabrizio.silvestri}@isti.cnr.it

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24582-4 e-ISBN 978-3-642-24583-1
DOI 10.1007/978-3-642-24583-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011937624

CR Subject Classification (1998): H.3, J.3, H.2.8, I.5, I.2.7, H.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

In the 18 years since its inauguration back in 1993 the International Symposium on String Processing and Information Retrieval (SPIRE) has become the reference meeting for the interdisciplinary community of researchers whose activity lies at the crossroads of string processing and information retrieval. This volume contains the proceedings of SPIRE 2011, the 18th symposium in the series. The first four events concentrated mainly on string processing, and were held in South America under the title "South American Workshop on String Processing" (WSP) in 1993 (Belo Horizonte, Brazil), 1995 (Valparaiso, Chile), 1996 (Recife, Brazil), and 1997 (Valparaiso, Chile). WSP was renamed SPIRE in 1998 (Santa Cruz, Bolivia) when the scope of the event was broadened to include information retrieval. The change was motivated by the increasing relevance of information retrieval and its close interrelationship with the general area of string processing. From 1999 to 2007, the venue of SPIRE alternated between South / Latin America (odd years) and Europe (even years), with Cancun, Mexico in 1999; A Coruña, Spain in 2000; Laguna de San Rafael, Chile in 2001; Lisbon, Portugal in 2002; Manaus, Brazil in 2003; Padova, Italy in 2004; Buenos Aires, Argentina in 2005; Glasgow, UK in 2006; and Santiago, Chile in 2007. This pattern was broken when SPIRE 2008 was held in Melbourne, Australia, but it was restarted in 2009 when the venue was Saariselkä, Finland, followed by Los Cabos, Mexico in 2010.

The SPIRE 2011 call for papers resulted in the submission of 102 papers. Each submitted paper was reviewed by three of the 64 members of the Program Committee, who eventually engaged in discussions coordinated by the two PC Chairmen in cases of lack of consensus. We believe this resulted in a very accurate selection of the truly best submitted papers. As a result, 30 long papers and 10 short papers were accepted and have been published in these proceedings.

The dense program of SPIRE 2011 started on October 17 with four tutorials providing in-depth coverage of both introductory as well as advanced topics in string processing ("Introduction to Sequence Learning", by Corinna Cortes and Mehryar Mohri, and "Space-Efficient Data Structure", by Francisco Claude and Gonzalo Navarro) and information retrieval ("Introduction to Web Retrieval" by Ricardo Baeza-Yates, and "Computational Geography", by Vanessa Murdock and Gary Gale). The main conference featured keynote speeches by Erik Demaine and Abdur Chowdhury, plus the presentations of the 30 full papers and 10 short papers. Following the main conference, on October 21, SPIRE 2011 hosted two workshops, i.e., the Workshop on the Algorithmic Analysis of Biological Data (WAABD 2011) and the Workshop on Compression, Text, and Algorithms (WCTA 2011). A Best Paper Award and a Best Student Paper Award were also assigned, each consisting of a check of 1000 EUR and sponsored by Google and NoemaLife, respectively.

We would like to take the opportunity to thank Google, NoemaLife, Microsoft Research, the Department for Information and Communication Technologies of the Italian National Council of Research, the Italian Association for Automatic Computation (AICA), Yahoo! Research, Twitter, and the ASSETS project. All of them provided generous sponsorship, which allowed the organizers to keep the registration fees as low as possible and thus to enhance participation.

We would also like to thank everybody involved in making SPIRE 2011 such an exciting event. Specifically, we would like to thank all conference, tutorial, and workshop participants and presenters, who provided a fascinating one-week program of high-quality presentations and intensive discussions. Thanks also to all the members of the Program Committee and to the additional reviewers, who went to great lengths to ensure the high quality of this conference, and to the coordinator of the SPIRE Steering Committee, Ricardo Baeza-Yates, who provided assistance and guidance in the organization.

Furthermore, we would like to thank all the members of the local organizing team at the Italian National Council of Research and at the University of Pisa. Particularly, we would like to thank Andrea Esuli who acted as Tutorials Chair, Nadia Pisanti who acted as Workshops Chair, our Webmaster Stefano Baccianella, Catherine Bosio and Giulio Galesi who gave us support in local arrangements, Beatrice Rapisarda who designed the official poster of the symposium, and all the student volunteers. They all made a tremendous effort to make sure that this event was exciting and enjoyable. It is due to them that the organization of SPIRE 2011 was not just hard work, but also a pleasure.

October 2011

Roberto Grossi
Fabrizio Sebastiani
Fabrizio Silvestri

Organization

Program Committee

Omar Alonso	Microsoft
Gianni Amati	Fondazione Ugo Bordoni
Amihood Amir	Bar-Ilan University and Johns Hopkins University
Leif Azzopardi	University of Glasgow
Rolf Backofen	Albert-Ludwigs-University Freiburg
Ricardo Baeza-Yates	Yahoo! Research
Alvaro Barreiro	University of A Coruña
Philip Bille	Technical University of Denmark
Paolo Boldi	Università degli Studi di Milano
Danny Breslauer	University of Haifa
Edgar Chavez	Universidad Michoacana
Charles Clarke	University of Waterloo
Maxime Crochemore	King's College London and Université Paris
Brian Davison	Lehigh University
Nadia El-Mabrouk	University of Montreal
Paolo Ferragina	University of Pisa
Frantisek Franek	McMaster University
Leszek Gasieniec	University of Liverpool
Dora Giammarresi	University of Rome "Tor Vergata"
Nazli Goharian	Georgetown University
Gregory Grefenstette	Exalead
Roberto Grossi	Università di Pisa
Concettina Guerra	University of Padova and Georgia Tech
Antonio Gulli	Microsoft
Jan Holub	Czech Technical University in Prague
Heikki Hyyrö	University of Tampere
Lucian Ilie	University of Western Ontario
Costas Iliopoulos	King's College London
Shunsuke Inenaga	Kyushu University
Shen Jialie	Singapore Management University
Jaap Kamps	University of Amsterdam
Takuya Kida	Hokkaido University
Marcin Kubica	Warsaw University
Gregory Kucherov	CNRS/LIGM
Mounia Lalmas	University of Glasgow
Moshe Lewenstein	Bar Ilan University
Alistair Moffat	University of Melbourne
Laurent Mouchard	University of Rouen

Gonzalo Navarro	University of Chile
Wolfgang Nejdl	L3S and University of Hannover
Iadh Ounis	University of Glasgow
Laxmi Parida	IBM Research
Kunsoo Park	Seoul National University
Marco Pellegrini	Institute for Informatics and Telematics of C.N.R.
Pierre Peterlongo	INRIA Rennes-Bretagne-Atlantique
Andrea Pietracaprina	University of Padova
Ely Porat	Bar-Ilan University
Venkatesh Raman	The Institute of Mathematical Sciences
Horacio Rodriguez	Universitat Politècnica de Catalunya
Marie-France Sagot	Université de Lyon
Cenk Sahinalp	Simon Fraser University
Leena Salmela	University of Helsinki
Jeanette Schmidt	Stanford University
Fabrizio Sebastiani	ISTI - CNR
Fabrizio Silvestri	ISTI - CNR
Steven Skiena	Stony Brook University
Dina Sokol	Brooklyn College of the City University of New York
Jens Stoye	Bielefeld University
Torsten Suel	Yahoo! Research
Fabio Vandin	Brown University
Stéphane Vialette	Université Paris-Est
Alain Viari	INRIA
Jeff Vitter	University of Kansas
Oren Weimann	Weizmann Institute of Science
Le Zhao	CMU
Nivio Ziviani	Federal University of Minas Gerais

Additional Reviewers

Andonov, Rumen
Antoniou, Pavlos
Arroyuelo, Diego
Badkobeh, Golnaz
Bernhardt, Daniel
Blanco, Roi
Bressan, Marco
Canovas, Rodrigo
Ceccarelli, Diego
Chikhi, Rayan
Claude, Francisco
Constant, Matthieu
Costa, Fabrizio
Dai, Na
Dan, Ovidiu
David, Julien
Epifanio, Chiara
Fernandes, David
Fertin, Guillaume
Fonseca, Paulo
Franek, Frantisek
Frigeri, Achille
Galle, Matthias
Gerlach, Wolfgang
Giraud, Mathieu
Gurevich, Maxim
Hamel, Sylvie
Hegerty, Ian

Husemann, Peter
Jahn, Katharina
Jaroš, Jakub
Jiang, Minghui
Karenos, Kyriakos
Kopelowitz, Tsvi
Levy, Avivit
Lonati, Violetta
Losada, David
Manzini, Giovanni
Markowetz, Alexander
Martinez-Prieto, Miguel A.
Menezes, Guilherme
Möhl, Mathias
Nanni, Mirco
Nardini, Franco Maria
Noe, Laurent
Peterlongo, Pierre
Pinkas, Benny
Pissis, Solon
Prochazka, Petr
Puglisi, Simon
Qi, Xiaoguang
Radoszewski, Jakub
Rojas, Pablo
Rosone, Giovanna
Russo, Luis M.S.
Santos, Rodrygo
Satti, Srinivasa Rao
Schmidt, Jeanette
Silva, Altigran
Silvestri, Francesco
Starikovskaya, Tatiana
Tannier, Eric
Tischler, German
Tolomei, Gabriele
Utro, Filippo
Veloso, Adriano
Venturini, Rossano
Vigna, Sebastiano
Vildhøj, Hjalte Wedel
Walen, Tomasz
Will, Sebastian
Wittler, Roland
Xu, Bojian
Xue, Zhenzhen
Yan, Hao
Yin, Dawei
Yorukoglu, Deniz
Zelikovitz, Sarah

Table of Contents

Constructing Strings at the Nano Scale via Staged Self-assembly 1
Erik D. Demaine

Discounted Cumulative Gain and User Decision Models 2
Georges Dupret

Cross-Lingual Text Fragment Alignment Using Divergence from Randomness 14
Sirvan Yahyaei, Marco Bonzanini, and Thomas Roelleke

Enhancing Document Snippets Using Temporal Information 26
Omar Alonso, Michael Gertz, and Ricardo Baeza-Yates

Spaced Seeds Design Using Perfect Rulers 32
Lavinia Egidi and Giovanni Manzini

Weighted Shortest Common Supersequence 44
Amihood Amir, Zvi Gotthilf, and B. Riva Shalom

Approximate Regular Expression Matching with Multi-strings 55
Djamal Belazzougui and Mathieu Raffinot

Persistency in Suffix Trees with Applications to String Interval Problems 67
Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat

Approximate Point Set Pattern Matching with L_p-Norm 81
Hung-Lung Wang and Kuan-Yu Chen

Detecting Health Events on the Social Web to Enable Epidemic Intelligence 87
Marco Fisichella, Avaré Stewart, Alfredo Cuzzocrea, and Kerstin Denecke

A Learned Approach for Ranking News in Real-Time Using the Blogosphere 104
Richard McCreadie, Craig Macdonald, and Iadh Ounis

Attribute Retrieval from Relational Web Tables 117
Arlind Kopliku, Karen Pinel-Sauvagnat, and Mohand Boughanem

Query-Sets^{++}: A Scalable Approach for Modeling Web Sites 129
Barbara Poblete, Myra Spiliopoulou, and Marcelo Mendoza

Indexing with Gaps 135
Moshe Lewenstein

Fast Computation of a String Duplication History under No-Breakpoint-Reuse (Extended Abstract) 144
Broňa Brejová, Gad M. Landau, and Tomáš Vinař

Near Real-Time Suffix Tree Construction via the Fringe Marked Ancestor Problem 156
Dany Breslauer and Giuseppe F. Italiano

Approximations and Partial Solutions for the Consensus Sequence Problem 168
Amihood Amir, Haim Paryenty, and Liam Roditty

Fixed Block Compression Boosting in FM-Indexes 174
Juha Kärkkäinen and Simon J. Puglisi

Space Efficient Wavelet Tree Construction 185
Francisco Claude, Patrick K. Nicholson, and Diego Seco

Computing the Longest Common Prefix Array Based on the Burrows-Wheeler Transform 197
Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger

A Succinct Index for Hypertext 209
Chris Thachuk

When Was It Written? Automatically Determining Publication Dates 221
Anne Garcia-Fernandez, Anne-Laure Ligozat, Marco Dinarelli, and Delphine Bernhard

A New Approach for Verifying URL Uniqueness in Web Crawlers 237
Wallace Favoreto Henrique, Nivio Ziviani, Marco Antônio Cristo, Edleno Silva de Moura, Altigran Soares da Silva, and Cristiano Carvalho

External Query Reformulation for Text-Based Image Retrieval 249
Jinming Min and Gareth J.F. Jones

A Knowledge-Based Semantic Kernel for Text Classification 261
Jamal Abdul Nasir, Asim Karim, George Tsatsaronis, and Iraklis Varlamis

Compressed Text Indexing with Wildcards 267
Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter

Fast q-gram Mining on SLP Compressed Strings 278
Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Succinct Gapped Suffix Arrays 290
Luís M.S. Russo and German Tischler

Finding Frequent Elements in Compressed 2D Arrays and Strings 295
Travis Gagie, Meng He, J. Ian Munro, and Patrick K. Nicholson

On Suffix Extensions in Suffix Trees 301
Dany Breslauer and Giuseppe F. Italiano

COCA Filters: Co-occurrence Aware Bloom Filters 313
Kamran Tirdad, Pedram Ghodsnia, J. Ian Munro, and Alejandro López-Ortiz

On-line Construction of Position Heaps 326
Gregory Kucherov

Computing All Subtree Repeats in Ordered Ranked Trees 338
Michalis Christou, Maxime Crochemore, Tomáš Flouri, Costas S. Iliopoulos, Jan Janoušek, Bořivoj Melichar, and Solon P. Pissis

Sparse Spatial Selection for Novelty-Based Search Result Diversification 344
Veronica Gil-Costa, Rodrygo L.T. Santos, Craig Macdonald, and Iadh Ounis

Candidate Document Retrieval for Web-Scale Text Reuse Detection 356
Matthias Hagen and Benno Stein

A Multi-faceted Approach to Query Intent Classification 368
Cristina González-Caro and Ricardo Baeza-Yates

Navigating the User Query Space 380
Ronan Cummins, Mounia Lalmas, Colm O'Riordan, and Joemon M. Jose

Improved Compressed Indexes for Full-Text Document Retrieval 386
Djamal Belazzougui and Gonzalo Navarro

ESP-Index: A Compressed Index Based on Edit-Sensitive Parsing 398
Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto

Compressed Indexes for Aligned Pattern Matching 410
Sharma V. Thankachan

Reference Sequence Construction for Relative Compression of Genomes 420
Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel

Author Index 427

Constructing Strings at the Nano Scale via Staged Self-assembly

Erik D. Demaine

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
32 Vassar St., Cambridge, MA 02139
edemaine@mit.edu

Abstract. Tile self-assembly is an intriguing approach to manufacturing desired shapes with nano-scale feature size. A recent direction in this theory allows the use of multiple stages—operations performed by the experimenter, such as mixing two self-assembling systems together. This flexibility transforms the experimenter from a passive entity into a parallel algorithm, and vastly reduces the number of distinct parts required to construct a desired shape, possibly making the systems practical to build.

We start with the relatively simple goal of constructing 1D strings of tiles with distinctive markers, while minimizing the number of mixing steps (work) performed by the staged assembly. In the practical situation of few different "glues", this problem turns out to be closely related to compressing a string into a context-free grammar with the fewest nonterminals. In general, however, the problems turn out to be quite different, as the implicit parallelism of a mixing operation can be exploited to reduce the number of steps.

The staged-assembly perspective also enables the possibility of additional operations, such as adding an enzyme that destroys all tiles with a special label. By enabling destruction in addition to the usual construction, we can perform tasks impossible in a traditional self-assembly system. For example, we can build a Replicator, which transforms a given object of unknown shape or size into many copies of that shape; and we find a vastly more efficient way to construct a nano computer through self-assembly.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, p. 1, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Discounted Cumulative Gain and User Decision Models

Georges Dupret

Yahoo!

Abstract. We propose to explain Discounted Cumulative Gain (DCG) as the consequences of a set of hypothesis, in a generative probabilistic model, on how users browse the result page ranking list of a search engine. This exercise of reconstructing a user model from a metric allows us to show that it is possible to estimate from data the numerical values of the discounting factors. It also allows us to compare different candidate user models in terms of their ability to describe the observed data, and hence to select the best one. It is generally not possible to relate the performance of a ranking function in terms of DCG with the clicks observed after the function is deployed on a production environment. We show in this paper that a user model make this possible. Finally, we show that DCG can be interpreted as a measure of the utility a user gains per unit of effort she is ready to allocate. This contrasts nicely with a recent interpretation given to average precision (AP), another popular Information Retrieval metric, as a measure of effort needed to achieve a unit of utility [7].

Introduction

An accurate method to quantify the quality of a document ranking is a fundamental requisite in the design of a search engine. Evaluation involves typically several distinct metrics; Some of them are *prognostic* metrics that can be computed before a new ranking function is deployed while others analyze how users interacted with the ranking and can be therefore considered as *diagnostic* metric. The main difference between these two types of metric is that the first one requires a prediction of the user actions, while the second simply use available observed actions or clicks.

The main argument of this paper is that to derive a reliable metric, we need to define how users interact with a ranking list [6]. Robertson [14] expressed this idea before us: "If we can interpret a measure (...) in terms of an explicit user model (...), this can only improve our understanding of what exactly the measure is measuring". To illustrate this, consider the traditional 5 labels of DCG that characterize the relevance of a document to a query: PERFECT, EXCELLENT, GOOD, FAIR and BAD (P, E, G, F and B in short). Say a first ranking function –*F1*– produces a sequence of document with relevances $BBPBB$, while another function *F2* produces $FFFBB$. Provided users scan the list sequentially, if users stop their search after the second position in the ranking, then *F2* is better. On the other hand, if most of them scan at least three positions, then ranking *F1*

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 2–13, 2011.
© Springer-Verlag Berlin Heidelberg 2011

might be preferred. In conclusion, the user behavior defines which ranking is best. Moffat & Zobel [13] also derive the Rank Biased Precision (*rbf*) metric based on a user model.

Resorting to user modeling is also a first step to break the "chicken and egg" problem we face when comparing two different metrics: Deciding which metric is best calls for a third "meta" metric to compare the original metrics [6]. Because various "meta" metrics are likely to co-exist, a meta metric for the meta metrics is also necessary, etc. User models on the other hand can be compared based on their predictive ability. If one model predicts more accurately future user interactions with a search engine than another, then the metric derived from the best user model is arguably better. This doesn't completely solve the problem though, as different metrics can be derived from a common user model.

This works concentrates on identifying possible user models for maybe the two most widely used metrics in web search: DCG. This exercise will help us identify the implicit assumptions behind this metric. We have been inspired by a similar study [7] on average precision (AP) that we leverage here to provide insight on the fundamental differences between AP and DCG.

1 Discounted Cumulative Gain

DCG [11] in its more general form can be written as

$$\mathtt{DCG}_R = \sum_1^R D_r \mathtt{G}_r \quad (1)$$

where D_r is a discounting factor decreasing with the rank r and $\mathtt{G}_r$ is the gain achieved by presenting document d_r at rank r. There are generally two related interpretation of this metric:

Utilitarian: The utility of a document to a user decreases when the document is low in the ranking.

Probabilistic: All documents are not examined with the same probability. Search Engine logs show that the probability of clicking a document decreases with its rank and it is natural to discount a document usefulness accordingly.

The original paper [11] introducing DCG doesn't relate it explicitly to user behavior or to a decision process, and is therefore closer to interpretation (a). Departing from this, we propose here two user models that lead to metrics interpretable as variant of DCG, each leading to a distinct set of discounting factor estimates. These models are generative models and can therefore be used to predict which documents a user clicks when presented with a list of search results. By comparing the predicted clicks with the actual clicks observed on a held out set of sessions, we can identify the best model, and hence the best set of discounting factors.

We first introduce some notations. Because we suppose that all documents are judged, we can understand a ranking as a sequence of labels $\ell_r, r = 1, \ldots, R$

where r indexes the position in the ranking. We often use the notation $\ell_{1:R}$ to represent the whole ranking up to position R. A user looking at a list search result will only click on a result if he looks at this specific result (They are no "accidental" clicks). We call the latter process the *examination* and define a binary variable E_r depending on the rank r, that indicates whether a particular rank r is examined by the user. The subscript r is dropped when there is no ambiguïty. Finally, the binary variable C_r indicates whether a document was clicked or not.

As discussed, we suppose that if a document is clicked, then its position is previously examined. We also use the following shorthand: e^+ and e^- are equivalent to "E is *true*" and "E is *false*", respectively. We also use $E = 1$ and $E = 0$ to denote e^+ and e^- when convenient. The same holds for c^+ and c^- or other binary variables introduced later.

2 Deterministic Click User Model

The first user model is also the simplest:

User Model 1 (Deterministic Click)

1. *The user chooses to examine a rank r between 1 and R with a probability $P(e_r^+)$.*
2. *She always clicks on the link to the document at rank r. The document at rank r has a utility $\mathtt{U}(\ell_r)$ for her, where ℓ_r is the document label.*

This model is unable to predict sessions where no clicks occur. Multiple clicks in a given sessions are understood as the same user repeating the above process[1]. In other words, a multiple clicks session is really a sequence of one-click sessions.

In the case of a one click session, the average utility achieved is:

$$\mathbb{E}(\mathtt{U}) = \sum_{r=1}^{R} P(e_r^+)\mathtt{U}(\ell_r)$$

It is then possible to interpret this expected utility as DCG, provided $P(e_r^+)$ is understood as the discounting factor. This shows that if we assume that (a) the Deterministic Click model holds, (b) the utility of a session is a valid measure of user satisfaction, then the average satisfaction coïncide with the DCG. The converse is not true as we will illustrate later by introducing another model that can also imply DCG as a metric.

If search logs are available, then an estimate of $P(E_r)$ can be obtained by maximizing the likelihood. The likelihood of a single session with a click at position r is:

$$\mathtt{L}_r = P(e_r^+) \times \prod_{s=1;s\neq r}^{R} (1 - P(e_s^+))$$

[1] Nothing prevents her from clicking twice on the same document.

If there are more than one click – say a click at positions r_1 and one at r_2, then the likelihood is the product $\mathtt{L}_{r_1} \times \mathtt{L}_{r_2}$ because a session with multiple clicks is equivalent to several sessions with one click. This holds even if $r_2 = r_1$.

3 Probabilistic Click User Model

The Deterministic Click model is fairly unrealistic because, among other reasons, it assumes that the user chooses one position in the ranking and ignores the other. Once she chooses a position, it is assumed

(a) She clicks blindly on the corresponding document. Real users don't click deterministically on a document they examine. Instead, they evaluate the document snippet before deciding whether to click.
(b) She completely ignores the snippet higher and lower in the ranking. Here again, real users behave differently. Eye tracking experiments suggest they tend to browse the result list sequentially [9].

These observations suggest the following model:

User Model 2 (Probabilistic User Model)

1. *The user examines sequentially the ranking up to a position r she chooses before starting the search. After reaching this position she abandons the search.*
2. *She clicks on a examined document with a probability $P(c^+|e^+;\ell)$ where ℓ is the editorial label of the document*[2].

Like in [10], we make no assumption on whether the user is satisfied or not when she abandons the search and once examined, the probability of click on a document is independent of the position. On the other hand, the probability of examination depends on the position.

3.1 Expected Utility

In order to evaluate the expected utility associated with a ranking $\ell_{1:R}$, we first define for convenience the multinomial variable A on $\{1, \ldots, R+1\}$ that describes up to which position the user is willing to examine the ranking: $A = r$ means that she decides to end the search at position r. The value $R+1$ has a special status and means that the user didn't abandon her search up to rank R, included. The event $A = r$ is sometimes written a_r for short. A is univoquely related to the set $E_{1:R}$: $A = r$ entails $\{e^+_{1:r}, e^-_{r+1:R}\}$, i.e. all snippets up to position r are examined and none is examined after r.

[2] This requires that the search engine generates snippets that represent fairly the document content. In other word, we assume that the perceived and intrinsic or real relevance are aligned. If we had editorial labels for the document snippets as well as the documents themselves we could estimate the probability of a click given the snippet label rather than the document label.

The joint distribution describing the model is:

$$P(A, C_{1:R}, E_{1:R}; \ell_{1:R}) = P(A) \prod_{r=1}^{R} P(C_r|E_r; \ell_r) P(E_r|A)$$

where $E_{1:R}$ and A depend on each other deterministically. The expected utility can be expressed as

$$\mathbb{E}(\mathtt{U}) = \mathbb{E}(\sum_{r=1}^{R} \mathtt{U}(\ell_r) C_r) = \sum_{r=1}^{R} \mathtt{U}(\ell_r) P(c_r^+)$$

where $C_r = 1$ if the document is clicked and 0 otherwise. We therefore need to compute the expectation of a click at an arbitrary position r. We can always write

$$\begin{aligned} P(c_r^+) &= P(c_r^+, A < r) + P(c_r^+, A \geq r) \\ &= 0 + P(c_r^+|e_r^+; \ell_r) P(A \geq r) \end{aligned}$$

therefore

$$\mathbb{E}(\mathtt{U}) = \sum_{r=1}^{R} \mathtt{U}(\ell_r) P(c_r^+|e_r^+; \ell_r) P(A \geq r)$$

The terms $\mathtt{U}(\ell_r) P(c_r^+|e_r^+; \ell_r)$ only depends on the document label, not the rank[3], and can therefore be identified with the DCG gain $\mathtt{G}_r$ in Eq. 1. The term $P(A \geq r)$ only depends on the position in the ranking and is associated with the discounting factor[4] :

$$\begin{cases} \mathtt{G}_r = \mathtt{U}(\ell_r) P(c_r^+|e_r^+; \ell_r) \\ D_r \propto P(A \geq r) \end{cases} \tag{2}$$

It is interesting to observe that the DCG gains $\mathtt{G}_r$ incorporate implicitly a click probability. Another way to state this is to recognize that the utility gained by a user who clicks on a document with label ℓ is now $\mathtt{G}_\ell / P(c^+|e^+; \ell)$ instead of $\mathtt{G}_\ell$ as predicted by the first model.

3.2 Parameters Estimation

Because we have defined a user model, we are able to predict user sessions and hence estimate the model parameters by maximum likelihood. The likelihood is obtained by marginalizing out the (partially) hidden variables A and $E_{1:R}$.

[3] The presence of r in the expression $\mathtt{U}_r P(c_r^+|e_r^+; \ell_r)$ doesn't imply a dependence on r of the gains; It's role is to identify the document label. For example, $\mathtt{U}_r P(c_r^+|e_r^+; \ell_r) = \mathtt{U}_s P(c_s^+|e_s^+; \ell_s)$ as long as the documents at positions r and s share the same label.

[4] It is always possible to derive discounting factors $D_{1:R}$ from the probabilities $P(A \geq r)$, $r = 1, \ldots, R$, but the opposite is not true because D_r is not required to be a probability in the original definition [11].

Suppose we observe a session $\mathcal{S}_t$ for which the last click is at position b. The sequence of clicks in this session can be described by $\{C_{1:b-1}, c_b^+, c_{b+1:R}^-\}$. The probability of $\mathcal{S}_t$ is:

$$P(\mathcal{S}_t) = \sum_A P(A, C_{1:b-1}, c_b^+, c_{b+1:R}^-, E_{1:R}; \ell_{1:R})$$

There is no need to sum over the states of $E_{1:R}$ because they are univoquely defined by the state of A.

In order to keep notations lighter, we will keep $E_{1:R}$ and $\ell_{1:R}$ implicit wherever possible. We also sometimes omit the reference to the label in $P(C_r|E_r; \ell_r)$ for convenience. We will estimate $P(A, C_{1:R})$ for $A = 1, \ldots R$ and then add the results to obtain $P(\mathcal{S}_t)$.

First, we observe that $P(A < b, C_{1:b-1}, c_b^+, c_{b+1:R}^-) = 0$ because by hypothesis users don't click on documents they don't examine. If $A = b$, that is if the user decided he would examine the ranking up to position b where he happened to also click the link, we have:

$$P(A = b, C_{1:b-1}, c_b^+, c_{b+1:R}^-, e_{1:b}^+, e_{b+1:R}^-) = P(a_b)P(c_b^+|e_b^+; \ell_b) \prod_{r=1}^{b-1} P(C_r|e_r^+; \ell_r)$$

If $A = b + 1$ – the case where the user decides to examine up to position $b + 1$, but the link at that position didn't raise her interest– we have:

$$\begin{aligned}
&P(A = b + 1, C_{1:b-1}, c_b^+, c_{b+1:R}^-, e_{1:b+1}^+, e_{b+2:R}^-) \\
&= P(a_{b+1})P(c_{b+1}^-|e_{b+1}^+)P(c_b^+|e_b^+) \prod_{r=1}^{b-1} P(C_r|e_r^+)
\end{aligned}$$

Generalizing up to R and adding, we obtain the likelihood of session $\mathcal{S}_t$:

$$\begin{aligned}
\mathtt{L}(\mathcal{S}_t) &= P(C_{1:b-1}, c_b^+, c_{b+1:R}^-; \ell_{1:R}) \\
&= P(c_b^+|e_b^+; \ell_b) \prod_{r=1}^{b-1} P(C_r|e_r^+; \ell_r) \\
&\quad \times \big[P(a_b) + P(a_{b+1})P(c_{b+1}^-|e_{b+1}^+; \ell_{b+1}) \\
&\quad +P(a_{b+2})P(c_{b+1}^-|e_{b+1}^+; \ell_{b+1})P(c_{b+2}^-|e_{b+2}^+; \ell_{b+2}) \ldots\big]
\end{aligned}$$

It is convenient to express the probability of abandoning the search in terms of the examination variables to enforce the search to be sequential:

$$\begin{aligned}
&P(a_r) = P(e_1^+, \ldots, e_{r-1}^+, e_r^+, e_{r+1}^-, \ldots e_R^-) \\
&= P(e_R^-|e_{R-1}^-)P(e_{R-1}^-|e_{R-2}^-) \ldots \\
&\quad \times P(e_{r+1}^-|e_r^+)P(e_r^+|e_{r-1}^+) \ldots P(e_2^+|e_1^+)P(e_1^+) \\
&= P(e_{r+1}^-|e_r^+)P(e_1^+) \prod_{s=1}^{r-1} P(e_{s+1}^+|e_s^+)
\end{aligned}$$

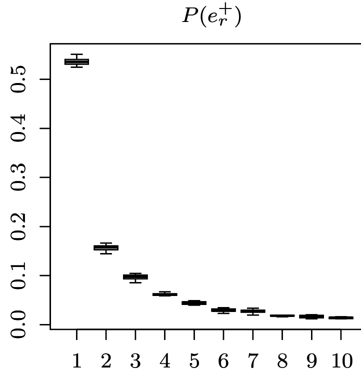


Fig. 1. Probabilities $P(e^+)$ for the Deterministic Click Model

It is a simple matter to multiply the likelihood of a set of observed sessions and maximize it with respect to $P(c_r^+|e_r^+;\ell_r)$ and $P(A = r)$ or $P(e_{r+1}^-|e_r^+)$, $r = 1 : R$ to obtain estimates of these probabilities. We used the Expectation Maximization algorithm [5] for this task.

4 Numerical Experiments

We collected from the logs of a commercial search engine a set of approximately 30.000 sessions with at least one click for which we have an editorial judgment on a 5 level scales for each of the top 10 urls, together with a record of which urls have been clicked. Each record in our data set has the following form: A sequence of 10 labels $\ell_{1:10}$ followed by a sequence of 10 *True* or *False* tokens that indicates the states of $C_{1:10}$.

We divided the data in 10 random subsets and use each of these subsets (i.e. 10% of the original set) as the data we maximize the likelihood on. This results in 10 different sets of estimates for the parameters of User Models 1 and 2 that we report in Tables 1 & 2 and in Figs. 1 & 2. The boxplots attest that the estimates turn out to be fairly stable. In table 2 we also report for comparison a set of discounting factors often chosen by default.

Table 1. Probabilistic Click Model: Median probability of click given a label, `DCG` gains and utilities

label	B	F	G	E	P
$P(c^+\|e^+;\ell)$	0.27	0.27	0.34	0.37	0.85
$\mathtt{G}(\ell)$	0.0	0.5	3.0	7.0	10.0
$\mathtt{U}(\ell)$	0.00	1.85	8.82	18.92	11.76

The boxplot in Fig. 1 reports the probability of examination of each rank for the Deterministic Click model. Results agree with intuition.

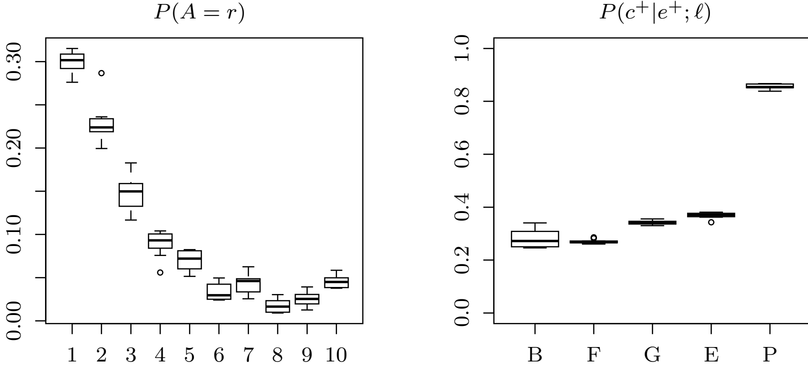


Fig. 2. Probability $P(A = r)$ of Abandoning the search at rank r and Probability $P(c^+|e^+, \ell)$ of click given a label for the probabilistic DCG User Model

In Fig. 2, the left boxplot reports the probability of abandoning the search at rank r for the probabilistic user model. Because abandoning at rank r entails in this model that the user examined all ranks up to r, these probabilities cannot be compared easily with the probabilities derived from the deterministic user model. Note that more users decide to end their search at ranks 9 or 10 than 8. Intuitively there is no contradiction because a user who examines up to rank 8 might as well go until the end of the page. The boxplot on the right reports the probability of click for examined links for the 5 different labels. Although nothing in the model enforces it, we see that the model predicts that document with a better label also have a higher probability of being clicked. BAD and FAIR documents have very similar probability of being clicked, as do GOOD and EXCELLENT. PERFECT stands out as a category of document with a particularly high probability of being clicked. This makes sense as they are defined as the target page for navigational query.

We have proposed two distinct models to explain DCG, the Deterministic and the Probabilistic Click User Models. The second model seems more realistic, but we would like to be able to confirm quantitatively this intuition. Both models are generative models and can be used to predict user behavior; We can therefore compare the accuracy of these predictions on the test sets. We use the *perplexity* –a common measure of the "surprise" of a model when presented with a new observation. Given a proposed probability model q of the true distribution p, one may evaluate q by asking how well it predicts a separate test sample of size N also drawn from p. The perplexity of the model q is defined as

$$2^{-\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)}$$

Better models q of the unknown distribution p will tend to assign higher probabilities to the test events. Thus, they have lower perplexity: they are less surprised by the test sample.

In the context of the user behaviors, the perplexity is a monotonically increasing function of the joint probability of the sessions in the test set. Analytically, this probability is identical to the likelihood of the test set, but instead of maximizing it with respect to the parameters, those are held fixed at the values that maximize the likelihood on the *training* set.

All the sessions in both the training and test sets have exactly 10 results per page ($R = 10$) so that by setting N to 10 times the number of sessions, the perplexity can be loosely[5] interpreted as the number of trials per correct prediction of a binary event: the click or skip of a document. The lower the perplexity, the better the model: A perplexity of 1 corresponds to perfect prediction, while a perplexity of 2 corresponds to randomly predicting the two possible outcomes with 50% chances. Perplexity larger than two characterize models that are so bad that simply inverting the binary predictions would lead to a more accurate model.

The mean perplexity of the Deterministic Click Model evaluated on the 10 random splits of the data is 1.29, while the mean perplexity of the Probabilistic Click Model is 1.27. A Welch Two Sample t-test lead to a p-value smaller than 2.2e-16. The Probabilistic Click Model is therefore statistically significantly better at predicting the user behavior.

Table 2. Left column: Mean probabilities of examination $P(e_r^+)$ for the Deterministic Click model. Right columns: The probability of abandoning beyond rank r for the Probabilistic Click Model and, for comparison, the popular $D_r = 1/\log_2(1+r)$ discount factor for $r = 1, \ldots, 10$. We observe that the empirical discounting factors decrease faster beyond rank 3 than what the logarithmic decay accounts for.

rank r	$P(e_r^+)$	$P(A \geq r)$	$1/\log_2(1+r)$
1	0.53	1.00	1.00
2	0.16	0.70	0.63
3	0.10	0.47	0.50
4	0.06	0.32	0.43
5	0.04	0.23	0.39
6	0.03	0.17	0.36
7	0.03	0.13	0.33
8	0.02	0.09	0.32
9	0.02	0.07	0.30
10	0.01	0.05	0.29

We can qualify the `DCG` as a *prognostic* metric because it is typically computed to evaluate a ranking function *before* it is presented to users. Its aim is to predict whether the new function is more likely to satisfy users. Once sufficient data is collected on the interactions of users with the new ranking function, a *diagnostic* metric can be evaluated: Instead of computing the expected utility

[5] This interpretation is not strictly correct because the clicks and skips in a session are not independent. The evaluation itself continues however to be valid.

of the ranking, we compute its empirical utility $\hat{\mathtt{U}}$ as the average of the sum of the utilities of the clicked documents. For example, the *Diagnostic* DCG of a session $\{c_1^-, c_2^+, c_3^-, c_4^+, c_{5:10}^-\}$ is $\hat{\mathtt{U}} = \mathtt{U}(\ell_2) + \mathtt{U}(\ell_4)$. Note that it is not possible to associate a diagnostic metric to a prognostic metric like DCG unless a user model is defined.

For the Deterministic Click model the utility is equal to the gain, while for the Probabilistic Click Model, the utility is related to the gain by Eq. 2. We plot in Fig. 3 the prognostic DCG vs. the diagnostic DCG for the Deterministic Click (left) and Probabilistic (right) models. We have divided the prognostic DCG in 10 bins of equal range to ease visualization [6]. We also computed the Pearson correlation between the DCG and the diagnostic values of both models: The values are 15% for the Deterministic Click Model and 34% for the Probabilistic Model. This argues in favor of distinguishing the gain from the utility and once again argues in favor of the Probabilistic Click Model.

Although none of the models provides a method to evaluate the gains, we can use the probability of clicks given an editorial label to estimate the utilities. Returning to Eq. 2, we see that we have the relation

$$\mathtt{U}(\ell_r) = P(c_r^+ | e_r^+; \ell_r)/\mathtt{G}_r$$

between the DCG gains and the utilities defined in the Probabilistic Click Model. If we plug-in the gain values commonly used into this formula, we obtain the results reported in Fig. 1. This seems to indicate that the gains associated to EXCELLENT and PERFECT values are inadequate because they don't respect the order of the labels on the utility scale. Note also that in order to evaluate the diagnostic DCG we must use the utility, not the gain. The sum of the gains of the clicked documents doesn't lead to the correct estimate unless utility and gains are equal like in the Deterministic Click model.

Different values for the gains $\mathtt{G}_r$ and discounting factors D_r are found in the Literature or proposed by the original authors [11]. Moffat and Zobel [13] propose the rbp metric as an improvement over Mean Average Precision. As it turns out rbp is similar to DCG with numerical values for the discount factor based on $P(e_{r+1}^+ | e_r^+) = p$ where p is adjusted to the click data.

5 Discussion

We have seen how the DCG metric can be derived from a set of simple assumptions on how the user interacts with a ranking list. Making these assumptions explicit enables us to derive a method to evaluate the discounting factors from past user interactions. It also suggests that for every prognostic metric based on a user model, there exists a "diagnostic" metric that is nothing but its empirical counter-part.

The popular PERFECT, EXCELLENT, GOOD, FAIR and BAD editorial labels used to evaluate web search ranking is often associated to the numerical

[6] We had to withdraw the numerical values out of confidentiality concerns.

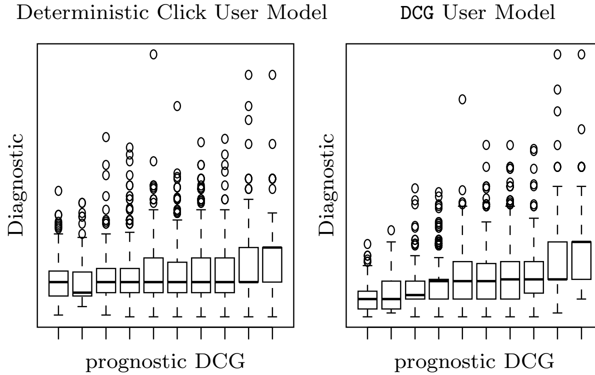


Fig. 3. Diagnostic vs. Prognostic DCG for the Deterministic Click and the DCG User Models

values 10, 7, 3, 0.5 and 0. We have shown for the data we use in this work that these values are not appropriate. This conclusion is likely to be true for web search in general.

The Probabilistic Click Model enables us to determine the discounting factors, but it doesn't help to determine the utilities $\mathtt{U}(\ell_r)$ or the gains $\mathtt{G}_r$. This is a consequence of a user model for which the relevance of the clicked documents have no influence on the user decision to stop the search. This is itself a consequence of the DCG definition: The discounting factors D_r are the same no matter how relevant the documents at the other positions and therefore the contribution of the document at rank r to the final DCG value is independent of the other documents in the ranking.

Although we believe the Probabilistic Click Model we propose here is the best explanation of this metric for web search, it is always possible that other, more accurate user models exist. If this is the case, it should be easy to compare the models in term of their predictive ability and adjust the gains and discounting factors accordingly. Note also that adequacy of a user model also depends on the search engine, because users might behave differently on different search engines. It is therefore possible that the Deterministic Click Model – or some other model – is more accurate for certain engines than the Probabilistic Click Model.

The user models for DCG we presented in this work have in common the fact that the user decide before hand how many links in the ranking she will examine. If we take this number of ranks as a proxy for the effort the user is willing to devote in order to reach the desired information, we can say that DCG is an *effort* based metric that estimates the amount of utility a user gains per unit of effort. This is to be contrasted with Average Precision AP: in [7], the authors show that AP can be understood as a measure related to the number of positions a user needs to examine in order to recover a pre-defined number of relevant documents. In this respect, AP can be understood as a measure of effort per unit of utility.

References

1. Bollmann, P., Raghavan, V.V.: A utility-theoretic analysis of expected search length. In: SIGIR 1988, pp. 245–256. ACM, New York (1988)
2. Buckley, C., Voorhees, E.M.: Retrieval evaluation with incomplete information. In: SIGIR 2004, pp. 25–32. ACM, New York (2004)
3. Carterette, B., Jones, R.: Evaluating search engines by modeling the relationship between relevance and clicks. Advances in Neural Information Processing Systems 20, 217–224 (2008)
4. Craswell, N., Zoeter, O., Taylor, M., Ramsey, B.: An experimental comparison of click position-bias models. In: First ACM International Conference on Web Search and Data Mining, WSDM 2008 (2008)
5. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. J. R. Statist. Soc. B 39, 1–38 (1977)
6. Dupret, G.: User models to compare and evaluate web IR metrics. In: Proceedings of SIGIR 2009 Workshop on The Future of IR Evaluation (2009), http://staff.science.uva.nl/~{k}amps/ireval/papers/georges.pdf
7. Dupret, G., Piwowarski, B.: A User Behavior Model for Average Precision and its Generalization to Graded Judgments. In: Proceedings of the 33th ACM SIGIR Conference (2010)
8. Führ, N.: A probability ranking principle for interactive information retrieval. In: Information Retrieval. Springer, Heidelberg (2008)
9. Granka, L., Joachims, T., Gay, G.: Eye-tracking analysis of user behavior in www search. In: ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), pp. 478–479 (2004)
10. Guo, F., Liu, C., Wang, Y.M.: Efficient multiple-click models in web search. In: WSDM 2009: Proceedings of the Second ACM International Conference on Web Search and Data Mining, pp. 124–131. ACM, New York (2009)
11. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of ir techniques. ACM Transactions on Information Systems (ACM TOIS) 20(4), 222–246 (2002)
12. Kelly, D.: Methods for Evaluating Interactive Information Retrieval Systems with Users. Foundations and Trends in Information Retrieval, vol. 3 (2009)
13. Moffat, A., Zobel, J.: Rank-biased precision for measurement of retrieval effectiveness. ACM Trans. Inf. Syst. 27(1), 1–27 (2008)
14. Robertson, S.: A new interpretation of average precision. In: SIGIR 2008, pp. 689–690. ACM, New York (2008)
15. Voorhees, E.M., Harman, D. (eds.): TREC: Experiment and Evaluation in Information Retrieval. MIT Press, Cambridge (2005)
16. Yilmaz, E., Aslam, J.A., Robertson, S.: A new rank correlation coefficient for information retrieval. In: SIGIR 2008: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 587–594. ACM, New York (2008)

Cross-Lingual Text Fragment Alignment Using Divergence from Randomness

Sirvan Yahyaei, Marco Bonzanini, and Thomas Roelleke

Queen Mary, University of London
Mile End Road, E1 4NS London, UK
{sirvan,marcob,thor}@eecs.qmul.ac.uk

Abstract. This paper describes an approach to automatically align fragments of texts of two documents in different languages. A text fragment is a list of continuous sentences and an aligned pair of fragments consists of two fragments in two documents, which are content-wise related. Cross-lingual similarity between fragments of texts is estimated based on models of divergence from randomness. A set of aligned fragments based on the similarity scores are selected to provide an alignment between sections of the two documents. Similarity measures based on divergence show strong performance in the context of cross-lingual fragment alignment in the performed experiments.

Keywords: fragment alignment, divergence from randomness, summarisation.

1 Introduction

A notable portion of the information available on the Internet is given by documents which are obtainable from more than one source. For example, the same web page might be published on different mirror web sites, or the same piece of news could be reported, in slightly different versions, possibly in different languages. This phenomenon has several implications.

In the context of web search, data redundancy in the search results has already been shown to be an issue [4]. For example, even if a document is considered to be relevant to an information need, when shown after a number of redundant documents, it does not provide the user any additional information. In other words, showing redundant documents does not benefit the user for the purpose of satisfying an information need.

Given the dynamic nature of the Web, it is common to find different versions of the same document. The task of identifying versioned or plagiarised documents, with a distinction between real plagiarism and mere topic similarity, is not trivial. Both versioning and plagiarism might affect a document as a whole, or just portions (e.g. sections, paragraphs, or more in general fragments) of it. An intelligent tool which helps in recognising duplicate text fragments could benefit editors and authors.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 14–25, 2011.
© Springer-Verlag Berlin Heidelberg 2011

To tackle one aspect of these implications, this paper investigates the possibility of aligning text fragments between documents written in two different languages. The main focus is identifying pairs of fragments with a strong content-based similarity. Figure 1 shows an example of aligning fragments of texts, which do not necessarily have the same length. Our approach, starts with measuring similarity at sentence level between the documents and then extract aligned fragments of texts based on the sentence similarities. The outcome will be a set of disjoint aligned fragments with the highest score based on the previously estimated sentence similarities.

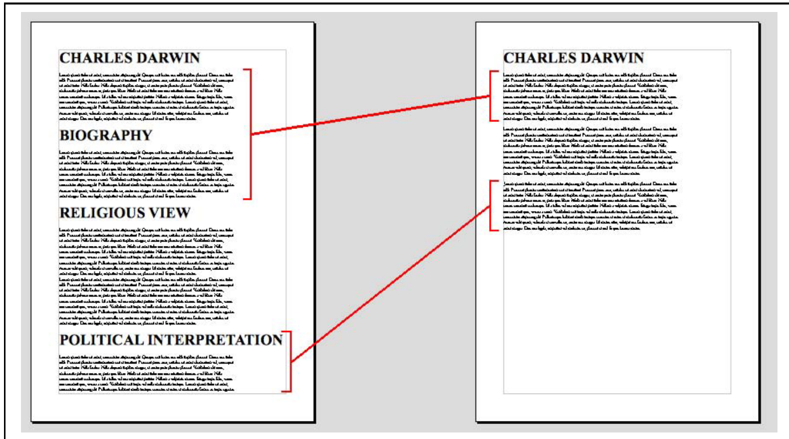


Fig. 1. An example of aligned text fragments

The main component of our method is measuring the similarity between two text fragments. We have chosen models of information retrieval based on divergence from randomness to estimate the similarities and examine the best performing model in the context of cross-lingual text alignment. An advantage of models based on divergence consists in having multiple choices of randomness models, and hence the opportunity to evaluate many IR models for this task. In addition, these models are non-parametric and do not require parameter tuning and training data to perform well.

The information about the fragments of the documents produced by the alignment algorithm, can be used later for specific applications. Such applications include the possibility of automatically creating training data sets for machine translation or document summarisation, as well as automatically synchronising complex multi-lingual web sites (e.g. Wiki-based encyclopedias, or other user-driven sites). Previous work in this area has explored both novelty detection for improving search effectiveness, and the use of fingerprinting techniques for identifying redundant documents [4], but mainly in a monolingual environment.

The remainder of this paper is organised as follows: Section 2 provides a review of current research and methods in fields related to cross-lingual text alignment.

Section 3 describes the alignment of text fragments algorithm and similarity measures to perform the sentence alignment. Construction of the test collection and experiments are reported in Section 4 and Section 5 concludes the paper.

2 Related Work

This work lays on the overlap between the two areas of document summarisation and machine translation. Despite their differences in concepts and techniques, both summarisation and translation systems are mostly built on top of statistical methods, which require training data to acquire statistical patterns. [6] propose an approach to automatically align documents to their respective summaries and extract transformation rules to shorten phrases to produce shorter and more informative summaries. Their algorithm is an extension to the standard HMM model and learns word-to-word and phrase-to-phrase alignment in an unsupervised manner.

In case of machine translation, availability of training data set is more crucial. Statistical machine translation, uses manually translated data in the forms of parallel sentences to learn translation patterns by statistical means. There has been extensive work focusing in finding parallel documents [14] and aligning sentences in fairly parallel corpora [8] and even non-parallel corpora [9]. [10] presents an approach to find sub-sentential segments from comparable corpora. Despite previous work, [14] propose a method that solely relies on textual content of the documents instead of meta-data or document structure to find near-duplicate documents. All documents are automatically translated and n-gram features are extracted to construct a small set of candidate documents in a very large collection of documents. One-by-one comparison is performed using *idf*-weighted cosine similarity among the documents in the candidate set. They report that incorporating term frequency or other retrieval ranking functions degrade the performance compared to the mentioned similarity measure. Our approach is also based on textual content only, but the alignment is performed on fragments (see Section 3) rather than sentences or entire documents.

In cross-lingual plagiarism, the aim is finding fragments of text that have been plagiarised from the source document written in a different language. [2] describe an statistical approach based on IBM model 1 [5] to retrieve the plagiarised fragment among a list of candidate fragments. The statistical approach is proposed to perform cross-lingual retrieval, bilingual classification and cross-lingual plagiarism and it focuses on the retrieval aspect of plagiarism. [12] investigates the performance and effectiveness of different models of cross-lingual retrieval for the purpose of plagiarism detection. They compare retrieval models based on parallel and comparable corpora to models based on dictionaries and syntax of the languages involved. Similarly to [2], IBM model 1 probabilities are used as translation probabilities in the statistical models and a length component is introduced to take into account the ration of length differences between the two languages.

Similar work, in a mono-lingual environment, involves the identification of redundant [4] and co-derivative [3] documents, using fingerprinting techniques.

Fingerprints are compact representations of text chunks. In these approaches, hash functions are used to calculate fingerprints of documents. Different documents are then identified as redundant, or as co-derivative, according to the fingerprint similarities. In our approach, the similarity is calculated on a fragment level, based on the content of the fragments.

3 Text Fragment Alignment

We define a text fragment as a list of continuous sentences in a document. Ideally, the content of a fragment is semantically coherent (i.e. it can be considered to be about a single topic). The aim of the proposed fragment alignment is to find fragment pairs in two documents, which are written in two different languages. Assume $\boldsymbol{d_e} =< s_{e_1}, s_{e_2}, \ldots, s_{e_n} >$ and $\boldsymbol{d_f} =< s_{f_1}, s_{f_2}, \ldots, s_{f_m} >$ are two documents in languages e and f, which contain n and m number of sentences respectively. We want to find a set of paired fragments that contains aligned text fragments that are related:

$$\{(\boldsymbol{\epsilon_i^{i'}}, \boldsymbol{\phi_j^{j'}}) | 1 \leq i \leq i' \leq n \wedge 1 \leq j \leq j' \leq m\} \tag{1}$$

where, $\boldsymbol{\epsilon_i^{i'}}$ represents a fragment that contains sentences i to i' from $\boldsymbol{d_e}$ and $\boldsymbol{\phi_j^{j'}}$ is a fragment that contains sentences j to j' from $\boldsymbol{d_f}$. Based on these definitions, fragments of a document can consist of different number of sentences and even relatively different number of sentences for each fragment in an aligned one. Since considering all the possible fragments in a document and aligning them with all the possible fragments in the other document is computationally very expensive, we restrict extracting the fragments by initial information about the alignment of sentences. The initial information is acquired by aligning sentences in the two documents and finding a few strong links between some of the sentences. A paired fragment can not contain a link to sentences outside the pair. This restriction significantly reduces the number of fragments that can be extracted.

Figure 2 sketches the text fragment alignment algorithm. The first step is to score all the sentence pairs and find a few links between the sentences. Next, all the fragments which are compatible with the links are extracted and sorted according to their scores. Finally, a set of non-overlap fragment pairs are selected as the output. It is important to note that the algorithm takes two documents as input and the computational cost only depends on the length of the documents. In other words, the algorithm of Figure 2 is run on a set of paired documents and does not depend on the document collection size.

3.1 Similarity Measures and Divergence from Randomness

A major step in finding aligned fragments of two documents is estimating similarity between sentences. As pointed out in the introduction, we have chosen a set of probabilistic models of information retrieval based on divergence from randomness [1]. A basic assumption of DFR (Divergence from Randomness)

Input: $\boldsymbol{d_e}$ and $\boldsymbol{d_f}$ {$\boldsymbol{d_e}$ is English document, $\boldsymbol{d_f}$ is foreign document}
Input: similarity threshold *min_score*
1: **for all** s_{e_i} in $\boldsymbol{d_e}$ **do**
2: **for all** s_{f_j} in $\boldsymbol{d_f}$ **do**
3: $score[i][j] \leftarrow$ estimate similarity between s_{e_i} and s_{f_j}
4: $link[i][j] \leftarrow (score[i][j] > min_score)$
5: **end for**
6: **end for**
7: $aligned \leftarrow$ extract fragment pairs compatible with *link*
8: $chosen \leftarrow \{\}$
9: **for all** *fragment* in (sort *aligned*) **do**
10: **if** *fragment* overlaps with no member of *chosen* **then**
11: $chosen \leftarrow chosen \cup fragment$
12: **end if**
13: **end for**

Fig. 2. Text fragment alignment algorithm. *aligned* is the set of all aligned fragments and *chosen* is the final set of selected fragments.

models is that non-informative words are randomly distributed in the collection. In DFR, a randomness model M is chosen to compute the probabilities and there are many ways to choose M, such as Bose-Einstein distribution or Inverse Document Frequency model. $Prob_1(\mathrm{tf})$ is defined as the probability of observing tf occurrences of a term in a randomly selected document according to M. Thus, if $Prob_1$ is relatively small for a term, then the term is an informative one. Another probability, $Prob_2$, is defined as the probability of occurrence of a term within a document with regard to a set of documents that contain the term.

The term weight, under the above definitions is the product of two factors: Firstly, information content of the term with respect to the whole collection, which is formulated as $Inf_1 = -\log_2 Prob_1$. Secondly, $Inf_2 = 1 - Prob_2$, information gain of the term with respect to its elite set, which is the set of documents that contain the term.

$$w = Inf_1 \times Inf_2 = (-\log_2 Prob_1) \times (1 - Prob_2) \tag{2}$$

Here, we are computing the similarity between two sentences in two different languages, s_e and s_f. Terms in s_f are translated based on a lexical translation model and converted to a bag-of-word with, s'_f, translation probabilities for each term. The lexical translation model is based on the IBM model 1 [5], that does not take into account the order of words in calculating the translation probabilities. The similarity between two sentences s_e and s_f is calculated as follows:

$$\mathrm{sim}(s_e, s_f) = \mathrm{sim}(s_e, s'_f) = \sum_{t \in \{s_e \cap s'_f\} \wedge \tau \in s_f} w_M(t, s_e) \times p(t|\tau) \tag{3}$$

where, $w(t, s_e)$ is the weight if term t in sentence s_e according to similarity model M and $p(t|\tau)$ is the translation probability of translating τ to t. The collection

Table 1. Similarity measures used to estimate the similarity between sentences. For detailed information on each model, please refer to [1].

	Name	Description
1	TF-IDF	The tf . idf weighting function, where tf is the total term frequency and *idf* is Sparck-Jones' formulation
2	TF_k-IDF	Same as above but with the BM25 tf quantification $\frac{tf}{tf+k}$
3	$I(n)L2$	Model with Inverse document frequency, with Laplace after-effect and 2nd normalisation
4	$I(F)B2$	Model with Inverse of the term frequency, with Bernoulli after-effect and 2nd normalisation
5	$I(n_e)B2$	Model with Inverse of the expected document frequency, with Bernoulli after-effect and 2nd normalisation in base 2
6	$I(n_e)C2$	Model with Inverse of the expected document frequency, with Bernoulli after-effect and 2nd normalisation in base e
7	$BB2$	Limiting form of Bose-Einstein, with Bernoulli after-effect and 2nd normalisation
8	$PL2$	Poisson approximation of the binomial model, with Laplace after-effect and 2nd normalisation
9	BM25b	BM25 probabilistic model
10	OkapiBM25	Okapi formulation of BM25; the same as BM25b with within-query term frequency (k_3) set to 0

for equation 3 is $\boldsymbol{d_e}$, which is the document that contains s_e and all the collection statistics in the similarity measures are computed based on $\boldsymbol{d_e}$. Table 1 shows a list of all the models used in this work to estimate the sentence similarity between two documents.

3.2 Extraction of Fragments

After scoring all the sentence pairs, only those with similarity score higher than a certain threshold are aligned. Aligned fragments are extracted by an algorithm adopted from phrase-based statistical machine translation [11]. Simply, two fragments are aligned if no sentence inside them is aligned to sentences outside the fragments and there is at least one link between the two fragments. Fragments in an extracted fragment pair are only aligned to each other and not to any fragment outside the fragment pair.

Many of the extracted aligned fragments overlap and there are sentences which belong to more than one fragment. Therefore, we sort all the aligned fragments according to their similarity score and drop those with lower scores and overlap. The score of an aligned fragment is estimated by averaging the similarity scores of its sentence pairs computed before. The remaining aligned fragments are the result of the algorithm.

4 Experimental Study

Since we did not have a manually annotated documents with aligned fragments, a pseudo-collection is constructed to perform the experiments. A collection of

documents and their summaries in English and Italian is built by crawling the web-site of the Press releases of the European Union[1] and pseudo-documents are created by randomly concatenating documents and summaries to each other. For the English side, x documents are randomly chosen and concatenated to create a document with multiple topics. On the Italian side, y documents are randomly chosen, added to the set of x aligned summaries of the chosen documents and randomly concatenated. As a result, we have an English document consisting of x documents and an Italian document consisting of $x + y$ summaries, including the summaries of the English documents. The task is now defined as aligning all the sentences of the summaries to their correct English documents or to not-align those with no corresponding document. In other words, in the English side there are x documents and in the Italian side there summaries with y more summaries mixed with them. Our algorithm tries to align the summaries to their corresponding documents. Table 2 shows statistics of the corpus. All the documents and summaries in the collection are processed by tokenisation, lower-casing and sentence splitting.

Table 2. English-Italian corpus statistics

	English	**Italian**	**Average**
Mean Document Length (sentences)	34.66	35.29	34.96
Mean Summary Length (sentences)	5.09	4.87	4.98
Mean Compression Ratio (sentences)	14.68%	13.81%	14.26%
Mean Document Length (words)	794.85	874.73	834.79
Mean Summary Length (words)	106.08	118.74	112.43
Mean Compression Ratio (words)	13.35%	13.58%	13.47%
Number of document/summary pairs		192	

4.1 Document-Summary Association

As a basic task compared to finding aligned fragments of text, we examine the problem of associating documents to their summaries. Association is the process of finding two related structures in a collection of structures. In a collection of documents and summaries, the aim is to find the most related summary to each document. We assume that there is a one-to-one association between the summaries and the documents.

The association process can be performed in two ways. Firstly, a two-stage method which translates and summarises the document and computes the similarity between the summaries. Secondly, a one-stage cross-lingual association approach that directly calculates the similarity between the document and the summary in different languages. An illustration of English-to-Italian association is drawn in Figure 3, which shows the two ways that the association can be performed in. The one-stage approach estimates the similarity between the document and the summary according to equation 3, but instead of similarity between sentences, its the similarity between documents and summaries.

[1] Available at `http://europa.eu/rapid`

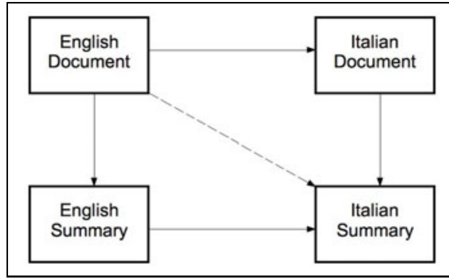


Fig. 3. Cross-lingual Summarisation Pipelines: Two-Stage vs. One-Stage

In the two-stage approach, the summarisation component relies on MEAD [13], which is an extractive summariser. The machine translation system used for translation form Italian to English is a phrase-based statistical MT system with translation model and language model as its main components. The full detail of the system is described in [15]. The training data for the SMT system is taken from the Europarl corpus [7]. 1.6 million parallel sentences were used for building the translation model and 50 million sentences to train the English language model. For both approaches, lexical probabilities are estimated based on IBM model 1 and the parallel training data mentioned before.

The scores for the one-stage system, which associates English documents to Italian summaries, are shown in Table 3, where one can observe that the OkapiBM25 function is performing the best. The best scores for the two-stage method are P@1= 78.1% and MRR= 82.0 and the results of the two-stage approach are in all the cases substantially lower than the one-stage one.

Table 3. Results of document-to-summary association of the one-stage approach with different similarity measures

Similarity	P@1	MRR	Similarity	P@1	MRR
TF-IDF	88.6	92.1	$I(n)L2$	90.1	93.1
$\mathrm{TF_k}$-IDF	89.1	92.4	$I(F)B2$	81.8	86.9
IDF	86.0	89.8	$I(n_e)B2$	86.5	90.6
BM25b	89.6	93.0	$I(n_e)C2$	86.0	89.9
OkapiBM25	**91.7**	**94.3**	$PL2$	90.1	93.2

In the two-stage approach approach, the summarisation and translation tasks lead to a loss of information which cannot be adequately captured by the association functions we have examined. After performing the association of English summaries and MEAD generated summaries from the documents, a basic similarity measure such as TF-IDF achieved a P@1 score of 98.0 and MRR of 99.2. This means that the translation component is the major source of precision loss in the two-stage method. The translation component translates each Italian sentence to exactly one English sentence. For translating each sentence, it selects

the translation with highest score according to its model to produce a fluent English. The produced sentence only contains one possible translation for each word or phrase. On the other hand, the one-stage approach considers all the possible translations in the lexical model for each word, hence having a higher chance of finding a match between document words and summary words. The 91% success rate of the one-stage approach, shows it is possible to associate the majority of the summaries to their documents in this collection. The results of the text fragment alignment show the difficulty of finding the same summaries, while they are mixed with other summaries.

4.2 Text Fragment Alignment Evaluation

To find out the cross-lingual effect of the task, we performed the text fragment alignment algorithm on mono-lingual data as well as the cross-lingual data. For each word only the top 5 translations based on the their translation weights are picked. The threshold is set to the average score of the alignment links, therefore alignment links with score less than the average are discarded. For each similarity measure, the alignment algorithm is run $2,000$ times to select different variations of the documents and summaries.

The goal of text fragment alignment is to find the longest relevant fragments of text on each side, without including irrelevant sentences. Therefore, both recall and precision are important in evaluating the algorithm. F-measure combines the two, to give one single score to demonstrate the performance of the algorithm. To calculate the F-measure, each sentence on the e side is labelled true positive if it belongs to a fragment, which is fully or partially correctly aligned. The sentence is labelled false positive if it belongs to a fragment which is incorrectly aligned. It is a false positive instance, if it is not aligned and it should not have been. A false negative instance is an unaligned sentence, which should have been aligned. F-measure is calculated based on these labels for both sides, English to foreign and foreign to English.

Table 4 shows the results of both mono-lingual and cross-lingual text fragment alignment experiments. As expected, the results of the mono-lingual text fragment alignment are higher than the cross-lingual runs. In all settings and in both directions (source to target and target to source), models based on DFR substantially outperform TF-IDF weighting methods. In both mono-lingual and cross-lingual runs OkapiBM25 performs consistently very well compared to others. It has been pointed out by [1] that BM25 formula can be derived from the model $I(n)L2$, which has the highest score in the target to source cross-lingual runs and it is very close to other BM25 scores. Substantial drop of F-measure score of the target to source direction of the cross-lingual runs compared to mono-lingual ones, shows that the summary to document alignment is more prone to translation than the other direction.

Two important components of all similarity methods used in these experiments, are document length and average document length in the collection. These factors are considered to reduce the effect of variance in document length in text collections. However, since in our experiments, a document is the collection

Table 4. The results of text fragment alignment, for mono-lingual and cross-lingual. For mono-lingual, source and target (src2trg and trg2src) are both English documents and summaries. In cross-lingual settings, source is English documents and target is Italian summaries.

	Mono-lingual				Cross-lingual			
	μF_1	μF_1	MF_1	MF_1	μF_1	μF_1	MF_1	MF_1
Similarity	**src2trg**	**trg2src**	**src2trg**	**trg2src**	**src2trg**	**trg2src**	**src2trg**	**trg2src**
TF-IDF	33.5	77.0	34.9	77.0	23.0	28.5	23.6	27.0
TF_k-IDF	35.2	76.4	35.4	76.3	22.4	28.7	21.5	26.6
$I(n)L2$	35.8	80.2	35.7	80.3	30.0	**32.9**	29.4	**31.8**
$BB2$	34.5	**88.1**	34.9	**88.0**	27.6	32.3	28.2	31.4
$I(F)B2$	35.0	81.9	35.2	81.9	27.4	31.6	27.9	30.4
$I(n_e)B2$	34.9	74.3	35.4	74.2	27.9	31.9	28.4	30.7
$I(n_e)C2$	**38.3**	71.7	**38.2**	71.5	29.0	31.4	28.7	30.3
$PL2$	35.8	79.1	35.5	79.0	29.6	32.5	28.8	31.2
BM25b	36.7	72.4	36.7	72.1	30.8	32.5	30.1	31.3
OkapiBM25	37.3	71.3	37.5	71.1	**31.5**	31.9	**31.0**	31.0

and its sentences are the documents, the variance of document length does not exist. To see the effect of this fact, we investigated two other ways to estimate sentence length and used them instead of the default method, which was number of tokens. One is sum of the term frequency in the document for each term in the sentence[2] and the other one, the sum of their selectivity (inverse sentence frequency)[3]. Both methods produced different results for all the runs, however, they were most of the times slightly worse than the number of tokens, and in general the differences were negligible. Only for TF-IDF similarity, the sum of the selectivity of the terms performs slightly better than the number of tokens, but in all other cases it was behind the latter. We concluded that even though there is a difference between sentence length variation and document length variation in large collections, the DFR models perform well, regardless of length estimation method, in the context of sentence similarity.

5 Conclusion and Future Work

We developed an algorithm to perform cross-lingual text fragment alignment and ran a series of experiments with different similarity measures based on models of divergence from randomness. The results show that term statistics based on divergence models are consistently superior to TF-IDF schemes. Despite the fact that sentences tend to be similar in length, we discovered that other ways of estimating sentence length does not improve the quality of the alignment compared to the basic method of counting the number of the tokens. In addition,

[2] $\text{len_tf}(s, \boldsymbol{d}) := \sum_{t \in s} \text{tf}(t, \boldsymbol{d})$, where s is a sentence in document $\boldsymbol{d}$.

[3] $\text{len_isf}(s, \boldsymbol{d}) := \sum_{t \in s} \text{sf}(t, \boldsymbol{d})^{-1}$, where s is a sentence in document $\boldsymbol{d}$ and $\text{sf}(t, \boldsymbol{d})$ is the number of sentences in $\boldsymbol{d}$ that contain t.

for the source to target alignment the cross-lingual scores were not substantially lower than the mono-lingual ones, which shows that the translation component performs well enough not to degrade the overall performance considerably.

Preliminary investigation of cross-lingual association of documents and their summaries showed that a one-stage direct computation of similarity using a probabilistic dictionary (lexical probabilities) significantly outperforms a method that translates and summaries the documents and estimates a mono-lingual similarity between the documents. Experiments on mono-lingual associating of generated summaries and manual summaries showed that the low performance of the two-stage method is mainly due to the selective nature of the translation component. One translation is chosen among a list of possible translations based on the context of the sentence and the rest of the candidates are discarded, therefore, the chance of a match between the words of the two documents are heavily degraded.

Although the scores of the basic similarity measures were lower than most of the models of DFR in the association task, the difference was not substantial. In other words, even the basic models of similarity performed well in finding the corresponding summary for a document in our experiments.

These research results can be used to align multi-lingual content in resources such as Wikipedia, or other Wiki-based web sites, where the documents are often not parallel in the different languages.

Acknowledgements. We would like to give special thanks to Hany Azzam for his valuable comments on the early draft of this work.

References

1. Amati, G., Van Rijsbergen, C.J.: Probabilistic models of information retrieval based on measuring the divergence from randomness. ACM Trans. Inf. Syst. 20, 357–389 (2002)
2. Barrón-Cedeño, A., Rosso, P., Pinto, D., Juan, A.: On cross-lingual plagiarism analysis using a statistical model. In: Proceedings of the ECAI 2008 PAN Workshop: Uncovering Plagiarism, Authorship and Social Software Misuse, Patras, Greece, pp. 9–13 (July 2008)
3. Bernstein, Y., Zobel, J.: A scalable system for identifying co-derivative documents. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 55–67. Springer, Heidelberg (2004)
4. Bernstein, Y., Zobel, J.: Redundant documents and search effectiveness. In: Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, pp. 736–743 (November 2005)
5. Brown, P.F., Pietra, V.J.D., Pietra, S.A.D., Mercer, R.L.: The mathematics of statistical machine translation: Parameter estimation. Comput. Linguist. 19(2), 263–311 (1993)
6. Daumé III, H., Marcu, D.: A phrase-based HMM approach to document/abstract alignment. In: Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing (EMNLP), Barcelona, Spain, pp. 119–126 (July 2004)
7. Koehn, P.: Europarl: A parallel corpus for statistical machine translations. In: MT Summit X, Phuket, Thailand, pp. 79–86 (September 2005)

8. Ma, X.: Champollion: A robust parallel text sentence aligner. In: Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC), Genova, Italy (May 2006)
9. Munteanu, D.S., Marcu, D.: Improving machine translation performance by exploiting non-parallel corpora. Comput. Linguist. 31, 477–504 (2005)
10. Munteanu, D.S., Marcu, D.: Extracting parallel sub-sentential fragments from non-parallel corpora. In: Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL), Sydney, Australia, pp. 81–88 (July 2006)
11. Och, F.J., Tillmann, C., Ney, H.: Improved alignment models for statistical machine translation. In: Proceedings of the Joint SIGDAT Conference of Empirical Methods in Natural Language Processing and Very Large Corpora, pp. 20–28. College Park, MD (1999)
12. Pouliquen, B., Steinberger, R., Ignat, C.: Automatic identification of document translations in large multilingual document collections. In: Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP), pp. 401–408 (September 2003)
13. Radev, D., Allison, T., Blair-Goldensohn, S., Blitzer, J., Çelebi, A., Dimitrov, S., Drabek, E., Hakim, A., Lam, W., Liu, D., Otterbacher, J., Qi, H., Saggion, H., Teufel, S., Topper, M., Winkel, A., Zhang, Z.: MEAD - a platform for multidocument multilingual text summarization. In: LREC 2004, Lisbon, Portugal (2004)
14. Uszkoreit, J., Ponte, J.M., Popat, A.C., Dubiner, M.: Large scale parallel document mining for machine translation. In: Proceedings of the 23rd International Conference on Computational Linguistics (COLING), Beijing, China, pp. 1101–1109 (August 2010)
15. Yahyaei, S., Monz, C.: The QMUL system description for IWSLT 2010. In: Proceedings of the Seventh International Workshop on Spoken Language Translation (IWSLT), Paris, France, pp. 157–162 (December 2010)

Enhancing Document Snippets Using Temporal Information

Omar Alonso[1], Michael Gertz[2], and Ricardo Baeza-Yates[3]

[1] Microsoft Corp., Mountain View, California, U.S.A.
omalonso@microsoft.com
[2] Institute of Computer Science, Heidelberg University, Germany
gertz@informatik.uni-heidelberg.de
[3] Yahoo! Research, Barcelona, Spain
rbaeza@acm.org

Abstract. In this paper we propose an algorithm to enhance the quality of document snippets shown in a search engine by using temporal expressions. We evaluate our proposal in a subset of the Wikipedia corpus using crowdsourcing, showing that snippets that have temporal information are preferred by the users.

1 Introduction

A very important feature of modern search engines is to present a *snippet* (also called excerpt or abstract) as part of each entry in the result list. Snippets are very popular because they provide a short summary of the retrieved document and users can quickly scan the few lines and assess its relevance. A challenge for constructing snippets is to determine the most relevant portions of the document and select the best fragments for presentation.

The temporal coverage and specificity of documents provide users with important *relevance cues* about time related information embedded in the documents and thus can be very useful in subsequent document exploration and search tasks. Based on this premise, we study the *temporal order* of a document, which concentrates on how the time information in a temporal coverage is actually organized in a document.

In this work, we use the concept of temporal order to enhance document snippets presented by search engines as part of the answers to user queries. Enhancing a snippet does not necessarily make it more relevant, but it gives a better context and, for many time related queries, an immediate answer. To achieve this, we present a method that selects a frequent temporal expression related to the query and adds that sentence that contains the expression to the snippets.

This paper is organized as follows. In Section 2, we present the concept of temporal order in a document. In Section 3, we give an algorithm that enhances snippets by temporal information. Then, in Section 4, we evaluate this enhancement using a subset of Wikipedia and crowdsourcing, showing that most of the time users prefer the enhanced snippet. We conclude the paper with some final remarks in Section 5.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 26–31, 2011.
© Springer-Verlag Berlin Heidelberg 2011

2 Temporal Order

As the basis for analyzing a document in time, we adopt the temporal framework presented in [2]. We assume a discrete representation of time based on the Gregorian Calendar, with a single day being an atomic time interval called *chronon*. Assume a document d from a document collection $\mathcal{D}$. Using the approach presented in [2], one can determine the sequence $t\text{-}seq(d) = \langle(c_1, p_1), (c_2, p_2), \cdots, (c_k, p_k)\rangle$ of chronon/position pairs that describe the temporal expressions occurring in d. We define *temporal richness* of d, denoted $t\text{-}rich(d)$, as the ratio of the number of chronons in d to the number of chronons in the collection $\mathcal{D}$. The higher this ratio, the richer a document d is in its temporal information content. A document d is said to be *temporally most specific* with respect to chronons of type $T \in \{\text{century, year, month, week, day}\}$, if most of the chronons in $t\text{-}seq(d)$ are of type T.

The approach applied to identify time related expressions in text data is named-entity extraction, with temporal entities being time-related concepts that need to be identified in the text. Such concepts are represented in the document text as sequences of tokens or words called *temporal expressions* [3]. In the following, we assume that the collection D has been temporally annotated with a tagger and has been marked in some standard such as TimeML[1].

The above measures are coarse-grained, i.e., they aim at describing *aggregated* information about the temporal content of a document. In the following, we introduce the concept of temporal order to represent the occurrences of chronons in document order.

A histogram based *temporal coverage* naturally describes a document's content using periods of time. It details how such periods are emphasized in terms of chronon distributions, frequencies and temporal specificity. In addition to this *content-specific* measure, we introduce another measure that helps describing how the temporal content and underlying chronons are *ordered* in a document. This measure is called *t-order* and is based on the positions chronons occur in a document.

Assume again $t\text{-}seq(d) = \langle(c_1, p_1), (c_2, p_2), \cdots, (c_k, p_k)\rangle$ is given for a document d with document length $len(d)$ and temporal boundaries $t\text{-}low(d)$ and $t\text{-}high(d)$, denoting the earliest and latest chronon in d, respectively. A chronological order for d, denoted $t\text{-}order(d)$, is constructed in a position/time coordinate system as follows: the range of the x-axis is 1 to $len(d)$, specifying the different token positions in d in increasing order. The range of the y-axis is determined by $t\text{-}low(d)$ (as minimum value) and $t\text{-}high(d)$ (as maximum value). Conceptually, the units underlying the y-axis are based on the finest granularity of any chronon in $t\text{-}seq(d)$. Assume the units underlying the y-axis are based on months (for other granularities the approach is analogous), that is, all chronons in $t\text{-}seq(d)$ are of type month or coarser. Each chronon/position pair (c_i, p_i) in $t\text{-}seq(d)$ is now processed as follows. If c_i is of type month, then simply a point at (p_i, c_i) is plotted. If the type of c_i is of granularity coarser than month, then

[1] http://timeml.org

a vertical bar is plotted at position p_i, where the bar covers all the y-values that are contained by c_i. For example, if c_i is of type year, then the bar at the position p_i spans twelve months.

Figure 1 indicates the *t-order* for two Wikipedia pages related to Pisa: the leaning tower[2] and the city[3]. The x-axis represents the position of the temporal expression in the document and y-axis represents chronons. The horizontal bar denotes the current timestamp ts = "April 2011" (year 2011 and 21st century, respectively). Note that in the case of the city, the predominant chronon type is centuries where in the page about the tower, it is year. Clearly, most of the temporal expressions refer to points in time in the past.

The rationale for having t-orders in addition to histograms is to describe how time periods represented by a histogram are in fact organized in the document. For example, two documents might have very similar histograms, but their t-orders might look completely different. One t-order, for example, might clearly show that the time period(s) are covered in a true chronological fashion, starting with the earliest chronon at the beginning of the document and having the most recent chronon occurring at the end of the document. In the second t-order, on the other hand, there might be no such patterns of (partial) chronological order. There are certain types of documents for which one would expect that many of the chronons derived from the document's content follow such a monotonicity property. Visualizing a t-order in addition to a temporal coverage for a document, of course, might give the user further cues on the temporal information content.

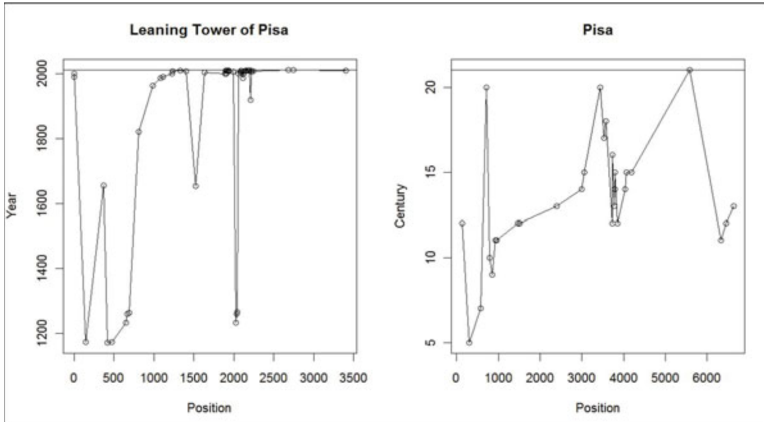


Fig. 1. Two examples of t-order for documents about Pisa

3 Enhancing Snippets with Temporal Information

Temporal order can also be helpful for detecting the frequency of certain chronons and where they occur in the document. If a temporal expression is very frequent,

[2] http://en.wikipedia.org/wiki/Leaning_Tower_of_Pisa

[3] http://en.wikipedia.org/wiki/Pisa

then we believe that this is an indication that the sentences containing such temporal expression are relevant to the user. As pointed out in [1], temporal information can be useful when designing document snippets. Let's take a look at the snippet for the query [Pisa] that a Web search engine would return, as presented in Figure 2.

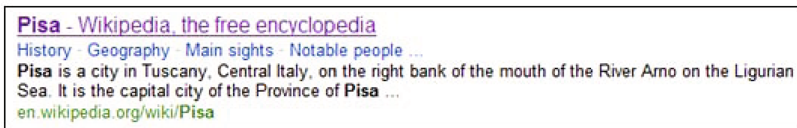


Fig. 2. Web snippet for the query [Pisa]

We claim that a snippet is more *informative* if it contains at least one frequent temporal expression. We consider a sentence that contains a highly frequent temporal expression to be representative of a *milestone in time*, that is, a significant event in the context of the document. In our example, "12th century" is the most common temporal expression in the document. The modified example for the [Pisa] snippet then is

```
Pisa is a city in Tuscany, Central Italy, on the right bank of the mouth
of the River Arno on the Ligurian Sea ... The city is also home of the
University of Pisa, which has a history going back to the 12th century.
```

What we are interested in is extracting a sentence, s_t, that contains a frequent temporal expression from the document and exchange its placement with the *last* sentence in the original snippet (usually second or third, depending on the Web search engine). The replacement step can be done with a query-dependent or query-independent approach. For this, we use the following algorithm **S**:

- **S1.** For a given a document d, construct $t\text{-}order(d)$
- **S2.** Compute frequency of temporal expressions and determine a list of all *timex t* elements for each temporal expression. A $timex_{id,f,t}$ is the description of the temporal expression identified by the temporal tagger. As example, for a short fragment of temporal expressions, the list would be (1991:2, (t6, t7)), (5th century:1 (t5)).
- **S3.** [Loop] For each sentence s in d, compute the average sentence length s_{len} and the average length of sentences that contain temporal expressions. Compute the mean position $\bar{x}$ and standard deviation σ of $timex_{id,t}$ within sentence s.
- **S4.** [Loop] Select the most frequent temporal expression $timex_{id}$ and for each t_i in $timex_{id}$:
  - Extract surrounding text fragment $text_{chunk}$ with clamp c_ℓ and c_r, where c_ℓ denotes number of characters to the left and c_r to the right of the position of t_i, respectively.
  - Clamps c_ℓ and c_r are set based on $\bar{x}$ and σ.
  - Extract sentence s_i within text fragment $text_{chunk}$ and store it in $list_s$.

- **S5.** Extract last sentence s_ℓ from snippet(d).
- **S6.** Compute Jaccard similarity $sim(s_i, s_\ell) = \frac{s_i \cap s_\ell}{s_i \cup s_\ell}$. If similarity is less than a threshold δ, then replace s_ℓ with s_i. Otherwise, select next sentence s_{i+1} from $list_s$ and compute similarity measure again.
- **S7.** Return snippet(d_t) that contains replacement sentence.

A beneficial side effect of this method is that it determines a complete sentence instead of chopped lines. Chopped lines have been a general problem for the constructions of snippets, as reported in [4].

4 Experimental Results

In this section we present the experiments we conducted using the techniques described in the previous sections and discuss the results. A high quality subset of articles with excellent content from Wikipedia are "featured articles". Featured articles are considered to be the best articles, as determined by Wikipedia's editors who review them according to accuracy, neutrality, completeness, and style.

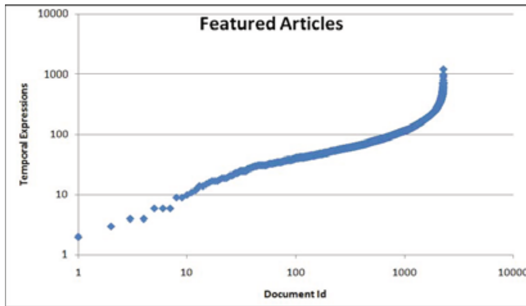


Fig. 3. Distribution of temporal expressions in Wikipedia's featured articles collection

The experimental setup is as follows. We pre-process the Wikipedia data set through a temporal annotation document pipeline that includes a Part-Of-Speech component, a temporal tagger, and other modules that compute all basic temporal measures. The distribution of temporal expressions is shown in Figure 3 in a log-log graph, to show that the distribution is a polynomial in the central part (the number of temporal expressions is approximately $3.3\sqrt{doc_{id}}$). The average number of expressions is 160.8 with a standard deviation of 115.8. This number may seem high, but Wikipedia entries have many references at the end that include temporal expressions. For this reason, Wikipedia is a good data set for temporal-based retrieval. Our query set consists of all document titles from our Wikipedia document collection (e.g., "Belton House", "1910 Cuba hurricane" etc.). We use a search engine API to extract the title, URL, and original snippet for the query set. We then apply our technique as described in Section 3 and produce the new snippets.

We use a crowdsourcing approach[4] to test our technique by selecting a random sample of the queries from the data set. The experiment design consists of showing a user a query and two snippets A and B. The worker has to select if, giving the query q, A is better, B is better or if both are the same. We randomize the order so it is not obvious which is the control and which is the treatment group. We pay $0.02 cents per task and require 5 workers each for aggregation and consensus of results. We also include a 10% of honey pots (well known answer in advance) as quality control mechanism for managing the crowd. Recall from Section 3 that our approach modifies the original snippet by replacing some text. The exact same title and URL were presented to the workers in case they wanted to look at the article in detail.

The results show that 42% of the time the modified snippet was preferred and in 35% of the time the original snippet was selected. For the remaining 23%, it was not possible to reach a consensus among workers. An example would be that 2 workers selected snippet A, 2 workers B, and the third one selected that A and B look the same. One possible explanation would be that the extra sentence that the sentence replaced doesn't give additional information to the user.

5 Conclusions and Outlook

The temporal order of a document can be valuable for selecting sentences or text fragments that should be relevant to the user. The combination of different temporal document measures can be very useful for understanding a document collection or individual documents. In this paper, we leverage temporal order to enhance a traditional document snippet by replacing part of the original content with a sentence that contains the most frequent temporal expression. The evaluation of our approach using a subset of Wikipedia indicates that users prefer the modified snippet version to the original one. Still, for cases where there is a tie, more experimentation and analysis need to be conducted.

References

1. Alonso, O., Strötgen, J., Baeza-Yates, R., Gertz, M.: Temporal Information Retrieval: Challenges and Opportunities. In: TWAW Workshop, WWW 2011, pp. 1–8 (2011)
2. Alonso, O., Gertz, M., Baeza-Yates, R.: Temporal analysis of document collections: Framework and applications. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 290–296. Springer, Heidelberg (2010)
3. Mani, I., Pustejovsky, J., Gaizauskas, R. (eds.): The Language of Time. Oxford University Press, Oxford (2005)
4. Rose, D., Orr, D., Kantamneni, R.: Summary Attributes and Perceived Search Quality. In: 16th WWW, pp. 1201–1202 (2007)

[4] www.mturk.com

Spaced Seeds Design Using Perfect Rulers*

Lavinia Egidi and Giovanni Manzini

Dipartimento di Informatica, Università del Piemonte Orientale, Italy
{lavinia.egidi,giovanni.manzini}@mfn.unipmn.it

Abstract. We consider the problem of lossless spaced seed design for approximate pattern matching. We show that, using mathematical objects known as perfect rulers, we can derive a family of spaced seeds for matching with up to two errors. We analyze these seeds with respect to the trade-off they offer between seed weight and the minimum length of the pattern to be matched. We prove that for patterns of length up to a few hundreds our seeds have a larger weight, hence a better filtration efficiency, than the ones known in the literature. In this context, we study in depth the specific case of *Wichmann rulers* and prove some preliminary results on the generalization of our approach to the larger class of *unrestricted rulers*.

1 Introduction

The use of spaced seeds for approximate pattern matching has been introduced in [1,12] and since then has received considerable attention. Spaced seeds are used to quickly filter-out highly dissimilar regions, and they are a fundamental tool, for example, for mapping to a reference genome the millions of reads produced by modern sequencing technologies (see [9] and references therein).

We consider the problem of designing spaced seeds to be used for detecting whether two strings of length m are at Hamming distance at most k; in the literature this is known as the (m, k)-detection problem. In particular we are interested in the design of lossless seeds, i.e., seeds that find *all* matches with the above properties. Spaced seeds consist of *solid* positions and *don't care* positions. The number of solid positions is called the seed *weight*. For a given pair of values (m, k) we want to find a seed with the largest possible weight since, under standard assumptions, this maximizes the filtration efficiency.

For the problem of lossless seed design, an important breakthrough has been obtained in [4] where, for any given pair (m, k), the authors provide a spaced seed with an asymptotically optimal weight. Although this result essentially solves the problem from the theoretical point of view, it remains open the problem of finding optimal, i.e. weight-maximal, seeds for the pattern lengths m used in practice (i.e. up to a few hundreds): the seeds in [4] are asymptotically optimal as $m \to \infty$, but we have no guarantees on their quality for small m. For a given pair

* This research is founded by the BioBITS Project *Converging Technologies* 2007, area: Biotechnology-ICT, Regione Piemonte.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 32–43, 2011.
© Springer-Verlag Berlin Heidelberg 2011

(m, k) one can find an optimal seed using a combinatorial search algorithm, but this problem is known to be a hard one [5,6,8,11,14] so the problem of designing whole families of (suboptimal) seeds of practical interest is still open.

Our starting point is the observation that we can derive a family of lossless spaced seeds using mathematical objects known as *perfect rulers* (sometimes also called *difference bases*) [3,7,15]. Informally, a perfect d-ruler is a binary string with a minimal number of **1**'s with the property that for any positive $\delta \leq d$ there exist two **1**'s at distance δ (see Section 2 for further details). This structural property makes them suitable to design spaced seeds able to detect strings at Hamming distance at most 2. The study of the properties of these spaced seeds is the main objective of the paper.

As a first step, in Section 3 we analyze the seeds obtained from perfect rulers with respect to the tradeoff they offer between seed weight and the minimum m^* for which the seed is guaranteed to solve the $(m, 2)$-detection problem for all $m \geq m^*$. In Theorem 1 we establish an upper bound for m^* for all "interesting" seeds derived from perfect rules. This upper bound suffices to establish that for m up to 498 the seeds derived from perfect rulers have a larger weight than the asymptotically optimal seeds defined in [4] and therefore justifies an in-depth study of this family of seeds.

In Section 4 we refine our analysis by establishing *lower bounds* on the minimum pattern length m^*. First, we prove that the upper bound of Theorem 1 is tight for the seed of maximal weight derived from a given ruler (Corollary 1). Then, we introduce the concept of skewness of a ruler and use it to derive general lower bounds for m^* (Theorem 3).

In Section 5, we analyze the special case of Wichmann rulers [15] which are a family of rulers particularly important since they can be easily derived by a "generating function", whereas other perfect rulers are usually found by trial and error. For Wichmann rulers we show that the upper bound of Theorem 1 is almost tight applying the results of Section 4 (Theorem 4) and with an ad-hoc analysis (Theorem 5).

Finally, in Section 6 we consider spaced seeds obtained from *unrestricted* rulers [7], which are a natural generalization of perfect rulers. We show that some of the results of the previous sections can be applied to unrestricted rulers as well. Although we do not provide a complete analysis, our preliminary results show that, somewhat counterintuitively, spaced seeds derived from unrestricted rulers are less effective that the ones derived from perfect rulers.

Due to space constraints we omit some of the technical proofs. Full details can be found in [2].

2 Notation

For spaced seeds we follow the notation introduced in [1,6]. A *spaced seed* is a string over the alphabet $\{\texttt{\#}, \texttt{-}\}$; the symbol '#' represents a solid position, the symbol '-' a don't care position. Informally, a spaced seed defines a set of non-contiguous positions in which we require two sequences to match. We say that a

spaced seed S solves the (m,k)-problem if for any pair of strings σ_1, σ_2 of length m and Hamming distance k, there exists an index i such that

$$S[j] = \texttt{\#} \qquad \Longrightarrow \qquad \sigma_1[i+j] = \sigma_2[i+j]. \tag{1}$$

In other words, we require that, starting from position i, the strings σ_1, σ_2 contain the same symbols in every position corresponding to a '#' in S, while we tolerate mismatches in positions corresponding to '-' in S.

In the context of approximate string matching, if a seed solves the (m,k) problem it can be used as a filter to quickly discard regions which *are not* at Hamming distance at most k (see [1,6] for further details). Note however, that (1) can hold for a given i even if σ_1 and σ_2 are not at Hamming distance k. These events are called *false positive matches* and it is desirable to reduce their number as much as possibile. The *weight* of a seed is defined as the number of #'s in it. Under standard assumptions, see [4, Sect. 1.1] the number of false positives decreases exponentially with the seed weight. Thus, it is desirable to solve the (m,k) problem with a seed with the largest possible weight.

The notion of *perfect ruler*, has been studied by mathematicians for more than sixty years [3,7,15] (in earlier works rulers were called *difference bases*). Here we recall the basic definitions using modern terminology [10]. We base the definition of rulers on the concept of *measure*:

Definition 1 (Measure). *Let U be a binary string. For any positive integer δ we say that U* measures *δ if there exist i,j, $0 \le i < j < |U|$, such that $j - i = \delta$ and $U[i] = U[j] = \mathbf{1}$. The pair (i,j) is said to be a* measure *of δ in U.* □

Definition 2 (Complete ruler). *Let R be a binary string of length $d+1$ such that $R[0] = \mathbf{1}$, $R[d] = \mathbf{1}$, and such that for any integer $\delta, 0 \le \delta \le d$, R measures δ. The string R is said to be a* complete d-ruler*, or simply a* complete ruler *when the length of R is clear from the context.* □

Intuitively, using the $\mathbf{1}$'s as marks, with a complete d-ruler we can measure all distances between 1 and d. For example, the string **110101** is a complete 5-ruler. Note that even the string $\mathbf{1}^6 = \mathbf{111111}$ is a complete 5-ruler, but not an interesting one: the challenge of rule design is to find complete d-rulers with as few marks as possible. This notion is captured by the following definition.

Definition 3 (Perfect ruler). *Let R be a complete d-ruler containing ℓ $\mathbf{1}$'s. If there exists no complete d-ruler with less than ℓ $\mathbf{1}$'s then R is said to be a* perfect d-ruler. □

Let $\ell(d)$ denote the number of $\mathbf{1}$'s in a perfect d-ruler. Table 1 reports the values $\ell(d)$ for $d = 10, \ldots, 90$. In [15] it is proven that $\lim_{d\to\infty}(\ell^2(d)/d)$ exists and that such limit is between 2.434 and 3. Perfect rulers are not easy to find: they are usually generated by exhaustive search procedures. An important exception are Wichman rulers which are discussed in Section 5. Tables of all perfect rulers of size up to 101 are available on the net [10].

Table 1. Number of **1**'s $\ell(d)$ in a perfect d-ruler for $d = 10, \ldots, 90$

d	10–13	14–17	18–23	24–29	30–36	37–43	44–50	51–58	59–68	69–79	80–90
$\ell(d)$	6	7	8	9	10	11	12	13	14	15	16

Perfect rulers should not be confused with Golomb rulers that measure each integer at most once (they are not necessarily complete). Golomb rulers have been used in [13,14] in relation to seed design, but with the totally different aim of analyzing the hardness of seed optimization.

3 From Rulers to Spaced Seeds

The structure of complete rulers naturally suggests their use for the design of spaced seeds. Given a d-ruler R, if we replace each **0** with a '#' symbol and each **1** with a '-' symbol we obtain a seed in which there is a pair of don't care symbols at distance δ for $\delta = 1, \ldots, d$. This seed solves the $(m, 2)$-problem for $m \geq 2d + 1$. However, this is not the only seed we can derive from R. For any pair s_0, s_1 the seed derived from the string $\mathbf{0}^{s_0} R \mathbf{0}^{s_1}$ also has pairs of don't care symbols at distance δ for $\delta = 1, \ldots, d$. Hence, it solves the $(m, 2)$-problem for a sufficiently large m. Clearly there is a trade-off here: the larger are s_0 and s_1 the higher is the weight of the corresponding seed (a good thing) and the larger is the value m for which the seed solves the $(m, 2)$-problem (a bad thing).

To evaluate to what extent rulers are useful for seed design it is clearly necessary to investigate this trade-off. In this section we give upper bounds to the minimum m for which the seed associated to the string $\mathbf{0}^{s_0} R \mathbf{0}^{s_1}$ solves the $(m, 2)$-problem. The results of this section are valid for any complete d-ruler R. However, since seeds of higher weight are preferable, it is natural to derive seeds from rulers with the minimum number of **1**'s, that is, from *perfect rulers*.

Since the main object of our study are rulers, for simplicity we will only work with strings over the alphabet $\{\mathbf{0}, \mathbf{1}\}$, with the *implicit* associations $\mathbf{0} \to$ '#', $\mathbf{1} \to$ '-'. We introduce Definition 4 and Lemma 1 that essentially restate known properties of seeds in the language of strings over the alphabet $\{\mathbf{0}, \mathbf{1}\}$.

Definition 4 (Completeness). *A binary string P is (m, k)-complete if, for any length-m binary string V containing exactly k **1**'s, there exists at least an index t, with $0 \leq t \leq |V| - |P|$, such that for $i = 0, \ldots, |V| - 1$, it is*

$$V[i] = \mathbf{1} \implies (i - t < 0) \lor (i - t \geq |P|) \lor (P[i - t] = \mathbf{1}). \tag{2}$$

If (2) holds we say that $P + t$ matches in V, or that P shifted by t matches in V. □

Note that $P + t$ matches in V if the **1**'s in V are either outside $P + t$ or correspond to a **1** in $P + t$. Equivalently, there is no **1** in V corresponding to a **0** in $P + t$.

Lemma 1. *The binary string P is (m, k)-complete if and only if the spaced seed obtained with the map $\mathbf{0} \to$ '#', $\mathbf{1} \to$ '-' solves the (m, k)-problem.* □

Having stated Lemma 1, in the rest of the paper most of the results will simply establish that certain binary strings are, or are not, (m, k)-complete, without even mentioning the immediate consequence that the corresponding seeds solve, or do not solve, the (m, k)-problem.

Definition 5 (Minimum length m^*_P). *Given a binary string P we denote by m^*_P the smallest integer m such that P is $(m, 2)$-complete.*[1] □

The following theorem provides an upper bound for m^*_P for $P = \mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ when R is a complete d-ruler with ℓ $\mathbf{1}$'s, and $\max(s_0, s_1) \leq d$. Since the seed associated to $\mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ has weight $s_0 + s_1 + d + 1 - \ell$ the theorem establishes a trade-off between seed weight and minimum pattern length m^*_P.

Theorem 1. *Let $P = \mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ where R is a complete d-ruler. If $\max(s_0, s_1) \leq d$, then $m^*_P \leq 2|P| - 1 - \min(s_0, s_1)$.*

Proof. To prove the theorem we show that P is $(m, 2)$ complete for $m = 2|P| - 1 - \min(s_0, s_1)$. Without loss of generality we assume that $s_0 \geq s_1$ (if this is not the case consider P^R, i.e. the string P reversed).

Let V any length-m binary string containing exactly two ones, in the positions v_1, v_2 ($0 \leq v_1 < v_2 \leq m - 1$). We need to show that for any such pair v_1, v_2 we can find a shift t, $0 \leq t \leq m - |P|$, such that $P + t$ matches in V. By construction we have $|P| = d + 1 + s_0 + s_1$ and $m = 2(d + 1 + s_0 + s_1) - 1 - s_1 = 2d + 2s_0 + s_1 + 1$. Hence the admissible range for t is $0 \leq t \leq m - |P| = d + s_0$.

Note that in our setting the condition in Definition 4 is equivalent to require that there exists a shift t, $0 \leq t \leq d + s_0$ such that for $i = 1, 2$

$$(v_i < t) \;\vee\; (v_i \geq t + |P|) \;\vee\; (P[v_i - t] = \mathbf{1}) \qquad (3)$$

The proof is based on techniques similar to the ones used in [6, Sect. 4.3]. We say that a binary string C of length $(n + 1)$ is a complete *cyclic n-ruler* if for all δ, $0 < \delta \leq n$, there exist $0 \leq i, j \leq n$ such that $C[i] = C[j] = \mathbf{1}$ and $\delta \equiv i - j \bmod n + 1$. We say that C *measures δ cyclically*. Clearly, any complete d-ruler is a complete cyclic d-ruler.

We now show that, if R is a complete d-ruler and $s_0 \leq d$, then $C = \mathbf{0}^{s_0} R$ is a complete cyclic $(d + s_0)$-ruler. Indeed, C measures all integers δ, $0 < \delta \leq d$, since it contains R. Now let $d + 1 \leq \delta \leq d + s_0$; $\delta' = d + s_0 + 1 - \delta$ is such that $1 \leq \delta' \leq s_0 \leq d$ so it is measurable in R. Let (i, j) be a measure of δ' in R, we show that $(j + s_0, i + s_0)$ is a cyclic measure of δ in C. Indeed, by construction $i - j = \delta'$ so

$$j + s_0 - (i + s_0) = j - i = -\delta' = \delta - (d + s_0 + 1).$$

[1] m^*_P also depends on k, but since in this paper we treat uniquely the case $k = 2$, k does not appear in m^*_P to make the notation less cumbersome.

Table 2. Comparison of seed weight as a function of pattern length. The second row reports the weights of the asymptotically optimal seeds defined in [4, Th. 5]. The third row reports the maximal weights for the seeds of the form $\mathbf{0}^s R_d \mathbf{0}^s$ (the last row shows the values of d and s yielding the maximal weights). The weights of the two families are both equal to 282 for $m = 498$.

m	16	32	48	64	80	96	200	300	400	500
[4]	1	3	10	12	21	30	88	150	210	284
Th. 1	7	14	23	31	41	50	109	166	224	283
(d, s)	(3,3)	(8,5)	(10,9)	(15,11)	(17,15)	(19,19)	(41,39)	(61,59)	(81,79)	(101,99)

Hence, $j + s_0 - (i + s_0) \equiv \delta \bmod (d + s_0 + 1)$. This proves that C is a complete cyclic $(d + s_0)$-ruler.

Now consider a length-m binary string V as above and let $m' = d + s_0 + 1$. For $i = 1, 2$, let $v'_i = v_i \bmod m'$. Let V' be the length-m' binary string with $V'[v'_i] = \mathbf{1}$ (for $i = 1, 2$) and $V'[j] = \mathbf{0}$ for all other j. Informally, V' is obtained as a projection of V modulo m'. Since $|v'_1 - v'_2| \leq m' - 1 = d + s_0$, and C is a complete cyclic $(d + s_0)$-ruler, there exists ρ, $0 \leq \rho \leq s_0 + d$, such that

$$C[(v'_i - \rho) \bmod m'] = \mathbf{1} \qquad \text{for } i = 1, 2. \tag{4}$$

We claim that for $t = \rho$, $P + t$ matches in V. First notice that $t = \rho$ is in the correct range, and that, the assumption $s_1 \leq s_0$ implies $P[j] = C[j \bmod m']$ for all j, $0 \leq j < |P|$. To prove that (3) holds for $i = 1, 2$ observe that if $v_i - \rho < 0$ or $v_i - \rho \geq |P|$ the proof is trivial. If $0 \leq v_i - \rho < |P|$, recalling that $v'_i = v_i \bmod m'$, we have

$$P[v_i - \rho] = C[(v_i - \rho) \bmod m'] = C[(v'_i - \rho) \bmod m'] = \mathbf{1}.$$

as claimed. □

As an immediate application of Theorem 1, for different pattern lengths m we computed the seed $\mathbf{0}^s R_d \, \mathbf{0}^s$ of maximal weight among those for which Theorem 1 guarantees $m^*_P \leq m$. The resulting maximal weights for some values of m are reported in Table 2, together with the weights of the asymptotically optimal seeds from [4]. We see that for m up to 498, seeds derived from perfect rulers have a larger weight. Hence, although such seeds are not asymptotically optimal, they are preferable for values of m which are of practical interest.

Theorem 1 assumes $\max(s_0, s_1) \leq d$. It turns out that if $\max(s_0, s_1) > d$ then $P = \mathbf{0}^{s_0} R \, \mathbf{0}^{s_1}$ has a much larger minimal pattern length m^*_P. The following theorem proves that this is true even replacing R with an arbitrary binary string of length $d + 1$.

Theorem 2. *Let $P = \mathbf{0}^{s_0} U \mathbf{0}^{s_1}$, where U is any binary string of length $d + 1$. Then:*

$$\max(s_0, s_1) > d \implies m^*_P \geq 2|P| \tag{5}$$

$$\min(s_0, s_1) > d \implies m^*_P \geq 2|P| + \min(s_0, s_1). \tag{6}$$

Proof. Without loss of generality we can assume $s_0 \geq s_1$. Let $p = |P|$. To prove (5) we show that if $\max(s_0, s_1) > d$ then P is not $(2p-1, 2)$-complete according to Definition 4. Let V denote the binary string of length $2p-1$ with $\mathbf{1}$'s in positions $v_1 = p-1-(d+1)$ and $v_2 = p-1$. Since P does not measure $d+1$, $P+t$ can match V only if $v_1 - t < 0$ which implies $t > p-1-(d+1) = v_2-(d+1)$. Moreover, since $t \leq |V| - |P|$ must hold, then it must be $t \leq p-1$. Hence, we must have $0 \leq v_2 - t < (d+1) \leq s_0$. This latter inequality implies $P[v_2 - t] = \mathbf{0}$ and (2) cannot hold. To prove (6), we show that if $\min(s_0, s_1) > d$ then P cannot be $(2p-1+\min(s_0, s_1), 2)$-complete by taking $v_1 = p-1$ and $v_2 = p + s_1 - 1$ and reasoning as above. □

Theorem 2 implies that the seed $\mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ is not interesting when $\max(s_0, s_1) > d$. To see this, compare for example $P = \mathbf{0}^{d+1} R\, \mathbf{0}^{d}$ with $P' = \mathbf{0}^{d+1} R'\, \mathbf{0}^{d+1}$ where R' is a complete $(d+1)$-ruler. We have $|P| = 3d+2$, $|P'| = 3d+4$, and P' has at least one $\mathbf{0}$ more than P. By Theorem 2 it is $m^*_P \geq 6d+4$ whereas by Theorem 1 it is $m^*_{P'} \leq 5d+6$ which is preferable for $d > 2$.

Summing up, we have that among the seeds that we can derive from perfect rulers there is a family whose members are of practical interest since they offer a competitive trade-off between seed weight and minimum pattern length m^*_P. In the next sections we will further investigate the properties of these seeds. Since the upper bound established in Theorem 1 will play an important role in our analysis, we introduce a notation for it.

Definition 6 (Upper bound m_P). *For any string $P = \mathbf{0}^{s_0} U \mathbf{0}^{s_1}$, we denote by m_P the value $m_P = 2|P| - 1 - \min(s_0, s_1)$.* □

4 Lower Bounds on the Minimum Pattern Length m^*_P

In this section we investigate whether the upper bound established in Theorem 1 is tight or there exist seeds of the form $\mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ which are $(m, 2)$-complete for m significantly smaller than the upper bound of Theorem 1.

We begin our analysis with the case of the seed associated to $\mathbf{0}^d R\, \mathbf{0}^d$ where R is a complete d-ruler. This is an important case since in Section 3 we saw that the only interesting seeds are those with $\min(s_0, s_1) \leq d$. Among them, $\mathbf{0}^d R \mathbf{0}^d$ is the one with the largest weight. For $P = \mathbf{0}^d R \mathbf{0}^d$ Theorem 1 yields $m^*_P \leq 5d+1$. The next result shows that it is indeed $m^*_P = 5d+1$ and that such value is the best possible even if we replace R with an arbitrary binary string of length $d+1$.

Lemma 2. *Let $P = \mathbf{0}^d U \mathbf{0}^d$, where U is any binary string of length $d+1$. Then $m^*_P \geq 5d+1$.*

Proof (sketch). We prove that P is not $(5d, 2)$-complete reasoning as in the proof of Theorem 2 taking $v_1 = d-1$ and $v_2 = 4d$. □

Corollary 1. *If $P = \mathbf{0}^d R \mathbf{0}^d$, where R is a complete d-ruler, then $m^*_P = 5d+1$.* □

For the general case $P = \mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ we provide lower bounds which depend on the distributions of the **1**'s in R. We make use of the following technical lemma.

Lemma 3. *Assume $P = \mathbf{0}^{s_0} U \mathbf{0}^{s_1}$ is $(m, 2)$-complete where U is an arbitrary binary string. For any δ that is measurable in U, let $(x, x + \delta)$ and $(y, y + \delta)$ denote respectively the leftmost and rightmost measures of δ in P. We have*

$$\delta \le s_0 \implies x + \delta \le m - |P| \tag{7}$$

$$\delta \le s_1 \implies |P| - 1 - y \le m - |P|. \tag{8}$$

A fundamental notion in our analysis is the one of (λ, σ)-skewness. Informally, a string is (λ, σ)-skew for small values of λ and σ if there are small integers that are measured only near the endpoints of the string. This property implies that the minimum length m^*_P is close to the upper bound m_P (see Theorem 3).

Definition 7 (Skewness). *Let U denote a binary string of length u. We say that U is (λ, σ)-skew if there exist δ_L and δ_R, not necessarily distinct, such that $(u_L, u_L + \delta_L)$ (resp. $(u_R, u_R + \delta_r)$) is the only measure of δ_L (resp. δ_R) in U, and the conditions*

$$\max(u_L, u - 1 - u_R - \delta_R) \le \lambda \qquad \textit{and} \qquad \max(\delta_L, \delta_R) \le \sigma \tag{9}$$

hold. □

Note that (9) implies that the range $(u_L, u_L + \delta_L)$ is entirely within the first $\lambda + \sigma$ positions of U and starts within the first λ positions of U. Symmetrically, the range $(u_R, u_R + \delta_R)$ is within the last $\lambda + \sigma$ positions of U and ends within the last λ positions of U.

Example 1. Let $U = \mathbf{1100110000111}$. The only measure of $\delta_L = 3$ is $(1, 4)$, and the only measure of $\delta_R = 2$ is $(10, 12)$. Since $|U| = 13$, U is $(1, 3)$-skew. □

Let $P = \mathbf{0}^{s_0} U \mathbf{0}^{s_1}$ where U is (λ, σ)-skew. The next theorem establishes a lower bound for m^*_P, given in terms of the upper bound m_P, under the assumption that $\min(s_0, s_1)$ is larger than the integers that are measured only near the endpoints of U. The assumption is not restrictive for practical applications, since more **0**'s in P translates to a spaced seed with larger weight.

Theorem 3. *For any string U, let $P = \mathbf{0}^{s_0} U \mathbf{0}^{s_1}$. If U is (λ, σ)-skew and $\min(s_0, s_1) \ge \sigma$, then $m^*_P \ge m_P - \lambda$.*

Proof. We prove the theorem showing that, if P is $(m, 2)$-complete with $m \le m_P - \lambda$, then necessarily $m \ge m_P - \lambda$. Let $p = |P|$, $u = |U|$.

Since U is (λ, σ)-skew, for δ_L and δ_R as in Definition 7 it is $\delta_L, \delta_R \le \sigma \le \min(s_0, s_1)$. Hence, Lemma 3 can be applied to both δ_L and δ_R.

Let, as in Definition 7, $(s_0 + u_L, s_0 + u_L + \delta_L)$ be the unique measure of δ_L in P. By (8) of Lemma 3 and (9) it is

$$m \ge 2p - 1 - (s_0 + u_L) \ge 2p - 1 - s_0 - \lambda. \tag{10}$$

Similarly, let $(s_0+u_R, s_0+u_R+\delta_R)$ be the unique measure of δ_R in P. By (7) of Lemma 3, it is $s_0+u_R+\delta_R \leq m-p$. Applying (9) and finally recalling that $p = u+s_0+s_1$, we get

$$m \geq p+s_0+u_R+\delta_R \geq p+s_0+u-1-\lambda = 2p-1-s_1-\lambda. \quad (11)$$

From (10) and (11) we get, as claimed, $m \geq 2p-1-\min(s_0,s_1)-\lambda = m_P-\lambda$. □

The above theorem holds for any (λ,σ)-skew string U. For a complete d-ruler, combined with Theorem 1, it yields the following result.

Corollary 2. *Let $P = \mathbf{0}^{s_0} R\, \mathbf{0}^{s_1}$ where R is a (λ,σ)-skew complete d-ruler. If $\max(s_0,s_1) \leq d$ and $\min(s_0,s_1) \geq \sigma$, then $m_P-\lambda \leq m_P^* \leq m_P$.* □

5 Wichmann Rulers

As we mentioned in Section 3, perfect rulers are difficult to find. The exception is the family of Wichmann rulers [15] which have a sort of "generating function". The Wichmann ruler $W_{r,s}$ is the binary string defined by

$$W_{r,s} = \mathbf{1}^{r+1}\, \mathbf{0}^r \mathbf{1}\, (\mathbf{0}^{2r}\mathbf{1})^r\, (\mathbf{0}^{4r+2}\mathbf{1})^s\, (\mathbf{0}^{2r+1}\mathbf{1})^{r+1}\, \mathbf{1}^r.$$

$W_{r,s}$ has length $w_{r,s} = 4(r+1)^2 + s(4r+3)$ and contains exactly $4r+s+3$ $\mathbf{1}$'s. It is a classical result that $W_{r,s}$ is a complete $(w_{r,s}-1)$-ruler [15].

In this section we consider the seeds of the form $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ and we analyze how tight is Theorem 1 for these seeds. Our first result proves that, if the number of leading and trailing $\mathbf{0}$'s in P is large enough, then the upper bound m_P of Theorem 1 is very accurate. Recall that large s_0 and s_1 are required to obtain seeds with large weight.

Theorem 4. *Let $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ with $r > 1$. We have*

$$\min(s_0,s_1) \geq 3r+1 \implies m_P^* \geq m_P - 1 \quad (12)$$
$$\min(s_0,s_1) \geq 2r+3 \implies m_P^* \geq m_P - r + 1 \quad (13)$$

Proof (sketch). We show that $W_{r,s}$ is $(1, 3r+1)$-skew and use Theorem 3 to get (12). Similarly, we show that $W_{r,s}$ is $(r-1, 2r+3)$-skew to obtain (13). □

We now establish lower bounds for $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ with no constraints on s_0 and s_1. These results are not based on the concept of skewness, but on the specific structure of Wichmann rulers.

Lemma 4. *Let $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ with $r > 1$. If $\max(s_0,s_1) \geq 2r+3$ then $m_P^* \geq m_P - (2r+2)$.*

Proof (sketch). We prove the lemma assuming that $m_P^* \leq m_P - (2r+3)$, and obtaining a contradiction. If $s_1 = \min(s_0,s_1)$, we apply Lemma 3 to $\delta = 2r+3 \leq s_0$. By (7) we get the contradiction $m_P^* \geq m_P + 1 - r$. If $s_0 = \min(s_0,s_1)$, we apply Lemma 3 to $\delta = r+2 \leq s_1$ and we get the same contradiction by (8). □

Lemma 5. *Let* $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ *with* $r > 0$. *If* $\max(s_0, s_1) \leq 2r + 2$ *then* $m^*_P \geq m_P - (4r + 2)$.

Proof (sketch). Let $p = |P|$, and consider the binary string V of length m^*_P with $\mathbf{1}$'s in positions $v_1 = p - s_1 - r - 2$ and $v_2 = v_1 + 1$. The closest pair of consecutive $\mathbf{1}$'s in P to the left of (v_1, v_2) are those in positions $(s_0 + r - 1, s_0 + r)$. Hence $P + t$ matches in V only for $t \geq v_1 - (s_0 + r - 1)$. Since the largest admissible shift t is $m^*_P - p$ we must have $m^*_P - p \geq v_1 - (s_0 + r - 1)$, which implies the thesis. □

Combining Lemmas 4 and 5 we get the following theorem that provides a lower bound for any seed P based on a Wichmann ruler with $r > 1$.

Theorem 5. *Let* $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ *with* $r > 1$. *It is* $m^*_P \geq m_P - (4r + 2)$. □

Note that $r = O(\sqrt{w_{r,s}})$, so Theorem 5 once more proves the estimate of Theorem 1 accurate. In the following example we present a specific case, in order to give a feeling of the values involved.

Example 2. The string $W_{2,1}$ is a complete 46-ruler. Consider the seed $P = \mathbf{0}^i W_{2,1} \mathbf{0}^i$. For $i =, 0, \ldots, 46$ it is $m_P = 94 + 3i - 1$. By Corollary 1, for $i = 46$ it is $m^*_P = m_P = 231$. By Theorem 4, for $7 \leq i \leq 45$ it is $m_P - 1 \leq m^*_P \leq m_P$. For example for $i = 7$ it is $113 \leq m^*_P \leq 114$. By Theorem 5, for $0 \leq i \leq 6$, it is $m_P - 10 \leq m^*_P \leq m_P$. For example, for $i = 0$ it is $83 \leq m^*_P \leq 93$. □

6 Restricted vs. Unrestricted Rulers

The complete rulers defined in Section 2 are sometimes called *restricted* rulers since we require that the string R measuring all integers between 1 and d has length exactly $d+1$. In the literature [3,7] there is also the notion of *unrestricted d-ruler* which is a binary string of arbitrary length that measures all integers between 1 and d. The next Lemma shows that every spaced seed must have an unrestricted ruler at its heart.

Lemma 6. *If* P *is* $(m, 2)$*-complete, then it must measure any integer* δ, $1 \leq \delta \leq 2|P| - m - 1$.

Proof. Let $p = |P|$. We prove the lemma showing that if P does not measure δ then $\delta \geq 2p - m$. Consider the length-m binary string V with $\mathbf{1}$'s in positions $v_1 = p - 1 - \delta$, and $v_2 = p - 1$. Since P is $(m, 2)$-complete there must exist $t \leq m - p$ such that $P + t$ matches in V according to Definition 4. However, if P does not measure δ, $P + t$ does not match in V whenever $t \leq v_1$. Hence, we must have $t \geq v_1 + 1 = p - \delta$, which is possible only if $p - \delta \leq m - p$ which implies $\delta \geq 2p - m$ as claimed. □

The above result suggests that it could be worthwhile to analyze also spaced seeds of the form $P = \mathbf{0}^{s_0} U_d \mathbf{0}^{s_1}$, where U_d is an unrestricted d-ruler. This interest is motivated theoretically by the fact that an unrestricted d-ruler can contain less $\mathbf{1}$'s than a perfect d-ruler.

Example 3. The minimum number of $\mathbf{1}$'s in a restricted 18-ruler is eight [10]. The string $U_{18} = \mathbf{1000001001100000010000101}$ has length 25, seven $\mathbf{1}$'s, and is an unrestricted 18-ruler since it measures all integers from 1 to 18. □

In the following we give some evidence that for seed design unrestricted rulers appear to be less effective than restricted rulers.

Let U_{18} denote the string defined in Example 3 and let m_Q^* denote the minimum m such that $Q = \mathbf{0}^s U_{18} \mathbf{0}^s$ is $(m,2)$-complete. The ruler U_{18} is $(0,6)$-skew, since 6 has the unique measure $(0,6)$, and 2 has the unique measure $(22,24)$. Theorem 3 implies that $m_Q^* \geq m_Q = 2|Q|-1-s$. Note, however, that Theorem 1 has been proven only for restricted rulers. Indeed, a direct verification shows that it does not hold for unrestricted ones, since for $s \geq 13$ it is $m_Q^* > m_Q$.

Based on the above observations, we compare the completeness properties of a seed obtained from U_{18} to one obtained from a restricted d-ruler R_d.

- Let $P = \mathbf{0}^{s_0} R_{18} \mathbf{0}^{s_1}$, where R_{18} is a restricted 18-ruler with eight $\mathbf{1}$'s (see [10]). Let $s_0 = s+4$ and $s_1 = s+3$. Then $|P| = |Q|+1$ and P and Q have the same weight. Since $m_P = 2(|Q|+1)-1-(s+3) = m_Q - 1$, then $m_Q^* \geq m_Q = m_P + 1 > m_P^*$, which implies $m_Q^* > m_P^*$.
- If we take $P = \mathbf{0}^{s+4} R_{18} \mathbf{0}^{s+4}$ with the same R_{18}, so that P is longer and has a larger weight than Q, we obtain $m_Q = m_P$ and therefore $m_Q^* \geq m_P^*$.
- Finally, let $P = \mathbf{0}^{s_0} R_{17} \mathbf{0}^{s_1}$ where R_{17} is a restricted 17-ruler with seven $\mathbf{1}$'s as U_{18} (see [10]). Let $s_0 = s+4$ and $s_1 = s+3$. Then, Q and P have the same length and weight, and $m_Q = m_P + 3$, therefore $m_Q^* \geq m_P^* + 3$.

We consider now Wichmann rulers. We observe that in general a restricted d'-ruler with $d' > d$ is an unrestricted d-ruler. Hence, for $b > 0$, $W_{r+b,s}$ can be seen as an unrestricted $(w_{r,s}-1)$-ruler. We show that a seed built out of the restricted ruler $W_{r,s}$ is better than one of the same length based on $W_{r+b,s}$.

Let $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$. To preserve length, in replacing $W_{r,s}$ with $W_{r+b,s}$ we reduce the number of leading and trailing zeroes. Since $|W_{r+b,s}| - |W_{r,s}| = 4b(b+2r+s+2)$, we let $\sigma_b = 2b(b+2r+s+2)$ and define $Q = \mathbf{0}^{s'_0} W_{r+b,s} \mathbf{0}^{s'_1}$ with $s'_i = s_i - \sigma_b$ $(i=0,1)$, so that $|Q| = |P|$. Notice that the following theorem holds, somewhat counterintuitively, even though P has a larger weight than Q.

Theorem 6. *Let $P = \mathbf{0}^{s_0} W_{r,s} \mathbf{0}^{s_1}$ and $Q = \mathbf{0}^{s'_0} W_{r+b,s} \mathbf{0}^{s'_1}$ as above, with $r > 1$. It is $m_Q^* \geq m_P^*$ for $b \geq 1$, and $m_Q^* > m_P^*$ for $b > 1$ or $s > 0$.*

Proof. Since $|Q| = |P|$, it is $m_Q = 2|Q|-1-\min(s_0-\sigma_b, s_1-\sigma_b) = m_P+\sigma_b$. From Theorem 5 applied to Q and the latter equality we get $m_Q^* \geq m_P+\sigma_b-4(r+b)-2$; the inequality is strict for $b > 1$ or $s > 0$ as claimed. □

7 Conclusions

We have proposed a new family of lossless seeds built using perfect rulers. We have proven upper and lower bounds on the effectiveness of these seeds and shown that they are of practical interest. A natural extension of this work is to study the use of our new family of seeds in the context of multiseed filtration for the detection of more than two mismatches.

Acknowledgements. We thank the anonymous referees for their useful comments. Some of them need a broader treatment than what is allowed by the space and time allotted for this conference version. We are indebted to one of the referees for the proof of Theorem 1 (our original, much less elegant, proof is in [2]).

References

1. Burkhardt, S., Kärkkäinen, J.: Better filtering with gapped q-grams. Fundam. Inform. 56(1-2), 51–70 (2003)
2. Egidi, L., Manzini, G.: Spaced seeds design using perfect rulers. Technical Report TR-INF-2011-06-01-UNIPMN, Computer Science Department, UPO (2011), http://www.di.unipmn.it
3. Erdós, P., Gál, I.S.: On the representation of $1, 2, \ldots, n$ by differences. Indagationes Math. 10, 379–382 (1948)
4. Farach-Colton, M., Landau, G.M., Sahinalp, S.C., Tsur, D.: Optimal spaced seeds for faster approximate string matching. J. Comput. Syst. Sci. 73(7), 1035–1044 (2007)
5. Keich, U., Li, M., Ma, B., Tromp, J.: On spaced seeds for similarity search. Discrete Applied Mathematics 138(3), 253–263 (2004)
6. Kucherov, G., Noé, L., Roytberg, M.A.: Multiseed lossless filtration. IEEE/ACM Trans. Comput. Biology Bioinform. 2(1), 51–61 (2005)
7. Leech, J.: On the representation of $1, 2, \ldots, n$ by differences. J. London Math. Soc. 31, 160–169 (1956)
8. Li, M., Ma, B., Kisman, D., Tromp, J.: Patternhunter II: Highly sensitive and fast homology search. J. Bioinformatics and Computational Biology 2(3), 417–440 (2004)
9. Lin, H., Zhang, Z., Zhang, M.Q., Ma, B., Li, M.: Zoom! zillions of oligos mapped. Bioinformatics 24(21), 2431–2437 (2008)
10. Luschny, P.: Perfect and optimal rulers (2003), http://www.luschny.de/math/rulers/prulers.html
11. Ma, B., Li, M.: On the complexity of the spaced seeds. J. Comput. Syst. Sci. 73(7), 1024–1034 (2007)
12. Ma, B., Tromp, J., Li, M.: Patternhunter: faster and more sensitive homology search. Bioinformatics 18(3), 440–445 (2002)
13. Ma, B., Yao, H.: Seed optimization is no easier than optimal Golomb ruler design. In: Brazma, A., Miyano, S., Akutsu, T. (eds.) APBC. Advances in Bioinformatics and Computational Biology, vol. 6, pp. 133–144. Imperial College Press, London (2008)
14. Nicolas, F., Rivals, E.: Hardness of optimal spaced seed design. J. Comput. Syst. Sci. 74(5), 831–849 (2008)
15. Wichmann, B.: A note on restricted difference bases. J. London Math. Soc. 38, 465–466 (1962)

Weighted Shortest Common Supersequence

Amihood Amir[1,⋆], Zvi Gotthilf[2], and B. Riva Shalom[3]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel and Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218
amir@cs.biu.ac.il

[2] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
gotthiz@cs.biu.ac.il

[3] Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel
riva.shalom@gmail.com

Abstract. The Shortest Common Supersequence (SCS) is the problem of seeking a shortest possible sequence that contains each of the input sequences as a subsequence. In this paper we consider applying the problem to *Position Weight Matrices* (PWM). The Position Weight Matrix was introduced as a tool to handle a set of sequences that are not identical, yet, have many local similarities. Such a weighted sequence is a 'statistical image' of this set where we are given the probability of every symbol's occurrence at every text location. We consider two possible definitions of SCS on PWM. For the first, we give a polynomial time algorithm, having two input sequences. For the second, we prove $\mathcal{NP}$-hardness.

1 Introduction

The *Shortest Common Supersequence* problem (*SCS*) is a well studied problem [10,11,14,15]. It is known as $\mathcal{NP}$-hard [11] even in the case of a binary alphabet [13]. However, if the number of input strings is fixed, their *SCS* can be found in polynomial running time [14,15].

In the SCS problem we are given a set of L strings and look for shortest possible string obtaining all input strings as subsequences. For example, let L=2, $A = addab$ and $B = ebadea$, a possible supersequence is *ebaddeab*.

1.1 Motivation

One of the applications that motivate our problem is *Planning research*. Given a set of goals (or tasks) which have to be accomplished, one needs to find the most cost efficient plan that achieves all the goals.

As mentioned in [5], for the AI planning research it is important to exploit the interactions between different parts of plans. *Merging* different actions in order to make the total plan more efficient can be viewed as a supersequence. Similar applications can be found in the area of Computational Biology.

⋆ Partly supported by NSF grant CCR-09-04581 and ISF grant 347/09.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 44–54, 2011.
© Springer-Verlag Berlin Heidelberg 2011

1.2 Weighted Sequences

This paper considers a more general version of the SCS problem, one where the input consists of *weighted sequences*. A weighted sequence is defined as a sequence $S = (s_1, v_1), ..., (s_{|S|}, v_{|S|})$ where $s_i \in \Sigma$, $v_i \in \mathbb{R}$, $i = 1..|S|$. For comparison of two weighted sequences a function $W : |S_1| \times |S_2| \rightarrow \mathbb{R}$ is defined. W assigns a value to every possible match between two characters, one from the first sequence and the other from the second sequence. The SCS variant for these weighted sequences aims at both maximizing the weight of the common supersequence and minimizing the length, rather than merely minimizing its length. This will be formally defined below.

A possible model of weighted sequences is such that, at each position of the sequence, any alphabet symbol can occur with a certain probability. To prevent ambiguity, we refer to such sequences as *p-weighted sequences*, though in the literature they are both named weighted sequences.

Definition 1. *([9]) A* p-weighted sequence $A = a_1..a_n$ over finite alphabet Σ, *is a sequence of sets* a_i, $1 \leq i \leq n$. *Every* a_i *is a set of pairs* $(s_j, \pi_i(s_j))$, *where* $s_j \in \Sigma$ *and* $\pi_i(s_j)$ *is the probability of having symbol* s_j *at location* i.
Formally, $a_i = \{(s_j, \pi_i(s_j)) \mid s_j \neq s_l$ *for* $j \neq l$, *and* $\sum_j \pi_i(s_j) = 1\}$.

The concept of p-weighted sequences was introduced as a tool for motif discovery and local alignment. A p-weighted sequence is called in the biological literature a "*Position Weight Matrix*" (PWM) [12]. A p-weighted sequence of length m is a $|\Sigma| \times m$ matrix that reports the frequency of each symbol from alphabet Σ for every possible location. Note that as each location of the input may contains all $\sigma \in \Sigma$ (and their probability of appearing there), we refer to the inputs as **sequences** rather than as strings. We also denote a location of the p-weighted sequence a **character** that contains several **symbols**.

The first use of PWM sequences was for relatively short sequences, for example binding sites or sequences resulting from multiple alignment. Iliopoulos et al. [9] considered building very large Position Weight Matrices that correspond, for example, to complete chromosome sequences that have been obtained using a whole-genome shotgun strategy [16]. By keeping all the information that the whole-genome shotgun produces, it is possible to ferret out information that has been previously undetected after being faded during the consensus step. This concept is true for other applications where local similarities are thus encoded. Therefore, the necessity of developing adequate algorithms for p-weighted sequences increases.

The *Longest Common Subsequence* of p-weighted strings was defined by Amir et al. [2] and named the LCWS problem. We similarly define the *Shortest Common Supersequence* of p-weighted strings. Note that when dealing with p-weighted sequences, every added symbol reduces the total weight. Therefore, we define a lower bound, below which the weight is not allowed to decrease and under this restriction the Shortest Common Supersequence is sought. The bound is set according to the certainty level required by the application.

Since we consider two p-weighted sequences, we differentiate between their probabilities by denoting as π_i^A the probability of occurring at the ith location

of sequence A. Throughout the paper we consider, for simplicity, sequences of the same length. Different lengths sequences can be similarly dealt with. The formal definition appears below.

Definition 2. The Shortest Common Weighted Supersequence Problem (*SCWS*):

Input: *Two p-weighted sequences A, B of length n over alphabet Σ, and a constant α, $0 < \alpha \leq 1$.*

Output: *The minimal ℓ such that there is a common supersequence of length ℓ, $s_1..s_\ell$, $s_i \in \Sigma$, $i = 1, ..., \ell$, where $\sigma_1^a..\sigma_n^a = s_{i_1}..s_{i_n}$ and $\sigma_1^b..\sigma_n^b = s_{j_1}..s_{j_n}$, for increasing indices $\{i_y\}$ and $\{j_y\}$ (σ_y^a stands for an alphabet symbol chosen to represent the yth character of A.)*
The supersequence ordering maintains:

$$\prod_{y=1}^{n}(\pi_y^A(\sigma_y^a) \cdot \pi_y^B(\sigma_y^b)) \geq \alpha.$$

Example: Let $\Sigma = \{a, b, c\}$, $\alpha = 0.1$ and the p-weighted sequences be as in Figure 1. There is no SCS of length 4, since a choice of a or b as the first element would already mean probability $0.1 \times 0.8 = 0.08 < \alpha$. A choice of c is even worse, with probability $0.1 \times 0.1 = 0.01 < \alpha$. However, if we choose $s_1 = a$,$s_2 = b, s_3 = c, s_4 = c, s_5 = b$ with $s_1, s_2, s_3, s_4 = abcc = \sigma_1^a, \sigma_2^a, \sigma_3^a, \sigma_4^a$, and $s_2, s_3, s_4, s_5 = bccb = \sigma_1^b, \sigma_2^b, \sigma_3^b, \sigma_4^b$, then the probability is $0.8 \times 0.8 \times 0.7 \times 0.7 \times 0.9 \times 0.6 \times 1 \times 0.6 = 0.1016064 > \alpha$.

A				
	$\Pi_1^A(a) = 0.8$	$\Pi_2^A(a) = 0.1$	$\Pi_3^A(a) = 0.1$	$\Pi_3^A(a) = 0.3$
	$\Pi_1^A(b) = 0.1$	$\Pi_2^A(b) = 0.7$	$\Pi_3^A(b) = 0$	$\Pi_3^A(b) = 0.1$
	$\Pi_1^A(c) = 0.1$	$\Pi_2^A(c) = 0.2$	$\Pi_3^A(c) = 0.9$	$\Pi_3^A(c) = 0.6$
	a_1	a_2	a_3	a_4

B				
	b_1	b_2	b_3	b_4
	$\Pi_1^B(a) = 0.1$	$\Pi_2^B(a) = 0.1$	$\Pi_3^B(a) = 0$	$\Pi_3^B(a) = 0.3$
	$\Pi_1^B(b) = 0.8$	$\Pi_2^B(b) = 0.2$	$\Pi_3^B(b) = 0$	$\Pi_3^B(b) = 0.6$
	$\Pi_1^B(c) = 0.1$	$\Pi_2^B(c) = 0.7$	$\Pi_3^B(c) = 1$	$\Pi_3^B(c) = 0.1$

$prob(a_i, b_j)$	a_1	a_2	a_3
b_1	4/9	1/3	1/3
b_2	8/27	4/9	4/9
b_3	56/81	7/27	7/27

Fig. 1. The two p-weighted sequences

Though the $SCWS$ problem seems natural for the position weighted matrices input, in case the probabilities of the characters of one input sequence S_1 are far from being uniformly distributed, our decision of symbol selection for the SCWS

may be biased and not reflect a real relation between the weighted sequences. In order to prevent this effect, and obtain informative results we present an additional definition to the $SCWS$ problem, Shortest Common Weighted Supersequence with two thresholds, referred to as $SCWS2$. In the $SCWS2$ problem, a separate probability bound is set for each of the p-weighted sequences. This problem can also be useful, when different biological conditions require measuring the probabilities of each sequence independently.

Definition 3. The Shortest Common Weighted Supersequence 2 ($SCWS2$) Problem*:*

Input: *Two p-weighted strings* A, B *of length* n *over alphabet* Σ, *and constants* α_1, α_2, $0 < \alpha_i \leq 1$.
Output: *The minimal* ℓ *such that there is a common supersequence of length* ℓ, $s_1..s_\ell$ *where* $\sigma_1^a..\sigma_n^a = s_{i_1}..s_{i_n}$ *and* $\sigma_1^b..\sigma_n^b = s_{j_1}..s_{j_n}$, *for increasing indices* $\{i_y\}$ *and* $\{j_y\}$. *The supersequence ordering maintains* $\prod_{y=1}^{n} \pi_y^A(\sigma_y^a) \geq \alpha_1$ *AND* $\prod_{y=1}^{n} \pi_y^B(\sigma_y^b) \geq \alpha_2$.

This paper is organized as follows: Section 2 describes related work. The $SCWS$ problem solution appear in Section 3. We then consider the $SCWS2$ problem and its hardness in Section 4. Section 5 concludes the paper and poses some open questions.

2 Related Work

Many hardness and approximation results regarding different variants of the SCS problem were presented during the last three decades, see e.g. [7,10,11,14,15].

Regarding the p-weighted sequences, Iliopoulos et al. [8] defined the problem of longest common sub*string* of p-weighted sequences, where the common sequence is *consecutive*.

Amir et al. [1] showed some conditions where p-weighted matching problems can be reduced to ordinary pattern matching problems. In their model, the probability of each symbol is fixed, regardless to the location in the sequence, and the text is p-weighted while the pattern is an ordinary string. Both assumptions are not valid for the $SCWS$ problem.

Finally, Amir et al. [2] solved the Longest Common Weighted Subsequence problem. They gave two slightly different definitions of the problem. For the first, they solved the weighted LCS problem of z sequences in time $O(zn^{z+1})$. For the second, they proved $\mathcal{NP}$-hardness and provided an approximation algorithm.

3 Shortest Common Weighted Supersequence

The S(SCS) problem is closely related to the Longest Common Subsequence problem, in which we seek common symbols of both sequences, appearing in the same order in both strings. In fact, The SCS can be solved using the LCS solution.

Observation 1. *$SCS(A,B) = |A| + |B| - LCS(A,B)$ where $SCS(A,B)$, $LCS(A,B)$ are the lengths of the SCS, LCS problems, when applied to strings A, B.*

A proof will be provided in the full version.

Similarly, we might consider using the Longest Common Weighted Subsequence (LCWS) to solve the Shortest Common Weighted Supersequence (SCWS) problem. However, in the weighted case, Observation 1 does not hold, as the LCWS considers probabilities of merely the symbols participating the output, while the SCWS takes into account the probabilities of all n characters of each sequence. For example where $\Sigma = \{0,1\}$, see Figure 2. For the inputs described

A	$\Pi_1^A(0) = 0.2$ $\Pi_1^A(1) = 0.8$	$\Pi_2^A(0) = 0.4$ $\Pi_2^A(1) = 0.6$
	a_1	a_2

	b_1	b_2
B	$\Pi_1^B(0) = 0.6$ $\Pi_1^B(1) = 0.4$	$\Pi_2^B(0) = 0.7$ $\Pi_2^B(1) = 0.3$

Fig. 2. An example for the difference between LCWS and SCWS

in Figure 2, the A sequence has higher probabilities for the $\sigma = 1$, whereas B prefers $\sigma = 0$. Therefore, any decision the LCWS solver makes includes matching a certain a_j with b_i yielding a lower probability than $\pi_i^A(1) \cdot \pi_j^B(0)$. Since each a_i, b_j is to be included in the supersequence, it may be more profitable to ignore at least part of the LCWS matching suggestions, due to the threshold demands.

A possible solution to the SCWS problem, can be selecting at each character of the sequences, its most probable symbol, getting ordinary strings. We then, can apply to these strings a regular SCS solution, like the one suggested in Observation 1, requiring $O(n^2)$ time.

Since the problem demands including all characters of the input strings, no solution can have a higher probability than the suggested procedure. Nevertheless, if the α bound is lower, one can decrease the probability, in case it leads to a shorter supersequence. In order to improve the length, it is necessary to choose a single character and select a different representative symbol for it, in a way that enlarges the LCS of the two strings. However, the option of undoing a certain match the LCS made to obtain two matches of less probable symbols, needs to be considered as well. Clearly a naive check of all the options is inefficient.

We therefore present a dynamic programming algorithm for the $SCWS$ problem. We construct a two dimensional table, where the rows represent the characters of the A sequence, and the columns refer to the characters of sequence B. A character in a p-weighted sequence is a table containing all symbols of Σ and the probability of appearing at that location.

As mentioned above, the core of the SCWS problem is minimizing the length of the supersequence under a weight restriction. Consequentially, we cannot save at the table entries merely the shortest supersequence achieved so far as its probability may, in the future, degrade below α and the potential supersequence would have to be discarded. We therefore save at entry i, j, for every possible length, the highest probability of a common supersequence that can be obtained from $A[1 \ldots i]$ and $B[1 \ldots j]$. We denote the variables containing this information by $p^k_{i,j}$, where k represents the length of the common supersequence. Saving these probabilities, when some $p^k_{i,j}$ is too small, we can compute $p^{k+1}_{i,j}$, which may have an increased probability exceeding α.

As each position in a p-weighted sequence consists of $|\Sigma|$ symbols and their probabilities, when considering the matching of a_i and b_j, as a mean of shortening the supersequence, we compute for each symbol $\sigma \in \Sigma$ the product $\pi^A_i(\sigma) \cdot \pi^B_j(\sigma)$ and select the highest value. We denote the selected value of entry i, j as $best_{i,j}$ and save the symbol yielding this probability. The SCWS may include characters originating in a single sequence as well, we therefore save for each weighted character its most probable symbol at the current location i: sym^A_i.

We fill the dynamic programming table in row-major order. Computing an entry i, j implies computing the most probable common supersequence of $A[1 \ldots i]$ and $B[1 \ldots j]$ of length k. k starts from $\max\{i, j\}$, which means the supersequence is of minimal length. The maximal k for $p^k_{i,j}$ is $i + j$, representing some sort of concatenation.

Considering $p^k_{i,j}$, the correlated supersequence can be constructed by matching the a_i and b_j, selecting their $best_{i,j}$ symbol, and by this extending a smaller supersequence by that symbol. Another option is to append the most probable symbol of a_i, sym^A_i, to the supersequence of $A[1 \ldots i-1]$ and $B[1 \ldots j]$, Adding sym^B_j, the representative of b_j, to the supersequence can be done likewise. The algorithm considers these three options and selects the one with highest probability.

The table initialization is as follows. Column 0, contains the probability of a supersequence containing merely prefixes of the A sequence. That is $p_{i,0} = \prod_{y=1}^{i} sym^A_y$. Row 0, is filled by $p_{0,j} = \prod_{y=1}^{j} sym^B_y$. Lemma 1 formally defines the computation required for filling an entry in the dynamic programming table.

Lemma 1. $SCWS(A[1 \ldots\ldots i], B[1 \ldots j]) =$

$$\{p^k_{i,j}\}^{i+j}_{k=\max\{i,j\}} = \max\{sym^B_j \cdot p^{k-1}_{i,j-1}, \quad sym^A_i \cdot p^{k-1}_{i-1,j}, \quad best_{i,j} \cdot p^{k-1}_{i-1,j-1}\}.$$

An inductive proof will appear in the full version.

We fill the table and then go over $\{p^k_{n,n}\}$ in increasing order of k, and check whether $p^k_{n,n} \geq \alpha$. The length of the shortest common supersequence with probability bounded by α is our output.

Note, that we do not have to compute $p^k_{i,j}$ for all possible k's in the same iteration, as the dependency on adjacent entries holds for each of the $p^k_{i,j}$'s separately. Therefore, The algorithm can fill the table layer after layer. At every step, it computes $p^k_{i,j}$ for a certain k for all possible i, j. Note that the k

		$\pi_1^B(a) = 0.5$ $\pi_1^B(b) = 0.4$ $\pi_1^B(c) = 0.1$ $sym_1^B - 0.5$	$\pi_2^B(a) = 0.3$ $\pi_2^B(b) = 0.2$ $\pi_2^B(c) = 0.5$ $sym_2^B - 0.5$	$\pi_3^B(a) = 0.1$ $\pi_3^B(b) = 0.1$ $\pi_3^B(c) = 0.8$ $sym_3^B - 0.8$	$\pi_4^B(a) = 0.4$ $\pi_4^B(b) = 0.3$ $\pi_4^B(c) = 0.3$ $sym_4^B - 0.4$
	$p^0 = 1$	$p^1 \downarrow$[a].5	$p^2 \downarrow$[ac].25	$p^3 \downarrow$[acc].2	$p^4 \downarrow$[acca].08
$\pi_1^A(a) = 0.2$ $\pi_1^A(b) = 0.4$ $\pi_1^A(c) = 0.4$ $sym_1^A - 0.4$	$p^1 \rightarrow$[b].4	(best - 0.16)[b] $p^1 \searrow$[b].16 $p^2 \rightarrow$[ab].0.2	(best - 0.2)[c] $p^2 \searrow$[ac].1 $p^3 \downarrow$ [abb].1	(best - 0.32)[c] $p^3 \searrow$[acc].08 $p^4 \downarrow$[abbc].08	(best - 0.12)[b] $p^4 \downarrow$[acca].032 $p^5 \rightarrow$.032
$\pi_2^A(a) = 0.5$ $\pi_2^A(b) = 0.1$ $\pi_2^A(c) = 0.4$ $sym_2^A - 0.5$	$p^2 \rightarrow$[ba].2	(best - 0.25)[a] $p^2 \rightarrow$[ba].1 $p^3 \downarrow$[aba].1	(best - 0.2)[c] $p^2 \searrow$[bc].032 $p^3 \rightarrow$[aca].05 $p^4 \rightarrow$[abba].05	(best - 0.32)[c] $p^3 \searrow$[acc].0256 $p^4 \downarrow$[acac].04 $p^5 \downarrow$ [abbac].04	(best - 0.2)[a] $p^4 \searrow$[acca].016 $p^5 \searrow$.016 $p^6 \rightarrow$.016
$\pi_3^A(a) = 0$ $\pi_3^A(b) = 0.9$ $\pi_3^A(c) = 0.1$ $sym_3^A - 0.9$	$p^3 \rightarrow$[bab].18	(best - 0.36)[b] $p^3 \rightarrow$[bab].09 $p^4 \downarrow$[baba].09	(best - 0.18)[b] $p^3 \rightarrow$[bcb].0288 $p^4 \rightarrow$[acab].045 $p^5 \downarrow$[babac].045	(best - 0.09)[b] $p^3 \searrow$[bcb].0029 $p^4 \downarrow$[bcbc].023 $p^5 \rightarrow$.036 $p^6 \rightarrow$.036	(best - 0.27)[b] $p^4 \searrow$[accb].0069 $p^5 \rightarrow$.0144 $p^6 \rightarrow$.0144 $p^7 \downarrow$.0144
$\pi_4^A(a) = 0.6$ $\pi_4^A(b) = 0.1$ $\pi_4^A(c) = 0.3$ $sym_4^A - 0.6$	$p^4 \rightarrow$[baba].108	(best - 0.3)[a] $p^4 \rightarrow$[abab].054 $p^5 \downarrow$[ababa].054	(best - 0.18) [a] $p^4 \rightarrow$[bcba].0173 $p^5 \rightarrow$[acaba].027 $p^6 \rightarrow$[babaca].027	(best - 0.24)[c] $p^4 \searrow$[bcbc].0069 $p^5 \rightarrow$.0138 $p^6 \downarrow$.0216 $p^7 \rightarrow$.0216	(best - 0.24)[a] $p^4 \searrow$[bcba].0007 $p^5 \searrow$.0055 $p^6 \searrow$.0086 $p^7 \rightarrow$.0086 $p^8 \rightarrow$.0086

Fig. 3. A SCWS Table

parameter differs as the i, j increase, as the minimal supersequence of growing prefixes increases. This computation is based on the known values surrounding $p_{i,j}^k$, according to Lemma 1.

At the end of the yth iteration, we check whether the current $p_{n,n}^k \geq \alpha$, where $k = n + y - 1$. In case the inequality is valid we consider $n + y - 1$ as the answer, we have just found a common supersequence of this length with a proper probability. There may be another supersequence with even better probability, nevertheless, we are interested in the shortest one.

If the contrary holds and $p_{n,n}^{n+y-1} < \alpha$ we know, due to Lemma 1, that there is no supersequence of such length, satisfying the weight demands. Yet, we compute $p_{i,j}^{k+1}$ for the table, in the hope of attaining a higher probability for a supersequence, in case undoing a single match and obtaining higher individual probabilities of the characters, as was shown in Figure 2. In addition, we discard all $p_{i,j}^{k-1}$'s, as their information is useless from now on.

An example of a SCWS table where $\alpha = 0.005$ appears in Fig. 3. Arrows attached to each $p_{i,j}^k$ enable us to backtrack and form the supersequence itself. For each probability we denote the common supersequence implied by it, till the length of 4, due to technical limitations.

Theorem 1. *The SCWS problem is solvable in $O(Ln^2)$ time and $O(n^2)$ space, where the output is $L + n$, the length of the shortest common weighted supersequence of the input sequences.*

Proof: The Algorithm stops after an iteration in which the weight bound is satisfied. The first applicable length of a $SCS(A[1..n], B[1..n])$ equals n, when the inputs are identical. Therefore, if the SCWS is of length $n + L$ the number of iterations performed is $L + 1$. In each iteration, $p^k_{i,j}$s are computed for all n^2 entries of the table. This computation involves a constant number of operation, as detailed in Lemma 1. In addition, $best_{i,j}$ is determined once in time $O(|\Sigma|n^2)$.

Since in most applications of the Position Weight Matrix, $|\Sigma|$ is a constant, the time complexity is $O(Ln^2)$.

Regarding space, at each iteration we consider for each entry, two probabilities $p^k_{i,j}$ and $p^{k+1}_{i,j}$ and the table consists of n^2 entries, the space requirement is $O(n^2)$.

In real-world applications it is rarely the case that one needs to compare only two data instances. Rather, it is important to be able to compare multiple sequences. Our algorithm can be generalized in the natural way to deal with multiple weighted sequences.

4 Shortest Common Weighted Supersequence with Two Thresholds

The Shortest Common Weighted Supersequence with Two Thresholds (SCWS2) problem, defined in Section 1, Definition 3, in which the probability of each of the sequences, taking part in a supersequence, should exceed its α_i threshold. This problem cannot be solved in the same manner as the $SCWS$ is solved, due to the fact that in the $SCWS$ problem all probabilities are associatively multiplied. As a consequence, $SCWS$ optimal solution can consider at every step increasing prefixes of the input strings, while $SCWS2$ with different threshold to each input, can not.

Observation 2. *It is not sufficient to consider at every step increasing prefixes of the input sequences in order to obtain an optimal solution for the $SCWS2$ problem.*

Intuition: In this problem we would like to multiply high probabilities in both sides. When $A[i], B[j]$ do not *agree* on a preferable symbol, where there is a σ_1 whose probability is maximal in $A[i]$ but $\sigma_2 \neq \sigma_1$ has maximal probability in $B[j]$, it is not clear which symbol one should choose for the common subsequence. It may be more profitable to choose σ_1, even causing the B probability to decrease a lot, since later on a reversed case will occur and balance the probabilities. It, therefore, seems intuitive that local considerations do not suffice for computing the $SCWS2$ problem. This intuition is proven in the next subsection.

We prove that the $SCWS2$ problem is $\mathcal{NP}$-hard using Turing reduction from the Partition problem. To this aim we define the $CWS2$ decision version:

Definition 4. Common Weighted Supersequence with 2 thresholds ($CWS2$)*:*
Input: *Two p-weighted strings A, B of length n over alphabet Σ, and constants L, α_1, α_2, $0 < \alpha_i \leq 1$.*
Output: *Is there a common weighted supersequence of length L, where $(\prod_{y=1}^{n} \pi_{i_y}(a_{i_y})) \geq \alpha_1$ **AND** $(\prod_{y=1}^{n} \pi_{i_y}(b_{j_y})) \geq \alpha_2$, and a_y, b_y reffer to the symbols representing each location of the inputs.*

Definition 5. The Partition *problem: [6]*
Input: *A finite set S and a "value" $v(s) \in Z^+$ for each $s \in S$.*
Output: *Is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} v(s) = \sum_{s \in S - S'} v(s)$?*

Theorem 2. *The $SCWS2$ problem is $\mathcal{NP}$-hard since Partition $\leq_T^p$ $CWS2$.*

Proof: We prove the hardness using a Turing reduction from the Partition problem. Given set $S = s_1, s_2, ..., s_n$ of integers, we construct two weighted sequences $A = A_1..A_n$, $B = B_1..B_n$ both over alphabet of size 6. In addition we need to set a pair of thresholds α_1, α_2 and L, the length of the optimal supersequence.

Observation 3. *The requirement that the product of the probabilities of the common sequence be higher than α_i is equivalent to demanding that the sum of the logarithm of the probabilities will be higher than $\log \alpha_i$.*

The logarithms of probabilities are all negative numbers. We can simply invert the signs of all numbers, making them all positive, and require adding as many numbers as possible without exceeding (the inverted) $\log \alpha_i$.

Note that, given a set $S = s_1, s_2, ..., s_n$ as an input of the Partition problem, if we multiply all $s_i \in S$ by a constant factor and create the set $S' = s'_1, s'_2, ..., s'_n$, then the Partition problem for the set S' remains $\mathcal{NP}$-hard. Therefore, we assume that we can normalize any instance of the Partition problem without changing its hardness.

We are now ready to define the reduction. Given a set $S = \{s_1, ..., s_n\}$ as an input of the partition problem, we set alphabet of the SCWS2 problem to be $\Sigma = \{\sigma_1, ..., \sigma_6\}$. We define two p-weighted sequences, A and B, of length $n + (n + 1)(n - 1)$.

We define the probabilities of the symbols of Σ in the following manner. Let $sum = \sum_{s \in S} s$, the sum of all elements of S. We first multiply every $s_i \in S$ by a constant factor such that: $\sum_{s \in S} 2^{-s_i} = 1/n^2$. Then, we set the probabilities as follows.

$$\pi_i^A(\sigma_j) = \begin{cases} 2^{-s_i} & j = 1 \\ 0.8 & j = 2 \\ x_i & j = 3 \\ 0 & otherwise \end{cases} \qquad \pi_i^B(\sigma_j) = \begin{cases} 2^{-(1/n^2 - s_i)} & j = 1 \\ 0.8 & j = 4 \\ y_i & j = 5 \\ 0 & otherwise \end{cases}$$

Notice that, $2^{-0.32192} = 0.8$, hence, the probability of σ_2 is 0.8 for every location in A and B. Since the sum of all the element in S is $1/n^2$, then the value of x_i is such that $2^{-s_i} + 2^{-x_i} = 0.2$. Therefore it is much more "expensive" to

select σ_1 or σ_3 relatively to σ_2. We can easily see that the sum of all probabilities for every π_i^A is 1. Similarly, the value of y_i (in string B) is such that $2^{s_i-1/n^2} + 2^{-y_i} = 0.2$.

In the above described construction we only refer to n locations in the SCWS2 instance. In addition we set $n+1$ characters according to the following probabilities between any two text locations in A and B (altogether we add $(n+1)(n-1)$ characters to each string). These characters are used as separation between any two characters from the input:

$$\pi_i^A(\sigma_j) = \begin{cases} 1 & j = 6 \\ 0 & otherwise \end{cases} \qquad \pi_i^B(\sigma_j) = \begin{cases} 1 & j = 6 \\ 0 & otherwise \end{cases}$$

We proceed with the Turing reduction. We perform up to $n/2$ iterations. In the ith iteration we set $\alpha_1 = (n-i)0.32192 + 0.5(1/n^2)$, $\alpha_2 = (n-i)0.32192 + (i-0.5)(1/n^2)$, and $L = (n-1)(n+1)+n+i$. We check whether there is a $CWS2$ with these parameters. In case there is a $CWS2$ in ith iteration, we declare a partition of S into sizes i and $n-i$. Otherwise, we increment i by one and start a new iteration. If no $CWS2$ was found after the $n/2$ iteration we terminate the search. A formal proof to Theorem 2 is to appear in the journal version of the article.

5 Conclusions and Open Problems

The main contribution of this paper is in introducing a generalization of the shortest common supersequence problem defined on position weight matrices instead of strings. Two different problem variants are defined. For the first model, a polynomial time algorithm is given, and for the other model, NP-hardness is shown. It remains unclear what is the actual complexity class of the $SCWS2$ problem, since we used a Turing reduction for the hardness proof. In addition, an approximation algorithm can be sought.

References

1. Amir, A., Chencinski, E., Iliopoulos, C.S., Kopelowitz, T., Zhang, H.: Property Matching and Weighted Matching. Theor. Comput. Sci. 395(2-3), 298–310 (2008)
2. Amir, A., Gotthilf, Z., Shalom, R.: Weighted LCS. J. Discrete Algorithms 8(3), 273–281 (2010)
3. Amir, A., Iliopoulos, C.S., Kapah, O., Porat, E.: Approximate Matching in Weighted Sequences. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 365–376. Springer, Heidelberg (2006)
4. Apostolico, A., Landau, G.M., Skiena, S.: Matching for run-length encoded strings. Journal of Complexity 15(1), 4–16 (1999)
5. Clifford, R., Gotthilf, Z., Lewenstein, M., Popa, A.: Restricted common superstring and restricted common supersequence. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 467–478. Springer, Heidelberg (to appear, 2011)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Co., New York (1979)

7. Gotthilf, Z., Lewenstein, M.: Improved Approximation Results on the Shortest Common Supersequence Problem. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 277–284. Springer, Heidelberg (2009)
8. Iliopoulos, C., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., Tsakalidis, A.K.: Efficient Algorithms for Handling Molecular Weighted Sequences. In: IFIP TCS, pp. 265–278 (2004)
9. Iliopoulos, C.S., Mouchard, L., Pedikuri, K., Tsakalidis, A.K.: Computing the repetitions in a weighted sequence. In: Proc. of the 2003 Prague Stringology Conference (PSC 2003), vol. 10, pp. 91–98 (2003)
10. Jiang, T., Li, M.: On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. SIAM Journal on Computing 24(5), 1122–1139 (1995)
11. Maier, D.: The Complexity of Some Problems on Subsequences and Supersequences. Journal of the ACM 25(2), 322–336 (1978)
12. Thompson, J.D., Higgins, D.G., Gibson, T.J.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res. 22, 4673–4680 (1994)
13. Räihä, K.-J., Ukkonen, E.: The Shortest Common Supersequence Problem over Binary Alphabet is NP-complete. Theoretical Computer Science 16(2), 187–198 (1981)
14. Sankoff, D.: Minimal Mutation Trees of Sequences. SIAM Journal on Applied Mathematics 28, 35–42 (1975)
15. Timkovsky, V.G.: Complexity of common subsequence and supersequence problems and related problems. Kibernetika 25, 565–580 (1989); English Translation in Cybernetics 25: 565-580, 1990
16. Venter, J.C., Celera Genomics Corporation: The Sequence of the Human Genome. Science 291, 1304–1351 (2001)

Approximate Regular Expression Matching with Multi-strings*

Djamal Belazzougui and Mathieu Raffinot

LIAFA, Univ. Paris Diderot - Paris 7, 75205 Paris Cedex 13, France
{dbelaz,raffinot}@liafa.jussieu.fr

Abstract. In this paper, we are interested in solving the approximate regular expression matching problem: we are given a regular expression R in advance and we wish to answer the following query: given a text T and a parameter k, find all the substrings of T which match the regular expression R with at most k errors (an error consist in deleting inserting, or substituting a character). There exists a well known solution for this problem in time $O(mn)$ where m is the size of the regular expression (the number of operators and characters appearing in R) and n the length of the text. There also exists a solution for the case $k = 0$ (exact regular expression matching) which solves the problem in time $O(dn)$, where d is the number of strings in the regular expression (a string is a sequence of characters connected with concatenation operator). In this paper, we show that both methods can be combined to solve the approximate regular approximate matching problem in time $O(kdn)$ for arbitrary k. This bound can be much better than the bound $O(mn/\log_{k+2} n)$ achieved by the best actual regular expression matching algorithm in case $d < \frac{m}{k \log_{k+2} n}$ (that is k is not too large and R contains much less occurrences of $\cup$ and $*$ than occurrences of $(\cdot)$).

1 Introduction

The need to search for regular expressions arises in many text-based applications, such as text retrieval, text editing, computational biology and network security. A *regular expression* is a generalized pattern composed of (i) basic strings, (ii) union, concatenation and Kleene closure of other regular expressions. Readers unfamiliar with the concept and terminology related to regular expressions are referred to a classical book such as [1]. We call m the length of our regular expression, not counting operator symbols. The alphabet is denoted Σ, and n is the length of the text.

Exact regular expression matching is a long standing problem and several algorithms have been proposed to obtain efficient algorithm in linear space. The best search results obtained so far are $O(nm(\log\log n)/(\log n)^{3/2})$ time [4].

A recent paper opened a breakthrough in this field. In [5], Bille and Thorup consider matching regular expression considering the number d of substrings

* This work is supported by the french ANR-2010-COSI-004 project MAPPI.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 55–66, 2011.
© Springer-Verlag Berlin Heidelberg 2011

(consecutive symbols concatenated) contained in the expression instead of the number of symbol m. The motivation for studying this kind of pattern is that most of the regular expression searches are in fact words or parts of complete words combined together by union and Kleene closure. For instance, in TATA.(GATA | CATG)*. GCATA, $m = 17$, while $d = 3$. Note that d can be asymptotically $o(m)$. They presented an $O(n(d \log w/w + \log d))$ (where w is the number w of bits in a memory word) algorithm to search a given regular expression in a text, that is, with a complexity depending on d instead of m.

In this paper we focus on approximate regular expression matching, that is, searching for a given regular expression in a text allowing a limited number of errors k, where k might be an insertion, a deletion or a substitution of a character by another. Approximately searching for a general regular expression might be performed in $O(nm/\log_{k+2}(n))$ by the algorithm of Wu, Manber and Myers [16] which space complexity has been improved by Bille and Farach in [2]. However, similarly to the exact matching case, most of the regular expressions approximately searched for contain many long strings. We thus consider in this paper the problem of approximately searching for such patterns and we propose an $O(kdn)$ algorithm which is faster that the general matching algorithm when $d < \frac{m}{k \log_{k+2} n}$.

2 Notations and Definitions

Some notations and definitions that are used in this paper follow.

2.1 Notation

A *word* is a string or sequence of characters over a finite alphabet Σ. The empty word is denoted ε and the set of all words built on Σ (ε included) is Σ^*. A word $x \in \Sigma^*$ x of p is called a *suffix* (resp. *prefix*) of p is $p = ux$ (resp. $p = xv$), $u, v \in \Sigma^*$.

We define also the language to denote regular expressions. Union is denoted with the infix sign "|", Kleene closure with the postfix sign "$*$", and concatenation simply by putting the sub-expressions one after the other. Parentheses are used to change the precedence, which is normally "$*$", ".", "|". We call RE our regular expression pattern, which is of length m and contains d strings. We note $L(RE)$ the set of words generated by RE. Eventually, given a text T we note by $T[i..j]$, the substring of T which starts at position i and ends at position j.

2.2 Definitions

Definition 1. *We define the edit distance between two strings s_1 and s_2 as the minimal number of edit operations needed to transform s_1 into s_2, where the possible edit operations are the deletion of a letter, the substitution of a letter by another and the insertion of a letter. Furthermore we note that number of operations by $\delta(s_1, s_2) = \delta(s_2, s_1)$.*

Definition 2. *In the approximate string matching problem we are given a string q, a threshold k and a text T, and we are asked to return all the substrings of T of the form $T[i..j]$ such that $\delta(T[i..j], q) \leq k$.*

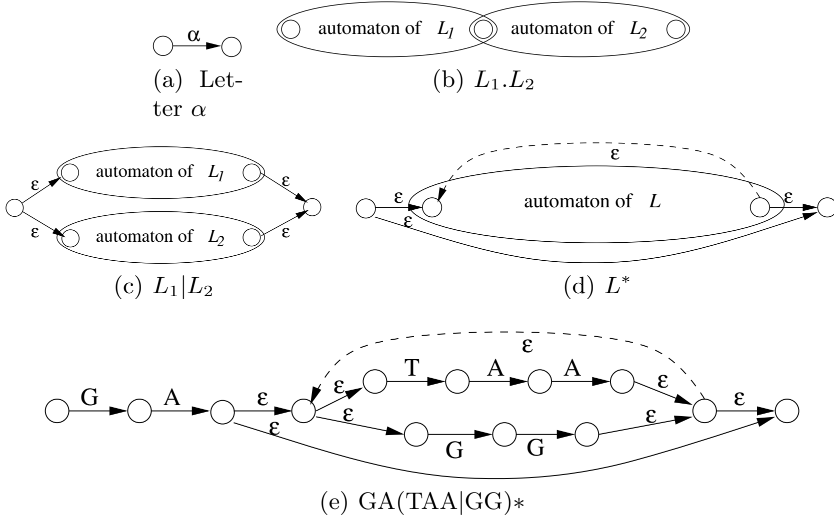
Definition 3. *In the regular expression matching we have a regular expression match R and a text T and we are asked to return every j such there exists a substring $T[j..i] \in L(R)$, where $L(R)$ is the language generated by R.*

Definition 4. *In the approximate regular expression matching we have a regular expression match R, a text T and a threshold k and we are asked to return every j such there exists a substring $T[j..i]$ and a string $p \in L(R)$ such that $\delta(p, T[j..i]) \leq k$.*

3 Thompson's Automaton

We re-use the classical construction of Thompson's automaton to build a non deterministic automaton accepting the language of a given regular expression. The automaton contains ε-transitions and we distinguish two types of nodes, ε-nodes, in which all ingoing transitions are ε-transitions, and the others, denoted L-nodes.

The construction of the automaton is done recursively on the expression using the patterns shown on figure 1.



(a) Letter α

(b) $L_1.L_2$

(c) $L_1|L_2$

(d) L^*

(e) GA(TAA|GG)∗

Fig. 1. Patterns for recursively building Thompson's automaton on a given regular expression and an instance of such an automaton built on GA(TAA|GG)∗

The construction is $O(m)$ and the number of states and transitions is also $O(m)$. We refer the reader to [1] for details on Thompson's construction.

Most of the regular expression matching are based on Thompson's automaton which is directly built from the regular expression using the following rules:

- a regular expression consisting in a single character c generates an automaton with two states I and F, linked with one transition labeled with the character c. The state I is the initial state of the automaton and the state F is the accepting state of the automaton (see Figure 1-(a)).
- a regular expression $R = R_1 \cdot R_2$ generates an automaton which contains all original states and transitions of automatons of R_1 and R_2 except that the final state of automaton of R_1 is merged with initial state of the automaton of R_2 (see Figure 1-(b)).
- a regular expression $R = R_1 \cup R_2$ generates an automaton which contains the states and the transitions which appear in automaton of the regular expressions R_1, R_2 with two new states I, F and four new transitions labeled with ε (see Figure 1-(c)).
- a regular expression $R = R_1^*$ generates an automaton which contains all the original states of R_1 with two new states I, F and four new transitions labeled with ε (see Figure 1-(d)).

The essential property of Thompson's automaton is that it contains $m = O(|R|)$ states and $O(m)$ transitions. Exploiting this property there exists a simple algorithm for regular expression matching on a text of length n which runs in $O(mn)$ time using $O(m)$ working space. This algorithm was extended By Miller and Myers for solving the approximate regular expression matching in the same time and space bounds. We present their algorithm in the following section.

4 Approximate Regular Expression Matching

We now describe more in detail the Myers and Miller's algorithm for approximate regular expression matching. The algorithm preprocesses the regular expression and builds Thompson's automaton on it. A counter c_i of $\lceil \log(k+1) \rceil$ bits is maintained with each state i of the automaton. When set to a value larger than k a counter saturates and gets the maximal value $k+1$.

During the scanning of the text T at each step j this counter will store the smallest distance between any suffix of $T[1..j]$ and the language represented by the state i. Thus at any step j the smallest distance between any suffix of j and the language $L(R)$ is indicated by the counter c_F which corresponds to the accepting state F.

More formally let $E[i, j]$ indicate the smallest distance between state i and the suffixes of $T[1..j]$. That is, at a step j during the scanning of the text character $T[j]$ we let $E[i, j] = c_i$. We denote the states of the automaton by I for the initial state and F for the final state. The states are numbered from 1 to D excluding the initial state. Thus our automaton will have $D+1$ states. We say that a state i is an L-node if it has only a single ingoing transition labeled by a character ℓ_i. Otherwise the state i will be an ε-node with all ingoing transitions being labeled by ε. For any node i we note by $Pre(i)$ the set of nodes with outgoing transitions leading to i (if i is an L-node then clearly the set $Pre(i)$ will have a single element in it). In addition for ε-nodes, we note by $\overline{Pre}(i)$ the set of nodes with outgoing transitions to i excluding the back transition.

The values of $E[i,j]$ are set by the following pseudo-code:

1. **for** $j = 1$ **to** n **do**
2. $\quad E[I,j] \leftarrow 0$
3. $\quad E'[I,j] \leftarrow 0$
4. **end for**
5. **for** $i \in [1,D]$ **do**
6. $\quad E[i,0] \leftarrow \begin{cases} \min E[\overline{Pre}(i),0]+1 & \text{if } i \text{ is an } L\text{-node} \\ \min E[\overline{Pre}(i),0] & \text{if } i \text{ is an } \varepsilon\text{-node} \end{cases}$
7. **end for**
8. **for** $j \in [1,n]$ **do**
9. $\quad$ **for** $i \in [1,D]$ **do**
10. $\quad\quad E'[i,j] \leftarrow \begin{cases} \text{if } i \text{ is an } L\text{-node} \\ \quad \min E[i,j-1]+1, E[\overline{Pre}(i),j-1]+\delta(\ell_i,T[j]), \\ \qquad E'[\overline{Pre}(i),j]+1) \\ \text{if } i \text{ is an } \varepsilon\text{-node} \\ \quad \min E'[\overline{Pre}(i),j] \end{cases}$
11. $\quad$ **end for**
12. $\quad$ **for** $i \in [1,D]$ **do**
13. $\quad\quad E[i,j] \leftarrow \begin{cases} \min\ (E'[i,j], E[\overline{Pre}(i),j]+1) & \text{if } i \text{ is an } L\text{-node} \\ \min\ (E'[Pre(i),j], E[\overline{Pre}(i),j]) & \text{if } i \text{ is an } \varepsilon\text{-node} \end{cases}$
14. $\quad$ **end for**
15. **end for**

An example of the underlying automaton of the algorithm and the values calculated when matching GA(TAA|GG)∗ in GCTAGG is given in Figure 2.

Lemma 1 ([11]). *Given a regular expression R of total size m and after $O(m)$ preprocessing time, we can solve the approximate regular expression matching for any given text T of size n in time $O(mn)$. The space used to solve the query is $O(m)$.*

5 Incremental String Comparisons

Our algorithm needs to solve the following problem while scanning the text. Given a pattern p of length m and a positive integer k we would want to solve the following problem: given a text T read character by character at any given step j, identify all suffixes of $T[1..j]$ which are at distance k from p. Note that because the length of those suffixes can differ from p by at most k (obviously if they are too short or too long then one needs too much insertions or deletions to get p) their number is at most $2k+1$. The following lemma shows an efficient solution to this problem.

Lemma 2 ([8]). *Given a pattern p of length m and after $O(m)$ preprocessing time, we can solve approximate string matching problem for a text T and a*

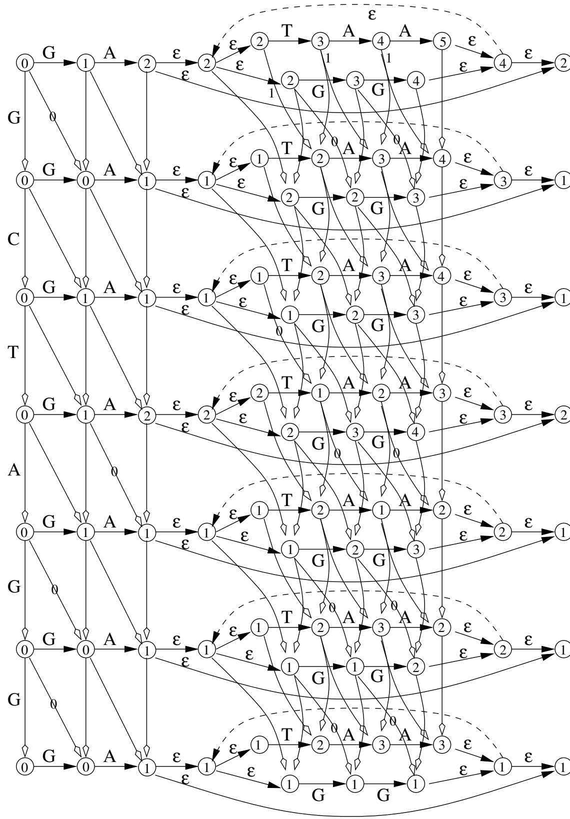


Fig. 2. Myers and Miller's algorithm applied for searching for GA(TAA|GG)∗ in GCTAGG

threshold k in time $O(kn)$. The space used by the query is $O(k^2)$. More specifically at any given time step i the algorithm is able to return $\delta(T[j..i], p)$ for every j such that $\delta(T[j..i], p) \leq k$.

6 Bille and Thorup's Algorithm

Our algorithm is based on Bille and Thorup's algorithm, which we describe now. The main contribution in Bille and Thorup's algorithm is to show that it is possible to do the matching in $O(dn)$ time only where d is the number of strings in the regular expression. Thus this algorithm has the potential to run much faster than the original Thompson algorithm in case the regular expression contains a lot of $(\cdot)$ operators. Seen in another way the number of occurrences of operators $\cup, *$ appearing in R is $O(d)$ only. Next we describe more in detail Billes and Thorup's algorithm [5].

Bille and Thorup Data Structure. First Bille and Thorup generate a new regular expression R' by grouping each maximal sequences of $(\cdot)$ consecutive operators in the original regular expression into a single entity which in fact is just a string of characters. Then each such string is added to a set S and is replaced by a single meta-character in the regular expression R'. It is easy to see that the new regular expression has size $O(d)$ states where d is the number of strings appearing in the regular expression. The alphabet of the new regular expression R' will be union of the original alphabet and the set of added metacharacters and thus will be of size $\sigma + O(d)$. In a second step Bille and Thorup build an Aho-Corasick automaton on the set of strings S which takes space $O(m)$ space and the ordinary Thomson Automaton is built on the regular expression R'.

Regular Expression Matching in Bille and Thorup. For matching the regular expression Bille and Thorup maintain a queue of length m_i for each string $s_i \in S$ of length m_i. This queue uses only m_i bits. We note this queue by $q_i[1..m_i]$. At the beginning all the bits in the queue are set to zero. At each step of the regular expression matching zero or one bit b_i is pushed in the back of each queue q_i. This is done by:

1. The element $q_i[m_i]$ (head of the queue) is thrown out of the queue.
2. Every element $q_i[j]$ is stored into $q_i[j+1]$ for j decreasing from $m_i - 1$ to 1.
3. Finally the bit b is written in $q[1]$.

The matching algorithm can be summarized by the following steps:

1. Read next character c from the text and advance the Aho-Corasick automaton based on character c.
2. The Aho-Corasick automaton will output a set of occurrences of patterns of S. For each pattern $s_i \in S$ reported by the Aho-Corasick automaton push the bit $b_i = 1$ on the back of queue q_i. For all other pattern $s_i \in S$ not reported by the automaton instead push the bit $b_i = 0$ on the back of the queue q_i.
3. Now advance Thompson's automaton by considering each transition outgoing from one of the currently active states and take that transition only if it is labeled with ε or labeled with a metacharacter $s_i \in S$ such that $q_i[m_i] = 1$.

Lemma 3 ([6]). *Given a regular expression R of size m, containing d strings and after $O(m)$ preprocessing time, we can solve the exact regular expression matching for any given text T of size n in time $O(dn)$. The space used to solve the query is $O(m)$.*

7 A New $O(kdn)$ Algorithm

Our solution combines three algorithms:

- The algorithm of [11] for updating and modifying the distances for each state of the automaton.
- The Landau et al [8] algorithm for computing the distances of suffixes of the text and each string in S.
- Additionally we use the approach of Bille and Thorup to maintain the old distance computed for each state. in the relevant past steps.

In our approach we shall use error counters of $\lceil \log(k+1) \rceil$ bits as defined in section 4.

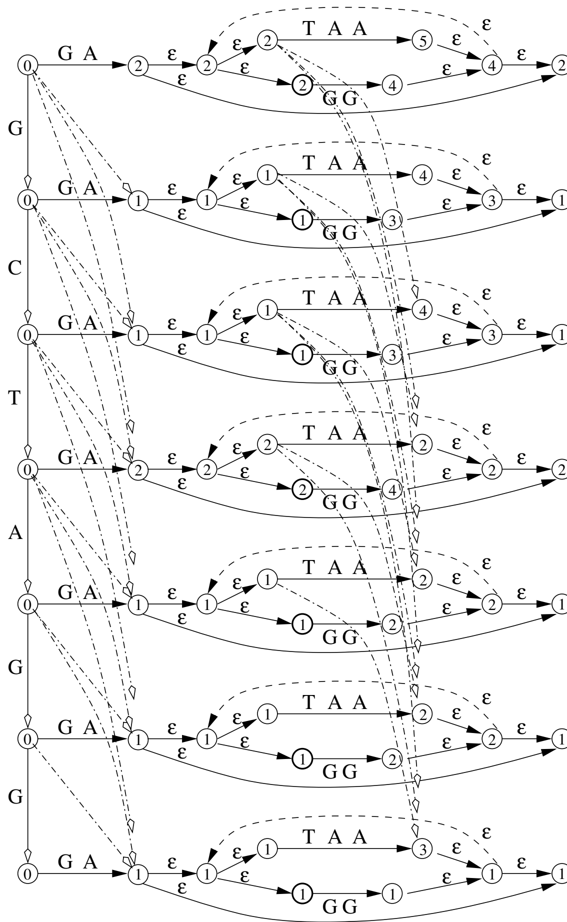


Fig. 3. Our new algorithm applied for searching for GA(TAA|GG)∗ in GCTAGG with $k = 1$. For clarity outgoing edges from bold rounded states are not drawn.

7.1 Preprocessing

The preprocessing phase is quite similar to Bille and Thorup's algorithm. Given the regular expression R, a new regular expression R' is generated by grouping each maximal sequences of consecutive operators $(\cdot)$ into strings adding each such string to a set S and replacing it with a single meta-character in the regular expression R'.

Then in a second phase for each string $s_i \in S$, we build the data structures needed by Landau et al's text approximate matching algorithm.

Bille and Thorup's algorithm maintains for each element $s_i \in S$ of length m_i, a queue q_i holding m_i elements where each element is a bit. In our case we instead maintain a queue q_i of size $m_i + k$, where each element is a counter of $\lceil \log(k+1) \rceil$ bits. At the beginning all values in the queue are initialized to the value $k+1$.

7.2 Matching Algorithm

During the matching we use the Landau et al's matching algorithm as a black-box. At any step j, we let the function $ED(i,j)$ as the function which returns the vector $V[1..\min(m_i, k+1)+k]$ containing all distances of s_i to the substrings $T[j-m_i-k+1..j]$,...,$T[j-\max(m_i,k+1)+k+1..j]$[1]. Also for any queue $q_i[1..m_i+k]$ we let push(q_i, x) denote the operation which pushes the value x on the back of the queue q_i. In our algorithm we will use a temporary vector Δ of size $2k+1$. The intermediate results of the algorithm are virtually stored in a matrix $E[1..D, 1..n]$ but only one column of that matrix need to be stored at any time. The algorithm pseudo-code follows:

1. $E[I,0] \leftarrow 0$
2. **for** $i \in [1, D]$ **do**
3.   **if** i is an L-node **then**
4. $E[i,0] \leftarrow \min E[\overline{Pre}(i), 0] + m_i$
      push$(q_i, E[\overline{Pre}(i), 0])$
5.   **else**
6. $E[i,0] \leftarrow \min E[\overline{Pre}(i), 0]$ // i is an ε-node
7.   **end if**
8. **end for**
9. **for** $j \in [1, n]$ **do**
10. $E'[I, j] \leftarrow 0$
11.   **for** $i \in [1, D]$ **do**
12.   **if** i is an L-node **then**
13. $\Delta[1..\min(m_i, k+1)+k] = ED(i,j)$
14. $E'[i,j] \leftarrow E'[\overline{Pre}(i), j] + m_i$
15.   **for** $t = \max\ (m_i - k, 1)$ to $m_i + k$ **do**

[1] By convention the distance to undefined substring $T[a, j]$ for $a < 1$ is $k+1$.

16. $E'[i,j] \leftarrow \min\ (E'[i,j], \underline{q_i[t]} + \Delta[m_i - t + k + 1])$
    /* Note that $q_i[t] = E[\overline{Pre}(i), j-t]$ and
    $\Delta[m_i - t + k + 1] = \delta(s_i, T[j-t+1..j])$). /*
17. **end for**
18. **else**
19. $E'[i,j] \leftarrow \min E'[\overline{Pre}(i), j]$ // i is an ε-node
20. **end if**
21. **end for**
22. $E[I,j] \leftarrow 0$
23. **for** $i \in [1, D]$ **do**
24. **if** i is an L-node **then**
25. $E[i,j] \leftarrow \min\ (E'[i,j], E[\overline{Pre}(i), j] + m_i)$
    push$(q_i, E[\overline{Pre}(i), j])$
26. **else**
27. $E[i,j] \leftarrow \min\ (E'[Pre(i), j], E[\overline{Pre}(i), j])$ // i is an ε-node
28. **end if**
29. **end for**
30. **end for**

A schematic representation of the underlying automaton of our new algorithm and the values calculated when matching GA(TAA|GG)$*$ in GCTAGG is given in Figure 3.

Theorem 1. *Given a regular expression R of total size m, containing d strings and after $O(m)$ preprocessing time, we can solve the approximate regular expression matching for any given text T of size n and a threshold k in time $O(kdn)$. The space used to solve the query is $O(k^2d + m)$.*

Proof. To prove the correctness of the algorithm, we need to prove that the computed distances for both L-nodes and ε-nodes are correct.

For the L-nodes we consider each L-node a with ingoing transition from a node b labeled with string s_i. This L-node a represents a language $A = B \cdot s_j$ and its predecessor b represents a language B. Now consider the algorithm above. At any step j, the variable $E[a,j]$ must store the smallest distance between any string in $s_a \in A$ and any suffix x of $T[1..j]$. Note that we must have $s_a = s_b \cdot s_i$ with $s_b \in B$. Now notice that x must be writable as $x = x_1 \cdot x_2$ where $\delta(x, s_a) = \delta(x_1, s_b) + \delta(x_2, s_i)$ where s_b must be the string in B with the smallest distance to any suffix of $T[1..j - |x_2|]$ and this suffix must be x_1. This implies that $E[b, j - |x_2|] = \delta(x_1, s_b)$. For the distance $\delta(x, s_a)$ to be at most k we must have both $\delta(x_1, s_b) \le k$ and $\delta(x_2, s_i) \le k$. The second condition implies that $-k \le |x_2| - |s_i| \le +k$ and $\delta(x_2, s_i)$ is actually computed by Landau et al's algorithm for precisely all possible x_2 of lengths in the range $[s_i - k, s_i + k]$. Back to the algorithm, we remark that the counter of an L-node is updated by lines 4,14,16 and 25 in the algorithm, which precisely compute the minimum of all possible values of $\delta(x_1, s_b) + \delta(x_2, s_i)$, that is lines 4, 14 and 25 take care

of the case $x_2 = \varepsilon$ while line 16 takes care of the other possible suffixes s_2 where $\delta(x_2, s_i)$ is retrieved from the vector Δ and $\delta(x_1, s_b)$ is retrieved from the queue q_i.

For proving the correctness of update steps of the ε-nodes, we remark our automaton is exactly the same as that of Myers and Miller's automaton except that we suppress all intermediate L-nodes which are only connected to other L-nodes. We also remark that the update loop for ε-nodes is exactly the same as in Myers and Miller's and the involved states (including L-nodes) are also present in our automaton. Therefore the correctness of our update loop for ε-nodes follows from the correctness of Myers and Miller's algorithm loop and from the fact that the distances of the involved L-nodes is correct as proved above.

The query time of the algorithm is clearly $O(kdn)$ as we have two nested loops with n and d iterations respectively with the latter loop containing an inner loop with at most $2k+1$ iterations in addition to a Landau et al's step (call to function ED) which takes $O(k)$ time.

Concerning the space usage, we note that only the last column of E and E' need to be used at any time. The temporary vector Δ uses space $O(k)$. The total space used by the queues q_i for all i amounts to $O(dk+m)$. The space usage of the algorithm is dominated by the Landau et Al's matching algorithm which uses $O(k^2d+m)$ space.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading (1985)
2. Philip, B., Farach-Colton, M.: Fast and compact regular expression matching. Theoretical Computer Science 409(3), 486–496 (2008)
3. Baeza-Yates, R.A., Gonnet, G.H.: Efficient text searching of regular expressions. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 46–62. Springer, Heidelberg (1989)
4. Bille, P., Thorup, M.: Faster regular expression matching. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 171–182. Springer, Heidelberg (2009)
5. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 1297–1308 (2010)
6. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1297–1308. SIAM, Philadelphia (2010)
7. Knight, J.R., Myers, E.W.: Approximate regular expression pattern matching with concave gap penalties. In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 67–78. Springer, Heidelberg (1992)
8. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM J. Comput. 27(2), 557–582 (1998)
9. Mužátko, P.: Approximate regular expression matching. In: Proceedings of the Prague Stringologic Club Workshop 1996, pp. 37–41 (1996)
10. Myers, E.W.: A four russians algorithm for regular expression pattern matching. J. Assoc. Comput. Mach. 39(2), 430–448 (1992)

11. Myers, E.W., Miller, W.: Approximate matching of regular expressions. Bull. Math. Biol. 51, 7–37 (1989)
12. Myers, E.W., Oliva, P., Guimãraes, K.: Reporting exact and approximate regular expression matches. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 91–103. Springer, Heidelberg (1998)
13. Navarro, G., Raffinot, M.: Fast regular expression search. In: Vitter, J.S., Zaroliagis, C.D. (eds.) WAE 1999. LNCS, vol. 1668, pp. 198–213. Springer, Heidelberg (1999)
14. Thompson, K.: Regular expression search algorithm. Commun. ACM 11, 419–422 (1968)
15. Wu, S., Manber, U., Myers, E.: A subquadratic algorithm for approximate limited expression matching. Algorithmica 15(1), 50–67 (1996)
16. Wu, S., Manber, U., Myers, E.W.: A subquadratic algorithm for approximate regular expression matching. J. Algorithms 19(3), 346–360 (1995)

Persistency in Suffix Trees with Applications to String Interval Problems

Tsvi Kopelowitz[1], Moshe Lewenstein[2,⋆], and Ely Porat[2,⋆⋆]

[1] Weizmann Institute of Science Rehovot, Israel
[2] Bar-Ilan University, Ramat-Gan, Israel

Abstract. The suffix tree has proven to be an invaluable indexing data structure, which is widely used as a building block in many applications. We study the problem of making a suffix tree persistent. Specifically, consider a streamed text T where characters are prepended to the beginning of the text. The suffix tree is updated for each character prepended. We wish to allow access to any previous version of the suffix tree. While it is possible to support basic persistence for suffix trees using classical persistence techniques, some applications which can make use of this persistency cannot be solved efficiently using these techniques alone.

A collection of such problems is that of queries on string intervals of the text indexed by the suffix tree. In other words, if the text $T = t_1 ... t_n$ is indexed, one may want to answer different queries on string intervals, $t_i ... t_j$, of the text. These types of problems are known as position-restricted and contain querying, reporting, rank, selection etc. Persistency can be utilized to obtain solutions for these problems on prefixes of the text, by solving these problems on previous versions of the suffix tree. However, for substrings it is not sufficient to use the standard persistency.

We propose more sophisticated persistent techniques which yield solutions for position-restricted querying, reporting, rank, and selection problems.

1 Introduction

Text indexing is one of the most important paradigms in searching. The idea is to preprocess a text $T = t_1 \cdots t_n$ over alphabet Σ and construct a mechanism that will later provide answers to queries of the form "report all of the occurrences of a pattern P in the text" in time proportional to the size of the *pattern* and output, rather than the size of the text. The suffix tree [10,14,16,17] has proven to be an invaluable data structure for indexing. It is also considered a building block for various other indexing and non-indexing problems.

Some of the suffix tree constructions work in the *online* model [16,17], in which one maintains a suffix tree for a text that arrives character by character, and at any given time we might receive a pattern query. For simplicity, we assume that

⋆ This research was partially supported by the ISF (grant no. 1848/04).
⋆⋆ This research was partially supported by the BSF (grant no. 2006334) and the ISF grant (grant no. 1484/08).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 67–80, 2011.
© Springer-Verlag Berlin Heidelberg 2011

the text arrives from the end towards the beginning. Otherwise a single character added at the end of the text can impose a linear number of changes to the suffix tree. Of course, if the text arrives from beginning to end we can view the text in reversed form and then a queried pattern is reversed as well in order to obtain the correct results. The best currently known results for the online suffix tree are an $O(\log |\Sigma|)$ amortized time per character by Weiner in [17], and $O(\log n)$ worst case per character by Amir et.al in [2]. We also note that for constant-size alphabets there is a different indexing structure by Amir and Nor [3].

Data structures which have the ability to allow access to previous versions of themselves over the updates are known as *persistent data structures* [5,8,9]. For a good survey see [12]. We focus on two types of persistent data structures. The first type is *fully persistent data structures*, in which an update can be made to any version of the data structure. In this type, one can imagine a tree of versions of the data structure as update operations are performed on various versions. The second type is known as *partially persistent data structures*, in which an update can be made only to the last version created. In this type, one can imagine a list of versions of the data structure as update operations are performed only on the tail of the list. In Section 2 we will provide a quick review on some of the known results in this field which we will later use.

To the best of our knowledge persistent suffix trees have not been considered before. Nevertheless, since suffix trees have constant indegree it follows that one can make suffix trees persistent using the result of [8]. However, this persistency is useful solely for navigation purposes, which is sufficient for various standard applications, e.g. queries of the sort "report all of the times in which a specific stock has a series of consecutive values in the stock market, before last March". More sophisticated queries cannot be answered with navigational data on the current text alone.

One subset of problems that we focus on is string interval problems, a.k.a. position restricted problems. Here one has a suffix tree for the full text $T = t_1 \cdots t_n$ but is interested in queries that are narrowed down onto an interval $t_i \cdots t_j$. One problem is known as position restricted indexing, see [13], position restricted reporting, where one desires to report all matches within an interval of the text, position restricted rank, where one desires to know the rank of a given pattern within the interval of the text, and position restricted select, where one desires to find the i'th appearance of a given pattern. The intuition for using a persistent suffix tree for these type of problems is that by accessing the version of the suffix tree just after t_i was added, one may reduce the problem to searching within the prefix of $t_i \cdots t_n$ of length $j - i + 1$.

Unfortunately, the persistent suffix tree on its own does not suffice for efficient solutions for these problems. This happens because versions of the data structure provide bounds for one side of the desired interval query, but not both. Hence, we need to provide a persistent mechanism which supplies the capability for answering the different queries for position restricted problems. We do this by providing a general framework solution, and then show how each of the above applications can be solved using this general framework.

The most natural problem that our general framework solves is the position restricted indexing (PRI) problem. In this problem we wish to preprocess the text $T = t_1 \cdots t_n$, to support subsequent queries of the form "Given a pattern $P = p_1 \cdots p_m$ and two indexes i, j report all of the occurrences of P in $t_i \cdots t_j$". This PRI problem is a very natural one, and was introduced by Mäkinen and Navarro in [13]. PRI has also been addressed by Chien et al [7] where some connections between PRI and range searching in 2D are shown. Recently, Bille and Gørtz in [4] provided a solution for the PRI problem using $O(n \log n)$ preprocessing time, $O(n \log^{\epsilon} n)$ space (for any constant $\epsilon > 0$), and achieve optimal query time of $O(m + occ_{i,j})$ (where $occ_{i,j}$ is the size of the output).

We show in this paper, using the notion of persistency and our general framework, how one can solve PRI using $O(n \log n)$ preprocessing time, $O(n \log^{\epsilon} n)$ space (for any constant $\epsilon > 0$), and achieve query time of $O(m + \log\log|\Sigma| + occ_{i,j})$. For constant size alphabets we achieve the same complexities as those of Bille and Gørtz in [4], however, for general alphabets their solution is more efficient. Nevertheless, we choose to demonstrate our solution to this application as it provides an easier understanding of the use of our general framework, which later will allow us to solve other applications efficiently.

We discuss several other problems for which our general framework can help to achieve better query time from what is currently known. For the counting version of PRI (where we need to count the number of occurrences of P in $t_i \cdots t_j$) we obtain a data structure using $O(n \log n / \log\log n)$ space data structure which answers queries in $O(m + \log\log|\Sigma|)$ time. This is comparable with the results of [13] which by using $n + o(n)$ space can answer queries in $O(m + \log n)$ time. We note that while the authors in [13] claim that they can construct a data structure achieving $O(m + \log\log n)$ query time, using $O(n \log^{\epsilon} n)$ space, is based on an (unproven) claim that the data structure of [1] can answer 2D orthogonal range counting queries in $O(\log\log n)$ time on a grid. However, this cannot be true as that would contradict the lower bounds of Patrascu [15]. In addition, our solution for the counting version improves on the substring rank (SR) problem. In the *substring rank (SR) problem* we need to preprocess T for a substring rank query: given a pattern P and an integer k return the number of occurrences of P in $t_1 \cdots t_k$. The SR problem is a special case of the counting version of PRI since we can answer it using by setting $i = 1$ and $j = k - m + 1$. Thus the bounds of the counting version of PRI apply to SR as well.

Finally, we address the *substring selection problem* where we wish to preprocess T for a substring selection query: given a pattern P and an integer k locate the kth occurrence of P in T. This problem was presented by Mäkinen and Navarro [13], where the authors construct a data structure that requires $O(n\ell \log_n |\Sigma|)$ space where ℓ is an upper bound on the size of the queried patterns, and answer queries in $O(m \log_{\log n} |\Sigma|)$ time. Our data structure requires $O(n \log n / \log\log n)$ space and can answer queries in optimal $O(m)$ time. Furthermore, we do not require any bound on the pattern length to be known in advance.

Our paper is organized as follows. In Section 2 we provide some definitions and preliminaries, including a quick review on some of the known persistent data structures which we will later use. In Section 3 we describe the persistent suffix tree. Then, in Section 4 we discuss the general framework used to solve the applications which we follow up on in Section 5.

2 Definitions and Preliminaries

Given a string S, denote by $|S|$ the length of S. An integer i is a *location* in S if $i = 1, \ldots, |S|$. Given a string $T = t_1 \ldots t_n$ (i.e. $|T| = n$, hereafter the *text*) where for every location i $t_i \in \Sigma$ (hereafter the alphabet), a *suffix* of T is a string of the form $t_i \ldots t_n$, for some location i. Given another string $P = p_1 \ldots p_m$ (hereafter the *pattern*), a location i in T is an *occurrence* of P in T if $t_i \ldots t_{i+m-1} = p_1 \ldots p_m = P$. The suffix tree of a string T is denoted by $ST(T)$.

We would like to assume that $|\Sigma| \leq n$. If this is not the case we can use a hash function in order to reduce Σ to a new alphabet Σ_T such that $|\Sigma_T| \leq n$ as there are at most n different characters in T. However, this can be done efficiently only if we assume that the subset of Σ which we use in T is known in advance. While this is true in the off-line (static) model of the text, it might not be true in the online model.

The suffix tree data structure is comprised of three different types of structures:

1. Tree Structure - The nodes and edges of the tree.
2. Text Structure - The label for each edge in the tree needs to be maintained. This can be costly, therefore, the suffix tree maintains for each edge a pointer into a substring of the text corresponding to the label, instead of explicitly maintaining it.
3. Navigation Structure - Each node in the suffix tree can have up to $|\Sigma|$ children, each one corresponding to one of the characters in Σ. Given a character $\sigma \in \Sigma$ the navigation structure allows us to quickly decide which of the edges to the node's children, if any, correspond to σ. There are two main approaches to solve this problem. The first is using a *pointer array* for each node, where location σ in the array corresponds to the appropriate outgoing edge. While this could induce a cost of $O(|\Sigma|)$ space per node, one can use hash functions in order to reduce the space to be linear in the number of children of each node. However, in an online setting this would require a dynamic hash-function which needs to be persistent, and we are not aware of any good solutions for this problem. The second option is using a balanced search tree over the children of every node, where the order is determined by the appropriate character. This will induce an extra $O(\log |\Sigma|)$ time per node encountered when traversing down the suffix tree.

Note that in the static setting one can maintain for each node two pointers to the beginning and end of the sub-list of leaves which are in the node's subtree. This

task is too difficult in the online setting. The main use of these pointers is to allow reporting the occurrences of a queried pattern. This happens once its appropriate node is located in time proportional to the size of the output (by scanning the list). However, we can overcome this in the online setting by performing a scan of the subtree of the node and outputting the leaves we encounter. This suffices as the size of the subtree is linear in the number of leaves of the subtree (since all of the inner nodes have more than one child).

2.1 Some Persistent Data Structures

Fully Persistent Arrays. Consider an array A, with the following operations:

1. $Store(v, i, x)$ - Store the information x in the ith location of the version named v of A. This also returns v' which is the name of the new version of A.
2. $Access(v, i)$ - Return the information in the ith location of the version named v of A.

Dietz in [8] presented a data structure (DDS for short) that performs *Store* operations in $O(\log\log m)$ expected amortized time, where m is the number of *Store* operations performed so far, and *Access* operations in $O(\log\log m)$ worst-case time. Unfortunately, it is unknown if improved bounds exist for the partially persistent version.

Fully and Partially Persistent Data Structures of Bounded In-degree. The task of making any data structure in which each node has a constant in-degree (meaning the number of pointers to any node is $O(1)$) fully persistent was solved by Dietz et. al. [9] with an overhead of amortized $O(1)$ space and time per operation. The partially persistent version was solved by Brodal [5] with a worst-case overhead of $O(1)$ per operation. Note that in order to be able to use this in a rooted tree with a non constant fan-out, one cannot maintain pointers to parents.

Fully and Partially Persistent Balanced Search Trees. Dietz et. al. in [9] show how one can support a fully persistent red-black tree over n nodes where each update (insertion or deletion) costs $O(\log n)$ worst-case time, and $O(1)$ worst-case space overhead.

2.2 Problems

Problem 1. **PRI-Report** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $PRI - Report(P = p_1 \cdots p_m, i, j)$, report all of the occurrences of P in $t_i \cdots t_j$.

Problem 2. **PRI-Count** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $PRI - Count(P = p_1 \cdots p_m, i, j)$, return the number of occurrences of P in $t_i \cdots t_j$.

Problem 3. **SSR** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $SSR(P = p_1 \cdots p_m, k)$, return the number of occurrences of P in $t_1 \cdots t_k$.

Problem 4. **SSS** Preprocess text $T = t_1 \cdots t_n$, such that given subsequent queries of the form $SSS(P, k)$, return the kth occurrence of P in T.

3 The Persistent Suffix Tree

In this section we briefly explain how to convert a suffix tree to be persistent in the online text model, where characters are prepended to the beginning of the text. Specifically, in a persistent suffix tree for $T = t_1 \cdots t_n$ we wish to be able to access the suffix tree of $t_i \cdots t_n$ for every $1 \leq i \leq n$.

Consider $ST(T)$ and $ST(\sigma T)$ for some $\sigma \in \Sigma$. It is a well know fact that the amount of tree structure changed or added in the transition from $ST(T)$ to $ST(\sigma T)$ is constant [2] [17]. Specifically, the new leaf corresponding to the new suffix is added, and in addition, a new node might be inserted into an existing edge in order to insert the new leaf's parent (if it does not already exist). The process of locating these changes can cost either amortized $O(\log |\Sigma|)$ [17], or worst case $O(\log n)$ [2]. We use the result of Brodal [5] in order to obtain partial persistency. Dealing with the text structure is standard. Thus the remaining task is that of maintaining the navigation structure, which depends upon implementation.

3.1 Using Pointer Arrays

If the pointer array solution is implemented, one can use the result by Dietz [8] in order to obtain an expected amortized overhead of $O(\log \log n)$ per update, and a worst case $O(\log \log n)$ overhead per node encountered when traversing down the suffix tree. While one might think that the number of changes to any pointer array is bounded by $|\Sigma|$ due to each pointer changing at most once, this is not the case as the pointer from, say, a node u corresponding to character σ can change many times when internal nodes are added. Thus the query time would be $O((m + occ) \log \log n)$. There are two ways to further reduce the query time. The first is by noting that when traversing the subtree corresponding to the queried pattern one can use the tree structure (and not the navigation structure) in order to scan the tree using, say, a post-order search. The second is using the techniques we discuss later in Section 4.2 which will allow us to reduce the overhead to $O(\log \log |\Sigma|)$. However, this is under the assumption that $\Sigma = \{1, 2, 3, \cdots, |\Sigma|\}$ to avoid the need of using a persistent dynamic hash function. Thus, the query time can be reduced to $O(m \log \log |\Sigma| + occ)$, and the following theorem has been established.

Theorem 5. *A persistent suffix tree can be maintained such that the cost of prepending a new character suffers an additive overhead of $O(\log \log |\Sigma|)$ expected amortized time and $O(1)$ worst-case space, and indexing queries can be answered in time $O(m \log \log |\Sigma| + occ)$.*

3.2 Using Balanced Search Trees

If the choice of implementation is to use a balanced search tree for the navigation data, one can use the results of Dietz et.al. [9] so that each update will cost an additional $O(\log|\Sigma|)$ time, and the traversal will suffer from an $O(\log|\Sigma|)$ time worst-case overhead per node. Thus, the query time would be $O(m\log|\Sigma|+occ)$ time.

Theorem 6. *A persistent suffix tree can be maintained such that the cost of prepending a new character suffers an additive overhead of $O(\log|\Sigma|)$ expected amortized time and $O(1)$ worst-case space, and indexing queries can be answered in time $O(m\log|\Sigma|+occ)$.*

4 The General Framework

While in some applications that text is static, it is useful to treat each text location as a timestamp. Each timestamp will have its own version of the suffix tree using persistency techniques. More precisely, for text location τ we have a data structure similar to the suffix tree containing only the suffixes of $t_\tau \cdots t_n$. Also, time is defined in reverse - so first create the version at time n and move backwards towards time 1. However, the focus here is on applications that a given query is confined to the substring $t_i \cdots t_j$ and there is a need for additional tools so that the version of the suffix tree at time stamp i can answer the queries for every possible j. Unfortunately, using the persistent suffix tree from the previous section does not suffice for answering such queries. The problem is that we would like the running time to depend only on the output in the restricted range given in the query, while the persistent suffix tree is only able to efficiently filter the locations which appear before i. Using the persistent suffix tree we still need to filter all of the locations after j.

We show a general framework of a data structure that for some height h constructs a data structure which implements a persistent version of the first h levels of the suffix tree. The choice of h is application dependent, but generally speaking we do not wish for h to be too large as the space used will depend on it. Nevertheless, the filtering process for all locations after j will be fast for all traversals in the suffix tree which end in a node of depth at most h using this persistent data structure.

4.1 Snapshots

We define the notion of a *snapshot* for a node u in the suffix tree of T at time τ. This snapshot is denoted by u^τ, and contains the following information:

- A pointer array A_u^τ of $|\Sigma|$ pointers to the at most $|\Sigma|$ children of u^τ which are the children of u in the version of the suffix tree at time τ. The pointer at location σ in A_u^τ will direct to the correct snapshot of child v of u^τ such that the label of the edge (u, v) begins with σ. Of course, if no such edge exists (whether it does not exist in the suffix tree of T, or it does not exist yet in the current timestamp) the pointer will be a null pointer.

– A pointer to the previous snapshot of u. This chain of pointers is called the snapshot list of u, or $SL(u)$.
– The timestamp of the snapshot τ.

For sake of simplicity, a conceptual timestamp $n+1$ is added. This timestamp has a snapshot for every node of distance at least h from the root in the suffix tree of T, in which the pointer array is all null pointers. We use this timestamp in order to have a so called first snapshot of every node. However, in order to save space we do not maintain an array for these snapshots (which would all be null pointers as at timestamp $n+1$ the suffix tree is empty), and instead we use a bit to indicate that these conceptual pointer are all null.

The only nodes that change between version $\tau+1$ of the suffix tree and version τ of the suffix tree are nodes on the path from the root of the suffix tree to the leaf corresponding to the suffix at location τ. Since only the first h nodes on this path are of interest, there is no need to create snapshots for the entire path. Furthermore, for any node u not on the aforementioned path we have that $u^\tau = u^{\tau+1}$ and hence, a new snapshot is not created for such nodes. Therefore, each timestamp induces at most h snapshots. Say that node u is *stamped* at time τ if u^τ is created.

Lemma 7. *For every node u which is stamped at time τ there exists at most one child v that is also stamped at time τ. Therefore, the difference between A_u^τ and $A_u^{\tau+1}$ is only at location σ which corresponds to the edge (u, v).*

The time required to create each snapshot other than the $n+1$ snapshots is $O(|\Sigma|)$ by copying the pointer array. Each of the n timestamps creates at most h snapshots, and so the total construction time for those timestamps is $O(n|\Sigma|h)$.

Navigation. Navigating down the suffix tree at timestamp i is done as usual through the pointer arrays, and thus the cost of locating the node corresponding to P is $O(m)$.

4.2 Using Persistent Arrays

Consider all of the different snapshots of a node u: $u^{\tau_1}, u^{\tau_2}, ..., u^{\tau_t}, u^{n+1}$, and their corresponding pointer arrays $A_u^{\tau_1}, A_u^{\tau_2}, ..., A_u^{\tau_t}, A_u^{n+1}$. Assume that there exists a positive integer k such that $t = k \cdot |\Sigma|$. It will be shown later how to deal with the case where t is not a multiple of $|\Sigma|$. Thus one can divide the snapshots of u to k lists, each of $|\Sigma|$ consecutive snapshots, and in addition the very last snapshot u^{n+1}. Again, for simplicity, assume that $k = 1$, and thus $t = |\Sigma|$. For larger k the process is a repetition of this simpler case k times.

A pointer array is only created for u^{τ_1} and u^{n+1} (using 1 bit). For the other versions of u one can use the data structure of Dietz [8] (denoted by DDS) which was briefly mentioned in Section 2. A bit is used within each snapshot to indicate if it is a *full* snapshot with a complete pointer array, or an *incomplete* snapshot using the DDS. Note that while the pointer array of u^{n+1} is implemented via

a bit to indicate that it is all null pointers, the data structure of Dietz can be adjusted to still work in this situation (the details are standard, but require an exposition of the data structure, and so we choose to omit it here).

The DDS is used on the pointer array of u^{n+1} (which as we mentioned does not actually exist). Next, iterate over u^{τ_x} which is a version of u in $u^{\tau_2}, ..., u^{\tau_t}$, starting with $x = t$ and ending with $x = 2$. Next, let σ be the character leading u^{τ_x} to its appropriate child v^{τ_x}. We use DDS to perform $Store(x, \sigma, (u^{\tau_x}, v^{\tau_x}))$ inserting the appropriate edge/pointer into the array. u^{τ_x} is given a pointer to the DDS, its current timestamp τ_x, the value $|\Sigma| - x$ which we call the relative timestamp of u^{τ_x}, and finally u^{τ_x} is added to $SL(U)$.

If $t < |\Sigma|$, the DDS is still used, but a pointer array is never fully constructed. If there is no integer k such that $t = k \cdot |\Sigma|$, and $t > |\Sigma|$, then the tail of snapshots use the DDS on the last fully constructed pointer array.

As before, each suffix tree version induces h snapshots. Each incomplete snapshot, the DDS uses $O(\log\log|\Sigma|)$ time. Each full snapshot, except for snapshots at time $n+1$, requires $O(|\Sigma|)$ time which amortizes to $O(1)$ time over the previous $O(\Sigma)$ incomplete snapshots. Thus, the total time spent is $O(nh \log\log|\Sigma|)$, and the space used is $O(nh)$.

Navigation. Navigation is basically done as before; the only difference is the method of traversing down the current version of the suffix tree without complete pointer arrays (for the full snapshots simply use the array). This is done using the DDS. If the navigation reached u^{τ}, whose relative timestamp is x, and needs to continue traversing with σ, then use $Access(x, \sigma)$ on the DDS of u^{τ} in order to obtain the correct pointer. This implies that each traversal through a node in the current version of the suffix tree will take $O(\log\log|\Sigma|)$ time. So the cost of locating the node corresponding to P is $O(m \log\log|\Sigma|)$.

4.3 Renaming

First a solution is provided for navigating with patterns of length which is a multiple of $\log\log|\Sigma|$. The more general case is considered in the following subsection. Given T over alphabet Σ, construct a new alphabet $\Sigma' \subseteq \Sigma^{\log\log|\Sigma|}$ by taking every substring of T of size $\log\log|\Sigma|$ and renaming it. The renaming scheme needs to maintain the lexicographical order between the substrings. For this, construct a trie for all of the substrings of T of size $\log\log|\Sigma|$, and then label each leaf. The labels should maintain the order defined by the leaves of the trie. Thus, the labels preserve the lexicographical ordering of the substrings corresponding to them. The construction of this trie takes $O(n \log\log|\Sigma|)$ time.

Next, construct $\log\log|\Sigma|$ new text strings, named $T_1, T_2, ..., T_{\log\log|\Sigma|}$ as follows. For each $1 \leq i \leq \log\log|\Sigma|$, the new text T_i is a text over alphabet Σ' where the jth character in T_i is the renamed label corresponding to the substring $t_{i+(j-1)\log\log|\Sigma|} \cdots t_{i+j\log\log|\Sigma|-1}$. In other words, T_i is the renamed version of $t_i t_{i+1} \cdots .t_n$, removing any tail of characters that might remain, as $n - i + 1$ might not be a multiple of $\log\log|\Sigma|$. Note that only the first $\log\log|\Sigma|$ suffixes

of T are being renamed, as the renamed versions of any other suffix (not one of the first $\log\log|\Sigma|$) is a proper suffix of one of the renamed suffixes. Next, construct the text $T' = T_1\$_1T_2\$_2...T_{\log\log|\Sigma|}\$$, and use the previous solution with persistent pointer arrays on this text, but only for height $h' = \frac{h}{\log\log|\Sigma|}$ as each renamed character corresponds to $\log\log|\Sigma|$ original characters (and we wish to only maintain this data structure for nodes in the original tree of height at most h).

It is important to note that while the order of the timestamps defined by the original text become scrambled, it is still possible to maintain the correct order of timestamps by first constructing the suffix tree for T', and then creating snapshots for paths by order of the original timestamps rather than the order defined by T'.

The construction time is now $O(n\log\log|\Sigma|)$ time for the renaming, and $O(nh') = O(nh/\log\log|\Sigma|)$ for the solution using the persistent pointer arrays for a total of $O(n(\log\log|\Sigma| + \frac{h}{\log\log|\Sigma|}))$.

Navigation. Recall the assumption that m is a multiple of $\log\log|\Sigma|$. So, one can use the renaming trie in order to obtain a new pattern P' over Σ' of size $\frac{m}{\log\log|\Sigma|}$, and then the node corresponding to P' in $O(m'\log\log|\Sigma'|) = O(\frac{m\log\log|\Sigma'|}{\log\log|\Sigma|})$ time, which is $O(m)$ as:

$$\log\log|\Sigma'| \leq \log\log(|\Sigma|^{\log\log|\Sigma|}) = \log\log|\Sigma| + \log\log\log|\Sigma| = O(\log\log|\Sigma|).$$

4.4 Renaming for Any Size Pattern

For the case in which m is not a multiple of $\log\log|\Sigma|$, assume that $m \equiv k$ modulo $\log\log|\Sigma|$ where $0 < k < \log\log|\Sigma|$. The previous renaming scheme can still be used on the first $m-k$ characters of P in order to obtain a pattern P' over Σ'. Then, one can locate the node corresponding to P' in $O(m)$ time. What is left to be done is to provide another renaming scheme in order to deal with the last k characters. To this end, the renaming process is done for every length from 1 to $\log\log|\Sigma| - 1$, by using a renaming trie for each possible length.

For every node u in the suffix tree of T', $\log\log|\Sigma| - 1$ pointer arrays are maintained, each corresponding to one of the renaming schemes for each possible length, and each of length corresponding to the length of that specific renaming scheme. So for the renaming scheme for every length $x < \log\log|\Sigma|$, node u uses a pointer array of size $|\Sigma|^x$. Each location in the pointer array corresponds to an edge corresponding to the appropriate renamed character in the renaming scheme of length x. It is crucial to note that this is only done for nodes u in T', and not recursively. Each of the $\log\log|\Sigma| - 1$ pointer arrays then goes through the process of becoming persistent using the same techniques as above.

The additional construction time is $O(nh'\log\log|\Sigma|)$, which combines together with the previous section to $O(n(\log\log|\Sigma| + h))$ time for construction. the space is $O(nh'\log\log|\Sigma|) = O(nh)$.

Navigation. As for the navigation time, if $m \geq \log\log|\Sigma|$, the time will still be $O(m)$ as the extra $O(\log\log|\Sigma|)$ cost induced by having to move to a new renaming scheme and locating the appropriate snapshot can be amortized over previous work done in the suffix tree for T'. However, if $m < \log\log|\Sigma|$, the cost of searching the correct snapshot cannot be amortized, and so the cost will be $O(m - k + \log\log(|\Sigma|^k)) = O(m + \log\log|\Sigma|)$ time.

5 Applications

In this section we will see how to use the general framework from Section 4 in order to solve some problems that fit the model.

5.1 PRI-Report

As mentioned by Mäkinen and Navarro [13], one can obtain a data structure which supports PRI-Report by using the suffix tree and the data structure of Alstrup et.al. [1]. This provides a data structure that uses $O(n\log n)$ preprocessing time with $O(n\log^\epsilon n)$ space (for any constant $\epsilon > 0$), and achieves query time of $O(m + \log\log n + occ_{i,j})$. Thus, by setting $h = \log\log n$ one can answer PRI-Report as follows.

If $m \geq \log\log n$ then by using the data structure from [13], the query time is $O(m + occ_{i,j})$, which is optimal. If $m < \log\log n$ then navigate through the data structure from Section 4 with P on version i of the suffix tree till a snapshot u^τ is reached, where u is the node corresponding to P. Note that $i \leq \tau$. The next procedure is based on the following lemma.

Lemma 8. *Let u be a node in the suffix tree of T corresponding to P, and let $u^{\tau_1}, u^{\tau_2}, ..., u^{\tau_t}, u^{n+1}$ be the snapshots of u. Then P appears in T only at locations $\tau_1, \tau_2, ..., \tau_t$.*

Proof. Each of the snapshots is created due to P being at the location of that timestamp. □

Given that the snapshot list for u is ordered by the timestamps, one can scan the list starting at u^τ till a snapshot $u^{\tau'}$ is reached where $\tau' > j$. Due to Lemma 8 every snapshot encountered by the scan except for the last one corresponds to an occurrence of P. Thus each location in the output costs $O(1)$ time to output after u is located. This provides the following:

Theorem 9. *The PRI-Report problem can be solved using $O(n\log^\epsilon n)$ space (for any constant $\epsilon > 0$) and $O(n\log n)$ preprocessing time, with $O(m + \log\log|\Sigma| + occ_{i,j})$ query time.*

5.2 PRI-Count

In the counting version (PRI-Count), one wishes to report only the number of occurrences of P in $t_i \cdots t_j$, without listing the occurrences. While this can be

solved in $O(m + \log\log|\Sigma| + occ_{i,j})$ time using the solution to the reporting version, this can be wasteful if the output size is fairly large. Instead, the following data structure which is based on ideas similar to the reporting scenario is presented, which manages to avoid the cost of the output size all together.

If the pattern happens to be large enough ($m \geq \log / \log\log n$) the following solution can be used. Given a PRI-Count query, the suffix tree is first used in order to locate the node u corresponding to P. This node covers a consecutive range of suffixes, sorted by their lexicographical ordering (this can be viewed as a consecutive sub-array of the suffix array). Locating this range (l, r) can be done offline in linear time per every node in the suffix tree. Next, a 2D orthogonal range counting query is performed using the data structure of [11]. In [11], it is shown how to construct a data structure using $O(n)$ space and $O(n \log n)$ preprocessing time which allows to answer 2D orthogonal range counting queries in (optimal for this space [15]) $\emptyset(\log n / \log\log n)$. Thus the total query time is $O(m + \log n / \log\log n) = O(m)$ and optimal.

For the case where $m < \log n / \log\log n$ the general framework is used again. This time set $h = \log n / \log\log n$. When given a query, navigate through the data structure from Section 4 with P on both versions i and j of the suffix tree till a snapshot u^τ is reached from the i-version and a snapshot $u^{\tau'}$ is reached from the j-version. All of the snapshots in the snapshot list of u between u^τ and $u^{\tau'}$, excluding $u^{\tau'}$, correspond to occurrences we are interested in counting. By computing the distance of each snapshot from the end of its snapshot list in the preprocessing phase, one can subtract the distance of $u^{\tau'}$ from the distance of u^τ during the query phase in constant time. The correctness of this answer follows directly from Lemma 8.

This provides the following:

Theorem 10. *The PRI-Count problem can be solved using $O(n \log n / \log\log n)$ space and $O(n \log n)$ preprocessing time, with $O(m + \log\log|\Sigma|)$ query time.*

Substring Rank. The solution for $SSR(P, i)$ is a direct application PRI-Count, by computing $PRI - Count(P, 1, i)$ as this will count the number of times P appeared in $t_1 \cdots t_n$.

5.3 Substring Select

In order to be able to answer substring select queries efficiently, a data structure which solves the *Range Selection* problem is required.

Problem 11. **Range Selection** Given an array of integers $A = (a_1, ..., a_n)$ where $\forall 1 \leq i \leq n, 1 \leq a_i \leq n$, preprocess A such that given a query (i, j, k) return the kth smallest value in the set $\{a_x | i \leq x \leq j\}$.

Brodal and Jørgensen [6] show a solution for the Range Selection problem which uses linear space and $O(n \log n)$ construction time, and uses $O(\log n / \log\log n)$ query time. The construction time there is dominated by the need to sort the

input. By constructing their solution on the suffix array, which is the array of the locations of the suffixes in their lexicographical ordering, one can answer substring select queries efficiently for $m \geq \log n / \log\log n$ as follows. Furthermore, this construction will take linear time in this case as the range is bounded by n. Given a SSS query, the suffix tree is first used in order to locate the node u corresponding to P. This node covers a consecutive range of suffixes, sorted by their lexicographical ordering (this can be viewed as a consecutive sub-array of the suffix array). Locating this range (l, r) is done like in the PRI-Count query. Next, perform a range selection query (l, r, k) in order to obtain the answer for the query.

For the case where $m < \log n / \log\log n$ the general framework is used again. This time set $h = \log n / \log\log n$. To answer a query, navigate through the data structure from Section 4 with P on version 1 of the suffix tree till a snapshot u^τ is reached, where u is the node corresponding to P. Next, one can preprocess the snapshot lists to all be in arrays. Thus jumping to the kth snapshot of u in $SL(u)$ to obtain $u^{\tau'}$ can be done in constant time. τ' is the answer to our query. Thus, the total query time is $O(m)$ which is optimal.

Theorem 12. *The SSS problem can be solved using $O(n \log n / \log\log n)$ space and preprocessing time, with $O(m)$ query time.*

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: IEEE Symposium on Foundations of Computer Science, pp. 198–207 (2000)
2. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards Real-Time Suffix Tree Construction. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 67–78. Springer, Heidelberg (2005)
3. Amir, A., Nor, I.: Real-time indexing over fixed finite alphabets. In: Proc. of the Symposium on Discrete Algorithms (SODA), pp. 1086–1095 (2008)
4. Bille, P., Gørtz, L.: Substring Range Reporting. To Appear in Proc. 22nd Combinatorial Pattern Matching Conference (2011)
5. Brodal, G.S.: Partially Persistent Data Structures of Bounded Degree with Constant Update Time. Nord. J. Comput. 3(3), 238–255 (1996)
6. Brodal, G.S., Jørgensen, A.G.: Data structures for range median queries. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 822–831. Springer, Heidelberg (2009)
7. Chien, Y., Hon, W., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In: Data Compression Conference (DCC), pp. 252–261 (2008)
8. Dietz, P.F.: Fully Persistent Arrays (Extended Array). In: Proc. of Symposium on Discrete Algorithms (SODA), pp. 235–244 (1999)
9. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)
10. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. 38th IEEE Symposium on Foundations of Computer Science, pp. 137–143 (1997)

11. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
12. Kaplan, H.: Persistent data structures. In: Handbook on Data Structures, pp. 241–246. CRC Press, Boca Raton (1995)
13. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. 387(3), 332–347 (2007)
14. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. of the ACM 23, 262–272 (1976)
15. Patrascu, M.: Lower bounds for 2-dimensional range counting. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), pp. 40–46 (2007)
16. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
17. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Approximate Point Set Pattern Matching with L_p-Norm

Hung-Lung Wang[1,⋆] and Kuan-Yu Chen[2]

[1] Institute of Information Science and Management
National Taipei College of Business, Taipei, Taiwan 100
hlwang@webmail.ntcb.edu.tw

[2] Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106
d94004@csie.ntu.edu.tw

Abstract. Given two sets of points, the text and the pattern, determining whether the pattern "appears" in the text is modeled as the point set pattern matching problem. Applications usually ask for not only exact matches between these two sets, but also approximate matches. In this paper, we investigate a one-dimensional approximate point set matching problem proposed in [T. Suga and S. Shimozono, Approximate point set pattern matching on sequences and planes, CPM'04]. What requested is an optimal match which minimizes the L_p-norm of the difference vector $(|p_2 - p_1 - (t'_2 - t'_1)|, |p_3 - p_2 - (t'_3 - t'_2)|, \ldots, |p_m - p_{m-1} - (t'_m - t'_{m-1})|)$, where $p_1, p_2, \ldots, p_m$ is the pattern and $t'_1, t'_2, \ldots, t'_m$ is a subsequence of the text. For $p \to \infty$, the proposed algorithm is of time complexity $O(mn)$, where m and n denote the lengths of the pattern and the text, respectively. For arbitrary $p < \infty$, the time complexity is $O(mnT(p))$, where $T(p)$ is the time of evaluating x^p for $x \in \mathbf{R}$.

Keywords: point set pattern matching, L_p-norm, dynamic programming.

1 Introduction

Given two sets of points, the text and the pattern, in an Euclidean space, the *point set pattern matching* problem (PSPM) determines whether the pattern "exactly matches" a subset of the text. For two sets of points A and B, A is said to exactly match B if there is a transformation $\mathcal{L}$ such that $\mathcal{L}(A) = B$. PSPM was first investigated by de Rezende and Lee [5], where point sets are in a d-dimensional Euclidean space, and transformations such as translation, rotation, reflection, and scaling are considered. Later in [3], Cardoze and Schulman investigated this problem from different aspects that affect the design and

⋆ The authors would like to thank Prof. Kun-Mao Chao for helpful comments. Kuan-Yu Chen and Hung-Lung Wang were supported in part by NSC grants 98-2221-E-002-081-MY3 and 99-2115-M-141-003-MY2, respectively, from the National Science Council, Taiwan.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 81–86, 2011.
© Springer-Verlag Berlin Heidelberg 2011

analysis of algorithms. Parameters that are widely discussed include the dimension of the space, the group of transformations, *the quality of matches*, and the input datatype. For the quality of matches, measurements adopted usually include a threshold on the number of mismatches allowed in the pattern or the *distance*, like the Hausdorff distance, between two sets. Normally, problems that seek for a subset of the text with the best quality of matches with respect to the pattern are called the *approximate point set pattern match* problems (APSPM). In this paper, we investigate a one-dimensional APSPM, where the quality of matches is measured by the L_p-*distance* between two sets of points [14]. For convenience, we view a set of one-dimensional points as an increasing number sequences, corresponding to the positions of the points, and call such sequences the *point sequences*. The L_p-distance between two point sequences $p_1, p_2, \ldots, p_m$ and $t_1, t_2, \ldots, t_n$ is infinite if $m \neq n$, and the L_p-norm of the vector $(|t_2 - t_1 - (p_2 - p_1)|, |t_3 - t_2 - (p_3 - p_2)|, \ldots, |t_n - t_{n-1} - (p_n - p_{n-1})|)$ otherwise. We name the problem as L_p-APSPM.

Related Results: Detailed surveys on PSPM and APSPM can be found in [3,15]. We summarize the results for the one-dimensional case and problems that use L_p-norm as the measurement. In the remainder of this paper, the cardinalities of the text and the pattern are denoted by n and m, respectively. For one-dimensional case, PSPM can be solved in $O(n \log n + mn)$-time [5], which is the currently best result. In [14], Suga and Shimozono investigated L_1-APSPM and proposed an $O(mn^2)$-time algorithm. Mäkinen [11,12,13] investigated a similar problem, where the cost on translations are considered. The technique used in [12] can be applied to solve L_1-APSPM in $O(mn)$ time. In [8,9,10], Lipsky and Porat investigated the one-dimensional APSPM problem where the pattern and the text are considered as two sequences of natural numbers less than or equal to σ. Matches are defined only between the pattern and a substring, i.e. a consecutive subsequence, of the text. Quality of matches is measured by L_1-, L_2-, and L_∞-norms, and $O(\frac{1}{\epsilon^2} n \log m \log \sigma)$-, $O(n \log n)$-, and $O(\frac{1}{\epsilon} n \log m \log \sigma)$-time approximation algorithms, with factor ϵ, are proposed, respectively.

We shall show that L_p-APSPM, for $p \geqslant 1$, is a generalization of PSPM in Section 2. For $p \to \infty$, the proposed algorithm is of time complexity $O(mn)$. For arbitrary $p < \infty$, we give an algorithm that runs in $O(mnT(p))$ time, where $T(p)$ is the time of evaluating x^p for $x \in \mathbf{R}$. It is noted that $T(p)$ depends on the model of computation considered (e.g. $O(1)$ if a "shift left" instruction is allowed [4]). The remainder of this paper is organized as follows. In Section 2, we formally define the problems and roughly sketch the main ideas of our approaches. Detailed algorithms are proposed in Sections 3 and 4. Due to the space limitation, some proofs are omitted in this paper.

2 Problem Definitions

The L_p-APSPM problem is defined as follows.

Problem 1. (L_p-APSPM) Given a text point sequence $T = t_1, t_2, \ldots, t_n$ and a pattern point sequence $P = p_1, p_2, \ldots, p_m$, L_p-APSPM seeks for a subsequence $T' = t'_1, t'_2, \ldots, t'_m$ of T such that

- $\sum_{2 \leqslant i \leqslant m} |p_i - p_{i-1} - (t'_i - t'_{i-1})|^p$ is minimized, if $p < \infty$;
- $\max_{2 \leqslant i \leqslant m} |p_i - p_{i-1} - (t'_i - t'_{i-1})|$ is minimized, if $p \to \infty$.

According to the definition, PSPM can be reduced to L_p-APSPM by asking whether the resulting norm equals zero. In the following, we simply denote L_p-APSPM for $p \to \infty$ as L_∞-APSPM.

Outline of the Methods: The problem can be solved via dynamic programming. However, straightforward implementations will result in less efficient algorithms. Our methods are to reduce the time for computing each row of the *DP-table*, which, without ambiguity in each section, is the two-dimensional array induced by the corresponding recurrence relation.

- L_p-APSPM: Each row of the DP-table is computed by searching the column minima of a *monge matrix* [1], defined in Section 3.
- L_∞-APSPM: Each row of the DP-table is computed via *range minimum query* [6].

For succinctness, a number sequence $\gamma_1, \gamma_2, \ldots, \gamma_k$ is also denoted by $\{\gamma_i\}_{i=1}^k$, and $w_j^- = t_j - (p_{i+1} - p_i)$. In the following, we shall focus on how the $(i+1)$th row of the DP-table is evaluated, given the ith row.

3 L_p-APSPM for Arbitrary $p < \infty$

For $p < \infty$, a recurrence relation for L_p-APSPM can be derived immediately as follows. For $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$, and $i \leqslant j$, we define

$$d_{i+1,j} = \begin{cases} 0, & \text{if } i = 0; \\ \infty, & \text{if } i > 0 \text{ and } j = 1; \\ \min_{i \leqslant k < j}\{d_{i,k} + |w_j^- - t_k|^p\}, & \text{otherwise,} \end{cases} \quad (1)$$

where $d_{i,j}$ denotes the minimum L_p distance between $p_1, p_2, \ldots, p_i$ and a length i subsequence of T ending at j. The value $\min_{1 \leqslant k \leqslant n} d_{m,k}$ corresponds to the minimum L_p distance between P and T.

For fixed i, we define an n by n matrix $A_i = [a_{xy}]$ with

$$a_{xy} = \begin{cases} d_{i,x} + |w_y^- - t_x|^p, & \text{if } i \leqslant x < y; \\ \infty, & \text{otherwise.} \end{cases}$$

Lemma 1. *For $A_i = [a_{xy}]$, we have $\min_{1 \leqslant x \leqslant n} a_{xy} = d_{i+1,y}$.*

According to Lemma 1, to compute $d_{i+1,j}$ for $2 \leqslant j \leqslant n$, it suffices to compute all column minima of matrix A_i. In Lemma 2, we show that A_i is a *monge matrix*. A matrix $A = [a_{ij}]$ is said to be monge if $a_{ij} + a_{\ell k} \leqslant a_{ik} + a_{\ell j}$, for all $1 \leqslant i < \ell \leqslant n$ and $1 \leqslant j < k \leqslant n$. To search all the column minima of an n by n monge matrix, Aggarwal *et al.* [1] proposed an efficient algorithm, the SMAWK algorithm, which runs in $O(n)$ time.

Lemma 2. *Matrix A_i is a monge matrix.*

For each row i in the DP-table, it takes $O(nT(p))$ time to compute all the elements, where $T(p)$ is the time of evaluating x^p for $x \in \mathbf{R}$. Together with Hirschberg's algorithm for reducing space [7], we derive the following theorem.

Theorem 1. *For $p < \infty$, L_p-APSPM can be solved in $O(mnT(p))$ time and $O(n)$ space.*

4 L_p-APSPM for p Approaching Infinity

In this section, we show how to solve L_∞-APSPM. Analogously, we let $d_{i,j}$ denote the minimum L_∞ distance between $p_1, p_2, \ldots, p_i$ and a length i subsequence of T ending at j. A recurrence relation similar to (1) can be derived except that $\min_{i \leqslant k < j}\{d_{i,k} + |w_j^- - t_k|^p\}$ is replaced by $\min_{1 \leqslant k \leqslant j} \max\{d_{i,k}, |w_j^- - t_k|\}$.The value $\min_{1 \leqslant k \leqslant n} d_{m,k}$ corresponds to the minimum L_∞ distance between P and T. A straightforward dynamic programming approach takes $O(mn^2)$ time. To reduce the time complexity, a primary claim is stated in Lemma 3, which will be proved in Section 4.3.

Lemma 3. *Given $d_{i,1}, d_{i,2}, \ldots, d_{i,n}$ for some i, the values $d_{i+1,1}, d_{i+1,2}, \ldots, d_{i+1,n}$ can be computed in $O(n)$ time.*

Based on Lemma 3, an $O(mn)$-time algorithm can be derived since there are m rows to be filled. Unfortunately, the Monge property mentioned in the previous section is not applicable. In the following, we prove Lemma 3 constructively by relying on the technique of *range minimum query* (RMQ for abbreviation) [2]. Section 4.1 explains why RMQ can be applied to compute $\{d_{i+1,j}\}_{j=1}^n$, given $\{d_{i,j}\}_{j=1}^n$. Section 4.2 shows how to determine the queried intervals efficiently.

4.1 Computing the DP-Table with RMQ

Consider xy-plane with $n+2$ points $(t_k, d_{i,k})$, for $0 \leqslant k \leqslant n+1$, where $t_0 = -\infty$, $t_{n+1} = \infty$, and $d_{i,0} = d_{i,n+1} = 0$. Let function $f(x) = |w_j^- - x|$. By definition, the value $d_{i+1,j}$ is either $d_{i,k}$ (Figure 1(a)) or $|w_j^- - t_k|$ (Figure 1(b)), for some k. In both cases, there is a specific interval $I = (t_\ell, t_r)$ satisfying the following: (i) $d_{i,\ell} < f(t_\ell)$ and $d_{i,r} < f(t_r)$, (ii) $w_j^- \in I$, and (iii) $\forall t_k \in I$, $d_{i,k} > f(t_k)$. We call such an interval I the *dominating interval.* It should be noted that the dominating interval always exists because of the two boundary points $(t_0, d_{i,0})$ and $(t_{n+1}, d_{i,n+1})$. According to the dominating interval, we divide the candidates that result in $d_{i+1,j}$ into three categories, which are stated as in Lemma 4.

Lemma 4. *Let $Q = I \cap [t_1, t_{j-1}]$. We have $d_{i+1,j} = \min\{\min_{t_k \in Q} d_{i,k}, |w_j^- - t_r|, |w_j^- - t_\ell|\}$.*

For convenience, we refer to the queried interval as a point pair of the text. Since $|w_j^- - t_k|$ can be computed in constant time, given i, j, and k, the remainder is to compute $\min_{t_k \in Q} d_{i,k}$. Notice that computing $\min_{t_k \in Q} d_{i,k}$ is equivalent to an

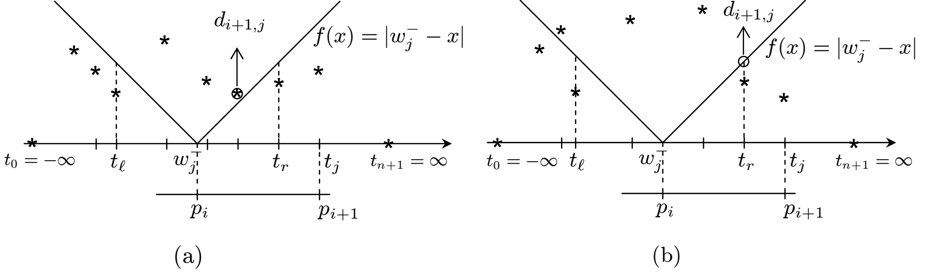


Fig. 1. xy-plane with "$\star$" being the points $(t_k, d_{i,k})$

RMQ query over the array $[\, d_{i1}\ d_{i2} \cdots d_{in} \,]$ with queried interval Q. Therefore, for $1 \leqslant j \leqslant n$, $d_{i+1,j}$ can be computed in amortized constant time ($O(n)$-time for preprocessing and constant time for each query), given the queried intervals.

4.2 Determining the Queried Intervals

To ease the representation, we introduce the notion of *pivots* of a number sequence. Given a number sequence $\Gamma = \gamma_1, \gamma_2, \ldots, \gamma_k$, the element γ_i is a *right pivot* (respectively, a *left pivot*) of Γ if $\gamma_i < \gamma_j$, for $i < j \leqslant k$ (respectively, $\gamma_j < \gamma_i$ for $1 \leqslant j < i$). The subsequence consisting of all right pivots is called the *right-pivot subsequence*. The term *left-pivot subsequence* is defined similarly. For example, the right-pivot and left-pivot subsequences of sequence $3, 1, 4, 1, 5, 9, 2, 6$ are $1, 2, 6$ and $3, 4, 5, 9$, respectively. Note that the right-pivot and left-pivot subsequences are both increasing subsequences. Let $X = \{x_k : x_k = t_k + d_{ik}\}_{k=1}^n$ and $Y = \{y_k : y_k = t_k - d_{ik}\}_{k=1}^n$. For convenience, the queried interval is represented as (t_a, t_b) instead of the index pair (a, b).The procedure for determining the queried interval works as follows:

- Compute the right-pivot subsequence $x_{\alpha_1}, x_{\alpha_2}, \ldots, x_{\alpha_s}$ of X.
- Compute the left-pivot subsequence $y_{\beta_1}, y_{\beta_2}, \ldots, y_{\beta_t}$ of Y.
- Find the intervals $R = [x_{\alpha_k}, x_{\alpha_{k+1}}]$ and $L = [y_{\beta_{k'}}, y_{\beta_{k'+1}}]$, where $w^- \in R \cap L$. The dominating interval is $I = (t_{\alpha_k}, t_{\beta_{k'+1}})$, and $I \cap [t_1, t_{j-1}]$ is the queried interval.

The correctness and time complexity are shown in Lemmata 5 and 6.

Lemma 5. *$I = (t_{\alpha_k}, t_{\beta_{k'+1}})$ is the dominating interval.*

Lemma 6. *The dominating intervals of $d_{i+1,k}$, for $1 \leqslant k \leqslant n$, can be computed in $O(n)$ time.*

4.3 Proof of Lemma 3

Proof. According to Lemma 4, $d_{i+1,j}$ is either $\min_{t_k \in Q} d_{i,k}$, $|w_j^- - t_r|$, or $|w_j^- - t_\ell|$. For each j, $\min_{t_k \in Q_j} d_{i,k}$ is a range minimum query on the array $[\, d_{i1}\ d_{i2} \cdots d_{in} \,]$ with queried interval Q_j. The queried intervals can be computed in $O(n)$ time

by Lemmata 5 and 6. Once the queried intervals are determined, the time complexity is $O(n)$ ($O(n)$-time for RMQ preprocessing and constant time for each query). Thus, the overall time complexity is $O(n)$. □

Theorem 2. *L_∞-APSPM can be solved in $O(mn)$ time and $O(n)$ space.*

5 Concluding Remarks

In this paper, two generalized PSPM problems, L_p-APSPM and L_∞-APSPM, are investigated, and algorithms with time complexity $O(mnT(p))$ and $O(mn)$ are proposed, respectively. The results match the currently best time bound of PSPM. For future work, we would like to further extend the problem such that mismatch of the pattern is allowed under some criteria.

References

1. Aggarwal, A., Klawe, M.M., Moran, S., Shor, P., Wilber, R.: Geometric applications of a matrix-searching algorithm. Algorithmica 2, 195–208 (1987)
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
3. Cardoze, D.E., Schulman, L.J.: Pattern matching for spatial point sets. In: FOCS 1998, pp. 156–165 (1998)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Intorduction to Algorithms, 2nd edn. MIT Press, Cambridge
5. de Rezende, P.J., Lee, D.T.: Point set pattern matching in d-dimensions. Algorithmica 13, 387–404 (1995)
6. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
7. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Communications of the ACM 18, 341–343 (1975)
8. Lipsky, O., Porat, E.: Approximate matching in the L_∞ metric. Information Processing Letters 105, 138–140 (2008)
9. Lipsky, O., Porat, E.: L_1 pattern matching lower bound. Information Processing Letters 105, 141–143 (2008)
10. Lipsky, O., Porat, E.: Approximate pattern matching with the L_1, L_2, and L_∞ metrics. Algorithmica 55, 212–223 (2009)
11. Mäkinen, V.: Using edit distance in point-patter matching. In: SPIRE 2001, pp. 153–161 (2001)
12. Mäkinen, V., Ukkonen, E.: Local similarity based point-pattern matching. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 115–132. Springer, Heidelberg (2002)
13. Mäkinen, V.: Parameterized approximate string matching and local-similarity-based point-pattern matching. Department of Computer Science, University of Helsinki, Report A-2003-6 (August 2003)
14. Suga, T., Shimozono, S.: Approximate point set pattern matching on sequences and planes. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 89–101. Springer, Heidelberg (2004)
15. Gavrilov, M., Indyk, P., Motwani, R., Venkatasubramanian, S.: Geometric pattern matching: A performance study. In: SoCG 1999, pp. 79–85 (1999)

Detecting Health Events on the Social Web to Enable Epidemic Intelligence

Marco Fisichella[1], Avaré Stewart[1], Alfredo Cuzzocrea[2], and Kerstin Denecke[1]

[1] Forschungszentrum L3S, Hannover 30167, Germany
{fisichella,stewart,denecke}@L3S.de
[2] ICAR-CNR and University of Calabria, Italy
cuzzocrea@si.deis.unical.it

Abstract. Content analysis and clustering of natural language documents becomes crucial in various domains, even in public health. Recent pandemics such as Swine Flu have caused concern for public health officials. Given the ever increasing pace at which infectious diseases can spread globally, officials must be prepared to react sooner and with greater epidemic intelligence gathering capabilities. Information should be gathered from a broader range of sources, including the Web which in turn requires more robust processing capabilities. To address this limitation, in this paper, we propose a new approach to detect public health events in an unsupervised manner. We address the problems associated with adapting an unsupervised learner to the medical domain and in doing so, propose an approach which combines aspects from different feature-based event detection methods. We evaluate our approach with a real world dataset with respect to the quality of article clusters. Our results show that we are able to achieve a precision of 62% and a recall of 75% evaluated using manually annotated, real-world data.

Keywords: Retrospective medical event detection, Clustering, Epidemic Intelligence.

1 Introduction

A public health event (PHE) is defined as a specific infection, disease or death that happens at a specific time and place, which may be consecutively reported by many medical articles in a period. Events such as emerging infectious diseases, are those considered to be either completely new or reoccurring. An important strategy used by officials to mitigate the impact of potential threats, is to find ways to detect the signs of a public health event as early as possible. The body of work devoted to this effort is known as event-based Epidemic Intelligence (EI) [16]. In order to provide information as timely as possible, by now all stages of event-based EI system, including document collection, filtering and processing, are done with little, or no human intervention. Unstructured and informal text of Web documents are used as data source to detect facts about current infectious disease activity within a population [3] .

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 87–103, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Existing event-based EI systems rely upon the enumeration of possible types of medical reporting patterns (e.g., MediSys [20]). This presents a huge limitation, since given the variety of natural language, many patterns may be required, and the recall for identifying relevant events can be low. Other systems rely on predefined keywords for identifying relevant information about public health events. In both cases, the algorithms are not robust enough for the task of detecting emerging public health events, since the only threat indicators (e.g., keywords) they can detect are those that are explicitly under surveillance.

One way to overcome the aforementioned limitations, is to cast a new light on the task of public health event detection - so that it is done in an unsupervised manner. In this area, there are two major approaches: those based on predictive event detection [12,19] which mine representative features, given an event; and those that detect an event from a list of highly correlated, bursty features [5,7]. The advantages of the former are that they are based on a generative model and have been shown to provide a more unified framework. They are capable of incorporating multiple modalities such as time stamps as well as explicit representations for various types of entities (i.e., features). A drawback is, however, that no prior burst analysis is done on these representations. In contrast, approaches based on the correlation of bursty features can filter out a vast number of potentially irrelevant features, yet many types of features relevant for public health event detection (i.e., symptoms, victims or medical conditions) are not modeled.

Proposed Solution. We hypothesize that applying an unsupervised algorithm to public health event detection will help to overcome the limitations of existing EI systems. To justify this hypothesis, we adapt an unsupervised approach to our problem domain in a way that events can be detected that are rare (aperiodic); reoccuring (periodic); and domain or task-specific. In more detail, we combine burst function analysis with the entity-centric feature representation in a generative model for predictive event detection. Going beyond a random initialization of the probabilities in this generative process, we instead exploit a known distribution of the features that are obtained directly from the burst function. Additionally, in our burst analysis, we refine the approach to feature representation by incorporating a Cauchy-Lorentz distribution to more closely model the true behavior of periodic, non-burst (trough) activity.

The remainder of this paper is organized as follows: works related to event detection are discussed in Section 2. Section 3 presents details of our approach, characterizing the nature of event detection in the public health domain, to lay the foundation for describing the task-specific adaptations required in this setting. Experimental results for our approach are given in Section 4. Finally, in Section 5, we conclude and outline future work.

2 Related Work

Event-Based Epidemic Intelligence. Numerous systems exist to detect public health events, notably, PULS [17] uses information extraction technology to

identify the disease and the location of a reported event. PULS is integrated into MedISys [17], which automatically collects articles concerning public health in various languages from news, and aggregates the extracted facts according to pre-defined categories, in a multi-lingual manner.

Proteus-BIO [6] automatically extracts infectious disease outbreak information from several sources including ProMed-mail [13], the World Health Organization (WHO)[1] and medical news web sites. EpiSpider [10] extracts publication dates, locations and topics from ProMed-mail reports and news, converting them to a semantic interchange format suitable for storage in a relational database.

BioCaster [2] is an ontology-based system which uses a domain-specific event ontology to perform named entity recognition on outbreak reports. The system analyzes documents reported from over 1,700 RSS web feeds[2], classifies them for topical relevance and plots them onto a Google map using geo-code information. BioCaster's Text Mining process is based on rules that are capable of matching a number of elements, including entity classes, skip-words, entity types, or regular expression patterns.

In general, the advantages of event extraction based on text mining process and information extraction are that they are: deterministic, produce clear granularity results (sentence level), produce explicit and well structured unit of information, and the outputs are easily interpretable. Further, event extraction allows a system to specifically tune learners to detect specific type of information and capture linguistics and semantic relations.

In contrast to these systems, we propose an approach that exploits unsupervised machine learning technology. None of the existing approaches consider the use of such approach. We believe that an unsupervised approach can complement existing systems since it allows to identify public health event (PHE) even if no matching keywords or linguistic patterns can be found.

Unsupervised Event-Detection. In the data mining community, the event detection task is divided into two categories: retrospective and new event detection, in either on-line or off-line mode [19]. Retrospective event detection refers to the detection of previously unidentified events from an accumulated historical collection. New event detection refers to the discovery of the onset of new events, either from live feeds in real-time (online model) or under a closed-world assumption.

Two main approaches have been considered to solve the problem of event detection, namely: document-based [1,12,19] or feature-based [5,7]. In document-based approaches, event detection is done by clustering documents (also named articles) based on semantics and time stamps using a generative model of documents. In feature-based approaches the temporal and document distributions of words are first studied and events are discovered using distributions of the features over the time, namely trajectory. In order to address the requirements of Epidemic Intelligence, we incorporate aspects from both the trajectory and

[1] http://www.who.int/csr/don/en/index.html

[2] http://en.wikipedia.org/wiki/RSS

generative model approaches. For the trajectory approach, we extend the way that periodic feature bursts are identified by using Cauchy-Lorentz distribution. In contrast to feature trajectory, generative modeling has been shown to provide a more unified framework incorporating multiple modalities such as time stamp of an article and its content. In the generative model, events are latent variables and articles are observations. Latent variables (as opposed to observable variables), are not directly observed, but are rather inferred through the model from some representation of the article's content that is observable and directly measured. Moreover, medical articles exhibit a similar behavior as other type of data: the documents themselves trigger events; and the document counts of an event are changed with time. Moreover, several events could be overlapping in time. Probabilistic models which use the Expectation Maximization (EM) algorithm have been used to capture this kind of behavior [4], and we follow this approach for the same reasons.

Finally, much of the work done in this area has focused on different ways to represent the features of a document and we follow a similar direction, adapting the representation to the public health domain [7,12]. In addition, we refine the generative model for features described in [12] by modeling the features based on trajectory distributions that have been computed from the dataset. Comparison results between [12] and our approach are shown in Section 4.3.

3 Unsupervised Public Health Event Detection

The goal of this work is to introduce an approach to detect PHE in an unsupervised manner. In a typical epidemic investigation task, public health officials must detect anomalous behavior. They periodically compute statistics about disease reporting events, using the recent past, in order to build a predictive model for the near future. The model is used as a baseline for detecting any anomalies. These statistics are based on aggregated information, which in our case, is derived from detecting events in an unsupervised manner from documents.

We consider a three stage process for detecting unsupervised public health events:

1. In the first stage, **Named Entity Feature Representation**, we build entity-centric document surrogates that are suitable for the medical domain. The manner in which we extract features and represent documents is outlined in Section 3.1.
2. We then perform **Feature Analysis** on the extracted set of features to prune the less relevant ones. Details are reported in Section 3.2.
3. The resulting set of features is then used as input for the **Unsupervised Public Health Event Detection** stage. Section 3.3 spells out how the detection is conducted.

Finally, to perform at the end an epidemic investigation where officials detect anomalous behaviors, relations between detected events need to be aggregated. This task will be considered in future work.

3.1 Named Entity Feature Representation

As a first step, we process raw text to build an entity-centric feature representation of the document. Given a collection of text documents, we define a finite set of articles, $\mathcal{A}$ as well as a Health Event Template, $\mathcal{T}$. The template $\mathcal{T}$ represents a set of feature types, which are important for describing public health events. More specifically, we describe a public health event by four attributes that provide information on *who* (victims) was infected by *what* (diseases), *where* (locations) and *when* (time, defined as the period between the first relevant article and the last relevant one). Thus, the template is instantiated as:

$$\mathcal{T} =< Victim, Disease, Location, Time > .$$

Instances of the template elements are represented as $< v, d, l, t >$, for victim, disease, location and time, respectively. We model the content and time with different types of models.

Content. The content of each document is represented by a bag-of-words whose type is given by the health event template. For each document, a vector is created for each of the feature types and each entry in the vector corresponds to the frequency with which a feature, of a given type, appears in the bag-of-words representation.

Time. Each event corresponds to a peak on an article count versus time distribution. In other words, the distribution is a mixture of many distributions of events. A peak is usually modeled by a Gaussian function, where the mean is the position of the peak and the variance is the duration of event. As a result, a $GaussianMixtureModel$ is chosen to model times. A medical article also can be represented by victims, diseases, locations and time (represented by a discrete value, i.e., the timestamp).

As a result, we describe a PHE and a medical article using the following tuples:

$$PHE = \{victims, diseases, locations, time(Period)\}$$

$$article = \{victims, diseases, locations, time(Timestamp)\}$$

In order to simplify our model, we assume the four kinds of information of the i-th medical article, given a PHE e_j, are conditional independent. Thus, the probability of an article is given by the product of the following individual probabilities:

$$p(article_i|e_j) = p(victims_i|e_j)p(diseases_i|e_j)p(locations_i|e_j)p(time_i|e_j)$$

3.2 Feature Analysis

In our work, we posit that Epidemic Intelligence involves a dual task: the detection of periodic as well as aperiodic events. In our domain, periodic events need to be determined in order to build statistical models for reoccurring infectious

diseases, or to track the changes in the prevalence level for an outbreak once it has occurred. With respect to a window of one year, for example, aperiodic events are also important, since they can represent a public health event that is annual (season flu) or quite severe and life threatening, such as a sudden outbreak of Ebola. Given the nature of public health event detection, it is important to be able to model both types of events for EI.

Spectral analysis is a common technique for identifying periodic and aperiodic features [7]. In this approach features are classified with respect to their periodicity (P_w) and their dominant power spectrum (S_w). The periodicity of a feature refers to its frequency of appearances. If the feature is *aperiodic*, then it occurs once within the period P, and its P_w has a value equal to the period itself. If the feature is *periodic*, then it happens regularly with a fixed periodicity, i.e., $P_w \leq \lceil P/2 \rceil$. The periodicity is a function of the dominant power spectrum which is computed via the discrete Fourier transform applied to the feature distributions, for more details refer to [7].

The dominant power spectrum, S_w, of a feature w is a strong indicator of its activeness at the specified frequency; the higher the S_w, the more likely it is for the feature to be relevant within the dataset. Thus S_w can be considered to filter out irrelevant features, i.e., features with a dominant power spectrum less than a prefixed threshold chosen according to the domain. After filtering out irrelevant features, the remaining features are meaningful and could potentially be representative for some events.

Identifying Burst for Aperiodic Features. Let $y_w(t)$ be the distribution of the feature w over the time t of the period under observation; further, let $y_w(t)$ be computed as in [7]. Then, for each *aperiodic* feature, we keep only the bursty period, which is modeled by a Gaussian distribution.

$$f_{ap}(y_w(t)) = \frac{1}{\sqrt{2\pi\sigma_w^2}} * e^{-\frac{1}{2\sigma_w^2}(y_w(t)-\mu_w)^2} \tag{1}$$

The well known Expectation Maximization (EM) algorithm is used to compute the Gaussian density parameters μ_k and σ_k [4].

Identifying Bursts for Periodic Features. To model the periodic features we chose a mixture of K Cauchy-Lorentz distributions, where $K = \lfloor P/P_w \rfloor$. Such a distribution is similar to the Gaussian, but differs in the thickness of its tails. This property, as observed from the computed $y_w(t)$, reflects better the distribution of *periodic* features, since, even for t far from the peak of the burst (non burst or trough activity), generally the feature distribution $y_w(t)$ reports a value that is important to be considered. The mixture is described as follows:

$$f_p(y_w(t)) = \sum_{k=1}^{K} \alpha_k * \frac{1}{\pi}\left[\frac{\gamma}{(y_w(t)-\mu_w)+\gamma^2}\right] \tag{2}$$

for the mixture proportions α_k of assigning y_w into the kth Cauchy-Lorentz distribution.

$$0 \leq \alpha_k \leq 1 \text{where} \sum_{k=1}^{K} \alpha_k = 1, \forall k \in [1, K] \subset \mathbb{N} \quad (3)$$

Furthermore, μ_w is the location parameter, specifying where is the peak of the distribution, and γ is the scale parameter which specifies the half-width at half-maximum. μ, γ and α are computed using the EM algorithm [4].

Feature Burst Distributions Algorithm. In this section, we present the algorithm for computing the feature burst distributions. Algorithm 1 wraps together the concepts and the approaches explained so far. The novelty of this approach is to represent periodic features with the Cauchy-Lorentz distribution. Finally, using the feature burst distributions for having a more representative model has been proved to be successful.

3.3 Detecting Public Health Events

A core step, in the unsupervised detection of events is the clustering of the articles and generation of events. Formally, from this stage we get the following sets of conditional probabilities that are shown in Table 1. We use these probabilities, as a basis for determining that a public health event has occurred, or is currently occurring.

Table 1. Probabilities obtained from unsupervised PHE detection

Probability	*Description*
$P_{a\|e}$:	Set of conditional probabilities for an article, a, given an event, e
$P_{w\|e}$:	Set of conditional probabilities for a feature, w, given an event, e
P_e :	Set of probabilities for an event, e

Approach. In this work, we choose to apply a retrospective event detection algorithm since it is important in EI to use data historical collection, in order to build a predictive model of public health events for the near future. The same idea is used in statistical methods used in public health to analyze event data from indicator-based systems (e.g., the Farrington Algorithm). Additionally, we have chosen a probabilistic generative model for event detection, because it has been proven to be a more unified framework for handling the multiple modalities (i.e., time, content, entity types) of an article and its content.

For these reasons, we base our unsupervised event detection algorithm on the Retrospective Event Detection (RED) algorithm presented by Li et al. [12]. It relies on a generative model where the articles are produced using multinomial distributions over the feature types. These articles are used later as starting points for a clustering relying on the iterative EM algorithm. In addition, in their work, the multinomial distributions are initialized with random probabilities. Thus, the generated articles are randomly picked.

Algorithm 1. Feature Analysis using feature burst distributions

```
Input: A set of extracted features W; a set of articles A; a fixed threshold τ;
Output: all the feature burst distributions
begin
    N := count the number of articles within A ;
    D := count the number of distinct date t in A ;
    P[|W|] := array storing the dominant period P_w for each feature w ;
    S[|W|] := array storing the dominant power spectrum S_w for each feature w ;
    FeatureDistributions[|W|][D] :=Matrix storing the vectors of feature distributions y_w for each
    feature w over the dates t ;
    FourierFeatureDistributions[|W|][D] := Matrix storing the decomposition of the vectors of feature
    distributions y_w into the sequence of complex vectors via the discrete Fourier transform DFT ;
    for each distinct date t in A do
        N(t) := count the number of documents at t ;
    for each feature w in W do
        DF_f := count the total number of documents containing entity w ;
        for each distinct date t in A do
            DF_f(t) := count the number of documents containing feature w at date t ;
            DF − IDF := (DF_f(t) / N(t)) * log(N / DF_f) ;
            Store DF − IDF into FeatureDistributions[w][t] ;
    Compute FourierFeatureDistributions using DFT on FeatureDistributions ;
    for each feature w in W do
        P[w] := compute the dominant period P_w of the corresponding feature ;
        S[w] := compute the dominant power spectrum S_w of the corresponding feature ;
        if S[w] ≥ τ then
            if P[w] > ⌈P/2⌉ then
                model the feature burst by a Gaussian distribution (aperiodic feature) ;
            else
                model the feature burst by a mixture of K = ⌊P/P_w⌋ Cauchy-Lorentz distributions
                (periodic feature) ;
end
```

As part of our approach, we refine the RED algorithm by going beyond this random initialization of probabilities - exploiting the feature distributions from our Feature Analysis stage. The underlying intuition for our approach is based on proven results, which show that an initial starting point estimated in a better-than-random way can, in fact, be expected to speed up the iterative EM algorithm converging closer to the optimum of the computed log lokelihood of a collection of articles, than an initial point that is picked at random. For more details refer to [21]. In our approach, we aggregate the computed feature distributions over the articles, and use this information into the multinomial distributions of the generative model. Thus, the generated articles, used as starting points by EM algorithm, are not totally randomly picked.

Although it has been proven that retrieved events are not influenced by the starting points [8,18], the EM algorithm needs to be restarted several times with several different random starting points in order to get a good approximation of events. Supported by the analysis in [21], we do not need multiple restarts of the EM algorithm, since an initial starting point estimated in this way, can be expected to be closer to the optimum than a randomly picked initial point.

Events in the unsupervised event detection are latent variables, whose value is defined with respect to the observed content of articles by a generative process. An event is defined by a pattern of entity-centric features that co-occur with such saliency, that an unlabled, real-world event, can be inferred, with some probability from the observable content of the articles that contain mentions of these features. Since the set of articles that describe the same event contain similar

sets of feature co-occurances, the articles themselves cluster and are described by the conditional probability of an article, given an event, $p(a_i|e_j, \theta^j)$ where the vector θ^j are mixing proportions, or the priors of events.

Generative Model for Public Health Events. Our generative model is described in Algorithm 2.

Algorithm 2. Detection of Public Health Events: the generative model

```
begin
    Choose an event e_j ~ Multinomial(θ_j);
    Generate a medical article a_i ~ p(a_i|e_j);
    Draw a time stamp time_i ~ N(μ_j, σ_j);
    for each feature of it, according to the type of current feature do
        Choose a victim_iv ~ Multinomial(θ^j_p|time_i);
        Choose a disease_id ~ Multinomial(θ^j_d|time_i);
        Choose a location_il ~ Multinomial(θ^j_l|time_i);
end
```

In the algorithm, the vector θ_j represents *event* probabilities initially instantiated randomly (here the definition of *event* is according to the formalization of the multinomial distribution); μ_j and σ_j are parameters of the conditional Gaussian distribution given event e_j; θ^j_p, θ^j_d, θ^j_l are vectors of probabilities computed by aggregating the feature burst distributions over the $time_i$ of a given event e_j.

4 Experiments and Evaluations

In order to analyze the results of the introduced method, we ran several experiments. For the specific task considered here which is public health event detection, no annotated data set is available. Anyway, we performed some analysis on a real-world data set. In this section, the data set used for the experiments is introduced together with the experimental settings and results.

Dataset. To build our data set, we collected source documents from the *url* column of the PULS online fact base [17], a state-of-the-art event-based system for Epidemic Intelligence which provides public health event summarization and search capabilities. The data were collected for a four month period, from September 1 - December 31, 2009, by crawling the website. In total 1,397 documents were collected. The data were processed by stripping all boilerplate and markup code using the method introduced by Kohlschütter et al. [11].

The reduced dataset size is due to the unavailability of standard evaluation data sets for public health event detection, thus a ground truth was manually built as explained in Section 4.3. Nevertheless, our dataset size is comparable with other datasets reported in relevant works [9,12,14,15].

Feature Set. In the experiments, the algorithm is run on a feature set consisting of named entities. Table 2 presents the main categories of features collected and their counts. The entities have been extracted using two different named entity recognition tools: UMLS MetaMap[3] and OpenCalais[4].

OpenCalais was used to recognize *medical conditions* and all variants of *location*. MetaMap was used to identify the *victim* features. MetaMap has originally been developed for indexing biomedical literature and relies upon the UMLS Metathesaurus, a very rich biomedical vocabulary. Thus, it allows extracting highly domain-specific concepts, but leads when applied to social media or news articles to false positives. For our feature set we are only interested in disease names and symptoms which are more reliable detected by OpenCalais. In contrast, the more detailed information on victims provided by MetaMap is very useful for our algorithm. For these reasons, we decided to exploit these two different named entity recognition tools.

Through manual inspection, we further found that noise introduced into the algorithm due to multi-word expressions causes an explosion of the number of features. This is particularly acute for a feature-centric approach such as ours in the medical domain, in which features consisting of many multi-word expressions quite commonly exacerbate the problem of producing irrelevant events. For this reason, we normalized the features by the use of a taxonomy. More specifically, the numerous and more specific concepts that are lower in the taxonomy are re-represented by a parent concept. For this we rely upon the taxonomy underlying the UMLS semantic network. As an example, the terms *boy, girl, baby, child, kid* were normalized to the single feature, *child*.

Table 2. Overview on the features collected. *norm* is the number of normalized features; *unnorm* is the number of unnormalized features.

Feature Types	**Feature Categories**	*norm*	*unnorm*
Victims	Population Group, Age Group, Family Group, Animal	28	4100
Diseases	Medical Condition	917	2754
Locations	City, ProvinceOrState, Country, Continent	955	982

4.1 Experiment I: Feature Pruning

Objectives: The features described before were analyzed by computing for each feature the periodicity, P_w, and their dominant power spectrum, S_w. The objective of this analysis was to prune the less important features, i.e., those that are potentially not representative to some event. The pruning was driven by looking at the S_w since it represents the relevance of features within the dataset. As claimed in [7], setting a threshold over S_w to identify which features to discard

[3] http://mmtx.nlm.nih.gov/

[4] http://www.opencalais.com

is more of an art than science and it is domain specific. We further provide some examples of pruned and kept extracted periodic and aperiodic features.

Results. In Figure 1, we report the values of the dominant periods, P_w, over the dominant power spectrums, S_w, for each feature extracted (indicated as a point in the diagram). In the graph one can notice an empty area between $P_w = 62$ and $P_w = 123$. According to the definition of periodicity, the bounds of this empty area correspond respectively to $\lceil P/2 \rceil$ and P, where P is the period under observation.

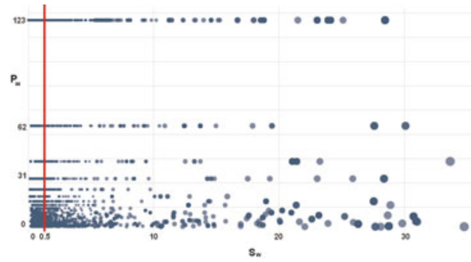


Fig. 1. Dominant period and dominant power spectrum of the features

Looking at the distributions of all the *feature* points over the graph, we chose to set the threshold τ over S_w to 0.5 (τ is specified in Algorithm 1). In Figure 1, the pruned features are those points to the left of the vertical line put at $S_w = 0.5$.

4.2 Experiment II: Selection of k

Objectives. A key consideration in a retrospective event detection is the determination of the number of events k to use as input for the generative model. The choice of the number of events can affect the interpretability of the results. For example, a solution with too few events will generally result in very broad events. A solution with too many events will result in un-interpretable events that pick out idiosyncratic feature combinations. We used a hill-climbing approach to discover the number of events as done in [12]. The objective of this experiment is to determine the value of k for which the algorithm performs best.

Results. We ran our method ten times, each time setting a different input value for the number of events k. Table 3 presents the *F1-measure* values computed for different number of events. It can be seen that the best response from our approach was achieved with $k = 15$, which correspond to the 50% of all peaks on the articles count-time distribution. Based on this result, we chose $k = 15$ for the manual assessment presented in Experiments III and IV.

Table 3. F1-measure over different number of events k

k	10	**15**	20	25	50	100
F1-measure	0.35	**0.68**	0.33	0.34	0.33	0.41

Table 4. Results for three clustering approaches

	P	R	F
PULS	40%	31%	34.9%
RED	40.6%	51.5%	45.4%
UPHED	**62.5%**	**74.8%**	**68.1%**

4.3 Experiment III: Cluster Quality

Objectives. An underlying intuition in our approach is that an unsupervised event detection algorithm which is tuned to detect public health events, would produce better quality clusters with respect to the Epidemic Intelligence task, than either a public health event extraction system based on information extraction or a generic event detection algorithm. In order to verify this assumption, we evaluated the quality of three clustering techniques. For the first clustering, *PULS* was used standing for a public health event extraction system based on information extraction. For the second clustering, the approach described in [12] was adopted as a generic event detection algorithm. This clustering considers time, locations, persons, as well as general keywords as features in the algorithms; we use *RED* to identify it. Finally, the third clustering was created using our approach, namely the Unsupervised Public Health Event Detection, *UPHED*. Due to the unavailability of standard evaluation data sets for public health event detection, a ground truth was manually built. To construct the ground truth, each document was manually examined by three subjects, who were instructed to assign each document to one of 15 clusters. Each document examined contained only a subset of the sentences - specifically, those containing at least one of the relevant named entities of medical condition, location and person. Each named entity was highlighted in the sentence. Discrepancies among the subjects were resolved by a fourth subject, based on mutual agreement. The manually built clusters were then used as ground truth to evaluate the precision, recall and f1-measure for the set of clusters that were created by the algorithm.

Results. Table 4 shows the results for precision (P), recall (R) and f1-measure (F). As it can be seen, *UPHED* performs better than the other two ones; thus, incorporating the medical conditions as entity type in fact, and integrating the burst function analysis in the generative model, helps to improve the clustering performance with respect to *RED*. Moreover, it has to be noticed that RED technique drops with reference to the results shown in [12]; this can be justified stating that RED works well in general domain, but less when adopting a particular domain as the Epidemic Intelligence. Regarding the clustering from *PULS*, although the achieved results seem relatively low, an important difference should be noted. There is a difference in vocabulary (feature set) due to the named entity extraction process used in PULS and our work. Although, we sought to use the same number of features, as present in the PULS fact base (by adjusting the minimum document frequency) - the distributions we obtained for the features still differ. For example, the locations used in PULS are exclusively at the granularity of the country, whereas those extracted from the corresponding raw PULS text, in our work, contain multiple geographical levels, such as: cities, providences, states, etc. Second, for the medical conditions we encountered a similar context: terms caught by UPHED (e.g., coughing, vomiting, disorder, etc) were not present in the PULS fact base, since they focused on actual disease mentions. We believe that accounting for the difference in the feature set, would improve PULS results. Additionally a standardized data set, for the field

Table 5. Detected aperiodic (E_1 - E_{12}) and periodic events (E_{13} - E_{15}). Columns respectively show extracted tags, number of documents and brief description of the real events.

Event Id	Event Terms	No. Docs	Event Description
E_1	wales, united kingdom, swine flu, flu, people, children	28	In November 2009, there were a couple of new swine flu cases occurring in Wales.
E_2	china, beijing, flu, swine flu, people, female	47	In October 2009, China had an increased number of swine flu cases. Further, the population was vaccinated to a large extent.
E_3	New York, united states, flu, swine flu, children, people	71	In November 2009, the flu death toll increased significantly in U.S.
E_4	bangalore, india, flu, infection, people, children	36	In September 2009, the swine flu toll in Bangalore increased.
E_5	japan, tokyo, disease, swine flu, people, children	44	In October 2009, Japan started with swine flu vaccinations.
E_6	france, europe, disorder, flu, people, female	32	In September 2009, an increased number of swine flu cases were reported in France.
E_7	surrey, london, e.coli, diarrhea, children, animals	50	In September 2009, there was an outbreak of E. coli in England. Mainly children were concerned who visited the same farm in Surrey.
E_8	manila, malaysia, disease, infection, people, father	57	In October 2009 there was an outbreak of leptospirosis in Manila, the capital of Philippines. Further, there was a typhoon which led to an increase of several infectious diseases which interested also the close Malaysia.
E_9	africa, Kenya, cholera, diarrhea, female, people	44	In December 2009 a deadly outbreak of cholera in north-western Kenya took place.
E_{10}	delhi, united states, flu, dengue, children, people	44	In November 2009 there was an outbreak of Dengue in Delhi. Contemporaneously, in United States several outbreaks of swine flu happened.
E_{11}	china, asia, disease, rabies, animals, people	30	In September 2009, China's health officials became aware that rabies had become one of the biggest public health risks facing China.
E_{12}	united states, finland, cancer, chronic fatigue syndrome, people, animals	15	An american researcher found out that a virus linked to prostate cancer may also be linked to Chronic Fatigue Syndrome. The same research was conducted by a finnish study (October 2009).
E_{13}	nigeria, south africa, disease, infection, children, people	31	Several studies found out that babies and children in Africa die from infections (September 2009). Further, there was a measles campaign in South Africa (October 2009). **Period**: every 7 days.
E_{14}	united states, canada, vomiting, swine flu, people, children	51	President Obama stated the fight against swine flu (October 2009). Also, several outbreaks of swine flu in U.S. and Canada happened (October and November 2009). **Period**: every 4 days.
E_{15}	united states, russia, swine flu, disease, people, female	24	Several news articles provide comparison of swine flu statistics for various countries, comparing mainly cases happening in Russia and U.S. **Period**: every 4 days.

would also allow for a strong comparison of the approaches. Finally, we noticed that the magnitude of our results does not exceed a value of 75% for recall. An explanation is that manually assigning documents to clusters is a very difficult task.

4.4 Experiment IV: Detected Public Health Events

Objectives. In another experiment, we manually analyzed the clusters together with the documents assigned to them and tried to describe the underlying event in own words or even to find an official information reporting the event. Further, we separated periodic from aperiodic events. The objective of this analysis is to provide a human interpretation of the clusters and have a critical analysis on them. The separation between aperiodic and periodic events provides us information about the earlier re-occurring of some medical event within our time window.

Results. Table 5 shows the characterizing cluster terms resulted from a clustering by the algorithm for a value of k of 15. The *Event Terms* shown in this table were presented to the testers in the manual assessment described in the section before. For each feature type, the two most probable features have been

selected for being shown. In addition, the number of documents assigned to the single clusters is shown, as well as a manually created description of the event happened that is reflected by most of the documents in the cluster. Evidently, most of them are very relevant events that express - of course - the global event happening in 2009 which is *swine flu*. For the events reporting such a disease, an outbreak was also detected by Google Flu Trends[5]. Nevertheless, some of the discovered events refer to some other diseases.

It can be seen that sometimes the extracted medical entity used to describe the cluster is the general term *disease*. This mainly happens when different diseases were described in the single documents. In the documents, the disease itself is often mentioned only a few times and often replaced by the more general term *disease*. For this reason, the calculated probability of the term *disease* became higher than for the more specific medical conditions and this term has been chosen as cluster describing term by the algorithm.

Some cluster labels reflect the content of the documents quite well - the terms seem to be consistent with the reported event (e.g., cluster E_1). For other clusters, one might get the impression that two different events are described when looking at the terms. For example, cluster E_{10}, where a subset of the documents assigned to this cluster deals with dengue in Delhi and another subset of documents refers to swine flu cases in United States. It can also be seen that the clusters are somehow overlapping. For example the event related to *swine flu* in *U.S.* is reflected by at least two clusters E_3 and E_{14}.

For some of the detected events we could even find official press releases from health organizations through manual assessment. For example, the event described by cluster E_7 which refers to an outbreak of E.coli in England can be confirmed by a press release of the Health Protection Agency on September 13, 2009[6]. The terms selected for describing the cluster reflect very well this event: Children were infected by E.coli after a visit of a farm in Surrey. Further, we separated and reported aperiodic (E_1 - E_{12}) from periodic events (E_{13} - E_{15}).

In summary, from this manual assessment we learned that documents reporting a similar or the same event are clustered together by the algorithm. The event terms that are selected based on their probability to describe the content of the clusters reflect the content of the documents quite well. They even reflect when events are assigned to the same cluster, but describe different events.

4.5 Experiment V: Efficiency Comparison

Objectives. In the last experiment, we compare three strategies for *Detecting Events*:

1. The baseline of our method, which initializes the EM algorithm with random points, as done in [12] and adapted to the medical domain. We use *Rdm* to represent it.

[5] http://www.google.org/flutrends/

[6] http://www.hpa.org.uk/NewsCentre/NationalPressReleases/2009PressReleases/

2. An approximation of our method identifying bursts for periodic features using a mixture of $K = \lfloor P/P_w \rfloor$ Gaussians, as suggested in [7]. We use *GaussApp* to identify it.
3. Our proposed method, namely the *UPHED*.

The intention of this analysis is to show that the selection of a good starting point can boost the EM algorithm to converge quickly to the optimum and that it is unnecessary to restart the EM algorithm multiple times with different random starting point.

Table 6. Efficiency comparison of different strategies

	Rdm	*GaussApp*	*UPHED*
Optimum (log-likelihood)	-3648	-3543	**-3497**
Best Starting Point (log-likelihood)	-3929	-3704	**-3665**
Average running time (seconds)	18.40	12.91	**11.32**
Average number of iterations for *EM*	10	7	**5**
Best trial: number of iterations for *EM*	7	**4**	**4**
Worst Trial: number of iterations for *EM*	14	10	**8**
Average number of restart of *EM* to get the optimum	5-6	**1-2**	**1-2**

Results. All the methods from the three strategies compared were ran on a machine with an Intel T2500 2GHz processor, 2GB memory under OS Fedora 9. The algorithms were all implemented in Java compiled by JDK 6.0. The experimental results are shown in Table 6. Here, we report the log likelihood both for the optimum and for the best starting points of the three strategies. The log likelihood indicates how likely the documents are generated by models, so it is the bigger the better. Then, we relate the average running time, and since this can be affected by implementation preferences, we describe the number of iterations of the *EM* algorithm to converge to the optimum in the best case, in the worst case, and on average. Finally, we show the number of restarts the *EM* algorithm needs for converging to the optimum, since the presence of local maxima can mislead the algorithm to reach the global maximum. From the results, we can conclude that in all the measures reported, our proposed method *UPHED* performs more efficiently than the other two ones. This also assists the conclusion that *UPHED* is much easier to get to the optimum than starting from a random point, thus needs less time and iterations to converge.

5 Conclusions and Future Work

In this paper, we presented an approach to public health event detection within the context of Epidemic Intelligence. More specifically, an unsupervised algorithm for detecting events from the data mining community was adapted to address the specific problems of public health event detection. The adaptations included the consideration of domain specific features that allow to detect only

domain-specific events. Further, a burst function analysis and the entity-centric feature representation were combined in a generative model that is the basis of the algorithm. The model was refined for representing periodic, non-burst features with the Cauchy-Lorentz distribution. The evaluations showed that better sampling is reached by such distribution which resulted also in better efficiency of the algorithm. For the task of detecting events, our approach achieved good quality results for a precision of 62% and a recall 75% on manually annotated data.

As future work, in order to simplify our model, we assumed the four kinds of information (victims, diseases, locations, and time) of the i-th medical article, given a $PHE\ e_j$, are conditional independent. This statement will be corroborated or confuted with further analysis. Of particular interest will be discovering if *disease* and *time* are conditional independent or not. Exploiting the results achieved, we will update our approach.

Acknowledgement. This work was funded, in part, by the European Commission Seventh Framework Programme (FP7/2007-2013) under grant agreement No.247829.

References

1. Allan, J., Papka, R., Lavrenko, V.: On-line new event detection and tracking. In: SIGIR 1998: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 37–45. ACM, New York (1998)
2. Collier, N., Doan, S., Kawazeo, A., Goodwin, R.M., Conway, M., Tateno, Y., Ngo, Q.H., Dien, D., Kawtrakul, A., Takeuchi, K., Shigematsu, M., Taniguchi, K.: Biocaster: detecting public health rumors with a web-based text mining system (2008), http://research.nii.ac.jp/~collier/research/publications.date.html
3. Hartley, D., et al.: The of international event-based biosurveillance. Emerging Health Threats (2009), http://www.eht-forum.org/ehtj/journal/v3/full/ehtj10003a.html
4. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society 39(1), 1–38 (1977)
5. Fung, G.P.C., Yu, J.X., Yu, P.S., Lu, H.: Parameter free bursty events detection in text streams. In: VLDB 2005: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 181–192. VLDB Endowment (2005)
6. Grishman, R., Huttunen, S., Yangarber, R.: Information extraction for enhanced access to disease outbreak reports. J. of Biomedical Informatics 35(4), 236–246 (2002), http://portal.acm.org/citation.cfm?id=827080
7. He, Q., Chang, K., Lim, E.P.: Analyzing feature trajectories for event detection. In: SIGIR, pp. 207–214 (2007)
8. Hofmann, T.: Probabilistic latent semantic analysis. In: UAI, pp. 289–296 (1999)
9. Kawamae, N.: Latent interest-topic model: finding the causal relationships behind dyadic data. In: CIKM, pp. 649–658 (2010)
10. Keller, M., Blench, M., Tolentino, H., et al.: Use of unstructured event-based reports for global infectious disease surveillance 15(5) (May 2009)

11. Kohlschütter, C., Fankhauser, P., Nejdl, W.: Boilerplate detection using shallow text features. In: Davison, B.D., Suel, T., Craswell, N., Liu, B. (eds.) WSDM, pp. 441–450. ACM, New York (2010), http://dblp.uni-trier.de/db/conf/wsdm/wsdm2010.html#KohlschutterFN10
12. Li, Z., Wang, B., Li, M., Ma, W.Y.: A probabilistic model for retrospective news event detection. In: SIGIR, pp. 106–113 (2005)
13. Madoff, L.C.: Promed-mail: An early warning system for emerging disease 2(39), 227–232 (July 2004)
14. Ming, Z., Wang, K., Chua, T.S.: Prototype hierarchy based clustering for the categorization and navigation of web collections. In: SIGIR, pp. 2–9 (2010)
15. Nallapati, R., Ahmed, A., Xing, E.P., Cohen, W.W.: Joint latent topic models for text and citations. In: KDD, pp. 542–550 (2008)
16. Paquet, C., Coulombier, D., Kaiser, R., Ciotti, M.: Epidemic intelligence: a new framework for strengthening disease surveillance in Europe. Euro Surveillence 11(12), 212–214 (2006), http://www.ncbi.nlm.nih.gov/pubmed/17370970
17. Steinberger, R., Fuart, F., van der Groot, E., Best, C., von Etter, P., Yangarber, R.: Text mining from the web for medical intelligence. Mining Massive Data Sets for Security 19, 295–310 (2008)
18. Steyvers, M., Griffiths, T.: Probabilistic Topic Models. Lawrence Erlbaum Associates, Mahwah (2007)
19. Yang, Y., Pierce, T., Carbonell, J.: A study of retrospective and on-line event detection. In: SIGIR 1998: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 28–36. ACM, New York (1998)
20. Yangarber, R.: Verification of facts across document boundaries. In: Proceedings International Workshop on Intelligent Information Access (2006)
21. Zhang, D., Zhai, C., Han, J., Srivastava, A., Oza, N.: Topic modeling for olap on multidimensional text databases: topic cube and its applications. Stat. Anal. Data Min. 2(5-6), 378–395 (2009)

A Learned Approach for Ranking News in Real-Time Using the Blogosphere

Richard McCreadie, Craig Macdonald, and Iadh Ounis

School of Computing Science
University of Glasgow, G12 8QQ, UK
{richardm,craigm,ounis}@dcs.gla.ac.uk

Abstract. Newspaper websites and news aggregators rank news stories by their newsworthiness in real-time for display to the user. Recent work has shown that news stories can be ranked automatically in a retrospective manner based upon related discussion within the blogosphere. However, it is as yet undetermined whether blogs are sufficiently fresh to rank stories in real-time. In this paper, we propose a novel learning to rank framework which leverages current blog posts to rank news stories in a real-time manner. We evaluate our proposed learning framework within the context of the TREC Blog track top stories identification task. Our results show that, indeed, the blogosphere can be leveraged for the real-time ranking of news, including for unpredictable events. Our approach improves upon state-of-the-art story ranking approaches, outperforming both the best TREC 2009/2010 systems and its single best performing feature.

1 Introduction

Large quantities of fresh news content from e-news providers are being continually published each day [1]. Meanwhile, millions of users consult e-newspapers and news aggregators to find out the most interesting events and stories occurring worldwide [1]. However, the volume and rate at which news content is currently created, highlights the need for automatic means to sort through this large volume of news in real-time, identifying the most currently newsworthy stories for display. This task can be seen as a ranking problem. For example, on the homepage of a news website, current news stories are ranked by their perceived newsworthiness at that time. Highly newsworthy stories receive prominent placement on the page, while lesser stories are displayed less prominently or not at all.

Recent work examining the automatic ranking of news stories has indicated that related blogging activity can be used as an indicator of story newsworthiness [4,14]. Indeed, the blogosphere is well known as a medium for news reporting and discussion [13,20,21]. Relatedly, almost 20% of searches to a blog search engine were reported to be news-related [17]. This shows that the blogosphere is likely to be a good source of information regarding current news.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 104–116, 2011.
© Springer-Verlag Berlin Heidelberg 2011

News stories can be roughly classified into those resulting from predictable and unpredictable events [2]. Of interest is that only predictable events, exhibit elevated levels of blog posting activity beforehand [14]. For this reason, the majority of previous models for news story ranking have focused on the retrospective ranking of news, i.e. at a later point in time [12]. It is not clear whether the blogosphere will remain an effective source of evidence for ranking news stories when moving to real-time setting. In particular, there may be as yet insufficient blog posts to accurately estimate newsworthiness for stories relating to unpredictable events.

In this paper, we investigate the extent to which it is possible to automatically rank news stories in real-time using the blogosphere. In particular, we propose a novel learning to rank (LTR) [8] approach for this task. LTR techniques are machine learning algorithms which take as input a set of *features* about each object to be ranked, i.e. a story in this case. They learn a weight for each feature, so that by combining weighted features a better overall ranking is produced than when ranking by any single feature alone. In this work, we aim to define suitable features which indicate the current newsworthiness of each story, allowing us to produce an accurate ranking of top news stories in real-time.

The advantages of an LTR approach to this problem in comparison to existing story ranking strategies are two-fold. Firstly, LTR provides a principled means for combining multiple sources of timely story ranking evidence as features. Secondly, LTR is extensible, hence should a new possible feature become available, e.g. the number of clicks on a specific story, then this can be easily integrated. Moreover, to our best knowledge, our framework is the first application of LTR for news story ranking using the blogosphere.

Existing story ranking strategies estimate the newsworthiness of a story based upon an aggregate of recent blog posts. Building upon recent work in the field of applying learning to rank techniques to aggregate problems [11], we propose a novel approach, which leverages existing story ranking approaches as features for use with learning to rank. In particular, we consider each feature to be comprised of a *story ranking component* that estimates a story's newsworthiness and a *temporal component* that specifies for which period of time newsworthiness should be estimated.

We evaluate the proposed learning to rank approach within the context of the TREC top news stories identification task. Our experiments examine the value that the blogosphere can bring to real-time news story ranking. The results show that our approach is effective at ranking news stories in real-time, including those relating to unpredictable events. Indeed, it markedly improves upon the best TREC 2009 and TREC 2010 system performances.

The remainder of the paper is structured as follows. In Section 2, we discuss prior work in the field of news article ranking. Section 3 describes our proposed learning to rank approach. Section 4 describes our experimental setup including corpora used and training/testing details. In Section 5, we report the performance of our LTR approach in comparison to the best TREC systems and discuss the most effective features. We provide concluding remarks in Section 6.

2 News Story Ranking

The Blog track at the Text REtrieval Conference (TREC) examined how news story ranking could be achieved in an automatic manner using evidence from the blogosphere [12]. In particular, the *top news stories identification task* examined whether the blogosphere could be used to identify the most newsworthy stories for a given day [12]. Participants were provided with a large number of news stories from the period of 2008 and had to rank those stories for a fixed set of topic days using only evidence extracted from the Blogs08 corpus - a 28.5 million blog post sample of the blogosphere [9]. Notably, the top news stories identification task was run during both TREC 2009 and TREC 2010. The 2009 task focused on a retrospective setting, i.e. participants were ranking the news stories at a later point in time, while the 2010 task simulated a real-time setting.

Various strategies to retrospectively measure the newsworthiness of a news story using the blogosphere have been proposed. For example, Mejova *et. al.* [15] use the number of 'citations', i.e. the number of blog posts linking to the news story, for ranking. However, this provided limited effectiveness due to the sparsity of links to each news story within the blogosphere. Lee *et. al.* [4] proposed a language modelling approach, whereby the likelihood of each story generating recent blog posts indicates the story's newsworthiness. This approach is more effective, as it avoids the sparsity problem by exploiting the textual similarity between a story and recent blog posts. Similarly, McCreadie *et. al.* [14] also exploited textual similarity between blog posts, proposing to model a story's newsworthiness as a voting process [10]. In particular, they retrieved a fixed number of blog posts related to the news story. Each blog post acts as a 'vote' for the story being newsworthy on the day that the blog post was published. The final score for a news story is the number of votes received for the day of the story.

For the real-time setting introduced in TREC 2010, similar strategies were proposed, however only blog posts published on the topic day or before can be used. For example, Xu *et. al.* [22] estimated the current newsworthiness of a story by summing the BM25 scores for each blog post that was published on the topic day for that story. Hence, only blog posts published on the same day as the story were considered. Similarly, Lin *et. al.* [6] built a vector-space story-to-blog-post representation, using only those blog posts from the story day. They estimated a story's news worthiness based upon the number of blog posts with a high cosine similarity to it.

Recall that for our proposed LTR approach, we define a set of story ranking features, each of which estimates the newsworthiness of a news story. We propose to use existing story ranking strategies, like those described above, as the basis for our story ranking features. In particular, these story ranking strategies act as the *story ranking component* of each of each feature.

However, it is of note that all of the above strategies, excepting that by Mejova *et. al.* [15], use a textual representation of the story. For instance, this could be the headline of an associated news article, or even the full content of such an article. Furthermore, prior work by McCreadie *et. al.* [14] indicated that

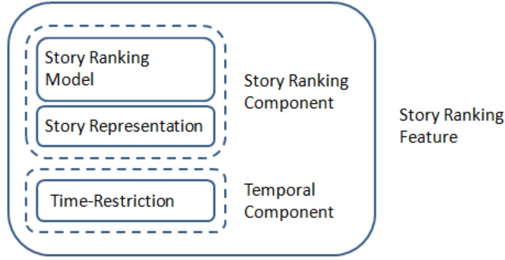


Fig. 1. Illustration of the components of a story ranking feature

by enhancing this representation, e.g. by enriching an article headline using query expansion, overall ranking performance could be improved. As such, we consider the story ranking component to be comprised of two sub-components: *the ranking model* and a *story representation*. Indeed, we experiment with eight different story representations in our subsequent experiments. An illustration of our feature components is shown in Figure 1. In the next section, we detail how these two sub-components are combined with a temporal component under our learning to rank approach to rank news stories in real-time.

3 Learning to Rank News Stories

We propose a new learning to rank approach to rank news stories in real-time. Learning to rank techniques are machine learning algorithms which take as input a set of document features and learn weights for each of those features within an information retrieval (IR) system [8]. The aim is to find the weighted linear combination of these features that results in the most effective document ranking.

Various learning to rank techniques have been proposed within the literature. These techniques fall into one of three categories. Point-wise techniques learn on a per-document basis, i.e. each document is considered independently. Pair-wise techniques optimise the number of pairs of documents correctly ranked. List-wise techniques optimise an information retrieval evaluation measure, like mean average precision, that considers the entire ranking list at one time [8]. Prior work has indicated that list-wise techniques learn more effective models [8]. As such, we use a list-wise learning to rank technique in this work. In particular, we use Metzler's Automatic Feature Selection algorithm (AFS) [16]. This is a greedy feature selection algorithm, which iteratively selects the feature that most improves retrieval performance. Notably, features that do not aid retrieval are not selected, i.e. they receive a weight of 0.

Traditional LTR techniques define features on the object that is to be ranked, i.e. the news story in this case. However, news story ranking is an aggregate ranking task, i.e. newsworthiness is defined in terms of a collection of related objects, i.e. blog posts relating to the story, rather than the news story itself. Inspired by prior work in the field of applying learning to rank techniques to aggregate

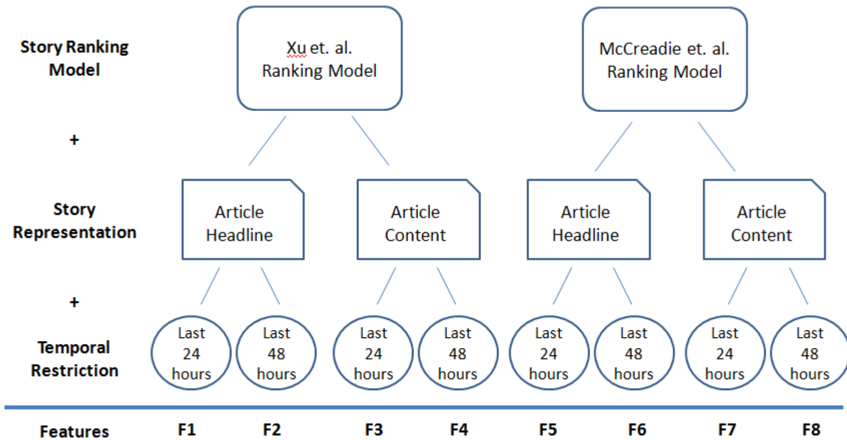


Fig. 2. Eight story ranking features generated from two story ranking models, two story representations (headline and content) and two time-restrictions (previous 24 and 48 hours)

problems [11], we propose an novel approach that generates LTR features by combining different components from multiple story ranking strategies.

In particular, under our LTR approach, a single feature is comprised of three components, a *ranking model*, *story representation* and *temporal restriction*, as illustrated previously in Figure 1. These components represent a real-time story ranking strategy in a generic manner, i.e. to rank a story, a story ranking model (ranking model) takes as input a textual representation of a story (story representation) to be ranked, and a collection of recent blog posts covering a fixed period of time (temporal restriction). For example, Xu *et. al.* [22]'s real-time approach estimates a story's newsworthiness by summing the BM25 scores for each blog post published during the last 24 hours using the headline from a related article as a query. Therefore, one possible example of a story ranking feature would be to combine the ranking model is that proposed by Xu *et. al.*, an article headline representation and blog posts from only the previous 24 hours. By varying the ranking model amongst those described in Section 2, using methods to enhance article headline or article content representations, and by considering more or less recent blog postings, we generate a large number of different features. Figure 2 illustrates this process.

In the next section, we describe our experimental setup for evaluating our learning to rank approach to real-time news story ranking and its individual features.

4 Experimental Setup

We evaluate our learning to rank approach within the context of the TREC 2009/2010 Blog track top news stories identification task. In particular, we rank

news stories from the period of January 2008 to February 2009 using evidence from the TREC Blogs08 corpus [9] which spans the same period. Notably, we rank news stories published by two different news providers, namely: the New York Times and Reuters. The New York Times corpus, denoted $NYT08$, was used during the TREC 2009 task [12], while the Reuters corpus, denoted $TRC2$, was used during TREC 2010 [5]. For each of these news corpora, for a set of 'topic days', stories published on those days were assessed in terms of their newsworthiness. We evaluate story rankings produced by our learning to rank approach, for both the 55 topic days from $NYT08$ and the 50 topic days from $TRC2$. Table 1 summarises the corpora used during our subsequent experiments.

Table 1. Statistics for the TREC corpora used during evaluation

Corpus	Quantity	Value
Blogs08	Time Range	14/01/08 → 10/02/09
	Number of blog posts	28,488,766
NYT08	Time Range	01/01/08 → 28/02/09
	# Stories	102,853
	Avg. Stories Per Day	264
	# Topic Days	55
	Avg. Headline Length	7
	Avg. Content Length	418
TRC2	Time Range	01/01/08 → 28/02/09
	# Stories	1,800,370
	Avg. Stories Per Day	4,628
	# Topic Days	50
	Avg. Headline Length	8
	Avg. Content Length	225

Of note is that the $TRC2$ news corpus provides both an article headline for each story, as well as the full article content, whilst the $NYT08$ corpus provides only the article headline. To make these corpora comparable, we independently crawled the missing article content for the $NYT08$ corpus, cleaning the resulting text with the BoilerPipe [3] article extractor. Furthermore, research has shown that using simple heuristics to prune away clearly unimportant stories from news corpora can have a positive impact on story ranking performance [14]. As such, we implement the following simple corpus pruning techniques. On both corpora, we reproduce the pattern, date and uppercase pruning heuristics suggested by McCreadie *et. al.* [14]. However, the editorial patterns that indicate non-newsworthy stories change from corpus to corpus. As such, for the new TRC2 corpus, we analysed three days' worth of news stories and propose an alternative pattern set[1]. We remove all news stories with article headlines starting with the named patterns.

[1] Patterns: "ADVISORY" "ANALYSIS" "BSE" "CBOT" "CHRONOLOGY" "CORRECTED" "CREDIT" "DIARY" "EUROPEAN" "Europe Daily Earnings" "FACTBOX" "FEATURE" "India call money" "INDICATORS" "INSTANT" "Japan Hot Stocks" "NASDAQ" "NSEI" "NYSE" "PRESS" "REFILE" "RESEARCH" "RPT" "SEALED" "SERVICE" "STOCKS" "TABLE" "TAKE" "TECHNICALS" "TEXT" "*TOP" "TRADING" "TREASURIES" "US STOCKS" "WORLD".

Using the aforementioned corpora, for all stories published on each of the 105 topic days spanning the two news corpora, we generate 160 story ranking features. These features are generated by combining a story ranking model, with a story representation and blog posts from a restricted period of time, as illustrated previously in Figure 2. In particular, we use two of the ranking models described earlier in Section 2, specifically the relevance-based model proposed by Xu *et. al.* [22], denoted *Relevance*, and the voting model proposed by McCreadie *et. al.* [14], denoted *Voting*. Any model-specific parameters are set as specified in the aforementioned papers. Furthermore, we use eight different story representations, four generated from the associated article headline for each story and four from the article content. Moreover, we vary the number of previous days of evidence that we make available using the temporal component. Specifically, we use up to the previous 10 days of published blog posts to rank news stories. Hence, the 2 ranking models, 8 story representations and 10 temporal restrictions multiply together to total 160 individual features. Table 2 lists each of the components which comprise these features and provides a short description.

Table 2. Feature components and sets for news story ranking

Component	Name	Description
Model	*Relevance*	Aggregated relevance-based story ranking model [22].
	Voting	Voting-based story ranking model [14].
Story Representation	*Headline*	The story headline.
	$QE_Blogs08$	The headline expanded using the Blogs06 blog post corpus [9].
	QE_NYT06	The headline expanded using 2000 news articles from the New York Times during May 2006.
	QE_TRC2	The headline expanded using 13 days of news stories from the TRC2 corpus but before the start of Blogs08 [5].
	Content	The article content.
	Entities	Named entities from the article content identified by a Wikipedia-based dictionary [18].
	$Noun - Phrases$	Noun Phrases extracted from the article content [19].
	Summary	Story summary generated using part-of-speech tagged article content [7].
Time Restriction	t_{Ndays}	Blog posts are available from the last N days, where $1 \leq N \leq 10$.

To train the weights for each of these features, we experiment with two different training regimes, namely Cross-Corpus and Per-Corpus. In particular, under Cross-Corpus training, we train using the topics from one corpus (either $NYT08$ or $TRC2$) and then test upon the topics from the other corpus and vice-versa. Under Per-Corpus training, we train and test on the same topic set using a 5-fold cross validation.

Due to slight differences in setting between the TREC 2009/2010 task formulations, we make the following changes to create a consistent setting and make cross-corpus training possible. Firstly, the TREC 2009 task ($NYT08$ topics) considered that stories both after and before each topic day might still be relevant due to differences in time-zone, which the 2010 task ($TRC2$ topics) did not. We follow the TREC 2010 setting and only rank the stories published on each topic

day. Secondly, the 2010 task introduced category classification of articles, i.e. each article was judged as to the degree to which it is important on the topic day with regard to one of five news categories. Importantly, these categories can introduce a confounding variable into the evaluation, as even a perfect article ranking system will be heavily penalised should it use a poor classifier. In this work, we focus on evaluating overall article ranking performance, and as such leave category classification for future work.

5 Results

In this section, we evaluate the performance of our learned solution and its component features for real-time news story ranking, in addition to examining the types of story (predictable vs unpredictable) that it favours. In particular, we evaluate the overall story ranking performance in Section 5.1. Section 5.2 examines the strongest features selected by our approach. In Section 5.3, we evaluate the importance of the three components of each story ranking feature used, while Section 5.4 investigates whether our approach overall is biased toward predictable events.

5.1 Story Ranking Performance

We begin by evaluating the overall story ranking performance of our approach in comparison to the TREC best system for each of the $NYT08$ (TREC 2009) and $TRC2$ (TREC 2010) topic sets. Table 3 reports the story ranking performance of the best TREC 2009 and 2010 systems as well as the performance of our best individual feature, in comparison to our learning to rank approach when trained under both Cross-Corpus and Per-Corpus regimes.

Table 3. Comparison between our learning to rank approach when trained under both Cross-Corpus and Per-Corpus training with the best TREC systems in terms of overall story ranking performance under the $NYT08$ (TREC 2009) and $TRC2$ (TREC 2010) story ranking topics. * denotes a statistically significant increase over the best individual feature (t-test $p < 0.05$).

Model	Training	$NYT08$ Topics (TREC 2009)	$TRC2$ Topics (TREC 2010)
TREC Best System	N/A	0.1862	0.1898
Best Individual Feature	N/A	0.1836	0.1949
Learned Model	Cross-Corpus	0.1165	0.1689
	Per-Corpus	**0.2042***	**0.2248***

We observe that under Cross-Corpus training, i.e. training on $NYT08$ (TREC 2009) and testing on $TRC2$ (TREC 2010), and vice versa, the performance of our approach is lower than the best TREC system. However, when moving to Per-Corpus training, i.e. a 5-fold cross validation, story ranking performance exceeds that of the best TREC system by 9% and 15% on the $NYT08$ and $TRC2$ topic

sets respectively. Moreover, the resulting trained model markedly outperforms the best individual feature used alone by a similar margin. This shows that our proposed learning to rank approach can indeed be effective for real-time story ranking (under Per-Corpus training).

The lesser performance when using Cross-Corpus training indicates that the best features for story ranking are different for the two news corpora and topic sets. This is somewhat to be expected, as the $NYT08$ and $TRC2$ corpora differ markedly in both the story writing style as well as the level of noise contained. In particular, as reported earlier in Table 1, Reuters ($TRC2$) published over 17 times as many stories during each day than the New York Times ($NYT08$), of which many are non-newsworthy stock reports. Furthermore, as a result of crawling and cleaning the article content for $NYT08$ ourselves, this content is likely noisier than pre-provided $TRC2$ article content. Hence, we would expect features based upon article content story representations to be less effective on the $NYT08$ topics and not to generalise between corpora. As such, in our further experiments, we report results using Per-Corpus training only.

5.2 Strongest Story Ranking Features

To examine our approach in more detail, we investigate which of the 160 features generated contribute most to the ranking of news stories. Table 4 reports the five strongest positive and negative features selected by our approach on each topic set.

Table 4. Strongest 5 positive and negative features on the $NYT08$ and $TRC2$ topic sets. Boldened feature weights indicate features with a high impact on the story ranking.

Feature	$NYT08$ Topics				$TRC2$ Topics			
Type	Components			Weight	Components			Weight
Positive	*Voting*	Headline	t_{1day}	**0.9703**	*Voting*	Headline	t_{1day}	**0.7719**
Positive	*Voting*	Summary	t_{1day}	**0.2762**	*Voting*	Content	t_{2days}	**0.1406**
Positive	*Voting*	Content	t_{1day}	**0.2646**	*Voting*	Noun-Phrases	t_{5days}	0.0468
Positive	*Relevance*	Summary	t_{2days}	**0.1213**	*Relevance*	Summary	t_{3days}	0.0196
Positive	*Voting*	QE_Blogs06	t_{3days}	0.0982	*Voting*	QE_TRC2	t_{3days}	0.0173
Negative	*Voting*	Entities	t_{8days}	-0.0063	*Voting*	Headline	t_{6days}	-0.0035
Negative	*Voting*	Content	t_{8days}	-0.0286	*Voting*	Content	t_{6days}	-0.0038
Negative	*Voting*	QE_TRC2	t_{7days}	**-0.1032**	*Voting*	QE_NYT06	t_{7days}	-0.0063
Negative	*Relevance*	Noun-Phrases	t_{1days}	**-0.1143**	*Voting*	Noun-Phrases	t_{10days}	-0.0077
Negative	*Voting*	Headline	t_{7days}	**-0.5215**	*Voting*	Summary	t_{6days}	-0.0101

In general, we observe that the $Voting$-based ranking model is preferred across both topic sets, indicating that it produces features better able to distinguish between newsworthy and non-newsworthy stories than the *Relevance*-based alternative. Of the eight story representations listed previously in Table 2, we see that the headline alone is the strongest story representation across topic sets. However, in contrary to our expectations, the content representations were also selected. This shows that although content is more noisy in the $NYT08$ corpus than its equivalent in $TRC2$, it appears to still provide valuable ranking

evidence. Indeed, it is of note that of the positive features selected using the $NYT08$ topics, a higher weight is assigned to the shortened summary of the content than the content unaltered. This indicates that the summarisation is removing noise from the content for $NYT08$ that is unnecessary for $TRC2$, although in contrast, the Noun-Phrase representation appears be noisy. In terms of the temporal restrictions that we place on the story ranking models for the real-time setting, we make the following two observations. Firstly, only features that use blog posts from the one or two days before the time of ranking appear to be useful. Secondly, as we relax the temporal restriction and use older blog posts, the story ranking features become negative, i.e. if a story has been discussed extensively beforehand then the story is less likely to be newsworthy. Indeed, for the $NYT08$ topics, the strongest positive feature ($Voting$ + Headline + t_{1day}) becomes the strongest negative feature by changing the temporal restriction. Notably, negative features appear not to add value on the $TRC2$ topics.

5.3 Story Ranking Components

We next examine whether the features generated by varying each of the story ranking components are useful. In particular, we follow a leave-one-out approach, whereby we discard any features generated by varying a given single component, leaving only a single instance of that component. In particular, for the story representation we keep only features using the article headline representation. Similarly, for the temporal component, keep only features that used blog posts 1 day old or less. For the ranking model we keep only those features generated by one or other of the two story ranking models considered.

Table 5 reports the story ranking performance of our learning to rank approach trained upon feature subsets. We see that by removing features generated by the $Voting$ model, the ranking performance markedly decreases. This confirms our earlier observation that the $Voting$-based model is more effective than the $Relevance$-based alternative. Indeed, we see that by removing $Relevance$-based features instead, little story ranking performance is lost.

Examining the story representations, performance decreases markedly on the $TRC2$ topics by discarding alternate representations, showing that indeed, story representations can have a strong impact on performance. However, unexpectedly, we see that by using the headline of the story alone, ranking performance

Table 5. Story ranking performance of our learning to rank approach in when training on different feature sets using Per-Corpus (5-fold cross validation) under the $NYT08$ (TREC 2009) and $TRC2$ (TREC 2010) story ranking topics

Voting Model	*Relevance* Model	Story Representations	Time Restrictions	*NYT08* Topics (TREC 2009)	*TRC2* Topics (TREC 2010)
✔	✔	✔	✔	0.2042	0.2248
✖	✔	✔	✔	0.1729	0.1130
✔	✖	✔	✔	0.2034	0.2026
✔	✔	✖	✔	**0.2120**	0.1900
✔	✔	✔	✖	0.1658	**0.2313**

is slightly increased on the $NYT08$ topics instead. The inability of the learner with all features to find this better solution highlights an issue with greedy learning to rank approaches. In particular, greedy learners, while effective, are not guaranteed to find the optimal solution and can be trapped in a local minima.

In terms of the temporal restrictions, we see that ranking performance is heavily degraded on the $NYT08$ topic set when only blog posts from the same day as the story (t_{1day}) are considered. On the other hand, story ranking performance on the $TRC2$ corpus is not negatively impacted, indeed performance gains are observed instead. This emphasises the differences in the $NYT08/TRC2$ topic sets. In particular, the performance gain observed on $TRC2$ indicates that the corpus contains a higher proportion of unpredictable events, i.e. those for which only very recent blog posts are relevant. Moreover, the different performances observed between the topic sets confirm our earlier observation that the learner on the $NYT08$ topics used older blog posting activity as negative features, while on the $TRC2$ topics it did not. Indeed, a key advantage that our approach has over the existing story ranking models that it employs as components, is the ability to adapt to different news corpora.

5.4 Predictable vs. Unpredictable Events

Lastly, we examine whether the blogosphere lacks sufficient freshness to accurately rank stories relating to unpredictable events in real-time. In particular, we select the top 5 most newsworthy stories as returned by our learned approach for each of the 50 topics in the $TRC2$ corpus, creating a set of 250 newsworthy stories. We manually annotated each of these as reporting about predictable or unpredictable events. Should the blogosphere lack sufficient freshness, then our story ranking approach will be more likely to identify predictable events over unpredictable ones, i.e. the vast majority of the 250 news stories would relate to predictable events.

However, in contrast, our results show that 46% of the top stories were unpredicable, while 54% were predicatable (a close to even spread). This indicates that, at least for the simulated real-time setting introduced by TREC, there is no evidence to indicate that bloggers react too slowly for unpredictable stories to be effectively ranked.

6 Conclusions

In this paper, we proposed a novel learning to rank approach which leverages current blog posts to rank news stories in a real-time manner. In particular, we used existing news story ranking models in conjunction with varying story representations and temporal restrictions to generate 160 story ranking features. We evaluated our proposed learning approach within the context of the TREC 2009 and 2010 Blog track top stories identification task. Our results show that the proposed approach is effective at ranking news stories in real-time. Indeed, it improves upon both the best TREC 2009 and TREC 2010 systems and its best

internal feature by over 9% and 13% respectively. Moreover, we examined both the individual features and story ranking components used by our learning to rank approach, highlighting those that were most useful in terms of impact on the story ranking. Lastly, we investigated whether our approach based upon measuring bloggers response to news stories, was biased toward predictable events, due to a lack of timely posting regarding unpredicatable events. However, we found that there was no evidence to indicate that this was the case, indeed 46% of top stories ranked by our system were related to unpredicatble events.

References

1. Newspaper Association of America (NAA): Newspaper Web sites attract more than 70 million visitors in June; over one-third of all Internet users visit newspaper Web sites (2010), http://www.naa.org/PressCenter/SearchPressReleases/2009/NEWSPAPER-WEB-SITES-ATTRACT-MORE-THAN-70-MILLION-VISITORS.aspx, (accessed on January 25, 2010)
2. Jones, R., Diaz, F.: Temporal profiles of queries. ACM Trans. Inf. Syst. 25(3), 14 (2007)
3. Kohlschütter, C., Fankhauser, P., Nejdl, W.: Boilerplate detection using shallow text features. In: Proceedings of WSDM 2010 (2010)
4. Lee, Y., Jung, H.y., Song, W., Lee, J.H.: Mining the blogosphere for top news stories identification. In: Proceeding of SIGIR 2010 (2010)
5. Leidner, J.L.: Thomson Reuters releases TRC2 news corpus through NIST (2010), http://jochenleidner.posterous.com/thomson-reuters-releases-research-collection (accessed on January 16, 2011)
6. Lin, Y.F., Wang, J.H., Lai, L.C., Kao, H.Y.: Top stories identification from blog to news in TREC 2010 Blog track. In: Proceedings of TREC 2010 (2010)
7. Lioma, C., Macdonald, C., Plachouras, V., Peng, J., He, B., Ounis, I.: University of Glasgow at TREC 2006: Experiments in Terabyte and Enterprise Tracks with Terrier. In: Proceedings of TREC 2006 (2006)
8. Liu, T.Y.: Learning to rank for Information Retrieval. Foundations and Trends® in Information Retrieval 3(3), 225–331 (2009)
9. Macdonald, C., Ounis, I.: The TREC Blogs06 collection: Creating and analysing a blog test collection. Tech report. Univ. of Glasgow
10. Macdonald, C.: The Voting Model for People Search. Ph.D. thesis, Univ. of Glasgow (2009)
11. Macdonald, C., Ounis, I.: Learning models for ranking aggregates. In: Clough, P., Foley, C., Gurrin, C., Jones, G.J.F., Kraaij, W., Lee, H., Mudoch, V. (eds.) ECIR 2011. LNCS, vol. 6611, pp. 517–529. Springer, Heidelberg (2011)
12. Macdonald, C., Soboroff, I., Ounis, I.: Overview of TREC-2009 Blog track. In: Proceedings of TREC 2009. NIST (2009)
13. Matheson, D.: Weblogs and the epistemology of the news: Some trends in online journalism. New Media and Society 6(4), 443–468 (2004)
14. McCreadie, R., Macdonald, C., Ounis, I.: News article ranking: Leveraging the wisdom of bloggers. In: Proceedings of RIAO 2010 (2010)
15. Mejova, Y., Ha Turc, V., Foster, S., Harris, C., Arens, B., Srinivasan, P.: TREC Blog and TREC Chem: A view from the corn fields. In: Proceedings of TREC 2009 (2009)

16. Metzler, D.A.: Automatic feature selection in the Markov random field model for Information Retrieval. In: Proceedings of CIKM 2007 (2007)
17. Mishne, G., de Rijke, M.: A study of blog search. In: Lalmas, M., MacFarlane, A., Rüger, S.M., Tombros, A., Tsikrika, T., Yavlinsky, A. (eds.) ECIR 2006. LNCS, vol. 3936, pp. 289–301. Springer, Heidelberg (2006)
18. Santos, R.L.T., Macdonald, C., Ounis, I.: Voting for related entities. In: Proceedings of RIAO 2010 (2010)
19. Schmid, H.: Treetagger. TC project at the Institute for Computational Linguistics of the University of Stuttgart (1994)
20. Sussman, M.: The state of the Blogosphere 2009 (2009), http://technorati.com/blogging/article/state-of-the-blogosphere-2009-introduction/ (accessed on May 13, 2010)
21. Thelwall, M.: Bloggers during the London attacks: Top information sources and topics. In: Proceedings of WWW 2006 Blog Workshop (2006)
22. Xu, X., Liu, Y., Xu, H., Yu, X., Peng, Z., Cheng, X., Xiao, L., Nie, S.: ICTNET at Blog track TREC 2010. In: Proceedings of TREC 2010 (2010)

Attribute Retrieval from Relational Web Tables

Arlind Kopliku, Karen Pinel-Sauvagnat, and Mohand Boughanem

IRIT, University of Toulouse, France
{Arlind.Kopliku,Karen.Sauvagnat,Mohand.Boughanem}@irit.fr

Abstract. In this paper, we propose an attribute retrieval approach which extracts and ranks attributes from HTML tables. Given an instance (e.g. Tower of Pisa), we want to retrieve from the Web its attributes (e.g. height, architect). Our approach uses HTML tables which are probably the largest source for attribute retrieval. Three recall oriented filters are applied over tables to check the following three properties: *(i)* is the table relational, *(ii)* has the table a header, and *(iii)* the conformity of its attributes and values. Candidate attributes are extracted from tables and ranked with a combination of relevance features. Our approach can be applied to all instances and is shown to have a high recall and a reasonable precision. Moreover, it outperforms state of the art techniques.

Keywords: information retrieval, attribute retrieval.

1 Introduction

Most information retrieval systems answer user queries with a list of documents, but there are many information needs that can be answered with information within documents. For instance, the query "features of Mac Book" is not asking for all pages speaking of Mac Books rather than for their features.

Inspired by the work in [5], we distinguish three types of information needs that can be answered differently:

- *query by attribute* (GDP of Italy, adress of Hotel Bellagio)
- *query by instance* (Samsung Galaxy, Italy, Oscar Wilde)
- *query by class* (Macintosh notebooks, British writers)

When the query is specifically asking for an attribute, we can return its value right away. When the query is an instance, we can propose a summary of salient attributes (properties). When the query is a class of instances, the result can be a comparative table of the class instances with their attributes. Figure 1 shows what these results might look like. A direct application can be Google Squared[1], a commerical tool that produces similar results. We will call search based on attributes, instances and classes *relational search*.

Relational search is not the only application where attributes play a crucial role. They can also be used for query suggestion [3], faceted search [4], or

[1] http://www.google.com/squared

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 117–128, 2011.
© Springer-Verlag Berlin Heidelberg 2011

aggregated search [11]. For instance, given the query "Indonesia" and using attributes of "Indonesia" we can produce suggestions such as "Indonesia climate", "Indonesia capital" and "Indonesia hotels", while in a faceted search context we can navigate returned results on the query "modern architecture" by attributes such as "country", "architecture style" and so on. Attributes can also be used to summarize content about an instance/class.

Query: *operating system of Samsung Galaxy* — Android

Query: *Tower of Pisa*

Location	Italy
Province	Tuscany
Height	55.8 m
Steps	296

Query: *Macintosh Notebooks*

	Mac Book	Mac Pro
Height	1.08 in	0.11-0.68 in
Width	13.00 in	11.8 in
Depth	9.12 in	7.56 in
Weight	4.7 pounds	2.3 pounds

Fig. 1. Examples of relational search results

In this paper, we treat attribute retrieval which falls in the above defined framework. More precisely, the research question we address is identifying relevant attributes for a given instance. We use HTML tables to extract attributes from. There are some main reasons behind this choice. First, tables are widespread across the Web. Second, many tables contain relational data [7]. Third, tables have already been used sucessfully for attribute extraction [9,8].

However, the task is not easy. Many of the HTML tables are used for layout design and navigation. In [7], authors estimate that only about 1% of the Web tables contain relational data. Furthermore, some relational tables have no headers (schema of attributes) and it is not easy to retrieve all relevant tables for a given instance.

We propose an attribute retrieval approach at Web scale which takes into account the above issues. First, we issue the instance as a query to a search engine. The retrieved documents are used to extract tables from. Successively, we apply 3 filters to candidate tables to check the following properties: *(i)* is the table relational, *(ii)* has the table a header, *(iii)* the conformity of its attributes and values. Extracted attributes are then ranked with relevance features.

Our approach integrates the work of Cafarella et al. [7]. Their work is at our knowledge the largest mining of Web tables for search purposes. They show we can filter out many tables that are not relational and that do not have a header. In addition to their work, we filter out many table columns (or rows) that do not contain attributes (name and values). We also integrate our previous work [12], which shows that we can retrieve attributes using a linear combination of relevance features such as table match, document relevance and external evidence from Web search, DBPedia and Wikipedia.

The paper is structured as follows. The next section is about related work. In section 3, we describe our approach including filtering and ranking of attributes. Then we describe the experimental setup (section 4) and results (section 5) following with conclusions.

2 Related Work

To acquire attributes from the Web, it is common to use HTML tags, decoration markup [21,19,10] and text [3,16,13]. Tags for tables, lists and emphasis have been shown to help for attribute extraction [21,19]. Wong et al. [19] combine tags and textual features in a Conditional Random Fields model to learn attribute extraction rules, but they need a seed of relevant documents manually fed.

Another common technique to acquire attributes is through the use of lexico-syntactic rules. For example, Pasca et al. [1,14] use rules such as “A of I” and “I’s A” to acquire attributes from query logs. Authors represent the class as a set of instances and multiple class instances are used to improve extraction. In [2], authors use more precise lexico-syntactic rules such as “the A of I is”, but recall of these rules is lower. In [15], Popescu et al. use lexico-syntactic rules to extract product attributes from reviews.

At last, tables are known to be a mine for relational data and attributes. Cafarella et al. [7,6] show we can identify billions of relational tables in the Web. In [9], Chen et al. identify attributes using column (or row) similarities. Another common technique to extract attribute from tables is through wrapper induction [8,10,18]. Given a training set or a set of similar documents, wrapper induction learns extraction rules. Many wrappers extract at record level, but they do not distinguish between attribute name and attribute value. Furthermore, wrappers are precision oriented and they work well only for some sites.

To summarize, current attribute acquisition techniques can obtain a high precision. Although many of these techniques produce a considerable number of attributes, they cannot cover the needs that can be answered with the Web. Most of them are conceived to work offline and they cannot extract instance attributes whatever the instance.

Our work is inspired by work in [1] and [6,7]. It differs from [1], in that we do not use lexico-syntactic rules but Web tables to identify attributes. It differs from the work in [6,7], in that we do not use tables for relation search. We make use of learnings in [6,7], but we introduce another filter at line level and we introduce relevance features. The combination of filters and relevance ranking allows us to enable attribute retrieval which was not investigated in [6,7].

3 Attribute Retrieval

We consider an information retrieval situation where the query is an instance and the results is a list of attributes. Concretely, **given an instance i, attribute retrieval should extract and rank attributes with respect to their relevance to i.**

We first collect a seed of potentially relevant tables. We issue the instance as a query to a search engine and we retrieve tables from the retrieved documents. We use these tables to extract attributes from. The problem is far away from being solved. Tables in the Web are quite heterogeneous and many of the retrieved tables are not relevant or they are partially relevant.

Before ranking attributes, we apply three filters on tables and attributes namely *relational filter*, *header filter*, and *attribute line filter*. The first two are the same as in [7]. They are recall oriented classifiers that can filter out many tables that are not relational and that do not have headers. Still, after applying these filters there remain many tables which are not relational. As well there are many tables which are almost relational such as table 2 in figure 2. Instead of filtering out this kind of tables, we introduce another filter at column (or row) level which we call *attribute line filter*. Each line is checked for its conformity for being an attribute line i.e. an attribute name followed with attribute values of similar format and length.

After applying the three filters, we rank the remaining attribute lines with respect to their relevance for the instance. The task is not easy. If we consider only tables that match the instance, we would lose many relevant tables(e.g. table 2 in figure 2 is relevant for “France”, but does not match it). To increase recall, we use all tables from relevant documents. Extracted attributes are ranked with relevance features as described in section 3.3.

	Population	Median age
France	64,768,389	39.7 years
Italy	58,090,681	43.7 years

Table 1

Facts	
Capital	Paris
Demonym	French
Population	
Population	64,768,389
Median age	39.7 years

Table 2

1	Loud	Rihanna
2	The king of limbs	Radiohead
3	Femme fatale	Britney Spears

Table 3

Fig. 2. Examples of interesting tables

3.1 Relational Tables and Headers

We build the relational filter and the header filter using the same features as done by Cafarella et al. in [7]. Features on the left of table 1 are used to learn a rule-based classifier for the relational filter and features on the right are used to learn a rule-based classifier for the header filter. Learning is done with a training set of human-classified tables described later in the experimental setup.

Features include the dimensions of the table (f_1, f_2), the fraction of lines with mostly nulls (empty cells) (f_3), the lines with non-string data (numbers, dates)

Table 1. Features for the relational and header filter

Relational Filter	Header Filter
f_1: ♯ rows	f_1: ♯ rows
f_2: ♯ cols	f_2: ♯ cols
f_3: %rows w/mostly NULLS	f_8: %head row cells with lower-case
f_4: ♯ cols w/non-string data	f_9: % head row cells with punctuation
f_5: cell strlen avg μ	f_{10}: % head row cells with non string data
f_6: cell strlen stddev σ	f_{11}: % cols w/non-string data in *body*
f_7: $\frac{\sigma}{\mu}$	f_{12}: % cols w/$\lvert len(row_1) - \mu\rvert > 2\sigma$
	f_{13} : % cols w/$\lvert\sigma < len(row_1) - \mu\rvert \leq 2\sigma$
	f_{14} : % cols w/$\lvert\sigma > len(row_1) - \mu\rvert$

(f_4), statistics on the character length of cells and their variation (f_5, f_6, f_7, f_{12}, f_{13}, f_{14}) and conformity of the header cells (lower-case, punctuation, non-string data) (f_8-f_{11}).

The main difference with the work of Cafarella et al. is that they consider that relational tables are all oriented vertically, i.e. the table header (if present) is on top of the table and the attribute lines are vertical (the columns). For example, tables 1 and 3 in figure 2 are oriented vertically and table 2 is oriented horizontally. We extend their approach to work for both horizontally and vertically oriented tables. This is not difficult. We consider the origin table t and another table $\overline{t}$ which is obtained from t considering its rows as columns and its columns as rows.

If t passes both the relational and header filter, table columns are considered as candidate attribute lines to extract attributes from. Similarly, if $\overline{t}$ passes both the relational and header filter, the table rows that are considered as candidate attribute lines. It can happen that both columns and rows are considered as candidate attribute lines. The latter are then passed to the attribute line filter.

3.2 Attribute Line Filter

The attribute (name and values) is extracted from a column of the table or a row of the table. Let a be the first cell of the line (row or column) and V be the rest of the cells of the line. A conform attribute line should contain an attribute name in the first cell and attribute values in the rest of the cells.

Typically, attribute names do not have much punctuation except of the colon and parenthesis. They rarely contain numbers. On the other hand, attribute values are usually in the same format (number, string, date) and their length is similar. Based on the above observations, we define the following features for the attribute line filter:

- presence of punctuation (except colon and brackets) in a
- presence of numbers in a; a is an English stop word
- length (char) of a; length (words) of a
- average and standard deviation of the length (char) of values: μ, σ
- $\sharp$values $v \in V$ with $|len(v) - \mu| > 2\sigma$
- $\sharp$values $v \in V$ with $|\sigma < len(v) - \mu| \leq 2\sigma$
- $\sharp$values $v \in V$ with $|\sigma > len(v) - \mu|$
- data conformity : $\frac{max_{T \in int, string, date}(\sharp values_of_type(T)}{|V|}$

These features are then used to learn a rule-based classifier from a training set of human classified attribute lines described later in the experimental setup. Once candidate attributes are filtered, we rank the remaining set as explained in the following section.

3.3 Relevance

It is not easy to tell whether an attribute is relevant for a given instance. There are many tables relevant to the instance where the instance is not even present in its cells. We propose combining different features to estimate a relevance score

for attributes. There features include a score on the match of the instance on the table, document relevance and external evidence from DBPedia, Wikipedia and Web search. We use a simple linear combination of these features to rank attributes. Each of the relevance features is described below.

Table match: Attributes should be extracted from tables that are relevant for the instance. A necessary condition but not sufficient for a table to be relevant is to be present in a relevant document for the instance. In some tables, the instance appears explicitly. Such tables might be better candidates for extraction. The table match feature will be described below. It is intended to measure at which extent the table matches the instance (query).

Let a be an attribute extracted from a table T for an instance i. The match of an instance within a table cell $T_{x,y}$ is measured with the cosine distance among the terms of the instance and the terms of the table cell. Let i and c be the vector representation of the instance and the table cell content. We have $i = (w_{1,i}, w_{2,i}, ..., w_{n,i})$ and $c = (w_{1,c}, w_{2,c}, ..., w_{n,c})$. Each dimension corresponds to a separate term. If a term occurs in the instance, its value is 1 in i. If it occurs in the table cell content, its value is 1 in c. The match is computed with the cosine similarity among the vectors.

The table match score is computed as the maximal match within table cells:

$$match(i, T) = \max_{T_{x,y} \in T} cos(i, T_{x,y})$$

However an attribute name is unlikely to be present in the same line (row or column) with the instance name, while the relevant attribute value is likely to appear in the same line with the instance. The latter can be also observed in table in figure 2. We will define the shadow area of a cell O as the set of cells in the same row and same column with O. But, there are some exceptions to this rule. We call headline cells, the ones that have are spanned in one line cell ($colspan > tablewidth$)[2] such as “Facts” and “Population” in table 2, figure 2. Headline cells usually act as titles that introduce parts of the table. We consider that the headline cells are not part of the shadow of another cell (see figure 3).

We define the match score of an attribute as the difference between the table match score and the shadow match score.

$$match(a, i, T) = match(i, T) - match(i, shadow(a))$$

where

$$match(i, shadow(a)) = \max_{T_{x,y} \in shadow(a)} cos(i, T_{x,y})$$

Document relevance (*drel*): If a document is relevant for the instance, the tables within the document are likely to be relevant for the instance. We should though take into account the document relevance. More precisely, let $\sharp results$ be the number of retrieved results for an instance i and $rank$ be the rank of a document d within this list. We compute $drel(d, i) = \frac{\sharp results - rank}{\sharp results}$.

[2] The colspan is an HTML attribute that defines the number of columns a cell should span.

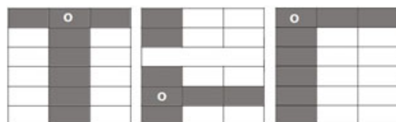


Fig. 3. The shadow for a cell **O**, 3 cases

Search hits: Search hits is a feature that has already been used for attribute extraction [21,15]. It corresponds to the number of search results returned by a Web search engine to a query "attribute of instance" within double quotes (successive ordering of terms is obligatory, e.g. "capital of France"). As done in literature, we use the logarithm (base 10) of the search hits count. To normalize, we used the above observation. Few attributes score higher than 6 i.e $log_{10}(search_hits_count(\text{a of i})) > 6$. All the attributes that score higher than 6 were given a score of 1, the other scores were normalized by 6. Doing so, we have all scores in the interval $[0, 1]$.

DBPedia feature: DBPedia represents a large ontology of information which partly relies on information extracted from Wikipedia. Given an instance i and an attribute a, the DBPedia feature is equal to 1 if a is found as an attribute of i in DBPedia.

Wikipedia feature: Although information in DBPedia is much more uniform, there exist many attributes in Wikipedia *infobox* tables which are not present in DBPedia. *Infobox* tables are available for many Wikipedia pages. They contain lists of attributes for the page. Given an instance i and a candidate attribute a, we set the Wikipedia feature to 1 if a can be found in the infobox of a page for i in Wikipedia.

4 Experimental Setup

Data set: To build our dataset, we first chose a set of classes and then 30 instances per class. To choose instances and classes, we use sampling to avoid biases. 5 participants had to write down 10 classes each. Classes could be broad (e.g. Countries) or specific (French speaking countries). We sampled 20 classes out of the 50. This is a reasonable amount as in state of the art approaches [16,21,3,20,13], the number of assessed classes varies from 2-25 classes.

Similarly for each class, we asked the 5 participants to write down 10 instances. Sampling and removing duplicates we obtained 10 instances per class. We do not apply any specific criteria for selecting instances.

This is the list of classes: *rock bands, laptops, american universities, hotels, software, british army generals, chancellors of German, American films, IR papers, SLR cameras, 2000 novels, Nirvana songs, Nissan vehicles, programmable calculators, countries, drugs, companies, cities, painters, mobile phones.* The entire dataset is in http://www.irit.fr/∼Arlind.Kopliku/FORCEdataset.txt .

General setup: For each instance of the dataset (200 instances), we retrieved top 50 search results using the Yahoo! BOSS API. These pages are used as a seed to extract tables and attributes. For each table, we apply the three filters. The remaining attribute lines are used as candidate attributes, which are ranked with the relevance features.

Learning filters: The three filters correspond to rule-based classifiers. They were trained using the previously mentioned features using the WEKA package [17]. From the extracted tables, we randomly selected a sample of 3000 tables (with more than 1 row and more than 1 column) which were judged by 3 assessors. For each table assessors had to tell, if the table is relational. If "yes" they had to tell if the table is oriented vertically or horizontally and whether the table has a header.

Similarly, we choose a sample of 3000 random attribute lines from our dataset of tables. They are as well assessed from our assessors. For each attribute line, the assessor has to tell if it is a conform attribute line i.e. it contains an attribute name and attribute values.

Similarly to [7], we cross-validated the trained classifiers by splitting the human-judged dataset into five parts. Each time four parts are used training and the fifth for testing. We trained five different classifiers, rotating the testing set each time. Performance numbers are averaged from the resulting five tests.

Ranking: Candidate attributes are ranked based on the relevance features. To measure the performance of ranking, the 30 top ranked attributes for each instance were assessed. 5 human participants shared this task assessing each a disjoint set of attributes.

Assessments were binary. An attribute is assigned 1 if it is *relevant* for the instance it was extracted for. It were assigned 0 otherwise. During evaluation, they could access the source page of the attribute or other sources (Web search, dictionary) to make their decision. Attributes were shown in alphabetical order to avoid any bias.

5 Evaluation Results

This section is about experimental results. First, we present the performance of the three filters. Then we analyze the performance of attribute retrieval.

5.1 Filtering

First, we analyzed all retrieved tables for all instances in our dataset. We found that only 16.9% of the tables had more than one row and more than one column. Within the 3000 tables that were assessed only 23% were considered relational. We can thus estimate a concentration of 3.9% relational tables in the retrieved Web pages. Now, we will analyze the effect of the filters. It is important to point out that our goal is not to compare with the results of Cafarella et al. [7], but to integrate their work and show its effect on attribute retrieval.

Relational filter: As we mentioned earlier, we learn separately a classifier for relational tables that are oriented horizontally and another classifier for relational tables that are oriented vertically. Results are shown in table 2 aside with the results obtained by Cafarella et al. [7].

Table 2. Precision and recall for the relational filter

Horizontal			Vertical			Cafarella		
	prec.	recall		prec.	recall		prec.	recall
yes	0.50	0.82	yes	0.38	0.81	yes	0.41	0.81
no	0.98	0.94	no	0.98	0.95	no	0.98	0.87

We tuned classification for high recall over positives. The performance of our classifiers is similar to Cafarella et al. The classifier of relational tables oriented horizontally retains 82% of the true positives and it filters out 94% of the true negatives. Similarly, the classifier of relational tables oriented vertically retains 81% of the true positives and it filters out 95% of the true negatives. After this filtering step, we will have about 45% of relational tables within our corpus out of an initial estimated concentration of about 3.9%.

Table 3. Precision and recall for the header filter in relational tables

Horizontal			Vertical			Cafarella		
	prec.	recall		prec.	recall		prec.	recall
yes	0.96	0.95	yes	0.76	0.89	yes	0.79	0.84
no	0.63	0.70	no	0.87	0.73	no	0.65	0.57

Header filter: In table 3, we show results for the header filter. Results are better than those of Cafarella et al. In particular, we found that most of the horizontally oriented relational tables have headers. They usually have two to three columns and are easier to classify.

The header classifier for relational tables oriented horizontally retains 95% of the true positives with a precision of 96%. Although, the header classifier of relational tables oriented vertically is less performant, it retains 89% of the true positives with a precision of 76%. After this filtering step, about 87.5% of the relational tables will have headers.

Table 4. Precision and recall for the attribute line filter

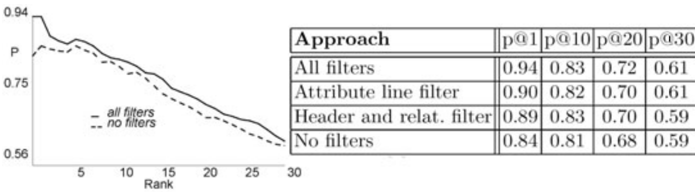
	prec.	recall
yes	0.56	0.95
no	0.69	0.55

Attribute line filter: As well as the other filters, the attribute line filter is tuned for recall over positives. Results are shown in table 4. This filter retains 95% of the correct attribute lines, while it filters out about 55% of the incorrect

attribute lines. It clearly helps in filtering out useless attribute lines at the cost of 5% of correct attribute lines.

5.2 Attribute Retrieval

Precision at rank: Attributes are ranked with a linear combination of the relevance features. Results are shown in figure 4. They are promising, especially when all three filters are applied. At rank 10, we have a precision of about 83%. At rank 20 we have a precision of about 72%. This means that if we apply this ranking for query suggestion, we will have that about 8.3 correct suggestions within top 10 suggestions and about 14.4 correct suggestions among the top 20.



Approach	p@1	p@10	p@20	p@30
All filters	0.94	0.83	0.72	0.61
Attribute line filter	0.90	0.82	0.70	0.61
Header and relat. filter	0.89	0.83	0.70	0.59
No filters	0.84	0.81	0.68	0.59

Fig. 4. Impact of filters

In figure 4, we can compare results for the ranking when no filter is applied and when all 3 filters are applied. Clearly, filters improve results. The table in figure 4 allows us to analyze the impact of each filter. We can see that the attribute line filter has a significant impact in the ranking as well as the relational and header filter.

Comparison with existing approaches: We first compared our approach to the one based on Lexico-syntactic rules [3,21,16,14] Lexico-syntactic rules are common for attribute extraction . We tested the lexico-syntactic extraction rules in our retrieval framework with our dataset. Concretely, we use the patterns "*A* of *I*" and "*I*'s *A*" as done in [14,16]. We collect candidate attributes for 10 instances of all classes. The top 50 search results for all instances are used as extraction seed for both techniques. The attributes extracted by the lexico-syntactic extraction method are ranked with the same scoring (excluding table match score which does not apply to lexico-syntactic rules).

Results are shown in table 5. Our method performs significantly better with 61% of relevant attributes at rank 30 against 33% for the lexico-syntactic rules.

We also compared both approach in term of recall, by considering only the 4 classes used to estimate recall. Lexico-syntactic rules have a lower recall, too.

Table 5. Comparison with lexico-syntactic scores

Approach	p@1	p@10	p@20	p@30
Our approach	0.94	0.83	0.72	0.61
Lexico-syntactic rules	0.46	0.48	0.43	0.33

For each class, lexico-syntactic rules identify 55 candidate attributes on average. Among these there are about 24 relevant attributes per class.

We can conclude that lexico-syntactic rules work well for certain applications such as queries, but they do not work well for long documents, especially in terms of precision.

Estimated recall: To estimate in a reasonable time the recall of our method, we randomly selected 4 classes namely “SLR cameras”, “countries”, “companies”, “Nissan vehicles”. For all instances of the class, our assessors evaluated all candidate attributes (the ones that are not filtered out).

We found an average of 918 distinct candidate attributes per class (SLR cameras 689, countries 1253, companies 804, vehicles 925). Among them, there are on average 256 relevant attributes per class (cameras 303, countries 213, companies 160, vehicles 347). This is a considerable amount of relevant attributes and it shows the potential of our method.

Comparison with DBPedia: We compared the recall of our approach with the one obtained using DBPedia, for the 4 classes mentioned above. To do so, we collected all Wikipedia pages related of the instances of each class. We then extracted attributes which are either present in DBPedia or in infoboxes from Wikipedia. We found an average of 25 distinct relevant attributes per class (cameras 8, countries 38, companies 37, vehicles 18). We can conclude that our approach has a high recall even if compared with quality and large sources such as DBPedia and Wikipedia.

6 Conclusions

We propose an attribute retrieval approach that can extract and rank attributes from HTML tables sparse in the Web. Our method is flexible and recall-oriented. Given an instance as a query, we retrieve attributes from top ranked search results. Then, we combine filtering and ranking to obtain a list of attributes ranked by relevance.

We combine a total three filters. Two of them are already known in literature. They are applied to HTML tables to filter out non relational tables and tables without header. In addition, we propose a third filter which is specific to attributes. All three filters are shown to have a high recall over povitives and they filter out a huge amount of useless data. The remaining data is candidate attributes which are ranked with relevance features. Our ranking algorithm combines document match, table match and other external evidence. The experimental setup shows that we can rank attributes with a reasonable precision and a high recall. Our approach outperforms lexico-syntactic rules and it provides a much larger quantity of attributes than quality sources such as DBPedia and Wikipedia (for our dataset).

For future work, we will investigate additional relevance features. We will focus more on attribute values and try combining different attribute acquisition techniques in a large-scale domain independent attribute retrieval framework.

References

1. Alfonseca, E., Pasca, M., Robledo-Arnuncio, E.: Acquisition of instance attributes via labeled and related instances. In: SIGIR 2010, pp. 58–65. ACM, New York (2010)
2. Almuhareb, A., Poesio, M.: Attribute-based and value-based clustering: An evaluation. In: EMNLP. ACL (2004)
3. Bellare, K., Talukdar, P.P., Kumaran, G., Pereira, O., Liberman, M., Mccallum, A., Dredze, M.: Lightlysupervised attribute extraction for web search. In: Proceedings of Machine Learning for Web Search Workshop, NIPS 2007 (2007)
4. Ben-Yitzhak, O., Golbandi, N., Har'El, N., Lempel, R., Neumann, A., Ofek-Koifman, S., Sheinwald, D., Shekita, E., Sznajder, B., Yogev, S.: Beyond basic faceted search. In: WSDM 2008, pp. 33–44. ACM, New York (2008)
5. Cafarella, M.J., Banko, M., Etzioni, O.: Relational Web Search. Technical report, University of Washington (2006)
6. Cafarella, M.J., Halevy, A., Wang, D.Z., Wu, E., Zhang, Y.: Webtables: exploring the power of tables on the web. Proc. VLDB Endow. 1(1), 538–549 (2008)
7. Cafarella, M.J., Halevy, A.Y., Zhang, Y., Wang, D.Z., Wu, E.: Uncovering the Relational Web. In: WebDB (2008)
8. Chang, C.-H., Kayed, M., Girgis, M.R., Shaalan, K.F.: A survey of web information extraction systems. IEEE Trans. on Knowl. and Data Eng. 18, 1411–1428 (2006)
9. Chen, H.-H., Tsai, S.-C., Tsai, J.-H.: Mining tables from large scale html texts. In: COLING 2000, USA, pp. 166–172 (2000)
10. Crescenzi, V., Mecca, G., Merialdo, P.: Roadrunner: Towards automatic data extraction from large web sites. In: VLDB 2001, USA, pp. 109–118 (2001)
11. Kopliku, A.: Aggregated search: From information nuggets to aggregated documents. In: CORIA RJCRI 2009, Toulon, France (2009)
12. Kopliku, A., Pinel-Sauvagnat, K., Boughanem, M.: Retrieving attributes using web tables. In: Joint Conference on Digital Libraries 2011, Ottawa, Canada (2011)
13. Pasca, M., Durme, B.V.: What you seek is what you get: Extraction of class attributes from query logs. In: IJCAI, pp. 2832–2837 (2007)
14. Pasca, M., Durme, B.V.: Weakly-supervised acquisition of open-domain classes and class attributes from web documents and query logs. In: ACL, pp. 19–27 (2008)
15. Popescu, A.-M., Etzioni, O.: Extracting product features and opinions from reviews. In: HLT 2005, pp. 339–346. ACL, Stroudsburg (2005)
16. Tokunaga, K., Kazama, J., Torisawa, K.: Automatic discovery of attribute words from web documents. In: Dale, R., Wong, K.-F., Su, J., Kwong, O.Y. (eds.) IJCNLP 2005. LNCS (LNAI), vol. 3651, pp. 106–118. Springer, Heidelberg (2005)
17. Witten, I.H., Frank, E.: Data mining: practical machine learning tools and techniques with java implementations. SIGMOD Rec. 31, 76–77 (2002)
18. Wong, T.-L., Lam, W.: A probabilistic approach for adapting information extraction wrappers and discovering new attributes. In: ICDM 2004, pp. 257–264. IEEE Computer Society, Washington, DC (2004)
19. Wong, T.-L., Lam, W.: An unsupervised method for joint information extraction and feature mining across different web sites. Data Knowl. Eng. 68, 107–125 (2009)
20. Wu, F., Hoffmann, R., Weld, D.S.: Information extraction from wikipedia: moving down the long tail. In: KDD 2008, pp. 731–739. ACM, New York (2008)
21. Yoshinaga, N., Torisawa, K.: Open-domain attribute-value acquisition from semistructured texts. In: Proceedings of the Workshop on Ontolex, pp. 55–66 (2007)

Query-Sets++: A Scalable Approach for Modeling Web Sites

Barbara Poblete[1,2], Myra Spiliopoulou[3], and Marcelo Mendoza[4]

[1] Department of Computer Science (DCC), University of Chile, Chile
[2] Yahoo! Research Latin-America, Chile
[3] Otto-von-Guericke-University Magdeburg, Germany
[4] Universidad Técnica Federico Santa María, Chile

Abstract. We explore an effective approach for modeling and classifying Web sites in the World Wide Web. The aim of this work is to classify Web sites using features which are independent of size, structure and vocabulary. We establish Web site similarity based on search engine query hits, which convey document relevance and utility in direct relation to users' needs and interests. To achieve this, we use a generic Web site representation scheme over different feature spaces, built upon query traffic to the site's documents. For this task we extend, in a non-trivial way, our prior work using query-sets for single document representation. We discuss why this previous methodology is not scalable for a large set of heterogeneous Web sites. We show that our models achieve very compact Web site representations. Furthermore, our experiments on site classification show excellent performance and quality/dimensionality trade-off. In particular, we sustain a reduction in the feature space to 5% of the size of the bag-of-words representation, while achieving 99% precision in our classification experiments on DMOZ.

Keywords: Web Sites, Query Mining, Classification.

1 Introduction

The fast expansion of the Web has made it increasingly important to find ways to extend the paradigm of Web Information Retrieval (IR) towards richer functionalities. In this work, we shift the focus of Web IR from the traditional retrieval of documents that satisfy a certain query, towards the retrieval of complete Web sites. Specifically, Web site retrieval can enhance search in several ways, for example: (1) retrieving sites that are relevant to a specific query, (2) retrieving sites that are similar to another given site, (3) grouping similar sites, and (4) building groups of sites, where each group is representative of a specific topic.

In particular, in this work we focus on organizing Web sites, which corresponds to applications (3) and (4). Specifically, we analyze Web site models which allow us to automatically classify sites into predetermined categories. Currently, this type of problem is solved through human-intensive initiatives such as the DMOZ[1] and Yahoo![2] directories, which, for a given taxonomy, manually find and assign Web sites that fit best each

[1] http://dmoz.org
[2] http://dir.yahoo.com

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 129–134, 2011.
© Springer-Verlag Berlin Heidelberg 2011

topic. Obviously, this approach does not scale well in a rapidly growing and heterogeneous environment like the Web; since it requires knowledge of all possible topics and regular inspection of all of the top Web sites for each topic.

We believe that by obtaining a suitable Web site model we will also have the basis for addressing applications (1) and (2). Our solution is based on the *wisdom of the crowds* paradigm. We consider *two Web sites as being similar if users access both of them to satisfy similar information needs*. This type of approach allows us to identify Web sites that are similar according to their *perceived information content*, independent of their structure, size, vocabulary and specific data.

Our contribution is threefold: First, we address the problem of classifying providing a solution that reflects how each Web site is perceived by users. Second, we show that our prior work [8], for creating compact document models using queries cannot be extended to large collections of Web sites in a trivial way. Third, we propose several feature spaces which are appropriate for representing Web sites on the Web.

2 Related Work

We make the distinction between *similar* and *duplicate* or *near-duplicate* document detection. Although there is extensive work in the latter area, it pursues a different goal than similarity research: the goal is to detect almost identical documents. In our work we focus on similarity research with the goal of finding Web sites that satisfy similar information needs from users, but are not necessarily alike in contents, extension or vocabulary.

There have been several efforts towards the automatic classification of Web sites. Ester et al. [3] propose to model Web sites as feature vectors. They consider each topic as a feature in a topic-based space, representing each Web site as a topic-based feature vector. Later, Kriegel and Schubert [6] applied the previous method for automatic maintenance of Web directories. They modified the method representing each topic by the centroid of the Web sites which belong to the topic. Lindemann and Littig [7] studied structural properties of Web sites, using them as features for Web site description. On the other hand, Rajalakshmi and Aravindan [9] present a soft computing approach for Web site classification. Their approach is based on features extracted from URLs, without fetching Web site contents.

3 General Concepts

Web site: A formal and unambiguous definition of *Web site* is an open problem [2]. Deciding how to automatically classify documents into Web sites is not the purpose of our work. Instead, we are interested in the task of organizing Web sites as wholes. Therefore, we adhere to a simple heuristic definition for *Web site* which considers as part of a Web site *all of the documents that appear under the same host name* [3].

Query-set mining: Query-sets are discovered by analyzing all of the search queries from which a document was clicked, to obtain groups of terms that are used together to reach the document. Formally, let $\mathcal{L}$ be a search engine query log and let $\mathcal{Q}$ be the set

of *distinct* queries registered in $\mathcal{L}$. Therefore, each query $q \in \mathcal{Q}$ can be repeated one or more times in $\mathcal{L}$. Then, for a given document d, we denote as $\mathcal{Q}(d)$ the set of distinct queries in $\mathcal{Q}$ that resulted in a request for d, and as $\mathcal{L}(d)$ the portion of $\mathcal{L}$ that contains clicks to d. Further, we denote as $QT(d)$ the set of query-terms used in $\mathcal{Q}(d)$.

4 Generic Web Site Vectorization

We introduce a simple scheme for modeling Web sites as vectors over an arbitrary feature space. We use an extension of the traditional *vector-space model* for documents, with the variation that a Web site vector is composed by the sum of the vectors of its documents. We use this for the creation of several query-based feature spaces.

Let $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ be a collection of documents and let $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ be the set of features that characterize those documents. Further, let $w_{i,j}$ be the *weight* associated to the pair (d_i, f_j). Then, the generic document vector for d_i is defined as $\overrightarrow{d_i} =< w_{i,1}, w_{i,2}, \ldots, w_{i,j}, \ldots, w_{i,m} >$. This vector is a generalization of the vector space document model, which incorporates an arbitrary m-dimensional feature space $\mathcal{F}$. In the traditional vector space representation, $\mathcal{F}$ corresponds to the set of terms in $\mathcal{D}$, and $w_{i,j}$ to the weight of the jth-term in the ith-document - given by the term frequency in d_i. Next, we create a representation for Web sites consisting of the aggregation of each site's document vectors. Let $SITES = \{s_1, s_2, \ldots, s_N\}$ be a set of Web sites and let $\mathcal{D}$ be the collection of all documents in $SITES$. Where $s_k \subseteq \mathcal{D}$ for $k = 1, \ldots, N$. Then, the vector representation of a Web site s_k built over the generic feature space $\mathcal{F}$ is: $\overrightarrow{s_k} =< c_{k,1}, c_{k,2}, \ldots, c_{k,j}, \ldots, c_{k,m} >$, where each $c_{k,j}$ corresponds to a weight associated to the pair (s_k, f_j) for $f_j \in \mathcal{F}$. The value of $c_{k,j}$ is the normalized counterpart of $w'_{k,j}$, according to the *tf-idf* scaling scheme proposed in [1,5]:

$$c_{k,j} = \left(0.5 + \frac{0.5 \cdot w'_{k,j}}{\max_{f_l \in \mathcal{F}}(w'_{k,l})}\right) \times \left(-log_2 \frac{n_j}{N}\right),$$

Where $w'_{k,j}$ is the sum of the weights of the documents in s_k for a given feature f_j: $w'_{k,j} = \sum_{d_i \in s_k} w_{i,j}$. and $\max_{f_l \in \mathcal{F}}(w'_{k,l})$ is the feature with the largest weight in s_k and n_j is the number of sites where f_j appears. In particular, this vectorization of a Web site $s_k \in SITES$ requires two parameters to be specified: 1) the *feature space* $\mathcal{F}$ over the documents constituting all the sites in $SITES$, and 2) the *weighting scheme* for the features over the documents.

5 Query-Based Feature Spaces for Web Sites

We define several variations of the feature space selection for Web sites. We model each site s in a given set of Web sites $SITES$, as a vector over a feature space consisting of elements that are either *queries*, *query-terms* or *query-sets*. In detail:

- "QUERYTERMS Model": The feature space $\mathcal{F}$ consists of all individual *query-terms* that constitute the queries leading to documents in the $SITES$:
  $\mathcal{F} = \cup_{s \in SITES} \left(\cup_{d \in s} QT(d)\right)$

- "FULLQUERIES Model": The feature space $\mathcal{F}$ consists of complete *queries*, namely the queries used to access the documents in $SITES$: $\mathcal{F} = \cup_{s \in SITES} \left(\cup_{d \in s} \mathcal{Q}(d) \right)$
- "FREQPATTERNS Model": The feature space $\mathcal{F}$ consists of the frequent *query-sets* for each document in $SITES$, subject to threshold τ: $\mathcal{F} = \cup_{s \in SITES} \left(\cup_{d \in s} QS(d, \tau) \right)$
- "FULLPATTERNS Model": The feature space $\mathcal{F}$ consists of all *query-set* elements for all documents, i.e. the support threshold τ is zero: $\mathcal{F} = \cup_{s \in SITES} \left(\cup_{d \in s} QS(d, 0) \right)$
- "MAXPATTERNS Model": The feature space $\mathcal{F}$ consists of all *maximal query-sets* for the documents in $SITES$, i.e. the frequency threshold to zero as above: $\mathcal{F} = \cup_{s \in SITES} \left(\cup_{d \in s} \widehat{QS}(d, 0) \right)$
- "FULLQUERIESPLUS Model": The feature space $\mathcal{F}$ contains for each document d those *query-sets* for which there is a query in $\mathcal{Q}$ (not necessarily in $\mathcal{Q}(d)$), i.e. independently of whether this query resulted in a request for d:
$\mathcal{F} = \cup_{s \in SITES} \left(\cup_{d \in s} (QS(d, 0) \cap \mathcal{Q}) \right)$

For the vectorization of a Web site using these feature spaces, we also need the weights of the features for the individual documents. Let f_j be a feature, i.e. a query-term, a query-set or a complete query, depending on the feature space we use. The weight of f_j for a document $d \in \mathcal{D}$ is either: a) the number of queries in $\mathcal{L}(d)$ that *contain* f_j, in the case that f_j is a query-terms or query-set, or b) the number of queries in $\mathcal{L}(d)$ that *match exactly* f_j, in the case that f_j is a query. In other words, the weight of each f_j for a document d is $clicks(f_j, d)$ as defined earlier. Then, the unnormalized weight of feature f_j for the site $s_k \in SITES$ is the sum: $w'_{k,j} = \sum_{d \in s_k} clicks(f_j, d)$.

Note that QUERYTERMS and FREQPATTERNS models, correspond to the models presented in [8], the other remaining feature spaces are novel.

6 Model Evaluation and Results

We present our methodology for modeling and classifying Web sites using the previously defined Web site representations. We present a comparison of their clustering and classification performance, and consider as a baseline *bag-of-words* model (denoted as "TEXT model" hereafter). As in related work, we use the DMOZ directory as a "gold standard" for this task.

Dataset. As a data source we use a sample of the Yahoo! UK query log, which contains 2,109,198 distinct queries and 3,991,719 query instances. The vocabulary of the query instances contains 239,274 distinct query terms. Our log sample contained 977 Web sites in DMOZ, which included $5,070$ URLs with hits from queries. These Web sites classified into 216 DMOZ categories.

Table 1 shows the number of features obtained for each Web site model in our dataset. Models based on query sets reduce significantly the dimensionality of the original feature space (obtained using the vector model). FULLPATTERNS reduces the dimensionality by approximately 1/3 of the original feature space, increasing the number of not null entries with respect QUERYTERMS by approximately 400%.

Difficulties with the FREQPATTERNS model. This model represents the aggregated version for the best single-document representation proposed in [8]. This method

involves a minimum frequency threshold τ for each query-set size (number of elements in a set). Empirically, we observe that the selection of this parameter is not possible when faced with documents from a large heterogeneous group of Web sites. In general, frequent itemset mining allows to identify many itemsets with little support and a few itemsets that have high support values. Interestingly, we observe that the aggregated distribution of pattern sizes for documents from *multiple* Web sites does not fit the previous assumption. Our collection shows many itemsets with low support and also many itemsets with high support. Therefore, the selection of τ would require manual inspection of each Web site's distribution, which is not desirable.

Experimental Validation. In this study, we consider DMOZ categories to be the *real* categories of the Web sites. Therefore, we evaluate the classification solutions against this external quality indicator, which we consider as a ground truth. In particular, categories in DMOZ follow a tree hierarchy, this allows to evaluate at different levels of granularity, ranging from general to very specific. By truncating the tree at height 3 (denoted *@3*) we consider 45 only categories, at 4 (*@4*), 75 categories, 5 (*@5*), 104 categories, and also using the complete tree (*Full Dir.*) with 216 categories. When truncating the hierarchy tree, all of the categories that extend beyond the cutting point are collapsed into their parent node.

We create a classifier in which each training instance is composed of a collection of features, defined using a Web site model, and a label that indicates which category the instance belongs to. Each testing instance is categorized using the classifier. The nominal label and the predicted label are compared to evaluate the performance of the model. We use a method based on logistic regression. Additionally, since this is a multi-class problem, we extend the logistic regression model using the one versus rest method (OVR) [4]. The OVR method has shown comparable precision performance to real multi-class methods reducing training time.

To evaluate the performance of the models we compare for each testing instance the nominal class and the predicted class. From the four possible cases presented when comparing both labels, true positives (tp), false positives (fp), false negatives (fn) and true negatives (tn), we calculate the accuracy measure for the tuning and training process ($\frac{tp+tn}{tp+tn+fp+fn}$) and the precision measure for the testing step ($\frac{tp}{tp+fp}$). Then the overall score is calculated by measuring the average.

The dataset is partitioned to perform 3-fold cross validation. Each fold preserves the original proportions existing among the categories. Table 2 shows the performance of the classifiers, each entry represents a precision value. As Table 2 shows, FULLPATTERNS outperforms the other models in all evaluations, with exceptional testing precision (99.69 @Full Dir.). In particular, FULLPATTERNS outperform TEXT by approximately 10% when we consider the full directory. Also, FULLPATTERNS reduces by approximately 50% the dimensionality of the TEXT feature space, increasing the precision performance. Regarding FULLQUERIES, FULLQUERIESPLUS or MAXPATTERNS, the reduction of the feature space dimensionality is more extreme (app. the 5% of the TEXT feature space) without losing precision. Regarding the use of frequent patterns in queries for Web site modeling, it can be observed that the models based on query-sets outperform QUERYTERMS. This result shows the usefulness of the Web site models based on patterns.

Table 1. Number of features of each model

Model	No. of Features	Not Null Entries
FULLPATTERNS	56,929	72,981
FULLQUERIES	9,151	9,875
FULLQUERIESPLUS	8,957	12,269
MAXPATTERNS	10,518	11,098
QUERYTERMS	6,763	16,096
TEXT	178,449	591,004

Table 2. Testing Precision

No. of Clusters	*46* *@3*	*75* *@4*	*104* *@5*	*216* *Full Dir.*
FULLPATTERNS	**98.97**	**99.38**	**98.15**	**99.69**
FULLQUERIES	90.48	83.31	89.96	97.03
FULLQUERIESPLUS	88.43	86.28	87.51	96.72
MAXPATTERNS	93.75	93.14	93.55	96.92
QUERYTERMS	87.64	87.23	87.64	88.97
TEXT	94.06	93.34	91.60	89.25

7 Conclusions

In this work we focus on generating simple and scalable Web site representations for classification. Our approach is centered on providing models which convey users' information needs when visiting relevant Web documents from search engines. Additionally we aim at finding similarities between Web sites in a way which is independent of content, structure and size.

The usefulness of our Web site models is measured by applying categorization to Web site vectors, with the objective of organizing similar sites. Our experimental evaluation shows that our models approach use significantly less features than the full text approach obtaining better results. In addition, the categorization process shows that FULLPATTERNS is the best discriminative model, achieving a precision performance of 99.69% (almost perfect), 10% over the baseline. This is an important result, especially considering that FULLPATTERNS reduces the dimensionality of the original feature space by approximately 50%.

References

1. Baeza-Yates, R., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press / Addison-Wesley (1999)
2. Bharat, K., Chang, B.W., Henzinger, M.R., Ruhl, M.: Who links to whom: Mining linkage between web sites. In: ICDM 2001 (2001)
3. Ester, M., Kriegel, H.P., Schubert, M.: Web site mining: a new way to spot competitors, customers and suppliers in the world wide web. In: KDD 2002 (2002)
4. Fan, R., Chang, K., Hsieh, C., Wang, X., Lin, C.: Liblinear: A library for large linear classification. JMLR 9, 1871–1874 (2008)
5. Karypis, G.: CLUTO a clustering toolkit, http://www.cs.umn.edu/~cluto
6. Kriegel, H.P., Schubert, M.: Classification of websites as sets of feature vectors. In: IASTED 2004 (2004)
7. Lindemann, C., Littig, L.: Coarse-grained classification of web sites by their structural properties. In: WIDM 2006 (2006)
8. Poblete, B., Baeza-Yates, R.: Query-sets: using implicit feedback and query patterns to organize web documents. In: WWW 2008 (2008)
9. Rajalakshmi, R., Aravindan, C.: Naive bayes approach for website classification. In: Das, V.V., Thomas, G., Lumban Gaol, F. (eds.) AIM 2011. CCIS, vol. 147, pp. 323–326. Springer, Heidelberg (2011)

Indexing with Gaps

Moshe Lewenstein⋆

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
moshe@cs.biu.ac.il

Abstract. In *Indexing with Gaps* one seeks to index a text to allow pattern queries that allow gaps within the pattern query. Formally a *gapped-pattern* over alphabet Σ is a pattern of the form $p = p_1 g_1 p_2 g_2 \cdots g_\ell p_{\ell+1}$, where $\forall i, p_i \in \Sigma^*$ and each g_i is a gap length $\in N$. Often one considers these patterns with some bound constraints, for example, all gaps are bounded by a gap-bound G.

Near-optimal solutions have, lately, been proposed for the case of one gap only with a predetermined size. More specifically, an indexing solution for patterns of the form $p_1 \cdot g \cdot p_2$, where g is known apriori. In this case the solutions mentioned are preprocessed in $O(n \log^\epsilon n)$ time and $O(n)$ space, where the pattern queries are answered in $O(|p_1| + |p_2|)$, for constant sized alphabets. For the more general case when there is a bound G these results can be easily adapted with a multiplicative factor of $O(G)$ for the preprocessing, i.e. $O(n \log^\epsilon nG)$ preprocessing time and $O(nG)$ preprocessing space. Alas, these solutions do not lend to more than one gap.

In this paper we propose a solution for k gaps one with preprocessing time $O(nG^{2k} \log^k n \log\log n)$ and space of $O(nG^{2k} \log^k n)$ and query time $O(m + 2^k \log\log n)$, where $m = \sum_{i=1} |p_i|$.

1 Introduction

Indexing refers to the preprocessing of data, in our case text, in order to answer subsequent pattern queries. Suffix trees and suffix arrays are two classical data structures that index text. We denote the text $T = t_1 t_2 \cdots t_n$. The queries are then answered online quickly. Query patterns are of the form $p = p_1 p_2 \cdots p_m$.

Pattern matching with wildcards is the problem of finding all appearances of a pattern in a text where the text and pattern are over alphabet $\Sigma \cup \{\phi\}$, where ϕ denotes the wildcard; where a wildcard matches all other characters. Pattern matching with wildcards was introduced and solved efficiently using convolutional methods in [10]. Slightly tighter solutions have been presented in [5,7,13,14].

Naturally the question was whether there were efficient solutions to indexing with wildcards. Initially it seemed that even solving indexing with one mismatch or wildcard is difficult. In [1] an efficient solution was given. A similar result was also proposed in [9]. Both solutions convert the problem to geometric representations and use fast and effective geometric data structures. Lately, a new result [3]

⋆ This research was supported by the Israel Science Foundation (grant no. 1848/04).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 135–143, 2011.
© Springer-Verlag Berlin Heidelberg 2011

was proposed with optimal query times. These were achieved by reducing to a data structure with tighter results. These solutions work very well for the case of one wildcard or mismatch, but do not carry over, efficiently, to a larger number of wildcards or mismatches. In [6] a solution for a larger number of wildcards was devised.

Pattern matching with gaps, a natural problem, has been considered extensively, see [4,8] for an example of one earlier paper and another recent one. The problem is important in Computational Biology and Computational Musicology. It is also an extension of pattern matching with wildcards, i.e. wildcards with a length (or an interval of lengths).

Indexing with gaps was considered in [15]. They present an interesting and detailed algorithm. However, their running time depends on parameters of the number of matches between the blocks of the non-gaps. In [18] the same problem is considered, only here only one gap in the pattern is assumed (and some other constraints). The problem was also considered in [3,12]. In all three one gap of a predetermined length was considered, i.e. the pattern is of the form $p = p_1\phi^g p_2$, where g is an input to the preprocessing. In the first paper [18] also the lengths of P_1 and P_2 were predetermined. While, the results of [18] were very initial the results in [12,3] were more sophisticated and followed along the lines of the one mismatch/wildcard solutions. Each reduces the problem to geometric data structural problems in the preprocessing and then, in the query stage, performs an appropriate query in the geometric data structure. Actually the one mismatch problem is more difficult than one wildcard (because with a mismatch we do not know where mismatch occurs). The one wildcard extends naturally to one g-length gap since the construction is basically the same.

The reduction in all these problems is to two dimensional range queries. The idea is to construct separate suffix trees one for the text T, denoted S_T, and one for it's reverse T^R, denoted S_{T^R}. The central idea is the same in all solutions. Consider the case of one wildcard then one can view any match (in the text) as having a pivot at the location i where the wildcard occurs. Therefore, if the pattern is $P_1\phi P_2$ then we can match P_2 in S_T and P_1^R in S_{T^R}. P_1^R starts at one before the pivot and P_2 starts one after the pivot. So, we construct on a 2D range grid a point for each (potential) pivot i. Specifically, we have a lexicographic ordering L_1 of the suffixes (the ordering of the suffixes at the bottom of the suffix tree = ordering of the suffix array) of T and a lexicographic ordering L_2 of the suffixes of T^R (which are prefixes of T). The point of pivot i is (x, y), where x is the location of suffix $i+1$ in L_1 and y is the location in L_2 of the prefix of T ending at $i-1$. Now, for query $P_1\phi P_2$ one walks down S_T with P_2 and down S_{T^R} and down S_{T^R} with P_1^R. In each we reach a node which defines a range of leaves. We need to find all the pivots which have a point in the pair of ranges. This translates into a two dimensional range query where all points in a rectangle need to be found. There are different data structures for this problem used in the different solutions. For g-length gaps all that is changed is that the pivot point is now constructed for a pair $i-1$ and $i+g$. It is easy to see that the solution does not carry over to more than one wildcard (or gap).

1.1 Our Results

In this paper we are interested in generalizing indexing with gaps in two senses. First we remove the constraint that the length of the gap g is known a-priori and rather assume that there is a bound G for which the gaps are no larger than it. Second, we allow more than one gap. Let us give the most general definition of pattern matching with gaps. We will later add some constraints. This definition follows along the lines of [4].

Given integers a and b, $0 \le a \le b$, a *variable length gap* $g\{a,b\}$ is an arbitrary string over alphabet Σ of length between a and b, both inclusive. A *variable length gap pattern* (for short VLG pattern) P is the concatenation of a sequence of strings and variable length gaps, that is, P is of the form

$$p = p_1 \cdot g\{a_1, b_1\} \cdot p_2 \cdot g\{a_2, b_2\} \cdots g\{a_k, b_k\} \cdot p_{k+1}.$$

A VLG pattern P matches a substring S of T iff $S = P_1 \cdot g_1 \cdots \cdot g_k \cdot P_{k+1}$, where g_i is any string of length between a_i and b_i, $i = 1, \cdots, k$. We say that P appears at location i of T if there is a substring S of T starting at i which P matches. We will classify VLG patterns into 3 classes; general VLG patterns, right-end VLG patterns, i.e. where all $a_i = 1$, and (simple) gapped-patterns, i.e. non-variable or $a_i = b_i$. We also say that a VLG pattern is G-bounded if $b_i \le G$ for all i. We say that a VLG pattern is k-gap-count-bounded if it contains at most k gaps.

For the sake of simplicity we will focus only on gapped-patterns throughout this paper and leave the handling of general VLG patterns to the journal version.

The *Indexing with Gaps problem* is defined as follows.

Preprocessing Input: Text $T = t_1 t_2 \cdots t_n$, G, and k.
Preprocessing Output: Data structure supporting G-bounded, k-gap-count-bounded gapped-pattern queries.
Query Input: A G-bounded, k-gap-count-bounded gapped-pattern P.
Query Output: All locations i in T where P appears.

To solve the first problem of removing the a-priori constraint of length g, we will use the results of [3] and update them accordingly. An $O(G)$ factor in the preprocessing seems unavoidable. On the other hand, an $O(G)$ factor is sufficient for extending the problem to G-bounded queries for the case of gapped-patterns. For the more general case of G-bounded general VLG pattern queries, where only one gap is allowed one can solve the problem with $O(nG^2)$ space with the same time bounds or with $O(nG \log G)$ overhead and $O(\log G)$ query time.

If we want to allow more than one gap we will need a different data structure than those that reduce to geometric data structures that we have seen before. We will follow along similar lines to the result presented in [6]. However, things get complicated when extending wildcards to G-length gaps. Unlike, the geometric solutions the transfer is not trivial. We will need to produce new updates.

The rest of our paper is organized as follows: in Sect. 2, we give some preliminaries and problem definitions. In Sect. 3 we show our methods for allowing multiple gaps in a few phases.

2 Problem Definitions and Preliminaries

2.1 Preliminary Definitions and Notations

Given a string S, $|S|$ is the length of S. Throughout this paper we denote $n = |S|$. An integer i is a *location* or a *position* in S if $i = 1, \ldots, |S|$. The substring $S[i\,..\,j]$ of S, for any two positions $i \leq j$, is the substring of S that begins at index i and ends at index j. Concatenation is denoted by juxtaposition. The *suffix* S_i of S is the substring $S[i\,..\,n]$.

The *suffix tree* [16,17,20,21] of a string S, denoted $\mathrm{ST}(S)$, is a compact trie of all the suffixes of $S\$$ (i.e. S concatenated with a delimiter symbol $\$ \notin \Sigma$). Each of its edges is labeled with a substring of S (actually, a representation of it, e.g. the start location and its length). The "compact" property is achieved by contracting nodes having a single child. The children of every node are sorted in the lexicographical order of the substrings on the edges leading to them. Consequently, each leaf of the suffix tree represents a suffix of S, and the leaves are sorted from left to right in the lexicographical order of the suffixes that they represent.

$\mathrm{ST}(S)$ requires $O(n)$ space. Algorithms for the construction of a suffix tree enable $O(n)$ preprocessing time when $|\Sigma|$ is constant (where Σ is the alphabet set), and $O(n \log \min(n, |\Sigma|))$ time when $|\Sigma|$ is not. In fact, the suffix tree can be constructed in linear time even for alphabets drawn from a polynomially-sized range, see [16].

LCA queries. An LCA query $lca(u, v)$ is given two nodes u, v in a tree T and reports the lowest common ancestor w of u and v in T. Once w is known, its height in the tree can also be determined. Often, data structures for constant time LCA queries are used with suffix trees, as will be the case here as well. Data structures for answering LCA queries in $O(1)$ time can be built in linear time [11,19]. These data structures also allow the reporting in $O(1)$ time of the edges exiting w on the paths to u and v. In addition, they yield the length of the longest common prefix of the suffixes s_u and s_v associated with u and v, again in $O(1)$ time.

Measured ancestor structure. We will also need a data structure for answering the following query on an n-node compressed trie in $O(\log \log n)$ time: Given a leaf u and a distance h, report the location at which the prefix of s_u of length $|s_u| - h$ ends, i.e. the location distance h above u, where edges are deemed to have length equal to their labels. We call this the measured ancestor structure. Again, such data structures can be built in linear time [2].

Centroid path decomposition. Our construction uses *centroid paths* and *centroid path decompositions*. For our setting, we define the *centroid path* of a tree T to be the path starting at T's root, which at each node v on the path branches to

v's "largest" child, with ties broken arbitrarily; the size of a node is simply the number of leaves in the subtree rooted at that node. In a centroid path decomposition, we decompose each off-path subtree of the centroid path recursively.

The weight of a node on a centroid path is defined to be the number of leaves in its off-path subtrees. In our applications, for node v on a centroid path with off-path child u, it is convenient to include edge (v, u) in the off-path subtree T_u incorporating node u. We will also say that T_u *hangs from* node v.

The following property of a centroid path decomposition of a tree is well known.

Property 1. Let T be an n-node tree with a centroid path decomposition. Let v be a node of T. The path from the root of T to v traverses at most $\log n$ centroid paths.

3 G-Bounded Queries with k Gaps

Given the text T of size $O(n)$ and a suffix tree $S(T)$, a brute-force search for a gapped-pattern $p = p_1 \cdot g_1 \cdot p_2 \cdot g_2 \cdots g_k \cdot p_{k+1}$, can proceed as follows: descend from the root of $S(T)$, find the path that exactly matches p_1. We refer to this path as the *first tier* path. Now, the search can allow for a g_1 gap by searching all the paths hanging off the first tier path at that point in the path (these are the *second tier* paths), taking the g_1 gap at the top of each of these paths. Recall that $|\Sigma|$ edges may hang off a single node of the suffix tree and, therefore, $|\Sigma|^{g_1}$ paths of length g_1 may hang off of v; this implies that we can search for a pattern containing a single gap (of length g_1) in $O(|p_1| + |\Sigma|^{g_1}|p_2|)$ time. It is not difficult to see that a simple extension of this algorithm can handle multiple gaps (for $k = 2$ we must search the *third tier* paths hanging off the second tier paths, etc.), yielding a run time of $O(|\Sigma|^{g_1+\cdots+g_k} m)$, where $m = \sum_{i=1}^{\ell} |p_i|$. This is accomplished without any modifications to the trie, and therefore only $O(n)$ space is used.

3.1 Speeding Up the Search

A first step towards improving the costly run time of the brute force method involves a tradeoff between the inefficient run time of the algorithm and the optimal space requirement of $O(n)$. Specifically, we remove the $|\Sigma|^{g_1+\cdots+g_k}$ term from the runtime, but require $\Theta(n^{k+1}G^k)$ space.

At each node of the trie, we wish to anticipate taking a single gap during the search. To this end, before having knowledge of the gapped-pattern p we preprocess the trie. The gap to be taken can be of any length between 1 and G. So, at every node we create G *gap subtrees*, one for each possible gap length, which will be searched if a gap is taken starting at that node. Note, that the traversal in a tier may end in the middle of an edge. However, in this case it is sufficient to "slide" to the node under it and adapt, g, the length of the gap, i.e. if there are x characters on the edge from the location the traversal reached on the edge until the node under it, we move to the node under it and continue the search with a gap of $g - x$.

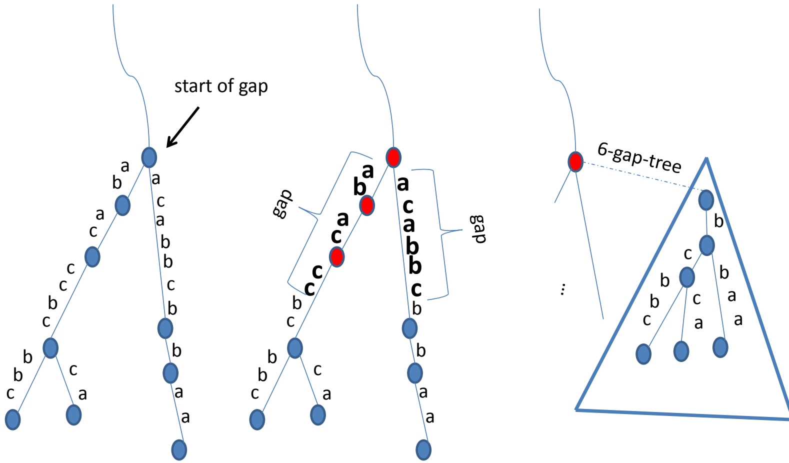


Fig. 1. (a) node v with two suffixes in subtree, (b) 6-length gaps marked, (c) 6-gap-tree created

A gap subtree for length r, $1 \leq r \leq G$, which we call a *gap-r-tree*, at node v contains a merge of all the subtrees at the end of the r-length paths starting at v, see Figure 1. Another way to envision the gap-r-tree is to think of all the leaves of the suffix tree in the subtree of v. Each corresponds to a suffix, say S_i. If we truncate the h-length prefix of S_i we have the suffix S_{i+h}. So, if the set of leaves in the subtree of v corresponds to suffixes $\{S_{i_1}, \cdots, S_{i_q}\}$ and the string associated with node v (the locus of the root-to-v node) is of length d then the gap-r-tree at node v is a compressed trie of the suffixes $\{S_{i_1+d+r}, \cdots, S_{i_q+d+r}\}$ (for those that satisfy $i_j + d + r \leq n$). To create the gap-r-tree at node v one traverses the suffixes and updates their index and uses the lexicographic ordering to resort them. The lexicographic ordering is obtainable from the suffix tree of the original tree (or one may use any other suffix sorting algorithm that one desires). Once they are resorted one uses standard techniques to construct the compressed trie over these updated suffixes giving the desired. The resorting can be done in $O(q \log\log n)$ time, where q is the number of elements in the subtree of v. The sorting is done with a van-Emde Boas tree or with a y-fast-trie. This can be done because all the "new" suffixes are really just suffixes from the original suffix tree and they have been enumerated from $1, \cdots, n$ according to their lexicographic ordering. Hence, the "universe" of the q elements that are sorted is of size n.

The size of the modified overall trie is $\Theta(n^2 G)$. This is because in the original suffix tree, every leaf u representing a suffix, may have $O(n)$ ancestors and each have G gap trees hanging off the node containing u.

A search for a gapped-pattern with one gap, i.e. $p = p_1 g_1 p_2$, on the new trie descends from the root, and as before finds the path for p_1. This is the first tier path. Now, the search will continue in the g_1-gap tree with the pattern p_2. This search takes time $O(|p_1| + |p_2|)$. It is not difficult to see that a simple extension

of this algorithm can handle multiple gaps – for $k = 2$ we must create secondary gap subtrees for each node of each primary gap tree, etc. – yielding a run time of $O(m)$, where $m = \sum_{i=1}^{\ell} |p_i|$ with space $O(n^{k+1}G^k)$.

3.2 Better Tradeoffs

Although the aforementioned technique improves the run time significantly, the space requirement is now problematic. However, a variant of this technique yields an $O(nG^{2k} \log^k n)$ structure that supports gapped-pattern queries in $O(2^k m)$ time.

As before, before having knowledge of p we preprocess the trie and create a gap subtree for each node. However, there is a slight twist here. We will consider the centroid partition of the suffix tree. The gap subtrees created at a node v will not contain all (truncated) suffixes in the subtree of v as before, but rather will contain only those whose v-to-suffix path leave the centroid path within the gap size, see Figure 2.

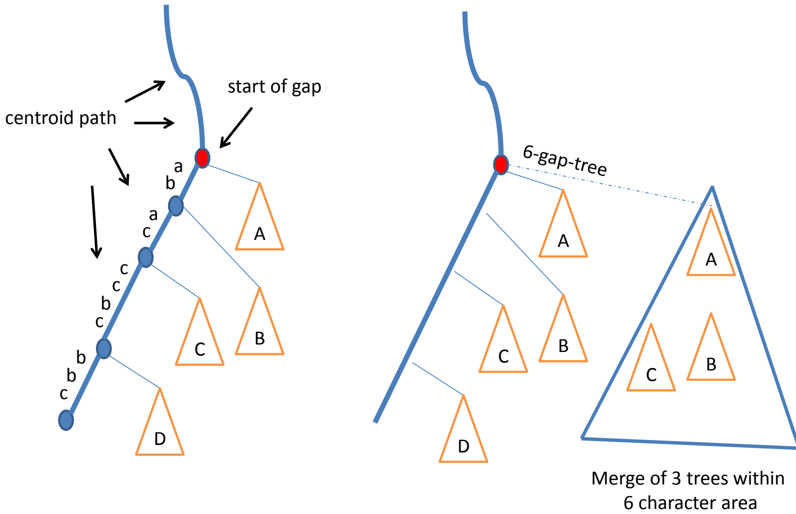


Fig. 2. (a) 1 centroid path with suffix tree subtrees hanging off, (b) merge of appropriate subtrees (off centroid-path trees) for 6-gap-tree

If during the execution of a search a gap is taken at the node, then both the gap tree and the subtree where the gap is along the centroid path must be searched. That is, taking a gap will spawn two searches – better than the $|\Sigma|^g$ searches spawned in the brute-force algorithm, but less efficient than the single search required in the search speedup.

The benefit of this tradeoff lies in reducing the size of the structure, which is now $O(nG^2 \log n)$. This bound follows easily when one considers each leaf in the original trie: Each gap tree that contains this leaf is associated with a distinct

ancestor of this leaf, and further each such ancestor lies on a centroid path from which the subtree containing the leaf diverges within at most G levels down. As a leaf may have at most $\lfloor \log_2 n \rfloor$ centroid paths on the path from root to leaf, a leaf may be found in the gap subtrees of at most $G\lfloor \log_2 n \rfloor$ ancestors in the modified trie. Each such ancestor has at most G such subtrees.

It is not difficult to see that, as before, a simple extension of this algorithm can handle multiple gaps – we create k-ary gap subtrees for each node of each $(k-1)$-ary gap subtree. This yields a run time for the query of $O(2^k m)$ with space $O(nG^{2k} \log^k n)$.

In the next subsection we will see how to reduce the query time to $O(2^k \log\log n + m)$.

3.3 Final Speedup of the Query

The multiplicative factor of $O(m)$ follows from the traversals on the p_i's and the 2^k and the $\log^k n$ factors are for the number of subtrees that we need to traverse within.

A circumvention to the traversal is as follows. Recall that each of these subtrees is actually a trie over a collection of suffixes of the original suffix tree. We have also assumed that these suffixes have been sorted (lexicographically) either by the suffix tree construction or by some other suffix sorting method. Therefore, each trie is actually a collection of numbers (the lexicographic ranks). Now, when we first see the gapped pattern for each i we traverse with p_i from the root of the suffix tree. When we reach the end of p_i we are at a node (or just above a node) u. The leftmost leaf and the rightmost leaf in the subtree of u represent suffixes with p_i as a prefix of this suffix. So, they, actually their lexicographic ranks, may represent p_i.

So, now when we reach a gap subtree instead of traversing it from top with p_i we will do a predecessor query with the lexicographic rank of p_i as the query and the lexicographic ranks of the suffixes of the subtree as the data. Once we have found the predecessor and successor we will be able to find the node in the subtree representing the end of the p_i search by applying an LCA query to the newly found predecessor and successor. If there is only a predecessor (or only a successor) we may use a measured ancestor query. This predecessor/successor queries and measured ancestor queries can be implemented in $O(\log\log n)$ time. This yields:

Theorem 1. *Let T be a text of size n. One can build an indexing scheme of size $O(nG^{2k} \log^k n)$ so that one can answer gapped queries bounded by gap-bound G with k gaps in time $O(m + 2^k \log\log n)$.*

References

1. Amir, A., Keselman, D., Landau, G., Lewenstein, N., Lewenstein, M., Rodeh, M.: Text indexing and dictionary matching with one error. J. of Algorithms 37(2), 309–325 (2000)
2. Amir, A., Landau, G., Lewenstein, M., Sokol, D.: Dynamic pattern, static text matching. ACM Transactions on Algorithms 3(2) (2007)

3. Bille, P., Gørtz, I.L.: Substring range reporting. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 299–308. Springer, Heidelberg (to apppear, 2011)
4. Bille, P., Li Gørtz, I., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 385–394. Springer, Heidelberg (2010)
5. Clifford, P., Clifford, R.: Self-normalised distance with don't cares. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 63–70. Springer, Heidelberg (2007)
6. Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of the Symposium On Theory of Computing (STOC), pp. 91–100 (2004)
7. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the Symposium On Theory of Computing (STOC), pp. 592–601 (2002)
8. Crochemore, M., Iliopoulos, C., Makris, C., Rytter, W., Tsakalidis, A., Tsichlas, K.: Approximate string matching with gaps. Nordic J. of Computing 9(1), 54–65 (2002)
9. Ferragina, P., Muthukrishnan, S., de Berg, M.: Multi-method dispatching: A geometric approach with applications to string matching problems. In: Proceedings of the Symposium on Theory of Computing (STOC), pp. 483–491 (1999)
10. Fischer, M., Paterson, M.: String matching and other products. In: Karp, R.M. (ed.) Complexity of Computation, SIAM-AMS Proceedings, vol. 7, pp. 113–125 (1974)
11. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)
12. Iliopoulos, C., Rahman, M.: Indexing factors with gaps. Algorithmica 55(1), 60–70 (2008)
13. Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: Proceedings of the Symposium on Foundations of Computer Science (FOCS), pp. 166–173 (1998)
14. Kalai, A.: Efficient pattern matching with don't cares. In: Proceedings of the Symposium on Discrete Algorithms (SODA), pp. 655–656 (2002)
15. Lam, T.-W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Space efficient indexes for string matching with don't cares. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 846–857. Springer, Heidelberg (2007)
16. Farach-Colton, S.M.M., Ferragina, P.: On the sorting-complexity of suffix tree construction. J. ACM 47(1), 987–1011 (2000)
17. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
18. Peterlongo, M.S.P., Allali, J.: Indexing gapped-factors using a tree. Int. J. Found. Comput. Sci. 19(1), 71–87 (2008)
19. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplifications and parallelization. SIAM Journal on Computing 17(6), 1253–1262 (1988)
20. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
21. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, pp. 1–11. IEEE, Los Alamitos (1973)

Fast Computation of a String Duplication History under No-Breakpoint-Reuse
(Extended Abstract)

Broňa Brejová[1], Gad M. Landau[2], and Tomáš Vinař[1]

[1] Faculty of Mathematics, Physics, and Informatics, Comenius University, Mlynská Dolina, 842 48 Bratislava, Slovakia
[2] Department of Computer Science, University of Haifa, Haifa 31905, Israel

Abstract. In this paper, we provide an $O(n \log^2 n \log\log n \log^* n)$ algorithm to compute a duplication history of a string under no-breakpoint-reuse condition. Our algorithm is an efficient implementation of earlier work by Zhang et al. (2009). The motivation of this problem stems from computational biology, in particular from analysis of complex gene clusters. The problem is also related to computing edit distance with block operations, but in our scenario the start of the history is not fixed, but chosen to minimize the distance measure.

Keywords: duplication, edit distance, sequence evolution, dynamic text.

1 Introduction

We are given a string T over a finite alphabet Σ, and we assume that it was created by the following process. We start with some string S which is a permutation of alphabet symbols. In each step, we apply a *duplication* which takes a substring of the current string (*source*), and inserts its copy at a different position (*target*) within the string (the target position cannot be inside the source). For example, if the current string is $uvwx$, the duplication with source v and target between w and x creates string $uvwvx$. The last string in this sequence of duplications should be T. We call a sequence of duplications leading to the construction of string T from some permutation of the alphabet a *duplication history* of T.

In the general model, starting from a simple string uv, string $(uv)^k$ can be created by $\lceil \log k \rceil$ duplication operations. However, we will consider a restricted version of the problem under the *no breakpoint reuse* rule. In this version of the problem, each duplication introduces three breakpoints: at the left boundary of the source, at the right boundary of the source, and between the two symbols around the target position. The example above does not satisfy the breakpoint reuse rule because we repeatedly reuse the breakpoint between u and v.

For each symbol x, we denote $|x$ its left boundary and $x|$ its right boundary. A breakpoint between symbols x and y uses the right boundary of x and the left boundary of y. Under the no-breakpoint-reuse rule, each boundary of each

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 144–155, 2011.
© Springer-Verlag Berlin Heidelberg 2011

symbol of the alphabet can be used at most once in the whole duplication history. Consider, for example, the sequence of duplications:

$$a\underline{bcde}fg \rightarrow abcdefb\underline{cd}eg \rightarrow abccddefbcdeg$$

The first duplication uses boundaries $a|$, $|b$, $e|$, $|f|$, and $|g$, and the second duplication $b|$, $|e$, $|c|$, and $|d|$. Thus no more duplications can be performed on this string without violating the no-breakpoint-reuse rule.

Definition 1 (Viable strings under no-breakpoint-reuse). *We call string T* viable *if there is a permutation of alphabet symbols S such that T can be created from S by a sequence of duplications under the no-breakpoint-reuse rule.*

In this paper, we introduce an efficient quasilinear algorithm that finds a shortest duplication history of a viable string. Our new algorithm builds on the work of Zhang et al. (2009) and is the first efficient algorithm presented for this problem. In the rest of the section, we describe motivation for this problem and introduce related work.

1.1 Motivation and Related Work

A similar problem has been studied in computational biology, in particular in the context of gene cluster evolution analysis. Briefly, each symbol of our alphabet corresponds to a segment of the DNA sequence, which may appear in several copies in the studied gene cluster. In the simplest scenario, we ignore small local changes (e.g., point mutations and short insertions and deletions), and only concentrate on large-scale evolutionary events, such as duplications. The segmentation of the DNA sequence is created based on the observed local self-alignments of the DNA sequence so that the breakpoints of duplications will always fall between the individual segments.

In this context, Zhang et al. (2009) introduced a problem of reconstruction of duplication histories, while considering an extended set of operations, including not only segmental duplications, but also tandem duplications and duplications with reversal; we do not consider these additional types of events in this paper. The no-breakpoint-reuse rule stems from the assumption that the DNA sequences are long, and since the evolution is considered to be a random process, it is unlikely that the same event would act on the same boundary twice (Nadeau and Taylor, 1984).

Zhang et al. (2009) developed an elegant solution to the problem of computing the number of duplications. In the input string, they identify a *candidate duplication* defined by a set of simple rules. They assume this duplication is the last (most recent) duplication in the history and undo it by deleting the string inserted by the duplication event. Then they continue the process on the resulting shorter string, until they obtain a string in which each symbol appears only once. We describe the notion of candidate duplications in detail in the next section.

Interestingly, although there can be several candidate duplications in a given step, undoing any of them will lead to an optimal history. Zhang et al. (2009) did not investigate the time complexity of the algorithm. Straightforward implementation takes $O(n^3)$ time, where n is the length of the input string, whereas in this paper, we provide a quasilinear algorithm.

Another model of duplicated strings has been introduced by Kahn et al. (2010). Here, duplications are not performed within the string, but instead we are given a source string S and a target string T, and our task is to assemble T by repeatedly inserting substrings of S into the initially empty target string. The structure of the target string is much simpler in this case, and Kahn et al. (2010) show a dynamic programming algorithm to compute the smallest number events needed to create the target string.

Several authors have considered also local mutations in the DNA sequences corresponding to individual segments and combined phylogenetic trees inferred on individual segments with the reconstruction of duplication histories. Optimization of such combined objectives is difficult to formulate as a clean combinatorial problem. Works of Song et al. (2010) and Vinar et al. (2010) extend on Zhang et al. (2009) model, while Elemento et al. (2002) and Lajoie et al. (2007) consider only a single type of segments (e.g., strings of the form u^k), making the problem similar to the gene tree vs. species tree reconciliation problem.

In a more combinatorial setting, several authors have considered computing the edit distance with block operations, where in addition to the usual symbol insertions, deletions, and substitutions, we allow moving, deleting, or copying whole blocks of text to transform one string into another. Computing the edit distance in the presence of such large-scale operations is typically NP-hard, depending on the exact model (Lopresti and Tomkins, 1997; Shapira and Storer, 2007), and authors concentrate on providing approximation algorithms (Shapira and Storer, 2003; Ergün et al., 2003; Cormode and Muthukrishnan, 2007) or algorithms for special cases where editing proceeds from left to right (Ann et al., 2010). In general, these problems differ from ours by allowing a richer set of operations, but also by fixing both endpoints of the history, while we minimize over all possible initial permutations.

1.2 Relationship to Breakpoint Graphs

Ma et al. (2008) give a polynomial-time algorithm for a very general problem of inferring an evolutionary history of multiple species under several large-scale operations, including duplications, insertions, deletions, and rearrangements. To achieve this result, they use the no-breakpoint-reuse rule and additionally they assume that it is possible to reconstruct the phylogeny of each segment type (each symbol in our alphabet). In our work, we do not use this assumption, and indeed it would not be practical in gene cluster analysis, where individual segments are often short and highly similar to each other, making phylogeny reconstruction difficult.

Nonetheless, techniques of Ma et al. (2008) can be used to compute the number of events in a duplication history of a viable string in linear time. We create

a breakpoint graph which has two vertices for each symbol in the alphabet, representing the boundaries $|a$ and $a|$. Two vertices are connected by an edge if they are adjacent in the input string. In a string without duplications, all connected components have size one or two. Each duplication replaces three components of size two by one component of size six or two components of size three. Therefore the number of duplication events in the history is simply $s + t/2$ where s is the number of components of size six and t is the number of components of size three. However, examination of the breakpoint graph of a string is not sufficient to reconstruct the history itself or even to distinguish if a string is viable or not (see Fig. 1). In the rest of this paper, we provide an algorithm capable of answering both of these questions.

Fig. 1. Left: Two strings with the same breakpoint graph have different candidate duplications. Right: Two strings have the same breakpoint graph, but one of them is viable and the other is not. In viable strings, destinations of candidate duplications are underlined, any copy of these strings can serve as a source of the duplication.

2 Candidate Duplications

In this section, we introduce necessary notation and define candidate duplications, which are the crucial ingredient of the algorithm from Zhang et al. (2009).

Consider a string $X = x_1 x_2 \dots x_n$. Let $X[i \dots j]$ denote the substring $x_i \dots x_j$. Let $\#_X(u)$ be the number of occurrences of a string u in a string X. Let $s_X(a)$ be the set of all symbols that immediately follow one of the occurrences of a in X, and similarly let $p_X(a)$ be the list of immediate predecessors of a in X. We say that two symbols a and b are *linked* in string X if $s_X(a) = \{b\}$ and $p_X(b) = \{a\}$. We say that two symbols a and b form a *unique pair* in X if $\#_X(ab) = 1$. For simplicity, we will implicitly assume two sentinel symbols at the beginning and at the end of the string which are never duplicated.

Definition 2 (Candidate duplication). *A* candidate duplication *in a string X is a pair of substrings $S = X[i \dots j]$ (source) and $D = X[k \dots \ell]$ (destination) such that all of the following conditions hold.*

1. *Intervals $[i, j]$ and $[k, \ell]$ are disjoint and non-empty.*
2. *Substrings $X[i \dots j]$ and $X[k \dots \ell]$ are identical.*
3. *Symbols x_{i-1}, x_j, and x_{k-1} are pairwise distinct; similarly x_i, x_{j+1} and $x_{\ell+1}$ are also three distinct symbols.*

4. *Let* X' *be the string obtained by deleting* D *from* X*, i.e.* $X' = X[1 \dots k-1]X[\ell+1 \dots n]$*. Then symbols* x_{k-1} *and* $x_{\ell+1}$ *are linked in* X'*.*
5. *Similarly, symbols* x_{i-1} *and* x_i*, as well as* x_j *and* x_{j+1}*, are also linked in* X'*.*

See Fig. 1 for an example of candidate duplications in several viable strings.

This definition of a candidate duplication slightly differs from the one in Zhang et al. (2009), since we allow linked symbols in X, whereas Zhang et al. (2009) assume each pair of linked symbols was collapsed to a single symbol. The following theorem forms the basis of the algorithm.

Theorem 1 (Zhang et al. (2009)). *Let us consider all duplication histories with no breakpoint reuse for a viable string* X*. Then the following holds.*

1. *The most recent event in each history is one of the candidate duplications in* X*.*
2. *For every candidate duplication in* X *there is a history in which this duplication is the most recent event.*
3. *All histories of* X *have the same number of events.*

Note that Zhang et al. (2009) allow a richer set of events, yet their proof of correctness can be modified and even simplified for our scenario. The full proof of the theorem is omitted due the space constraints. However, to build the intuition about the problem, we at least prove the statement 1 of the theorem, that is, that the most recent duplication necessarily satisfies the definition of the candidate duplication.

Clearly, conditions 1 and 2 are satisfied by every duplication. Condition 3 is satisfied because otherwise the duplication would use the same breakpoint twice. Finally, in the starting string of any history, every symbol occurs only once, and therefore every symbol is linked to both its neighbours. If a and b are linked before a duplication but not after the duplication, the duplication needs to use a breakpoint after a or before b. If one of the conditions 4 and 5 was not satisfied, then one of the characters x_{i-1}, x_i, x_j, x_{j+1}, x_{k-1}, $x_{\ell+1}$ could never become linked to its neighbour in X' without breakpoint reuse.

According to this theorem, we can obtain the shortest duplication history by repeatedly undoing an arbitrary candidate duplication, until we arrive at a string in which each symbol occurs exactly once. Regardless of the choice of candidate duplications, we always obtain a history of equal length, and the history will not contain breakpoint reuse. (Note that if two symbols are linked, in the algorithm they will never become a breakpoint of a candidate duplication.)

In this paper, we are focused on efficient implementation of this algorithm, and particularly on efficient detection of candidate duplications. To this end, we formulate a lemma that will allow us to quickly verify whether a given substring is a destination string of a candidate duplication; note that a single destination can be often paired with multiple sources, but we will never have to list all these source strings explicitly.

Lemma 1 (Destination detection). *A substring* $D = X[k \dots \ell]$ *is the destination of a candidate duplication if and only if the following conditions hold:*

C1 x_{k-1} *and* x_k *are a unique pair in* X *and so are* x_ℓ *and* $x_{\ell+1}$.
C2 *String* $X[k \ldots \ell]$ *has at least one other copy in* X, *i.e.* $\#_X(X[k \ldots \ell]) \geq 2$
C3 $x_\ell \neq x_{k-1}$ *and* $x_k \neq x_{\ell+1}$.
C4 *The two symbols surrounding* D *occur paired with each other everywhere else in* X, *i.e.* $s_X(x_{k-1}) \subseteq \{x_k, x_{\ell+1}\}$, $p_X(x_{\ell+1}) \subseteq \{x_\ell, x_{k-1}\}$
C5 $p_X(x_k) = \{x_{k-1}, c\}$ *for some* $c \neq x_{k-1}$, *and similarly* $s_X(x_\ell) = \{x_{\ell+1}, d\}$ *for some* $d \neq x_{\ell+1}$.
C6 *Symbols* c *and* d *from the previous condition have only one neighbor, more precisely* $s_X(c) = \{x_k\}$ *and* $p_X(d) = \{x_\ell\}$

We omit the proof of the lemma due to space constraints.

3 Finding a Candidate Duplication in Linear Time

In this section, we introduce a simple linear-time algorithm for finding a candidate duplication in a string X. We can simply repeat this algorithm in each iteration, until we undo all duplications. Since there are $O(|\Sigma|)$ duplications in any valid history under the no breakpoint reuse rule, this yields $O(n|\Sigma|)$ algorithm for duplication history reconstruction. We will later introduce additional data structures to achieve $\tilde{O}(n)$ overall time.

According to condition C1, each destination of a candidate duplication is bordered by a unique pair on each side. Clearly, a unique pair cannot occur inside any destination, otherwise it would not have a source copy (C2). Therefore, we can simply split string X into *possible destinations* bordered by successive unique pairs and to check for each such possible destination whether it indeed satisfies the conditions from Lemma 1 of being the destination of some candidate duplication.

In order to quickly find all unique pairs and check conditions C4-C6, we create a data structure that lists for each symbol $a \in \Sigma$ its successors $s_X(a)$, and for each successor $b \in s_X(a)$, the number of occurrences of the pair ab in X. Similarly, we will keep a list of predecessors and the number of their occurrences. Both tables can be easily created by a single linear-time pass through the string, because each symbol has at most two distinct successors, as implied by the following lemma.

Lemma 2. *Consider a valid duplication history under no-breakpoint-reuse, that is, a sequence of strings* $X_1 \ldots X_m$ *such that* X_1 *contains each symbol at most once and* X_{i+1} *was obtained from* X_i *by a duplication. Then for every symbol* a *the set of its successors* $s_{X_i}(a)$ *changes at most once in the history and only the following three cases can occur:*

- *The set* $s_{X_i}(a)$ *has size 1 and does not change throughout the history.*
- *The set* $s_{X_i}(a)$ *starts with size 1 at* $i = 1$ *and at some point another element is added, creating a set of size 2.*
- *The set* $s_{X_i}(a)$ *starts with size 1 at* $i = 1$ *and at some point it changes to a different singleton set.*

In addition to the successor and predecessor tables, we construct a suffix array of string X and the longest common prefix (lcp) table, both in $O(n)$ time (Kärkkäinen et al., 2006; Kasai et al., 2001). With these data structures, we can easily enumerate all possible destinations and to check conditions from Lemma 1 as follows.

First of all, we scan through the string and for each pair of adjacent symbols ab use the successor table to verify if b follows a only once. This will give us a list of unique pairs, and we process each two successive unique pairs $x_{k-1}x_k$ and $x_\ell x_{\ell+1}$ in turn.

Condition C1 is already satisfied and condition C3 can be checked trivially. Condition C2 can be checked in $O(1)$ time by testing if the lcp value of the suffix $X[k \dots n]$ with one of its neighbors in the suffix array is at least $\ell - k + 1$. Conditions C4-C6 can be checked easily using the successor and predecessor tables. The overall running time for finding all destinations of candidate duplications is $O(n)$, because we need $O(n)$ time for preprocessing and we check $O(n)$ potential destinations, each in $O(1)$ time.

4 Quasi-Linear Algorithm for History Reconstruction

In order to obtain a faster algorithm, we notice that it is not necessary to build the data structures from scratch after undoing each duplication. Instead, we will show how to update the data structures in each iteration so that fast evaluation of all conditions C1-C6 can be achieved.

Testing the existence of the source. The main hurdle is testing of C2, i.e. evaluating whether a substring has more than one copy in the current string. In the previous section, a suffix array was sufficient to facilitate this query, but it would be difficult to update the array after deleting the substring corresponding to a candidate destination. Instead, we use a data structure of Alstrup et al. (2000) capable of supporting the following operations over a dynamic collection of texts:

- STRING(a) where $a \in \Sigma$ creates a new string of length one in the collection,
- CONCATENATE(s_1, s_2) concatenates two existing strings in the collection,
- SPLIT(s, i) splits string s into two strings at position i, and
- FIND(s) finds all occurrences of string s from the collection in the remaining collection strings.

Operations CONCATENATE and SPLIT work in $O(\log^2 n \log\log n \log^* n)$ time, STRING works in $O(\log n \log^* n)$ time, and FIND works in $O(occ \log n + \log n \log\log n)$ time, where occ is the number of occurrences of the string.

In our scenario the number of occurrences can be high, but we only need two of them (arbitrarily chosen). The underlying data structure for the FIND operation is a dynamic range tree (Mehlhorn and Näher, 1990), which can be easily modified to retrieve two arbitrary occurrences (instead of all of them), and the resulting running time will be $O(\log n \log\log n)$.

With this data structure, we will support operations needed in our scenario as follows. At the beginning, we use n times STRING and $n-1$ times

CONCATENATE to create a representation of X in the data structure, in the overall time $O(n \log^2 n \log\log n \log^* n)$. To delete a destination $D = X[k \ldots \ell]$, we use SPLIT twice to split X into $X[1 \ldots k-1]$, $X[k \ldots \ell]$, and $X[\ell+1 \ldots n]$. The middle part is split into individual singletons by additional $\ell - k$ splits, and the two remaining parts of X are concatenated together. The overall time for a deletion of a substring of length p from X is thus $O((p+1) \log^2 n \log\log n \log^* n)$. Since every symbol in X is deleted at most once over the course of the whole algorithm, we spend the total of $O(n \log^2 n \log\log n \log^* n)$ time on all deletions together.

Finally, to check if a possible destination $X[k \ldots \ell]$ has at least one other copy in X, we split X into $X[1 \ldots k-1]$, $X[k \ldots \ell]$ and $X[\ell+1 \ldots n]$ and use operation FIND on the middle string. Regardless of the total number of occurrences of $X[k \ldots \ell]$ in X, we only need to find out if there is at least one additional copy, and if yes, at which position. Finally, we concatenate the two strings back together. The whole operation can be achieved in $O(\log^2 n \log\log n \log^* n)$ time. (Note that singletons from deleted substrings technically remain in the collection, and therefore we need to test for occurrences of strings of length one by a different method. This can be easily achieved by other data structures maintained in our algorithm.)

Efficient maintenance of possible destinations. Even though we have a data structure that can efficiently answer queries about the existence of a source for a particular destination, to achieve quasi-linear time, we cannot afford to test all possible destinations in each round of the algorithm.

To address this problem, we will maintain a catalog of all possible destinations (i.e., regions between unique pairs), and we will also maintain a status on each such destination. The status can achieve one of three values: discarded, waiting or valid. A possible destination is *discarded*, if it does not satisfy at least one condition out of C3-C6. The destination is *valid*, if it satisfies C2 (it has another occurrence in X). A valid destination may or may not satisfy conditions C3-C6, but we have never seen those conditions violated in any check done so far. Finally, *waiting* destinations do not satisfy condition C2 and we have never seen conditions C3-C6 violated.

Waiting destinations may become valid destinations by emergence of new sources after removal of some of the duplications, and valid destinations can also become waiting. However, once a destination is discarded (by checking conditions C3-C6 and finding that one of them is not satisfied), it can never become a valid destination again, as outlined in the following lemma.

Lemma 3 (Life cycle of possible duplications). *If a possible duplication bordered by two unique pairs fails one of the conditions C3-C6, it will never meet all conditions of Lemma 1 in the course of the history reconstruction algorithm.*

Proof. Clearly, condition C3 can be changed only by a change in the unique pairs at the boundaries which would lead to the disappearance of this particular possible duplication. The remaining conditions C4-C6 concern the sets of predecessors and successors of the symbols at the proposed breakpoints. Let us

assume, without loss of generality, that the condition which is not satisfied involves the list of successors of symbol a. The only time a set of successors of a changes is when we create a breakpoint after a. However, that means that in the rest of the history a cannot be used as a left side of the breakpoint, and therefore no duplication involving a breakpoint after a will ever satisfy all conditions of Lemma 1. □

To maintain the list and status of possible destinations, we use several simple data structures. First of all, we augment the table of predecessors and successors of each symbol with the list of occurrences for each pair of symbols a and $b \in s_X(a)$. Coordinates of occurrences will be kept with respect to the original input string before any deletions. The list of occurrences will be kept in a balanced binary search tree, facilitating deletions of occurrences in $O(\log n)$ time.

The entire string is kept in an augmented binary search tree with the original position used as the key. This will allow us to find predecessor and successor of each symbol of the current string and to find a character at a given position in either current or original coordinates. All of these operations can be performed in $O(\log n)$ time, deletion of any character is also $O(\log n)$ time.

Finally, we keep coordinates of all unique pairs in a binary search tree with the original coordinate of the first letter in the pair as the key. For each unique pair, we will also maintain the status of the possible destination in the region between this pair and its successor. The set of valid destinations is kept in a separate binary search tree, indexed by the original coordinate.

Upon removal of destination D, all these data structures need to be updated. In particular:

- All symbols of D need to be removed from the binary tree storing X.
- Predecessor and successor data structures need to be updated for every pair of adjacent symbols from D.
- Let a and b be symbols adjacent to D before its deletion. We need to add pair ab to the successor and predecessor tables.

As a consequence of these steps, there can be new unique pairs created, as well as some of the previous unique pairs deleted. Such pairs can be easily identified during the above mentioned update steps. The binary tree storing unique pairs thus needs to be updated. Finally, we create a "due for revision" list of potential destinations whose status needs to be recalculated after the removal of D. This list includes newly created destinations, that is those that are adjacent to newly created unique pairs as well as those that were created by merging adjacent possible destinations after unique pairs were removed. Some existing destinations might also change status from waiting to valid, if a copy was created by removal of D. All such destinations must contain pair ab as a substring. We can find all destinations containing ab using our data structures and add them to the "due for revision" list. The only destination that may change status from valid to waiting is the source of the current destination D, and we treat it separately in the algorithm.

Algorithm and time complexity. Now we are ready to state the final version of our algorithm. Its key part is the operation CHECK(k, ℓ) which checks if a possible destination $X[k \dots \ell]$ in the current string X satisfies the conditions of Lemma 1, and returns the new state of the destination: discard, valid, or waiting; if the new state is valid, it also returns one possible source S.

The running time of this procedure is $O(\log^2 n \log\log n \log^* n)$ time. We need $O(\log n)$ time to convert between different forms of coordinates and one call to operation FIND in the dynamic text data structure.

At the beginning we initialize the dynamic text data structure, create all necessary binary trees and test status of every possible destination. Afterwards we proceed as follows:

1. Let D be one of the valid destinations. Remove D from the list of valid destinations. If there are no more valid destinations, finish.
2. Use CHECK to classify D. If the result is "discarded", mark D as "discarded" and repeat from step 1. Otherwise, let S be the returned source. (At this step, the result cannot be "waiting").
3. Remove D from the dynamic text data structure.
4. Remove D from the data structures used for the maintenance of possible destinations. Let R be the list of destinations "due for revision".
5. For each D' from R, run CHECK(D') and update its state.
6. If S is a possible destination, run CHECK(S) and update its state.
7. Repeat from step 1.

To analyze the running time, first note that throughout the whole algorithm each symbol of the alphabet may be the left part of at most two unique pairs, and therefore overall at most $O(|\Sigma|)$ unique pairs are both created and deleted. The overall number of possible destinations considered in the algorithm is also $O(|\Sigma|) = O(n)$.

Status of each new possible destination is checked once in step 5, when it is created. Each possible destination is handled at most once in step 2, and afterwards it is either discarded or undone. Step 6 is run $O(n)$ times in the whole algorithm, at most once for every duplication. Finally, step 5 is also run for all possible destinations containing currently created new pair ab. For each $a \in \Sigma$, this happens at most once due to the no breakpoint reuse assumption. Therefore for each occurrence of a in the original string X this rule will trigger the check of the possible destination currently containing a at most once. The total number of calls to CHECK is thus $O(n)$.

The remaining operations necessary to update and use our data structures take either $O(\log n)$ per iteration, or $O(\log n)$ per deleted character, leading to an $O(n \log n)$ contribution to the overall running time. The total running time of the algorithm is thus $O(n \log^2 n \log\log n \log^* n) = \tilde{O}(n)$.

5 Conclusion and Open Problems

In this paper, we have presented an efficient algorithm to compute the shortest duplication history of a string under no-breakpoint-reuse rule. The original

algorithm is due to Zhang et al. (2009), who did not study its complexity or an efficient implementation.

Many questions in this area still remain open. First of all, our algorithm computes one possible history, and it can be easily extended to enumerate all possible histories in $O(n^2)$ time per history. However, a faster algorithm should be possible by appropriate changes in the data structures. Instead of enumerating all histories, one might be interested in efficiently counting or sampling possible histories, which remains an open problem. A natural extension asks for a history with a fixed starting string. If the nice structure of the problem carries into this scenario, the problem may become one of the few tractable block edit distance problems.

Another class of open problems concerns models in which the no-breakpoint-reuse assumption is partially or completely relaxed or in which additional operations, such as deletions, are allowed. An interesting scenario, with strong biology motivation, allows an operation of speciation. In this scenario, we work with a collection of strings, each from a different biological species. The history starts with a single string, and at any point in the duplication history, we may create another copy of one of the strings in the collection. No further copying between the different strings in the collection is allowed; all duplications must act within the confines of one of these strings (i.e., the strings in the collection evolve independently). Towards solution of this problem, Zhang et al. (2009) states without a proof that under the no-breakpoint-reuse scenario, this problem can be solved by a slight modification of their algorithm, but a more detailed investigation of this problem is warranted.

Acknowledgements. This research was partially supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG, European Community FP7 grants IRG-224885 and IRG-231025, and grant VEGA 1/0210/10.

References

Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: Symposium on Discrete Algorithms (SODA), pp. 819–828 (2000)

Ann, H.-Y., Yang, C.-B., Peng, Y.-H., Liaw, B.-C.: Efficient algorithms for the block edit problems. Information and Computation 208(3), 221–229 (2010)

Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. ACM Transactions on Algorithms 3(1), 1–19 (2007)

Elemento, O., Gascuel, O., Lefranc, M.-P.: Reconstructing the duplication history of tandemly repeated genes. Molecular Biology and Evolution 19(3), 278–278 (2002)

Ergün, F., Muthukrishnan, S.M., Şahinalp, S.C.: Comparing sequences with segment rearrangements. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 183–194. Springer, Heidelberg (2003)

Kahn, C.L., Mozes, S., Raphael, B.J.: Efficient algorithms for analyzing segmental duplications with deletions and inversions in genomes. Algorithms for Molecular Biology 5(1), 11 (2010)

Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM 53(6), 918–936 (2006)

Kasai, T., Lee, G.H., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)

Lajoie, M., Bertrand, D., El-Mabrouk, N., Gascuel, O.: Duplication and inversion history of a tandemly repeated genes family. Journal of Computational Biology 14(4), 462–468 (2007)

Lopresti, D.P., Tomkins, A.: Block edit models for approximate string matching. Theoretical Computer Science 181(1), 159–179 (1997)

Ma, J., Ratan, A., Raney, B.J., Suh, B.B., Miller, W., Haussler, D.: The infinite sites model of genome evolution. Proceeding of the National Academy of Sciences USA 105(38), 14254–14261 (2008)

Mehlhorn, K., Näher, S.: Dynamic fractional cascading. Algorithmica 5(2), 215–241 (1990)

Nadeau, J.H., Taylor, B.A.: Lengths of chromosomal segments conserved since divergence of man and mouse. Proceeding of the National Academy of Sciences USA 81(3), 814–818 (1984)

Shapira, D., Storer, J.A.: Large edit distance with multiple block operations. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 369–377. Springer, Heidelberg (2003)

Shapira, D., Storer, J.A.: Edit distance with move operations. Journal of Discrete Algorithms 5(2), 380–392 (2007)

Song, G., Zhang, L., Vinar, T., Miller, W.: CAGE: Combinatorial Analysis of Gene-cluster Evolution. Journal of Computational Biology 17(9), 1227–1232 (2010)

Vinar, T., Brejova, B., Song, G., Siepel, A.: Reconstructing histories of complex gene clusters on a phylogeny. Journal of Computational Biology 17(9), 1267–1269 (2010)

Zhang, Y., Song, G., Vinar, T., Green, E.D., Siepel, A., Miller, W.: Evolutionary history reconstruction for mammalian complex gene clusters. Journal of Computational Biology 16(8), 1051–1060 (2009)

Near Real-Time Suffix Tree Construction via the Fringe Marked Ancestor Problem*

Dany Breslauer[1] and Giuseppe F. Italiano[2]

[1] Caesarea Rothchild Institute, University of Haifa, Haifa, Israel
[2] Università di Roma "Tor Vergata", Rome, Italy

Abstract. We contribute a further step towards the plausible *real time* construction of suffix trees by presenting an on-line algorithm that spends $O(\log\log n)$ time processing each input symbol and takes $O(n \log\log n)$ time in total. Our results improve on a previously published algorithm that take $O(\log n)$ time per symbol and $O(n \log n)$ time in total. The improvements are achieved using a new data structure for the fringe marked ancestor problem, a special case of the nearest marked ancestor problem, which may be of independent interest.

1 Introduction

The suffix tree is a ubiquitous data structure at the heart of numerous text algorithms. Weiner [25] introduced suffix trees and gave a linear-time on-line algorithm for their reverse right-to-left construction. Ukkonen [24] derived a linear-time left-to-right on-line algorithm that is a close relative of an earlier algorithm by McCreight [20]. The analysis of these three algorithms is amortized and the algorithms may spend up to $O(n)$ time processing some input symbols. Suffix arrays, that were introduced by Manber and Myers [19], provide similar theoretical benefits to suffix trees and are much more efficient in practice thanks to their use of a compact array representation, but lose some of their advantages in the on-line setting. Throughout this paper, unless specified otherwise, we assume that the input alphabet has constant size.

Amir et al. [2,16] reported some progress towards constructing the suffix tree in *real-time*, namely, attempting to limit the time spent while processing each individual input symbol in the worst case. Their algorithm uses balanced search trees to maintain a *balanced indexing structure* that quickly finds the *suffix tree insertion points*, for an input text that is extended from right-to-left over an arbitrarily large but ordered alphabet, spending $O(\log n)$ time processing each symbol and $O(n \log n)$ time in total. They also note that similar results could be derived by using existing more complicated dynamic data structures for searching

* Work partially supported by the European Research Council (ERC) Project SFEROT, by the Israeli Science Foundation Grant 347/09, by the 7th Framework Programme of the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future - From Theory to Design") and by MIUR, the Italian Ministry of Education, University and Research, under Project AlgoDEEP.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 156–167, 2011.
© Springer-Verlag Berlin Heidelberg 2011

multidimensional keys [11,13]. The *suffix tree insertion point* of any given suffix is its longest prefix that has already appeared earlier in the text; it has numerous applications in text algorithms and in data compression.

In related work, Kosaraju [17] and Amir and Nor [3] solve the real-time pattern matching and indexing problems by building a suffix tree in *quasi real-time* using Kosaraju's "candelabra" approach. Although Slisenko [22] claimed to have solved these problems and even to classify all periodicities in a string in real-time, a convincing solution was considered to be an open problem [3,12,17]. *Quasi real-time* means that sufficient parts of the suffix tree are built "just in time" before needed by an algorithm that is traversing the suffix tree *starting from its root*. The size of the candelabra can be quite large, however, and neither algorithm guarantees any meaningful upper bound on the time required to find the insertion points of a specific text suffix, but only that parts of the suffix tree will be completed before reached. Thus, finding the suffix tree insertion points is *at least as hard, if not harder*, than the real-time pattern matching and indexing problems that offer further amortization opportunities, distilling the following question: *is it possible to compute the suffix tree insertion points in real-time*?

We present an algorithm that spends only $O(\log\log n)$ time processing each input symbol and takes $O(n \log\log n)$ time in total, contributing a further step towards the plausible real-time construction of suffix trees. To achieve these bounds, we design a data structure for the dynamic fringe marked ancestor problem, a special case of the nearest marked ancestor problem on trees [1], where the marked nodes form a contiguous subtree. Our data structure, which may be of independent interest, supports updates and queries in worst case $O(\log\log n)$ time. We use our new fringe marked ancestor data structure in an adaptation of Weiner's [25] right-to-left on-line suffix tree construction algorithm, shortcutting the occasional long path traversals using fringe-ancestor queries and de-amortizing certain skipped invisible internal updates over time.

Weiner's [9,14,25] algorithm maintains *suffix links*, which are invisible internal components that are not part of the suffix tree's definition, but are, nonetheless, useful in many applications. Our suffix links, which are related to the edges of the directed acyclic word graph [4,8,9] and were also used by Kosaraju [17] and by Amir and Nor [3], turn out to be extremely helpful in navigating the suffix tree by allowing us to use our fringe marked ancestor data structure instead of the nearest marked ancestor data structure that was used by Breslauer [6] to build the suffix tree of a tree in a related approach that spends $O(\frac{\log n}{\log\log n})$ time per symbol [1]. However, these invisible suffix links need to be individually created and repeatedly updated, tasks that are postponed and later de-amortized over time while *the complete visible suffix tree is available immediately* at all times. The invisible suffix links could be made available "just in time", if required, to an algorithm that traverses the suffix tree starting from its root, similarly to Kosaraju's [17] "quasi" real-time construction. Our use of the word "quasi" in this context has a double meaning here: not only are parts of the construction de-amortized and completed later, but the time spent processing each input symbol is up to $O(\log\log n)$ rather than constant time.

The on-line construction of a suffix tree from left-to-right is a much more complicated matter that requires a suitably defined de-amortization. Now, all the text's suffixes are extended simultaneously, and therefore, the visible suffix tree may undergo many structural changes that are effectuated in large batches inserting multiple new nodes and leaves at once [5,9,14,24]. We use our near real-time adaptation of Weiner's right-to-left algorithm, applied to the reverse left-to-right text, to de-amortize the new node insertions in Ukkonen's [24] left-to-right algorithm. Moreover, by applying our adaptations of both Weiner's and Ukkonen's algorithms to either end of a bi-directionally extended text and to its reverse, we obtain near real-time bi-directional suffix tree and affix tree [23] construction algorithms, enhancing respective results of Inenaga [15] and Maaß [18] by limiting the worst case time processing each individual text symbol.

2 The Fringe Marked Ancestor Problem

The fringe marked ancestor problem is a special case of the nearest marked ancestor problem with the additional restriction that the marked nodes must form a contiguous subtree at the root. Specifically, the fringe marked ancestor problem is concerned with maintaining a rooted tree whose nodes are either marked or unmarked, under an intermixed sequence of the following operations: *make-tree*(x) returns a tree consisting of only an unmarked node x; *insert*(u, x, v) inserts a new node x in the middle of an edge (u, v), where x becomes a child of u, a parent of v and adopts v's marked status; *insert-leaf*(u, x) inserts a new unmarked node x as a child of u; *delete*(u, x, v) deletes node x with an only child v and replaces it with an edge (u, v); *delete-leaf*(x) deletes leaf x from the tree; *mark*(x) marks node x, if x is the root or x's parent is already marked; *unmark*(x) unmarks node x, if x has no marked children; *fringe-ancestor*(x) returns the nearest marked ancestor of x (which is x itself if it is marked).

These operations maintain the invariant that the marked nodes constitute a contiguous subtree at the root, a restriction that enables faster algorithm, circumventing an $\Omega(\frac{\log n}{\log \log n})$ worst case time lower bound for the nearest marked ancestor problem [1]. We do not detail the decremental operations *delete*, *delete-leaf* and *unmark* which are not used in our suffix tree algorithm, but remark that their implementation is analogous to the incremental operations.

Let T be the given tree. We maintain an Euler tour $ET(T)$ of T, as follows. $ET(T)$ is a path that starts and ends at the tree root, and traverses each edge exactly twice, once from the parent to the child and once from the child to the parent, according to a depth-first traversal of the tree. Note that for each edge (x, y) in T there are exactly two corresponding edges in $ET(T)$, and for each node in T of degree k there are exactly k corresponding nodes in $ET(T)$ (except for the root that has $k + 1$).

We store the Euler tour $ET(T)$ in a linear list, such that each tree node in T holds pointers to all its corresponding elements in the linear list, and each edge in T store pointers to its two corresponding edges in $ET(T)$. We maintain this linear list as a dynamic union-split-find data structure, which is capable

of performing the following operations: *add*(x, y) inserts a new element y after element x in $ET(T)$; *remove*(x) deletes element x from $ET(T)$; *split*(x) marks element x if x was not marked already; *union*(x) unmarks element x if x was previously marked; *find*(x) returns the previous marked element (closest to x) in the linear list. Using the data structure by Dietz and Raman [10], each of these five operations can be implemented in $O(\log \log n)$ time in the worst case. In addition, we also maintain in tandem a copy of the tree T in the *least common ancestor (lca)* data structure of Cole and Hariharan [7], which supports *insert*, *insert-leaf*, *delete*, *delete-leaf* and *lca* queries, all in worst-case constant time.

We now show how these data structures are used to implement the operations in the fringe marked ancestor problem. A *make-tree* operation creates a tree consisting of a single node and no edge. The corresponding Euler tour consists of one single element and is initialized in $O(1)$ worst-case time. Operations *insert-leaf* and *insert* require the insertion of one or two elements into the Euler tour $ET(T)$, respectively, and thus are implemented in $O(\log \log n)$ worst-case time with a constant number of *add* operations in the incremental union-split-find data structure. The *lca* data structure is maintained in tandem in $O(1)$ worst-case time for each update. To *mark* a tree node x, we perform a *split* on the *first* element corresponding to x in the Euler tour $ET(T)$ in $O(\log \log n)$ time. The operations *delete-leaf*, *delete*, and *unmark* are implemented analogously.

Finally, the *fringe-ancestor* query is supported through one *find* query in the union-split-find data structure and and one *lca* query in the lca data structure, taking $O(\log \log n)$ time in the worst-case, as the following lemma shows.

Lemma 1. *Let T be a tree, let x be a node of T and let x_0 be the first element of the Euler tour $ET(T)$ corresponding to node x. Let v_0 be the closest marked element to the left of element x_0 in $ET(T)$ and let v be the tree node corresponding to v_0. Then lca(v, x) is the fringe ancestor of x in T.*

Proof. Let y be the fringe ancestor of x in T, and assume by contradiction that $y \neq \text{lca}(v, x)$. Node v_0 is a marked element of $ET(T)$, and thus the corresponding node v in T must be marked. Since the Euler tour $ET(T)$ follows a depth-first visit of tree T, denote by DFS(u) be the depth-first number of node u according to the Euler tour. The fact that v_0 is the closest marked element to the left of x_0 in the Euler tour is equivalent to saying that $\text{DFS}(v) \leq \text{DFS}(x)$ and that no marked node u is such that $\text{DFS}(v) < \text{DFS}(u) \leq \text{DFS}(x)$ (i.e., a depth-first traversal in T enters v before entering x and it does not enter any other marked node while going from v to x). Since lca(v, x) is an ancestor of v and v is marked, lca(v, x) must be marked as well; furthermore, since lca(v, x) is an ancestor of x and $\text{lca}(v, x) \neq y$, lca(v, x) must be a proper ancestor of y, i.e., y must be in the tree path from lca(v, x) and x. But then, the depth-first traversal of T would enter marked node y while going from v to x, clearly a contradiction. □

Thus, we have proved the following theorem.

Theorem 1. *The above data structure solves the fringe marked ancestor problem on an unbounded degree tree in $O(\log \log n)$ worst-case time per operation.*

3 Suffix Trees and Suffix Links

We assume that the reader has some textbook familiarity with suffix trees [9,14]. Given a text $w = w_1 \cdots w_n$ over the alphabet Σ, denote its *reverse* by $\widetilde{w} = w_n \cdots w_1$. The *suffix tree* $\mathcal{T}_w$ is a rooted tree with edges and nodes that are labeled with substrings of w. The suffix tree satisfies the following properties: (1) edges leaving any given node are labeled with non-empty strings v that start with different alphabet symbols (v is a substring of w); (2) each node is labeled with a string v formed by the concatenation of the edge labels on the path from the root to that node (v is a substring of w); (3) each branching internal (non-leaf) node has at least two descendants (the root may be an exception in the degenerate case when a string is empty or it is formed by repetitions of a single alphabet symbol); (4) for each substring v of w, there exists a vertex labeled u, such that v is a prefix of u.

The *locus* of a substring v of w is the unique location in $\mathcal{T}_w$ that is labeled with v. Whenever possible, it is convenient to append at the end of the text w a special unique alphabet symbol \$ which does not appear anywhere within w. This guarantees that $\mathcal{T}_{w\$}$ has exactly $|w| + 1$ leaves that are labeled with all the distinct non-empty suffixes of $w\$$. The number of branching internal nodes is no larger than $|w|$. However, in on-line algorithms that construct the suffix trees for a left-to-right streaming text, it is not possible to append and then remove the special alphabet symbol \$ at each step. Therefore, an on-line algorithm for left-to-right extended text must deal also with suffix tree nodes representing text suffixes which may not be branching out of the tree. The locus of such text suffixes may coincide with an internal branching suffix tree node, but it may also be in the middle of a suffix tree edge; in the latter case, it is called an *implicit node* and it is not represented explicitly. We associate an edge between parent u and child v with the child suffix tree node v. A locus in the middle of such edge is defined by the node v and the slack distance in number of symbol to v.

Weiner [25], McCreight [20] and Ukkonen [24] all augment the suffix tree $\mathcal{T}$ with shortcuts called *suffix links* that are used to efficiently traverse the suffix tree. For a suffix tree node $v = v_1 \cdots v_k$, the M-link $\mathcal{M}(v) = u$, McCreight suffix link (also used by Ukkonen), is defined to be a pointer to the suffix tree node $u = v_2 \cdots v_k$ that is obtained by chopping off the first symbol $a = v_1$ of $v = au$. These M-links are well defined for any branching suffix tree node (except for the root), since if v is a branching suffix tree node, then its suffix u must be a branching suffix tree node as well. However, the M-link $\mathcal{M}(v)$ of a leaf v might be an implicit node, and therefore undefined. Nonetheless, each leaf $v = au$ represents a suffix and if the M-link $\mathcal{M}(v) = u$ is defined then it is also a suffix. Observe that M-links cannot form cycles and each branching suffix tree node (except for the root) has exactly one M-link. Therefore, the M-link form a tree, which becomes a trie when labeled with the chopped first symbols. This trie is a subtree of the suffix trie for the reversed text.

Similarly, the W-Link $\mathcal{W}_a(u) = v$, Weiner's suffix link, of a suffix tree node u and symbol $a \in \Sigma$, is defined to be a pointer to the suffix tree locus labeled $v = au$, obtained by appending the symbol a before u. The W-link is only

defined for those symbols $a \in \Sigma$, such that $v = au$ is a substring of the text w, and undefined otherwise. If $v = au$ is a suffix tree node, then the W-link $\mathcal{W}_a(u) = v$ is called a *hard W-link* and is just the opposite pointer of the M-link $\mathcal{M}(v) = u$. However au may also be the locus in the middle of the edge ending at $vx = aux$, $x \neq \epsilon$, rather than a suffix tree node, in which case it is a *soft W-link* that is defined as a pointer to the shortest extension $vx = aux$ that is a branching suffix tree node or a leaf. Thus, if $\mathcal{W}_a(u)$ is defined, then au is always prefix of the node $\mathcal{W}_a(u)$. Observe that when new nodes are inserted into the suffix tree, if a new node auy is created between au and $\mathcal{W}_a(u) = aux$, then the soft W-link $\mathcal{W}_a(u)$ must be updated to point to auy; hard W-links and their opposite M-links are not affected by the insertion of new nodes.

We sometimes also maintain $\mathcal{T}_{\widetilde{w}}$, the suffix tree of the reversed text $\widetilde{w}$. We will need to identify nodes $v \in \mathcal{T}_w$ with their reverse nodes $\widetilde{v} \in \mathcal{T}_{\widetilde{w}}$. If both $v \in \mathcal{T}_w$ and $\widetilde{v} \in \mathcal{T}_{\widetilde{w}}$ are branching nodes, then we define the R-link to be a pointer from the node v in the suffix tree $\mathcal{T}_w$ to the node $\widetilde{v}$ in the reverse suffix tree $\mathcal{T}_{\widetilde{w}}$. These pointers will be maintained in both directions. For simplicity, we do not define R-links for leaves. The next lemma shows that the W-links are contiguous in the suffix tree, what allows us to use the fringe marked ancestor data structure.

Lemma 2. *If the W-link $\mathcal{W}_a(u)$ is defined for a node $u \in \mathcal{T}_w$ and symbol $a \in \Sigma$, then all ancestors $u' \in \mathcal{T}_w$ of u must also have their W-link $\mathcal{W}_a(u')$ defined.*

Proof. If $u \in \mathcal{T}_w$ has a W-link $\mathcal{W}_a(u)$ defined, then au is a substring of w and any prefix au' of au is also a substring of w. □

4 Right-to-Left Construction

Weiner's [25] on-line algorithm constructs the suffix tree for a text that is extended from right-to-left, where in each step, the existing set of suffixes does not change and only one new suffix, the longest suffix equal to the whole text, is added to the suffix tree. In this case, we can also conveniently assume that the text is terminated with the special unique symbol \$ and therefore, that all suffixes are represented by leaves.

Suppose that the text $w\$$ is extended from right-to-left with the next alphabet symbol $a \in \Sigma$. Then, the suffix tree $\mathcal{T}_{w\$}$ has to be updated to become $\mathcal{T}_{aw\$}$ by inserting the new leaf $aw\$$ hanging off some internal branching node v, that might already exist in $\mathcal{T}_{w\$}$ or might need to be inserted as well. Observe that the insertion point v is the longest prefix of the text $aw\$$ that is equal to a substring of $w\$$, also called sometimes the *longest repeated prefix*. Unless v is the suffix tree root, which may only happen if $a \in \Sigma$ is a new alphabet symbol never seen before ($aw\$$ will be hanging off the root), the suffix tree $\mathcal{T}_{aw\$}$ must also contain the branching node $u = \mathcal{M}(v)$. Furthermore, this node u was already in the suffix tree $\mathcal{T}_{w\$}$ before $w\$$ was extended to $aw\$$. Observe that u is an ancestor and a prefix of $w\$$. Moreover, $u \in \mathcal{T}_{w\$}$ is the deepest ancestor and longest prefix of $w\$$, that has the W-link $v'' = \mathcal{W}_a(u)$ defined. The possibly new node $v \in \mathcal{T}_{aw\$}$ is an ancestor of v''.

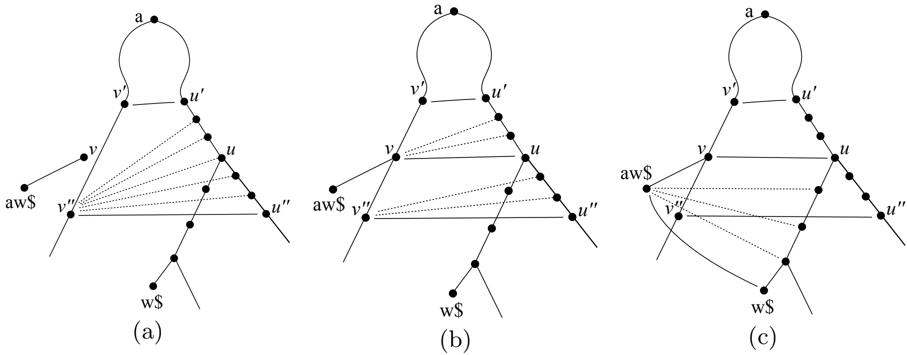


Fig. 1. Extending $\mathcal{T}_{w\$}$ to $\mathcal{T}_{aw\$}$. (a) W-links before extending. (b) W-links to v'' from all nodes on the path between u up to u' are adjusted to point to v instead. (c) New W-links to $aw\$$ are created from all nodes on the path between $w\$$ up to u. Observe that each one of these adjusted (new and updated) W-links is at a different suffix tree depth. The new node v also adopts all the outgoing W-links of v'' (not shown).

To extend $\mathcal{T}_{w\$}$ into $\mathcal{T}_{aw\$}$ the algorithm finds the node u by tracing the path from the leaf $w\$$ towards the root until the first ancestor u of $w\$$ whose W-link $\mathcal{W}_a(u)$ is defined. If $v'' = \mathcal{W}_a(u)$ is a soft W-link, then a new branching node v, such that $\mathcal{M}(v) = u$, is also created on the edge between v'' and its parent v' in the suffix tree $\mathcal{T}_{w\$}$, and its W-links are initialized to be the same as its child v''. If $v'' = \mathcal{W}_a(u)$ is a hard W-link, then the node $v = v' = v''$ already exists. Now, the algorithm hangs the new leaf $aw\$$ off the branching node v. See Figure 1.

We must not forget the W-links that need to be created and updated along the way. The nodes on the path between $w\$$ up to and excluding u, will now have new W-links defined $\mathcal{W}_a(x) = aw\$$ (all soft, except the one hard $\mathcal{W}_a(w\$) = aw\$$). The nodes on the path between u up to and excluding u', will now have their W-links updated from v'' to $\mathcal{W}_a(x) = v$ (all soft, except the one hard $\mathcal{W}_a(u) = v$). Observe that each one of these new and updated W-links is at a different suffix tree depth. In addition, if a new branching node v was created, then v also adopts all the W-links of v'' (all adopted W-links become soft).

The amortized analysis of Weiner's algorithm is based on the fact that the suffix tree depth of $aw\$$ is by at most one larger than the suffix tree depth of $w\$$. This is the case because each ancestor of $aw\$$, except for the root, has an M-link that points to a different ancestor of $w\$$. Thus, the number of steps traversing the path from $w\$$ towards the root to find u is bounded by the depth reduction and the overall depth increases throughout the algorithm are bounded by the number of text symbols. To get the near real-time adaptation of Weiner's algorithm, we maintain a *de-amortization stack* with the postponed W-link adjustment (new W-link insertion and existing W-link update) tasks.

Theorem 2. *We can adapt Weiner's right-to-left on-line suffix tree algorithm over constant size alphabets to take up to $O(\log\log n)$ time processing each input symbol and spend $O(n \log\log n)$ time in total.*

Proof. For each alphabet symbol $a \in \Sigma$, we maintain in tandem a separate fringe marked ancestor data structure mirroring the suffix tree, where a node u is marked if and only if the W-link $\mathcal{W}_a(u)$ is defined. By Lemma 2, the W-links are contiguous and the fringe marked ancestor data structure may be used. Over constant size alphabets, all these data structures are updated in additional $O(\log \log n)$ worst case time per each new suffix node and leaf and each new W-link (W-link updates do not affect the fringe marked ancestor data structure). Thus, we created an alternative mechanism to find the suffix tree insertion point instead of tracing the path to the root; we use symbol a's fringe marked ancestor data structure to directly find the nearest ancestor u of $w\$$ that is marked, or in other words, has W-link $\mathcal{W}_a(u)$ defined, in $O(\log \log n)$ worst case time.

While this allows us to insert the new leaf and branching node quickly (with their associated M-links, opposite hard W-links and the adopted W-links), we also need to create the new soft W-links on the path between $w\$$ and u and update the existing soft W-links on the path between u and u'. We maintain a de-amortization stack for these soft W-link adjustment tasks, and execute these tasks later, adjusting the W-links from shallow to deep. Since the depth of the suffix tree insertion point increases at most by one in each step, if we update at least two or more W-links from the de-amortization stack at each step, we guarantee that the depths of the remaining pending W-link adjustment tasks on the stack are strictly increasing. □

The hard W-links (opposite M-links) were updated immediately for each new branching node and new leaf. An algorithm may also wish to gain access to the internal invisible soft W-links through “just in time” de-amortizaion.

Corollary 1. *An algorithm may access the internal invisible W-links that will be available “just in time”, if it executes the delayed W-link adjustment tasks from the de-amortization stack, provided that such algorithm traverses the suffix tree starting from the root.*

Remark. The common textbook description of Weiner's [9,14] algorithm, that can probably be traced back to Seiferas' [21] simplified presentation, uses Boolean indicator variables instead of soft W-links (Figure 2a-b; the fringe marked ancestor data structure is actually used to maintain these indicators). This is very appealing since neither hard W-links nor Boolean indicators need to be updated once set, unlike soft W-links that require constant maintenance. We wish to point out a rather trivial observation, that the Boolean indicators are not required at all and it suffices to maintain only the hard W-links and use a second quick scan to locate the suffix tree insertion point v, borrowing a technique from McCreight's [20] and Ukkonen's [24] algorithms. See Figure 2c, where the edge between v' and its child v'' corresponds to the path between $u' = \mathcal{M}(v')$ and $u'' = \mathcal{M}(v'')$; $u = \mathcal{M}(v) = \text{lca}(w\$, u'')$ and the edge labels on the path between u' and u are equal to the corresponding parts of the edge label between v' and v. Thus, u is the most shallow node on this path between u' and u'', where the first symbol on the edge at u is not equal to its corresponding symbol on the edge between v' and v''.

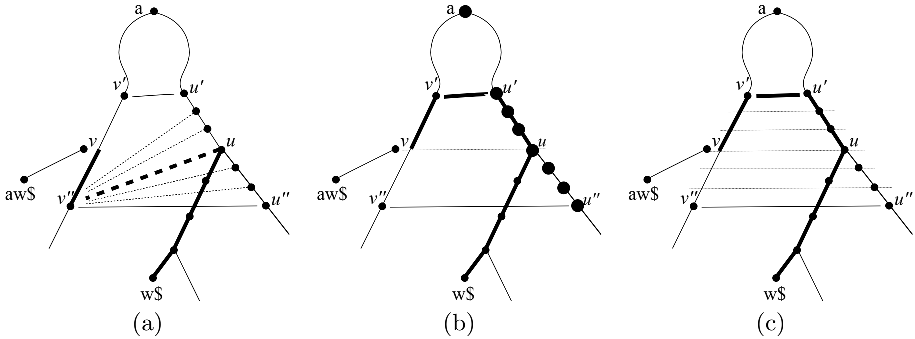


Fig. 2. Finding the insertion point v. (a) Soft and hard W-links: follow the first ancestor u with W-link $v'' = \mathcal{W}_a(u)$ defined. (b) Hard W-links and indicators (nodes with indicators shown as thicker dots): follow the first ancestor u' with hard W-link $v' = \mathcal{W}_a(u')$ defined, using the offset of the first ancestor u with set indicator. (c) Only hard W-links: follow the first ancestor u' with hard W-link $v' = \mathcal{W}_a(u')$ defined and compute the offset of v from v' in a second scan.

5 Left-to-Right and Bi-directional Construction

Constructing the suffix tree from left-to-right is a much more complicated matter that requires suitably defined de-amortization, because all the text's suffixes are simultaneously extended, and therefore, the visible suffix tree may undergo many structural changes that are effectuated in large batches inserting multiple new nodes and leaves [5,9,14]. We briefly review Ukkonen's [24] left-to-right algorithm and use our adaptation of Weiner's right-to-left algorithm, applied to the reverse left-to-right text, to de-amortize the new node insertion in Ukkonen's algorithm. Due to tight space we skimp on details in this conference abstract.

Ukkonen observed that once some text suffix is a leaf, it will remain forever a leaf after all future left-to-right extensions. By labeling the external suffix tree edges leading to leaves "open ended", reaching to the current *growing* end of the text, Ukkonen invented an automatic gratuitous extension mechanism for these edge labels. Unfortunately, the remaining text suffixes are called "implicit nodes" and do not branch out. The algorithm maintains the *active suffix*, the longest suffix of the text that has not branched out to become a leaf, which is the *longest repeated suffix* of the text that appeared previously in the text. If the active suffix cannot be extended within the suffix tree, an *insertion batch* creates leaves for all suffixes between the old active suffix and up to (excluding) the new active suffix. In particular: those implicit nodes that were in the middle of an edge must branch out by creating a branching node splitting the edge and inserting a leaf; and those implicit nodes that coincided with an existing explicit node branch out by creating a leaf hanging off the existing explicit node.

Ukkonen's algorithm maintains the locus of current active suffix, the longest repeated suffix, that is updated while tracing the suffix tree, selecting the appropriate branch at each internal suffix tree node according to the first symbol

on the branching edges. During an insertion batch, the algorithm follows the M-links to find the next active suffix. While following suffix links, more suffix tree nodes may appear on the path between the suffix tree node and the offset representing the implicit node's locus and the representation has to be updated to the *canonical* representation specifying implicit nodes' locus by their offset relative to the beginning of the edge where they are located. Ukkonen's algorithm, like the algorithm by McCreight, only has to navigate the suffix tree by selecting the edges at each branching node according to their first branching symbol, quickly moving down the suffix tree path towards the implicit node. The total amount of work is amortized to linear time, but an insertion batch might be very long.

To get the quasi real-time adaptation of Ukkonen's algorithm, we maintain a *de-amortization stack* with the postponed suffix tree node insertion tasks. We execute Weiner's right-to-left algorithm simultaneously, applied to the reverse left-to-right text, and shortcut long insertion batches in Ukkonen's algorithm by effectuating the node insertion in reverse to de-amortize the insertions. We rely on the following trivial observation.

Lemma 3. *The active point in Ukkonen's algorithm is the reverse of the insertion point in Weiner's right-to-left algorithm applied to the reverse text.*

Proof. The active point v is the longest repeated suffix in the text $\$wa$ that also appeared earlier in $\$w$. That is, the insertion point $\widetilde{v}$ is the same as the longest repeated prefix of the reversed text $a\widetilde{w}\$$ that appeared earlier in $\widetilde{w}\$$. □

Thus, by maintaining R-links from the right-to-left suffix tree $\mathcal{T}_{\widetilde{w}\$}$ to the left-to-right suffix tree $\mathcal{T}_{\$w}$, we can use Weiner's insertion point information to jump ahead to the end of Ukkonen's insertion batch and perform the node insertions in Ukkonen's left-to-right suffix tree in reverse order from short to long insertion points, de-amortizing the insertions over time. This is done by maintaining a de-amortization stack with delayed insertions at lengths that are strictly increasing. Since the lengths of the active suffix increases at most by one with each input symbol, if we insert at least one pending suffix from the de-amortization stack we guarantee that the lengths of the pending suffix insertion tasks on the stack are strictly increasing. Special attention is also required to the existing nodes which are traversed between the newly inserted nodes.

Lemma 4. *We can find Ukkonen's new active point in $\mathcal{T}_{\$wa}$ in constant time with the assistance of Weiner's reverse suffix tree $\mathcal{T}_{a\widetilde{w}\$}$.*

Proof. First, apply the text extension to Weiner's algorithm to get $\mathcal{T}_{a\widetilde{w}\$}$. There are two cases. If the new insertion point v is exactly one symbol longer than the old insertion point, then the old insertion point was u, the immediate ancestor of $\widetilde{w}\$$ with defined W-link $\mathcal{W}_a(u) = v$. Ukkonen's active point's length also increases by one and its locus is computed by moving ahead in the suffix tree $\mathcal{T}_{\$w}$. Otherwise, the search for the ancestor u with defined W-link $\mathcal{W}_a(u)$ took a few steps up the reverse suffix tree $\mathcal{T}_{a\widetilde{w}\$}$. Observe that in this case the node u in Weiner's algorithm not only has the W-link $\mathcal{W}_a(u)$ defined, but also some

W-link $\mathcal{W}_b(u)$ defined, for another alphabet symbol $b \neq a$, and therefore, $\widetilde{u}$ must be a branching node in Ukkonen's suffix tree $\mathcal{T}_{\$w}$. We get to the node $\widetilde{u}$ in $\mathcal{T}_{\$w}$ by following the R-link from u in the reverse suffix tree $\mathcal{T}_{a\widetilde{w}\$}$ to $\mathcal{T}_{\$w}$, and then advance to the locus of $\widetilde{u}a$ by following the alphabet symbol a in $\mathcal{T}_{\$w}$. □

To be able to use the previous lemma, it is crucial to correctly maintain the R-links from the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$ to $\mathcal{T}_{\$w}$. When Weiner's algorithm inserts a new node $\widetilde{v}$ into the reverse suffix tree $\mathcal{T}_{\widetilde{w}\$}$, Lemma 4 gives us the corresponding locus of the Ukkonen's new active point in $\mathcal{T}_{\$w}$ and if it is a node, then we set the R-link accordingly. An insertion batch in Ukkonen's algorithm is de-amortized, but since we have the new active point, we can proceed while de-amortizing the inserts, but still set the R-links when the nodes are eventually inserted from shallow to deep. The corresponding loci in Weiner's reverse suffix tree are contiguous ancestor loci of Weiner's old insertion point, between the node u that had its W-link set and the old insertion point of $\widetilde{w}\$$.

Thus, we can prove the following theorem.

Theorem 3. *An algorithm may access the suffix tree constructed for the left-to-right extended text and its internal invisible M-links (opposite hard W-links). The suffix tree and the M-links will be available just in time, if the algorithm executes the delayed update tasks from the de-amortization stack, provided that such algorithm traverses the suffix tree starting from the root by increasing the its traversal locus by one symbol in each step.*

Inenaga [15] and Maaß [18] combined Weiner's and Ukkonen's algorithms to obtain bi-directional suffix tree and affix tree [23] construction algorithms. Combining our adaptations with Inenaga's and Maaß' observations, we can symbiotically construct the suffix tree of a bi-directionally text and of the reverse text, yielding also an affix tree construction algorithm.

Remark. Weiner's algorithm immediately inserts each new text suffix and proceeds with suffix tree depth amortization. In contrast, McCreight's and Ukkonen's algorithms hold back on inserting some suffixes and proceed with trie depth (length in symbols) amortization that is intermixed with suffix tree depth arguments, and therefore, the de-amortization here requires that the traversal locus' length increases by one symbol rather than depth increases by one edge.

Acknowledgments. We thank Amir Ben-Amram, Johannes Fischer, Roberto Grossi, Gadi Landau and Oren Weimann for discussions about this work and the anonymous reviewers for their useful comments.

References

1. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: FOCS, pp. 534–544 (1998)
2. Amir, A., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Towards real-time suffix tree construction. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 67–78. Springer, Heidelberg (2005)

3. Amir, A., Nor, I.: Real-time indexing over fixed finite alphabets. In: Teng, S.H. (ed.) SODA, pp. 1086–1095. SIAM, Philadelphia (2008)
4. Blumer, A., Blumer, J., Haussler, D., McConnel, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. Assoc. Comput. Mach. 34(3), 578–595 (1987)
5. Breslauer, D., Italiano, G.F.: On Suffix Extensions in Suffix Trees (2011), this conference proceedings
6. Breslauer, D.: The Suffix Tree of a Tree and Minimizing Sequential Transducers. Theor. Comput. Sci. 191(1-2), 131–144 (1998)
7. Cole, R., Hariharan, R.: Dynamic LCA Queries on Trees. SIAM J. Comput. 34(4), 894–923 (2005)
8. Crochemore, M.: Transducers and repetitions. Theoret. Comput. Sci. 12, 63–86 (1986)
9. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, Oxford (1994)
10. Dietz, P.F., Raman, R.: Persistence, amortization and randomization. In: SODA, pp. 78–88 (1991)
11. Franceschini, G., Grossi, R.: A general technique for managing strings in comparison-driven data structures. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 606–617. Springer, Heidelberg (2004)
12. Galil, Z.: Open problems in stringology. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO ASI Series F, vol. 12, pp. 1–8. Springer, Berlin (1984)
13. Grossi, R., Italiano, G.F.: Efficient techniques for maintaining multidimensional keys in linked data structures. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 372–381. Springer, Heidelberg (1999)
14. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
15. Inenaga, S.: Bidirectional construction of suffix trees. Nord. J. Comput. 10(1), 52–67 (2003)
16. Kopelowitz, T.: From Off-line to On-line Indexing Data-Structures. Ph.D. thesis, Dept. of Computer Science, Bar-Ilan University (2011)
17. Kosaraju, S.R.: Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In: STOC, pp. 310–316 (1994)
18. Maaß, M.G.: Linear bidirectional on-line construction of affix trees. Algorithmica 37(1), 43–74 (2003)
19. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
20. McCreight, E.: A space economical suffix tree construction algorithm. J. Assoc. Comput. Mach. 23, 262–272 (1976)
21. Seiferas, J.: Subword Trees (undated manuscript)
22. Slisenko, A.: Detection of periodicities and string-matching in real time. J. of Soviet Mathematics, 1316–1386 (1983)
23. Stoye, J.: Affix Trees. Master's thesis, Technische Fakultat, Universitat Bielefeld, Bielefeld, Germany (2000), report 2000-04
24. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
25. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Approximations and Partial Solutions for the Consensus Sequence Problem

Amihood Amir[1,2,⋆], Haim Paryenty[1,⋆⋆], and Liam Roditty[1]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
{amir,liamr}@cs.biu.ac.il, haimpa@gmail.com

[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218

Abstract. The problem of finding the consensus of a given set of strings is formally defined as follows: given a set of strings $S = \{s_1, \ldots s_k\}$, and a constant d, find, if it exists, a string s^*, such that the Hamming distance of s^* from each of the strings does not exceed d.

In this paper we study an LP relaxation for the problem. We prove an additive upper bound, depending only in the number of strings k, and randomized bounds. We show that empirical results are much better. We also compare our program with some algorithms reported in the literature, and it is shown to perform well.

1 Introduction and Related Work

The *Consensus Sequence problem* is a basic problem in the area of sequence comparison. Formally the Consensus Sequence problem asks, given a parameter d and a set of sequences $S = \{s_1, ..., s_k\}$ each of length ℓ, whether there exists a sequence s^*, called a *consensus*, that is of distance at most d from each sequence in S. The consensus sequence need not be contained in S. The distance metric between two strings s and t has historically been the Hamming distance, $H(s, t)$. The Consensus Sequence Problem is $\mathcal{NP}$-complete, even when the characters in the strings are drawn from the binary alphabet.

Thus, attention has been restricted to approximation solutions [1, 3, 4, 9–12] and fixed-parameter solutions [5–7, 12]. Furthermore, some efficient algorithms for a small constant k have been developed [5, 8, 2]. For brief surveys on the approximation solutions, readers are referred to [2, 12].

There has been activity in seeking practical implementations for the Consensus Sequence problem. Two major directions were attempted: Various successful Integer Linear Programming (ILP) approaches [13, 1], and pruning a search tree [14]. The ILP algorithms work well for strings of relatively small length. The search tree algorithms do not work when the ratio between the number of errors d and the string length ℓ is large. Thus a general practical consensus sequence algorithm is still an elusive goal.

⋆ Partly supported by NSF grant CCR-09-04581 and ISF grant 347/09.

⋆⋆ Partly supported by a BIU President Fellowship. This work is part of H. Paryenty's Ph.D. thesis.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 168–173, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The main contribution of this paper is by proposing and analysing a *Linear Programming (LP) relaxation* for the problem. Recently, [13] also showed an LP relaxation. However, we use a different Linear Program, which requires a smaller number of variables, thus our algorithm runs faster.

In Section 2 we present two approximation algorithms based on LP relaxation. The first algorithm is deterministic. It computes an approximated consensus with an additive error of at most $(k-1)(k!-1)$ when k is the number of input strings. The second algorithm is randomized. It computes, whp, an approximated consensus with an additive error of at most $O(\sqrt{d \log k})$ when d is the maximal distance of the optimal consensus from the input strings. In practice, our algorithms run very fast even for very large inputs, and the results prove much better than the additive worst case upper bound, in fact it rarely exceeds a Hamming distance 2 from the optimal consensus sequence.

In Section 3 we present the results of an extensive empirical study on the LP based algorithms presented in Section 2 We show the high quality of the approximation result. We also compare our algorithms to several other algorithms reported recently in the literature. Our experiments indicate that the running times of our algorithms are better.

Notation: Throughout this paper, we will be considering our input as a matrix. If the set S has k strings, $s_1, ..., s_k$, each of length ℓ, we view S as a $k \times \ell$ matrix. We can thus refer to columns and rows. We will say *column* to denote a column of this matrix, and *row* to denote a row in the matrix. Thus, e.g., the element is the second row of column i, is the ith symbol of s_2. Each column has a corresponding entry in the consensus sequence, which we call the column's *consensus value*. The distance of a string $\hat{s}$ from S is $\max_{s \in S} d(\hat{s}, s)$. Given a string s we denote with $c_i(s)$ the character in the ith position of s.

We refer to two identical columns as having the same column type. We denote with $T(S)$ the column types of S. For a column type $t \in T(S)$ we denote with $\#t$ the number of columns of type t in S. For a string s and column type $t \in T(S)$ we denote with $s_{|t}$ the string that we get from s when we delete all characters but those at locations of column type t.

2 An Approximation Algorithm for the Consensus Problem

In this section we present Linear Programming based algorithms to approximate the consensus sequence. The main result is an algorithm with an additive approximation. Let S be a set of k strings over alphabet Σ and let $T(S)$ be its column types. Let d^* be the distance of the optimal consensus from S. Our first algorithm computes an approximated consensus with distance of at most $d^*+(k-1)|T(S)|$ from S. Our second algorithm computes, whp, an approximated consensus with distance of at most $d^* + O(\sqrt{d^* \log k})$ from S.

2.1 Integer Programs

Ben-Dor et al. [1] suggested an Integer Program for the consensus problem. The number of variables in their IP is $|\Sigma| \cdot \ell$. Ben-Dor et al. [1] used LP relaxation together with randomized rounding to solve their program and then to convert the fractional solution into an integral one. Recently, Chimani et al. [13] showed that it is possible to slightly reduce the number of variables to $|\Sigma| \cdot (\ell - 1)$.

Gramm et al. [6] suggested the following, different, Integer Programming formulation to this problem:

$$\begin{aligned}
&\min d \\
&\text{s.t. } \sum_{t \in T(S)} \sum_{\sigma \in \Sigma \setminus \{\sigma_{t,i}\}} x_{t,\sigma} \leq d && \forall i \in \{1, \dots, k\} \\
&\qquad \sum_{\sigma \in \Sigma} x_{t,\sigma} = \#t && \forall t \in T(S) \\
&\qquad x_{t,\sigma} \in \{0, 1, \dots, \#t\} && \forall \sigma \in \Sigma, \forall t \in T(S)
\end{aligned}$$

In this formulation, for every $t \in T(S)$ and $\sigma \in \Sigma$, the variable $x_{t,\sigma}$ is the number of occurrences that σ has in the consensus sequence at locations that correspond to column type t. Let $s_i \in S$ and let $\sigma_{t,i}$ be the character of string s_i in column type t (note that s_i has the same character in all the locations that correspond to column type t). The distance of s_i from the consensus sequence is $\sum_{t \in T(S)} \sum_{\sigma \in \Sigma \setminus \{\sigma_{t,i}\}} x_{t,\sigma}$, that is, for each column type t we sum the characters in the consensus sequence at locations that correspond to t that are different from the character s_i has in these locations.

Gramm et al. [6] solved the decision version of this IP directly using the algorithm of Lenstra [15] that has an exponential dependency in the number of variables. Thus, they were not able to solve the IP for more than four strings.

2.2 Linear Program Relaxation

We present a LP relaxation for the IP suggested by Gramm et al. [6] and two different rounding techniques. The only change in the LP with respect to the IP is that for every $t \in T(S)$ and $\sigma \in \Sigma$ the variable $x_{t,\sigma}$ is allowed to be a real non-negative number. This means that for alphabet $\Sigma = \{\sigma^1, \sigma^2, \sigma^3, \sigma^4\}$ and a set of strings S that has a certain column type t that repeats 14 times it might be that $x_{t,\sigma^1} = 2.3$, $x_{t,\sigma^2} = 3.8$, $x_{t,\sigma^3} = 4.6$ and $x_{t,\sigma^4} = 3.3$.

Let $a_{t,\sigma}$ be the value of $x_{t,\sigma}$ for every column type t and character $\sigma \in \Sigma$ in the LP optimal solution. In our first rounding procedure for every column type t and character $\sigma \in \Sigma$ we set $\hat{a}_{t,\sigma} = \lfloor a_{t,\sigma} \rfloor$. As there are $|\Sigma|$ variables for column type t we are left with at most $|\Sigma| - 1$ locations of column type t in the consensus sequence to fill. We then fill the empty places with characters in a non-increasing order of the size of their fractional part, $a_{t,\sigma} - \hat{a}_{t,\sigma}$, starting from the character with the largest fractional part. We increment $\hat{a}_{t,\sigma}$ by one for every character σ that is chosen in this process. We denote with $\hat{d}$ the value of the integral solution obtained by this rounding process. The lemma below bounds the error of this rounding process. Its proof will appear in the journal version.

Lemma 1. $\hat{d} \leq d^* + (k-1)|T(S)|$.

For k strings the number of different column types is at most the bell number $B(k) \leq k!$. Hence, our additive error is at most $(k-1)k!$. In practice the additive error is rarely more than 2 as we show in Section 3.

Another approach to obtain an integral solution is to use randomized rounding in a similar manner to Ben-Dor et al. [1]. As before, let $a_{t,\sigma}$ be the value of $x_{t,\sigma}$ for every column type t and every character $\sigma \in \Sigma$ in the LP optimal solution. The probability of a character σ to be placed in the consensus sequence at a location of column type t is $\frac{a_{t,\sigma}}{\#t}$. The expected distance of the consensus string from an arbitrary string of the input is presented in the theorem below, whose proof is ommitted for lack of space.

Theorem 1. *Let S be a set of k strings, and let s^* be an optimal consensus. Let d^* be the distance between S and s^*. Let $\hat{s}$ be the approximated consensus obtained using randomized rounding and let $\hat{d}$ be its distance from S. Then $Pr(\hat{d} > d^* + \sqrt{3d^* \log \frac{k}{\epsilon}}) < \epsilon$, where $\epsilon > 0$ is a constant.*

3 Empirical Results

All algorithms have been implemented with Visual C++ 2008 Compiler. Our experiments were done on a Intel Centrino T6600, 2.20 GHz, 4 GB RAM running Windows 7 in 32 bit mode. We used GNU GLPK as our LP and ILP solver. We applied a time limit, as usual in ILP approaches, of 30 minutes as used by [13].

We used the data set of Hufsky et al. [14], as was conveniently organize by [13]. This is a biological data set that was obtained using approximate gene clustering to five γ-proteobacteria genomes (see Hufsky et al. [14] for more details).

We have implemented five algorithms:

1. The ILP algorithm suggested by Ben-Dor et al. [1] (BD-IP). In this ILP there is a binary variable that corresponds to every character from the alphabet and location in the string.
2. The LP relaxation of Ben-Dor et al. [1] (BD-LP) using standard randomized rounding to obtain an integral solution.
3. The ILP algorithm suggested by Gramm et al. [6] (GR-IP). In this ILP there is a variable that corresponds to every character from the alphabet and column type (rather than any possible location). For more details, see Section 2
4. An LP relaxation of Gramm et al. [6], (NEW-LP-1) with randomized rounding to obtain an integral solution. For more details, see Section 2
5. An LP relaxation of Gramm et al. [6], (NEW-LP-2) where fractional variables are rounded using a simple rounding procedure. For more details, see Section 2.

In [13] the first two algorithms are implemented and tested. We have conducted extensive experiments using all these algorithms. One goal was to analyze the quality of the approximation produced by our suggested algorithms (4 and 5). Another goal was to compare between the running time of these five algorithms.

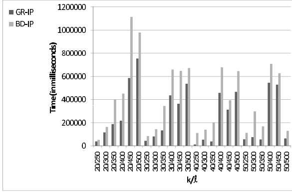


Fig. 1. Comparing the running time of GR-IP and BD-IP

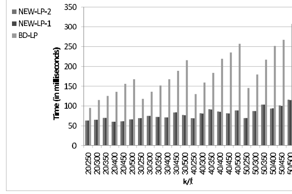


Fig. 2. Comparing BD-IP, NEW-LP-1 and NEW-LP-2

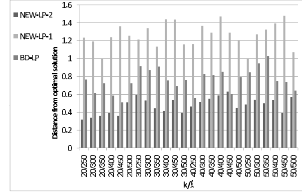


Fig. 3. Comparing approximated and exact consensus

We started by comparing between the running time of BD-IP and GR-IP. In Figure 1 we present the average time (in milliseconds) that it took for each algorithm to compute the consensus with respect to the number of strings in the input and their length. As can be seen GR-IP is much faster than BD-IP. The reason for that is that the number of variables used by GR-IP is much smaller than the number of variables used by BD-IP. While BD-IP has a variable for each character and each location in the string (i.e, $|\Sigma| \times \ell$ variables), in GR-IP there is a variable for each character and each column type.

Next, we compared between the running time of BD-LP, NEW-LP-1 and NEW-LP-2. In Figure 2 we present the average time (in milliseconds) that it took for each algorithm to compute the consensus with respect to the number of strings in the input and their length. Algorithms NEW-LP-1 and NEW-LP-2 were significantly better than BD-LP. As before, the reason for that is that these algorithms are using an LP with much less variables than BD-LP. The performance of NEW-LP-1 and NEW-LP-2 were pretty much the same, since the difference between them is only in the rounding, and the running time is dominated by solving the LP.

It follows from the above experimental results that the LP-based algorithms are much faster than the IP-based algorithms. This is not surprising as in LP-based algorithms we compute a fractional solution rather than integral and the algorithms for that are much faster. The main question is whether we pay for this time efficiency in the quality of the approximated consensus that these algorithms compute. The theoretical results of Section 2 suggest that the approximation would be good. Testing the theoretical results, was another goal of our experiments.

In Figure 3 we present a comparison between the approximated consensus computed by the LP-based algorithm to the optimal consensus as found by the IP algorithm. In particular, we take the approximated consensus and find the string that is farthest from it. We do the same with the optimal consensus. The difference between these two values is the error of the approximated consensus. We compute the average difference for a given number of strings and their length. The results of NEW-LP-2 were the best. The results of BD-LP are better than the results of NEW-LP-1. This may be explained by the fact that exploiting the freedom of rounding every column separately improves the accuracy with respect to the case in which a bundle of columns are rounded together.

References

1. Ben-Dor, A., Lancia, G., Perone, J., Ravi, R.: Banishing bias from consensus sequences. In: Proceedings of the 8th Symposium on Combinatorial Pattern Matching, pp. 247–261 (1997)
2. Boucher, C., Brown, D.G., Durocher, S.: On the structure of small motif recognition instances. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 269–281. Springer, Heidelberg (2008)
3. Gasieniec, L., Jansson, J., Lingas, A.: Efficient approximation algorithms for the Hamming center problem. In: Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms, pp. 905–906 (1999)
4. Gasieniec, L., Jansson, J., Lingas, A.: Approximation algorithms for Hamming clustering problems 2, 289–301 (2004)
5. Gramm, J., Niedermeier, R., Rossmanith, P.: Exact solutions for closest string and related problems. In: Proceedings of the 12th International Symposium on Algorithms and Computation, pp. 441–453 (2001)
6. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37(1), 25–42 (2003)
7. Stojanovic, N., Berman, P., Gumucio, D., Hardison, R., Miller, W.: A linear-time algorithm for the 1-mismatch problem. In: Rau-Chaplin, A., Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 1997. LNCS, vol. 1272, pp. 126–135. Springer, Heidelberg (1997)
8. Sze, S.-H., Lu, S., Chen, J.: Integrating sample-driven and pattern-driven approaches in motif finding. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 438–449. Springer, Heidelberg (2004)
9. Lanctot, K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. In: Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms, pp. 633–642 (1999)
10. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. In: Proceedings of the 31st Annual ACM Symposium on Theory of Computing, pp. 473–482 (1999)
11. Li, M., Ma, B., Wang, L.: On the closest string and substring problems. Journal of the ACM 49(2), 157–171 (2002)
12. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 396–409. Springer, Heidelberg (2008)
13. Chimani, M., Woste, M., Bocker, S.: A Closer Look at the Closest String and Closest Substring Problem, pp. 13–24. ALENEX (2011)
14. Hufsky, F., Kuchenbecker, L., Jahn, K., Stoye, J., Böcker, S.: Swiftly computing center strings. In: Moulton, V., Singh, M. (eds.) WABI 2010. LNCS, vol. 6293, pp. 325–336. Springer, Heidelberg (2010)
15. Lenstra, H.W.: Integer programming with a fixed number of variables. Mathematics of Operations Research 8, 538–548 (1983)

Fixed Block Compression Boosting in FM-Indexes*

Juha Kärkkäinen[1] and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Helsinki, Finland
juha.karkkainen@cs.helsinki.fi
[2] Department of Informatics, King's College London, London, United Kingdom
simon.puglisi@kcl.ac.uk

Abstract. A compressed full-text self-index occupies space close to that of the compressed text and simultaneously allows fast pattern matching and random access to the underlying text. Among the best compressed self-indexes, in theory and in practice, are several members of the FM-index family. In this paper, we describe new FM-index variants that combine nice theoretical properties, simple implementation and improved practical performance. Our main result is a new technique called *fixed block compression boosting*, which is a simpler and faster alternative to optimal compression boosting and implicit compression boosting used in previous FM-indexes.

1 Introduction

A compressed full-text self-index [13] of a text string T is a data structure that stores T in a compressed form that allows fast random access to T and also supports fast pattern matching queries. We focus here on the *count* query that, given a pattern string P, returns the number of occurrences of P in T. [1] Many of the best compressed self-indexes, in theory and in practice, belong to the FM-index family originating from the FM-index of Ferragina and Manzini [5]. In particular, they combine good compression with fast count queries [6,11,4,2]. In this paper, we describe new variants of the FM-family achieving even better compression and faster count queries.

The main components of most FM-indexes are:

– The *Burrows–Wheeler transform* (BWT) [1]: an invertible permutation of the text T. A procedure called *backward search* [5] turns a count query on T into a sequence of *rank* queries on the BWT.
– The *wavelet tree* [7]: a representation of the BWT that turns a BWT rank query into a sequence of rank queries on bitvectors.
– A bitvector *rank index*, which supports fast rank queries on bitvectors.

* Juha Kärkkäinen is supported by Academy of Finland grant 118653 (ALGODAN). Simon J. Puglisi is supported by a Newton Fellowship.

[1] Our indexes support other common queries such as *locate* and *extract*, but the algorithmic and implementation issues in engineering them are quite different and outside the scope of this paper.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 174–184, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The total length of standard wavelet tree bitvectors is equal to the size of the original, uncompressed text in bits. All other data structures can be fitted in less space: asymptotically less in theory, and significantly less in practice. Basic zero-order compression is achieved either with compressed bitvector rank structures, such as RRR [15], or Huffman-shaped wavelet trees [8]. For higher order compression, we can use a technique called *compression boosting* [3,6], where the BWT is partitioned into blocks of varying sizes based on the context of symbols in T, and there is a separate, zero-order compressed wavelet tree for each block. An optimal partitioning into context blocks can be found in linear time [3].

Our main result is a technique called *fixed block compression boosting*. It is similar to context block boosting, but divides the BWT into blocks of fixed sizes without any regard to the symbol contexts. Such a division is inoptimal, but we show that it cannot be much worse than the optimal one. What we gain is simpler and faster data structures. The difference is particularly dramatic in the construction phase.

The RRR-structure for compressed bitvector rank [15] divides the bitvectors into small blocks of fixed sizes. Mäkinen and Navarro [11] show that this achieves a similar compression boosting effect without any explicit division of the BWT. This is called *implicit compression boosting*. Their analysis of the effect of fixed blocks inspired our analysis, but the extension from small blocks on bitvectors to larger blocks and larger alphabets is non-trivial.

There are implementations of FM-indexes without any compression boosting [4], with optimal context block boosting [4], and with implicit boosting [2]. Fixed block boosting has practical advantages over all these, which we demonstrate experimentally.

2 Basic Algorithmic Machinery

Let $T = T[0..n-1] = T[0]T[1]\dots T[n-1]$ be a string of n symbols or characters drawn from an alphabet $\Sigma = \{0, 1, .., \sigma - 1\}$. We assume that $T[n-1] = 0$ and 0 does not appear anywhere else in T. In the examples, we use '\$' to denote 0 and letters to denote other symbols.

For any $i \in 0..n-1$, the string $T[i..n-1]T[0..i-1]$ is a *rotation* of T. Let $\mathcal{M}$ be the $n \times n$ matrix whose rows are all the rotations of T in lexicographic order. Let F be the first and L the last column of $\mathcal{M}$, both taken to be strings of length n. The string L is the ***Burrows–Wheeler transform*** of T. An example is given in Fig. 1. Note that F and L are permutations of T.

The FM-family of compressed text self-indexes is based on a procedure called ***backward search***, which finds the range of rows in $\mathcal{M}$ that begin with a given pattern P. This range represents the occurrences of P in T. Fig. 2 shows how backward search is used for implementing the count query. In the algorithm, $C[c]$ is the position of the first occurrence of the symbol c in F, and the function rank_L is defined as

$$\mathrm{rank}_L(c, j) \equiv \left|\{i \mid i < j \text{ and } L[i] = c\}\right|$$

F		L
$	BANAN	A
A	$BANA	N
A	NA$BA	N
A	NANA$	B
B	ANANA	$
N	A$BAN	A
N	ANA$B	A

Fig. 1. BWT matrix $\mathcal{M}$ for text $T =$ `BANANA$`

Algorithm. FM-Count($P[0..m-1]$**)**
1: $b \leftarrow 0$; $e \leftarrow n$
2: **for** $i \leftarrow m-1$ **downto** 0 **do**
3: $\quad c \leftarrow P[i]$
4: $\quad b \leftarrow C[c] + \text{rank}_L(c, b)$
5: $\quad e \leftarrow C[c] + \text{rank}_L(c, e)$
6: $\quad$ **if** $b = e$ **then break** $\quad$ *//The range is empty*
7: **return** $e - b$ $\quad$ *//The range is* $b..e-1$

Fig. 2. Counting pattern occurrences using backward search

The main difference between the members of the FM-family is how they implement the rank_L-function. The best ones use wavelet trees.

A wavelet tree of a string X over an alphabet Σ is a binary tree with leaves labelled by the symbols of Σ. Each node v is associated with the subsequence of X consisting of those symbols that appear in the subtree rooted at v. The associated strings are not stored; instead each internal node v stores a bitvector $B(v)$ that tells for each character in the associated string whether it is in the left or right subtree of v. Fig. 3 shows examples of the two commonly used variants of wavelet trees, the balanced and the Huffman-shaped.

The balanced wavelet tree is easy to implement with low overhead. The total length of the bitvectors is $|X|\lceil\log|\Sigma|\rceil$, which is exactly the length of X in bits using the standard representation. On the other hand, the Huffman-shaped wavelet tree (HWT) is the one that minimizes the total length of the bitvectors, which equals the size of the Huffman compressed string X.

A rank query $\text{rank}_X(c, r)$ over a wavelet tree is evaluated by a traversal from the root to the leaf labelled by c, as shown in Fig. 4. The procedure involves rank queries over the bitvectors stored on the root-to-leaf path.

There are many data structures for representing bitvectors so that rank queries can be answered efficiently [14,16,2]. They can be divided into two main categories. Uncompressing techniques leave the bitvector intact but use a small (usually sublinear) data structure on top of it. Compressing techniques compress the bitvector as well as prepare it for rank queries.

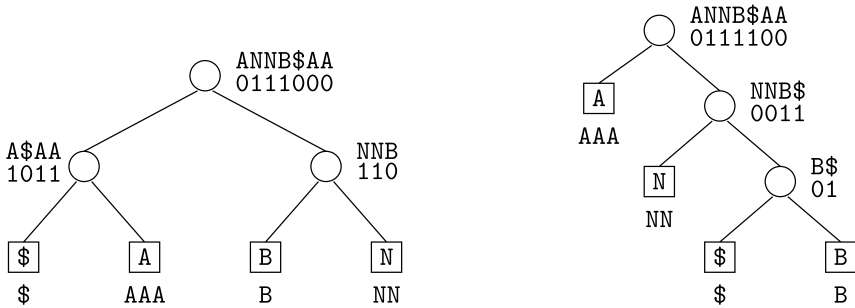


Fig. 3. Balanced (left) and Huffman-shaped (right) wavelet trees

---

Algorithm. WT-Rank(c, r)
1: $v \leftarrow$ root; $q \leftarrow r$
2: **while** v is not a leaf **do**
3: **if** c is in the left subtree of v **then**
4: $q \leftarrow q - rank_{B(v)}(1, q)$
5: $v \leftarrow$ leftchild(v)
6: **else**
7: $q \leftarrow rank_{B(v)}(1, q)$
8: $v \leftarrow$ rightchild(v)
9: **return** q

Fig. 4. Rank operation using a wavelet tree

3 Compression Boosting

Recall that T is a string of length n over an alphabet Σ of size σ. For each $c \in \Sigma$, let n_c denote the number of occurrences of c in T. The zero-order empirical entropy [12] of T is

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c} = \log n - \frac{1}{n} \sum_{c \in \Sigma} n_c \log n_c. \tag{1}$$

Let n_w be the number of occurrences of a string w in T, and let $T|w$ be the subsequence of T consisting of those characters that appear in the *(right) context* w, i.e., that are immediately followed by w. Here T is taken to be a cyclic string, so that each character has a context of every length. The kth order empirical entropy is

$$H_k(T) = \sum_{w \in \Sigma^k} \frac{n_w}{n} H_0(T|w) \ .$$

The value $nH_k(T)$ represents a lower bound on the number of bits needed to encode T by any compressor that considers a context of size at most k when encoding a symbol. Note that $H_{k+1}(T) \leq H_k(T)$ for all k.

A remarkable property of L, the BWT of T, is that $T|w$ is a contiguous substring of L for any w; we call the substring the w-context block of L. For example, if $T = \texttt{BANANA\$}$, then $T|\texttt{A} = \texttt{NNB} = L[1..3]$ (see Fig. 1). Thus we get the following result.

Lemma 1 ([12]). *For any $k \geq 0$, there exists a partitioning of $L_1 L_2 \cdots L_\ell = L$ of the BWT L of T into $\ell \leq \sigma^k$ blocks so that*

$$\sum_{i=1}^{\ell} |L_i| H_0(L_i) = nH_k(T) \ .$$

In other words, by compressing each BWT block to zero-order entropy level, we obtain k^{th} order entropy compression for the whole text. This is called *compression boosting* [3].

The space requirement of an FM-index is usually dominated by the wavelet tree bitvectors. The total length of the bitvectors in the balanced wavelet tree of L is $n\lceil \log \sigma \rceil$. Using a Huffman-shaped wavelet tree reduces this down to at most $n(H_0(T) + 1)$. An alternative way to achieve zero-order compression is to use compressed bitvector rank indexes. For example, using a rank index of Raman, Raman and Rao (RRR) [15], the total size of the rank indexes (without HWT or boosting) is $nH_0(T) + o(n) \log \sigma$.

Compression boosting improves the $H_0(T)$ factor to $H_k(T)$ [6]: Divide the BWT into context blocks using context of length k and implement a separate wavelet tree for each block. There is an additional space overhead of $\mathcal{O}(\sigma \log n)$ bits per block from having many blocks and wavelet trees instead of just one. The total overhead is $o(n)$ for $k \leq ((1-\epsilon)\log_\sigma n) - 1$ and any constant $\epsilon > 0$.

It may not be optimal to use the same context length in all parts of L. Ferragina et al. [3] show how to find an optimal partitioning with varying context length in linear time. The resulting compression is at least as good as with *any* fixed k.

Mäkinen and Navarro [11] show that the boosting effect is achieved with the RRR bitvector rank index without any explicit context partitioning. This is called *implicit compression boosting*. First, they observe that instead of partitioning the BWT, we could partion the bitvectors and obtain the same boosting effect. Second, the RRR technique partitions the bitvectors into blocks of size $b = (\log n)/2$ and compresses each independently. The RRR partitioning is not optimal, but Mäkinen and Navarro show that the overhead due to the inoptimality is at most $2\sigma\ell b \leq \sigma^{k+1} \log n = o(n)$ under the assumptions mentioned above.

Theorem 2 ([6,11]). *The FM-index either with explicit boosting and optimal partitioning [6] or with implicit boosting [11] can be implemented in $nH_k(T) + o(n)\log\sigma$ bits of space for any $k \leq ((1-\epsilon)\log_\sigma n) - 1$ and any constant $\epsilon > 0$.*

4 Fixed Block Compression Boosting

In this section, we show that the compression boosting effect can also be achieved with a partitioning into blocks of fixed sizes without any regard to symbol context.

Let $H(x, y) = |B|H_0(B)$, where B is a bitvector containing x 0's and y 1's. Let $|X|_c$ denote the number of occurrences of a symbol c in a string X. The following lemma shows what can happen to the total zero-order entropy when two strings are concatenated.

Lemma 3. *For any two strings X and Y over an alphabet Σ,*

$$\begin{aligned}0 &\le |XY|H_0(XY) - |X|H_0(X) - |Y|H_0(Y)\\ &= H(|X|, |Y|) - \sum_{c\in\Sigma} H(|X|_c, |Y|_c) \le H(|X|, |Y|) \le |XY| \ .\end{aligned}$$

Proof. The last two inequalities are trivial and the first is a standard application of Gibb's inequality. We will prove the equality part. For brevity, we write $x = |X|$, $y = |Y|$, $x_c = |X|_c$ and $y_c = |Y|_c$. Using (1), we can write the left-hand side terms as follows

$$\begin{aligned}(x + y)H_0(XY) &= (x + y)\log(x + y) - \sum_{c\in\Sigma}(x_c + y_c)\log(x_c + y_c)\\ xH_0(X) &= x\log x - \sum_{c\in\Sigma} x_c \log x_c\\ yH_0(Y) &= y\log y - \sum_{c\in\Sigma} y_c \log y_c\end{aligned}$$

and the right-hand side terms as follows

$$\begin{aligned}H(x, y) &= (x + y)\log(x + y) - x\log x - y\log y\\ H(x_c, y_c) &= (x_c + y_c)\log(x_c + y_c) - x_c \log x_c - y_c \log y_c\end{aligned}$$

From this it is easy to see that the terms on both sides match. □

In other words, the concatenation cannot reduce the total entropy, and the entropy can increase by at most one bit per character. Furthermore, the maximum increase happens only if the two strings have the same length and no common symbols.

Using the above lemma we can bound the increase in entropy when we switch from a context block partitioning to a fixed block partitioning.

Lemma 4. *Let $X_1X_2\cdots X_\ell = X$ be a string partitioned arbitrarily into ℓ blocks. Let $X_1^bX_2^b\cdots X_m^b = X$ be a partition of X into blocks of size at most b. Then*

$$\sum_{i=1}^{m} |X_i^b|H_0(X_i^b) \le \sum_{i=1}^{\ell} |X_i|H_0(X_i) + (\ell - 1)b \ .$$

Proof. Consider a process, where we start with the first partitioning, add the split points of the second partitioning, and then remove the split points of the first partitioning (that are not split points in the second). By Lemma 3, adding split points cannot increase the total entropy, and removing each split point can increase the entropy by at most b bits. □

If we assume the same number of blocks in the two partitionings, the very worst case increase in the entropy is $n - b$ bits.

This increase in entropy can be reduced by reducing the block size in the fixed block partitioning (thus increasing the number of blocks). In particular, if we set the block size to $b = \sigma(\log n)^2$, we obtain the following result.

Theorem 5. *The FM-index with explicit boosting and blocks of fixed sizes can be implemented in* $H_k(T) + o(n)\log\sigma$ *bits of space for any* $k \leq ((1-\epsilon)\log_\sigma n) - 1$ *and any constant* $\epsilon > 0$.

Proof. Using context block boosting with fixed context length k and RRR to compress the bitvectors, the size of the FM-index is $nH_k(T) + o(n)\log\sigma$ bits. When we switch from context blocks to fixed blocks, we must add two types of overhead. First, by Lemma 4, the total entropy increases by at most $\sigma^k b = \sigma^{k+1}(\log n)^2 = n^{1-\epsilon}(\log n)^2 = o(n)$ bits. Second, the space needed for everything else but the bitvector rank indexes is $\mathcal{O}(\sigma \log n)$ bits per block. In total, this is $\mathcal{O}(n/\log n) = o(n)$ bits. Thus the total increase in the size of the FM index is $o(n)$ bits.

Thus, we have the same theoretical result as with context block boosting or implicit boosting.

The advantages of fixed block boosting compared to context block boosting are:

- To compute $\mathrm{rank}_L(c, r)$, we have to find the block containing the position r. With fixed blocks this is simpler and faster than with varying size context blocks.
- Computing the optimal partitioning is complicated and expensive in practice. With fixed blocks, construction is much simpler and faster.

Explicit boosting (with either context blocks or fixed blocks) enables faster queries than implicit boosting for the following reasons:

- Compressed bitvector rank indexes are slower than uncompressed ones by a significant constant factor. Explicit boosting can achieve higher order compression with Huffman-shaped wavelet trees allowing the use of the faster uncompressed rank indexes.
- With implicit boosting, i.e., with a single wavelet tree for the whole BWT, the average count query time for a pattern P is $\Theta(|P|\log\sigma)$ with a balanced wavelet tree and $\Theta(|P|H_0(T))$ with a HWT. With explicit boosting and HWTs, the average query time is reduced down to $\mathcal{O}(|P|H_k(T))$.

Table 1. Data sets used for empirical tests. For each type of data (DNA, XML, ENGLISH, SOURCE) a 100Mb file was used.

Data set name	σ	H_0	Mean Longest Common Prefix
XML	97	5.23	44
DNA	16	1.98	31
ENGLISH	239	4.53	2,221
SOURCE	230	5.54	168

5 Experimental Results

To assess practical performance we used the files listed in Table 1[2]. All tests were conducted on a 3.0 GHz Intel Xeon CPU with 4Gb main memory and 1024K L2 Cache. The machine had no other significant CPU tasks running. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 4.1.1) executed with the -O3 option. The times given were recorded with the C `getrusage` function. The memory requirements are sums of the sizes of all data structures as reported by the `sizeof` function.

We measured the following FM-Index variants:

- SSA [10] simply stores a single HWT for the whole L, consuming $nH_0 + o(n \log \sigma)$ bits of space. This is the fastest index according to experiments in both [2] and [4].
- SSA+RRR is the implicit compression boosting approach of Mäkinen and Navarro [11]. As with SSA it builds a single HWT of L, however the bitvectors of the wavelet tree are now stored in a RRR compressed rank data structure. This method was first implemented by Claude and Navarro [2].
- AFFMI [6] uses optimal context block boosting with a separate HWT for each block. The implementation we use is from [4].
- Fixed Block and Fixed Block+RRR are implementations of the new fixed block boosting technique that use, respectively, plain and RRR preprocessed HWTs to represent blocks.

Figure 5 shows the trade-off between index size and pattern counting time. Following the methodology of [2,4] we report query times averaged over a large number of random patterns of length 20, extracted from the underlying text. In the figure, the multiple points for the Fixed Block indexes correspond to different blocksizes; and for SSA+RRR a tradeoff is given by the sample rate in the RRR implementation.

With the compressible texts (XML, SOURCE and ENGLISH) the fixed block indexes dominate the others in both space and time. On DNA, which is not very compressible, fixed block indexes are still small and fast, but the ranks stored at block boundaries are no longer paid for by compression and the SSA+RRR, which does not need to store ranks at block boundaries, is the smallest index. The small alphabet of DNA means the single HWT of the SSA is shallow, making it fast.

[2] Available from `http://pizzachili.dcc.uchile.cl/`

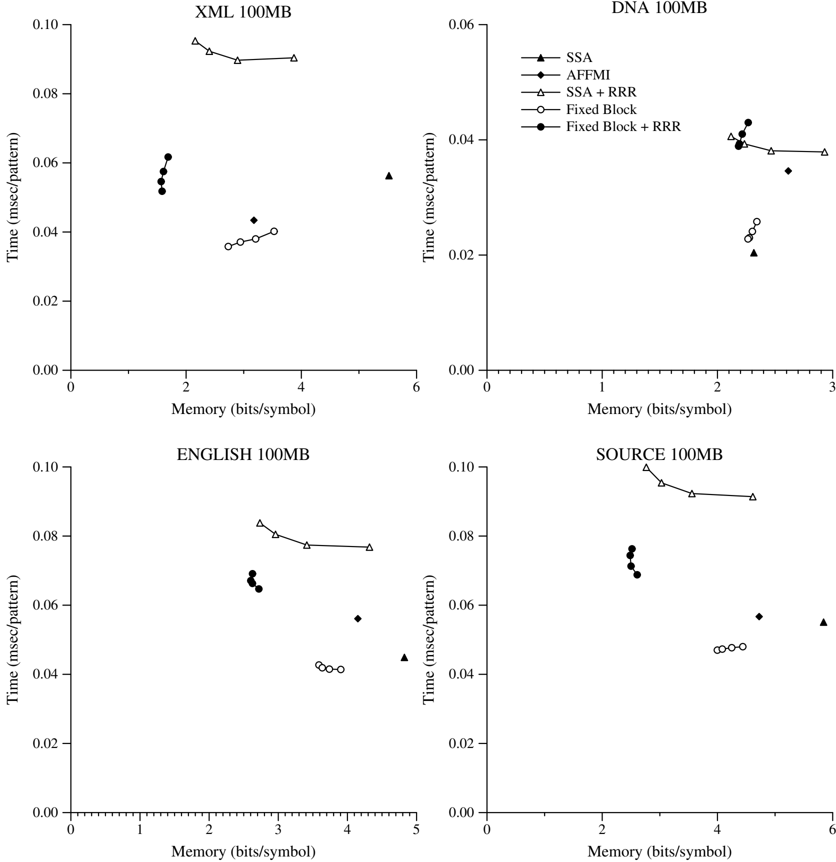


Fig. 5. Time-Space tradeoff for various self-indexes. Memory (abscissa) is the index size in bits per input symbol. Time (ordinate) is the average number of milliseconds taken to count pattern occurrences (averaged over 10^6 patterns of length 20).

The AFFMI, despite using optimal partitioning, is larger and slower than the fixed block indexes. AFFMI stores a bitvector marking the partitioning and issues a rank query on this bitvector to determine the appropriate wavelet tree to use at each step in the backward search process. This adds a significant time and space overhead which the fixed block approach avoids entirely.

6 Concluding Remarks

The indexes we have presented based on fixed block compression boosting are the most practical self-indexes to date, but we believe there is yet more room for improvement. Our current focus is on improving the RRR data structure to better exploit the structure of wavelet tree bitvectors produced by the BWT.

We are also exploring an improved implementation of Huffman-shaped wavelet trees which use substantially less space, enabling smaller blocks and thus better compression.

A virtue of fixed block compression boosting our experiments have not touched on is construction, which is easier with fixed blocks. The final phase of indexing, where the BWT is turned into an FM-index, now requires only $nH_k + o(n)\log\sigma + b\log\sigma$ bits of space, instead of the (at least) $n\log\sigma + o(n)\log\sigma$ bits required by variants to date. If the final index does not have to reside in memory then at most $2b\log\sigma$ bits are needed for construction of the index from the BWT. Construction time remains linear and is fast in practice as the BWT is scanned only once, from left to right. For example, constructing a fixed block index for the XML file takes just 12 seconds, while to build an index with optimal compression boosting requires 273 seconds. Ease of construction is also important when the aim is full inversion of the BWT in a general purpose file compressor [9].

References

1. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California (1994)
2. Claude, F., Navarro, G.: Practical rank/Select queries over arbitrary sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
3. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM 52, 688–713 (2005)
4. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM Journal of Experimental Algorithmics 13, 1.12–1.31 (2009)
5. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52, 552–581 (2005)
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3, Article 20 (2007)
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850. SIAM, Philadelphia (2003)
8. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: experiments with compressing suffix arrays and applications. In: Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 636–645. SIAM, Philadelphia (2004)
9. Kärkkäinen, J., Puglisi, S.J.: Medium-space algorithms for inverse bwt. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6346, pp. 451–462. Springer, Heidelberg (2010)
10. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12, 40–66 (2005)
11. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
12. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48, 407–430 (2001)

13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39 (2007)
14. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia (2007)
15. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3 (2007)
16. Vigna, S.: Broadword implementation of rank/select queries. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 154–168. Springer, Heidelberg (2008)

Space Efficient Wavelet Tree Construction*

Francisco Claude[1], Patrick K. Nicholson[1], and Diego Seco[2]

[1] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[2] University of A Coruña, A Coruña, Spain
{fclaude,p3nichol}@cs.uwaterloo.ca, dseco@udc.es

Abstract. Wavelet trees are one of the main building blocks in many space efficient data structures. In this paper, we present new algorithms for constructing wavelet trees, based on in-place sorting, that use virtually no extra space. Furthermore, we implement and confirm that these algorithms are practical by comparing them to a known construction algorithm. This represents a step forward for practical space-efficient data structures, by allowing their construction on more massive data sets.

1 Introduction

Succinct data structures supporting rank, select and access operations represent the core of most space efficient data structures [24,21,4]. For example, structures supporting rank and select over binary sequences allow for space efficient representation of trees [6,7,23,12,2].

One of the most elegant generalizations of such structures for binary rank, select and access is the *wavelet tree* [17]. Wavelet trees have not only proven to be a crisp theoretical solution, but also perform well in practice [13,8]. As such, they are used by many practical compressed text indexes, such as the SSA [24], LZ77-Index [20] and SLP-Index [9,10]. Another fact that makes wavelet trees interesting as a structure is that they support richer queries than initially expected. For example, they have been used for representing binary relations [4,11], discrete set of points [5] and permutations [3], among others. In all of these domains, wavelet trees support a rich set of operations efficiently.

There has been a great deal of study on the performance of different variants of wavelet trees [8], considering the shape and internal representation of the bitmaps [18,7,25]. However, not much effort has been put into space efficient construction algorithms for wavelet trees.

We present several new algorithms for constructing wavelet trees, using virtually no extra space. This is a significant improvement over the naïve construction algorithm, as well as a previously known technique [22]. We discuss our results in detail at the end of this section, but first we elaborate on the problem.

Given a sequence A of length n, denoted $A[0..n-1]$, drawn from an alphabet Σ of size σ. A wavelet tree is a data structure that supports the following operations: (1) $\texttt{access}(A, i)$: retrieve the symbol at position i in A. (2) $\texttt{rank}_a(A, i)$:

* This work was supported in part by the David R. Cheriton scholarships program (first author) and an NSERC of Canada PGS-D Scholarship (second author).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 185–196, 2011.
© Springer-Verlag Berlin Heidelberg 2011

count how many times the symbol a appears in $A[0..i]$. (3) $\mathtt{select}_a(A, j)$: retrieve the position of the j-th occurrence of a in A.

Supporting these operations in constant time, using $nH_0(A) + o(n)$ bits of space[1], was first solved for binary alphabets (i.e., $\Sigma = \{0, 1\}$) by Raman et al. [25]. Wavelet trees extend this result to arbitrary alphabets in the following way. Every node in a wavelet tree has two children, and we identify them as left and right. Each node represents a range $R \subseteq [1, \sigma]$ of the alphabet Σ, and its left child represents a subset $R_\ell \subset R$ and the right child a subset $R_r = R \setminus R_\ell$. Every node virtually represents a subsequence A' of A composed of elements whose value is in R. This subsequence is stored as a bitmap B of length $|A'|$, and, for each position i in the bitmap, a 0 bit means that position i belongs to R_ℓ and a 1 bit means that it belongs to R_r.

One can access a position in A by following the path from the root to a leaf guided by the bit representing the position at each level. When moving to the left child the position i is mapped to $\mathtt{rank}_0(B, i)$ and when moving to the right child the position is mapped to $\mathtt{rank}_1(B, i)$. By similar observations, it is an easy exercise to show that a wavelet tree can support rank, select and access operations in $O(\lg \sigma)$ time[2], assuming that rank and select are supported in constant time on the bitmaps in each level. Furthermore, the wavelet tree uses $n \lg \sigma + o(n \lg \sigma)$ bits because the bitmaps representing the nodes use $n \lg \sigma$ bits in total and the little-oh term covers the cost of the rank and select structures.

Wavelet trees were later extended to *generalized wavelet trees*, where the fan out of each node is increased from 2 to $O(\text{polylog}(n))$, increasing the speed of the three operations to $O(\lg \sigma / \lg \lg n)$ time [14]. In favour of clarity, we focus on binary wavelet trees, though our results also extend to the generalized version.

A simple construction algorithm works in the following way. First, we build the root of the wavelet tree. To do this, we partition the alphabet Σ into Σ_ℓ and Σ_r according to some balancing rule, and then create a bitmap of length n that stores a 0 in position i if $A[i] \in \Sigma_\ell$ and a 1 in position i otherwise. Then, we generate a copy of the subsequence of elements whose bit at position i is a 0 and recurse on this subsequence to build the left subtree. Finally, we recurse on the 1 bits to construct the right subtree.

It is easy to see that this method is inefficient and requires $O(n \lg^2 \sigma)$ bits, since we copy the array A at each of the $\lg \sigma$ levels. This can be improved by realizing that, at each level in the recursion, we can reuse the space from the previous level and avoid recopying the sequence. The only exception to this is the root, since our application might require us to leave A unmodified after the construction process. Thus, we can construct the wavelet tree in $O(n \lg \sigma)$ extra bits, since we must copy A. An example of such an application is a library for succinct data structures, such as LIBCDS[3], where the user might want to further process the sequence used to build the wavelet tree.

[1] $H_0(A)$ denotes the 0th-order empirical entropy of the string A.
[2] Throughout this paper we denote $\lceil \log_2 n \rceil$ as $\lg n$.
[3] http://libcds.recoded.cl

Our Results: In this paper we present several new algorithms for constructing wavelet trees space efficiently. All of our results hold in the word-RAM model with word size $\Omega(\lg n)$. Throughout this section, we denote the input array as A and the wavelet tree as T. All of our results are for uncompressed wavelet trees, that is, the case where the bitmaps of the wavelet tree occupy $n \lg \sigma$ bits. Thus T occupies $n \lg \sigma + S(n \lg \sigma)$ bits, where $S(m)$ denotes the extra space required by auxiliary rank and select structures on a bitmap of length m. Extending our results to compressed wavelet trees, e.g., Huffman shaped wavelet trees, is left as an open problem. Before discussing the results, we need the following definitions.

We refer to a construction algorithm as *non-destructive* if A is unmodified after T has been constructed. If A is modified, rendering it unusable, we say the construction algorithm is *destructive*.

Since there are many choices of rank and select data structures for bitmaps, we will attempt to state our results as generally as possible. We do, however, assume that the auxiliary structures used for bitmaps support rank and select in constant time. Let $C(m)$ denote the construction time for auxiliary rank and select structures on a bitmap of length m, and $E(m)$ denote the extra bits required to construct these structures. We note that in many existing algorithms, $E(n) = O(\lg n)$ extra bits. Given a bitmap $B[0..n-1]$, suppose we construct auxiliary rank and select structures in time $C(p)$ on a prefix of B, $B[0..p]$ where $0 \leq p < n$, such that we can answer rank and select queries on $B[0..p]$. If we can extend our rank and select structures to support queries on $B[0..q]$ for $p < q < n$ in $C(q-p)$ time, we say these rank and select structures can be constructed *incrementally*.

In Section 2 we present a non-destructive algorithm for constructing the wavelet tree T in $O(n \lg \sigma + C(n \lg \sigma))$ time, using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space occupied by A and T. This section serves as a warm-up, introducing many of the concepts used later in the paper.

In Section 3 we present a destructive algorithm for constructing the wavelet tree T in $O(n \lg n \lg^2 \sigma + C(n \lg \sigma))$ time using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for T. In other words, this algorithm replaces A with the bitmaps for each node in T. We also present a more practical algorithm that runs in $O(n \lg \sigma + C(n \lg \sigma))$ time and uses $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for T. In all of the previous results, we show how to replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if the rank and select structures for T can be constructed incrementally.

Finally, in Section 4 we provide experimental results for the algorithms discussed in Section 3. These results show that our algorithms are practical.

2 Encoding Scheme

In this section, we show how to reorder the elements of an array $A[0..n-1]$ according to the bitmap $B[0..n-1]$ representing the root of the wavelet tree. This allows us to construct T without copying the subsequences of A into separate arrays at each level. We refer to this process as *partitioning*, and it is the

bottleneck in any space efficient wavelet tree construction algorithm. After describing partitioning, we show how to reverse a partitioning step. That is, given the subsequence of elements from the left subtree, A_ℓ, the elements from the right subtree, A_r, and the bitmap representing the root of the wavelet tree, B, we show how to rebuild A. We refer to this process as *merging*. In other words, denote string concatenation by $\cdot$ and let $A' = A_\ell \cdot A_r$. Partitioning is the process of constructing A' from A and B, and merging is the process of reconstructing A from A' and B.

2.1 Partitioning

We describe the method implemented in LIBCDS that shows a simple way of partitioning the array A given that we can support constant time rank and select queries on the bitmap B [22]. Let n_ℓ denote $\mathtt{rank}_0(B, n-1)$. It is easy to see that the bitmap B defines a permutation π on the elements of array A as follows:

$$\pi(i) = \begin{cases} \mathtt{rank}_0(B, i) - 1 & \text{if } B[i] = 0, \\ \mathtt{rank}_1(B, i) - 1 + n_l & \text{otherwise.} \end{cases} \quad (1)$$

One way of partitioning A is to create an *auxiliary bitmap Aux* of length n, where initially all of the bits are set to 0. We then do a scan of the array A and, for each position $p = 0..n-1$, if $Aux[p] = 0$, we move the element at position p to its corresponding place, position $q = \pi(p)$, and set $Aux[p] = 1$. We repeat this process with the element that was at position q in A until returning to a position q' where $Aux[q'] = 1$ [22]. It is not hard to see that $q' = p$: following standard terminology [15], we call position p a *cycle leader* of π, since it has the smallest index in the *cycle* of element swaps. Furthermore, we say that the elements in a cycle are *rotated* according to π. Thus, the auxiliary array is used to identify cycle leaders.

The procedure just described requires $n + O(\lg n)$ extra bits to identify cycle leaders and perform the partitioning in $O(n)$ time, assuming we can support rank operations on B in constant time. The $O(\lg n)$ term comes from the constant number of pointers needed to scan the array and rotate the elements.

Let $\pi^k(i) = \pi(\pi^{k-1}(i))$, for $k > 1$ and $\pi^1(i) = \pi(i)$. If $\pi(i) = i$ then we say position i is a *self-cycle*. Let A' denote the array A after it has been partitioned. In order to improve the space requirements, we prove the following property of the permutation π:

Lemma 1. *If $\pi(i) \neq i$ (i.e., position i is not a self-cycle), then $\pi^k(i) \geq n_\ell$ for some $k \geq 1$. Similarly, if $j \geq n_\ell$ and $\pi(j) \neq j$, then $\pi^k(j) < n_\ell$ for some $k \geq 1$.*

Proof. The relative ordering of the elements that are symbols in Σ_ℓ in $A'[0..n_\ell - 1]$ is the same as the relative ordering of these elements in $A[0..n-1]$. Thus, if a rotation begins at an element in Σ_ℓ in position i, it will be moved to a position $0 \leq j < i$. Since the rotation will end at position i, at some point we must

encounter an element in Σ_r. By the definition of π, this element will be moved to a position $j' \geq n_\ell$. The second part of the lemma follows by symmetry. □

Based on the previous lemma, we make the following observation:

Observation 1. *Rotating only the cycle leaders in positions where $B[i] = 0$ is sufficient to complete the partitioning of A into A', since every cycle that is not a self-cycle involves elements from both Σ_ℓ and Σ_r.*

We now show how to perform the partitioning *without* access to Aux. We continue assuming that $|\Sigma_\ell| \leq |\Sigma_r|$. If this condition does not hold, we can apply Observation 1 symmetrically, considering only positions where $B[i] = 1$.

The idea that allows us to discard Aux is to encode it inside A as we perform the partitioning. We do this by defining an invertible function $f : \Sigma_\ell \rightarrow \Sigma_r$. This function exists since $|\Sigma_\ell| \leq |\Sigma_r|$. We run exactly the same partitioning algorithm described in the beginning of this section, except that, during a rotation, every time we move a symbol $s \in \Sigma_\ell$ at position i to position $\pi(i)$, we write $f(s)$ in position $\pi(i)$ instead. Since we do not have to rotate cycle leaders from Σ_r by Observation 1, this encoding step is functionally equivalent to having access to Aux. Every time we encounter an element in Σ_ℓ, that element would have had a 0 in its corresponding position in Aux, and we can ignore elements that either would have a 1 in Aux or were originally in Σ_r.

After finishing this process, we need one extra pass to decode the values that are supposed to be in Σ_ℓ. We do this by traversing A' and replacing position i by $f^{-1}(A[i])$ if $i < n_\ell$. Recall our assumption that T is an uncompressed wavelet tree, i.e., each partitioning step moves the elements with a 0 as their most significant bit to the left child and each element with a 1 as their most significant bit to the right child. Since the ranges of Σ spanned by Σ_ℓ and Σ_r are contiguous and adjacent, computing $f()$ and $f^{-1}()$ boils down to a simple addition and subtraction, respectively.

Lemma 2. *Given an array A over an alphabet Σ and support for constant time rank and select operations on the bitmap B, we can partition A in-place to generate A' in $O(n)$ time using $O(\lg n)$ extra bits of space.*

2.2 Merging

The merging process is just the partitioning process in reverse. We describe this problem in a similar way, using the inverse permutation π^{-1}:

$$\pi^{-1}(i) = \begin{cases} \texttt{select}_0(B, i+1) & \text{if } i < n_l, \\ \texttt{select}_1(B, i+1-n_l) & \text{otherwise.} \end{cases} \tag{2}$$

It is easy to see that Lemma 1 and Observation 1 also hold in the merging case. The only difference is that now elements in Σ_ℓ are rotated to the right and elements from Σ_r are rotated to the left. Thus, there is at least one element in Σ_ℓ and one in Σ_r for each cycle of length greater than 1. These observations allow us to apply the same method as in Lemma 2, obtaining the following lemma.

Lemma 3. *Given the array A' and support for constant time rank and select operations on the bitmap B, we can reconstruct A in-place in $O(n)$ time using $O(\lg n)$ extra bits of space.*

Using a stack of size $O(\lg \sigma)$ pointers to keep track of the node in T that we are currently processing, we can recursively apply *partitioning* to A, to construct T. After T is constructed, we can reverse the process by *merging* to recover A. We can compute B in linear time for each node.

Theorem 2. *There exists a non-destructive algorithm for constructing a wavelet tree T that uses $O(n \lg \sigma + C(n \lg \sigma))$ time, and $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for T and the input array A.*

2.3 Extension to Generalized Wavelet Trees

The encoding scheme for generalized wavelet trees works in a similar way to that of the binary case. We begin by stating the generalized partitioning problem: given an array $A[0, n-1]$ with values in $\Sigma = [0, \sigma - 1]$ and a sequence S with values in $D = [0, k-1]$ that partitions Σ into k disjoint sets $\Sigma_0, \ldots, \Sigma_{k-1}$, we want to generate an array A' where elements of A are stable-sorted by their corresponding value in D. We can describe the re-ordering in A as a permutation: $\pi_k(i) = \texttt{rank}_j(S, i) + (\sum_{v<j} \texttt{rank}_v(S, n-1)) - 1$, where $j = S[i]$.

Lemma 4. *The permutation π_k is strictly increasing for positions containing the same value in D.*

Proof. We can write $\pi_k(i)$ as $\texttt{rank}_j(S, i) + g(j)$, where $j = S[i]$ and $g(j) = \sum_{v<j} \texttt{rank}_v(S, n-1)$, and $\texttt{rank}_j(S, i)$ is strictly increasing in i. □

Lemma 5. *Any cycle $\mathcal{C}$ in π_k such that $|\mathcal{C}| > 1$, contains at least two positions i, j such that $S[i] \neq S[j]$.*

Proof. By contradiction, assume $|\mathcal{C}| > 1$ and all positions $p_i \in \mathcal{C}$ satisfy $S[p_i] = s$ for a fixed s. Let $p = \min \mathcal{C}$. Since all positions in $\mathcal{C}$ point to elements in S whose value is the same, then π_k is strictly increasing in $\mathcal{C}$, thus, there is no p_j such that $\pi_k(p_j) = p$, therefore, $\mathcal{C}$ is not a cycle. □

Now we can present the encoding method. Let $m \in [0, k-1]$ be the index such that $|\Sigma_m| \geq |\Sigma_j|$. We then generate $f_j : \Sigma_j \to \Sigma_m$ such that ${f_j}^{-1}$ exists. Then, the algorithm for partitioning works in the following way: for each cycle leader, we start rotating the elements if the position is not in Σ_m. Every time we rotate an element e in Σ_j, we replace it with $f_j(e)$.

Once we finish the process, we do a final pass through A' fixing the values at position p using ${f_j}^{-1}$, where j is determined by the position, i.e., $j = \min\{r | \sum_{v<r} \texttt{rank}_v(S, n-1) > p\}$.

There is one detail remaining, and this is how to compute $g(j) = \sum_{v<j} \texttt{rank}_v(S, n-1)$. We can do this by pre-computing all possible answers in linear time. This option requires $O(\sigma \lg n)$ bits of extra space. Another option

is to use compressed bitmaps to represent this in $\sigma \lg(n/\sigma) + o(n)$ bits, while supporting queries in constant time [25]. We now state the partitioning theorem for the generalized wavelet tree:

Theorem 3. *We can solve the partitioning problem for the generalized wavelet tree in $O(n\tau)$ time using $\min(\sigma \lg(n/\sigma) + o(n), O(\sigma \lg n))$ bits of extra space, where τ represents the maximum between the time to answer rank and access in a sequence over an alphabet of size k.*

The generalized merging process is similar, and the permutation π_k^{-1} is defined as follows: $\pi_k^{-1}(i) = \texttt{select}_j(S, i+1-g(j))$, where $j = S[i]$ and $g(j) = \sum_{v<j} \texttt{rank}_v(S, n-1)$.

Lemmas 4 and 5 also apply to π_k^{-1}, since select is strictly increasing for positions that contain the same element. This allows us to apply the same encoding technique as before; the only difference is the transformation at the end. Instead of examining the range we are in, we examine the character in S associated with our position.

Theorem 4. *We can solve the merging problem for the generalized wavelet tree in $O(n\tau)$ time using $\min(\sigma \lg(n/\sigma) + o(n), O(\sigma \lg n))$ bits of extra space, where τ represents the maximum between the time to answer access and select in a sequence over an alphabet of size k.*

Regarding the rank, select and access times in Theorems 3 and 4, there are many alternatives [17,14,16], in particular, it is possible to adapt the solution by [14] to compute the function $g()$ in constant time, achieving the following Corollary.

Corollary 1. *We can solve the partitioning and merging problems for the generalized wavelet tree in $O(n)$ time using $\min(n \lg(n/\sigma) + o(n), O(\sigma \lg n))$ extra bits of space, if the branching factor of the generalized wavelet tree is $k = O(\text{polylog}(n))$.*

3 Construction by Permuting Bits

In this section we show how to destructively permute the bits of an input array A, converting them into the bit vectors of a *binary* wavelet tree T.

Let B_v represent the bitmap stored at the root v of T, and v_l and v_r represent the left and right children of v. Define $\mathbb{B}(v) = B_v \cdot \mathbb{B}(v_l) \cdot \mathbb{B}(v_r)$. Thus, $\mathbb{B}(v)$ is the concatenation of the bitmaps stored at the nodes of T, in depth first order from the root, v. We describe an algorithm for computing $\mathbb{B}(v)$ that works by permuting the bits of A in place. For our purposes in this section, we consider A to be a bitmap of length $n \lg \sigma$. Let ϕ be the permutation that maps A to $\mathbb{B}(v)$; we abuse notation and denote this as $\phi(A) = \mathbb{B}(v)$. We show that ϕ is the composition of 2σ permutations: two permutations, χ and ψ, corresponding to each node in T.

3.1 Overview of Permutations

In the next section we describe the two permutations χ and ψ that correspond to the root v of T. Before specifying the technical details of these permutations, we first briefly outline *what* they do to the array A.

The first permutation χ shifts the most significant bit of each character in A to a prefix of the bitmap, preserving relative order. More precisely, $\chi(A)$ consists of an n bit prefix $B_v[0..n-1]$, representing the most significant bits of each element in A (which *is* the bitmap stored at the root of T), followed by n *truncated characters* of length $\lg \sigma - 1$; the i-th truncated character is $A[i]$ without its most significant bit, for $0 \leq i < n$.

Let $A_t[0..n-1]$ represent the n truncated characters. The second permutation, ψ, partitions $A_t[0..n-1]$ according to the bits $B_v[0..n-1]$. Thus, applying ψ is equivalent to the partitioning step in a standard wavelet tree construction algorithm: if $B_v[i]$ is a 0, then $A_t[i]$ is partitioned to the left and, if $B_v[i]$ is a 1, then $A_t[i]$ is partitioned to the right.

After applying χ and ψ to A, $\psi(\chi(A))$ consists of $B_v[0..n-1]$, which are the first n bits of $\mathbb{B}(v)$, followed by the partitioned truncated characters, which can then be recursively converted into the remaining bits of $\mathbb{B}(v)$ in a depth first manner. In the sequel, we rely heavily on the following result of Fich et al.:

Theorem 5 (Theorem 2.2 [15]). *In the worst case, permuting an array of length n, given the permutation and its inverse, can be done in $O(n \lg n)$ time and $O(\lg n)$ additional bits of storage.*

The next two sections define the permutations χ and ψ, and their respective inverses.

3.2 Chopping the Most Significant Bits

Since A is a bitmap of length $n \lg \sigma$, we refer to individual bits in A. Let $A = b_0, \ldots, b_{n \lg \sigma - 1}$, where $b_{j \lg \sigma}, ..., b_{(j+1) \lg \sigma - 1}$ are the bits in $A[j]$ for $0 \leq j < n$, in *decreasing order of significance*[4]. Using this notation, we can now describe $\chi(A, i)$, the i-th bit of $\chi(A)$ in terms of the bits in A, for $0 \leq i < n \lg \sigma$. If $\chi(A, i) = j$, then the i-th bit of $\chi(A)$ is the j-th bit of A, or b_j; in the equations we write j instead of b_j for readability.

$$\chi(A, i) = \begin{cases} \frac{i}{\lg \sigma} & \text{if } \lg \sigma \text{ divides } i, \\ i + n - \left\lfloor \frac{i}{\lg \sigma} \right\rfloor - 1 & \text{otherwise.} \end{cases}$$

Similarly, we can describe the permutation $\chi^{-1}(A, i)$ as follows:

$$\chi^{-1}(A, i) = \begin{cases} i \lg \sigma & \text{if } i < n, \\ i - n + \left\lfloor \frac{i-n}{\lg \sigma - 1} \right\rfloor + 1 & \text{otherwise.} \end{cases}$$

[4] If the characters are stored in increasing order of significance, then we can easily reverse their bits in $O(n \lg \sigma)$ time and $O(\lg n)$ extra bits.

Running Time: Using χ and χ^{-1} we can apply Theorem 5 to A. This allows us to compute $\chi(A)$ in place using $O(n \lg \sigma \lg(n \lg \sigma)) = O(n \lg \sigma \lg n)$ time.

3.3 Partitioning the Truncated Letters

We now describe how to compute $\psi(\chi(A))$ from $\chi(A)$. Note that $\chi(A) = B_v[0..n-1] \cdot A_t[0..n-1]$, and suppose we build a rank and select data structure over B_v. We discuss the space issues associated with this in the next section. The permutations ψ and ψ^{-1} make use of rank and select queries in order to determine how to partition $A_t[0..n-1]$. Not surprisingly, ψ and ψ^{-1} are *almost* identical to the permutations described in Equations 1 and 2, respectively. The only difference is that we need to account for the fact that the truncated characters A_t begin at an offset of n from the beginning of A and consist of $\lg \sigma - 1$ bits. We omit the exact details since they are not difficult, but tedious.

Running Time: As before, using ψ and ψ^{-1} we can apply Theorem 5 to $\chi(A)$. Observe that we can easily swap $\lg \sigma - 1$ bit elements in constant time, assuming the word size is $\Omega(\lg n)$ and $n \geq \sigma$. This allows us to compute $\psi(\chi(A))$ in place using $O(n \lg n)$ time.

3.4 Overall Requirements

Running Time: We must apply χ and ψ appropriately at each node in T in order to compute $\phi(A)$. This means that our overall running time is $T(n, \lg \sigma) = T(n_l, \lg \sigma - 1) + T(n - n_l, \lg \sigma - 1) + O(n \lg n \lg \sigma)$, where $T(n, 1) = O(1)$. Since the height of the tree is bounded in terms of $\lg \sigma$ rather than n, $T(n, \lg \sigma) = O(n \lg n \lg^2 \sigma)$.

Extra Space: As discussed in Section 3.3, at each node v in T we construct auxiliary rank and select structures for the bitmap of length $m \leq n$, associated with v. Let $S(m)$ denote the number of bits required for the auxiliary structures. The auxiliary structures for T require $S(n \lg \sigma)$ extra bits, since $\mathbb{B}(v)$ is a bitmap of length $n \lg \sigma$. Furthermore, we can release the memory used by the auxiliary structures for each $v \in T$ after we have applied the permutations to v. Thus, we can avoid using extra space for the auxiliary structures associated with v, with careful memory management.

In addition to the $O(1)$ extra pointers required by Theorem 5, we need a stack of size $O(\lg \sigma)$ pointers in order to remember our current location within T. However, we can get rid of the stack at the cost of some complexity. Suppose $w_1, ..., w_\sigma$ are the nodes of T in depth-first order. Then by storing only n and the bits representing the path to w_i, we can compute the offset and length of the bitmap representing w_{i+1} in $O(\lg \sigma)$ time using the rank and select structures constructed for $w_1, ..., w_i$. Note that $w_1, ..., w_i$ represent a contiguous prefix of the bitmap $\mathbb{B}(v)$. Thus, if we can construct rank and select structures for $\mathbb{B}(v)$ incrementally, we can discard the stack.

Trade Off: If we have an extra n bits available to us, then we can apply χ and ψ to each node in T in $O(n)$ time per level of T, using a strategy similar to Arroyuelo and Navarro [1]. To apply χ, we copy the most significant bits in A into the auxiliary bitmap, then shift the remaining bits to the end of A, and finally, overwrite the first n bits of A with those stored in the auxiliary bitmap. This requires $O(n)$ time if we make use of word-level parallelism during the shifting stage, or $O(n \lg \sigma)$ time if we shift the bits one at a time. To apply ψ we use the auxiliary bitmap to store the cycle leaders, as described in Section 2. Thus, with the n extra bits, we can compute $\phi(A)$ in $O(n \lg \sigma)$ time, or $O(n \lg^2 \sigma)$ time without word-level parallelism during the shifting stage.

Theorem 6. *Suppose we are given an array A of n symbols drawn from an alphabet of size σ. Let $C(m)$ denote the construction time for the auxiliary rank and select structures on a bitmap of length m, $E(m)$ denote the extra bits required during the construction of these structures, and $S(m)$ denote the total number of bits occupied by these structures. Furthermore, assume these structures support rank and select in constant time. We can permute the $n \lg \sigma$ bits of A, replacing A with the bitmaps of a wavelet tree T occupying $n \lg \sigma + S(n \lg \sigma)$ bits in:*

1. *$O(n \lg n \lg^2 \sigma + C(n \lg \sigma))$ time using $O(\lg \sigma \lg n) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*
2. *$O(n \lg \sigma + C(n \lg \sigma))$ time, and using $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*

In both of the previous results, we can replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if we can construct the rank and select structures for T incrementally.

The same idea can be extended to the generalized setting, though we defer the details to a later version of this paper.

4 Experiments

As a proof of concept we implemented two of the algorithms described in Theorem 6. We refer to the algorithm that uses $O(\lg \sigma \lg n)$ extra bits and runs in $O(n \lg n \lg^2 \sigma)$ time as Permute, and the algorithm that uses $n + O(\lg \sigma \lg n)$ extra bits and runs in $O(n \lg^2 \sigma)$ time (not making use of word-level parallelism) as Permute2. For a base line comparison, we used the space efficient destructive construction algorithm found in Libcds, which is similar to the algorithm described in Section 2. The code for Permute is an adaptation of the code from [15, Figure 5], modified to use a simple heuristic speed-up called Fast-Break [19]. The code for Permute2 is even more straightforward since it uses an auxiliary bitmap.

The machine used for the experiments has an AMD Athlon™ 64 X2 Dual Core Processor 5600+, core speed 2900MHz, L1 Cache size 256KB and L2 Cache size 1024KB. It has 4GB of 800MHz main memory. The operating system installed is GNU/Linux (Ubuntu 10.04 LTS) and the code was compiled using GNU/g++ version 4.4.3, with optimization flags -O9.

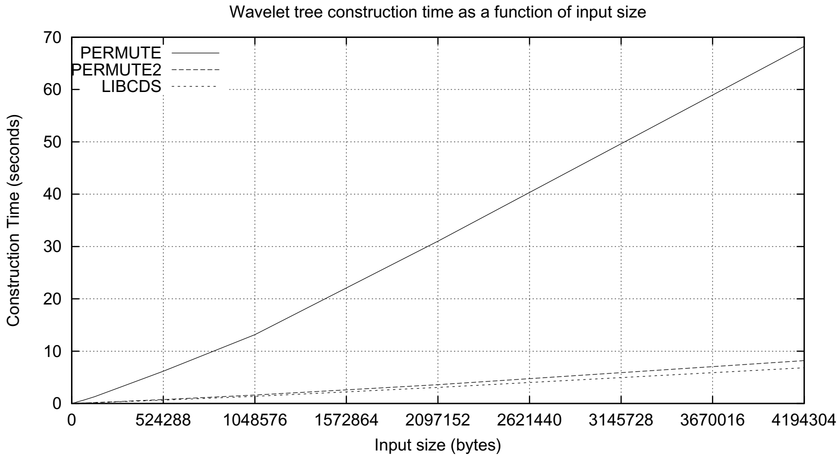


Fig. 1. A comparison of the space efficient construction algorithms PERMUTE and PERMUTE2 with the standard LIBCDS construction algorithm. In this experiment $\lg \sigma = 8$.

We ran experiments for the case when $\lg \sigma = 8$, i.e., the array A consists of 8-bit characters. For $n = 8, 16, 32, ..., 4194304$, we generated 10 strings of length n, where each string consists of n 8-bit integers drawn *uniformly at random*. For each n we kept track of the best, worst and average running time of the three construction algorithms. Since the best and worst times were very close to the average, we report only on the average time. A graph comparing the average construction times of the algorithms can be found in Figure 1. We obtained similar results when testing on prefixes of English and DNA text from `http://pizzachili.dcc.uchile.cl`.

As expected, PERMUTE is slower than LIBCDS for all tested values of n, by a factor that increases with n. For all values tested this factor was less than 11. Although this is a significant slow down, this experiment demonstrates that the space efficient algorithm we describe is implementable and that its performance is not impractical.

On the other hand, PERMUTE2 ran only slightly slower than LIBCDS, which has complexity $O(n \lg \sigma)$. This suggests that PERMUTE2 is highly competitive as a construction algorithm, due to low constant factors. Furthermore, since it only uses $n + O(\lg n \lg \sigma)$ extra bits on top of the space required for the wavelet tree, it provides a nice compromise between the slower PERMUTE algorithm and the construction algorithm currently used by LIBCDS.

References

1. Arroyuelo, D., Navarro, G.: Space-efficient construction of lempel-ziv compressed text indexes. Information and Computation 209(7), 1070–1102 (2011)
2. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: Proc. ALENEX, pp. 84–97 (2010)

3. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. STACS, pp. 111–122 (2009)
4. Barbay, J., Claude, F., Navarro, G.: Compact rich-functional binary relation representations. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 170–183. Springer, Heidelberg (2010)
5. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
6. Clark, D.: Compact Pat Trees. Ph.D. thesis, University of Waterloo (1996)
7. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. SODA, pp. 383–391 (1996)
8. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008)
9. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 235–246. Springer, Heidelberg (2009)
10. Claude, F., Fariña, A., Martínez-Prieto, M., Navarro, G.: Compressed q-gram indexing for highly repetitive biological sequences. In: Proc. BIBE, pp. 86–91 (2010)
11. Farzan, A., Gagie, T., Navarro, G.: Entropy-bounded representation of point grids. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 327–338. Springer, Heidelberg (2010)
12. Farzan, A.: Succinct Representation of Trees and Graphs. Ph.D. thesis, University of Waterloo (2009)
13. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM JEA 13, 30 pages (2009)
14. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. on Alg. 3(2), article 20(2007)
15. Fich, F., Munro, J.I., Poblete, P.: Permuting in place. SIAM J. on Comp. 24, 266 (1995)
16. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. SODA, pp. 368–373 (2006)
17. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850 (2003)
18. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. FOCS, pp. 549–554 (1989)
19. Keller, J.: A heuristic to accelerate in-situ permutation algorithms. Inf. Proc. Lett. 81(3), 119–125 (2002)
20. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 41–54. Springer, Heidelberg (2011)
21. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theo. Comp. Sci. 387, 332–347 (2007)
22. Mäkinen, V., Välimäki, N.: Personal communication
23. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
24. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
25. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. SODA, pp. 233–242 (2002)

Computing the Longest Common Prefix Array Based on the Burrows-Wheeler Transform

Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
{Timo.Beller,Simon.Gog,Enno.Ohlebusch,Thomas.Schnattinger}@uni-ulm.de

Abstract. Many sequence analysis tasks can be accomplished with a suffix array, and several of them additionally need the longest common prefix array. In large scale applications, suffix arrays are being replaced with full-text indexes that are based on the Burrows-Wheeler transform. In this paper, we present the first algorithm that computes the longest common prefix array directly on the wavelet tree of the Burrows-Wheeler transformed string. It runs in linear time and a practical implementation requires approximately 2.2 bytes per character.

1 Introduction

A suffix tree for a string S of length n is a compact trie storing all the suffixes of S (so it is a full-text index). It is an extremely important data structure with applications in string matching, bioinformatics, and document retrieval, to mention only a few examples; see e.g. [8]. Suffix arrays can replace suffix trees and they use less memory than those. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S. To be precise, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \ldots < S_{\mathsf{SA}[n]}$, where S_i denotes the i-th suffix $S[i..n]$ of S; see Fig. 1 for an example. In the last decade, much effort has gone into the development of efficient suffix array construction algorithms (SACAs); see [20] for a survey.

However, the meteoric increase of DNA sequence information produced by next-generation sequencers demands new computer science approaches to data management because the data must be stored, analyzed, and mined. To analyze the massive quantities of data, established data structures like the suffix array (and the suffix tree) are being replaced by less space consuming data structures like the wavelet tree of the Burrows-Wheeler transformed sequence. It goes as follows: the sequence is subjected to the Burrows-Wheeler transform (BWT) [2], the Burrows-Wheeler transformed sequence is stored in a wavelet tree (or, more generally, in an FM-index [4]), and the wavelet tree [7] supports backward search on the original sequence. Let us recall the backward search technique in more detail. Let Σ be an ordered alphabet of size σ whose smallest element is the so-called sentinel character \$. In the following, S is a string (sequence) of length n over Σ having the sentinel character at the end (and nowhere else). The BWT transforms the string S into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 197–208, 2011.
© Springer-Verlag Berlin Heidelberg 2011

i	SA	LCP	BWT	$S_{\mathsf{SA}[i]}$
1	19	-1	n	$
2	3	0	l	_anele_lepanelen$
3	9	1	e	_lepanelen$
4	4	0	_	anele_lepanelen$
5	13	5	p	anelen$
6	8	0	l	e_lepanelen$
7	1	1	$	el_anele_lepanelen$
8	6	2	n	ele_lepanelen$
9	15	3	n	elen$
10	17	1	l	en$
11	11	1	l	epanelen$
12	2	0	e	l_anele_lepanelen$
13	7	1	e	le_lepanelen$
14	16	2	e	len$
15	10	2	_	lepanelen$
16	18	0	e	n$
17	5	1	a	nele_lepanelen$
18	14	4	a	nelen$
19	12	0	e	panelen$
20		-1		

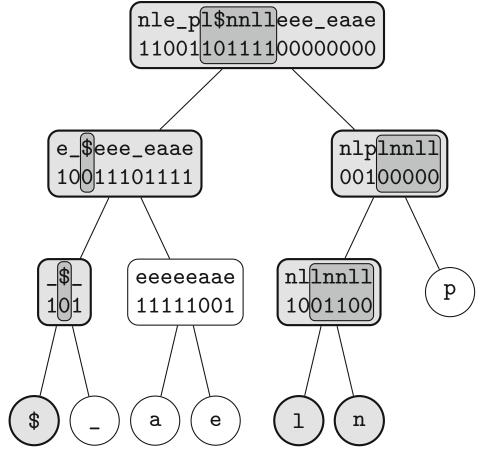


Fig. 1. Left: Suffix array, LCP-array, and Burrows-Wheeler-transformed string BWT of string S = `el_anele_lepanelen$`. Right: Conceptual illustration of the wavelet tree of the string BWT = `nle_pl$nnlleee_eaae`. Only the bit vectors are stored; the corresponding strings are shown for clarity. The shaded regions will be explained later.

for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. Ferragina and Manzini [4] showed that it is possible to search a pattern backwards, character-by-character, in the suffix array SA of string S, without storing SA. Let $c \in \Sigma$ and ω be a substring of S. Given the ω-interval $[i..j]$ in the suffix array SA of S (i.e., ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leq k \leq j$, but ω is not a prefix of any other suffix of S), $backwardSearch(c, [i..j])$ returns the $c\omega$-interval $[C[c] + Occ(c, i-1) + 1 \; .. \; C[c] + Occ(c, j)]$, where $C[c]$ is the overall number of occurrences of characters in S which are strictly smaller than c and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$.

The approach described above was used for example in the software-tools Bowtie, BWA, SOAP2, and 2BWT for short read alignment (mapping short DNA sequences to a reference genome); see [5]. More recently, it was suggested to use it also in *de novo* sequence assembly [23]. In the field of genome comparisons, this approach was first used in the software-tool *bbwt* [13], which uses k-mers (exact matches of fixed length k that appear in both sequences) as a basis of the comparison. It should be stressed that all these software-tools rely on the BWT (backward search) but not on the LCP-array. However, there is at least one algorithm that uses both BWT and LCP [18], and we expect that others will follow. The algorithm from [18] can be used in genome comparisons because it

computes maximal exact matches (exact matches that cannot be extended in either direction towards the beginning or end without allowing for a mismatch) between two long strings (e.g. chromosomes).

In the last years, several algorithms have been proposed that construct the BWT either directly or by first constructing the suffix array and then deriving the BWT in linear time from it. The latter approach has a major drawback: all known SACAs require at least $5n$ bytes of main memory (provided that $n < 2^{32}$). If one has to deal with large datasets, it is therefore advantageous to construct the BWT directly. For example, Okanohara and Sadakane [19] have shown that the SACA devised by Nong et al. [17] can be modified so that it directly constructs the BWT. Because it does not have to store the suffix array, it requires only $O(n \log \sigma \log \log_\sigma n)$ bits to construct the BWT [19] (ca. $2.5n$ bytes in practice; Sadakane, personal communication 2011).

As described above, some sequence analysis tasks require the longest common prefix array (LCP-array): an array containing the lengths of the longest common prefix between every pair of consecutive suffixes in SA; see Fig. 1. Formally, the LCP-array is defined by $\mathsf{LCP}[1] = -1$, $\mathsf{LCP}[n+1] = -1$, and $\mathsf{LCP}[i] = |lcp(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$, where $lcp(u, v)$ denotes the longest common prefix between the strings u and v. There are several linear time LCP-array construction algorithms (LACAs); see Section 2. They all first construct the suffix array and then obtain the LCP-array in linear time from it. So these LACAs suffer from the same drawback as mentioned above: at least $5n$ bytes of main memory are required. Here, we present the first LACA that acts directly on the Burrows-Wheeler transformed string (and not on the suffix array). The algorithm has a worst case time complexity of $O(n\sigma)$. Hence its run time is linear for a constant size alphabet. Moreover, we provide a practical implementation with a worst case time complexity of $O(n \log n)$ that requires approximately $2.2n$ bytes (throughout the paper, log stands for $\log_2$). Finally, we show that the approach can be used in other applications.

2 Related Work

In their seminal paper [14], Manber and Myers did not only introduce the suffix array but also the longest-common-prefix array. They showed that both the suffix array and the LCP-array can be constructed in $O(n \log n)$ time for a string of length n. Kasai et al. [12] gave the first linear time algorithm for the computation of the LCP-array. Their algorithm uses the string S, the suffix array, the inverse suffix array, and of course the LCP-array. Each of the arrays requires $4n$ bytes, thus the algorithm needs $13n$ bytes in total (for an ASCII alphabet). The main advantage of their algorithm is that it is simple and uses at most $2n$ character comparisons. But its poor locality behavior results in many cache misses, which is a severe disadvantage on current computer architectures. Manzini [15] reduced the space occupancy of Kasai et al.'s algorithm to $9n$ bytes with a slow down of about $5\% - 10\%$. Kärkkäinen et al. [11] proposed another variant of Kasai et al.'s algorithm, which computes a permuted LCP-array (PLCP-array).

In the PLCP-array, the lcp-values are in text order (position order) rather than in suffix array order (lexicographic order). This algorithm is much faster than Kasai et al.'s algorithm because it has a much better locality behavior. However, in virtually all applications lcp-values are required to be in suffix array order, so that in a final step the PLCP-array must be converted into the LCP-array. In a different approach, Puglisi and Turpin [21] tried to avoid cache misses by using the difference cover method but their algorithm is still slower than Kasai et al.'s. Just recently, Gog and Ohlebusch [6] presented a very space efficient and fast LACA, which trades character comparisons for cache misses.

3 Wavelet Tree

The *wavelet tree* introduced by Grossi et al. [7] supports one backward search step in $O(\log \sigma)$ time. To explain this data structure, we may view the ordered alphabet Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] = \$ < \Sigma[2] < \ldots < \Sigma[\sigma]$. We say that an interval $[l..r]$ is an *alphabet interval*, if it is a subinterval of $[1..\sigma]$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows-Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the sub-alphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string BWT = `nle_pl$nnlleee_eaae` and the alphabet interval $[1..4]$. The string $\mathsf{BWT}^{[1..4]}$ is obtained from `nle_pl$nnlleee_eaae` by deleting the characters `l`, `n`, and `p`. Thus, $\mathsf{BWT}^{[1..4]}$ = `e_$eee_eaae`. The wavelet tree of the string BWT over the alphabet $\Sigma[1..\sigma]$ is a balanced binary search tree defined as follows. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector $B^{[l..r]}$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the sub-alphabet $\Sigma[l..m]$ and 1 if it belongs to the sub-alphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it belongs to the right subtree; see Fig. 1. Moreover, each bit vector B in the tree is preprocessed so that the queries $rank_0(B, i)$ and $rank_1(B, i)$ can be answered in constant time [10], where $rank_b(B, i)$ is the number of occurrences of bit b in $B[1..i]$. Obviously, the wavelet tree has height $O(\log \sigma)$. Because in an actual implementation it suffices to store only the bit vectors, the wavelet tree requires only $n \log \sigma$ bits of space plus $o(n \log \sigma)$ bits for the data structures that support rank queries in constant time.

Instead of reviewing the implementation of one backward search step on a wavelet tree, we present a generalization thereof: for an ω-interval $[i..j]$, the procedure *getIntervals*($[i..j]$) presented in Algorithm 1 returns the list of all $c\omega$-intervals. More precisely, it starts with the ω-interval $[i..j]$ at the root and traverses the wavelet tree in a depth-first manner as follows. At the current node v, it uses constant time rank queries to obtain the number $b_0 - a_0$ of zeros

Algorithm 1. For an ω-interval $[i..j]$, the function call $getIntervals([i..j])$ returns the list of all $c\omega$-intervals, and is defined as follows.

$getIntervals([i..j])$
  $list \leftarrow [\,]$
  $getIntervals'([i..j],[1..\sigma],list)$
  return $list$

$getIntervals'([i..j],[l..r],list)$
  if $l = r$ **then**
    $c \leftarrow \Sigma[l]$
    $add(list, [C[c]+i..C[c]+j])$
  else
    $(a_0,b_0) \leftarrow (rank_0(B^{[l..r]}, i-1), rank_0(B^{[l..r]}, j))$
    $(a_1,b_1) \leftarrow (i-1-a_0, j-b_0)$
    $m = \lfloor \frac{l+r}{2} \rfloor$
    if $b_0 > a_0$ **then**
      $getIntervals'([a_0+1..b_0],[l..m],list)$
    if $b_1 > a_1$ **then**
      $getIntervals'([a_1+1..b_1],[m+1..r],list)$

in the bit vector of v within the current interval. If $b_0 > a_0$, then there are characters in $\mathsf{BWT}[i..j]$ that belong to the left subtree of v, and the algorithm proceeds recursively with the left child of v. Furthermore, if the number of ones is positive (i.e. if $b_1 > a_1$), then it proceeds with the right child in an analogous fashion. Clearly, if a leaf corresponding to character c is reached with current interval $[p..q]$, then $[C[c]+p \,..\, C[c]+q]$ is the $c\omega$ interval. In this way, Algorithm 1 computes the list of all $c\omega$-intervals. This takes $O(k \log \sigma)$ time for a k-element list. Because the wavelet tree has less than 2σ nodes, $O(\sigma)$ is another upper bound for the wavelet tree traversal. Consequently, Algorithm 1 has a worst-case time complexity of $O(\min\{\sigma, k \log \sigma\})$, where k is the number of elements in the output list. As an illustration of Algorithm 1, we compute all intervals of the form $c\mathtt{e}$ in the suffix array of Fig. 1 by invoking $getIntervals([6..11])$; note that $[6..11]$ is the $\mathtt{e}$-interval. In the wavelet tree of Fig. 1, the visited nodes of the depth-first traversal are marked grey. The resulting list contains the three intervals $[1..1]$, $[13..15]$, and $[17..18]$. Algorithm 1 was developed by the fourth author [22] but others apparently had the same idea [3].

4 A LACA Based on the BWT

Pseudo-code of our new LACA is given in Algorithm 2; it relies on Algorithm 1.

We will illustrate Algorithm 2 by an example. The first interval which is pulled from the queue is the ε-interval $\langle[1..19],0\rangle$, and $getIntervals([1..19])$ returns a list of the seven ω-intervals $[1..1]$, $[2..3]$, $[4..5]$, $[6..11]$, $[12..15]$, $[16..18]$, and $[19..19]$, where $|\omega| = 1$ for each ω. For every interval $[i_k..j_k]$, $\mathsf{LCP}[j_k+1]$ is set to

Algorithm 2. Computation of the LCP-array in $O(n\sigma)$ time.

initialize the array $\mathsf{LCP}[1..n+1]$ /* i.e., $\mathsf{LCP}[i] = \bot$ for all $1 \leq i \leq n+1$ */
$\mathsf{LCP}[1] \leftarrow -1$; $\mathsf{LCP}[n+1] \leftarrow -1$
initialize an empty queue
$enqueue(\langle [1..n], 0 \rangle)$
while queue is not empty **do**
  $\langle [i..j], \ell \rangle \leftarrow dequeue()$
  $list \leftarrow getIntervals([i..j])$
  for each $[lb..rb]$ **in** *list* **do**
    if $\mathsf{LCP}[rb+1] = \bot$ **then**
      $enqueue(\langle [lb..rb], \ell + 1 \rangle)$
      $\mathsf{LCP}[rb+1] \leftarrow \ell$

0, except for the last one, because $\mathsf{LCP}[20] = -1$. These six intervals are pushed on the queue, with the ℓ-value 1. Next, $\langle [1..1], 1 \rangle$ is pulled from the queue. The list returned by $getIntervals([1..1])$ just contains the n$-interval [16..16]. Since $\mathsf{LCP}[17]$ has not been computed yet, $\langle [16..16], 2 \rangle$ is pushed on the queue, $\mathsf{LCP}[17]$ is set to 1, and so on.

It is not difficult to see that Algorithm 2 maintains the following invariant: The set of the second components of all elements of the queue has either one element ℓ or two elements ℓ and $\ell + 1$, where $0 \leq \ell < n$. In the latter case, the elements with second component ℓ precede those with second component $\ell + 1$.

Theorem 1. *Algorithm 2 correctly computes the* LCP*-array.*

Proof. We proceed by induction on ℓ. In the base case, we have $\ell = 0$. For every character $c = \Sigma[k]$ occurring in S, the c-interval $[lb..rb]$ is in the list returned by $getIntervals([1..n])$, where $lb = C[c] + 1$ and $rb = C[d]$ with $d = \Sigma[k+1]$. The algorithm sets $\mathsf{LCP}[rb{+}1] = 0$ unless $rb = n$. This is certainly correct because the suffix $S_{\mathsf{SA}[rb]}$ starts with the character c and the suffix $S_{\mathsf{SA}[rb+1]}$ starts with the character d. Clearly, because for every character c occurring in S the c-interval is in the list returned by $getIntervals([1..n])$, all entries of LCP with value 0 are set. Let $\ell > 0$. By the inductive hypothesis, we may assume that Algorithm 2 has correctly computed all lcp-values $< \ell$. After the last LCP-entry with value $\ell - 1$ has been set, the queue solely contains elements of the form $\langle [i..j], \ell \rangle$, where $[i..j]$ is the ω-interval of some substring ω of S with $|\omega| = \ell$. Let the $c\omega$-interval $[lb..rb]$ be in the list returned by $getIntervals([i..j])$. If $\mathsf{LCP}[rb+1] = \bot$, then we know from the induction hypothesis that $\mathsf{LCP}[rb+1] \geq \ell$, i.e., the suffixes $S_{\mathsf{SA}[rb]}$ and $S_{\mathsf{SA}[rb+1]}$ have a common prefix of length at least ℓ. On the other hand, $c\omega$ is a prefix of $S_{\mathsf{SA}[rb]}$ but not of $S_{\mathsf{SA}[rb+1]}$. Consequently, $\mathsf{LCP}[rb{+}1] < \ell{+}1$. Altogether, we conclude that Algorithm 2 assigns the correct value ℓ to $\mathsf{LCP}[rb+1]$.

We still have to prove that all entries of the LCP-array with value ℓ are really set. So let k, $0 \leq k < n$, be an index with $\mathsf{LCP}[k{+}1] = \ell$. Since $\ell > 0$, the longest common prefix of $S_{\mathsf{SA}[k]}$ and $S_{\mathsf{SA}[k+1]}$ can be written as $c\omega$, where $c \in \Sigma$, $\omega \in \Sigma^*$, and $|\omega| = \ell - 1$. Consequently, ω is the longest common prefix of $S_{\mathsf{SA}[k]+1}$ and

$S_{\mathsf{SA}[k+1]+1}$. Let $[i..j]$ be the ω-interval, p be the index with $\mathsf{SA}[p] = \mathsf{SA}[k] + 1$, and q be the index with $\mathsf{SA}[q] = \mathsf{SA}[k+1] + 1$. Clearly, $i \le p < q \le j$. Because ω is the longest common prefix of $S_{\mathsf{SA}[p]}$ and $S_{\mathsf{SA}[q]}$, there must be at least one index t with $p < t \le q$ so that $\mathsf{LCP}[t] = |\omega| = \ell - 1$. If there is more than one index with that property, let t denote the smallest. According to the inductive hypothesis, Algorithm 2 assigns the value $\ell - 1$ to $\mathsf{LCP}[t]$. Just before that, a pair $\langle [s..t-1], \ell \rangle$ must have been pushed to the queue, where $[s..t-1]$ is some ω'-interval with $|\omega'| = \ell$. By the definition of t, we have $\mathsf{LCP}[r] > \ell - 1$ for all r with $p < r \le t - 1$. Thus, p lies within the interval $[s..t-1]$. In other words, ω' is a prefix of $S_{\mathsf{SA}[p]}$. Moreover, $\mathsf{BWT}[p] = c$ implies that the $c\omega'$-interval, say $[lb..rb]$, is not empty. Since $\mathsf{BWT}[r] \ne c$ for all $p < r < q$, it follows that $rb = k$. At some point, $\langle [s..t-1], \ell \rangle$ is removed from the queue, and $[lb..k]$ is in the *list* returned by *getIntervals*$([s..t-1])$. Consequently, $\mathsf{LCP}[k+1]$ will be set to ℓ. □

Algorithm 2 has a worst case time complexity of $O(n\sigma)$. This can be seen as follows. The number of iterations of its while-loop depends on the overall number of elements that enter the queue. Because an element is pushed on the queue only if an LCP-entry is set, the while-loop is executed $n - 1$ times. Within the while-loop, both the procedure call *getIntervals*$([i..j])$ and the for-loop take $O(\sigma)$ time, whereas the remaining statements take constant time.

Algorithm 2 uses only the wavelet tree of the BWT of S, a queue to store the ω-intervals and the LCP-array. Because a practical implementation should use as little space as possible, we next show how to reduce the space consumption of the latter two. The second components (ℓ-values) of the queue entries need not be stored because one can simply count how many ω-intervals with $|\omega| = \ell$ enter the queue; see Algorithm 3 for details. For a fixed ℓ, the ω-intervals with $|\omega| = \ell$ do not overlap. Thus, they can be stored in two bit vectors, say B and E of size n, and an ω-interval $[i..j]$ is stored by setting the bits $B[i]$ and $E[j]$ (due to singleton intervals, we actually need two bit vectors). By the invariant mentioned above, at any point in time Algorithm 2 has to deal with at most two different ℓ-values. Therefore, we can replace the queue with four bit vectors of length n. The price to be paid for this is an increase in the worst case time complexity. For each ℓ-value, two bit vectors of length n are scanned to determine all ω-intervals with $|\omega| = \ell$. So the number of scans is proportional to the maximum lcp-value. Since this can be $n - 2$ (consider the string $S = a^{n-1}\$$), the time complexity rises to $O(n^2)$. For this reason, we prefer the following hybrid approach. For a fixed ℓ, if there are more than $\frac{n}{2\log n}$ ω-intervals with $|\omega| = \ell$ (note that this can happen only $O(\log n)$ times), we use the bit vectors; otherwise we use the queue. More precisely, we start with the queue, switch to the bit vectors when there are more than $\frac{n}{2\log n}$ ω-intervals with the current ℓ-value, and switch back if there are less than $\frac{n}{2\log n}$ ω-intervals with the current ℓ-value. Note that the queue uses at most n bits because each queue entry is an interval that can be represented by two numbers using $\log n$ bits each. In this hybrid approach, the identification of ω-intervals takes $O(n \log n)$ time because the bit vectors of length n are scanned $O(\log n)$ times. Under the assumption that $\sigma \le c \log n$ for some constant c, this is also the worst case time complexity of the whole algorithm. However, our

experiments showed that this $O(n \log n)$ time implementation of Algorithm 2 is actually faster than the $O(n\sigma)$ time implementation which solely uses the queue (from the STL of C++). This can be attributed to fewer cache misses due to the fact that the bit vectors are accessed sequentially.

Up to now, our LACA needs $n \log \sigma$ bits for the wavelet tree of the BWT plus $o(n \log \sigma)$ bits for the data structures that support rank/select queries in constant time, $4n$ bits for the storage of the ω-intervals, and $4n$ bytes for the LCP-array itself. Our goal is to stay below $2.5n$ bytes because this is (currently) the space that is needed to build the BWT; cf. Sect. 1. To meet this goal, we stream the LCP-array to disk. This is possible because Algorithm 2 calculates lcp-values in ascending order. Clearly, the LCP-array requires only $k \cdot n$ bits to store all lcp-values less than 2^k. During the computation of these lcp-values, the i-th bit of a bit vector D of length n is set when a value is assigned to $\mathsf{LCP}[i]$. Afterwards the LCP-array is written to disk, but the bit vector D tells us which LCP-entries are already done and we preprocess D so that rank queries can be answered in constant time. Let m be the number of zeros in D. We use a new array A of length m that also occupies $k \cdot n$ bits. In other words, each array element of A consists of $b = \lfloor \frac{k \cdot n}{m} \rfloor$ bits, which are initially set to zero. Then, we compute all lcp-values less than $2^k + 2^b - 1$. When a value ℓ is to be assigned to $\mathsf{LCP}[i]$, we store the value $\ell - 2^k + 1$ in $A[rank_0(D, i)]$. After all lcp-values less than $2^k + 2^b - 1$ have been computed, we further proceed as follows. During a scan of the bit vector D, we count the number of zeros seen so far. So when an index i with $D[i] = 0$ is encountered, we know that this is, say, the j-th zero seen so far. Now we use a case analysis. If $A[j] = 0$, then $\mathsf{LCP}[i]$ has not been computed yet and there is nothing to do. Otherwise, the value $2^k - 1 + A[j]$ is written at index i to the LCP-array on the disk, and $D[i]$ is set to one. When the scan of D is completed, the (updated) bit vector D is preprocessed so that rank queries can be answered in constant time. This process is iterated (calculate the new values of m and b, initialize a new array A, etc.) until the LCP-array is completely filled.

In several applications, the access to the LCP-array is sequential, so it can be streamed from disk. If random access is needed, one can get a compressed representation of the LCP-array by streaming it from disk, and then this compressed version is kept in main memory. There are several compressed versions of the LCP-array which use about 1 byte per entry in practice, while the access time remains essentially the same as for the uncompressed version; see e.g. [1].

5 Experimental Results

Programs were compiled using `gcc` version 4.4.3 with options `-O9 -DNDEBUG` on a 64 bit Ubuntu (Kernel 2.6.32) system equipped with a six-core AMD Opteron processor 2431 with 2.4 GHz and 32GB of RAM. The data originates from the Pizza&Chili Corpus (`http://pizzachili.dcc.uchile.cl/`). For space reasons, we report only the results for files of 200 MByte. Additionally, the genome of the house mouse (NCBIm36, `http://www.ensembl.org/Mus_musculus`) was used.

Table 1. Experimental results: for each file, the first column shows the real runtime in seconds and the second column shows the maximum memory usage per character. As an example, consider the 200MB file dna. The construction of its suffix array takes 67 sec. and $5n$ bytes (1000MB), whereas the direct construction of its BWT takes 88 sec. and $1.9n$ bytes (380MB). Rows 3-7 refer to the construction of the LCP-array under the assumption that the suffix array (the BWT, respectively) has already been built. In rows 8-12, the first column shows the overall runtime and the second shows the overall maximum memory usage per character. For a fair comparison of the run time, the data was chosen in such a way that all data structures fit in the main memory. Of course, for very large files the space usage of $9n$ bytes is disadvantageous because then the data structures must reside in secondary memory, and this slows down the algorithms.

	dna 200MB		english 200MB		proteins 200MB		sources 200MB		xml 200MB		mouse genome 3242MB	
SA constr.	67	5	67	5	73	5	44	5	48	5	1876	8.9
BWT constr.	88	1.9	106	2.2	147	2.6	89	2.2	83	2.2	1556	2.1
KLAAP	54	9	47	9	47	9	31	9	32	9	1530	9
Φ	35	9	30	9	30	9	21	9	23	9	1122	9
Φ64	79	5.1	82	5.1	77	5.1	59	5.1	72	5.1	-	-
goPHI	50	2	74	2	71	2	51	2	49	2	1338	2.3
new algorithm	66	1.8	123	2	133	2	131	2.2	99	2.1	1488	1.8
KLAAP	121	9	114	9	120	9	75	9	80	9	3406	9
Φ	102	9	97	9	103	9	65	9	71	9	2998	9
Φ64	146	5.1	149	5.1	150	5.1	103	5.1	120	5.1	-	-
goPHI	117	5	141	5	144	5	95	5	97	5	3214	8.9
new algorithm	154	1.9	229	2.2	280	2.6	220	2.2	182	2.2	3044	2.1

We compared our new algorithm with the KLAAP-algorithm of Kasai et al. [12], the Φ and Φ64 algorithms[1] of Kärkkäinen et al. [11], and the goPHI algorithm of Gog and Ohlebusch [6]. The suffix array construction was done by Mori's libdivsufsort-algorithm[2] (`http://code.google.com/p/libdivsufsort/`), while the direct BWT construction is due to Okanohara and Sadakane [19]. The KLAAP-algorithm is our own implementation, all other programs were kindly provided by the authors. Looking at the experimental results in Table 1, one can see that the Φ algorithm is the fastest LACA. However, in large scale applications its space usage of $9n$ bytes is the limiting factor. The memory usage of algorithm goPHI (row 6) is similar to that of our algorithm but it relies on the suffix array, so its overall space usage (row 11) is due to the suffix array construction (row 1). By contrast, our new algorithm solely depends on the BWT, so that its overall maximum memory usage per character is approximately $2.2n$ bytes (row 12). It can be attributed to the usual space-time trade-off that our new algorithm is the slowest LACA in the contest for the 200MB files. However, this changes in large scale applications when memory is tight.

[1] The Φ64 algorithm is limited to files of size $\leq 2^{31}$, because it is implemented with signed 32 bit integers.

[2] Because the 32 bit version is also limited to files of size $\leq 2^{31}$, we had to use the 64 bit version for the mouse genome (which needs $9n$ bytes).

6 Other Applications

There are at least two other problems to which our new approach can be applied. As a first application, we will briefly describe how to find all shortest unique substrings in optimal time. This is relevant in the design of primers for DNA sequences; As a second application, our approach allows us to compute shortest absent words; see e.g. [9]. This is relevant because short DNA sequences that do not occur in a genome are interesting to biologists. For example, the fact that the human genome does not contain all possible DNA sequences of length 11 may be due to negative selection. For space reasons, however, we can present only the first application. Because \$ is solely used to mark the end of the string S, we have to exclude it in the considerations below.

Definition 1. *A substring* $S[i..j]$ *is* unique *if it occurs exactly once in* S*. The* shortest unique substring problem *is to find all shortest unique substrings of* S*.*

Clearly, every suffix of S is unique because S is terminated by the special symbol \$. Since we are not interested in these, we will exclude them. One can show that the $(\ell+1)$-length prefix of $S_{\mathsf{SA}[i]}$, where $\ell = \max\{\mathsf{LCP}[i], \mathsf{LCP}[i+1]\}$, is the shortest unique substring of S that starts at position $\mathsf{SA}[i]$. Using this observation, we can modify Algorithm 2 so that it computes a shortest unique substring of S. The resulting Algorithm 3 can easily be changed so that it computes all shortest unique substrings or even all unique substrings of S. We make use of the fact that Algorithm 2 computes lcp-values in ascending order. So when Algorithm 2 executes the statement $\mathsf{LCP}[rb+1] \leftarrow \ell$ and $\mathsf{LCP}[rb]$ has been set before, then $\max\{\mathsf{LCP}[rb], \mathsf{LCP}[rb+1]\} = \ell$ and $S[\mathsf{SA}[rb]..\mathsf{SA}[rb]+\ell]$ is the shortest unique substring of S that starts at position $\mathsf{SA}[rb]$. Analogously, if $\mathsf{LCP}[rb+2]$ has been set before, then $\max\{\mathsf{LCP}[rb+1], \mathsf{LCP}[rb+2]\} = \ell$ and $S[\mathsf{SA}[rb+1]..\mathsf{SA}[rb+1]+\ell]$ is the shortest unique substring of S that starts at position $\mathsf{SA}[rb+1]$. Because the current value of ℓ is always available, all we have to know is whether or not $\mathsf{LCP}[rb]$ ($\mathsf{LCP}[rb+2]$, respectively) has been computed before. Consequently, we can replace the LCP-array with the bit vector D of length n, and $D[i]$ is set to one instead of assigning a value to $\mathsf{LCP}[i]$. However, there are two subtleties that need to be taken into account. First, the suffix array is not at hand, so we have to find an alternative way to output the string $S[\mathsf{SA}[rb]..\mathsf{SA}[rb]+\ell]$. Second, we have to exclude this string if it is a suffix. Fortunately, the wavelet tree provides the needed functionality, as we shall see next. The LF-mapping is defined by $LF(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i]-1]$ for all i with $\mathsf{SA}[i] \neq 0$ and $LF(i) = 0$ otherwise (where SA^{-1} denotes the inverse of the permutation SA). Its long name *last-to-first column mapping* stems from the fact that it maps the last column $L = \mathsf{BWT}$ to the first column F, where F contains the first character of the suffixes in the suffix array, i.e., $F[i] = S[\mathsf{SA}[i]]$. Its inverse is defined by $\psi(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i]+1]$ for all i with $2 \leq i \leq n$ and $\psi(1) = \mathsf{SA}^{-1}[1]$. With the wavelet tree, both $LF(i)$ and $\psi(i)$ can be computed in $O(\log \sigma)$ time; see [16]. Moreover, the character $F[i]$ can be determined in $O(\log \sigma)$ time by a binary search on C. Since $S[\mathsf{SA}[rb]] = F[rb]$, $S[\mathsf{SA}[rb]+1] = F[\psi(rb)]$, $S[\mathsf{SA}[rb]+2] = F[\psi(\psi(rb))]$ etc., it follows that the string $S[\mathsf{SA}[rb]..\mathsf{SA}[rb]+\ell]$ coincides with $F[rb]\ F[\psi(rb)]\ \ldots\ F[\psi^{\ell}(rb)]$ (which

Algorithm 3. Computation of a shortest unique substring.

```
initialize a bit vector D[1..n+1] /* i.e., D[i] = 0 for all 1 ≤ i ≤ n+1 */
D[1] ← 1; D[n+1] ← 1
initialize an empty queue
enqueue([1..n])
ℓ ← 0; size ← 1; idx ← 1
while queue is not empty do
  if size = 0 then
    ℓ ← ℓ + 1
    size ← current size of the queue
    idx ← LF(idx)
  [i..j] ← dequeue()
  size ← size − 1
  list ← getIntervals([i..j])
  for each [lb..rb] in list do
    if D[rb+1] = 0 then
      enqueue([lb..rb])
      D[rb+1] ← 1
      if D[rb] = 1 and rb ≠ idx then
        return F[rb] F[ψ(rb)] ... F[ψ^ℓ(rb)] /* the string S[SA[rb]..SA[rb]+ℓ] */
      if D[rb+2] = 1 and rb+1 ≠ idx then
        return F[rb+1] F[ψ(rb+1)] ... F[ψ^ℓ(rb+1)]
```

can be computed in $O(\ell \log \sigma)$ time). This solves our first little problem. The second problem was to exclude suffixes from the output. This can be done by keeping track of the suffix of length $\ell + 1$, where ℓ is the current length. To be precise, initially $\ell = 0$ and the suffix of length 1 is the character \$, which appears at index $idx = 1$. Every time ℓ is incremented, we obtain the index of the suffix of length $\ell + 1$ by the assignment $idx \leftarrow LF(idx)$. Consequently, a unique substring at index rb is output only if $rb \neq idx$.

References

1. Brisaboa, N.R., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 122–130. Springer, Heidelberg (2009)
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
3. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
5. Flick, P., Birney, E.: Sense from sequence reads: Methods for alignment and assembly. Nature Methods 6(11 suppl.), S6–S12 (2009)

6. Gog, S., Ohlebusch, E.: Lightweight LCP-array construction in linear time (2011), arxiv.org/pdf/1012.4263
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc.14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
8. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
9. Herold, J., Kurtz, S., Giegerich, R.: Efficient computation of absent words in genomic sequences. BMC Bioinformatics 9, 167 (2008)
10. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th Annual Symposium on Foundations of Computer Science, pp. 549–554. IEEE, Los Alamitos (1989)
11. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
12. Kasai, T., Lee, G.H., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
13. Lippert, R.A.: Space-efficient whole genome comparisons with Burrows-Wheeler transforms. Journal of Computational Biology 12(4), 407–415 (2005)
14. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
15. Manzini, G.: Two space saving tricks for linear time LCP array computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
16. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article 2 (2007)
17. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. Data Compression Conference, pp. 193–202. IEEE Computer Society, Los Alamitos (2009)
18. Ohlebusch, E., Gog, S., Kügel, A.: Computing matching statistics and maximal exact matches on compressed full-text indexes. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 347–358. Springer, Heidelberg (2010)
19. Okanohara, D., Sadakane, K.: A linear-time burrows-wheeler transform using induced sorting. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 90–101. Springer, Heidelberg (2009)
20. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
21. Puglisi, S.J., Turpin, A.: Space-time tradeoffs for longest-common-prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
22. Schnattinger, T.: Bidirektionale indexbasierte Suche in Texten. Diploma thesis, University of Ulm, Germany (2010)
23. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics 26(12), i367–i373 (2010)

A Succinct Index for Hypertext

Chris Thachuk

Department of Computer Science, University of British Columbia, Vancouver, Canada
cthachuk@cs.ubc.ca

Abstract. Recent advances in nucleic acid sequencing technology has motivated research into succinct text indexes to represent reference genomes that support efficient pattern matching queries. Similar sequencing technology can also produce millions of reads (patterns) derived from transcripts which need to be aligned to a reference transcriptome. A transcriptome can be modeled as a hypertext. Motivated by this application, we propose the first succinct index for hypertext. The index can model any hypertext and places no restriction on the graph topology. We also propose a new pattern matching algorithm, capable of aligning a pattern to any path in the hypertext, that is especially efficient when few nodes of the hypertext share the same text—in this important case, our algorithm is a significant improvement over all existing approaches.

1 Introduction

Fueling the discovery of genetic variation amongst populations and individuals has been the application of next generation sequencing technology (NGS). The new technology focuses on massively parallel sequencing and is capable of producing millions of reads (patterns) in a typical run [10,9]. Due to the sheer volume of data, the task of efficiently aligning reads to a reference genome is one of the most actively researched problems in contemporary bioinformatics. NGS is also being utilized to capture data from the transcriptome; a process referred to as RNA-Seq [13]. Instead of sequencing genomic DNA, RNA-Seq aims to sequence the complementary DNA (cDNA) of RNA molecules in a cell. Transcriptome read alignment is providing valuable information to researchers, beyond genomic sequencing. In particular, this technology can be used to quantify the level of expression of various transcripts by sequencing messenger RNA, thus implicating the relative expression level of proteins.

Much more progress has been made in mapping reads from genome data to reference genomes than on aligning reads derived from transcriptomes. The latter problem is harder by the very nature of the events it is capable of capturing compared to genomic sequencing. Since introns are spliced from genes in the process of transcription (see Figure 1), *spliced reads* may map to two regions of the genome that are separated by many hundreds or thousands of bases. The difficulty of aligning NGS reads that span intron boundaries is exacerbated by their short length, and often is not attempted, resulting in a significant loss of information. The transcriptome read alignment problem is modeled more accurately by the problem of aligning patterns to a *hypertext*.

Informally, hypertext is a generalization of text from a linear structure to a directed graph, $G = (V, E)$, with each node being a fragment of text and edges implying which

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 209–220, 2011.
© Springer-Verlag Berlin Heidelberg 2011

fragments of text can be appended; thus, any path in the graph is a substring of the hypertext. The example transcriptome in Figure 1 consists of five overall exons between two genes. The splicing events, and valid transcripts are also shown. The resulting hypertext model of this transcriptome has a node for each exon, and an edge between exons joined by a splicing event, resulting in two components (one for each gene).

The seminal work on pattern matching in hypertext is due to Manber and Wu [12] who proposed a $O(|V|+m|E|+occ \log \log m)$ time algorithm, where m is the length of the pattern and occ are the number of matches. Akutsu [1] proposed an $O(n)$ algorithm for matching in hypertext forming a tree structure, where n is the total length of text in all nodes. Park and Kim [16] considered the case where the hypertext forms a directed acyclic graph by proposing a $O(n + m|E|)$ time algorithm, under the assumption that no node in G matches to more than one position in the pattern. Amir *et al.,* [2] proposed an algorithm with the same runtime complexity; however, theirs was the first algorithm for the case of hypertext forming a general graph. Amir *et al.,* [2] and Navarro [14] also considered the problem of approximate matching in hypertext.

In all cases, the runtimes of the previously proposed pattern matching algorithms in hypertext are impractical for alignment of millions of transcriptome reads. Surprisingly, no succinct index for hypertext has been previously proposed. In this work, we propose a succinct index to model hypertext and focus on the case where the space required to index the text dominates the space to represent the topology of the graph. We note that in the case of the Human transcriptome, exons number in the tens of thousands, splicing events in the hundreds of thousands, and the combined length of exons in the hundreds of millions. Our index can model any hypertext forming a general graph. We also propose a new pattern matching algorithm, capable of aligning a pattern to any path in the hypertext, that is especially efficient for hypertexts where few nodes share the same text. In particular, our new algorithm can report all patterns crossing at most one edge—a valid assumption for current transcriptome read datasets—in $O(m \log \sigma + m\frac{\log |V|}{\log \log |V|} + occ_1 \log n + occ_2 \frac{\log |V|}{\log \log |V|})$ time, where occ_1 (occ_2) is the number of matches that cross no (one) edge. We also consider a restricted version of the problem, where only certain paths in the hypertext are considered valid.

2 Preliminaries

For a string T of length n over an alphabet Σ, let $T[i]$ denote the ith character of T, and let $T[i \dots j]$ denote the substring from the ith to the jth character of T, for $i \leq j$. The ith suffix of T is the substring $T[i \dots n]$. A suffix array of $T\$$, the string T followed by a special sentinel character \$, is a permutation of the integers $[1, \dots, n+1]$ giving the lexicographic order of all suffixes of $T\$$, where $\$ \notin \Sigma$ and $\$ < c, \forall c \in \Sigma$. Conceptually, the suffix array can be thought of as a matrix where each row is a different suffix of $T\$$, and the rows are in lexicographic order. See Figure 2 for an example.

2.1 Compressed Suffix Arrays

A compressed suffix array F is a *succinct* representation of the Burrows-Wheeler transform (BWT) of the string T, denoted as $\mathsf{F}^{\mathtt{BWT}}$, in addition to some auxiliary

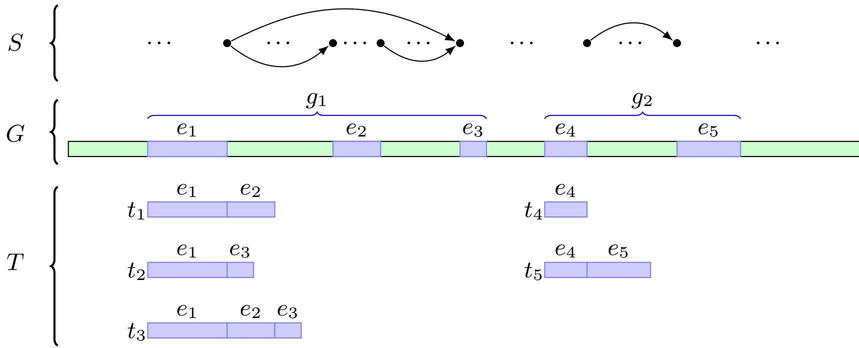


Fig. 1. A simple genome, G, having five exons contained in two genes, along with its transcriptome, T, consisting of five transcripts. Also shown is a splicing graph S where each directed edge denotes a splicing event found in T.

data structures. The ith character in the string $\mathsf{F}^{\mathtt{BWT}}$ corresponds to the character, in T, that *precedes* the ith lexicographically smallest suffix of T. See Figure 2 for an example. The structure of the suffix array of T can be inferred directly from $\mathsf{F}^{\mathtt{BWT}}$ by the so-called LF-mapping. Specifically, the jth occurrence of a character c in $\mathsf{F}^{\mathtt{BWT}}$ corresponds to the jth lexicographically smallest suffix of T that begins with the character c.

Ferragina and Manzini [7] showed that a pattern $P[1 \dots m]$ can be matched against T by performing a *backward search* in F. The backward search initially finds matches of $P[m \dots m]$ in T, then attempts to extend those into matches of $P[m-1 \dots m]$ in T, and so on. The search maintains a *suffix array range* denoting the interval in the sorted suffixes that match the current pattern as a prefix. If the final range $[a, b]$ is non-empty, then P matches in T exactly $b - a + 1$ times. A compressed suffix array can also report the locations of all matches in T, if any. Details of backward search, and compressed suffix arrays in general, can be found in the review by Navarro and Mäkinen [15].

Lemma 1 (Grossi *et al.*, [8]). *A compressed suffix array for a string T, of length n, over an alphabet of size σ, can be represented in $(1 + o(1))n \log \sigma$ bits of space, such that the suffix array range for every suffix of a string P can be computed in $O(|P| \log \sigma)$ time, and each match of P in T can be reported in an additional $O(\log n)$ time.*

A *full-text dictionary* is designed to index a collection of patterns. It has the same capabilities as a compressed suffix array, and in addition can report all patterns from the dictionary that are contained in an input text P.

Lemma 2 (Thachuk [18]). *A succinct full-text dictionary $\mathcal{D}$ of a set of k patterns, having combined length n, over an alphabet of size σ, can be represented in $(1 + o(1))n \log \sigma + O(n) + k(\lceil \log \frac{n}{k} \rceil + 2 + o(1))$ bits such that the matching statistics of a string P with respect to $\mathcal{D}$ can be determined in $O(|P| \log \sigma)$ time, all occ_1 patterns contained in P can be reported in an additional $O(occ_1)$ time, and all occ_2 positions where P is contained within a pattern can be reported in an additional $O(occ_2 \log n)$ time.*

	F^BWT		R^BWT
$	t	$	c
ϕacaϕgϕgaϕcgϕct$	$	ϕacaϕgϕagϕgcϕtc$	$
ϕcgϕct$	a	ϕagϕgcϕtc$	g
ϕct$	g	ϕgϕagϕgcϕtc$	a
ϕgϕgaϕcgϕct$	a	ϕgcϕtc$	g
ϕgaϕcgϕct$	g	ϕtc$	c
aϕcgϕct$	g	aϕgϕagϕgcϕtc$	c
aϕgϕgaϕcgϕct$	c	acaϕgϕagϕgcϕtc$	ϕ
acaϕgϕgaϕcgϕct$	ϕ	agϕgcϕtc$	ϕ
caϕgϕgaϕcgϕct$	a	c$	t
cgϕct$	ϕ	cϕtc$	g
ct$	ϕ	caϕgϕagϕgcϕtc$	a
gϕct$	c	gϕagϕgcϕtc$	ϕ
gϕgaϕcgϕct$	ϕ	gϕgcϕtc$	a
gaϕcgϕct$	ϕ	gcϕtc$	ϕ
t$	c	tc$	ϕ

Fig. 2. (left) An example of the underlying compressed suffix array in F, for the text $T = \phi aca\phi g\phi ga\phi cg\phi ct\$$, representing the serialization of possible text in exons $e_1, \ldots, e_5$ from Figure 1. (right) The compressed suffix array R for the text $T^R = \phi aca\phi g\phi ag\phi gc\phi tc\$$.

The *matching statistics* of a string P, with respect to T, is a list of tuples $(q, [a, b])$, one for each suffix of P, where q denotes the longest match of the suffix anywhere in T, and $[a, b]$ is the suffix array range of the matches.

2.2 Orthogonal Range Query Structures

An orthogonal range query data structure indexes a set of points from a two-dimensional grid so that given as input a bounding rectangle, all points contained within the bounds can be reported efficiently.

Lemma 3 (Bose *et al.*, [4]). *A set N of points from universe $M = [1..k] \times [1..k]$, where $k = |N|$, can be represented in $(1 + o(1))k \log k$ bits to support orthogonal range counting in $O(\frac{\log k}{\log \log k})$ time, and orthogonal range reporting in $O(occ\frac{\log k}{\log \log k})$ time, where occ is the size of the output.*

2.3 Hypertext

A hypertext generalizes the notion of text to be a directed graph $G = (V, E)$ such that each node $v \in V$ contains text over an alphabet Σ and the outgoing edges of v are incident to nodes containing text that can follow v's. A match of a pattern P to the hypertext G is a path $p = v_1, \ldots, v_k$ through G, and an offset l into the first node v_1, such that P matches the concatenation of the text in nodes $v_1, \ldots, v_k$, beginning at position l in v_1, and ending at some prefix of v_k.

Pattern Matching in Hypertext Problem

Instance: A hypertext G and a pattern P.
Question: Which paths in G match P?

Previous algorithms for matching in hypertext focused on reporting only the initial node, and offset within that node, of paths in G matching P. For our motivating problem of aligning patterns to a transcriptome, the actual path is required to be known, and that is our focus in the remainder of the paper. However, our matching algorithm can be simplified if only the initial node of a match (and the offset within the node) is desired.

3 Construction of the Hypertext Index

The succinct hypertext index is a collection of three sets of data structures: those storing the node text, those storing the graph topology, and a useful auxiliary structure. In our pattern matching algorithms we find it useful to identify nodes of the graph by two different identifiers: *forward id*, and *reverse id*. This is reflected in our descriptions of the data structures below. The forward id gives the prefix lexicographic rank of the text contained within the node as compared with all other nodes in V. Similarly, the reverse id gives the rank with respect to the suffix lexicographic rank. We show how these ids can be determined in Section 3.3.

3.1 Indexing Node Text

For a given hypertext $G = (V, E)$ over an alphabet Σ of size σ, we construct a text $T = \phi v_1 \phi v_2 \phi \dots \phi v_{|V|}\$$, of length n, that is a serialization of the combined text of the nodes of V, each prefixed by a character $\phi < c, \forall c \in \Sigma$. We will construct and store a *full-text dictionary* index F of T. We also construct and store a compressed suffix array R for T^R, the serialization of the reverse of all node text. We let $\mathsf{F}^{\mathtt{BWT}}$ ($\mathsf{R}^{\mathtt{BWT}}$) denote the BWT string for F (R). See Figure 2 for an example.

3.2 Storing Graph Topology

We store the graph topology in a 2D range query index, Q, that is heavily utilized in our pattern matching algorithm. Conceptually, the y-axis corresponds to *forward id*s and the x-axis corresponds to *reverse id*s. A point (a, b) is added to the index if and only if in E there is an edge from the node with reverse id a to the node with forward id b.

3.3 Auxiliary Data Structures

Each node can be ranked according to its prefix lexicographic order in the forward index F. For instance, we can determine the prefix lexicographic order of all $|V|$ nodes by performing backward search on the serialized text T. After the text '$v_{|V|}\$$' has been matched in F, three facts are known: (i) the matching suffix array range $[a, b]$ will be a size one interval (*i.e.,* $a = b$) since \$ occurs only in one position of T, (ii) $\mathsf{F}^{\mathtt{BWT}}[a] = \phi$ since each node is prefixed by a ϕ character, and (iii) the rank of the ϕ character at position a in $\mathsf{F}^{\mathtt{BWT}}$ corresponds to the prefix lexicographic rank of node $v_{|V|}$, with respect to all other nodes in V, due to the properties of the LF mapping. We can continue to determine the prefix lexicographic order of all nodes in V in a single traversal of T. Similarly, we can determine the suffix lexicographic order of all nodes using the

reverse index. We find it convenient to store a permutation $\Pi_{R \to F}$ that maps the suffix lexicographic order of a node in the reverse index to the prefix lexicographic order of that node in the forward index. We report all matches with respect to the forward id label and assume edges are defined with respect to forward ids. Should matches require reporting of an alternative labeling of the nodes, such as exon identifiers in the case of transcriptomes, then another auxiliary permutation can be stored for that purpose. Our overall space usage is summarized in Table 1 and Lemma 4.

Table 1. Inventory of space usage for succinct hypertext index

Symbol	Description	Space (bits)
F	full-text dictionary of forward text	$(1+o(1))n\log\sigma + O(n) + \lvert V\rvert(\lceil\log\frac{n}{\lvert V\rvert}\rceil + 2 + o(1))$
R	compressed suffix array of reverse text	$(1+o(1))n\log\sigma$
Q	2D point structure containing graph topology	$(1+o(1))\lvert E\rvert\log\lvert E\rvert$
$\Pi_{R\to F}$	mapping of reverse id to forward id	$\lvert V\rvert\lceil\log\lvert V\rvert\rceil$

Lemma 4. *A hypertext $G = (V, E)$ can be represented in $(2 + o(1))n \log \sigma + O(n) + (1 + o(1))|E| \log |E| + |V|(\log |V| + \log \frac{n}{|V|} + 2 + o(1))$ bits by the above scheme, where the text of the nodes in V are over an alphabet of size σ and have a combined length of $n - |V|$.*

4 Pattern Matching in the Hypertext Index

We now demonstrate that extensions of techniques used to solve the problem of matching a pattern against a text containing wildcards are applicable and effective for matching a pattern in a hypertext. In a solution to the wildcard matching problem, Lam *et al.,* [11] classified a match of a pattern P to the text T into three cases: P matches a position in T containing no wildcard group, P matches a position in T containing one wildcard group, and P matches a position in T containing more than one wildcard group, where a wildcard group is a consecutive sequence of wildcard characters. We will solve the problem of matching a pattern P in a hypertext $G = (V, E)$ by considering three analogous cases: (i) P does not span any edge from E, (ii) P spans exactly one edge from E, and (iii) P spans more than one edge from E. As we will see, case (i) is identical, case (ii) is a restriction, and case (iii) is a generalization of the respective wildcard cases.

4.1 Preprocessing the Pattern

Considering that a match of P is a path through G, we need an efficient means to determine which nodes contain P as a substring, which are contained within P, which nodes contain a prefix of P as a suffix, and which contain a suffix of P as a prefix. Consider for a moment how we may determine which nodes contain the suffix $P[i \ldots m]$ as a prefix. Suppose $P[i \ldots m]$ has a non-empty suffix array range $[a, b]$ in the forward index F. If $P[i \ldots m]$ is a prefix of some node v_i then two things must be true: (i) $[c, d]$, the suffix array range of v_i in F, must be a sub-interval of $[a, b]$, and (ii) $\mathsf{F}^{\mathtt{BWT}}[c \ldots d]$ must contain a ϕ character corresponding to v_i, since all node texts are prefixed by the

ϕ character in the construction of F. Therefore, by determining the rank of the first and the last ϕ characters in $\mathsf{F}^{\mathtt{BWT}}[a \dots b]$, we will determine a range of *forward ids* corresponding to nodes that contain $P[i \dots m]$ as a prefix. Using backward search we can determine the range of matching forward ids, if any, for each suffix of P in $O(m \log \sigma)$ time (by Lemma 1). To determine which nodes contain a prefix of P as a suffix, we can instead determine which nodes, when their text is reversed, contain a suffix of P^R, the reverse of P, as a prefix. Therefore, by performing a backward search of P^R in the reverse index R, a range of *reverse ids* can be determined in $O(m \log \sigma)$ time (by Lemma 1). Finally, we also determine the matching statistics of P and $P[2 \dots m-1]$ with respect to F in $O(m \log \sigma)$ time (by Lemma 2). The matching statistics for P are used to determine matches within nodes, while the matching statistics for $P[2 \dots m-1]$ are used to determine matches of nodes within $P[2 \dots m-1]$. The forward and reverse id ranges for every prefix and suffix of P as well as the matching statistics for P and $P[2 \dots m-1]$ can be stored in $O(m \log n)$ bits.

4.2 Matching within a Node

If a match of P in G does not span an edge, then P must match as a substring of some node. Let $(q, [a, b])$ be the matching statistics of $P[1 \dots m]$ calculated in the preprocessing step. If $q = m$ then there exists at least one match of P as a substring of a node of G. Furthermore, the suffix array range $[a, b]$ is non-empty and P is contained as a substring of one or more nodes exactly $b - a + 1$ times. In each instance, the forward id and offset into the node can be determined by the usual location reporting capabilities of a compressed suffix array.

Lemma 5. *For a pattern P of length m, the occ number of matches of P within a node of G can be counted in $O(m \log \sigma)$ time. Their locations can be reported in an additional $O(occ \log n)$ time. The working space is $O(m \log n)$ bits.*

4.3 Matching across a Single Edge

If a match of P in G spans a single edge, then there must exist some i, $1 < i \leq m$, such that $P[1 \dots i-1]$ is a suffix of some node $v_j \in V$, $P[i \dots m]$ is a prefix of some node $v_k \in V$, and the edge $(j, k) \in E$. In the preprocessing step, the range $[c, d]$ of forward ids corresponding to nodes that contain $P[i \dots m]$ as a prefix, as well as the range $[a, b]$ of reverse ids corresponding to nodes that contain $P[1 \dots i-1]$ as a suffix, were stored. Similar to other applications in stringology [18,17,5], we make use of the range query data structure to relate the two ranges. If both ranges are non-empty, we can determine exactly which pairs of ids are connected by a forward edge by reporting all points in Q contained in the range $[a, b] \times [c, d]$. If (x, y) is a reported point, then the forward id of x can be determined as $\Pi_{R \to F}[x]$. We can repeat this procedure $m - 1$ times, for $1 < i \leq m$.

Lemma 6. *For a pattern P of length m the occ number of matches of P that cross a single edge of G can be counted in $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|})$ time. Their path descriptions can be reported in an additional $O(occ \frac{\log |V|}{\log \log |V|})$ time. The working space is $O(m \log n)$ bits.*

4.4 Matching across Multiple Edges

If a match of P in G spans more than one edge, then $P[2 \dots m-1]$ must contain at least one node of G as a substring. The strategy here, as in previous solutions to the text with wildcard problem [11,17,18], is an extension of the dictionary matching problem: first identify all γ nodes contained within $P[2 \dots m-1]$, and second, for each of those γ candidate matches, determine if it can be extended into a full match of P in G. Consider a candidate match of a node v_j, with forward id j, to the substring $P[i \dots i+k-1]$, where $i > 1$ and $i+k-1 < m$. This candidate can be extended into a full match if both the *suffix condition*—there exists a path leaving v_j matching $P[i+k \dots m]$ —and the *prefix condition*—there exists a path ending at v_j matching $P[1 \dots i-1]$—are satisfied. This strategy is motivated by applications where few nodes share identical text (*e.g.,* transcriptomes). Under this assumption, γ is expected to be within a small constant of m. More specifically, in the case of transcriptomes where exons are generally long and unique, γ is expected to be a small constant.

Recently, Thachuk proposed a dynamic programming algorithm to solve the corresponding text with wildcards version of the problem [18]. The algorithm works in m stages, by considering successively longer suffixes of P, and in the process determines if the suffix and prefix conditions of candidate matches are satisfied. We will adapt this algorithm for our purposes here. However, we note that verifying the suffix (prefix) condition in the hypertext problem is more challenging as we must consider any path beginning (ending) at a node representing a candidate match. In the text with wildcards problem, one must only verify the text immediately preceding (succeeding) a candidate match position. This can be viewed as verifying a single path. For this reason, the algorithm we propose below is a generalization of the original algorithm.

Overview of the Algorithm. Conceptually, the algorithm will consider successively longer suffixes of $P[2 \dots m-1]$. Specifically, for each suffix $P[i \dots m-1]$, for $i = m-1, m-2, \dots, 2$, the algorithm will consider all γ_i nodes of G that prefix $P[i \dots m-1]$ using the full-text dictionary F. Each of these γ_i nodes is a *candidate* requiring the suffix condition to be first verified, and if successful, the prefix condition is tested. The algorithm will maintain a compact list of all sub-paths of G that are matched by $P[i \dots m]$. The head of the list for suffix i will be stored at $\mathsf{W}[i]$, a working space array maintained during the search. In later stages of the algorithm, we will determine if these sub-paths can be extended to match longer suffixes of P. Overall, there are γ candidate positions that will be evaluated. Note that the exact count of candidates, γ, can be determined using F, prior to the first stage of the algorithm in $O(m \log \sigma)$ time. This permits us to allocate sufficient working space. Our algorithm attempts to track all matching sub-paths in as little working space as possible. We describe the information tracked during the course of the algorithm, and comment at the end on the overall working space complexity. In what follows, we describe how the suffix and prefix conditions are verified for a candidate node v_j that matches a k length prefix of $P[i \dots m-1]$. This same procedure will be used to verify all γ candidates, across all suffixes of $P[2 \dots m-1]$.

Verifying the Suffix Condition. We must verify that there exists a sub-path in G matching $P[i+k \ldots m]$ that begins at some node v_t such that $(j,t) \in E$. There are two cases to consider: such a sub-path is a prefix of v_t (and thus ends within v_t), or it properly contains v_t. We will refer to the former as a *sub-path initiation event*, and the latter as a *sub-path extension event*. For each candidate node v_j we must consider both types.

To verify an initiation event, we first determine the range $[a,b]$ of forward ids corresponding to nodes that contain $P[i+k \ldots m]$ as a prefix. This range was stored in the preprocessing step. Then, we must determine if any of those nodes have an incoming edge from node v_j. Recall that node v_j has forward id j and therefore reverse id $j' = \Pi_{R \to F}[j]$. A counting query in the range $[j',j'] \times [a,b]$ of Q determines cnt_{init}, the number of matching nodes. Specifically, cnt_{init} is the number of sub-paths that originate at node v_j, match $P[i \ldots m]$, and end within a node connected to v_j.

An extension event implies that $P[i+k \ldots m]$ must match a sub-path that contains, but does not end within, a node connected to v_j. Therefore, to verify an extension event, we must determine if any putative sub-paths stored in the list at $\mathsf{W}[i+k]$ begin with a node v_t such that $(j,t) \in E$. (Note that if one or more of these sub-paths do exist, they would have been stored in $\mathsf{W}[i+k]$ at an earlier stage of the algorithm.) For each of the at most γ entries in $\mathsf{W}[i+k]$, we use the forward id of its initial node, and j', the reverse id of v_j, to perform a range query in Q in order to determine if they are adjacent. Note that since we only need to establish adjacency in the range query, then no assumption on the graph topology is made and therefore any directed graph (possibly cyclic) is handled correctly. Let cnt_{ext} be the number of entries in $\mathsf{W}[i+k]$ that are connected by an edge from v_j.

If the suffix condition is satisfied we append a new putative sub-path entry to the list at $\mathsf{W}[i]$ and associate with it v_j as the *initial node*. If $cnt_{\text{ext}} > 0$, then we associate with that entry a list of the cnt_{ext} offsets that denote the entries in $\mathsf{W}[i+k]$ which are connected to v_j and form new putative sub-path matches for $P[i \ldots m]$. We also associate with each entry the count of sub-paths it begins. Consider that each of the cnt_{ext} entries in $\mathsf{W}[i+k]$ connected to v_j may represent many sub-paths. The number of sub-paths each represents is stored in the entry. Therefore, the number of sub-paths represented by our new entry is the sum of the counts for these previous cnt_{ext} entries, plus cnt_{init}.

Verifying the Prefix Condition. If the suffix condition is satisfied for a candidate node v_j, we can test the prefix condition. We need to determine if there exists one or more nodes that contain $P[1 \ldots i-1]$ as a suffix and have an outgoing edge to node v_j. In the preprocessing step, we stored the range $[a,b]$ of *reverse id*s corresponding to nodes that contain $P[1 \ldots i-1]$ as a suffix. The cnt_p number of matching nodes can be found by querying the range $[a,b] \times [j,j]$ in Q. If $cnt_p > 0$, and the current count of sub-paths beginning at v_j that match $P[i \ldots m]$ is cnt_s, then G contains $cnt_p \times cnt_s$ paths matching P.

Reporting all Matching Paths. Whenever a prefix condition is verified, all matching paths can be enumerated by a simple backtracking procedure using the information previously stored in W and the new prefix matches. The point data structure Q is once again queried to determine the forward ids of nodes that contain the end of a matching path. The permutation $\Pi_{R \to F}$ is used to ensure all nodes are reported by forward id.

Lemma 7. *For a pattern P of length m, the occ number of matches of P that cross more than one edge in G, can be determined in $O(m \log \sigma + \gamma^2 \frac{\log |V|}{\log \log |V|})$ time. Their path descriptions can be reported in an additional $O(h + occ \frac{\log |V|}{\log \log |V|})$ time, where h is the total number of nodes in all occ sub-paths matched by P. The working space is $O(m \log n + \gamma^2 \log \gamma + \gamma \log |V|)$ bits.*

Proof. For each candidate node v_j, the suffix condition must be verified by checking for both initiation events and for extension events. When verifying initiation events, a range query on Q is performed in $O(\frac{\log |V|}{\log \log |V|})$ time. When verifying extension events for v_j, at most $O(\gamma)$ previous entries representing putative sub-paths are considered to be extended by v_j. For each putative sub-path, a range query is performed in $O(\frac{\log |V|}{\log \log |V|})$ time. Thus, verifying extension events for v_j takes $O(\gamma \frac{\log |V|}{\log \log |V|})$ time. The suffix condition can be verified for all γ candidate nodes in $O(\gamma^2 \frac{\log |V|}{\log \log |V|})$ time. Verifying the prefix condition is analogous to verifying suffix initiation events and takes $O(\gamma \frac{\log |V|}{\log \log |V|})$ time to verify the at most γ candidates that satisfy the suffix condition. This yields an overall worst case time $O(m \log \sigma + \gamma^2 \frac{\log |V|}{\log \log |V|})$.

The working space includes the $O(m \log n)$ bits from the preprocessing step, and $O(\gamma \log |V|)$ bits to store counts of putative sub-paths and initial nodes of those sub-paths. However, the working space is dominated by storing back pointers (offsets) in the putative sub-path entries of the working array W to each previous entry that forms a valid sub-path match for a suffix of P. There are at most $O(\gamma)$ entries in W. Thus, each can be uniquely identified with $\lceil \log \gamma \rceil$ bits. In the worst case, each entry stores $O(\gamma)$ back pointers giving an overall working space of $O(m \log n + \gamma \log |V| + \gamma^2 \log \gamma)$ bits. Importantly, we note that the total number of matches γ can be determined by F in a preprocessing step in $O(m \log \sigma)$ time in order to allocate sufficient working space. □

Combining Lemmas 4 through 7 we have our main result.

Theorem 1. *A hypertext $G = (V, E)$ can be represented in $(2 + o(1))n \log \sigma + O(n) + (1 + o(1))|E| \log |E| + |V|(\log |V| + \log \frac{n}{|V|} + 2 + o(1))$ bits, where the text of the nodes in V are over an alphabet of size σ and have a combined length of $n - |V|$, such that all matches of a pattern P of length m can be counted in $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|} + \gamma^2 \frac{\log |V|}{\log \log |V|})$ time, and reported in an additional $O(occ_1 \log n + (occ_2 + occ_3) \frac{\log |V|}{\log \log |V|} + h)$ time, where occ_1 is the number of matches within a node of G, occ_2 is the number of matches crossing a single edge, and h is the total number of nodes in all occ_3 sub-paths matching P that cross more than one edge. The working space is $O(m \log n + \gamma^2 \log \gamma + \gamma \log |V|)$ bits.*

5 Considering Path Constraints in Hypertext

While our motivating problem of aligning transcripts to transcriptomes is better modeled using a hypertext index than a linear text index, the model does not completely capture all necessary information. Specifically, the hypertext models the splicing graph; however, the set of valid transcripts in the transcriptome is a set $\mathcal{P}$ of paths through the

splicing graph. Not every path in the splicing graph is necessarily a valid transcript. We now show how we can easily extend our index to only report matches of a pattern P if they are a sub-path of at least one path in $\mathcal{P}$.

For illustrative purposes, assume we have constructed a hypertext index for the example in Figure 1 and all transcripts are valid, expect for t_3. Further suppose the node labels in the hypertext correspond to the exon labels in the figure. Then the set of valid paths are $\mathcal{P} = \{[e_1, e_2], [e_1, e_3], [e_4], [e_4, e_5]\}$. We construct a serialization of these paths as a string $S = \phi e_1 e_2 \phi e_1 e_3 \phi e_4 \phi e_4 e_5$, over the integer alphabet $[1 \ldots |V|] \cup \{\phi\}$, where $\phi = 0$. Next, we create a compressed suffix array S for S. We use the same matching algorithm as before, and for each candidate path $p = p_1, \ldots, p_k$ through G reported, we perform an additional verification step. Specifically, we see if the string '$p_1 \ldots p_k$' exists in S as a substring by performing a backward search query in S. Clearly, if the path description of p is a substring in S, then p is a sub-path of some valid path in $\mathcal{P}$.

Lemma 8. *A set of valid paths $\mathcal{P}$ can be represented in $(1 + o(1))h \log |V|$ bits, where h is the total number of nodes in all paths in $\mathcal{P}$, such that a candidate path p, crossing k nodes, can be verified in $O(k \log |V|)$ time.*

We note that the space required to store all valid paths (by node ids) of a transcriptome is still dominated by and nearly negligible compared to the space to index the node text.

6 Conclusions

We proposed a succinct index to model hypertext and focused on the case where the space required to index the text dominates the space to represent the topology of the graph. The index can model any hypertext and places no restriction on the graph topology. We proposed a new pattern matching algorithm, capable of aligning a pattern to any path in the hypertext, that is especially efficient for hypertexts where few nodes share the same text. In this case, which is motivated by the problem of aligning patterns to transcriptomes, the proposed algorithm is a significant improvement in runtime complexity compared with all existing algorithms. We also showed how matches in a hypertext can be restricted to a set of valid paths. Development of an efficient algorithm to report approximate matches in the hypertext index is an important future direction. Another interesting direction is to consider time and space trade-offs when using different graph topology representations [6,3].

Acknowledgments. The author is grateful to the anonymous reviewers who suggested a more space efficient means to store the graph topology and other useful ideas and also to Anne Condon for feedback on this manuscript.

References

1. Akutsu, T.: A Linear Time Pattern Matching Algorithm Between a String and a Tree. In: Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.) CPM 1993. LNCS, vol. 684, pp. 1–10. Springer, Heidelberg (1993)
2. Amir, A., Lewenstein, M., Lewenstein, N.: Pattern matching in hypertext. Journal of Algorithms 35(1), 82–99 (2000)

3. Barbay, J., Claude, F., Navarro, G.: Compact rich-functional binary relation representations. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 170–183. Springer, Heidelberg (2010)
4. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
5. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 235–246. Springer, Heidelberg (2009)
6. Farzan, A., Munro, J.: Succinct representations of arbitrary graphs. In: 16th Annual European Symposium on Algorithms, pp. 393–404 (2008)
7. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Symposium on Foundations of Computer Science, pp. 390–398 (2002)
8. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
9. Horner, D., Pavesi, G., Castrignano, T., De Meo, P., Liuni, S., Sammeth, M., Picardi, E., Pesole, G.: Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing. Briefings in Bioinformatics (2009)
10. Jay, S., Ji, H.: Next-generation DNA sequencing. Nature Biotechnology 26(10), 1135–1145 (2008)
11. Lam, T.-W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Space efficient indexes for string matching with don't cares. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 846–857. Springer, Heidelberg (2007)
12. Manber, U., Wu, S.: Approximate String Matching With Arbitrary Costs for Text and Hypertext. In: IAPR International Workshop on Structural and Syntactic Pattern Recognition, pp. 22–33 (1992)
13. Mortazavi, A., Williams, B., McCue, K., Schaeffer, L., Wold, B.: Mapping and quantifying mammalian transcriptomes by RNA-Seq. Nature Methods 5(7), 621–628 (2008)
14. Navarro, G.: Improved approximate pattern matching on hypertext. Theoretical Computer Science 237(1-2), 455–463 (2000)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2 (2007)
16. Park, K., Kim, D.: String Matching in Hypertext. In: Symposium on Combinatorial pattern matching, p. 318 (1995)
17. Tam, A., Wu, E., Lam, T.-W., Yiu, S.-M.: Succinct text indexing with wildcards. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 39–50. Springer, Heidelberg (2009)
18. Thachuk, C.: Succincter text indexing with wildcards. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 27–40. Springer, Heidelberg (2011)

When Was It Written? Automatically Determining Publication Dates

Anne Garcia-Fernandez[1,⋆], Anne-Laure Ligozat[1,2], Marco Dinarelli[1], and Delphine Bernhard[1]

[1] LIMSI-CNRS, Orsay, France
[2] ENSIIE, Evry, France
{annegf,annlor,marcod,bernhard}@limsi.fr

Abstract. Automatically determining the publication date of a document is a complex task, since a document may contain only few intra-textual hints about its publication date. Yet, it has many important applications. Indeed, the amount of digitized historical documents is constantly increasing, but their publication dates are not always properly identified via OCR acquisition. Accurate knowledge about publication dates is crucial for many applications, e.g. studying the evolution of documents topics over a certain period of time.

In this article, we present a method for automatically determining the publication dates of documents, which was evaluated on a French newspaper corpus in the context of the DEFT 2011 evaluation campaign. Our system is based on a combination of different individual systems, relying both on supervised and unsupervised learning, and uses several external resources, e.g. Wikipedia, Google Books Ngrams, and etymological background knowledge about the French language. Our system detects the correct year of publication in 10% of the cases for 300-word excerpts and in 14% of the cases for 500-word excerpts, which is very promising given the complexity of the task.

1 Introduction

Automatically determining the publication date of a document is a complex task, since a document may contain only few intra-textual hints about its publication date. This task has many important applications including temporal text-containment search [13] and management of digitized historical documents. Indeed, the amount of digitized historical documents is constantly increasing, but their publication dates are not always properly identified by automatic methods.

In this article, we present a novel method for automatically determining the publication dates of documents, which was evaluated on a French newspaper corpus in the context of the DEFT 2011[1] evaluation campaign [5]. Our approach combines a large variety of techniques, based on both a training corpus and

⋆ The author is now working at CEA-LIST, DIASI-LVIC lab at Fontenay-Aux-Roses, France.

[1] http://deft2011.limsi.fr/

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 221–236, 2011.
© Springer-Verlag Berlin Heidelberg 2011

external resources, as well as supervised and unsupervised methods. The main contributions of the paper are as follows:

– We use the Google Books Ngrams, which were made recently available by Google, in order to automatically identify neologisms and archaisms.
– We build classification models on a corpus covering a large range of historical documents and publication dates.
– We apply Natural Language Processing techniques on challenging OCRized data.
– We study and evaluate different independent systems for determining publication dates, as well as several combination techniques.

In the next section, we discuss the state of the art. In section 3 we detail the training and evaluation corpora as well as the evaluation methodology. In section 4 we describe corpus independent approaches, which we call "chronological methods", while in section 5 we describe supervised classification methods. Combination techniques for aggregating the individual systems are detailed in section 6. Finally, we evaluate the systems in section 7 and conclude in section 8 providing some perspectives for future work.

2 State of the Art

Though there is an extensive literature on text categorization tasks, research on temporal classification is scarce. Existing approaches are based on the intuition that, for a given document, it is possible to find its publication date by selecting the time partition whose term usage has the largest overlap with the document. The models thus assign a probability to a document according to word statistics over time.

De Jong et al. [3] aim at linking contemporary search terms to their historical equivalents and at dating texts, in order to improve the retrieval of historical texts. They propose building independent language models for documents and time partitions (with varying granularities for model and output), using unigram models only. Then the divergence between the models of a partition and a tested document is measured by a normalized log-likelihood ratio with smoothing. Due to the lack of huge digitized reference corpora, the experiments are performed on contemporary content only, consisting of articles from Dutch newspapers, with a time span ranging from 1999 to 2005. The models based on documents outperform those based on time partitions.

Kanhabua and Nørvåg [8] reuse the previous model, but incorporate several preprocessing techniques: part-of-speech tagging, collocation extraction (e.g. "United States"), word sense disambiguation, concept extraction and word filtering (tf-idf weighting and selection of top-ranked terms). They also propose three methods for improving the similarity between models: word interpolation (smoothing of frequencies to compensate for the limited size of corpora), temporal entropy (to measure how well a term is suited for separating a document from other documents in a document collection) and external search statistics

from Google Zeitgeist (trends of search terms). They created a corpus of about 9,000 English web pages, mostly web versions of newspapers, covering on average 8 years for each source. The techniques were evaluated for time granularities ranging from one week to one year. The preprocessing techniques improved the results obtained by de Jong et al. [3]. This work lead to the creation of a tool for determining the timestamp of a non-timestamped document [9].

The DEFT 2010 challenge proposed a task whose goal was to identify the decade of publication of a newspaper excerpt [6]. The corpus was composed of articles from five French newspapers, automatically digitized with OCR (Optical Character Recognition) and covering a time range of a century and a half. The best performing system [1] obtained an f-measure of 0.338 using spelling reforms, birth dates, and learning of the vocabulary. The second best system [15] used orthographic correction, named entity recognition, correction with Google Suggest, date search on Wikipedia, and language models.

3 Methodology

3.1 Corpus Description

The dataset used for training and evaluating our system was provided in the context of the DEFT 2011 challenge.

The corpora were collected from seven French newspapers available in Gallica:[2] *La Croix, Le Figaro, Le Journal de l'Empire, Le Journal des Débats, Le Journal des Débats politiques et littéraires,* and *Le Temps* plus an unknown newspaper present only in the evaluation data set. The corpus is composed of article excerpts, called *portions*, containing either 300 or 500 words and published between 1801 and 1944. The excerpts with 300 or 500 words were obtained without taking the structure of the source article into account so that the last sentence of each excerpt can be incomplete. Moreover dates present in the excerpts were removed, in order to circumvent the bias of dates available within the document itself.

Table 1 summarizes general statistics about the corpora.[3] The training corpus provided by DEFT contains 3,596 newspaper portions. We divided this corpus in two parts: an actual training set (TRN) and a development set (DEV). The evaluation corpus (EVAL) was unavailable at the time of system development and contains 2,445 portions.

The corpora were automatically digitized with OCR. Figure 1 shows an example of digitized text in which erroneous words are underlined, while Figure 2 shows the original corresponding document.

Different kinds of errors can be identified, such as erroneous uppercasing, additional and/or missing letters, punctuation, or space, sequence of one or several erroneous letters... There are also archaic forms of words, such as "fragmens". We

[2] http://gallica.bnf.fr/

[3] The number of portions per year is 24 for each year except for 1815: 21 portions were proposed in the training set and 17 in the evaluation set.

Table 1. General description of training and test corpora

	Training data				Evaluation data	
	300 words		500 words		300 words	500 words
	TRN	DEV	TRN	DEV	EVAL	EVAL
# portions	2396	1200	2396	1200	2445	2445
# words	718,800	360,000	1,198,000	600,000	733,500	1,222,500
# different words	73,921	48,195	107,617	67,012	78,662	110,749
# different newspapers	6	6	6	6	7	7
Mean # portions per year	16	8	16	8	14	14

La séance musicale de M. Félicien David au Palais de l'Industrie a obtenu un succès complet_les fragmens du Désert, de Christophe Colomb et de Moïse au Sinaï ont été très vivemçnt applaudis; le Chant du soir a été redemandé par une acclamation unanime. Jeudi 22, le même programme sera de nouveau exécuté dans les mêmes conditions: l,2S0 choristes et'instrumentistes. Samedi 24, seconde exécution du concert dirigé par M. Berlioz. Dimanche 2S, fermeture de la nef centrale du Palais de l'Industrie et clôture des fêtes musicales. Lotecfètairedela rédaction, F. Carani.

Fig. 1. Digitized text from a 1855 document

— La séance musicale de M. Félicien David au Palais de l'Industrie a obtenu un succès complet : les fragmens du *Désert*, de *Christophe Colomb* et de *Moïse au Sinaï* ont été très vivemȩnt applaudis ; le *Chant du soir* a été redemandé par une acclamation unanime. Jeudi 22, le même programme sera de nouveau exécuté dans les mêmes conditions : 1,250 choristes et instrumentistes. Samedi 24, seconde exécution du concert dirigé par M. Berlioz. Dimanche 25, fermeture de la nef centrale du Palais de l'Industrie et clôture des fêtes musicales.
Le secrétaire de la rédaction, F. Camus.

Fig. 2. Excerpt from a 1855 document

estimated the number of out of vocabulary (OOV) words using a contemporary spell checker: *hunspell*.[4] There are between 0 and 125 OOV words in 300-word portions and a mean of 22 OOV words per portion. We observed that there is no clear correlation between the publication year of an excerpt and the number of OOV words, i.e., the quality of the OCR document.

This kind of text is especially challenging for NLP tools, since traditional techniques such as part-of-speech tagging or named entity recognition are likely to have much lower performance on these texts.

3.2 Corpus Pre-processing

The corpus was preprocessed by the TreeTagger [17] for French, and words were replaced by their lemmas. The goal was to reduce the vocabulary, to improve the similarity between documents. For the portions of the TRN corpus for example, the vocabulary thus dropped from 74,000 to 52,000 different words.

3.3 Evaluation Score

The evaluation measures that we use for our final system are the percentages of correct decades and years given by our systems. Yet the aim is to be as close as

[4] Open source spell checker: `http://hunspell.sourceforge.net/`

possible to the reference year so we also use an evaluation metric which takes into account the distance between the predicted year and the reference year, which is the official DEFT 2011 evaluation score [5]. Given a text portion a_i whose publication year in the reference is $d_r(a_i)$, a system gives an estimated publication date $d_p(a_i)$. The system then receives a score S which depends on how close the predicted year is to the reference year. This similarity score is based on a gaussian function and is averaged on the N test portions. The precise formula is given by equation 1.

$$S = \frac{1}{N} \sum_{i=1}^{N} e^{-\frac{\pi}{10^2}(d_p(a_i)-d_r(a_i))^2} \quad (1)$$

This score is thus a variant of the fraction of correctly predicted years, where wrong predictions at a certain distance from the correct answer are given less points than correct answers, instead of no point as do more traditional measures. For example, the score is of 1.0 if the predicted year is correct, of 0.97 if off by one year, of 0.5 if off by 4.7 years, and falls to 0 if it is off by more by 15 years.

3.4 Description of the Methods

We used two types of methods. Chronological methods (see section 4) yield the periods of time which are most plausible for each portion, but without ranking the corresponding years. In the above example (Figure 1), several cues give indications on the publication date of the document: several persons are mentioned ("M. Félicien David" and "M. Berlioz" for example), which means that the publication date is (at least) posterior to their birthdates; moreover, the spelling of the word "fragmens" is an archaism, since it would now be written "fragments", which means that the text was written before the spelling reform modifying this word; finally, the exhibition hall "Palais de l'Industrie" was built in 1855 and destroyed in 1897, so the document date must be posterior to 1855, and is likely to be anterior to 1897 (as word statistics over time such as Google Books Ngrams can show). These are the kinds of information used by chronological methods to reduce the possible time span. These methods make use of external resources, and are thus not dependent on the corpora used.

Classification methods (see section 5) make use of the training corpora to calculate temporal similarities between each portion and a reference corpus.

4 Chronological Methods

4.1 Named Entities

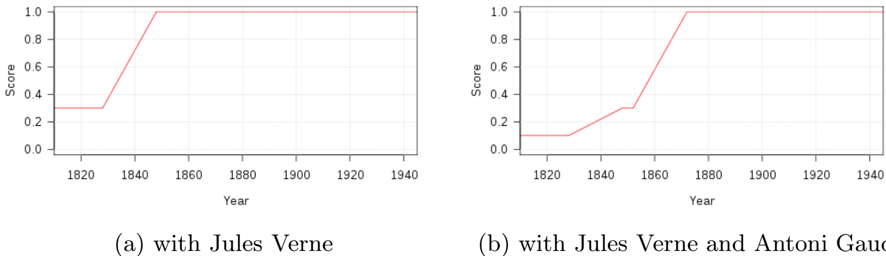
The presence of a person's name in a text portion is an interesting clue for determining its date, since the date of the document must be posterior to the birthyear of this person.

We used the following strategy: we automatically gathered the birthyears of persons born between 1781 and 1944 by using Wikipedia's "Naissance_en_*AAAA*"

categories.[5] About 99,000 person names were thus retrieved, out of which we selected 96,000 unambiguous ones (for example two "Albert Kahn" were found), since we have no simple way of knowing which particular instance is mentioned in the texts.

For each text portion, we extracted occurrences of person names using WMatch,[6] which allows for fast text annotation [4,16]. For the TRN corpus, 529 names were detected in 375 portions (out of 2,359 portions), out of which 16 (3%) were actually namesakes or false detections (for example, Wikipedia has an entry for the French novelist "Colette", whose name is also a common first name).

A score was then given to each candidate year for a given portion, according to the person mentions found in that portion. We considered that before the person birthyear Y_b, the probability of a year $y < Y_b$ being the correct answer is low (here 0.3), then for a year y between the birthyear and 20 years after[7] ($Y_b \leq y \leq Y_b + 20$), the probability raises linearly reaching 1.0 (see Figure 3a).



(a) with Jules Verne (b) with Jules Verne and Antoni Gaudí

Fig. 3. Scoring function given person mentions

For a given text portion p, the score for each year is the product of the score for each person mention found in p. Figure 3b shows the score obtained in the presence of two person mentions, Jules Verne, born in 1828 and Antoni Gaudí, born in 1852.

4.2 Neologisms and Archaisms

Neologisms correspond to newly created words, while archaisms refer to words which cease being used at some time. Both neologisms and archaisms constitute interesting cues for identifying publication dates: given the approximate year of apparition of a word, one can assign a low probability for all preceding years and a high probability to following years (the reverse line of argument can be applied to archaisms). However, there is no pre-compiled list of words with their year

[5] Category:$YYYY$_birth.

[6] Rule-based automatic annotation tool, available upon request.

[7] Intuitively, a person that is less than 20 years old will not be cited in a newspaper and, in the absence of a more appropriate model, we considered that then s/he has a equal probability to be cited all over his/her life.

of appearance or disappearance. This type of information is sometimes included in dictionaries, but depends on the availability of these resources. We therefore developed a method to automatically extract neologisms and archaisms from Google Books unigrams for French [10].

Automatic Acquisition of Neologisms and Archaisms. Automatically determining the date of appearance and disappearance of a word is not a trivial task. In particular, metadata associated with Google Books are not always precise [14]. It is therefore not possible to use a simple criterion such as extracting the first year when the occurrence count of a word exceeds 1 to identify neologisms. We developed instead a method relying on the cumulatice frequency distribution, i.e., for each year, the number of occurrences of the word since the beginning of the considered time span divided by the total number of occurrences:

1. Get the word's count distribution for years ranging from 1700 to 2008;[8]
2. Smooth the distribution with a flat smoothing window[9] of size 3;
3. Get the word's cumulative frequency distribution and determine the appearance/disappearance date as the first year where the cumulative frequency exceeds a given threshold.

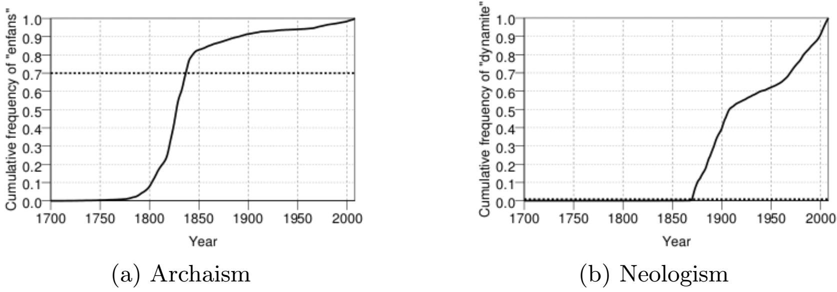
We defined the best cumulative frequency thresholds by using manually selected development sets consisting of 32 neologisms (e.g. "photographie" – *photography*, "télévision" – *television*) and 21 archaisms (old spellings which are no longer in use, see Section 4.3). This number of neologisms and archaisms was sufficient to find reliable thresholds. The obtained thresholds were 0.008 for neologisms and 0.7 for archaisms. Moreover, we only kept neologisms with a mean occurrence count of at least 10 and archaisms with a mean occurrence of at least 5 over the considered year range. Overall, we were able to extract 114,396 neologisms and 53,392 archaisms with appearance/disappearance year information.

Figure 4 displays two cumulative frequency curves: one for an archaism (the old spelling of the word "enfants", *children*), and the other for a neologism ("dynamite", invented in 1867). The thresholds correspond to the horizontal dotted lines. The curves have very different profiles: archaisms are characterised by a logistic curve, which reaches a plateau well before the end of the considered year range. On the other hand, neologisms correspond to an increasing curve.

We calculated the error rate on the DEV corpus: for 90% of the archaisms found in the corpus, the date of the portion is anterior to the disappearance date, and for 97% of them, it is anterior to the disappearance date plus 20 years. For the neologisms, the date of the portion is posterior to the appearance date for 97% of them, and to the appearance date minus 20 years for 99.8% of them. This 20-years *"shift"* (20 years giving the most accurate and precise results on the training corpus) is taken into account in the scoring formula.

[8] The first available year in Google Books ngrams is actually 1536. However, given the year-range of our task, we considered that 1700 was an adequate lower threshold.

[9] As defined in `http://www.scipy.org/Cookbook/SignalSmooth`



(a) Archaism (b) Neologism

Fig. 4. Cumulative frequency distributions

Scoring with Neologisms and Archaisms. The automatically extracted lists of neologisms and archaisms are used to assign a score for each year, given a text portion. For neologisms, years following the appearance date are given a high score, while preceding years are assigned a lower score. The following formula is used for neologisms. p corresponds to text portion, w is a word, y a year in the considered year range 1801-1944 and year(w) is the date of appearance extracted for a neologism.

$$score_{neo}(p, y) = \frac{\sum_{w \in p} score\text{-}neo(w,y)}{|p|} \text{ where: } score\text{-}neo(w, y) =$$

$$\begin{cases} 1.0 & \text{if } w \notin \text{neologisms} \\ 1.0 & \text{if } w \in \text{neologisms and } y \geq year(w) \\ 0.2 & \text{if } w \in \text{neologisms and } (year(w) - y) > 20 \\ 0.2 + 0.04 \cdot (20 + y - year(w)) & \text{otherwise} \end{cases}$$

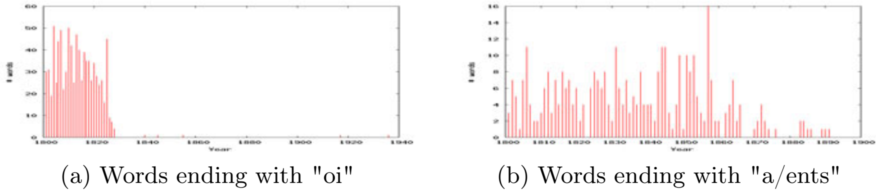
An equivalent formula is used for archaisms, by considering that years following the disappearance of a word have a low score.

4.3 French Spelling Reforms

During the 1801-1944 period, French spelling underwent two major reforms: one in 1835 and another in 1878. The main change induced by the first reform is that conjugated verbs ending with “oi” changed to “ai”: e.g. the inflected form “avois” of the verb “avoir” (*to have*), was changed into “avais”. The second reform mostly concerned names ending with “ant” or “ent”, whose plural changed to “ants”/“ents” instead of “ans”/“ens” (for example “enfans” was changed into “enfants”–*children*).

Figure 5 displays the distribution of each type of words (“oi” and “a/ents”) in the training corpus for each year. The first type of words is present mostly before 1828, and the second type only before 1891, which roughly correspond to the reform dates.

Scoring with Spelling Reforms. Following Albert et al. [1], we use this information as a clue to determine the date of a text. We assign a score for each



(a) Words ending with "oi" (b) Words ending with "a/ents"

Fig. 5. Distributions of pre-reforms words in the TRN corpus

year to each text portion. In order to determine old spellings in use before the reforms, we use the following method:

- Get unknown words with *hunspell* (with the French DELA as a dictionary [2]);
- If the word ends with "ois/oit/oient", replace "o" with "a";
  If the new word is in the dictionary, increment the counter n_{28}, which corresponds to the number of old word spellings in use before the first reform;
- Else, if the word ends with "ans/ens", insert "t" before "s";
  If the new word is in the dictionary, increment the counter n_{91}, which corresponds to the number of old word spellings in use before the second reform.

Then, a function was used to determine a score for each year y and a portion p based on the counters n_{28} and n_{91}, according to the following formulas (where r in f_r can be either 28 or 91):

$score_{spell}(p, y) = score_{28}(p, y) \cdot score_{91}(p, y)$ with:

$$score_r(p, y) = \begin{cases} f_r(y) & if\ y > r \\ 1 & if\ y \leq r \end{cases}, \quad f_{28}(y) = \begin{cases} 1 & if\ n_{28} = 0 \\ 0.15 & if\ n_{28} = 1 \\ 0 & if\ n_{28} > 1 \end{cases} \ and \ f_{91}(y) = \begin{cases} 1 & if\ n_{91} = 0 \\ 0 & if\ n_{91} > 0 \end{cases}$$

For example, if $n_{28} = 1$ and $n_{91} = 1$ for a text portion, the score for years before 1828 is 1.0, for years between 1828 and 1891, the score is 0.15, which corresponds to the error rate for using this criterion on our training corpus, and for years after 1891, the score is 0 since the presence of an old spelling in use before the second reform is a very strong indication that the text was written before 1891.

4.4 Intermediate Conclusion

As we have shown in the previous section, chronological methods yield very accurate indications for a text's time span (with a maximum error rate of 3%). However, they only discriminate between large time periods, and are not precise enough for identifying the publication date (e.g. if a portion contains a person's name whose birthyear is 1852, we can only say the portion has not been published before 1852). Thus, we also used corpus-based classification methods: a cosine

similarity relying on a feature vector representation of text portions and using the standard $tf \cdot idf$ feature weighting; and a machine learning approach based on SVMs. These approaches are described in next sections.

5 Classification Methods

Temporal similarity methods calculate similarities between each portion and a reference corpus.

5.1 Cosine Similarity-Based Classification

Using the Training Corpus. The training corpus provides examples of texts for each year in the 1801-1944 year range. These texts can be used as a reference to obtain word statistics over time. We grouped all portions for the same year in the TRN corpus and used these portion groups as references for the corresponding years. For classification, the similarity is computed between a group of portions in the same year and the portion to be classified. Each group and each portion were converted into feature vectors using the $tf \cdot idf$ score as feature weighting. Given an n-gram i and a portion (or group of portions) j:

$$tf \cdot idf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \cdot \log \frac{|Y|}{|\{y_j : w_i \in y_j + smoothing\}|}$$

where $n_{i,j}$ is the number of occurrences of n-gram w_i in portion (or group) j, $|Y|$ is the number of years in the training corpus, y_j is the group of text portions for year j; $smoothing = 0.5$ is applied to take into account words in the test corpus which were not found in the training corpus.

For a text portion in the test corpus, we computed the similarities between the portion and each group representing a year with a standard cosine similarity. Experiments were made for word n-grams with n ranging from 1 to 5; yet, for n>2, the small size of the training corpus leads to sparse data. For word n-grams, we used the lemmatized version of the corpora since it gave better results in preliminar experiments.

As the corpus is composed of OCRized documents, there are many errors in the texts, which poses many problems for tf.idf scoring: the tfs and dfs are smaller than what would be expected for "real" words since errors impede the identification of some occurrences, and some erroneous words have higher idfs than would be expected. In order to cope with this difficulty, we also computed the similarity using character n-grams (following [12] for information retrieval on an OCRized corpus). Thus, for example for the text "sympathie1" which contains a "real" word "sympathie" and an OCR error "1", character n-grams (for n<9) will match all n-grams of the word "sympathie", despite the OCR error. Then, portions were indexed as before, and a cosine similarity was also applied to match each portion with the best corresponding year.

Using Google Books Ngrams. The training corpus is rather small, and we therefore also experimented with using Google Books Ngrams as training data. Due to the huge amount of data in Google Books Ngrams, we only used the n-grams with alphanumeric content and with more than 10 occurrences in a given year. The resulting data was used instead of our training corpora. The tf.idf formula is slightly modified for the training corpus, since $n_{i,j}$ is the number of occurrences of n-gram w_i for year j and y_j is the Google Ngram data for year j.

5.2 Support Vector Machines (SVM)

SVMs are well-known machine learning algorithms belonging to the class of maximal margin linear classifiers [18]. For our experiments with SVM we used `svm-light`[10] [7]. Two kernel functions have been tested for our task: `polynomial kernel` and `radial basis function`, both available in the `svm-light` package. Given the small amount of data available for each year (25 portions for each year, except for 1815 which has 21 portions), the one-VS-all training approach was used: a model was created for each year against all other years. The SVM system consists of 144 binary models, one corresponding to each year, from 1801 to 1944. In each model, positive instances are those extracted from portions belonging to the target year to be detected, negative instances are all the others. Each model is able to distinguish portions belonging to the corresponding year. At classification time, each portion is evaluated with all 144 models and the one providing the highest score is chosen as the correct answer.

SVM Settings and Tuning. SVM parameters as well as feature sets were tuned on the TRN and DEV sets. Neither all parameters, nor all features types were optimized. A full optimization of all parameters and features requires a huge number of experiments. Instead, based also on our experience, in some cases we used default or a-priori parameters. The SVM parameter C for soft margin (see [18]) was set to 1. In most of the tasks the best value is between 1 and 10, 1 gives always fair results. The *cost-factor* parameter, affecting the weighting of errors made on positive and negative instances, was set to the ratio between the number of negative and positive instances, as suggested in [11]. Concerning kernel functions, the `polynomial kernel` was more effective than the `radial basis function` on the DEV set and it was kept for following system tuning. Default values for polynomial kernel parameters were used (1 for c and 3 for polynomial degree d).

Concerning the feature set, we tried several sets for preliminary studies, and for further experiments we kept only the most promising in terms of performance on DEV. We first experimented with some configurations typical of text categorization tasks. For example we removed stop-words and we replaced words by their lemmas (in inflectional languages like French, they provide roughly the same information as stems). Surprisingly this led to a degradation of performances. In contrast, using both words and lemmas and keeping stop-words,

[10] Available at `http://svmlight.joachims.org/`

gave better results than those obtained using only words. This configuration was chosen as baseline SVM system. Further experiments were performed to tune the size of word n-grams to be used in feature vectors. We tried to use n-grams of size from 1 up to 4. 2-grams gave best results.

Using this configuration we integrated the information provided by systems described in section 4: birth dates of persons, neologisms and archaisms, French spelling reforms. In particular each of these systems provides information that could be encoded in SVM feature vectors as *feature:year*, where *feature* is a person name in case of birth dates, a neologism or archaism word or a word that has been reformed in one of the two French spelling reforms. Given the sparsity of feature vectors representation, feature values in the baseline system are always much smaller than any of the *year* provided by any of the chronological methods. This has been a problem for learning the SVM models. The problem still holds when shifting year values from the range 1801..1944 to the range 1..144. Indeed we experienced training problems or performance degradation when using such a representation. In order to overcome this problem we split the information provided by chronological methods in two parts, corresponding to two sets of binary features (the value is 0 if the feature is absent, 1 if present): one for the information alone, e.g. *NEOLOGISM_ <WORD>* or *REFORMED_ <WORD>* for neologisms or reformed words,[11] respectively; another for the year the information appears in, e.g. *NEOLOGISM-YEAR_ <YEAR>* or *REFORMED-YEAR_ <YEAR>*. This representation always led to performance improvements.

Since in preliminar studies experiments on 500-word portions reflected the behavior of 300-word portions, we did not carry out all experiments also on 500-word portions. Instead we applied directly the best configuration found for 300-word portions.

6 Scoring Combination

Given the differences in characteristics of individual systems described in previous sections, we made a combination of the score provided by each individual system with the aim of improving the final result. The methods do not have the same overall performance nevertheless they all provide useful information: for instance, archaisms indicate an upper limit for the publication date. For the combination of scores, we experimented with two different strategies: simple multiplication and linear regression of scores provided by individual systems.

Multiplication of Scores. This combination consists in multiplying the scores provided by the different methods, for each portion and for each year:

$$score_{multiplication}(p,y) = \prod_k score_k(p,y)$$

where $score_k(p,y)$ is the score of the system k labelling portion p as being published in year y.

[11] <WORD> is a place holder for any word belonging to the specified category.

Linear Regression on Scores. In this case, scores from different systems are not multiplied but summed according to the following formula:

$$score_{regression}(p, y) = \sum_k \alpha_k \cdot score_k(p, y) + \varepsilon$$

with α_k the coefficient for the system k, $score_k(p, y)$ the score given by the system k to the portion p for the year y and ε the error term.

Coefficients were fitted on the training corpus using the R function `lm()`. The linear regression process finds the best model (ie. α values) to predict a numerical value from clues (system scores in our case). In our case, the numerical value to be predicted depends on the distance *dist* between a year and the true year of publication of the portion : the value is $1.0 - dist/143$.

In the development phase, we fitted the α and ε values on the TRN corpus and tested the combination on the DEV corpus. As the cosine and SVM systems need to be trained, we did not include the score of those systems in our regression model. We thus computed a *regression score* based on scores from neologism, archaism, birth dates of person, and spelling reforms information. The scores of the cosine and SVM systems were multiplied by this regression score. For the test phase, we fitted the values on the entire training data set.

7 Results

We evaluate our approach using the measures described in section 3.3. We first present the results of the cosine and SVM approaches and then the results of the two scoring combination methods described in section 6. The systems used for the evaluation data have been trained on the entire training data (TRN + DEV).

7.1 Results for Classification Methods

Cosine Similarity. The results of the cosine similarity system are presented in table 2 (only the best scoring settings are given). With the training corpus, characters 5-grams have the best results on both portion sizes, which was expected since the documents are quite noisy. Word unigrams are better on 300-word portions than bigrams. Yet bigrams perform better on 500-word portions, which tends to show that they benefit from an increased amount of data.

Table 2. Results obtained for the cosine based methods

	Training corpus				**Google Ngrams**			
	DEV		**EVAL**		**DEV**		**EVAL**	
	300 w.	**500 w.**	**300 w.**	**500 w.**	**300 w.**	**500 w.**	**300 w.**	**500 w.**
word 1-grams	0.260	0.299	0.267	0.321	0.210	0.221	0.200	0.216
word 2-grams	0.209	0.319	0.263	0.327	0.238	0.295	0.241	0.264
char 5-grams	**0.287**	**0.327**	**0.311**	**0.363**	-	-	-	-

For the cosine method based on Google Ngrams, the corpus used was not lemmatized, since Google Ngrams contain inflected words. The best results were obtained with bigrams. Results are lower than those using only the training corpus which was not expected because Google Ngrams is a much larger data set. This could be due to the different nature of documents: our corpus is composed only of newspaper excerpts. Moreover the publication dates in Google Books are not completely reliable [14].

SVM System. Results obtained with the system based on SVM are reported in tables 3 and 4. As can be seen from table 3, incrementally adding features encoding the information provided by chronological methods leads to consistent performance improvements. In table 4 we detail all the results obtained with the best system on 300 and 500-word portions.

Table 3. Additive results of the SVM system with different features on the DEV corpus for 300 words

Baseline (word 2-grams + lemmas)	0.228
+**neologisms**	0.234
+**spelling reforms**	0.242
+**birth dates**	0.243

Table 4. Results of SVM system

DEV		EVAL	
300 words	**500 words**	**300 words**	**500 words**
0.243	0.293	0.272	0.330

	DEV		EVAL	
	300 w.	**500 w.**	**300 w.**	**500 w.**
mult.	0.343	0.401	0.378	0.452
regress.	0.356	0.390	0.374	0.428

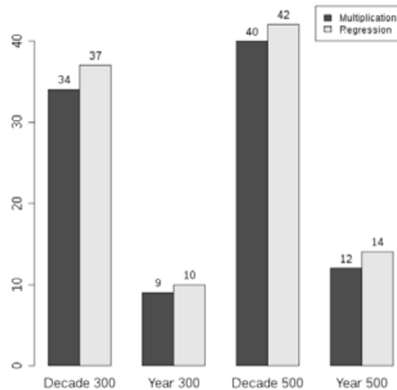


Fig. 6. Scores and correct decades/years obtained with fusion

Scoring Fusion. Figure 6 displays the results obtained on the training and evaluation data sets for the various system combinations. Scoring fusions consistently improve the scores of individual systems. Results on 500-word portions are much higher than results on 300-word portions. For the evaluation data, fusion by multiplication performs better than fusion using linear regression.

Figure 6 shows results in terms of correct decades and years at the first rank. 35% of first rank decades are the correct ones for 300-word portions and 40% for 500-word portions. For years, the fusion using linear regression detects the correct year for respectively 10% and 14% of the 300 and 500-word text portions. Those results are much higher than the random selection of a decade or a year in the time span (7% for decades and 0.7% for years). For decades, using the DEFT 2010 evaluation metric, our results are also higher than results obtained by the best participants to the DEFT 2010 challenge [6].

8 Conclusions and Future Work

In this article, we present a system for automatically dating historical documents. It is based on several methods, both supervised and unsupervised, and takes advantage of different external resources, such as Google Ngrams or knowledge about spelling reforms. We obtain 14% of correct years and 42% of correct decades in our best-performing setting.

The results show that this is a challenging task for several reasons: the documents may not contain many intra-textual hints about their publication dates, digitized historical documents can be of a low quality, the vocabulary is different from the vocabulary currently in use, and external resources are not always completely reliable.

These experiments made it possible to observe the quality of digitized documents, and to adapt the NLP techniques we used to this specific condition, for example by considering characters n-grams instead of word n-grams. In order to improve the quality of documents, we plan to use OCR correction. We would also like to investigate the application of named entity recognition, including event detection. Finally, we plan to work on different corpora in order to test the robustness of our methods, and to perform experiments with whole documents without date anonymisation instead of text portions.

References

1. Albert, P., Badin, F., Delorme, M., Devos, N., Papazoglou, S., Simard, J.: Décennie d'un article de journal par analyse statistique et lexicale. In: DEFT 2010, TALN (2010)
2. Blandine, C., Silberzstein, M.: Dictionnaires électroniques du français. Langue française 87 (1990)
3. De Jong, F., Rode, H., Hiemstra, D.: Temporal language models for the disclosure of historical text. In: Humanities, Computers and Cultural Heritage, p. 161 (2005)
4. Galibert, O.: Approches et méthodologies pour la réponse automatique à des questions adaptées à un cadre interactif en domaine ouvert. Ph.D. thesis, Université Paris-Sud 11, Orsay, France (2009)
5. Grouin, C., Forest, D., Paroubek, P., Zweigenbaum, P.: Présentation et résultats du défi fouille de texte DEFT2011. In: Actes TALN (2011)
6. Grouin, C., Forest, D., Sylva, L.D., Paroubek, P., Zweigenbaum, P.: Présentation et résultats du défi fouille de texte DEFT 2010: Oú et quand un article de presse a-t-il été écrit? In: Actes TALN (2010)

7. Joachims, T.: Making large-scale SVM learning practical. In: Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge (1999)
8. Kanhabua, N., Nørvåg, K.: Improving temporal language models for determining time of non-timestamped documents. In: Research and Advanced Technology for Digital Libraries, pp. 358–370 (2008)
9. Kanhabua, N., Nørvåg, K.: Using temporal language models for document dating. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS, vol. 5782, pp. 738–741. Springer, Heidelberg (2009)
10. Michel, J.B., Shen, Y.K., Aiden, A.P., Veres, A., Gray, M.K., The Google Books Team, Pickett, J.P., Hoiberg, D., Clancy, D., Norvig, P., Orwant, J., Pinker, S., Nowak, M.A., Aiden, E.L.: Quantitative Analysis of Culture Using Millions of Digitized Books. Science 331(6014), 176–182 (2011)
11. Morik, K., Brockhausen, P., Joachims, T.: Combining statistical learning with a knowledge-based approach - a case study in intensive care monitoring. In: Proceedings of ICML 1999, pp. 268–277. Morgan Kaufmann Publishers Inc., San Francisco (1999)
12. Naji, N., Savoy, J., Dolamic, L.: Recherche d'information dans un corpus bruité (OCR). In: CORIA (2011)
13. Nørvåg, K.: Supporting temporal text-containment queries in temporal document databases. Data & Knowledge Engineering 49(1), 105–125 (2004)
14. Nunberg, G.: Google's Book Search: A Disaster for Scholars. The Chronicle of Higher Education (August 2009) (Online, accessed April 13, 2011)
15. Oger, S., Rouvier, M., Camelin, N., Kessler, R., Lefèvre, F., Torres-Moreno, J.: Système du LIA pour la campagne DEFT 2010: datation et localisation d'articles de presse francophones. In: DEFT 2010, TALN (2010)
16. Rosset, S., Galibert, O., Bernard, G., Bilinski, E., Adda, G.: The LIMSI participation to the QAst track. In: Working Notes of CLEF 2008 Workshop, Aarhus, Danemark (2008)
17. Schmid, H.: Probabilistic part-of-speech tagging using decision trees. In: International Conference on New Methods in Language Processing, pp. 44–49 (1994)
18. Vapnik, V.N.: Statistical Learning Theory. John Wiley and Sons, Chichester (1998)

A New Approach for Verifying URL Uniqueness in Web Crawlers

Wallace Favoreto Henrique[1], Nivio Ziviani[1], Marco Antônio Cristo[2], Edleno Silva de Moura[2], Altigran Soares da Silva[2], and Cristiano Carvalho[1]

[1] Universidade Federal de Minas Gerais,
Department of Computer Science, Belo Horizonte, Brazil
{wallace,nivio,cristiano.dcc}@dcc.ufmg.br
[2] Universidade Federal do Amazonas,
Department of Computer Science, Manaus, Brazil
{marco.cristo,edleno,alti}@dcc.ufam.edu.br

Abstract. The Web has become a huge repository of pages and search engines allow users to find relevant information in this repository. Web crawlers are an important component of search engines. They find, download, parse content and store pages in a repository. In this paper, we present a new algorithm for verifying URL uniqueness in a large-scale web crawler. The verifier of uniqueness must check if a URL is present in the repository of unique URLs and if the corresponding page was already collected. The algorithm is based on a novel policy for organizing the set of unique URLs according to the server they belong to, exploiting a locality of reference property. This property is inherent in Web traversals, which follows from the skewed distribution of links within a web page, thus favoring references to other pages from a same server. We select the URLs to be crawled taking into account information about the servers they belong to, thus allowing the usage of our algorithm in the crawler without extra cost to pre-organize the entries. We compare our algorithm with a state-of-the-art algorithm found in the literature. We present a model for both algorithms and compare their performances. We carried out experiments using a crawling simulation of a representative subset of the Web which show that the adopted policy yields to a significant improvement in the time spent handling URL uniqueness verification.

1 Introduction

In July 2008, Google has reported more than 1 trillion unique URLs in its scheduler queue[1]. To get to this number of unique URLs, the crawler starts at a set of initial pages and follow each of their links to new pages. Then it follows the links on those new pages to more pages, in a continuous fashion. In fact, they found more than 1 trillion individual URLs, but not all of them lead to unique web pages, as many pages are duplicates or auto-generated copies of each other.

[1] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 237–248, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Search engines consist of web crawlers (which find, download, parse content and store each page), indexers (which construct an inverted file index for fast retrieval of pages), and query processors (which rank documents by relevance to answer user queries). Search engines download and index only a subset of the pages related to the set of unique URLs.

This paper focuses on the problem of verifying URL uniqueness in a web crawler. The verifier of uniqueness must check if a URL is present in the repository of unique URLs and if the corresponding page was already collected. For a repository of URLs stored in a disk file, URLs can be checked against a buffer of popular URLs and only those not present are searched in the disk file. In [2] and [4], the RAM buffer is a LRU cache with an array of recently added URLs, in [5], a general purpose database is used with a RAM caching, and in [6], a balanced tree of URLs is used for disk checking.

A better idea is to accumulate URLs into a RAM buffer and check several URLs sequentially in one pass. This *batch disk check* strategy is used by DRUM (*Disk Repository with Update Management*), which is part of the IRLBot web crawler [3]. DRUM can store large volumes of arbitrary hashed data on disk and implement very fast check, update, and check+update operations using bucket sort. Through a set of comparison experiments it is shown that DRUM outperforms all the previously cited ones. Thus, in this work, we will use DRUM as our comparison baseline to evaluate our proposed strategy. While DRUM can be applied to other tasks such as handling "robots.txt" files and DNS caching, we here focus on its use for verifying URL uniqueness.

For a large number of pages, the task of verifying URL uniqueness becomes very complex. As shown in [3], the complexity of verifying URL uniqueness is quadratic on the number of links that must pass through URL check. In the *batch disk check* strategy large sets of URLs are overwritten in lexicographic order in disk files, which is a time consuming operation.

In this paper, we present VEUNIQ (VErifier of UNIQueness). VEUNIQ is based on a novel policy for organizing the set of known URLs according to the server they belong. Following such a policy, the algorithm exploits a locality of reference property, which is inherent in Web traversals. This property follows from the skewed distribution of links within a web page, which favors references to other pages from a same server. Previous work [1,7] have observed and exploited this property for ranking purposes, and, in here, we take advantage of such property to speed up the crawling process.

VEUNIQ is part of a larger web crawling project developed by us. Our scheduling strategy in this crawler also selects URLs taking into account information about the servers they belong to, thus allowing the usage of VEUNIQ in the crawler without extra cost to pre-organize the entries.

We present a model for both algorithms, DRUM and VEUNIQ, and compare their performance. We carried out experiments using a crawling simulation of a representative subset of the Web which show that the adopted policy yields to an improvement in the time spent handling URL uniqueness verification in comparison to the baseline algorithm.

2 Related Work

Few works in literature have addressed the problem of ensuring that a certain page will not be collected more than once during a crawling cycle. A very first approach to deal with this problem was proposed by Pinkerton [5]. He used the B-Tree structure of a database management system (DBMS) to verify the uniqueness of each page URL, thus avoiding repeated occurrences in the list of URLs to be collected. This strategy is simple to implement since it takes advantage of the large availability of commercial DBMS systems. However, the use of a DBMS to store and retrieve a massive number of URLs leads to poor disk performance with large impact over the whole crawling process.

To cope with this problem, Heydon and Najork [2] proposed the use of a cache in memory to minimize random disk access operations. Their algorithm, called Mercator-A, would only perform disk accesses if the current URL being checked was not found in the cache. In such a case, to minimize the probability of new misses, a new set of URLs would be retrieved from the disk instead of only the missing one. The disadvantage of Mercator-A lies in the fact that, in the worst case, it requires a disk access for each URL and that access operation comprises much more data to be transferred.

A better strategy to minimize random disk-accesses operations is to check several URLs sequentially in one pass, a strategy called *batch disk check*. The idea is to accumulate URLs into a memory buffer. When this buffer is full, the URLs are sorted in place. They are then merged with blocks of (already sorted) URLs retrieved from the disk, so that duplicates are not stored. This is the main idea behind the approaches referred to as Mercator-B [4] and Polybot [6]. Polybot avoids the sorting step because it uses a binary tree as memory buffer.

A more recent algorithm, called DRUM (*Disk Repository with Update Management*), was proposed for the IRLBot web crawler [3]. As this algorithm was used as the baseline for an experimental comparison with VEUNIQ, a detailed description of DRUM is provided in Section 4.

3 Crawler Architecture

In this section, we present a high-level description of our crawler. It has four main components: fetcher, extractor of URLs, verifier of uniqueness and scheduler. Figure 1 illustrates the crawl cycle involving the four components. The fetcher is the component that sees the Web. In step 1, the fetcher receives from the scheduler a set of URLs to download web pages. In step 2, the extractor of URLs parses each downloaded page and obtains a set of new URLs. In step 3, the uniqueness verifier checks each URL against the repository of unique URLs. In step 4, the scheduler chooses a new set of URLs to be sent to the fetcher, thus finishing one crawl cycle.

Considering cycle i, the *fetcher* locates and downloads web pages. It receives from the *scheduler* a set of candidate URLs to be crawled and returns a set L_i of URLs actually downloaded. The set of candidate URLs is small, determined

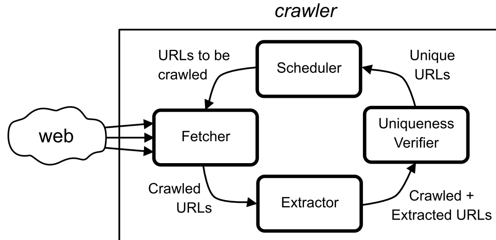


Fig. 1. Web page crawling cycle

by the amount of memory space available to VEUNIQ. The downloaded pages are stored on disk for efficient access and retrieval.

Important to notice that there are many different policies in the literature to select the set of candidates to be crawled from a given set of servers at each cycle. Also important is that the interval between two accesses to the same server must follow politeness rules, which might cause a significant slowdown in the whole process. In this paper, we will not address these issues but rather concentrate on the algorithm for verifying URL uniqueness.

The *extractor of URLs* parses each downloaded page and output two sets: the set E_i of URLs just extracted and the set M_i with auxiliary metadata for each page such as the repository disk file address and its offset.

The *verifier of uniqueness* receives as input the three sets L_i, E_i and M_i and creates as output an updated repository of unique URLs. It identifies from L_i and E_i the following sets: (i) URLs crawled in previous cycles; (ii) URLs seen in previous cycles but not crawled yet; (iii) URLs crawled in the current cycle; (iv) New URLs. The URLs from sets (i) and (ii) are discarded (they are already in the set of unique URLs), the set M_i and the information on disk location for URLs from set (iii) are updated, and new URLs from set (iv) are merged to the repository of unique URLs.

4 The Baseline Algorithm (DRUM)

The verifier of uniqueness must check if a URL is present in the repository of unique URLs. Before presenting our approach VEUNIQ for checking URL uniqueness we describe DRUM, which we use as a comparison baseline approach.

DRUM (Disk Repository with Update Management) is a technique for efficient storage of key/value pairs (KV-pairs) with asynchronous support to three operations: *check* (a key is checked against a disk repository and, if found, its corresponding value can be retrieved), *update* (KV-pairs are merged into the disk repository), and *check+update* (the two previous operations combined). While DRUM is a more general solution to external hashing, as far as we know it is also the best solution published in literature, to verify uniqueness of URLs in a web crawler. Further, as it can be seen in the article where DRUM was proposed, verifying uniqueness in web crawlers is its main application focus.

DRUM keeps the set of unique URLs in a large persistent repository, which is sorted by means of a bucket sort strategy, described as follows. Input received by DRUM is inserted into two sets of memory arrays, according to their key values. The KV-pairs are stored in the first set while auxiliaries are sent to the second one. For each array, two corresponding buckets are kept in disk, one for KV-pairs (KV-bucket) and another for auxiliaries (A-bucket). Once an array is filled, their values are moved to the corresponding disk buckets. The arrays continue to receive KV-pairs until one of the KV-buckets reaches a certain size r. When this happens, all the KV-buckets are merged with the central repository of URLs. To accomplish this, each KV-bucket is loaded into a memory buffer and sorted. The buffer content is then merged with the central repository. After the pairs have been processed, the buffer is restored to its original order so that KV-pairs match those in the corresponding A-bucket. Key, value and auxiliary data are then dispatched for further processing. All these steps are repeated for all buckets sequentially.

Note that sorting is only done when a KV-bucket is read into memory, which allows an efficient synchronization of all buckets with the central repository in a single pass. Further, to ensure fast sequential writing/reading operations, all buckets are pre-allocated on disk before they are used.

DRUM Model

As shown by Lee et al. [3], after some time, a crawler reaches a steady state where the probability p of a page to be unique remains constant. Assume that in such state: (i) the crawler is scheduled to visit $\mathcal{N}$ pages, each one with an average of ℓ links. Thus, $n = \mathcal{N}\ell$ pages have to be checked to verify their uniqueness; (ii) a RAM of $\mathcal{R}$ bytes is allocated to this task which, as previously described, will require the merge of n input URLs with U unique URLs stored in a central disk repository; (iii) the average URL length is b, H is the size of the URL hashes used by the crawler, and P is the size of a memory pointer.

As previously mentioned, DRUM starts a merge each time one of the KV-buckets reaches a size r. If we assume that the hashing distributes the URLs evenly into k buckets, when one of them reaches size r, a total of kr disk bytes was filled with URL hashes. Thus, DRUM performs $\frac{nH}{kr}$ merges to check n URLs. To fill the KV-buckets, in each iteration i, it is necessary:

– Read/write the i-th A-bucket once to load into memory the URL text of $\frac{kr}{H}$ URLs, for a total amount of $2\frac{krb}{H}$ bytes.
– Read/write all the KV-buckets, i.e., $2kr$ bytes.
– Read/write URLs from the central repository, i.e., $2UH + (i-1)2krp$ bytes.
– Append new URL hashes, i.e., krp bytes.

Thus, after adding the final overhead to store pbn bytes of unique URLs, the read/write DRUM overhead to check n URLs is given by (see Eq.(15) from [3]):

$$w(n,\mathcal{R}) = pbn + \sum_{i=1}^{\frac{nH}{kr}} \left(2UH + \frac{2krb}{H} + 2kr + 2krpi - krp\right)$$
$$= nb\left(\frac{(2UH + pHn)H}{bkr} + 2 + p + \frac{2H}{b}\right) \quad (1)$$

Now assume that to support efficient read and write operations, a buffer of size $\mathcal{M}$ is maintained to each opened file, a buffer of size Δ is used to load the repository into memory, and $r \geq \Delta$. Thus, if $\mathcal{R} \geq \frac{2\Delta(H+P)}{H}$ and DRUM can use up to D bytes of disk for the checking, final overhead is $w(n,\mathcal{R}) = \alpha(n,\mathcal{R})bn$, where $\alpha(n,\mathcal{R})$ is a proportion of the input size, given by:

$$\alpha(n,\mathcal{R}) = \begin{cases} \frac{8M(H+P)(2UH+pHn)}{b\mathcal{R}^2} + 2 + p + \frac{2H}{b} & \mathcal{R}^2 < 8\mathcal{M}D\frac{H+P}{H+b} \\ \frac{(H+b)(2UH+pHn)}{bD} + 2 + p + \frac{2H}{b} & \mathcal{R}^2 \geq 8\mathcal{M}D\frac{H+P}{H+b} \end{cases} \quad (2)$$

Finally, to avoid unnecessary allocation of disk space, $D = \frac{\mathcal{R}^2(H+b)}{8\mathcal{M}(H+P)}$ and $.k = \frac{\mathcal{R}}{4\mathcal{M}}$. We refer the interested reader to [3] for a more detailed explanation of this model. Note that DRUM strategy requires that a large set of U URLs be overwritten several times to check n input URLs. This is a time consuming operation with large impact on the overall performance.

VEUNIQ takes advantage of the site order the scheduler is able to provide to maximize URL locality. This allows the overwriting of the disk repository using much less disk accesses, as we will show in the next section. The parameters used in both DRUM and VEUNIQ models are summarized in Table 1.

Table 1. Summary of parameters used in DRUM and VEUNIQ models

Variable	Meaning	Unit
b	URL average size	bytes
β	Probability of a link point to a page not in the current repository	-
D	Disk size available for buckets in DRUM	bytes
Δ	Memory buffer size for loading repository of unique URLs	bytes
H	URL hash size	bytes
k	# of URL buckets in DRUM	-
ℓ	Average number of links per page	-
$\mathcal{M}$	Memory buffer size for each opened file in DRUM	bytes
n	# of links requiring URL checking	-
N	# of repositories in VEUNIQ	-
p	Probability of URL uniqueness	-
P	Memory pointer size	bytes
r	Size of buffer used to load URLs to be checked	bytes
$\mathcal{R}$	RAM memory allocated to uniqueness checking	bytes
R_i	Disk repository of unique URLs maintained by VEUNIQ	-
U	# of URLs in the set of unique URLs stored on disk	-

5 Algorithm VEUNIQ

In VEUNIQ, the set of unique URLs is stored in N persistent repositories R_i ($0 \leq i < N$). Each repository R_i contains URLs from a set of servers. The servers are distributed according to a hashing strategy whereas the URLs are sorted lexicographically within each server. VEUNIQ benefits from the locality of reference provided by the skewed distribution of links within each web page, which tends to reference other pages in the same server. For instance, in a sample of 400 servers crawled from the Web, the total number of links extracted was 29,580, with 18,712 (63.3%) links pointing to one of the 400 servers.

The crawler visits and updates the URLs of the repositories using a round robin strategy, so that after processing repository i, the next repository to process is $(i+1) \mod N$. At each cycle, the scheduler loads from current disk repository R_i the seed set of unique URLs. From this set, it derives the set of candidate URLs to be crawled and send it to the fetcher, which returns the set L_i. From $|L_i|$ downloaded pages, the extractor parses new URLs (E_i). These URLs ($L_i \cup E_i$) are then delivered to VEUNIQ. Note that $|L_i|$ is the number of URLs crawled in a cycle. Considering that we have about 10 links per page, we thus should set $|L_i|$ to roughly 1/11 of the number of URLs that we want VEUNIQ to verify in each cycle.

Algorithm 1 describes the usage of VEUNIQ in each crawling cycle. First, each URL found in a cycle is inserted into its corresponding buffer (lines 1 to 8). In line 4, the URLs that correspond to the current repository i are stored in buffer P_M^i and URLs corresponding to other repositories (which are expected to be the minority of them, being about 20% in our experiments) are inserted into their corresponding buffer TMP_M^j, $0 \leq j < N$ and $j \neq i$. Note that the hash function $h'(u)$ assigns each URL a number that corresponds to its appropriate repository. Further, as it takes only the URL server into account, all URLs of a same server are assigned to the same repository. After inserting all URLs in memory, the URLs related to repositories other than R_i are stored in their corresponding temporary disk files $TMP_D^{h \neq i}$ (line 10) and the memory used by them is deallocated.

Further, VEUNIQ recovers from the disk the URLs in TMP_D^i, which should also be stored in R_i and were found in previous cycles that processed other repositories than R_i. URLs from TMP_D^i are loaded and inserted in P_M^i (line 12), obtaining as a result a buffer P_M^i containing all the URLs related to repository R_i available at that moment. We then sort P_M^i in lexicographical order (line 13) and merge its content to the corresponding repository R_i on disk. After this final step, a new crawling cycle starts.

VEUNIQ Model

As in Section 4, we assume that the crawler has reached a steady state and has to check n URLs. VEUNIQ needs $\mathcal{R}$ bytes for buffer P_M^i, which achieves its maximum number of elements in line 4 of the VEUNIQ algorithm. In fact, a

Algorithm 1. Using Algorithm VEUNIQ to verify the uniqueness of URLs found in a crawling cycle and considering that the current repository is R_i

Input: Set of URLS $L_i \cup E_i$ crawled in a cycle
Output: Disk repository R_i updated
1: **for all** u in $L_i \cup E_i$ **do**
2: Let $\mathcal{H}$ be the value obtained by applying hash function $h'(u)$
3: **if** $\mathcal{H} = i$ **then**
4: Insert u into P_M^i
5: **else**
6: Insert u into $TMP_M^{\mathcal{H}}$
7: **end if**
8: **end for**
9: **for all** $0 \leq j < N$ AND $i \neq j$ **do**
10: Move contents from TMP_M^j to TMP_D^j and delete the memory buffer TMP_M^j
11: **end for**
12: Load TMP_D^i, inserting its content into P_M^i
13: Sort P_M^i in lexicographical order
14: R_i = Merge of P_M^i and R_i (disk merge)

few constant-size read/write buffers are used to minimize seek. The size of these buffers is proportional to the size of a disk block. Since such sizes are small, they will not be considered here.

An interesting property of VEUNIQ is that it does not require much memory to achieve good performance. The only restriction is that the number of URLs in each cycle should be large enough to allow the time required for sequential disk access performed on each cycle to be higher than the time required for disk seek operations related to the change of repositories among cycles. VEUNIQ does not require much memory due to its usage of locality properties to reduce the merge costs each time the memory is filled. Notice that when using DRUM, whenever the memory is filled, it is necessary to perform a merge with the whole set of pages in the repository, an operation that makes the memory requirement a bottleneck.

In VEUNIQ, the number of URLs crawled in a cycle is chosen based on restrictions related to the scheduling process, since the algorithm can be adjusted to use a large range of RAM memory. For instance, in our crawler, a typical value is 1 million of URLs crawled per cycle, but we run experiments in machines that would allow more than 15 million URLs in memory. When VEUNIQ uses less RAM, one can increase the number of repositories, so that each merge operation in a cycle have its cost reduced. We consider this smaller dependency of RAM as one of the most important properties of VEUNIQ.

Unlike DRUM, the U unique URLs stored on disk by VEUNIQ are divided into N repositories. Thus, VEUNIQ performs a complete merge after N steps. When considering a model where disk seeks are not taken into account, we could say that VEUNIQ runs faster as we increase N. However, in practice, when we increase N two practical problems arise. First, the disk seek operations become a significant portion of the total run time (note that this also happens when

we use a large number of buckets in DRUM). Second, when N increases, the number of servers in each repository might become too small to allow effective scheduling techniques. While we do not discuss scheduling here, we remember that it is important to the crawling process.

Disk seeks play an important role in DRUM and VEUNIQ computational costs. However, disk seek costs heavily depend on the adopted hardware architecture, with many possible scenarios. As the model presented in [3] does not include disk seeks, we also decided do not include disk seeks in our cost model. This decision has no impact on the comparison with the DRUM model. Moreover, we compare VEUNIQ with DRUM using their optimal parameters, and disk seeks would not affect VEUNIQ even if we had taken disk seeks into consideration in both models. Finally, as in [3], we also assume that the URLs requiring uniqueness checking are made available to VEUNIQ as a stream in memory. Thus, its loading into memory is not modeled.

Consider that VEUNIQ is configured to take the maximum amount of memory ($\mathcal{R}$) available. Consider that the *c-th* VEUNIQ cycle processes the *i-th* repository. Then, the costs related to this run for the $\frac{n}{N}$ URLs of one repository is:

- Write temporary buffers TMP_M^j, $0 \leq j < N$, $j \neq i$ to the corresponding TMP_D^j (Algorithm 1, line 10). In each cycle, this corresponds to $\beta\mathcal{R}$, being β the maximum number of links external to current repository R_i.
- Read bucket TMP_D^i into memory (Algorithm 1, line 12). This also corresponds to $\beta\mathcal{R}$ bytes.
- Read/write the current repository R_i and update it (Algorithm 1, line 14). Note that the current repository grows along the c cycles at a rate $p\mathcal{R}$, where p is the uniqueness probability. Thus, the merge cost to update R_i is $\left[p\mathcal{R} + 2\left(\frac{U}{N}b + (c-1)p\mathcal{R}\right)\right]$ bytes.

The total number of cycles depends on the number of bytes required to store each URL in memory, which is about $b+P$. Thus, the number of URLs that can be processed at each cycle is $\frac{\mathcal{R}}{b+P}$, and the total number of cycles necessary to process $\frac{n}{N}$ URLs is $\frac{n}{N}\frac{b+P}{\mathcal{R}}$.

Considering the number of cycles and the cost per cycle, the read/write VEUNIQ overhead to check the n URLs of the N repositories is given by:

$$\begin{aligned} w(n,\mathcal{R}) &= \sum_{i=0}^{N-1} \sum_{c=1}^{\frac{n}{N}\frac{b+P}{\mathcal{R}}} \left[p\mathcal{R} + 2\left(\frac{U}{N}b + (c-1)p\mathcal{R}\right) + 2\beta\mathcal{R}\right] \\ &= nb\left[\frac{(b+P)(bnp + npP + 2bU + 2N\mathcal{R}\beta)}{bN\mathcal{R}}\right] \end{aligned} \tag{3}$$

6 Comparison between Methods

We now compare the performance of VEUNIQ and DRUM using their computational cost model and a crawling simulation.

6.1 Computational Cost Model

Table 2 shows the overhead of the two methods as RAM size $\mathcal{R}$, disk size D, and number of URLs n increase. The overhead unit is the number of times ($\alpha(n, \mathcal{R})$) that bn bytes are written to/read from disk. The performance was calculated considering $p = 1/9$, $U = 1$ billion URLs, $b = 110$ bytes, $\ell = 59$ links per page, $P = 4$ bytes, $H = 8$ bytes, and $M = 256{,}000$ bytes. In the case of DRUM, optimum recommended values were used for D and k. Note that, despite we did not model disk seek cost, in DRUM, such a cost is proportional to the number of disk bucket files. Unlike DRUM, VEUNIQ seek cost is less sensible to the growth of its set of repositories, given by N, because most of the URLs $(1 - \beta)$ will be sent to the same repository (the current one). Thus, in this analysis, the number of repositories N starts equal k and grows proportionally to $ln(n)$. In this way, we ensure that very few repository reorganizations are carried out as n grows. As we can see, VEUNIQ outperforms DRUM for all sets of parameters.

Table 2. Overhead $\alpha(n, \mathcal{R}) = \frac{w(n,\mathcal{R})}{nb}$ calculated for DRUM and VEUNIQ

		$\mathcal{R} = 2$ Gb, $D = 22.1$ Tb, $k = 2{,}097$		$\mathcal{R} = 4$ Gb, $D = 88.6$ Tb, $k = 4{,}194$		$\mathcal{R} = 8$ Gb, $D = 354.2$ Tb, $k = 8{,}388$		$\mathcal{R} = 16$ Gb, $D = 1{,}417.1$ Tb, $k = 16{,}777$	
n	N	Drum	Veuniq	Drum	Veuniq	Drum	Veuniq	Drum	Veuniq
300 Mb	2,097	2.257	0.673	2.257	0.635	2.257	0.625	2.257	0.623
3 Gb	2,097	2.257	0.681	2.257	0.637	2.257	0.626	2.257	0.623
30 Gb	6,926	2.258	0.664	2.257	0.632	2.257	0.624	2.257	0.622
300 Gb	38,822	2.269	0.672	2.260	0.634	2.257	0.625	2.257	0.623
3 Tb	306,991	2.386	0.682	2.289	0.636	2.265	0.626	2.259	0.623

6.2 A Crawling Simulation

The computaional cost models for DRUM and VEUNIQ do not consider seek times and other computational costs, such as CPU costs. These points may raise questions about the usefulness of the models for comparison purposes. We thus decided to also perform a practical experiment to give more insight to the reader. However, due to space restrictions, we do not present detailed practical comparison experiments, focusing our comparison in the model, which is more hardware free, and letting detailed experiments for future work.

To evaluate the performance of the algorithms we proposed a crawling simulation. In such a simulation, VEUNIQ and DRUM take as input in each crawl cycle the following: (a) a set of crawled URLs; (b) a set of metadata collected from the pages corresponding to these URLs, and (c) a set of URLs extracted from collected pages.

For this experiment, a simulated crawling is preferable to actually carrying out a crawling over the Web, thus focusing only on the URL uniqueness verifier, isolating its analysis from the other components of the crawler (i.e., fetcher, scheduler and URL extractor). Furthermore, simulation yields more control over the experiments, enforcing well defined limits and easing its reproduction.

For this simulation we use the ClueWeb09 web page collection[2], containing approximately 1 billion Web pages and occupies approximately 25 terabytes. We simulate the crawling of 350 million URLs, divided into 35 cycles that handle 1 million URLs each. In the simulation, 100,000 of these URLs are collected and 900,000 are extracted from the collected pages. For each cycle we measure the time it takes to update the data structures used by VEUNIQ and DRUM.

The results of this experiment are shown in Table 3. We use a public C++ implementation of DRUM for URL uniqueness verification[3]. This implementation performs the final merge of the repository by storing the URLs in a database using the BerkeleyDB DBMS. However, to ensure fairness, the times reported disregard the time spent accessing the data stored in the database. In the experiment, we also set the value of N to be equal to k, which gives DRUM an advantage in the experiments.

Figure 2 illustrates the time spent in each crawl cycle. Notice the improvements achieved by VEUNIQ over DRUM. For instance, with 350 million URLs stored, the baseline requires 82.85 seconds to enter the URLs of the current crawl cycle, while VEUNIQ spent just 9.26 seconds to accomplish the same task. Finally, we observed that the total time spent by VEUNIQ in one cycle to crawl 100,000 URLs was approximately 267.3 seconds, being 240 seconds by the fetcher, 12 seconds by the extractor, 9.3 by the uniqueness verification and 6 seconds by the scheduler. Taking out the time spent by the fetcher, which depends mainly on the bandwidth available, the percentual of the total time (27.3 seconds) spent by the uniqueness verification is approximately 34% (44% for the extractor and 22% for the scheduler).

Table 3. Summary of the time required to update the respective data structures – DRUM vs. VEUNIQ

Millions of URL		1	50	100	150	200	250	300	350
Time (s)	DRUM	16.28	26.39	35.62	44.70	53.99	64.46	73.18	82.85
	VEUNIQ	1.38	2.78	3.62	4.50	5.47	7.20	7.44	9.26

7 Conclusions

In this paper, we presented VEUNIQ (VErifier of UNIQueness), a new algorithm for verifying URL uniqueness in a web crawler. VEUNIQ uses the idea of accumulating URLs into a RAM buffer and check several URLs sequentially in one pass. This batch disk check strategy is also used by DRUM (Disk Repository with Update Management), which is part of the IRLBot web crawler [3]. The algorithm to verify URL uniqueness in DRUM was used as our comparison baseline to evaluate our proposed strategy.

VEUNIQ is a based on a novel policy for organizing the set of unique URLs according to the server they belong to, exploiting a locality of reference property

[2] ClueWeb09 Dataset, `http://www.lemurproject.org/clueweb09.php`

[3] `http://www.codeproject.com/KB/recipes/cppdrumimplementation.aspx`

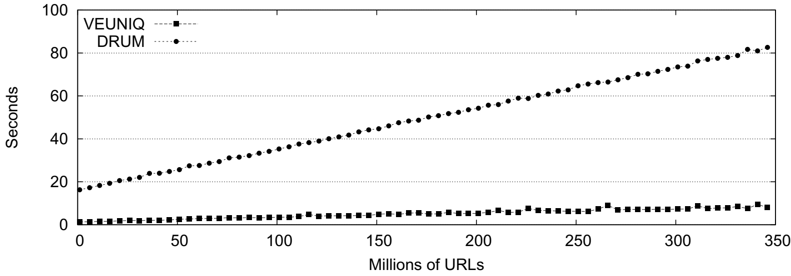


Fig. 2. Time required to update the respective data structures – DRUM vs. VEUNIQ

inherent in Web traversals. We presented a model for both DRUM and VEUNIQ and compare their performances. We also carried out experiments which show that the adopted policy yields to an improvement in the crawling rate in comparison to the baseline algorithm.

Acknowledgments. This work was partially sponsored by the Brazilian National Institute of Science and Technology for the Web (grant MCT/CNPq 573871/2008-6) and authors' individual grants and scholarships from CNPq.

References

1. Berlt, K., Moura, E., Carvalho, A., Cristo, M., Ziviani, N., Couto, T.: Modeling the web as a hypergraph to compute page reputation. Information Systems 35(5), 530–543 (2010)
2. Heydon, A., Najork, M.: Mercator: A scalable, extensible web crawler. World Wide Web 2(4), 219–229 (1999)
3. Lee, H.-T., Leonard, D., Wang, X., Loguinov, D.: Irlbot: Scaling to 6 billion pages and beyond. ACM Transactions on the Web 3(3), 1–34 (2009)
4. Najork, M., Heydon, A.: High-performance web crawling. Technical report, SRC Research Report 173, Compaq Systems Research, Palo Alto, CA (2001)
5. Pinkerton, B.: Finding what people want: Experiences with the web crawler. In: WWW, pp. 30–40 (1994)
6. Shkapenyuk, V., Suel, T.: Design and implementation of a high-performance distributed web crawler. In: ICDE, pp. 357–368 (2002)
7. Xue, G.-R., Yang, Q., Zeng, H.-J., Yu, Y., Chen, Z.: Exploiting the hierarchical structure for link analysis. In: SIGIR, pp. 186–193 (2005)

External Query Reformulation for Text-Based Image Retrieval

Jinming Min and Gareth J.F. Jones

Centre for Next Generation Localisation
School of Computing, Dublin City University
Dublin 9, Ireland
{jmin,gjones}@computing.dcu.ie

Abstract. In text-based image retrieval, the Incomplete Annotation Problem (IAP) can greatly degrade retrieval effectiveness. A standard method used to address this problem is pseudo relevance feedback (PRF) which updates user queries by adding feedback terms selected automatically from top ranked documents in a prior retrieval run. PRF assumes that the target collection provides enough feedback information to select effective expansion terms. This is often not the case in image retrieval since images often only have short metadata annotations leading to the IAP. Our work proposes the use of an external knowledge resource (Wikipedia) in the process of refining user queries. In our method, Wikipedia documents strongly related to the terms in user query ("definition documents") are first identified by title matching between the query and titles of Wikipedia articles. These definition documents are used as indicators to re-weight the feedback documents from an initial search run on a Wikipedia abstract collection using the Jaccard coefficient. The new weights of the feedback documents are combined with the scores rated by different indicators. Query-expansion terms are then selected based on these new weights for the feedback documents. Our method is evaluated on the ImageCLEF WikipediaMM image retrieval task using text-based retrieval on the document metadata fields. The results show significant improvement compared to standard PRF methods.

1 Introduction

The volume of online images has been expanding at an increasing rate in recent years. Searching for interesting and useful images from among the enormous number of images available generally relies either on content-based image retrieval using visual image features or text based search using text queries to search for images based on textual annotations of the images. Often it is difficult to find a sample image to use as a query image for visual search, and thus the text-based method is often the most commonly used by search engine vendors such as *Google*, *Bing* and *Yahoo!*. Where high quality detailed annotations are available, the text-based method can be very effective. However, annotations

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 249–260, 2011.
© Springer-Verlag Berlin Heidelberg 2011

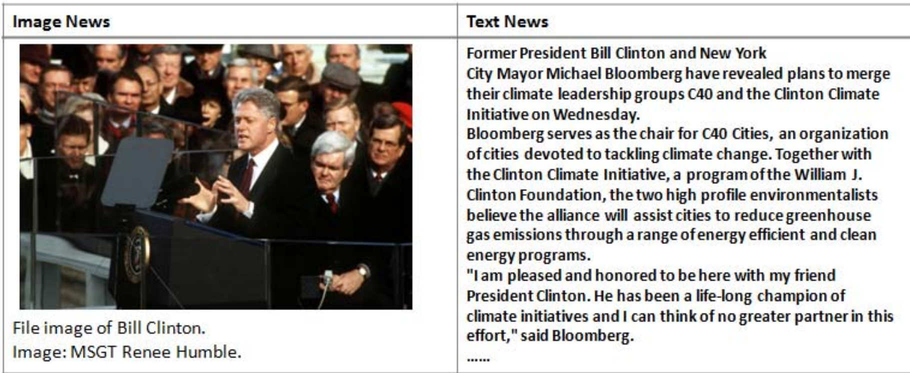


Image News	Text News
File image of Bill Clinton. Image: MSGT Renee Humble.	Former President Bill Clinton and New York City Mayor Michael Bloomberg have revealed plans to merge their climate leadership groups C40 and the Clinton Climate Initiative on Wednesday. Bloomberg serves as the chair for C40 Cities, an organization of cities devoted to tackling climate change. Together with the Clinton Climate Initiative, a program of the William J. Clinton Foundation, the two high profile environmentalists believe the alliance will assist cities to reduce greenhouse gas emissions through a range of energy efficient and clean energy programs. "I am pleased and honored to be here with my friend President Clinton. He has been a life-long champion of climate initiatives and I can think of no greater partner in this effort," said Bloomberg. ......

Fig. 1. Incomplete annotation problem example

are unfortunately often found to be noisy or incomplete, e.g. Picasa[1], Flickr[2]. The annotations are generally provided by those contributing the images who often only provide very brief or sometimes inaccurate details. These issues of poor image annotation can greatly affect image retrieval effectiveness based on textual metadata. Without complete textual description of an image, it is difficult to reliably match the image with text queries, since relevant images may not contain useful annotation terms. Thus is is not possible for the retrieval system to return the relevant images with high accuracy. We refer to this effect as the *incomplete annotation problem (IAP)* in image retrieval. An example of this problem is shown on the left of Figure 1 which is an image example from Wikinews [3]. Compared to the text version of same content, there are many fewer terms used to describe content of the image in the annotation.

In ad-hoc information retrieval (IR) tasks, a popular method to address the more general problem of query-document mismatch is query expansion (QE). This seeks to add terms to the user query which will match with terms appearing in relevant documents. Standard QE methods can also be applied to improve retrieval effectiveness in image retrieval. In our research, we aim to address the question: is standard QE the most suitable method to address the IAP problem in text-based image retrieval? Classical QE methods greatly depend on the target collection to provide useful terms for QE. The IAP problem means that the assumption that the target collection provides enough information for feedback in image retrieval may often be violated. We propose that an effective solution to IAP for image search could be the introduction of an external resource in the feedback process. Furthermore, since the search query in image retrieval is usually a noun phrase for which there is a high chance that Wikipedia contains specific articles to describe it, Wikipedia is a suitable external source for QE in

[1] http://picasaweb.google.com/
[2] http://www.flickr.com/
[3] http://en.wikinews.org/

image search. In our work we refer to Wikipedia articles which directly describe the contents of a user query as *definition documents (DDs)*. Based on this analysis, we propose a definition document based relevance feedback (DRF) method for text-based image retrieval task.

The remainder of this paper is structured as follows: Section 2 overviews background and related work to our investigation; Section 3 presents our DRF method including identifying DDs using query key terms, feedback document weighting and feedback term selection; Section 4 describes our experimental setup and results, this compares standard PRF methods based on the target annotated search collection, external PRF which conducts QE on the external resource collection, feedback term selection from DDs only and our DRF method; and finally Section 5 gives conclusions and directions for further work.

2 Background and Related Work

The IAP problem in text-based image retrieval task is typically addressed by relevance feedback (RF) approach [1]. One standard approach to RF is QE where terms from top ranked documents from an initial search are added to the original query before performing another search run. A popular method of RF is pseudo relevance feedback (PRF) where top ranked documents are assumed to be relevant without being judged by the searcher. PRF via QE traditionally focuses on selecting expansion terms from top ranked documents from an initial retrieval on the target document collection. In recent research however, with the rapid growth of the web and other electronic document resources, QE from external resources has received increased attention. This approach, the initial retrieval run is carried out on an external corpus, and feedback terms are then selected from the top ranked documents in external corpus. The new expanded query is applied to the target corpus to conduct the final retrieval run. These research topics are strongly related to our query reformulation method for image retrieval tasks.

Various techniques and resources have been investigated for RF using external resources in existing work. Elsas et al. [2] utilize the link structure of Wikipedia for QE in a Blog distillation task and yield significant improvement for retrieval effectiveness, this work also showed that standard PRF does not perform well in a Blog distillation task. Yang et al. [3] classify user queries into three types: entity queries, ambiguous queries, and broader queries. In this work, for entity queries expansion terms are selected only from an entity page in Wikipedia. Their experiments show improvement on several TREC evaluation tasks. Yin et al. [4] compare two QE methods from an external resource, one selects QE terms from user search query logs, the other method selects feedback terms from snippets gathered from search engine output results. Their results show that the snippet approach was more effective. Kwok et al. [5] use a technique of collection enrichment for QE which is essentially QE from an external resource. Their system performs between 9% to 26% better than the initial retrieval as measured using mean average precision (MAP) as reference. Xu et al. [6] identify an entity

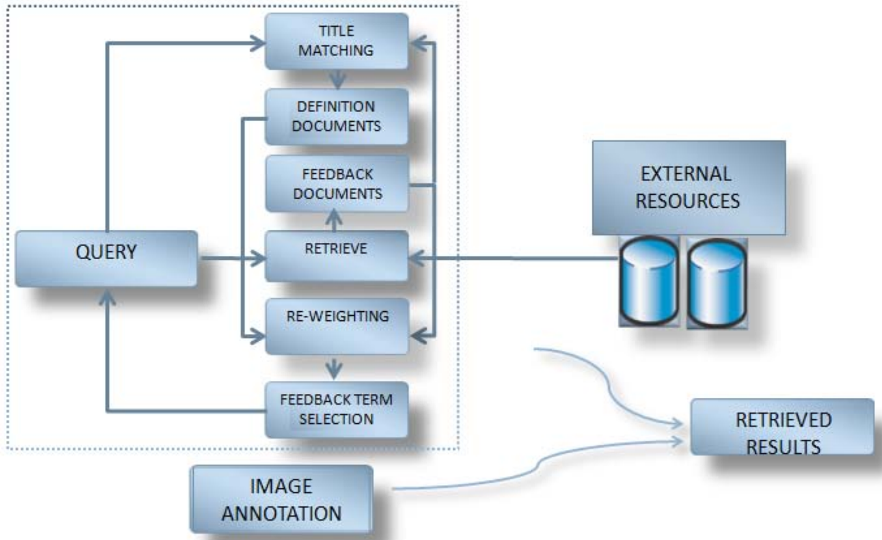


Fig. 2. Flowchart of the algorithm

page and reformulate the query with phrases from the entity page in Wikipedia. Their results show improvement in several TREC evaluation tasks compared to a language modelling IR baseline. Weerkamp et al. [7] explore different ways of using external corpora to expand the original query in a Blog post retrieval task. They achieve their best results when using external expansion on a combination of news, Wikipedia and blog posts. Custis et al. [8] apply language modelling keyword search augmented with Berger and Lafferty's (1999) translation model for QE to formulate three QE methods using word co-occurrence statistics from a large external corpus and user clickthrough data. Results show that QE using the translation model is effective for retrieval in the legal domain. Weerkamp et al. [9] propose a generative model for expanding queries using external collections in which dependencies between queries, documents, and expansion documents are explicitly modeled. Results using two external collections (news and Wikipedia) show texternal expansion for retrieval of user generated content to be effective. Hersh et al. [10] expand the query from web pages online in a genomic IR task. Our own previous work [11] reports initial experiments on QE from Wikipedia for a text-based image retrieval tasks, and shows improvement compared with the QE from the target corpus. We extend this earlier work in this paper.

3 Definition Documents Based Relevance Feedback

In this section, we introduce our definition documents (DDs) based RF method. This method utilises DDs identified by key-term title matching to re-weight the feedback documents. Our document-based relevance feedback (DRF) algorithms consist of the following steps as shown in Figure 2:

1. The user query is applied to Wikipedia to conduct an initial retrieval run to produce a ranked list;
2. The user query is applied to the top ranked documents retrieved in stage 1 to conduct key-term title matching to find the DDs for this query;
3. The DDs identified in the second stage are used as indicators to compute the similarity score with the top k ranked documents from the initial retrieval run in stage 1 by the Jaccard coefficient;
4. The similarity score from DDs is used to form a new weight for every feedback document;
5. Feedback terms are selected from the feedback documents from stage 2 with new associated weights;
6. The new query updated with feedback terms is applied to the target search collection to carry out the final retrieval run.

In the following subsections, we introduce the standard PRF method used in our work in subsection 3.1, then the key-term title matching method is described in subsection 3.2, and the feedback term weighting method is addressed in subsection 3.3.

3.1 Pseudo Relevance Feedback

PRF is the standard method used for QE. It has been found to improve average search effectiveness in many ad-hoc text search tasks. Equation 2 is a typical method for selection of feedback terms in PRF [12].

$$RW(i) = log[\frac{(r+0.5)(N-n-R+r+0.5)}{(n-r+0.5))(R-r+0.5)}] \tag{1}$$

$$Weight_{PRF}(t) = r * RW(i) \tag{2}$$

where r is the number of top-ranked feedback documents which contain the term t and $RW(i)$ is computed using Equation 1 where N is the total number of documents in the corpus and n is the number of documents where the term t appears. R is the number of known relevant documents for a query. Another simple version for PRF is Equation 3 where idf can be computed using Equation 4.

$$Weight_{PRF}(t) = r * idf(t) \tag{3}$$

$$idf(t) = \log \frac{N}{n} \tag{4}$$

3.2 Identifying Definition Documents by Key-Term Title Matching

In this section, we address the problem of how to find the DDs for a query. For a query, some documents very strongly related to the query can be found in Wikipedia, we can refer to these documents as "relevant" to the query in the sense that they essentially describe one or more concepts contained in the query. For example, given a query "Ferrari" a Wikipedia DD will appear among the top

ranked list after a prior retrieval run. Figure 3 illustrates an example user query with a DD in Wikipedia. Since exact DDs may not be found for all queries, our DRF method allows a more relaxed matching approach in these cases. We use a partial matching approach with regard to Wikipedia documents in top ranked documents from the prior retrieval whose title contains the key term of the user query as the DD of this query.

> *Ferrari is an Italian sports car manufacturer based in Maranello, Italy. Founded by Enzo Ferrari in 1929, as Scuderia Ferrari, the company sponsored drivers and manufactured race cars before moving into production of street-legal vehicles as Ferrari S.p.A. in 1947. Throughout its history, the company has been noted for its continued participation in racing, especially in Formula One, where it has had great success.*

Fig. 3. Definition Document Example

Given a query Q: $\{q_1,q_2,...,q_m\}$ and a document D with title T: $\{t_1, t_2, ..., t_n\}$, the key term q_k of the Q is the term with highest *idf* score given by Equation 4. Document D whose title contains q_k is called the DD of query Q. There may be more than one DD for a given query in Wikipedia. We use the *idf* values of the terms in the target collection to identify the key term in the query.

3.3 Feedback Term Weighting

Our RF method is built on the simple version of PRF in Equation 3. In Equation 3, PRF assigns all the top documents from the prior retrieval the same importance, which is usually not actually true. This is built on the assumption that the top k feedback documents in the prior retrieval are all relevant to the query. Of course, this assumption is generally not true for most retrieval tasks.

When some DDs have already been identified by title matching, we assume those feedback documents which are similar to the DDs have a higher probability of providing useful external knowledge to the query. The similarity of the feedback documents and the DDs is computed using a pairwise method using the Jaccard coefficient in Equation 5 where V_i, V_j is the vocabulary set of document i and j [13].

$$sim(D_i, D_j) = \frac{V_i \bigcap V_j}{V_i \bigcup V_j} \tag{5}$$

By knowing which document is more useful to the query, new document weights are assigned to the feedback documents using Equation 9. DDs are used as indicators for the top-ranked feedback documents in the prior retrieval as shown in Equation 7. For each feedback document, we have an initial retrieval score for query S_i. We normalize the scores into the range [0, 1] using Equation 6.

$$S_{nm}(i) = \frac{S_i - S_{min}}{S_{max} - S_{min}} \tag{6}$$

In Equation 7, J is the set of all DDs and $j \in J$; $sim(D_i, D_j)$ is the similarity score of document i for DD j calculated using Equation 5; $S_{nm}(j)$ is the normalized retrieval score of DD j; $sim(J)_{avg}$ is the average similarity score for all definition documents J with all feedback documents;

$$G(i) = \frac{\sum_J (sim(D_i, D_j) - sim(j)_{avg}) S_{nm}(J)}{\sum_J S_{nm}(j)} \tag{7}$$

Before combining these scores into the final weight score for a feedback document, $G(i)$ is normalized into the range [0, 1] using Equation 8 where G_{max} and G_{min} are the highest and lowest $G(i)$ scores calculated using Equation 7 for all feedback documents.

$$G_{nm}(i) = \frac{G_i - G_{min}}{G_{max} - G_{min}} \tag{8}$$

$$W_{new}(D_i) = \alpha * \overline{S_{nm}} + \beta * G_{nm}(i) \tag{9}$$

where $\overline{S_{nm}}$ is the average normalized retrieval score for all the feedback documents. α and β are parameters to adjust the rating system ($\alpha \geq 0$, $\beta \geq 0$). In Equation 9, the new weight of a FD is combined from two parts: one is the average normalized retrieval score which is identical for every FD; the other is from the rating scores of different DDs. If we set $\beta = 0$ and $\alpha \neq 0$, our method automatically falls into the simple version of PRF method; if we set $\beta \neq 0$ and $\alpha = 0$, the weights of FDs are all decided by the rating scores of the DDs.

With the new weights for the all feedback documents in the prior retrieval, the top feedback terms are selected using Equation 10 where r is the set of feedback documents which contain term t.

$$Weight_{DRF}(t) = idf(t) \cdot \sum_{D_i \in r} W_{new}(D_i) \tag{10}$$

4 Experimental Setup and Results

In this section, we describe our experimental setup and results. The data and retrieval model used in experiments are described in subsection 4.1, while the manual evaluation of precision of DDs is described in subsection 4.2. The effect of parameter setting on DRF and PRF is presented in subsection 4.3, with results of comparing DRF and PRF in subsection 4.4, and a comparison with our method of selecting feedback terms only from DDs in Section 4.5.

4.1 Experimental Setup

In this section, we describe our experimental setup. Experiments were conducted using the collection from the ImageCLEF WikipediaMM 2008 and 2009 tasks. The corpus is taken from the (INEX MM) Wikipedia image collection and includes 151,519 images [14]. Every image is associated with a metadata file. These

Table 1. Data Average Length

Data	Average Length (by terms)
Topics	2.8
Annotation Documents	24.4
English DBpedia Documents	99.7

Table 2. Overview on the definition documents

No. of topics	120
No. of overall definition documents	421
Average No. of DDs per topic	3.5
DDs with total match	77
Topics with total match DDs	46

metadata documents are typically very short, meaning that there is a high chance of IAP problems in this collection.

All our experimental results are based on the 120 official queries in this collection. Another important resource we use is the English Wikipedia abstract collection (DBpedia) including 2,452,726 documents which is used as the external resource for QE. We chose the English DBpedia collection as the external resource for QE in this study since: 1) the DBpedia dataset contains only the abstract documents of Wikipedia terms and so contains less noise than full articles; 2) the DBpedia corpus covers many topics which holds the promise that we can find relevant documents for a large number of queries. The average length of data are shown in Table 1. We use the Okapi BM25 model in the Lemur toolkit[4] for retrieval tasks.

4.2 Evaluation on Definition Documents

Our DDs are selected by key term matching from document titles. A further question in this process is how good are DDs as indicators for feedback? We manually evaluated the DDs for the official queries from WikipediMM tasks. These DDs are selected from the top 30 Wikipedia documents in the prior retrieval run. In Table 2, DDs with "total match" means those DD's with titles which exactly match the query terms after removal of stop words. We also manually evaluate the relevance of the DDs with the original topics. The results are shown in Table 3. As shown in Table 3, all the total match DDs and most partial match DDs are relevant to the original topics. The results indicate that DDs are a good feedback source for text queries in image retrieval.

4.3 Parameters Setting

To find suitable parameters for our DRF method for Equation 9, several combinations of α, β values were tested in our experiments as shown in Table 4. Firstly

[4] http://www.lemurproject.org/

Table 3. Evaluation on the definition documents

	Relevant	Non-relevant
total match definition documents	100%	0
partial match definition documents	85.5%	14.5%

Table 4. Parameters Choice for DRF Method

Parameters Setting	MAP	NDCG	P@10	R-Prec
$\alpha = 1, \beta = 0$	0.2529	0.5322	0.3157	0.2899
$\alpha = 1, \beta = 1$	0.2619	0.5409	0.3386	0.2986
$\alpha = 1, \beta = 2$	0.2623	0.5413	0.3400	0.2980
$\alpha = 1, \beta = 5$	0.2641	**0.5414**	0.3414	0.2987
$\alpha = 0, \beta = 1$	**0.2650**	0.5404	**0.3457**	**0.2995**
$\alpha = 2, \beta = 1$	0.2568	0.5350	0.3343	0.2900
$\alpha = 5, \beta = 1$	0.2503	0.5147	0.3157	0.2803

we set $\alpha = 1$, the results show that larger values of β give better results; secondly we set $\beta = 1$, the results show that smaller α gives better results. From Table 4, we can see that $\alpha = 0$ and $\beta = 1$ gives the best result in our experiments. In Table 4, the number of feedback documents is 30 (a higher number than is typically used for standard PRF, but is more effective when using DBpedia) and the number of feedback terms is 10 (a typical value for QE using PRF).

To further investigate the impact of parameter setting, we compare the performance of DRF on external resource (Run: DRF) to the PRF on external resource (Run: PRF2) and PRF on target annotation collection results (Run: PRF1) with different parameter settings. On the left side of Figure 4, the number of feedback terms in all Runs is set as 10; on the right side, the number of feedback documents in all Runs is set as 30. As shown in Figure 4, DRF outperforms PRF1 and PRF2 when the number of feedback documents is larger than 15, where the number of feedback terms is fixed at 10; DRF outperforms PRF1 and PRF2 for all choices of number of feedback terms for a fixed number of feedback documents.

4.4 Comparing DRF with PRF

Table 5 shows results comparing DRF, PRF1 (baseline Run) and PRF2 with their best performance. The results in Table 5 indicate that PRF from DBpedia achieves higher retrieval effectiveness than the baseline based on the criterion of MAP. Furthermore, DRF outperforms PRF for all retrieval criteria. A paired t-test was applied to compare MAP for PRF2 and DRF ($p = 0.0069 < 0.05$; significant improvements are indicated by * in Table 5). Comparing the DRF method with PRF on the target annotation collection, the result gains 11.67% in terms of MAP.

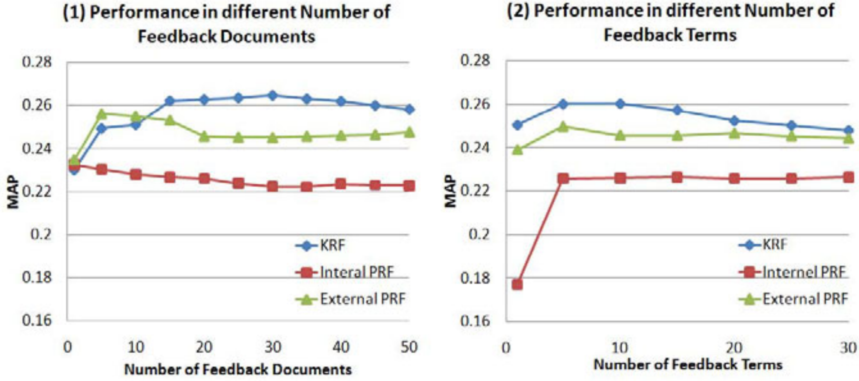


Fig. 4. Performance in Different Parameters Setting

Table 5. Results Comparison

Runs	MAP		NDCG	P@10	R-Prec
PRF1	0.2373		0.5055	0.3200	0.2772
PRF2	0.2529	+6.58%	0.5322	0.3157	0.2899
DRF	**0.2650**	**+11.67%**	**0.5404**	**0.3457**	**0.2995**

4.5 Comparing DRF with Feedback from DDs

Since DDs play a very important role in the feedback process, one question is why not directly select terms from the DDs as the feedback terms. We carry out the PRF method to select feedback terms from DDs only using Equation 3. The results of this experiment are shown in Table 6. This experiment shows that using DRF is more effective in call cases.

4.6 Discussion

The key issue in QE is selecting feedback terms from the top ranked documents from the prior retrieval run. As stated previously, PRF assumes that all the top ranked documents are relevant, which will generally not be true. The identified relevant documents from Wikipedia help to judge which documents are more relevant to the query. Our results show that the DRF method can be effective for queries for which the DDs can be found in Wikipedia. However, feedback terms selected from non relevant documents can introduce a query drift problem for in the QE process.

Our results show that directly selecting feedback terms from DDs only does not perform better than our proposed method. The main reason for this is the fact that the number of DDs is very small and cannot not provide enough information in the feedback process. Our term weighting method fully utilizes the characteristic of queries in image retrieval where all queries are noun phrases.

Table 6. Compare DRF with term selection from DDs only

Parameters Setting	MAP	NDCG	P@10	R-Prec
DRF	**0.2650**	**0.5404**	**0.3457**	**0.2995**
DDs only	0.2403	0.5221	0.3180	0.2831

We assume that it is easy to find DDs among the Wikipedia dataset for these queries. Our manual evaluation of the relevance of DDs on original topics proves that our assumption is true.

5 Conclusion and Future Work

In this paper we have introduced the incomplete annotation problem in image retrieval. As a solution to this, an external knowledge resource was introduced in the relevance feedback process. Comparing PRF on the target annotation collection to PRF on external resource, the external method achieves better results in our experiments. Furthermore, we presented a DD based relevance feedback method for QE from external resources. The key idea of the DRF method is to use the DDs identified from Wikipedia as an indicator to judge the quality of the feedback documents. The assumption is that the DDs provide more useful external knowledge in the process of feedback term selection. Thus combining the new weights from different rating scores, the DRF method can help to ensure that selected expansion terms from these documents with a high probability of being useful are used to expand the query knowledge, with the objective of solving the IAP problem in image retrieval. Our results show that the DRF method outperforms the PRF using the same external resource significantly.

We conclude that using the DDs as an evidence to help in QE is a good direction for utilizing Wikipedia related resources in text-based image retrieval research. For future work, the DRF method will be explore for other information retrieval tasks, including those which do not suffer so obviously from incomplete annotation.

Acknowledgments. This research is supported by the Science Foundation Ireland (Grant 07/CE/I1142) as part of the Centre for Next Generation Localisation (CNGL) project.

References

1. Tsikrika, T., Kludas, J.: Overview of the wikipediamm task at imageCLEF 2008. In: Peters, C., Deselaers, T., Ferro, N., Gonzalo, J., Jones, G.J.F., Kurimo, M., Mandl, T., Peñas, A., Petras, V. (eds.) CLEF 2008. LNCS, vol. 5706, pp. 539–550. Springer, Heidelberg (2009)
2. Elsas, J.L., Arguello, J., Callan, J., Carbonell, J.G.: Retrieval and feedback models for blog feed search. In: SIGIR 2008: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development In Information Retrieval, pp. 347–354. ACM, New York (2008)

3. Yang, X., Jones, G.J.F., Wang, B.: Query dependent pseudo-relevance feedback based on Wikipedia. In: SIGIR 2009: Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 59–66. ACM, New York (2009)
4. Yin, Z., Shokouhi, M., Craswell, N.: Query expansion using external evidence. In: Boughanem, M., Berrut, C., Mothe, J., Soule-Dupuy, C. (eds.) ECIR 2009. LNCS, vol. 5478, pp. 362–374. Springer, Heidelberg (2009)
5. Kwok, K.L.: Improving English and Chinese ad-hoc retrieval: A Tipster text phase 3 project report. Inf. Retr. 3(4), 313–338 (2000)
6. Xu, Y., Ding, F., Wang, B.: Entity-based query reformulation using Wikipedia. In: CIKM 2008: Proceeding of the 17th ACM Conference on Information and Knowledge Management, pp. 1441–1442. ACM, New York (2008)
7. Weerkamp, W., de Rijke, M.: External query expansion in the blogosphere. In: Seventeenth Text Retrieval Conference (TREC 2008), NIST (February 2009)
8. Custis, T., Al-Kofahi, K.: Investigating external corpus and clickthrough statistics for query expansion in the legal domain. In: CIKM 2008: Proceeding of the 17th ACM Conference on Information and Knowledge Management, pp. 1363–1364. ACM, New York (2008)
9. Weerkamp, W., Balog, K., de Rijke, M.: A generative blog post retrieval model that uses query expansion based on external collections. In: ACL-IJCNLP 2009: Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP, vol. 2, pp. 1057–1065. Association for Computational Linguistics, Morristown (2009)
10. Hersh, W.R., Bhupatiraju, R.T., Price, S.: Phrases, boosting, and query expansion using external knowledge resources for genomic information retrieval. In: TREC, pp. 503–509 (2003)
11. Min, J., Wilkins, P., Leveling, J., Jones, G.J.F.: DCU at WikipediaMM 2009: Document expansion from Wikipedia abstracts. In: Working Notes for the CLEF 2009 Workshop, Corfu, Greece (2009)
12. Robertson, S., Spärck Jones, K.: Simple, proven approaches to text retrieval. Technical Report UCAM-CL-TR-356, University of Cambridge, Computer Laboratory (December 1994)
13. Jaccard, P.: Etude comparative de la distribution florale dans une portion des alpes et des jura. Bulletin de la Socit Vaudoise des Sciences Naturelles 37, 547C–579C (1901)
14. Westerveld, T., van Zwol, R.: The INEX 2006 multimedia track. In: Fuhr, N., Lalmas, M., Trotman, A. (eds.) INEX 2006. LNCS (LNAI), vol. 4518, pp. 331–344. Springer, Heidelberg (2007)

A Knowledge-Based Semantic Kernel for Text Classification

Jamal Abdul Nasir[1], Asim Karim[1], George Tsatsaronis[2], and Iraklis Varlamis[3]

[1] School of Science and Engineering, LUMS, Pakistan
[2] Biotechnology Center (BIOTEC), Technische Universität Dresden, Germany
[3] Department of Informatics and Telematics, Harokopio University of Athens, Greece
{jamaln,akarim}@lums.edu.pk,
george.tsatsaronis@biotec.tu-dresden.de,
varlamis@hua.gr

Abstract. Typically, in textual document classification the documents are represented in the vector space using the "Bag of Words" (*BOW*) approach. Despite its ease of use, BOW representation cannot handle word synonymy and polysemy problems and does not consider semantic relatedness between words. In this paper, we overcome the shortages of the *BOW* approach by embedding a known *WordNet*-based semantic relatedness measure for pairs of words, namely *Omiotis*, into a semantic kernel. The suggested measure incorporates the TF-IDF weighting scheme, thus creating a semantic kernel which combines both semantic and statistical information from text. Empirical evaluation with real data sets demonstrates that our approach successfully achieves improved classification accuracy with respect to the standard *BOW* representation, when *Omiotis* is embedded in four different classifiers.

Keywords: Text Classification, Thesaurus, Semantic Kernels.

1 Introduction

The key steps in text classification are document representation and classifier training using a corpus of labeled documents. In the commonly used 'Bag of Words' (*BOW*) representation, documents are represented by vectors whose components are weights given to different words or terms occurring in the document. Weights indicate the importance of each word, typically quantified by measures like TF-IDF. However, the *BOW* representation has some significant limitations: (1) It disregards the sequential order of words in documents. (2) It considers synonyms as distinct components of the vector (synonymy problem). (3) It disregards polysemy of words (i.e. words having multiple senses or meanings – polysemy problem). The lack of semantics in the *BOW* representation limits the effectiveness of automatic text classification methods.

In the absence of external semantic knowledge, corpus-based statistical methods, such as Latent Semantic Analysis (LSA) [1] can be applied to alleviate the synonymy problem, but the problem of polysemy still remains. The application

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 261–266, 2011.
© Springer-Verlag Berlin Heidelberg 2011

of Word Sense Disambiguation (WSD) techniques [2] during document preprocessing can be helpful; however, this is usually computationally expensive, and the performance of the unsupervised techniques is poor while use of supervised techniques requires large amounts of hand-annotated text documents. The use of external semantic knowledge provided by word thesauri or ontologies to adjust or "smooth" the *BOW* representation has shown much promise [3,4]. However, the embedding of semantic information is usually computationally expensive.

In this paper, we present and evaluate a semantically-enriched *BOW* representation for text classification. We adopt a recently proposed semantic relatedness measure called *Omiotis* [5] for building a smoothing matrix and a kernel for semantically adjusting the *BOW* representation. *Omiotis* is constructed from the word thesaurus and lexical ontology *WordNet*, and is capable of handling the synonymy and polysemy problems. We evaluate four popular text classification methods on four different data sets with and without *Omiotis*-based semantic smoothing of *BOW* representation. The results demonstrate that our semantic kernel produces significant improvement in text classification performance.

The paper is organized as follows. Section 2 discusses the related work. Section 3 presents the *Omiotis* measure for the semantic relatedness between pairs of terms. In Section 4, we develop our semantic kernel and semantic smoothing matrix, and discuss its computational complexity. Section 5 presents our experimental results. Finally, Section 6 discusses our next steps.

2 Semantics in Text Mining and Information Retrieval

The importance of embedding semantic relatedness between two text segments for text classification was initially highlighted in [6] where semantic similarity between words has been used for the *semantic smoothing* of the *TF-IDF* vectors.

Semantic-aware kernels have been proposed by Mavroeidis et al. [4] who propose a generalized vector space model with *WordNet* senses and their hypernyms to improve text classification performance. Bloehdorn at al. [7] propose smoothing kernels for text classification by implicitly encoding a super concept expansion and achieve satisfactory results under poor training data and data sparseness. In [8] authors use the Latent Semantic Indexing (LSI) approach for capturing semantic relations between terms and embed them into their semantic kernel. Basili et al.[9] propose kernel functions to use prior knowledge in learning algorithms for document classification by means of the term similarity based on the *WordNet* hierarchy (conceptual density). Results show the benefit of the approach for Support Vector Machines when few training examples are available.

In this work, we present a new semantic smoothing matrix and kernel for text classification, based on a semantic relatedness measure that takes into account all of the available semantic relations in *WordNet*, by embedding the *Omiotis* measure introduced by Tsatsatonis et al. [5]. Our experimental evaluation offers an additional empirical evidence towards the claim that embedding semantic information from a knowledge base, such as *WordNet*, through a semantic kernel, improves the text classification performance.

3 Semantic Relatedness and the *Omiotis* Measure

Lexical relatedness measures can be roughly classified in three categories: (1) knowledge-based measures; (2) corpus-based measures; and (3) hybrid measures. In this work, we are using the *Omiotis*[5] knowledge-based measure for computing the relatedness between terms or words. *Omiotis* is based on a sense relatedness measure, called *SR*. Due to space limitations, we suggest readers to consult [5] for the details of *SR*, which given a pair of senses s_1, s_2, finds all the paths that connect s_1 to s_2 in the WordNet's graph and defines the pair's relatedness as:

$$SR(s_1, s_2) = \max_{P=\langle s_1,\ldots,s_2\rangle} \{SCM(P) \cdot SPE(P)\}$$

where P ranges over all the paths that connect s_1 to s_2, SCM and SPE capture respectively the notions of the *value* of the path connecting two senses in WordNet, as well as of the *depth* of path's edges in the path with respect to the *height* of the used thesaurus/ontology. If no path exists, then $SR(s_1, s_2) = 0$.

The measure can be expanded to measure the semantic relatedness between terms, by selecting the maximum for each of the pairwise sense combinations for a pair of terms. More precisely, given a pair of terms $T : (t_1, t_2)$ for which there are entries in O, let X_1 be the set of senses of t_1 and X_2 be the set of senses of t_2 in O. Let $S : \{S_1, S_2, \ldots, S_{|X_1|\cdot|X_2|}\}$ be the set of pairs of senses, $S_k = (s_i, s_j)$, with $s_i \in X_1$ and $s_j \in X_2$. Then $SR(T, S, O)$ is defined as:

$$\max_{S_k}\{\max_{P}\{SCM(S_k, O, P) \cdot SPE(S_k, O, P)\}\} = \max_{S_k}\{SR(S_k, O)\} \forall k = 1..|X_1| \cdot |X_2|. \quad (1)$$

Semantic relatedness between two terms t_1, t_2 where $t_1 \equiv t_2 \equiv t$ and $t \notin O$ is defined as 1. Semantic relatedness between t_1, t_2 when $t_1 \in O$ and $t_2 \notin O$, or vice versa, is considered 0. This latter definition of *SR* for a pair of terms is the definition of the *Omiotis* measure that we are using in our case.[1]

4 *Omiotis*-Based Semantic Kernel

4.1 Semantic Smoothing Matrix and Semantic Kernel Design

A document d is represented in the *BOW* representation as follows:

$$\phi : d \mapsto \phi(d) = [tf\text{-}idf(t_1, d), tf\text{-}idf(t_2, d), \ldots, tf\text{-}idf(t_D, d)]^T \in \Re^D$$

where $tf\text{-}idf(t_i, d)$ is the TF-IDF weight of term t_i in document d, and D is the total number of terms (e.g. words) in the dictionary (the superscript T denotes the transpose operator). In the above expression, the function $\phi(d)$ represents the document d as a TF-IDF vector. This function, however, can be any other mapping from a document to its vector space representation.

[1] A Web service implementation of *Omiotis* with pre-computed *SR* scores for all *WordNet* sense pairs is made available by the authors in [5], at `http://omiotis.hua.gr/`.

To enrich the *BOW* representation with semantic information, we construct the semantic relatedness matrix R using the *Omiotis* semantic relatedness measure. Specifically, the i, j element of matrix R is given by $SR(T, S, O)$ (refer to Eq. 1), which quantifies the semantic relatedness between terms $T : (t_i, t_j)$. Thus, R is a $D \times D$ symmetric matrix with 1's in the principal diagonal. This smoothing matrix can be used to transform the documents' vectors in such a way that semantically related documents are brought closer together in the transformed (or feature) space (and vice versa). Mathematically, the semantically enriched *BOW* representation of a document d is given as:

$$\bar{\phi}(d) = (\phi(d)^T R)^T$$

Although the feature space defined above can be used directly in many classification methods, it is sometimes helpful to define the feature space implicitly via the kernel function. This is particularly important in kernel-based methods or kernel machines when the feature space is very large or even infinite in size. By definition, the kernel function computes the inner product between documents d_i and d_j in the feature space. For our case, this can be written as:

$$\kappa(d_i, d_j) = \bar{\phi}(d_i)^T \bar{\phi}(d_j) = \phi(d_i)^T R R^T \phi(d_j) \tag{2}$$

For this to be a valid kernel function, the Gram matrix G (where $G_{ij} = \kappa(d_i, d_j)$) formed from the kernel function must satisfy the Mercer's conditions [8]. These conditions are satisfied when the Gram matrix is positive semi-definite. It has been shown in [8] that the matrix G formed by the kernel function (Eq. 2) with the outer matrix product RR^T is indeed a positive semi-definite matrix.

4.2 Computational Aspects

The computational complexity of the suggested semantic kernel depends on two main factors: (1) the similarity measure between two documents d_1 and d_2, which requires the evaluation of all the unique term pairs' relatedness values and has a theoretical complexity of $O(|d_1| \cdot |d_2|)$, where $|d|$ denotes the total number of distinct terms in document d; (2) the computational complexity of *Omiotis* for all $|d_1| \cdot |d_2|$ term pair combinations, which comprises the construction time of the semantic network to compute the paths connecting the senses of two words, and the time needed to execute the *Dijkstra*'s algorithm in order to find the optimal path connecting two senses. The complexity of the former is $O(2 \cdot k^{l+1})$ [10], where k is the maximum branching factor of the used thesaurus nodes and l is the maximum semantic path length in the thesaurus, and of the latter is $O(nL+mD+nE)$, where n is the number of nodes in the network, m the number of edges, L is the time for insert, D the time for decrease-key and E the time for extract-min. Using The use of Fibonacci heaps reduces the cost of extract-min to $O(\log n)$ and $L = D = O(1)$, thus significantly reducing the cost of execution. The pre-computation of all the pairwise sense and term relatedness values, which are publicly available through the *Omiotis* service[2] makes the semantic kernel computation applicable even for large data sets.

[2] http://omiotis.hua.gr/WebSite/wsinfo.html

5 Empirical Evaluation

We evaluate the performance of our semantic smoothing approach by using four classification methods on four popular text classification data sets (Ohsumed[3], 20 Newsgroups[4], WebKB[5] and Movie Reviews[6]). All data sets are preprocessed via tokenization, stop word removal, and TF-IDF vector construction (the standard BOW representation).

Supervised text classification methods can be based on a generative or a discriminative model of the problem. We employ two discriminative methods, Support Vector Machines (SVM) and Balanced Winnow (BW), and two common generative methods, Naive Bayes (NB) and Maximum Entropy (ME). We perform our experiments using the software RapidMiner[7] (for SVM and NB) and the Mallet toolkit[8] (for ME and BW). For each method, we evaluate its performance under two settings: (1) standard BOW representation and (2) semantically smoothed BOW represented using the *Omiotis* measure. We report the performance with average classification accuracy obtained from an 10-fold cross-validation process.

Table 1 shows the results of our empirical evaluation. It gives the percent accuracy obtained from 10-fold cross-validation by each method on the four data sets. The methods identified with the *Omiotis* subscript are the ones using our *Omiotis*-based semantic kernel (or semantic smoothing approach). These results demonstrate that enriching the BOW representation with our semantic smoothing approach improves text classification performance. This improvement is seen across different classification methods and different data sets. From among the 16 pairs of results, the performance of the *Omiotis*-based methods is better than the standard methods in 14 pairs.

Table 1. Text classification performance in percent accuracy

	$MovieReview$	$Ohsumed$	$20Newsgroups$	$WebKB$
SVM	83.30	55.15	90.08	86.37
$SVM_{Omiotis}$	**91.97**	**57.17**	**92.93**	84.58
NB	77.41	50.32	87.27	84.17
$NB_{Omiotis}$	84.13	51.29	90.44	88.52
ME	79.11	51.47	85.31	91.02
$ME_{Omiotis}$	81.86	50.17	87.35	**91.52**
BW	76.23	50.93	81.66	81.42
$BW_{Omiotis}$	79.25	51.83	84.58	85.34

[3] http://ir.ohsu.edu/ohsumed/ohsumed.html
[4] http://people.csail.mit.edu/jrennie/20Newsgroups/
[5] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/
[6] http://www.cs.cornell.edu/people/pabo/movie-review-data/
[7] http://www.rapid-i.com/
[8] http://mallet.cs.umass.edu/

To verify the consistency of the observed results, we applied the Wilcoxon signed-ranks test, which is recommended for our case [11], on the observed differences in performances of all methods on all the data sets. In our test, we found that the observed differences are statistically significant and the null hypothesis is rejected (having achieved a very low p-value of only 0.0023). This test confirms that our semantic kernel produces consistent and statistically significant improvement in text classification performance.

6 Conclusions and Future Work

In this paper, we present a semantic kernel for smoothing the BOW representation. We evaluate the impact of our semantic kernel on text classification problems using four popular classifiers on four commonly-used text corpora. We find that the *Omiotis* enhanced representation produces significant improvement in classification accuracy for all classifiers. As a next step, we will extend the BOW representation by incorporating discrimination information for text classification and evaluate and compare our representation approaches for text clustering tasks.

References

1. Deerwester, S.C., Dumais, S.T., Landauer, T.K., Furnas, G.W., Harshman, R.A.: Indexing by Latent Semantic Analysis. JASIS 41(6), 391–407 (1990)
2. Navigli, R.: Word sense disambiguation: A survey. ACM Computing Surveys 41(2), 10:1–10:69 (2009)
3. Basili, R., Cammisa, M., Moschitti, A.: A semantic kernel to exploit linguistic knowledge. In: Proc. of the AI*IA 2005, pp. 290–302 (2005)
4. Mavroeidis, D., Tsatsaronis, G., Vazirgiannis, M., Theobald, M., Weikum, G.: Word sense disambiguation for exploiting hierarchical thesauri in text classification. In: Jorge, A.M., Torgo, L., Brazdil, P.B., Camacho, R., Gama, J. (eds.) PKDD 2005. LNCS (LNAI), vol. 3721, pp. 181–192. Springer, Heidelberg (2005)
5. Tsatsaronis, G., Varlamis, I., Vazirgiannis, M.: Text relatedness based on a word thesaurus. Journal of Artificial Intelligence Research 37, 1–39 (2010)
6. Siolas, G., d'Alché-Buc, F.: Support vector machines based on a semantic kernel for text categorization. In: Proc. of IEEE IJCNN 2000, Washington, DC, USA (2000)
7. Bloehdorn, S., Basili, R., Cammisa, M., Moschitti, A.: Semantic kernels for text classification based on topological measures of feature similarity. In: Proc. of ICDM 2006, pp. 808–812 (2006)
8. Cristianini, N., Taylor, J.S., Lodhi, H.: Latent Semantic Kernels. In: Proc. of the Eighteenth International Conference on Machine Learning, pp. 66–73 (2001)
9. Basili, R., Cammisa, M., Moschitti, A.: A Semantic Kernel to classify texts with very few training examples. Informatica 30(2), 163–172 (2006)
10. Tsatsaronis, G., Vazirgiannis, M., Androutsopoulos, I.: Word sense disambiguation with spreading activation networks generated from thesauri. In: Proc. of IJCAI, pp. 1725–1730 (2007)
11. Demsar, J.: Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research 7, 1–30 (2006)

Compressed Text Indexing with Wildcards*

Wing-Kai Hon[1], Tsung-Han Ku[1], Rahul Shah[2],
Sharma V. Thankachan[2], and Jeffrey Scott Vitter[3]

[1] National Tsing Hua University, Taiwan
{wkhon,thku}@cs.nthu.edu.tw
[2] Louisiana State University, USA
{rahul,thanks}@csc.lsu.edu
[3] The University of Kansas, USA
jsv@ku.edu

Abstract. Let $T = T_1\phi^{k_1}T_2\phi^{k_2}\cdots\phi^{k_d}T_{d+1}$ be a text of total length n, where characters of each T_i are chosen from an alphabet Σ of size σ, and ϕ denotes a wildcard symbol. The text indexing with wildcards problem is to index T such that when we are given a query pattern P, we can locate the occurrences of P in T efficiently. This problem has been applied in indexing genomic sequences that contain single-nucleotide polymorphisms (SNP) because SNP can be modeled as wildcards. Recently Tam et al. (2009) and Thachuk (2011) have proposed succinct indexes for this problem. In this paper, we present the first compressed index for this problem, which takes only $nH_h + o(n\log\sigma) + O(d\log n)$ bits space, where H_h is the hth-order empirical entropy ($h = o(\log_\sigma n)$) of T.

1 Introduction

Text indexing is a fundamental problem in computer science, where the task is to index a given text $T[1..n]$ for locating all the occurrences of an online query pattern $P[1..p]$ within T efficiently. Suffix trees [20,14] and suffix arrays [13] are the most popular indexes which can answer this query in $O(p + occ)$ and $O(p + \log n + occ)$ times respectively, where occ is the number of occurrences of P in T. Both indexes take $O(n\log n)$ bits space. Note that the query time is (almost) optimal, but the index size can be asymptotically higher than the optimal $n\lceil\log\sigma\rceil$ bits required to store the text in plain form; here, σ denotes the size of the alphabet Σ from which the characters of T and P are chosen. The goal of designing optimal-size indexes was first achieved by Grossi and Vitter (Compressed Suffix Arrays) [8] and Ferragina and Manzini (FM-index) [5]. Their indexes are based on Burrows-Wheeler transform (BWT) [2].

A more general problem of text indexing deals with the case where wildcard characters may appear in the input text T [4,17,18]. Let $T = T_1\phi^{k_1}T_2\phi^{k_2}\cdots\phi^{k_d}T_{d+1}$ be a text of total length n, where characters of each T_i are chosen from an alphabet Σ of size σ, and ϕ represents a wildcard symbol that may match any single character in Σ. The text indexing problem with

* This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 267–277, 2011.
© Springer-Verlag Berlin Heidelberg 2011

wildcards is to index T such that when we are given a query pattern P, we can locate the occurrences of P in T efficiently. This problem has been applied in indexing genomic sequences that contain single-nucleotide polymorphisms (SNP) because SNP can be modeled as wildcards.

The problem of text indexing with wildcards was first studied by Cole et al [4]. They proposed an $O(n \log^k n)$-word index with $O(p+\log^k n \log\log n + occ)$ query time, where k is the total number of wildcards. Later, Lam et al. [17] proposed an $O(n)$-word index, but the query time introduces an additional term of $\gamma = \sum_{i,j} \mathit{prefix}(P[i..p], T_j)$, where $\mathit{prefix}(P[i..p], T_j) = 1$ if the text segment T_j is a prefix of $P[i..p]$, else 0. Note that γ is upper bounded by pd. Tam et al. [18] further reduced the space requirement of this index to $(3 + o(1))n \log \sigma + O(d \log n)$ bits. Recently, Thachuk [19] proposed a more space-efficient solution, taking $(2 + o(1))n \log \sigma + O(n) + O(d \log n) + O(k \log k)$ bits and requiring a smaller working space of $O((d + \log n)p)$ bits.

In all the above solutions (which has a γ term in query time), the common approach is to categorize the occurrences into the following 3 types and build separate data structures for reporting each type of occurrences of P in T.

Type-1: P matching a substring of T with no wildcard groups;
Type-2: P matching a substring of T with exactly 1 wildcard group;
Type-3: P matching a substring of T with 2 or more wildcard groups.

In this paper, we propose an index which takes near-optimal $nH_h + o(n \log \sigma) + O(d \log n)$ bits space, where H_h is the hth-order empirical entropy of T. The central technique is to make use of the *same* data structure (which is an FM-index of T) for handling all types of occurrences. We need auxiliary structures in locating type-2 and type-3 occurrences, but the space of those structures is bounded by $O(d \log n) + o(n)$ bits. Moreover, the working space requirement is $O((p + \gamma) \log n)$. As γ is upper bounded by dp, therefore our working space will be at most a $\log n$ factor worse than Thachuk's index. However, our index has its advantage when γ is small ($\gamma = o(dp/\log n)$). The table below summarizes the results of our index along with the previously known results supporting matching with wildcard characters. Here k, d and $\hat{d}$ represents the number of wildcards, wildcard groups, distinct wildcard group lengths, respectively; occ_1, occ_2 and occ represents the number of type-1, type-2 and overall occurrences of P in T respectively, and $\epsilon' > 0$ is any fixed constant.

Ref	Index (bits)	Query Time	Working (bits)
[4]	$O(n \log^{k+1} n)$	$O(p + \log^k n \log\log n + occ)$	–
[17]	$O(n \log n)$	$O(p \log n + \gamma + occ)$	$O(n \log n)$
[18]	$(3 + o(1))n \log \sigma + O(d \log n)$	$O(p(\log \sigma + \min(p, \hat{d}) \log d) + occ_1 \log n + occ_2 \log d + \gamma)$	$O(n \log d + p \log n)$
[18]	$(3 + o(1))n \log \sigma + O(d \log n)$	$O(p(\log \sigma + \min(p, \hat{d}) \log d) + occ_1 \log n + occ_2 \log d + \gamma \log_\sigma d)$	$O(n \log \sigma + p \log n)$
[19]	$(2 + o(1))n \log \sigma + O(n + d \log n + k \log k)$	$O(p(\log \sigma + \min(p, \hat{d}) \log k/\log\log k) + occ_1 \log n + occ_2 \log k/\log\log k + \gamma)$	$O(dp + p \log n)$
Ours	$nH_h + o(n \log \sigma) + O(d \log n)$	$O(p(\log^{1+\epsilon'} n + \min(p, \hat{d}) \log d) + occ_1 \log^{1+\epsilon'} n + occ_2 \log^{\epsilon'} d + \gamma \log \gamma)$	$O(\gamma \log n + p \log n)$

2 Preliminaries

2.1 Bit Vectors with Rank/Select

Let B be a bit vector of length n, the rank and select operations are defined as $rank(k) = \sum_{i=1}^{k} B[i]$ and $select(k) = i$ such that $A[i] = 1$ and $rank(i) = k$. Let d be the number of 1s in B, then B can be maintained in $d \log(n/d) + O(d + n \log\log n/ \log n)$ bits such that both rank and select operations can be performed in constant time [16].

2.2 Suffix Trees and Suffix Arrays

Suffix trees [20,14] and suffix arrays [13] are two classic data structures for online pattern matching queries. For a text $T[1..n]$, a substring $T[i..n]$ with $i \in [1, n]$ is called a suffix of T. The suffix tree for T is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. For any node v, the string formed by concatenating the edge labels from root to v is called $path(v)$. The locus node v of a pattern P is defined as the node closest to the root, such that P is a prefix of $path(v)$.

Suffix array $SA[1..n]$ is an array of length n, such that $SA[i]$ is the starting position of the ith lexicographically smallest suffix of T. The suffix range of a pattern P in SA is defined as the the maximal range $[L, R]$ such that for all $j \in [L, R], SA[j]$ is the starting point of a suffix of T with P as a prefix. Both suffix trees and suffix arrays take $O(n \log n)$ bits space and can perform pattern matching in $O(p + occ)$ and $O(p + \log n + occ)$ time respectively, where $p = |P|$ and occ is the number of occurrences of P within T.

2.3 Compressed Text Indexes

Text indexes which take space close to the size of the text is called compressed/ succinct text indexes. There are different compressed text indexes available in the literature, such as [8] and FerMan05. For our purpose, we use the FM-index by Ferragina et al. [6] which takes only $nH_h + o(n \log \sigma)$ bits space, where H_h denotes the hth-order empirical entropy ($h = o(\log_\sigma n)$) of T. For $\sigma = O(poly \log(n))$, this index can count the number of occurrences of P within T in $O(p)$ time, locate each pattern occurrence in $O(\log^{1+\epsilon} n)$ and display a text substring of length ℓ in $O(\ell + \log^{1+\epsilon} n)$ time.

2.4 Compressed Index for Dictionary Matching

The dictionary matching problem is to index a set of (short) text segments $\{T_1, T_2, \ldots, T_{d+1}\}$ of total length n, such that all the occurrences of these text segments within an online (long) query pattern P can be computed efficiently. This is a well-studied problem and many indexes are available in the

literature [1,9,10]. The recently proposed indexes by Belazzougui [1] and Hon et al. [9] can solve this problem in (almost) optimal space and optimal time. For our purpose, we choose the most space-efficient index (even though the query time is not optimal) by Hon et al. [10]. Their index takes $nH_h + o(n \log \sigma) + O(d \log n)$ bits space and the query time is $O(p(\log^\epsilon n + \log d) + \gamma)$, where H_h denotes the hth-order empirical entropy of the text segments collection and γ denotes number of occurrences of text segments in P, and $\epsilon > 0$ is any fixed constant. In [10], it is assumed that the text segments are stored using Ferragina-Venturini scheme, where the storage space is $nH_h + o(n \log \sigma)$ bits and displaying a text substring of length ℓ takes $O(\ell / \log_\sigma n + 1)$ time [7] .

2.5 Orthogonal Range Reporting

Let $\mathcal{R} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be a set of n points in the two-dimensional space. An orthogonal range reporting query on $\mathcal{R}$ is defined as follows: Given a query range $[x_\ell, x_r] \times [y_\ell, y_r]$, report all points (x_i, y_i) such that $x_\ell \leq x_i \leq x_r$ and $y_\ell \leq y_i \leq y_r$. For our purpose, we use the $O(n \log n)$ bits structure by Nekrich [15] which can perform an orthogonal range reporting of t output points in $O(\log n + t \log^\epsilon n)$ time.

2.6 Sparse Suffix Trees

Sparse suffix tree of a text is a compact trie, which consists of only selected suffixes of the text [3,11,12]. We shall define the sparse suffix tree for a collection $\{T_1, T_2, \ldots, T_{d+1}\}$ of $d+1$ text segments of total length n as follows: Let α be a *sampling factor*, and for each text segment $T_j[1..|T_j|]$, the suffix $T[i..|T_j|]$ such that $i \ mod(\alpha) = 1$ is called an α-sampled suffix of T_j. A trie of all α-sampled suffixes of all text segments is called a (forward) sparse suffix tree Δ_f. Similarly a trie of all α-sampled suffixes of $\overleftarrow{T_j}$ for $j = 1, 2, \ldots, d+1$ is called (reverse) sparse suffix tree Δ_r, where $\overleftarrow{T_j}$ is the reverse of T_j. The number of nodes in Δ_f (Δ_r) can be bounded by $O(n/\alpha + d)$. For each internal node $u \neq root$, there exists a unique node v such that $path(v)$ can be obtained by deleting the first α characters of $path(u)$. Then we maintain a pointer from node u to node v, which we called an α-sampled suffix link. The contiguous range of all α-sampled suffixes in the subtree of u is called the suffix range of a pattern P, where u is the locus node (node closest to root such that P is a prefix of $path(u)$) of P. Note that neither Δ_f, nor Δ_r is a self-index, hence we maintain the original text $T = T_1 \phi^{k_1} T_2 \phi^{k_2} \cdots \phi^{k_d} T_{d+1}$ in the form of FM-index [6] in $nH_h + o(n \log \sigma)$ bits, which is capable of retrieving any substring of T of length ℓ in $O(\ell + \log^{1+\epsilon} n)$ time ($\sigma = \text{polylog}(n)$). We store the starting character of every edges explicitly and from every node ($\neq root$), we maintain a pointer to its ancestor. By choosing $\alpha = \log^{1+\epsilon} n, \epsilon > 0$, the size of Δ_f and Δ_r, together with the encoding of T, can be bounded by $nH_h + o(n \log \sigma) + O(d \log n)$ bits.

Lemma 1. *Given a pattern $P[1..p]$, the suffix ranges of all its suffixes ($P[i..p]$ for $i = 1, 2, 3, \ldots, p$) in Δ_f can be computed in $O(p \log^{1+\epsilon'} n)$ time. Similarly the suffix ranges of the reverse of all its prefixes ($\overleftarrow{P[1..i]}$, for $i = 1, 2, 3, \ldots, p$) in Δ_r can be computed in $O(p \log^{1+\epsilon'} n)$ time, where $\epsilon' > 0$.*

Proof: Firstly we show how to compute the suffix ranges of all suffixes of P in Δ_f. The procedure works in α stages. At stage k (for $k = 1, 2, 3, \ldots, \alpha$), we compute the suffix ranges of all α-sampled suffixes ($P[k..p]$, $P[(k + \alpha)..p]$, $P[(k + 2\alpha)..p]$, ...) of $P[k..p]$. The main challenge comes from the fact that the time for retrieving a substring of T of length ℓ is $O(\ell + \log^{1+\epsilon} n)$, which means $O(\log^{1+\epsilon} n)$ time is needed for retrieving even a single character in T. We handle this situation carefully as follows: Firstly, we do a blind matching of first α characters of $P[k..p]$ in Δ_f by only matching the starting characters of the edges. Next, we verify if this matching is correct by retrieving α characters from the FM-index and matching them in $O(\alpha \log \sigma)$ time. If the first α characters are matched, we continue matching the next α characters, and so on. We stop when all characters in $T[k..p]$ are matched or encounter the first mismatch. Let x be the number of characters matched, then the matching time can be bounded by $O(x \log \sigma + \log^{1+\epsilon} n)$ ($O(x + \log^{1+\epsilon} n)$ time for retrieving x characters from FM-index of T and matching those x characters with a prefix of $P[k..p]$ takes $O(x \log \sigma)$ time).

If $x = p - k + 1$, then $P[k..p]$ is fully matched with prefixes of some text segments and the first node obtained by traversing further down in Δ_f will be the locus node u_k of $P[k..p]$. Now the locus node of $P[(k+\alpha)..p]$ can be computed as follows: First reach the node u'_k by chasing the α-sampled suffix link from u_k, then the locus node $u_{k+\alpha}$ of $P[(k+\alpha)..p]$ is given by the node closest to the root in the path from u'_k to root, which is at least $x-\alpha$ distance away from root. The locus node $u_{k+2\alpha}$ of $P[(k+2\alpha)..p]$ can be obtained similarly by further chasing the α-sampled suffix link of $u_{k+\alpha}$ and so on. The total number of α-sampled suffix links chased will be $O(p/\alpha + 1)$ and given the locus node of a pattern, its suffix range can be obtained in constant time. Hence, the total time is bounded by $O(p \log \sigma + \log^{1+\epsilon} n)$.

If $x < p - k + 1$, then $P[k..p]$ is not fully matched. Let v_k be the node closest to the root such that x is the length of the longest common prefix of $P[k..p]$ and $path(v_k)$. Then, we chase the α-sampled suffix link from v_k and reach a node v'_k, and continue matching from the position $x - \alpha$ distance away from root in $path(v'_k)$. Thus, we continue the matching of $P[k..p]$, $P[(k+\alpha)..p]$, $P[(k+2\alpha)..p]$, and and so on by chasing the α-sampled suffix links. Each time we match a small portion of P (that portion will not be matched again) of length say ℓ' in $O(\ell' + \log^{1+\epsilon} n)$ time and the number of such portions (number of α-sampled suffix links chased) is $O(p/\alpha + 1)$. Hence, the time for computing the locus node (if exists) and the corresponding suffix ranges of all α-sampled suffixes of $P[k..p]$ is given by $O(p \log \sigma + \log^{1+\epsilon} n(p/\alpha + 1))$. Thus, the total time for computing the suffix ranges of all suffixes of $P[1..p]$ (i.e. α-sampled suffixes of $P[k..p]$, for $k = 1, 2, 3, \ldots, \alpha$) is given by $O((p \log \sigma + \log^{1+\epsilon} n) \log^{1+\epsilon} n) = O(p \log^{1+\epsilon} n \log \sigma)$, when $p > \log_\sigma n \log^\epsilon n$.

For computing the suffix range of short patterns ($p \leq \log_\sigma n \log^\epsilon n$), we maintain an $o(n)$ bits additional information on Δ_f as follows: Along the path of every $(\log^2 n)$th α-sampled suffix in Δ_f, we write the first $\log^{1+\epsilon} n$ characters explicitly, which takes only $O((n/\alpha + d)/\log^2 n) \log^{1+\epsilon} n) = o(n)$ bits extra space. Thus, to find the suffix range of a pattern P, we first compute the suffix range $[L, R]$ by considering only those suffixes whose first $\log^{1+\epsilon} n$ characters are explicitly written. This traversal will take only $O(p \log \sigma)$ time. After this, to find the exact suffix range, we need to perform a binary search only among $\log^2 n$ α-sampled suffixes on either side of the $[L, R]$ and check if the pattern P is matching with its prefix. Thus we need to retrieve only $O(\log(\log^2 n))$ substrings of T (each of length $p \leq \log_\sigma n \log^\epsilon n$). Hence, the time for computing the suffix range of P is $O(p \log \sigma + \log^{1+\epsilon} n \log\log n) = O(\log^{1+\epsilon'} n)$, where $\epsilon' \geq 2\epsilon$. Now the suffix range of each suffix of P can be computed independently and the total time is $O(p \log^{1+\epsilon'} n)$.

The query time for finding the suffix ranges of the reverse of all prefixes of P ($\overleftarrow{P[1..i]}$, for $i = 1, 2, 3, \ldots, p$) in Δ_r can be analyzed in the same fashion. □

3 Matching with Wildcards in Compressed Text

A wildcard is a character which can match with any character in an alpahbet Σ, and it is denoted by ϕ in this paper. Given a text $T = T_1 \phi^{k_1} T_2 \phi^{k_2} \cdots \phi^{k_d} T_{d+1}$ of total length n, where each T_i are strings drawn over the alphabet Σ of size σ (assume $\sigma = O(poly \log(n))$), and ϕ^j denotes a string of j consecutive wildcards, our objective is to construct an index for T for locating all the occurrences of an online pattern P of length p efficiently. The general way to approach this problem is to categorize the occurrences of P in T into the following 3 types [17,18,19] and build a dedicated data structure for handling each type.

Type-1: P matching a substring of T with no wildcard groups;
Type-2: P matching a substring of T with exactly 1 wildcard group;
Type-3: P matching a substring of T with 2 or more wildcard groups.

We also follow the same approach, however we reduce the index space by making use of the *same* data structure (which is an FM-index of T) for handling all the three types of occurrences. We need auxiliary structures in type-2 and type-3 occurrences, but the space of those structures is bounded by $O(d \log n) + o(n)$ bits.

3.1 Type-1 Matching

In type-1 matching, we are looking for exact matches of P within T (without matching any wildcards). This case is the same as the basic text indexing problem, and by using FM-index [6], we have the following lemma.

Lemma 2. *By maintaining an $nH_h + o(n \log \sigma)$ bits index, all the type-1 occurrences can be reported in $O(p + occ_1 \log^{1+\epsilon} n)$ time, where occ_1 is the number of type-1 occurrences.*

3.2 Type-2 Matching

In type-2 matching, we are looking for each substring S within T which matches with P, such that S contains exactly one wildcard group (could be partial). We further divide such an occurrence S into the following 3 subcases:

1. S is a concatenation of a suffix of T_j , ϕ^{k_j} and a prefix of T_{j+1}. For this, we need to find all those text segments T_j such that $\overleftarrow{P[1..i]}$ is a prefix of $\overleftarrow{T_j}$ and $P[i+k_j+1..p]$ is a prefix of T_{j+1} for $1 \leq i < (i+k_j+1) \leq p$. If these conditions are satisfied, then P matches at position $z-i$ in T, where z is the starting position of jth wildcard group within T.
2. S is a concatenation of a portion of ϕ^{k_j} and a prefix of T_{j+1}. For this, we need to find all those text segments T_{j+1} such that $P[i+1..p]$ is a prefix of T_{j+1} and $1 \leq i \leq k_j$. Then, the matching position of P within T is $z+k_j-i$.
3. S is a concatenation of a suffix of T_j and a portion of ϕ^{k_j}. For this, we need to find all those text segments T_j such that $\overleftarrow{P[1..i]}$ is a prefix of $\overleftarrow{T_j}$ and $1 \leq p-i \leq k_j$. Then, the matching position of P within T is $z-i$.

The above conditions can be verified easily by maintaining different 2-dimensional orthogonal range reporting structures, where each structure corresponds to a distinct wildcard group lengths. Let RS_β represents the orthogonal range searching structure which links two text segments on either side of all wildcard groups of length β, then RS_β contains all those points (x_j, y_j) such that the (lexicographically) x_jth (α-sampled) suffix in Δ_r is $\overleftarrow{T_j}$, the (lexicographically) y_jth (α-sampled) suffix in Δ_f is T_{j+1}, and $k_j = \beta$ (i.e., there are exactly β wildcard symbols between T_j and T_{j+1} in T). The number of 2-dimensional orthogonal range reporting structures is $\hat{d}$ (the number distinct wildcard group lengths) and the total number of points among all those $\hat{d}$ structures is $O(d)$.

Now, type-2 matching can be performed as follows: first we obtain the suffix ranges $[L_i^f, R_i^f]$ of $P[i..p]$ for $i = 1, 2, \ldots, p$ in Δ_f and the suffix ranges $[L_i^r, R_i^r]$ of $\overleftarrow{P[1..i]}$ for $i = 1, 2, \ldots, p$ in Δ_r in $O(p \log^{1+\epsilon'} n)$ (using Lemma 1). Then, we have:

- All case-1 occurrences can be obtained by reporting all those points in RS_β with query range $[L_i^r, R_i^r] \times [L_{(i+\beta+1)}^f, R_{(i+\beta+1)}^f]$ for all $\beta < p$ and $1 \leq i < (i+\beta+1) \leq p$.
- All case-2 occurrences can be obtained by reporting all those points in RS_β with query range $[-\infty, \infty] \times [L_{(i+1)}^f, R_{(i+1)}^f]$ for all $\beta < p$ and $1 \leq i < \beta$.
- All case-3 occurrences can be obtained by reporting all those points in RS_β with query range $[L_i^r, R_i^r] \times [-\infty, \infty]$ for all $\beta < p$ and $1 \leq p-i < \beta$.

The number of orthogonal range searching queries is $O(p \min(p, \hat{d}))$ and the total number of points among all the orthogonal range searching structures is $O(d)$. Therefore, by using an $O(d \log n)$-bit orthogonal range searching structure

by [15], the query time can be bounded by $O(p\log^{1+\epsilon'} n + p\min(p,\hat{d})\log d + occ_2 \log^{\epsilon'} d)$, where occ_2 is the number of type-2 occurrences. Moreover, P can trivially be matched at all $(k_j - p + 1)$ positions within a wildcard group of length $k_j \geq p$.

Lemma 3. *By maintaining an $nH_h + o(n\log\sigma) + O(d\log n)$ bits index, all the type-2 occurrences of P in T can be reported in $O(p(\log^{1+\epsilon'} n + min(p,\hat{d})\log d) + occ_2 \log^{\epsilon'} d)$ time, where $\epsilon' > 0$.*

3.3 Type-3 Matching

In type-3 matching, we are looking for each substring of T which matches with P and contains more than one wildcard groups. Therefore, P contains at least a whole text segment T_j. We follow a similar approach proposed by Lam et al. [17] for handling this case, where we first retrieve all those text segments T_js which are completely contained in P, and check if each such T_j can be extended for a type-3 matching. This is equivalent to the dictionary matching problem as described in Section 2.4.

The dictionary matching can be performed in $O(p(\log^{\epsilon} n + \log d) + \gamma)$ time using an $o(n\log\sigma) + O(d\log n)$ bits index, where γ is the number of dictionary matching outputs (Section 2.4) [10]. In [10], it is assumed that the text segments are stored using Ferragina-Venturini scheme [7], where the storage space is $nH_h + o(n\log\sigma)$ bits and a text substring of length ℓ can be displayed in $O(\ell/\log_\sigma n + 1)$ time. However, we cannot afford to store the text segments using this scheme as it will double the index space. Since FM-index is already stored for the type-1 and type-2 matchings, we will use the same FM-index as the storage scheme for the text collection (even though retrieval time is slower). It remains to check how the dictionary matching time will be affected by using FM-index (as a storage scheme) instead of Ferragina-Venturini scheme. For this, we consider the following two facts: (i) The time for displaying a substring of length $\ell < \log^{1+\epsilon} n$ using FM-index is $O(\log^{1+\epsilon} n)$ and that of Ferragina-Venturini scheme is $\Omega(1)$. Hence, FM-index is worse by at most a factor of $O(\log^{1+\epsilon} n)$. (ii) When $\ell \geq \log^{1+\epsilon} n$, FM-index takes $O(\ell)$ time, where as Ferragina-Venturini scheme takes $O(\ell/\log_\sigma n)$ time. Hence, FM-index is worse by a factor of $\Theta(\log_\sigma n)$. Therefore, by using FM-index as a storage scheme, the dictionary matching time can get worse by a factor $O(max(\log^{1+\epsilon} n, \log_\sigma n)) = O(\log^{1+\epsilon} n)$. We remark[1] that only the term $p\log^{\epsilon} n$ will get multiplied by $\log^{1+\epsilon} n$. Choosing $\epsilon' \geq 2\epsilon > 0$, we have the following lemma.

Lemma 4. *Dictionary matching can be performed in $O(p\log^{1+\epsilon'} n + \gamma)$ time by maintaining an $nH_h + o(n\log\sigma) + O(d\log n)$ bits index.*

[1] In the paper by Hon et al. [10], the computation of the locus of all suffixes of P in their dictionary matching index takes $O(p\log^{\epsilon} n)$ time and reporting all the occurrences takes $O(p(\log^{\epsilon} n + \log d) + \gamma)$ time. Note that the pattern matching is needed only in the first step.

To perform type-3 matching, we first use the above lemma to find all the γ occurrences of text segments within P. Corresponding to each such occurrence, we compute a pair (s, e) as follows: Let the text segment T_j has a match within P at position i (i.e. $T_j = P[i..(i + |T_j|] - 1)])$ and T_j starts at position j' in T. This match will be a part of a valid occurrence of P in T if and only if P starts at position $(j' - i + 1)$ within T. Then (s, e) for this match is given by $s = j' - i + 1$ and $e = j'$. Further, we obtain a sorted sequence of all γ pairs $(s_1, e_1), (s_2, e_2), (s_3, e_3), \ldots, (s_\gamma, e_\gamma)$ such that $s_k < s_{k+1}$ or $s_k = s_{k+1}$ and $e_k < e_{k+1}$ in $O(\gamma \log \gamma)$ time. Along with this γ pairs of values, we also maintain the suffix ranges corresponding to all suffixes of P and $\overleftarrow{P}$ in Δ_f and Δ_r respectively (using Lemma 1). Therefore, our query working space is $O((p + \gamma) \log n)$ bits.

We maintain two bit vectors $B_s[1..n]$ and $B_e[1..n]$ for marking the starting and ending positions of text segments within T, such that $B_s[i] = 1$ if $T[i] \neq \phi$ and $T[i - 1] = \phi$ or $i = 1$, else 0, and $B_e[i] = 1$ if $T[i] \neq \phi$ and $T[i + 1] = \phi$ or $i = n$, else 0. Using these two bit vectors, the staring position and ending position ($select_{B_s}(j)$ and $select_{B_e}(j)$, respectively) of any given text segment T_j can be computed in constant time (Section 2.1). Similarly the text segment T_j or wildcard group ϕ^{k_j} corresponding to a given position i in T can be computed in constant time ($j = rank_{B_s}(i)$ and $T[i]$ will be within a text segment if $i \leq select_{B_e}(j)$, else $T[i]$ will be a part of jth wildcard group). We also maintain two arrays $A_f[1..d]$ and $A_r[1..d]$, such that A_f stores the lexicographic ordering of T_j in Δ_f ($A_f[j] = k$ if T_j is the kth lexicographically smallest α-sampled suffix in Δ_f) and A_r stores the lexicographic ordering of $\overleftarrow{T}_j$ in Δ_r ($A_r[j] = k$ if $\overleftarrow{T}_j$ is the kth lexicographically smallest α-sampled suffix in Δ_r) . Thus, given the suffix range of any pattern in Δ_f, in constant time we can check if this is a prefix of given text segment using A_f. Similarly, given the suffix range of the reverse of any pattern in Δ_r, in constant time we can check if this is a prefix of given text segment using A_r. Now for a pattern P to match at position s in T (i.e. $P = T[s..(s + p - 1)]$), the following conditions should be satisfied:

1. Corresponding to each text segment T_j which is completely contained in $T[s..(s+p-1)]$, there should be a pair (s, e) with e being the staring position of the T_j within T, and $e = select_{B_s}(j)$.
2. The longest prefix of $T[s..(s+p-1)]$ without any wildcard should be a prefix of P. In other words, if $T[s] \neq \phi$ and $T[s - 1] \neq \phi$ (i.e., a prefix $P[1..x]$ of P has a match with a suffix ($\neq T_{j'}$) of a text segment $T_{j'}$), $\overleftarrow{P[1..x]}$ should be a prefix of $\overleftarrow{T}_{j'}$ (i.e., $A_r[j']$ should be within the suffix range of $\overleftarrow{T}_{j'}$ in Δ_r) , where $j' = rank_{B_s}(s)$ and $x = select_{B_e}(j') - s + 1$.
3. The longest suffix of $T[s..(s+p-1)]$ without any wildcard should be a suffix of P. In other words, if $T[s+p-1] \neq \phi$ and $T[s+p] \neq \phi$ (i.e., a suffix $P[y..p]$ of P has a match with a prefix ($\neq T_{j''}$) of a text segment $T_{j''}$), $P[y..p]$ should be a prefix of $T_{j''}$ (i.e. $A_f[j'']$ should be within the suffix range of $T_{j''}$ in Δ_f), where $j'' = rank_{B_s}(s + p - 1)$ and $y = select_{B_s}(j'') - s + 1$.

In each contiguous sublist of (s, e) with same s value, the first condition can be verified by scanning the corresponding e values and the remaining conditions can

be verified in constant time. Thus, the time for filtering all type-3 occurrences from the sorted list of (s, e) values can be bounded by $O(\gamma)$. Thus, we have the following lemma.

Lemma 5. *By maintaining an $nH_h + o(n \log \sigma) + O(d \log n)$ bits index, all the type-3 occurrences of P in T can be reported in $O(p \log^{1+\epsilon'} n + \gamma \log \gamma)$ time, where $\epsilon' > 0$. The working space required is $O((p + \gamma) \log n)$ bits.*

Combining the results of type-1 (Lemma 2), type-2 (Lemma 3), and type-3 (Lemma 5) matchings, we have the following theorem.

Theorem 1. *There exists an index of size $nH_h + o(n \log \sigma) + O(d \log n)$ bits for wildcard matching, which can answer each online query for a pattern P of length p in $O(p(\log^{1+\epsilon'} n + \min(p, \hat{d}) \log d) + occ_1 \log^{1+\epsilon'} n + occ_2 \log^{\epsilon'} d + \gamma \log \gamma)$ time, for any fixed $\epsilon' > 0$. The working space required is $O((p + \gamma) \log n)$ bits.*

References

1. Belazzougui, D.: Succinct Dictionary Matching with No Slowdown. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 88–100. Springer, Heidelberg (2010)
2. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA (1994)
3. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In: DCC, pp. 252–261 (2008)
4. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary Matching and Indexing with Errors and Don't Cares. In: STOC, pp. 91–100 (2004)
5. Ferragina, P., Manzini, G.: Indexing Compressed Text. Journal of the ACM 52(4), 552–581 (2005)
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed Representations of Sequences and Full-Text Indexes. ACM Transactions on Algorithms 3(2) (2007)
7. Ferragina, P., Venturini, R.: A Simple Storage Scheme for Strings Achieving Entropy Bounds. Theoretical Computer Science 372(1), 115–121 (2007)
8. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM Journal on Computing 35(2), 378–407 (2005)
9. Hon, W.-K., Ku, T.-H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster Compressed Dictionary Matching. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 191–200. Springer, Heidelberg (2010)
10. Hon, W.K., Lam, T.W., Shah, R., Tam, S.L., Vitter, J.S.: Compressed Index for Dictionary Matching. In: DCC, pp. 23–32 (2008)
11. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: On Entropy-Compressed Text Indexing in External Memory. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 75–89. Springer, Heidelberg (2009)
12. Kärkkäinen, J., Ukkonen, E.: Sparse Suffix Trees. In: COCOON, vol. 219–230 (1996)
13. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)

14. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
15. Nekrich, Y.: Orthogonal Range Searching in Linear and Almost-Linear Space. Computational Geometry 42(4), 342–351 (2009)
16. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. ACM Transactions on Algorithms 3(4) (2007)
17. Lam, T.-W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Space Efficient Indexes for String Matching with Don't Cares. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 846–857. Springer, Heidelberg (2007)
18. Tam, A., Wu, E., Lam, T.-W., Yiu, S.-M.: Succinct Text Indexing with Wildcards. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 39–50. Springer, Heidelberg (2009)
19. Thachuk, C.: Succincter Text Indexing with Wildcards. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 27–40. Springer, Heidelberg (2011)
20. Weiner, P.: Linear Pattern Matching Algorithms. In: FOCS, pp. 1–11 (1973)
21. Ziv, J., Lempel, A.: Compression of Individual Sequences via Variable Length Coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Fast q-gram Mining on SLP Compressed Strings*

Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Department of Informatics, Kyushu University
{keisuke.gotou,bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

Abstract. We present simple and efficient algorithms for calculating q-gram frequencies on strings represented in compressed form, namely, as a straight line program (SLP). Given an SLP of size n that represents string T, we present an $O(qn)$ time and space algorithm that computes the occurrence frequencies of *all* q-grams in T. Computational experiments show that our algorithm and its variation are practical for small q, actually running faster on various real string data, compared to algorithms that work on the uncompressed text. We also discuss applications in data mining and classification of string data, for which our algorithms can be useful.

1 Introduction

A major problem in managing large scale string data is its sheer size. Therefore, such data is normally stored in compressed form. In order to utilize or analyze the data afterwards, the string is usually decompressed, where we must again confront the size of the data. To cope with this problem, algorithms that work directly on compressed representations of strings without explicit decompression have gained attention, especially for the string pattern matching problem [1] where algorithms on compressed text can actually run faster than algorithms on the uncompressed text [23]. There has been growing interest in what problems can be efficiently solved in this kind of setting [17,8].

Since there exist many different text compression schemes, it is not realistic to develop different algorithms for each scheme. Thus, it is common to consider algorithms on texts represented as *straight line programs* (SLPs) [12,17,8]. An SLP is a context free grammar in the Chomsky normal form that derives a single string. Texts compressed by any grammar-based compression algorithms (e.g. [21,15]) can be represented as SLPs, and those compressed by the LZ-family (e.g. [24,25]) can be quickly transformed to SLPs [22]. Recently, even *compressed self-indices* based on SLPs have appeared [6], and SLPs are a promising representation of compressed strings for conducting various operations.

In this paper, we explore a more advanced field of application for compressed string processing: mining and classification on string data given in compressed form. Discovering useful patterns hidden in strings as well as automatic and accurate classification of strings into various groups, are important problems in

* This work was supported by KAKENHI 22680014 (HB).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 278–289, 2011.
© Springer-Verlag Berlin Heidelberg 2011

the field of data mining and machine learning with many applications. As a first step toward *compressed* string mining and classification, we consider the problem of finding the occurrence frequencies for all q-grams contained in a given string. q-grams are important features of string data, widely used for this purpose in many fields such as text and natural language processing, and bioinformatics.

In [10], an $O(|\Sigma|^2 n^2)$-time $O(n^2)$-space algorithm for finding the *most frequent* 2-gram from an SLP of size n representing text T over alphabet Σ was presented. In [6], it is mentioned that the most frequent 2-gram can be found in $O(|\Sigma|^2 n \log n)$-time and $O(n \log |T|)$-space, if the SLP is pre-processed and a self-index is built. It is possible to extend these two algorithms to handle q-grams for $q > 2$, but would respectively require $O(|\Sigma|^q qn^2)$ and $O(|\Sigma|^q qn \log n)$ time, since they must essentially enumerate and count the occurrences of all substrings of length q, regardless of whether the q-gram occurs in the string. Note also that any algorithm that works on the uncompressed text T requires exponential time in the worst case, since $|T|$ can be as large as $O(2^n)$.

The main contribution of this paper is an $O(qn)$ time and space algorithm that computes the occurrence frequencies for *all* q-grams in the text, given an SLP of size n representing the text. Our new algorithm solves the more general problem and greatly improves the computational complexity compared to previous work. We also conduct computational experiments on various real texts, showing that when q is small, our algorithm and its variation actually run faster than algorithms that work on the uncompressed text.

Our algorithms have profound applications in the field of string mining and classification, and several applications and extensions are discussed. For example, our algorithm leads to an $O(q(n_1+n_2))$ time algorithm for computing the q-gram spectrum kernel [16] between SLP compressed texts of size n_1 and n_2. It also leads to an $O(qn)$ time algorithm for finding the optimal q-gram (or emerging q-gram) that discriminates between two sets of SLP compressed strings, when n is the total size of the SLPs.

Related Work. There exist many works on *compressed text indices* [20], but the main focus there is on fast search for a *given* pattern. The compressed indices basically replace or simulate operations on uncompressed indices using a smaller data structure. Indices are important for efficient string processing, but note that simply replacing the underlying index used in a mining algorithm will generally increase time complexities of the algorithm due to the extra overhead required to access the compressed index. On the other hand, our approach is a new mining algorithm which exploits characteristics of the compressed representation to achieve faster running times.

Several algorithms for finding characteristic sequences from compressed texts have been proposed, e.g., finding the longest common substring of two strings [19], finding all palindromes [19], finding most frequent substrings [10], and finding the longest repeating substring [10]. However, none of them have reported results of computational experiments, implying that this paper is the first to show the practical usefulness of a compressed text mining algorithm.

Algorithm 1. Calculating $vOcc(X_i)$ for all $1 \leq i \leq n$

Input: SLP $\mathcal{T} = \{X_i\}_{i=1}^n$ representing string T.
Output: $vOcc(X_i)$ for all $1 \leq i \leq n$

```
1  vOcc[X_n] ← 1;
2  for i ← 1 to n − 1 do  vOcc[X_i] ← 0;
3  for i ← n to 2 do
4      if  X_i = X_ℓ X_r then
5          vOcc[X_ℓ] ← vOcc[X_ℓ] + vOcc[X_i]; vOcc[X_r] ← vOcc[X_r] + vOcc[X_i];
```

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. For any integer $q > 0$, an element of Σ^q is called an *q-gram*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of a string T that begins at position i and ends at position j is denoted by $T[i:j]$ for $1 \leq i \leq j \leq |T|$. For convenience, let $T[i:j] = \varepsilon$ if $j < i$.

For a string T and integer $q \geq 0$, let $pre(T, q)$ and $suf(T, q)$ represent respectively, the length-q prefix and suffix of T. That is, $pre(T, q) = T[1 : \min(q, |T|)]$ and $suf(T, q) = T[\max(1, |T| - q + 1) : |T|]$.

For any strings T and P, let $Occ(T, P)$ be the set of occurrences of P in T, i.e., $Occ(T, P) = \{k > 0 \mid T[k : k + |P| - 1] = P\}$. The number of elements $|Occ(T, P)|$ is called the *occurrence frequency* of P in T.

2.1 Straight Line Programs

A *straight line program* (*SLP*) $\mathcal{T}$ is a sequence of assignments $X_1 = expr_1, X_2 = expr_2, \ldots, X_n = expr_n$, where each X_i is a variable and each $expr_i$ is an expression, where $expr_i = a$ ($a \in \Sigma$), or $expr_i = X_\ell X_r$ ($\ell, r < i$). Let $val(X_i)$ represent the string derived from X_i. When it is not confusing, we identify a variable X_i with $val(X_i)$. Then, $|X_i|$ denotes the length of the string X_i derives. An SLP $\mathcal{T}$ *represents* the string $T = val(X_n)$. The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$. (See Fig. 1)

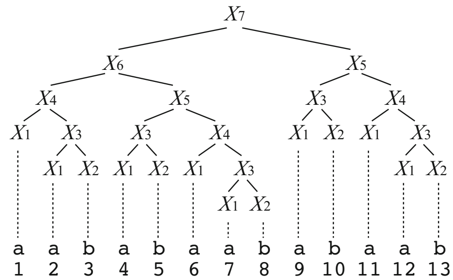


Fig. 1. The derivation tree of SLP $\mathcal{T} = \{X_i\}_{i=1}^7$ with $X_1 = \mathtt{a}$, $X_2 = \mathtt{b}$, $X_3 = X_1X_2$, $X_4 = X_1X_3$, $X_5 = X_3X_4$, $X_6 = X_4X_5$, and $X_7 = X_6X_5$, representing string $T = val(X_7) = \mathtt{aababaababaab}$

The substring intervals of T that each variable derives can be defined recursively as follows: $itv(X_n) = \{[1 : |T|]\}$, and $itv(X_i) = \{[u + |X_\ell| : v] \mid X_k = X_\ell X_i, [u : v] \in itv(X_k)\} \cup \{[u : u + |X_i| - 1] \mid X_k = X_i X_r, [u : v] \in itv(X_k)\}$ for

Algorithm 2. A naïve algorithm for computing q-gram frequencies

Input: string T, integer $q \geq 1$
Report: $(P, |Occ(T,P)|)$ for all $P \in \Sigma^q$ where $Occ(T,P) \neq \emptyset$.

```
1 S ← ∅; // empty associative array
2 for i ← 1 to |T| − q + 1 do
3     qgram ← T[i : i + q − 1];
4     if qgram ∈ keys(S) then S[qgram] ← S[qgram] + 1;
5     else S[qgram] ← 1; // new q-gram
6 for qgram ∈ keys(S) do Report (qgram, S[qgram])
```

$i < n$. For example, $itv(X_5) = \{[4:8], [9:13]\}$ in Fig. 1. Considering the transitive reduction of set inclusion, the intervals $\cup_{i=1}^{n} itv(X_i)$ naturally form a binary tree (the derivation tree). Let $vOcc(X_i) = |itv(X_i)|$ denote the number of times a variable X_i occurs in the derivation of T. $vOcc(X_i)$ for all $1 \leq i \leq n$ can be computed in $O(n)$ time by a simple iteration on the variables, since $vOcc(X_n) = 1$ and for $i < n$, $vOcc(X_i) = \sum\{vOcc(X_k) \mid X_k = X_\ell X_i\} + \sum\{vOcc(X_k) \mid X_k = X_i X_r\}$. (See Algorithm 1).

2.2 Suffix Arrays and LCP Arrays

The suffix array SA [18] of any string T is an array of length $|T|$ such that $SA[i] = j$, where $T[j : |T|]$ is the i-th lexicographically smallest suffix of T. The *lcp* array of any string T is an array of length $|T|$ such that $LCP[i]$ is the length of the longest common prefix of $T[SA[i-1] : |T|]$ and $T[SA[i] : |T|]$ for $2 \leq i \leq |T|$, and $LCP[1] = 0$. The suffix array for any string of length $|T|$ can be constructed in $O(|T|)$ time (e.g. [11]) assuming an integer alphabet. Given the text and suffix array, the lcp array can also be calculated in $O(|T|)$ time [13].

3 Algorithm

3.1 Computing q-gram Frequencies on Uncompressed Strings

We describe two algorithms (Algorithm 2 and Algorithm 3) for computing the q-gram frequencies of a given uncompressed string T.

A naïve algorithm for computing the q-gram frequencies is given in Algorithm 2. The algorithm constructs an associative array, where keys consist of q-grams, and the values correspond to the occurrence frequencies of the q-grams. The time complexity depends on the implementation of the associative array, but requires at least $O(q|T|)$ time since each q-gram is considered explicitly, and the associative array is accessed $O(|T|)$ times: e.g. $O(q|T| \log |\Sigma|)$ time and $O(q|T|)$ space using a simple trie.

The q-gram frequencies of string T can be calculated in $O(|T|)$ time using suffix array SA and lcp array LCP, as shown in Algorithm 3. For each $1 \leq i \leq |T|$, the suffix $SA[i]$ represents an occurrence of q-gram $T[SA[i] : SA[i] + q - 1]$, if the

Algorithm 3. A linear time algorithm for computing q-gram frequencies

Input: string T, integer $q \geq 1$
Report: $(i, |Occ(T, P)|)$ for all $P \in \Sigma^q$ and some position $i \in Occ(T, P)$.

```
1 SA ← SUFFIXARRAY(T); LCP ← LCPARRAY(T, SA); count ← 1;
2 for i ← 2 to |T| + 1 do
3     if i = |T| + 1 or LCP[i] < q then
4         if count > 0 then Report (SA[i − 1], count); count ← 0;
5     if i ≤ |T| and SA[i] ≤ |T| − q + 1 then count ← count + 1;
```

suffix is long enough, i.e. $SA[i] \leq |T| - q + 1$. The key is that since the suffixes are lexicographically sorted, intervals on the suffix array where the values in the lcp array are at least q represent occurrences of the same q-gram. The algorithm runs in $O(|T|)$ time, since SA and LCP can be constructed in $O(|T|)$. The rest is a simple $O(|T|)$ loop. A technicality is that we encode the output for a q-gram as one of the positions in the text where the q-gram occurs, rather than the q-gram itself. This is because there can be a total of $O(|T|)$ different q-grams, and if we output them as length-q strings, it would require at least $O(q|T|)$ time.

3.2 Computing q-gram Frequencies on SLP

We now describe the core idea of our algorithms, and explain two variations which utilize variants of the two algorithms for uncompressed strings presented in Section 3.1. For $q = 1$, the 1-gram frequencies are simply the frequencies of the alphabet and the output is $(a, \sum\{vOcc(X_i) \mid X_i = a\})$ for each $a \in \Sigma$, which takes only $O(n)$ time. For $q \geq 2$, we make use of Lemma 1 below. The idea is similar to the *mk Lemma* [5], but the statement is more specific.

Lemma 1. *Let $\mathcal{T} = \{X_i\}_{i=1}^n$ be an SLP that represents string T. For an interval $[u : v]$ $(1 \leq u < v \leq |T|)$, there exists exactly one variable $X_i = X_\ell X_r$ such that for some $[u' : v'] \in itv(X_i)$, the following holds: $[u : v] \subseteq [u' : v']$, $u \in [u' : u' + |X_\ell| - 1] \in itv(X_\ell)$ and $v \in [u' + |X_\ell| : v'] \in itv(X_r)$.*

Proof. Consider length 1 intervals $[u : u]$ and $[v : v]$ corresponding to leaves in the derivation tree. X_i corresponds to the lowest common ancestor of these intervals in the derivation tree. □

From Lemma 1, each occurrence of a q-gram $(q \geq 2)$ represented by some length-q interval of T, corresponds to a single variable $X_i = X_\ell X_r$, and is split in two by intervals corresponding to X_ℓ and X_r. On the other hand, consider all length-q intervals that correspond to a given variable. Counting the frequencies of the q-grams they represent, and summing them up for all variables give the frequencies of all q-grams of T.

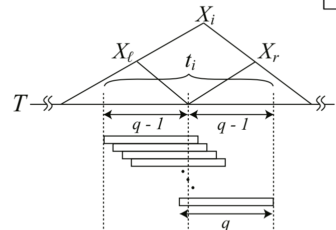


Fig. 2. Length-q intervals corresponding to $X_i = X_\ell X_r$

For variable $X_i = X_\ell X_r$, let $t_i = \mathit{suf}(X_\ell, q-1)\mathit{pre}(X_r, q-1)$. Then, all q-grams represented by length q intervals that correspond to X_i are those in t_i. (Fig. 2). If we obtain the frequencies of all q-grams in t_i, and then multiply each frequency by $\mathit{vOcc}(X_i)$, we obtain frequencies for the q-grams occurring in all intervals derived by X_i. It remains to sum up the q-gram frequencies of t_i for all $1 \leq i \leq n$. We can regard it as obtaining the *weighted* q-gram frequencies in the set of strings $\{t_1, \ldots, t_n\}$, where each q-gram in t_i is weighted by $\mathit{vOcc}(X_i)$.

We further reduce this problem to a weighted q-gram frequency problem for a single string z as in Algorithm 4. String z is constructed by concatenating t_i such that $q \leq |t_i| \leq 2(q-1)$, and the weights of q-grams starting at each position in z is held in array w. On line 8, 0's instead of $\mathit{vOcc}(X_i)$ are appended to w for the last $q-1$ values corresponding to t_i. This is to avoid counting unwanted q-grams that are generated by the concatenation of t_i to z on line 6, which are not substrings of each t_i. The weighted q-gram frequency problem for a single string (Line 9) can be solved with a slight modification of Algorithm 2 or 3. The modified algorithms are shown respectively in Algorithms 5 and 6.

Theorem 1. *Given an SLP $\mathcal{T} = \{X_i\}_{i=1}^n$ of size n representing a string T, the q-gram frequencies of T can be computed in $O(qn)$ time for any $q > 0$.*

Proof. Consider Algorithm 4. The correctness is straightforward from the above arguments, so we consider the time complexity. Line 1 can be computed in $O(n)$ time. Line 2 can be computed in $O(qn)$ time by a simple dynamic programming. For $\mathit{pre}()$: If $X_i = a$ for some $a \in \Sigma$, then $\mathit{pre}(X_i, q-1) = a$. If $X_i = X_\ell X_r$ and $|X_\ell| \geq q-1$, then $\mathit{pre}(X_i, q-1) = \mathit{pre}(X_\ell, q-1)$. If $X_i = X_\ell X_r$ and $|X_\ell| < q-1$, then $\mathit{pre}(X_i, q-1) = \mathit{pre}(X_\ell, q-1)\mathit{pre}(X_r, q-1-|X_\ell|)$. The strings $\mathit{suf}()$ can be computed similarly. The computation amounts to copying $O(q)$ characters for each variable, and thus can be done in $O(qn)$ time. For the loop at line 4, since the length of string t_i appended to z, as well as the number of elements appended to w is at most $2(q-1)$ in each loop, the total time complexity is $O(qn)$. Finally, since the length of z and w is $O(qn)$, line 9 can be calculated in $O(qn)$ time using the weighted version of Algorithm 3 (Algorithm 6). □

Note that the time complexity for using the weighted version of Algorithm 2 for line 9 of Algorithm 4 would be at least $O(q^2 n)$: e.g. $O(q^2 n \log |\Sigma|)$ time and $O(q^2 n)$ space using a trie.

4 Applications and Extensions

We showed that for an SLP $\mathcal{T}$ of size n representing string T, q-gram frequency problems on T can be reduced to *weighted* q-gram frequency problems on a string z of length $O(qn)$, which can be much shorter than T. This idea can further be applied to obtain efficient compressed string processing algorithms for interesting problems which we briefly introduce below.

Algorithm 4. Calculating q-gram frequencies of an SLP for $q \geq 2$

Input: SLP $\mathcal{T} = \{X_i\}_{i=1}^n$ representing string T, integer $q \geq 2$.
Report: all q-grams and their frequencies which occur in T.

1 Calculate $vOcc(X_i)$ for all $1 \leq i \leq n$;
2 Calculate $pre(X_i, q-1)$ and $suf(X_i, q-1)$ for all $1 \leq i \leq n-1$;
3 $z \leftarrow \varepsilon$; $w \leftarrow []$;
4 **for** $i \leftarrow 1$ **to** n **do**
5   **if** $X_i = X_\ell X_r$ **and** $|X_i| \geq q$ **then**
6     $t_i = suf(X_\ell, q-1)pre(X_r, q-1)$; z.append(t_i);
7   **for** $j \leftarrow 1$ **to** $|t_i| - q + 1$ **do** w.append($vOcc(X_i)$);
8   **for** $j \leftarrow 1$ **to** $q - 1$ **do** w.append(0);

9 **Report** q-gram frequencies in z, where each q-gram $z[i : i+q-1]$ is *weighted* by $w[i]$.

Algorithm 5. A variant of Algorithm 2 for weighted q-gram frequencies

Input: string T, array of integers w of length $|T|$, integer $q \geq 1$
Report: $(P, \sum_{i \in Occ(T,P)} w[i])$ for all $P \in \Sigma^q$ where $\sum_{i \in Occ(T,P)} w[i] > 0$.

1 $\mathbf{S} \leftarrow \emptyset$; `// empty associative array`
2 **for** $i \leftarrow 1$ **to** $|T| - q + 1$ **do**
3   $qgram \leftarrow T[i : i+q-1]$;
4   **if** $qgram \in$ keys($\mathbf{S}$) **then** $\mathbf{S}[qgram] \leftarrow \mathbf{S}[qgram] + w[i]$;
5   **else if** $w[i] > 0$ **then** $\mathbf{S}[qgram] \leftarrow w[i]$; `// new q-gram`

6 **for** $qgram \in$ keys($\mathbf{S}$) **do Report** $(qgram, \mathbf{S}[qgram])$

4.1 *q*-gram Spectrum Kernel

A string kernel is a function that computes the inner product between two strings which are mapped to some feature space. It is used when classifying string or text data using methods such as Support Vector Machines (SVMs), and is usually the dominating factor in the time complexity of SVM learning and classification. A q-gram spectrum kernel [16] considers the feature space of q-grams. For string T, let $\phi_q(T) = (|Occ(T,p)|)_{p \in \Sigma^q}$. The kernel function is defined as $K_q(T_1, T_2) = \langle \phi_q(T_1), \phi_q(T_2) \rangle = \sum_{p \in \Sigma^q} |Occ(T_1, p)||Occ(T_2, p)|$. The calculation of the kernel function amounts to summing up the product of occurrence frequencies in strings T_1 and T_2 for all q-grams which occur in both T_1 and T_2. This can be done in $O(|T_1|+|T_2|)$ time using suffix arrays. For two SLPs $\mathcal{T}_1$ and $\mathcal{T}_2$ of size n_1 and n_2 representing strings T_1 and T_2, respectively, the q-gram spectrum kernel $K_q(T_1, T_2)$ can be computed in $O(q(n_1 + n_2))$ time by a slight modification of our algorithm.

4.2 Optimal Substring Patterns of Length *q*

Given two sets of strings, finding string patterns that are frequent in one set and not in the other, is an important problem in string data mining, with many problem formulations and the types of patterns to be considered, e.g.: in Bioinformatics [3], Machine Learning (optimal patterns [2]), and more recently KDD

Algorithm 6. A variant of Algorithm 3 for weighted q-gram frequencies

Input: string T, array of integers w of length $|T|$, integer $q \geq 1$
Output: $(i, \sum_{i \in Occ(T,P)} w[i])$ for all $P \in \Sigma^q$ where $\sum_{i \in Occ(T,P)} w[i] > 0$ and some position $i \in Occ(T, P)$.

```
1 SA ← SUFFIXARRAY(T); LCP ← LCPARRAY(T, SA); count ← 1;
2 for i ← 2 to |T| + 1 do
3     if i = |T| + 1 or LCP[i] < q then
4         if count > 0 then Report (SA[i − 1], count); count ← 0;
5     if i ≤ |T| and SA[i] ≤ |T| − q + 1 then count ← count + w[SA[i]];
```

(emerging patterns [4]). A simple optimal q-gram pattern discovery problem can be defined as follows: Let $\mathbf{T_1}$ and $\mathbf{T_2}$ be two multisets of strings. The problem is to find the q-gram p which gives the highest (or lowest) score according to some scoring function that depends only on $|\mathbf{T_1}|$, $|\mathbf{T_2}|$, and the number of strings respectively in $\mathbf{T_1}$ and $\mathbf{T_2}$ for which p is a substring. For uncompressed strings, the problem can be solved in $O(N)$ time, where N is the total length of the strings in both $\mathbf{T_1}$ and $\mathbf{T_2}$, by applying the algorithm of [9] to two sets of strings. For the SLP compressed version of this problem, the input is two multisets of SLPs, each representing strings in $\mathbf{T_1}$ and $\mathbf{T_2}$. If n is the total number of variables used in all of the SLPs, the problem can be solved in $O(qn)$ time.

4.3 Different Lengths

The ideas in this paper can be used to consider all substrings of length *not only* q, but *all lengths up-to* q, with some modifications. For the applications discussed above, although the number of such substrings increases to $O(q^2 n)$, the $O(qn)$ time complexity can be maintained by using standard techniques of suffix arrays [7,13]. This is because there exist only $O(qn)$ substring with distinct frequencies (corresponding to nodes of the suffix tree), and the computations of the extra substrings can be summarized with respect to them.

5 Computational Experiments

We implemented 4 algorithms (NMP, NSA, SMP, SSA) that count the frequencies of all q-grams in a given text. NMP (Algorithm 2) and NSA (Algorithm 3) work on the uncompressed text. SMP (Algorithm 4 + Algorithm 5) and SSA (Algorithm 4 + Algorithm 6) work on SLPs. The algorithms were implemented using the C++ language. We used `std::map` from the Standard Template Library (STL) for the associative array implementation.[1] For constructing suffix arrays, we used the divsufsort library[2] developed by Yuta Mori. This implementation is

[1] We also used `std::hash_map` but omit the results due to lack of space. Choosing the hashing function to use is difficult, and we note that its performance was unstable and sometimes very bad when varying q.

[2] http://code.google.com/p/libdivsufsort/

not linear time in the worst case, but has been empirically shown to be one of the fastest implementations on various data.

All computations were conducted on a Mac Xserve (Early 2009) with 2 x 2.93GHz Quad Core Xeon processors and 24GB Memory, only utilizing a single process/thread at once. The program was compiled using the GNU C++ compiler (`g++`) 4.2.1 with the `-fast` option for optimization. The running times are measured in seconds, starting from after reading the uncompressed text into memory for NMP and NSA, and after reading the SLP that represents the text into memory for SMP and SSA. Each computation is repeated at least 3 times, and the average is taken.

5.1 Fibonacci Strings

The i th Fibonacci string F_i can be represented by the following SLP: $X_1 = \mathtt{b}$, $X_2 = \mathtt{a}$, $X_i = X_{i-1}X_{i-2}$ for $i > 2$, and $F_i = val(X_i)$. Fig. 3 shows the running times on Fibonacci strings $F_{20}, F_{25}, \ldots, F_{95}$, for $q = 50$. Although this is an extreme case since Fibonacci strings can be exponentially compressed, we can see that SMP and SSA that work on the SLP are clearly faster than NMP and NSA which work on the uncompressed string.

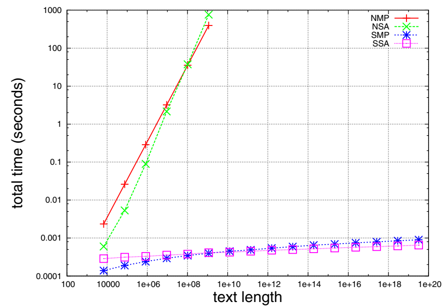


Fig. 3. Running times of NMP, NSA, SMP, SSA on Fibonacci strings for $q = 50$

5.2 Pizza and Chili Corpus

We also applied the algorithms on texts XML, DNA, ENGLISH, and PROTEINS, with sizes 50MB, 100MB, and 200MB, obtained from the Pizza & Chili Corpus[3]. We used RE-PAIR [15] to obtain SLPs for this data.

Table 1 shows the running times for all algorithms and data, where q is varied from 2 to 10. We see that for all corpora, SMP and SSA running on SLPs are actually faster than NMP and NSA running on uncompressed text, when q is small. Furthermore, SMP is faster than SSA when q is smaller. Interestingly for XML, the SLP versions are faster even for q up to 9.

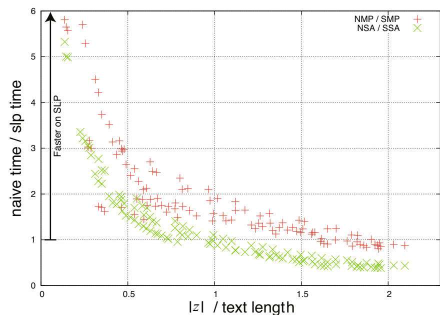


Fig. 4. Time ratios NMP/SMP and NSA/SSA plotted against ratio $|z|/|T|$

[3] http://pizzachili.dcc.uchile.cl/texts.html

Table 1. Running times in seconds for data from the Pizza & Chili Corpus. Bold numbers represent the fastest time for each data and q. Times for SMP and SSA are prefixed with ▷, if they become fastest when all algorithms start from the SLP representation, i.e., NMP and NSA require time for decompressing the SLP (denoted by decompression time). The bold horizontal lines show the boundary where $|z|$ in Algorithm 4 exceeds the uncompressed text length.

XML

	50MB					100MB					200MB				
	SLP Size:				2,702,383	SLP Size:				5,059,578	SLP Size:				9,541,590
	decompression time:				0.82 secs	decompression time:				1.73 secs	decompression time:				3.52 secs
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	8,106,861	5.9	9.8	**1.1**	2.0	15,178,446	12.0	21.0	**2.1**	4.3	28,624,482	24.7	46.9	**4.3**	8.9
3	13,413,565	13.0	9.8	**2.5**	3.2	25,160,162	27.8	21.1	**4.9**	6.8	47,504,478	58.7	46.1	**9.8**	14.3
4	18,364,951	21.0	9.8	5.7	**4.7**	34,581,658	47.2	21.3	11.3	**9.9**	65,496,619	100.3	46.2	22.5	**20.0**
5	22,873,060	28.7	9.8	10.2	**5.9**	43,275,004	63.0	21.1	20.4	**12.5**	82,321,682	139.4	46.2	40.1	**25.1**
6	27,032,514	35.2	9.8	14.9	**7.1**	51,354,178	77.1	21.0	29.6	**14.8**	98,124,580	172.4	46.3	59.4	**30.2**
7	30,908,898	40.0	9.8	19.4	**8.2**	58,935,352	87.4	21.1	38.9	**16.9**	113,084,186	197.7	46.8	78.5	**34.9**
8	34,559,523	44.3	9.8	26.0	**9.3**	66,104,075	97.5	21.1	52.5	**19.1**	127,316,007	218.3	46.3	103.9	**39.9**
9	37,983,150	49.0	**9.8**	31.0	▷ 10.1	72,859,310	105.3	21.1	60.9	**20.9**	140,846,749	234.6	46.3	124.7	**44.1**
10	41,253,257	52.5	**9.9**	35.8	11.2	79,300,797	115.3	**21.2**	72.2	▷ 22.7	153,806,891	253.6	**46.3**	148.8	▷ 48.8

DNA

	50MB					100MB					200MB				
	SLP Size:				6,406,324	SLP Size:				12,233,978	SLP Size:				23,171,463
	decompression time:				1.23 secs	decompression time:				2.54 secs	decompression time:				5.21 secs
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	19,218,924	2.2	13.7	**1.9**	5.7	36,701,886	4.7	30.5	**3.9**	12.6	69,514,341	9.8	70.0	**8.0**	26.1
3	32,030,826	4.4	13.7	**3.0**	8.6	61,169,030	9.1	30.5	**5.8**	18.6	115,856,038	18.7	70.1	**11.8**	38.8
4	44,833,624	6.5	13.7	**4.5**	12.3	85,624,856	13.4	30.5	**8.9**	25.3	162,182,697	28.0	70.0	**17.6**	52.9
5	57,554,843	8.6	13.8	**6.7**	15.5	109,976,706	17.8	30.5	**13.1**	32.3	208,371,656	37.0	69.9	**26.3**	67.9
6	69,972,618	11.1	13.7	**10.1**	19.0	133,890,719	23.3	31.0	**19.8**	40.0	253,939,731	47.6	70.2	**39.5**	86.6
7	81,771,222	15.3	**13.6**	▷ 14.7	23.0	156,832,841	31.0	30.5	**28.6**	49.3	298,014,802	63.2	69.9	**56.1**	104.5
8	92,457,893	21.1	**13.6**	22.9	27.3	177,888,984	42.2	**30.5**	44.9	58.5	338,976,517	85.4	**69.9**	88.5	126.3
9	101,852,490	33.0	**13.7**	42.8	31.4	196,656,282	65.7	**30.4**	81.5	67.5	375,928,060	132.1	**69.9**	159.3	147.9
10	109,902,230	56.5	**13.7**	65.9	34.9	213,075,531	113.2	**30.5**	129.2	75.9	408,728,193	226.0	**69.9**	248.4	166.3

ENGLISH

	50MB					100MB					200MB				
	SLP Size:				4,861,619	SLP Size:				10,063,953	SLP Size:				18,945,126
	decompression time:				1.15 secs	decompression time:				2.43 secs	decompression time:				5.07 secs
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	14,584,329	5.7	13.1	**1.9**	4.5	30,191,214	11.5	28.2	**4.2**	10.3	56,834,703	23.5	64.2	**8.5**	21.7
3	24,230,676	11.4	13.0	**4.0**	7.4	50,196,054	23.8	28.2	**8.3**	16.8	94,552,062	50.3	65.5	**16.5**	34.9
4	33,655,433	20.0	12.9	**8.2**	9.9	69,835,185	42.1	28.2	**17.6**	22.1	131,758,513	89.7	64.2	**34.1**	45.8
5	42,640,982	33.1	12.9	16.1	**12.7**	88,711,756	72.6	**28.2**	35.1	▷ 28.6	167,814,701	156.9	64.2	68.2	**59.7**
6	51,061,064	49.5	**12.9**	27.1	15.5	106,583,131	111.8	**28.5**	59.7	35.3	202,293,814	240.8	**64.4**	116.1	74.3
7	58,791,311	65.1	**12.9**	40.1	18.4	123,180,654	143.6	**28.3**	88.3	42.3	234,664,404	313.7	**64.3**	173.5	90.3
8	65,777,414	79.6	**12.9**	59.1	20.8	138,382,443	176.8	**28.3**	131.3	48.5	264,668,656	385.9	**64.8**	256.7	104.5
9	71,930,623	92.7	**12.9**	74.2	23.0	152,010,306	207.8	**28.5**	166.0	54.2	291,964,684	454.6	**64.5**	335.0	118.0
10	77,261,995	105.3	**13.0**	89.7	25.1	164,021,382	235.9	**28.4**	205.2	59.8	316,387,791	521.2	**64.7**	425.3	131.4

PROTEINS

	50MB					100MB					200MB				
	SLP Size:				10,357,053	SLP Size:				18,806,316	SLP Size:				32,375,988
	decompression time:				1.67 secs	decompression time:				3.51 secs	decompression time:				7.05 secs
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	31,071,084	4.5	14.5	**4.0**	10.2	56,418,873	9.0	32.2	**7.6**	20.4	97,127,889	18.0	69.0	**13.6**	38.0
3	51,749,628	9.4	14.5	**7.6**	16.2	93,995,974	18.7	32.1	**14.1**	32.3	161,825,337	37.3	69.0	**25.5**	60.0
4	70,939,655	22.4	**14.3**	21.3	24.6	129,372,571	45.4	**32.2**	39.1	49.0	223,413,554	91.5	**69.0**	▷ 69.1	91.8
5	86,522,157	66.6	**14.4**	54.9	32.2	159,110,124	137.5	**32.2**	100.5	65.6	275,952,088	270.9	**69.4**	175.5	125.1
6	95,684,819	116.7	**14.5**	107.7	37.6	178,252,162	251.5	**32.3**	204.4	79.1	311,732,866	502.8	**69.4**	356.0	151.7
7	99,727,910	142.8	**14.5**	143.7	40.8	187,623,783	327.6	**32.4**	299.8	85.6	330,860,933	675.2	**69.7**	586.4	168.0
8	100,877,101	147.8	**14.4**	166.3	42.5	190,898,844	343.0	**32.4**	363.6	88.7	337,898,827	731.0	**69.6**	771.8	175.5
9	101,631,544	149.3	**14.4**	171.6	42.8	192,736,305	348.1	**32.4**	393.0	91.2	341,831,651	742.2	**69.7**	820.3	181.8
10	102,636,144	150.5	**14.4**	178.6	43.4	195,044,390	350.4	**32.5**	404.2	93.1	346,403,103	747.7	**69.7**	831.9	185.8

Fig. 4 shows the same results as time ratio: NMP/SMP and NSA/ SSA, plotted against ratio: (length of z in Algorithm 4)/(length of uncompressed text). As expected, the SLP versions are basically faster than their uncompressed counterparts, when $|z|$/(text length) is less than 1, since the SLP versions run the weighted versions of the uncompressed algorithms on a text of length $|z|$. SLPs generated by other grammar based compression algorithms showed similar tendencies (data not shown).

6 Conclusion

We presented an $O(qn)$ time and space algorithm for calculating all q-gram frequencies in a string, given an SLP of size n representing the string. This solves, much more efficiently, a more general problem than considered in previous work. Computational experiments on various real texts showed that the algorithms run faster than algorithms that work on the uncompressed string, when q is small. Although larger values of q allow us to capture longer character dependencies, the dimensionality of the features increases, making the space of occurring q-grams sparse. Therefore, meaningful values of q for typical applications can be fairly small in practice (e.g. $3 \sim 6$), so our algorithms have practical value.

A future work is extending our algorithms that work on SLPs, to algorithms that work on collage systems [14]. A Collage System is a more general framework for modeling various compression methods. In addition to the simple concatenation operation used in SLPs, it includes operations for repetition and prefix/suffix truncation of variables.

This is the first paper to show the potential of the compressed string processing approach in developing efficient and *practical* algorithms for problems in the field of string mining and classification. More and more efficient algorithms for various processing of text in compressed representations are becoming available. We believe texts will eventually be stored in compressed form by default, since not only will it save space, but it will also have the added benefit of being able to conduct various computations on it more efficiently later on, when needed.

References

1. Amir, A., Benson, G.: Efficient two-dimensional compressed matching. In: Proc. Data Compression Conference (DCC 1992), pp. 279–288 (1992)
2. Arimura, H., Wataki, A., Fujino, R., Arikawa, S.: A fast algorithm for discovering optimal string patterns in large text databases. In: Richter, M.M., Smith, C.H., Wiehagen, R., Zeugmann, T. (eds.) ALT 1998. LNCS (LNAI), vol. 1501, pp. 247–261. Springer, Heidelberg (1998)
3. Brazma, A., Jonassen, I., Eidhammer, I., Gilbert, D.: Approaches to the automatic discovery of patterns in biosequences. J. Comp. Biol. 5(2), 279–305 (1998)
4. Chan, S., Kao, B., Yip, C.L., Tang, M.: Mining emerging substrings. In: Proc. 8th International Conference on Database Systems for Advanced Applications (DASFAA 2003), p. 119 (2003)
5. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., abhi shelat: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)

6. Claude, F., Navarro, G.: Self-indexed grammar-based compression. In: Fundamenta Informaticae (to appear), preliminary version: Proc. MFCS 2009, pp. 235–246 (2009)
7. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, Cambridge (1997)
8. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A unified algorithm for accelerating edit-distance computation via text-compression. In: Proc. STACS 2009, pp. 529–540 (2009)
9. Hui, L.C.K.: Color set size problem with application to string matching. In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 230–243. Springer, Heidelberg (1992)
10. Inenaga, S., Bannai, H.: Finding characteristic substring from compressed texts. In: Proc. The Prague Stringology Conference 2009, pp. 40–54 (2009)
11. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
12. Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing 4, 172–186 (1997)
13. Kasai, T., Lee, G.H., Arimura, H., Arikawa, S., Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
14. Kida, T., Shibata, Y., Takeda, M., Shinohara, A., Arikawa, S.: Collage system: A unifying framework for compressed pattern matching. Theoret. Comput. Sci. 298, 253–272 (2003)
15. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Proc. Data Compression Conference (DCC 1999), pp. 296–305 (1999)
16. Leslie, C., Eskin, E., Noble, W.S.: The spectrum kernel: A string kernel for SVM protein classification. In: Pacific Symposium on Biocomputing, vol. 7, pp. 566–575 (2002)
17. Lifshits, Y.: Processing compressed texts: A tractability border. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 228–240. Springer, Heidelberg (2007)
18. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
19. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theoret. Comput. Sci. 410(8–10), 900–913 (2009)
20. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2 (2007)
21. Nevill-Manning, C.G., Witten, I.H., Maulsby, D.L.: Compression by induction of hierarchical grammars. In: Proc. Data Compression Conference (DCC 1994), pp. 244–253 (1994)
22. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoret. Comput. Sci. 302(1–3), 211–222 (2003)
23. Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., Arikawa, S.: Speeding up pattern matching by text compression. In: Bongiovanni, G., Petreschi, R., Gambosi, G. (eds.) CIAC 2000. LNCS, vol. 1767, pp. 306–315. Springer, Heidelberg (2000)
24. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–349 (1977)
25. Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Succinct Gapped Suffix Arrays*

Luís M.S. Russo[1,2,**] and German Tischler[3]

[1] Instituto Superior Técnico-Universidade Técnica de Lisboa (IST/UTL), Av. Rovisco Pais, 1049-001 Lisboa, Portugal
luis.russo@ist.utl.pt

[2] INESC-ID, Knowledge Discovery and Bioinformatics Group, R. Alves Redol, 9,1000-029 Lisbon, Portugal

[3] Lehrstuhl für Informatik II, Universität Würzburg, Am Hubland, 97074 Würzburg, Germany
tischler@informatik.uni-wuerzburg.de

Abstract. Gapped suffix arrays (also known as bi-factor arrays) were recently presented for approximate searching under the Hamming distance. These structures can be used to find occurrences of a pattern P, where the characters inside a gap do not have to match. This paper describes a succinct representation of gapped suffix arrays.

1 Introduction

In this paper we consider gap matching, which consists in determining all the positions of a text string T where the pattern P occurs, except possibly for the characters inside a gap. We study efficient succinct representations of index structures for this problem. This type of search is relevant for bioinformatic applications. Typically such applications use large genome sequences and need to compute a large amount of queries. Genome sequences are error prune and thus exact matching retrieves only a partial amount of information.

Gapped suffix arrays were first introduced as bi-factor arrays by Peterlongo et al. (cf.[10]), who also proposed a tree variant [9]. These arrays were reinvented by Crochemore et al. [3], who introduced a simple compression scheme not reaching any succinct space guarantees. Up to now, gapped suffix arrays were defined as classical data structures and therefore have large space requirements. In this paper we study different succinct representations of gapped suffix arrays, using state of the art succinct trees, such as wavelet trees [6].

2 Definitions

Throughout this paper log refers to the binary logarithm $\log_2$ and in runtime and space descriptions we will use $\log v$ as a short form of $\lceil \log v \rceil$.

* Supported by FCT through projects TAGS PTDC/EIA-EIA/112283/2009, HELIX PTDC/EEA-ELC/113999/2009 and the PIDDAC Program funds (INESC-ID multiannual funding).

** Corresponding author.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 290–294, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Let $\Sigma = \{0, 1, \ldots, \sigma-1\}$ be an ordered alphabet and $T = t_0 t_1 \ldots t_{n-1}$ a string of length $|T| = n$ over Σ. We write $i \mod n$ for integers i as $i\%n$. We denote the symbol $t_{i\%n}$ by $T[i]$ and write $T[i\,..\,j]$ to represent the factor $T[i]T[i+1]\ldots T[j]$ where $j \geq i$ and $T[i\,..\,j] = \epsilon$ otherwise. We assume that T is terminated by a unique minimal character \$. The string S is a prefix of T if $S = T[0]T[1]\ldots T[m-1]$ for some $m \in \mathbb{N}$, a suffix of T if $S = T[m]T[m+1]\ldots T[n-1]$ for some integer m, a factor of T if $S = T[i\,..\,j]$ for some integers i, j.

The operation $\text{RANK}(S, a, i)$ counts for $S = s_0, s_1, \ldots, s_{m-1}$ how many instances of a appear in $s_0, s_1, \ldots, s_i$ while $\text{SELECT}(S, a, j)$ yields the smallest number i such that $\text{RANK}(S, a, i) = j+1$. For binary sequences both can be performed in constant time using indexes of $o(m)$ bits (cf. [7,8]). Balanced wavelet trees (cf. [6]) can be used to support the operations for sequences over general alphabets $\Gamma = \{0, 1, \ldots, \gamma-1\}$, of size γ, in $O(\log \gamma)$ time and $m \log \gamma + o(m) \log \gamma$ bits. The order 0 entropy $H_0(S)$ of a sequence S of length m over the finite alphabet $\Gamma = \{0, 1, \ldots, \gamma-1\}$ of size γ is defined as $H_0(S) = -\sum_{a \in \Gamma} m_a \log \frac{m_a}{m}$ where $m_a = \text{RANK}(S, a, m-1)$. Using a Huffman shaped wavelet tree (cf. [4]) RANK and SELECT operations on a sequence S of length m can be performed in average time $O(\kappa)$ using index space of $m\kappa + o(m)\kappa$ bits, where κ is the average code length of a Huffman code used for coding the sequence S. In the best case we have $\kappa = H_0(S)$.

A factor S of T is lexicographically smaller than a factor R of T ($S < R$) if $S \neq R$ and either $|S| = 0$ or $|S|, |T| > 0$ and $S[0] < R[0]$ or $|S|, |T| > 0$ and $S[0] = T[0]$ and $S[1\,..\,|S|-1] < R[1\,..\,|R|-1]$ (where we use the bracket operator on S and R in analogy to the use on T). The suffix array SA of T is the permutation of $0, 1, \ldots, n-1$ such that $T[\mathsf{SA}[0]\,..\,n-1] < T[\mathsf{SA}[1]\,..\,n-1] < \ldots < T[\mathsf{SA}[n-1]\,..\,n-1]$. The inverse permutation for SA is called the inverse suffix array of T and denoted by ISA. The Burrows-Wheeler transform (cf. [1]) BWT of T is a string of length n given by $\mathsf{BWT}[r] = T[(\mathsf{SA}[r]-1)\%n]$. This string is useful to compute the LF function, given by $\mathsf{LF}(r) = \mathsf{ISA}[(\mathsf{SA}[r]-1)\%n]$. The longest common prefix of two strings $S = s_1 s_2 \ldots s_s$ and $R = r_1 r_2 \ldots r_r$ written as $\mathsf{lcp}(S, R)$ is the largest ℓ such that $s_1 s_2 \ldots s_\ell$ equals $r_1 r_2 \ldots r_\ell$. The array LCP of the string T is defined by $\mathsf{LCP}[0] = 0$ and $\mathsf{LCP}[r] = \mathsf{lcp}(T[\mathsf{SA}[r-1]\,..\,n-1], T[\mathsf{SA}[r]\,..\,n-1])$ for $r = 1, 2, \ldots, n-1$. Assuming $\sigma \in O(n^c)$ for some constant c the arrays SA and other arrays based on SA can be computed in time $O(n)$ from T (cf. [2]).

We define the cyclic factor of T of length n at position i written as $T[i\,..]$ by $T[i\,..] = T[i\,..\,i+n-1]$. Given two parameters $p, q \in \mathbb{N}$ we say that $T[i\,..]$ is (p,q)-lexicographically smaller than $T[j\,..]$ for $0 \leq i, j < n$ written as $T[i\,..] <_{(p,q)} T[j\,..]$ if $i\%n \neq j\%n$ and either $T[i\,..\,i+p-1] < T[j\,..\,j+p-1]$ or $T[i\,..\,i+p-1] = T[j\,..\,j+p-1]$ and $T[i+p+q\,..\,i+n-1] < T[j+p+q\,..\,j+n-1]$ or $T[i\,..\,i+p-1] = T[j\,..\,j+p-1]$ and $T[i+p+q\,..\,i+n-1] = T[j+p+q\,..\,j+n-1]$ and $T[i+p\,..\,i+p+q-1] < T[j+p\,..\,j+p+q-1]$. Note that for $i\%n \neq j\%n$ we have either $T[i\,..] <_{(p,q)} T[j\,..]$ or $T[j\,..] <_{(p,q)} T[i\,..]$ due to the terminator symbol of T. We define the gapped suffix array $\mathsf{gSA}_{(p,q)}$ of T as the permutation of $0, 1, \ldots, n-1$ satisfying $T[\mathsf{gSA}_{(p,q)}[0]\,..] <_{(p,q)} T[\mathsf{gSA}_{(p,q)}[1]\,..] <_{(p,q)} \ldots <_{(p,q)}$

$T[\mathsf{gSA}_{(p,q)}[n-1]\,..]$. Note that this definition using circular strings slightly differs from the definition given in [3]. The circular definition is better suited for our goal of a succinct representation. In both cases the gapped suffix array is easily computed in linear time from the array SA and LCP of T (see [3]). We denote the inverse permutation of $\mathsf{gSA}_{(p,q)}$ by $\mathsf{IgSA}_{(p,q)}$. The gapped version $\mathsf{gLF}_{(p,q)}$ of the LF function is defined by $\mathsf{gLF}_{(p,q)}(r) = \mathsf{IgSA}_{(p,q)}((\mathsf{gSA}_{(p,q)}(r) - 1)\%n)$.

r	$\mathsf{gSA}[r]$	$\mathsf{gLF}(r)$		$T[\mathsf{gSA}[r]\,..]$	
			L	P G	B
0	15	1	a	$\$a\ ab\ bbbaabaabbba$	1
1	14	9	b	$a\$\ aa\ bbbbaabaabbb$	1
2	6	8	b	$aa\ ba\ abbba\$aabbbb$	0
3	9	10	b	$aa\ bb\ ba\$aabbbbaab$	0
4	0	0	$\$$	$aa\ bb\ bbaabaabbba\$$	0
5	10	3	a	$ab\ bb\ a\$aabbbbaaba$	0
6	1	4	a	$ab\ bb\ baabaabbba\$a$	0
7	7	2	a	$ab\ aa\ bbba\$aabbbba$	0
8	5	15	b	$ba\ ab\ aabbba\$aabbb$	1
9	13	13	b	$ba\ \$a\ abbbbaabaabb$	0
10	8	7	a	$ba\ ab\ bba\$aabbbbaa$	0
11	11	5	a	$bb\ ba\ \$aabbbbaabaa$	0
12	2	6	a	$bb\ bb\ aabaabbba\$aa$	0
13	12	11	b	$bb\ a\$\ aabbbbaabaab$	0
14	3	12	b	$bb\ ba\ abaabbba\$aab$	0
15	4	14	b	$bb\ aa\ baabbba\$aabb$	0
16					1

Fig. 1. Matrix of cyclic factors of length n for string $T = aabbbbaabaabbba\$$ sorted by $<_{(2,2)}$. We denote column $n-1$ by L, column $p-1$ by P and column $p+q-1$ by G.

3 Computing the Gapped LF Function in Succinct Space

In this section we show how to compute the function $\mathsf{gLF}_{(p,q)}$ based on a dictionary of $O(n \log \sigma)$ bits. This is in contrast to using an array for $\mathsf{gSA}_{(p,q)}$ and another for $\mathsf{IgSA}_{(p,q)}$, which requires $O(n \log n)$ bits. Based on $\mathsf{gLF}_{(p,q)}$ we can represent $\mathsf{gSA}_{(p,q)}$ in space $O(n \log \sigma)$ such that each element can be accessed in time $O(\log n)$ by using a sampled $\mathsf{gSA}_{(p,q)}$ array and the $\mathsf{gLF}_{(p,q)}$ function (cf. [4]). Throughout this section we assume fixed parameters p and q and consequently strip them from notations.

Theorem 1. *The function gLF can be computed in time $O(\log \sigma)$ using a data structure of $(3 \log \sigma + 1)(n + o(n))$ bits.*

Proof. The computation of $\mathsf{gLF}(r)$, where $0 \le r < n$, is performed in three steps:

1. Remove the last letter of the prefix, the letter at column $p-1$.
2. Add a letter before the prefix, the letter at column $n-1$.
3. Add a letter at the end of the gap, the letter at column $p+q-1$, .

The first step is in fact the most complicated one and involves two calculations. The first uses a bitmap B, defined as $B[r] = 1$ for $r = n$ or if $\mathsf{LCP}[r] < p-1$ and $B[r] = 0$ otherwise, for $0 \le i < n$, which means it uses $n+1$ bits. The second will be presented along with the third step.

Assume $\delta = \lceil \log \sigma \rceil$ and consider the strings L, G, P of length n defined by $L[r] = T[(\mathsf{gSA}[r]-1)\%n]$, $G[r] = T[(\mathsf{gSA}[r]+p+q-1)\%n]$, $P[r] = T[(\mathsf{gSA}[r]+p-1)\%n]$, for $r = 0, 1, \ldots, n-1$. In fact we represent L and G combined into a string LG were each letter is a pair, with a letter from L and a letter from G, therefore $LG[r] = (L[r], G[r])$. A simple way to encode the pairs is as $L[r]2^{\delta} + G[r]$. We store the strings LG and P using wavelet trees of depth 2δ and δ respectively. The strings L and G are and not stored explicitly, we access them via the wavelet tree that stores LG.

The wavelet trees, including supporting data structures for Rank and Select, that requires $(2\delta+\delta)(n+o(n)) = 3\delta(n+o(n))$ bits. We also store a representation of B, in $n + o(n)$ bits, that supports Rank and Select. The total index size adds up to $(3\delta + 1)(n + o(n))$ bits. The wavelet trees also support *range quantile queries* (RQQ, cf. [5]) and *smaller queries* (smaller, cf. [11]). Given $\ell, r, k \in \mathbb{N}$ where $\ell \leq r, k \in [l, r]$ the query $\mathsf{RQQ}(S, [\ell, r], k)$ returns the k-th element of $S[\ell \mathinner{.\,.} r]$ if it were in sorted order, for a wavelet tree over string S. The query $\mathsf{smaller}(S, [\ell, r], k)$ returns the number of elements smaller than k in $S[\ell \mathinner{.\,.} r]$. Hence we can compute these queries over LG and P in time $O(\delta)$.

We illustrate the procedure with the computation of $\mathsf{gLF}(12)$ on our example string T, see Fig. 1. We compute a sequence of intervals, $I = [\alpha, \beta]$, in the first step, $J = [\iota, \lambda]$, in the second step, a reduction of J to $J' = [\iota', \lambda]$, in the third step. The process terminates by choosing an element of J'.

First set $I = [\text{Select}(B, 1, \text{Rank}(B, 1, r)), \text{Select}(B, 1, \text{Rank}(B, 1, r)+1)-1]$, this interval corresponds to the region that contains the prefix $p-1$ of $\mathsf{gSA}[r]$. Note that implicitly we are discarding the letter in column $p-1$. In the example $I = [8, 15]$, the prefix is b and we are discarding the second b. The second step is a standard LF step. Let $c = L[r]$, in the example $L[12] = a$. Compute $J = [\mathsf{smaller}(L, [0, n-1], c) + \mathsf{rank}(L, c, \alpha - 1), \ldots]$. This LF mapping works because the interval I was chosen irrespectively of the gap, meaning that the cyclic factors of SA in I are the same as those of gSA in I, albeit in a different order. In the example $J = [5, 7]$. The relative order, compared to SA, can be altered by two factors: The first case is when character previously in column $p + g - 1$, $(d = G[r])$ is pushed outside the gap to column $p + g$, which makes it relevant for sorting. The second case is when the letter in column $p-1$ is moved into the gap area, and is thus no longer relevant for sorting.

The first case is solved in the third step. The second case is related to the first step and is determined in the final selection. In the third step we use $\mathsf{smaller}$ and Rank operations in LG and can count the number of characters that are smaller than, or equal to, d and simultaneously have $L = c$. Therefore the equation for J' is $J' = [\iota+\mathsf{smaller}(LG, [\alpha, \beta], (c, d))-\mathsf{smaller}(L, [\alpha, \beta], c)), \ldots]$. In our example we obtain $[6, 7]$. Now to address the second case and finish the first step we determine the relative position of r in J', $\zeta = \text{Rank}(LG, (c, d), r) - \text{Rank}(LG, (c, d), \alpha - 1) - 1$, this determines the relative position of rank r among the rows where L is c and G is d, inside I. In the example $\zeta = 1$. The computation returns

$\mathsf{RQQ}(P, J', \zeta)$ as the final result, in our example 6. The process used a constant number of calls to functions which run in time $O(\delta) = O(\log \sigma)$, so the total running time is $O(\log \sigma)$.

Bringing the balanced wavelet trees to Huffman shape is not straight-forward, as smaller and RQQ require need to encode the symbols in the lexicographic order, which is generally destroyed by Huffman coding. However in most applications the actual alphabet order is unimportant, and this encoding is preferable over a general encoding that uses more space.

Theorem 2. *The function gLF for some alphabet ordering can be computed in average time $O(\kappa)$ using a data structure of $(3\kappa + 1)(n + o(n))$ bits.*

The gLF operation is useful to represent gSA structure, by sampling the values of gSA and IgSA, hence supporting forward search. It can also be extended to support general backward search, although this process is not guaranteed to find all the gap matches of P. We discuss this issue in an extended version.

References

1. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report (1994)
2. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings, 392 pages. Cambridge University Press, Cambridge (2007)
3. Crochemore, M., Tischler, G.: The gapped suffix array: A new index structure for fast approximate matching. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 359–364. Springer, Heidelberg (2010)
4. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 150–160. Springer, Heidelberg (2004)
5. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
6. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850 (2003)
7. Jacobson, G.: Space-efficient Static Trees and Graphs. In: Foundations of Computer Science, pp. 549–554 (1989)
8. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
9. Peterlongo, P., Allali, J., Sagot, M.-F.: Indexing gapped-factors using a tree. Int. J. Found. Comput. Sci. 19(1), 71–87 (2008)
10. Peterlongo, P., Pisanti, N., Boyer, F., Sagot, M.-F.: Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 179–190. Springer, Heidelberg (2005)
11. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 40–50. Springer, Heidelberg (2010)

Finding Frequent Elements in Compressed 2D Arrays and Strings

Travis Gagie[1], Meng He[2], J. Ian Munro[2], and Patrick K. Nicholson[2]

[1] Department of Computer Science and Engineering, Aalto University, Finland
[2] Cheriton School of Computer Science, University of Waterloo, Canada

Abstract. We show how to store a compressed two-dimensional array such that, if we are asked for the elements with high relative frequency in a range, we can quickly return a short list of candidates that includes them. More specifically, given an $m \times n$ array A and a fraction $\alpha > 0$, we can store A in $\mathcal{O}\big(mn(H+1)\log^2(1/\alpha)\big)$ bits, where H is the entropy of the elements' distribution in A, such that later, given a rectangular range in A and a fraction $\beta \geq \alpha$, in $\mathcal{O}(1/\beta)$ time we can return a list of $\mathcal{O}(1/\beta)$ distinct array elements that includes all the elements that have relative frequency at least β in that range. We do not *verify* that the elements in the list have relative frequency at least β, so the list may contain false positives. In the case when $m = 1$, i.e., A is a string, we improve this space bound by a factor of $\log(1/\alpha)$, and explore a space-time trade off for verifying the frequency of the elements in the list. This leads to an $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bit data structure for strings that, in $\mathcal{O}(1/\beta)$ time, can return the $\mathcal{O}(1/\beta)$ elements that have relative frequency at least β in a given range, without false positives, for $\beta \geq \alpha$.

1 Introduction

An α-*majority* in an array is an element that has relative frequency at least α, for $0 < \alpha < 1$. Finding α-majorities in strings is a well-studied problem: Misra and Gries [12], Demaine, López-Ortiz and Munro [4] and Karp, Shenker and Papadimitriou [10] all independently discovered the same algorithm that, given α, makes one pass over the string to build a list of $1/\alpha$ candidate α-majorities, and then makes a second pass to verify them, all using $\mathcal{O}(1/\alpha)$ words of working memory. Recently, this problem has been considered in the approximate setting, using the term *heavy hitters* instead of α-majorities, where false positives and even false negatives are allowed [3,7]. Very recently, Durocher, He, Munro, Nicholson and Skala [5] showed how to store a string in $\mathcal{O}(\log(1/\alpha)\log n)$ bits[1] per character such that, given a range, the list of α-majorities in that range can be returned in $\mathcal{O}(1/\alpha)$ time; an improvement in both time and space over the previous best result [11]. As with Misra and Gries' algorithm, they first build a list of $\mathcal{O}(1/\alpha)$ candidates and then verify them, but this time using a wavelet tree [9] for batched rank queries. In this paper we extend Durocher et al.'s result

[1] We use $\log n$ to denote $\log_2 n$.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 295–300, 2011.
© Springer-Verlag Berlin Heidelberg 2011

in three directions: we consider two-dimensional arrays, we achieve compression, and we show how to answer queries faster when asked only for the β-majorities, for some fraction $\beta > \alpha$.

In Section 2 we describe a data structure for storing an $m \times n$ array A such that later, given a rectangular range in A and a fraction $\beta \geq \alpha$, in $\mathcal{O}(1/\alpha)$ time we can return a list of $\mathcal{O}(1/\beta)$ distinct array elements that *includes* all the elements that have relative frequency at least β in that range. We do not consider how to *verify* which of the $\mathcal{O}(1/\beta)$ candidates are truly β-majorities, so the list may contain false positives. We assume throughout that A's elements are from the alphabet $\{1, \ldots, mn\}$. This data structure occupies $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits, where H is the entropy of the elements' distribution in A. We then show how, by increasing the space by a factor of $\mathcal{O}(\log 1/\alpha)$, we can build the list of candidates in $\mathcal{O}(1/\beta)$ time.

In Section 3 we show that if A is a string, then our space bounds are reduced by a factor of $\log(1/\alpha)$. We also explore a space-time trade off for verifying the candidates in a string. Ultimately, we present a data structure that occupies $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bits, and can return all the $\mathcal{O}(1/\beta)$ verified β-majorities in a range in $\mathcal{O}(1/\beta)$ time. If we reduce the space to $\mathcal{O}(n(H+1))$ bits, we can still return the verified β-majorities in $\mathcal{O}(\log\log n/\alpha)$ time.

2 Two-Dimensional Arrays

For the moment, assume that we are always willing to spend $\mathcal{O}(1/\alpha)$ time to compute the list, regardless of β. We store a Huffman-coded copy of A and a bit-vector that supports select in $\mathcal{O}(1)$ time [2], with a 1 marking the position of the first bit of each codeword. This takes a total of $mn(H + \mathcal{O}(1))$ bits and allows us $\mathcal{O}(1)$-time access to any element in A.

For the sake of simplicity, assume m and n are powers of 2; otherwise, we pad A with null values to make this true. Let I be the set

$$\left\{[i_1..i_2] \subseteq [1..m] : i_2 - i_1 + 1 = 2^k, i_1 = t2^{k-1}, (i_1, i_2, t, k \in \mathbb{Z}^*)\right\} ,$$

and J be the set

$$\left\{[j_1..j_2] \subseteq [1..n] : j_2 - j_1 + 1 = 2^k, j_1 = t2^{k-1}, (j_1, j_2, t, k \in \mathbb{Z}^*)\right\} .$$

For each vertical interval $[i_1..i_2] \in I$ and horizontal interval $[j_1..j_2] \in J$ with $\ell_1\ell_2 > 1/\alpha$, where $\ell_1 = i_2 - i_1 + 1$ is the length of $[i_1..i_2]$ and $\ell_2 = j_2 - j_1 + 1$ is the length of $[j_1..j_2]$, we store a list of the at most $9/\alpha$ distinct array elements that each occurs at least $\alpha\,\ell_1\ell_2$ times in $A[i_1 - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ (we assume these indices are valid; border cases can be handled similarly), in non-increasing order by frequency. Figure 1 shows an example. For any $\beta \geq \alpha$, the first $9/\beta$ elements in this list include all β-majorities for any range that is contained in $A[i_i - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ but contains $A[i_1..i_2, j_1..j_2]$. We call $A[i_1..i_2, j_1..j_2]$ a *block*, and we define the *size* of the block to be $\ell_1 \times \ell_2$.

Finally, we build a Huffman code for the all the blocks' lists and store them encoded, together with another bit-vector that supports select in $\mathcal{O}(1)$ time, with

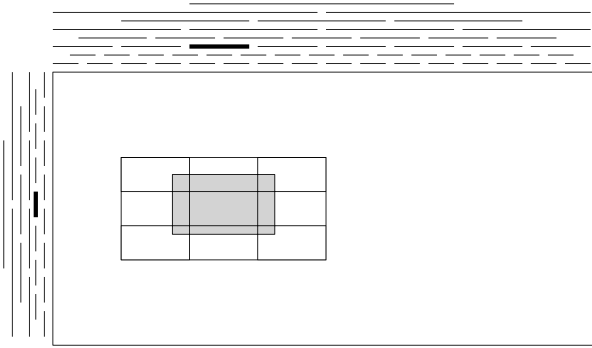


Fig. 1. Consider a vertical interval $[i_1..i_2] \in I$ (shown as thick line at left) of length $\ell_1 = i_2 - i_1 + 1$ and a horizontal interval $[j_1..j_2] \in J$ (shown as thick line at top) of length $\ell_2 = j_2 - j_1 + 1$, with $\ell_1 \ell_2 > 1/\alpha$. The first $9/\beta$ elements in the list we store include all β-majorities for any range that is contained in $A[i_i - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ but contains $A[i_1..i_2, j_1..j_2]$. One such range is shown in grey, with $A[i_1 - \ell_1..i_2 + \ell_1, j_1 - \ell_2..j_2 + \ell_2]$ shown as the 9-part rectangle containing it and $A[i_1..i_2, j_1..j_2]$ as the center part.

a 1 marking the position of the first bit of the first codeword in each list. We now describe how our structure answers queries, and then analyse its space usage.

Answering Queries: Given a rectangular query range in A, we can compute the block size and dimensions of the *query block*, i.e., the largest block contained in the query range, in $\mathcal{O}(1)$ time. Once we have the block size and dimensions, the offset of the query block in the Huffman-coded block lists is easily obtained based on the position of the query rectangle, and we can access the corresponding list in $\mathcal{O}(1)$ time using select. Thus, in $\mathcal{O}(1/\beta)$ time we can retrieve the list, and report the first $\mathcal{O}(1/\beta)$ candidates, if the query block has size greater than $1/\alpha$. If we are asked to find the β-majorities in a smaller range, then we scan the range with Misra and Gries' algorithm [12] in $\mathcal{O}(1/\alpha)$ time.

To reduce the time bound from $\mathcal{O}(1/\alpha)$ to $\mathcal{O}(1/\beta)$ when the query block is smaller than size $1/\alpha$, we store $\log(1/\alpha)$ instances of our data structure. The k-th instance is set up to return candidate $(1/2^k)$-majorities in $\mathcal{O}(2^k)$ time. To find candidate β-majorities, we use the $\lceil \log(1/\beta) \rceil$-th instance[2].

Space Analysis: Since I and J contain $\mathcal{O}(m)$ and $\mathcal{O}(n)$ intervals, respectively, it is not difficult to see that we use a total of $\mathcal{O}((mn/\alpha)\log(mn))$ bits. In fact, we will show that because we do not store lists for ranges with size $1/\alpha$ or less, the $1/\alpha$ factor in the space bound is reduced to $\log(1/\alpha)$. Moreover, encoding the lists using Huffman codes reduces the $\log(mn)$ factor to $H + 1$. The following three lemmas solidify these arguments, bounding the space occupied by our structure.

[2] Some of the block lists in the k-th instance are prefixes of lists in the $(k+1)$-th instance. However, removing this redundancy does not asymptotically reduce space.

Lemma 1. *Any block B of size 2^b overlaps $\mathcal{O}\bigl(\alpha\, 2^b \log(1/\alpha)\bigr)$ blocks of size at most 2^b.*

Proof. The number of such blocks that B overlaps is at most 9 times the number it contains (including itself), so we count the latter. For $k \leq b$, there are $k+1$ possible rectangular shapes with size 2^k and sides whose lengths are powers of 2: i.e., 1×2^k, $2 \times 2^{k-1}$, ..., $2^k \times 1$. For each such shape, B contains at most $4 \cdot 2^b/2^k$ blocks with that shape. Therefore, since $\sum_{k=\log(1/\alpha)}^{b} (k+1)(2^b/2^k) = \mathcal{O}\bigl(\alpha\, 2^b \log(1/\alpha)\bigr)$, B overlaps $\mathcal{O}\bigl(\alpha\, 2^b \log(1/\alpha)\bigr)$ blocks of size at most 2^b. □

Lemma 2. *Any array element c occurs in blocks' lists $\mathcal{O}(\log(1/\alpha))$ times as often as it occurs in A.*

Proof. Let s be the total size of all the blocks B such that c is in B's list but not in the list of any larger block that overlaps B. Some of the choices of B could overlap but c still occurs at least $\alpha\, s/9$ times in A. On the other hand, it follows from Lemma 1 that c occurs in $\mathcal{O}(\alpha\, s \log(1/\alpha))$ blocks' lists. □

Lemma 3. *The Huffman-coded lists take a total of $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits.*

Proof. Suppose we encode the lists with a Huffman code based on A, which cannot give better compression than using one based on the lists themselves. Since encoding A takes $mn(H + \mathcal{O}(1))$ bits, it follows from Lemma 2 that encoding the lists takes $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits. □

Recall that if we are always willing to spend $\mathcal{O}(1/\alpha)$ time to find the candidates, regardless of β, then we store only one instance of our data structure and, by Lemma 3, this occupies $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits. Otherwise, we store a data structure for each of the $\log(1/\alpha)$ fractions, $1/2, 1/4, \ldots, \alpha$, and use a total of $\mathcal{O}\bigl(mn(H+1)\log^2(1/\alpha)\bigr)$ bits. We get the following theorem:

Theorem 1. *Given an $m \times n$ array A and a fraction $\alpha > 0$, A can be stored in $\mathcal{O}\bigl(mn(H+1)\log^2(1/\alpha)\bigr)$ bits, where H is the entropy of the elements' distribution in A, such that later, given a rectangular range in A and a fraction $\beta \geq \alpha$, a list of $\mathcal{O}(1/\beta)$ distinct array elements that includes all the elements that have relative frequency at least β in that range can be returned in $\mathcal{O}(1/\beta)$ time. Alternatively, A can be stored in $\mathcal{O}(mn(H+1)\log(1/\alpha))$ bits, and return the same list in $\mathcal{O}(1/\alpha)$ time, regardless of β.*

3 Improvements for Strings

Space Reduction: In the special case when A is a string — i.e., when $m = 1$ — a range of given size can have only one shape. Using the following lemma instead of Lemma 1, we reduce our space bounds by a $\log(1/\alpha)$ factor.

Lemma 4. *In one dimension, any block B of size 2^b overlaps $\mathcal{O}\bigl(\alpha\, 2^b\bigr)$ blocks of size at most 2^b.*

Proof. The number of such blocks that B overlaps is at most 3 times the number it contains (including itself), which is $\sum_{k=\log(1/\alpha)}^{b} 2^b/2^k = \mathcal{O}(\alpha\, 2^b)$. □

Theorem 2. *Given a string A of length n and a fraction $\alpha > 0$, A can be stored in $\mathcal{O}(n(H+1)\log(1/\alpha))$ bits, where H is the 0th-order empirical entropy of A, such that later, given a range in A and a fraction $\beta \geq \alpha$, a list of $\mathcal{O}(1/\beta)$ distinct characters that includes all those with relative frequency at least β in that range can be returned in $\mathcal{O}(1/\beta)$ time. Alternatively, A can be stored in $\mathcal{O}(n(H+1))$ bits, and return the same list in $\mathcal{O}(1/\alpha)$ time, regardless of β.*

Verifying the Elements: Barbay et al. [1] showed how to store A in $nH+o(n)(H+1)$ bits and answer rank queries on it in $\mathcal{O}(\log\log n)$ time. Combining this with Theorem 2, we can easily verify each candidate c in $\mathcal{O}(\log\log n)$ time using a rank_c query at either end of the range. In the sequel, we show how to verify the $\mathcal{O}(1/\beta)$ candidates returned by the data structure in Theorem 2, in $\mathcal{O}(1)$ time per candidate, using $\mathcal{O}(n\min(\log(1/\alpha), H+1)\log n)$ bits.

For each block $A[j_1..j_2]$ in the string, we store a copy S of the substring $A[j_1-\ell, j_2+\ell]$, where $\ell = j_2 - j_1 + 1$, replacing each character with its rank in the list for $A[j_1..j_2]$; characters not in the list are replaced with 0. Thus, each substring S contains characters drawn from the alphabet $\{0,\ldots,\sigma\}$, where $\sigma \leq \lceil 3/\alpha \rceil$. Furthermore, recall that the list for $A[j_1..j_2]$ is sorted by the characters' frequency in the substring $A[j_1-\ell, j_2+\ell]$ in non-increasing order. This means that, with the exception of 0, the characters' lexicographic order in S is the same as their order by frequency. We call S the *rank sequence* for the block $A[j_1..j_2]$.

Observation 1. *Let S be a sequence of m characters, drawn from the alphabet $\Sigma = \{0,1,...,\sigma\}$, where, ignoring character 0, character i is the i-th most frequent character in S. There exists a skewed wavelet tree T, representing the sequence S using Elias gamma coding [6], that has the following properties:*

1. *The leaf representing i has depth $\mathcal{O}(\log(i+2))$ in T (the root has depth 0).*
2. *For $1 \leq k \leq \sigma$, the leaves representing $1,...,k$ can be traversed in $\mathcal{O}(k)$ time.*
3. *The wavelet tree T occupies $\mathcal{O}(m(H(S)+1))$ bits, where $H(S)$ denotes the 0th-order empirical entropy of the sequence S.*

We store the concatenation of the rank sequences of all blocks of size 2^b, in ascending order of block starting position, as a string Y_b, for $\log(1/\alpha) < b \leq \log n$, noting that $|Y_b| = \mathcal{O}(n)$. Using the wavelet tree from Observation 1 to represent these strings will allow us to count the frequency of candidate elements efficiently. However, at this point we have not built concatenated strings for blocks of size smaller than $1/\alpha$. We now explain how to construct $Y_{b'}$, for block sizes $2^{b'}$, where $1 \leq b' \leq \log(1/\alpha)$, in order to verify the frequency of candidates in these smaller blocks in $\mathcal{O}(1/\beta)$ time.

Recall that the first result in Theorem 2 keeps $\log(1/\alpha)$ copies of the data structure, constructed for values in the set $F = \{1/2, 1/4, ..., \alpha\}$. Let $D_{\alpha'}$ denote the data structure constructed for $\alpha' \in F$. The string $Y_{b'}$ is constructed using the block lists from $D_{\alpha'}$, where α' is the smallest value in F such that $D_{\alpha'}$ stores candidate lists for blocks of size $2^{b'}$. We now argue that representing all of the $\log n$ strings using Observation 1 requires $\mathcal{O}(n\min(\log(1/\alpha), H+1)\log n)$ bits.

Consider the rank sequence S of a block B of size 2^b, and the original substring S' of A, to which S corresponds. Applying a many-to-one mapping to a string's alphabet — e.g., from characters to their ranks or 0 — cannot increase its entropy, so $H(S) \leq H(S')$. Moreover, if we partition a string, then the average of the substrings' entropies, weighted by length, is at most the string's entropy [8], so $H(Y_b) = \mathcal{O}(H)$ for $1 \leq b \leq \log n$. Furthermore, it is clear that $H(Y_b) = \mathcal{O}(\log(1/\alpha))$, since the alphabet size is at most $\lceil 3/\alpha \rceil$. Since there are $\log n$ strings, the space is $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bits overall.

Verifying the candidates for a block of size 2^b involves traversing the first $\mathcal{O}(1/\beta)$ leaves of the skewed wavelet tree representing Y_b, which takes $\mathcal{O}(1/\beta)$ time by Observation 1. At each leaf, which represents a candidate, at most two $\mathcal{O}(1)$ time rank queries on a bit vector can be used to determine the frequency of the candidate in a range, in $\mathcal{O}(1/\beta)$ time overall.

Theorem 3. *Given a string A of length n and a fraction $\alpha > 0$, A can be stored in $\mathcal{O}(n \min(\log(1/\alpha), H+1) \log n)$ bits, where H is the 0th-order empirical entropy of A, such that later, given a range in A and a fraction $\beta \geq \alpha$, the list of $\mathcal{O}(1/\beta)$ distinct characters with relative frequency at least β in that range can be returned in $\mathcal{O}(1/\beta)$ time. Alternatively, A can be stored in $\mathcal{O}(n(H+1))$ bits, and return the same list in $\mathcal{O}(\log\log n/\alpha)$ time.*

References

1. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/Select and applications. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 315–326. Springer, Heidelberg (2010)
2. Clark, D., Munro, J.I.: Efficient Suffix Trees on Secondary Storage (extended abstract). In: Proc. SODA, p. 383 (1996)
3. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. of Alg. 55(1), 58–75 (2005)
4. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 348–360. Springer, Heidelberg (2002)
5. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6755, pp. 244–255. Springer, Heidelberg (2011)
6. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. on Inf. Theory 21(2), 194–203 (1975)
7. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.: Computing iceberg queries efficiently. In: Proc. VLDB, pp. 299–310 (1998)
8. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. on Alg. 3(2) (2007)
9. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850 (2003)
10. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Data. Sys. 28(1), 51–55 (2003)
11. Karpinski, M., Nekrich, Y.: Searching for frequent colors in rectangles. In: Proc. CCCG, pp. 11–14 (2008)
12. Misra, J., Gries, D.: Finding repeated elements. Sci. Comp. Prog. 2, 143–152 (1982)

On Suffix Extensions in Suffix Trees*

Dany Breslauer[1] and Giuseppe F. Italiano[2]

[1] Caesarea Rothchild Institute, University of Haifa, Haifa, Israel
[2] Università di Roma "Tor Vergata", Rome, Italy

Abstract. Suffix trees are inherently asymmetric: prefix extensions only cause a few updates, while suffix extensions affect all suffixes causing a wave of updates. In his elegant linear-time on-line suffix tree algorithm Ukkonen relaxed the prevailing suffix tree representation and introduced two changes to avoid repeated structural updates and circumvent the inherent complexity of suffix extensions: (1) open ended edges that enjoy gratuitous leaf updates, and (2) the omission of implicit nodes. In this paper we study the implicit nodes as the suffix tree evolves. We partition the suffix tree's edges into collections of similar edges called *bands*, where implicit nodes exhibit identical behavior, and generalize the notion of open ended edges to allow implicit nodes to "float" within bands, only requiring updates when moving from one band to the next, adding up to only $O(n)$ updates. We also show that internal implicit nodes are separated from each other by explicit suffix tree nodes and that all external implicit nodes are related to the same periodicity. These new properties may be used to keep track of the waves of implicit node updates and to build the suffix tree on-line in amortized linear time, providing access to all the implicit nodes in worst-case constant time.

1 Introduction

Suffix trees and *suffix arrays* are perhaps the most prevalent data structures in the study of string algorithms, with numerous applications [2,9,16]. Weiner [30] introduced suffix trees and gave a reverse right-to-left on-line algorithm for their construction in linear-time. McCreight [24] gave a linear-time left-to-right algorithm that is not on-line since it must process sufficient "lookahead" of text symbols to find the correct insertion points in the suffix tree. Suffix arrays, that were introduced by Manber and Myers [23], provide similar theoretical benefits to suffix trees, but are much more efficient in practice thanks to their use of efficient array representation, avoiding the extra space and irregular memory access patterns inherent in pointer-based data structures. The compact and efficient suffix array representation in a contiguous memory block, however, is less

* Work partially supported by the European Research Council (ERC) Project SFEROT, by the Israeli Science Foundation Grant 347/09, by the 7th Framework Programme of the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future - From Theory to Design") and by MIUR, the Italian Ministry of Education, University and Research, under Project AlgoDEEP.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 301–312, 2011.
© Springer-Verlag Berlin Heidelberg 2011

amenable to *on-line* construction, because the eventual insertions in the middle of an array would be costly. In fact, the fastest existing linear-time suffix array construction algorithms over *large integer alphabets* [18,19,20] and their earlier linear-time suffix tree counterparts [12,13] usually use bucket sort and other techniques that are unfortunately *off-line*.

Ukkonen's [29] on-line algorithm is often regarded as the simplest and the most intuitive among the suffix tree construction algorithms. To develop an efficient linear-time "on-line" left-to-right suffix tree algorithm, Ukkonen had to relax the prevailing representation of suffix trees in order to avoid repeated structural updates to the suffix tree. In particular, Ukkonen introduced two changes: (1) *open ended edges* leading to suffix tree leaves that represent ever growing suffixes of the text, introducing gratuitous leaf updates; and (2) the omission of *implicit non-branching suffix nodes* until such nodes can be made explicit by inserting branching nodes and leaves. Analogous issues arise in an incrementally maintained suffix array or the closely related Burrows-Wheeler Transform compressed text [25,26].

While open ended edges are an elegant concept that was celebrated by Geigerich and Kurtz [15] who speculate that "if Weiner had seen this idea in 1973, he would have designed Ukkonen's algorithm then", the omission of "implicit" internal suffix nodes is necessitated by their frequent updates. Gusfield [16] calls Ukkonen's intermediate suffix trees "implicit suffix trees" and Ukkonen [29] writes that "when explicit final states are needed in some application, they are obtained gratuitously by adding to $\mathcal{T}$ an end marking symbol that does not occur elsewhere in $\mathcal{T}$." However, it might be costly to follow this suggestion repeatedly. Indeed, always updating explicitly all the implicit nodes in a text of length n takes at least $\Omega(n \log_{|\Sigma|} n)$ updates and up to $O(n^2)$ updates, depending on the structure of the text (e.g., $a^n b a^n c$). The need for an end marking symbol to complete the construction is related to subsequential transducers [5,8] where the computation is completed only upon a terminating input symbol.

Throughout the paper, we assume that the input alphabet Σ has constant size and refer to edges leading to suffix tree leaves either as open ended edges or as external leaf edges and to the other edges as internal edges. Our contributions in this paper are two-fold. First, we use properties of periodicities in the text to draw conclusions about the locations of the implicit nodes as the suffix tree evolves. Specifically, we prove that (1) there can be only one implicit node floating within each internal suffix tree edge, and that (2) external open ended suffix tree edges may contain several implicit floating nodes, but all such implicit nodes are related to the same periodicity; these external implicit nodes do not require updates until they branch out since they never reach the end of an open ended edge. These two properties stem from the fact that longer irregular periods which are not multiples of a string's shortest period correspond to terminated prefix periods that introduce branching nodes in the suffix tree, separating the implicit nodes. While periodicity properties are often used in comparison based string matching algorithms, suffix trees capture all the internal structure of the text and algorithms typically do not need periodicity properties in their construction.

Periodicity properties were occasionally used either implicitly or explicitly, however, in several suffix tree based algorithms [3,7,17,22].

We then push Ukkonen's approach one step further and generalize the notion of open ended edges to allow implicit nodes to "float" within suffix tree edges, requiring updates only when an implicit node moves from one edge to the next. We group the suffix tree edges containing implicit floating nodes into *bands*, which are collections of similar edges whose *both* endpoints are connected by suffix links, where implicit floating nodes exhibit identical behavior. We prove that if we maintain only one *implicit node representative* in each band, then the number of representative updates throughout the suffix tree construction drops to $O(n)$, significantly fewer than the $O(n^2)$ total explicit updates of all the implicit nodes.

Based on these properties, we present a linear-time on-line algorithm that maintains the representatives of the implicit nodes in each band via an auxiliary data structure, providing access to all the implicit nodes in worst-case constant time via queries that produce the implicit nodes on any given suffix tree edge, enabling algorithms that might require these nodes to use the intermediate implicit suffix trees without continuously adding the end marking symbol.

2 Suffix Trees and Ukkonen's Algorithm

Given a text w, the *suffix tree* of w is a rooted tree with edges and nodes that are labeled with substrings of w. The suffix tree satisfies the following properties: (1) edges leaving any given node are labeled with non-empty strings v that start with different alphabet symbols (v is a substring of w); (2) each node is labeled with a string v formed by the concatenation of the edge labels on the path from the root to that node (v is a substring of w); (3) each branching internal (non-leaf) node has at least two descendants (the root may be an exception in the degenerate case when a string is empty or it is formed by repetitions of a single alphabet symbol); (4) for each substring v of w, there exists a vertex labeled u, such that v is a prefix of u.

It is a common practice to append at the end of the text w a special unique alphabet symbol \$, which does not appear anywhere within w. This guarantees that the suffix tree has exactly $|w|+1$ leaves that are labeled with all the distinct non-empty suffixes of $w\$$. The number of branching internal nodes is no larger than $|w|$. However, in on-line algorithms that construct the suffix trees for a left-to-right streaming text, it is not possible to append the special alphabet symbol \$ at each step. Therefore, an on-line algorithm must deal also with suffix tree nodes representing text suffixes which may not be branching out of the tree. Such text suffixes might end at internal branching suffix tree nodes, but also in the middle of suffix tree edges. We use the convenient notation of associating an edge between parent u to child v with the child suffix tree node v. One can realize such suffix trees by taking the suffix trees with the special terminating symbol \$, and removing all edges that are labeled only with the symbol \$ and the leaves connected to these edges. This process may introduce internal non-branching

nodes into the suffix tree, which are sometimes called *implicit nodes*, that may or may not be represented explicitly in the suffix tree. Such suffix trees were called *extended suffix trees* by Breslauer and Hariharan [6] and *implicit suffix trees* by Gusfield [16].

Weiner [30], McCreight [24] and Ukkonen [29] all augment the suffix trees with shortcuts called *suffix links*, that are used to efficiently traverse the suffix tree. We define the *suffix link* for a suffix tree node labeled $v = au$, $a \in \Sigma$, to be a pointer to the suffix tree node labeled with its suffix u, obtained by chopping off v's first symbol a. Suffix links are always defined for each internal branching node. If the node $v = au$ branches with edges that begin with alphabet symbols b and c, $b \neq c$, then the suffix tree also contains the substrings ub and uc and there must be a node labeled u, branching on the symbols b and c. The situation with suffix tree leaves is a little more complicated. Leaves clearly represent text suffixes, since only suffixes of the text end abruptly on their right side. If the text is terminated with a unique symbol, then all suffixes of the text are leaves and, therefore, if $v = au$ is a leaf then its suffix u must also be a leaf. However, if the text is not terminated, then the suffix u might be an implicit non-branching node in the middle of an edge or coincide with an existing branching node.

Since each non-root node has one suffix link and suffix links cannot introduce cycles, suffix links define a tree rooted at the suffix tree root. In fact, if each tree edge between nodes $v = au$ and u is labeled with the alphabet symbol a, this tree is actually an un-compacted trie, which is a subtree of the suffix trie for the reversed text. In this paper we maintain the edges of this trie in both directions (McCreight and Ukkonen only need edges pointing towards the root), and call this tree the *suffix link trie*. The path in the suffix link trie from the longest leaf representing the full text to the root goes through all the suffixes of the text, which are the only substrings that get extended while the input text is processed from left to right. We call this path the *suffix chain* (see Figure 1).

Lemma 1. *The suffix chain can always be partitioned into the following consecutive segments: (1) leaves; (2) external implicit nodes within leaf edges; (3) internal implicit nodes within internal edges; and (4) implicit nodes that coincide with explicit nodes.*

Proof. Let v be a suffix of the text. If v is not a leaf in the suffix tree, then it can be extended and so can all its suffixes, making all suffixes of v implicit nodes. If v is not an external implicit node, then it can be extended to some branching node and so can all its suffixes, making all suffixes of v internal implicit nodes. Similarly, if v coincides with some branching node, then all its suffixes must also coincide with branching nodes. □

Ukkonen [29] noted the great resemblance between his and McCreight's algorithms, by observing that "in its final form our algorithm is a rather close relative of McCreight's method." The parallels between McCreight's, Ukkonen's and Weiner's algorithms have been studied by Geigerich and Kurtz [15], who showed that the difference between Ukkonen's and McCreight's algorithms boils down to their "control structure" and that, essentially, they update the suffix tree in

Fig. 1. (a) The suffix tree for the text "*abaababaababaabab*" with internal node suffix links. (b) The suffix tree with implicit nodes and the suffix chain. The suffix link trie consists of the suffix chain in figure (b) plus the suffix links shown in (a).

exactly the same insertion batches. McCreight's algorithm reads ahead sufficient "lookahead" symbols to find the final insertion points of new suffix leaves, while Ukkonen's algorithm holds off inserting new suffixes until sufficient text symbols have been scanned to insert new branching suffix leaves, omitting the implicit nodes. While the input text is streamed on-line from left to right, all the suffixes of the text are extended with each right text extension. Ukkonen observed that once some text suffix is a leaf, it will remain forever a leaf after all future extensions. By labeling the external suffix tree edges leading to leaves "open ended", reaching to the current *growing* end of the text, Ukkonen invented an automatic gratuitous extension mechanism for these edge labels. Unfortunately, the remaining text suffixes that are not represented by leaves still move around the suffix tree, and it would be too costly to update all their locations. Hence, Ukkonen chose not to update these "implicit" nodes explicitly.

Ukkonen's on-line algorithm maintains the *active suffix*, the longest suffix of the text that has not branched out to become a leaf, which is the *longest repeated suffix* of the text that appeared previously in the text. By Lemma 1, any suffix of the active suffix must have also appeared previously and has not branched out yet. If upon the next symbol the active suffix cannot be extended within the suffix tree, an *insertion batch* creates leaves for all suffixes between the old active suffix and up to the new active suffix. In particular: those implicit nodes on the suffix chain that were in the middle of an edge must branch out by creating an internal branching node splitting the edge and inserting a leaf; and those implicit nodes that coincided with an explicit node branch out by creating a leaf hanging off the existing explicit node. Ukkonen's algorithm maintains the current active suffix by a pointer to a suffix tree node and an offset within a suffix tree edge that are updated while tracing the suffix tree, selecting the appropriate branch at each internal node according to the first symbol on the branching edges. During an insertion batch, the algorithm follows the suffix links to find the next active suffix. While following suffix links, more suffix tree nodes may appear on the path between the suffix tree node and the offset representing the implicit node

and the representation must be updated to the *canonical* representation specifying implicit nodes by their offset relative to the beginning of the edge where they are located. Ukkonen's algorithm, like McCreight's algorithm, only has to navigate the suffix tree by selecting edges at each branching node according to their first branching symbol, quickly moving down the suffix tree path towards the implicit node. The total amount of work is amortized to linear time.

3 The Locations of Implicit Nodes

Periodicity is often used in efficient string matching algorithms. However, suffix trees and related index data structures that express all internal repetition structure of a string typically rely instead on the mechanics of maintaining various graph pointers and on identifying alphabet symbols via direct array access. In this section we use simple periodicity properties to sort through the locations of implicit nodes. Before doing that, we need to review some basic terminology.

A string u is *a period* of a string w if w is a prefix of u^k for some integer k, or equivalently if w is a prefix of uw. The shortest period of w is called *the period* of w and w is called *periodic* if it is at least twice as long as its period. If v is a prefix of w, then the period of v is said to *continue* in w if v and w have the same period and otherwise the period of v *terminates* in w. A string v is called a *border* of w, if v is both a prefix and a suffix of w. By these definitions v is a border of $w = uv = vu'$ if and only if u is a period of w, and therefore, u is the shortest period of w if and only if v is the longest proper border. The following *Periodicity Theorem* is due to Fine and Wilf [14].

Theorem 1. *If a string u has periods of length p and q, and its length $|u| \geq p + q - \gcd(p, q)$, then u also has a period of length $\gcd(p, q)$.*

The following simple observation connects periods and borders of strings to their suffix trees and suffix link tries.

Lemma 2. *A string v is an ancestor of w both in the suffix tree and in the suffix link trie if and only if v is a border of w.*

Proof. Recall that v is an ancestor of w in the suffix tree if and only if v is a prefix of w, i.e., $w = vu'$. Similarly, v is an ancestor of w in the suffix link trie if and only if v is a suffix of w, i.e., $w = uv$. Therefore, v is an ancestor of w both in the suffix tree and in the suffix link trie if and only if v is a border of w. □

We will next exploit the connection given in Lemma 2 to prove the following two properties on suffix trees: (1) there might be only one implicit node within each internal edge, and (2) although external leaf edges may contain many implicit nodes, those implicit nodes enjoy some nice structural properties.

Theorem 2. *An internal suffix tree edge may contain at most one implicit node. This node may coincide with the branching node at the end of the edge.*

Proof. Let w be an internal branching suffix tree node. Assume by contradiction that there exist two different implicit nodes u and v on the edge leading to w, such that $|u| < |v| \leq |w|$. Node u is clearly an ancestor of v in the suffix tree. Since implicit nodes always represent suffixes of the text, it is not difficult to see that u must be an ancestor of v also in the suffix link trie. Then, by Lemma 2, u is a border of v and $v = xu$ has a period x. Let vy be the longest prefix of w which continues that period x of v. If vy is a proper prefix of w, then let vya be the prefix of w where that period x terminated and let $vyb = xuyb$, which is not in the suffix tree, be a string that continues the periodicity, such that uyb is a prefix of $vy = xuy$. Otherwise, if $vy = w$, then w is an internal branching node and it must have at least two outgoing edges, at least one of which contains $wa = vya$ that does not continue that period x, while $wb = vyb = xuyb$, $a \neq b$, may continue the periodicity x. In either case, the suffix tree contains both uya and uyb, $a \neq b$, and uy must be a branching suffix tree node, either between u and v if $|u| \leq |uy| < |v|$ or between v and w if $|v| \leq |uy| < |w|$ contradicting the assumption that u, v and w are on the same edge. □

Leaf edges may each contain several external implicit nodes. The following theorem characterizes all the external implicit nodes and shows that these nodes must obey a simple arithmetic progression formula derived from the total number of external implicit nodes, the longest external implicit node and the period length of its associated leaf.

Theorem 3. *Let $v_0, \ldots, v_{k-1}$ be all the external implicit nodes ordered in decreasing length, and let $w_0, \ldots, w_{k-1}$ be the leaves at the end of their respective edges. Let $p = |w_0| - |v_0|$, $m = \lfloor k/p \rfloor$ and $r = k - pm$. Then:*

(1) The implicit node v_i has length $|v_i| = |v_0| - i$, $i = 0, \ldots, k-1$.

(2) There are at most p distinct leaves with lengths $|w_i| = |w_0| - i$, $i = 0, \ldots, \min\{p, k\} - 1$. For $i = p, \ldots, k-1$, we have $w_i = w_{i-p}$.

(3) The implicit nodes within the leaf edges to w_i have lengths

$$|w_i| - \ell p \quad \begin{cases} \text{for } \ell = 1, \ldots, m+1, & \text{if } 0 \leq i < r \\ \text{for } \ell = 1, \ldots, m, & \text{if } r \leq i < p \end{cases}$$

(4) All leaves w_i, $i = 0, \ldots, \min\{p, k\} - 1$, have a period of length p.

(5) If the text is periodic, then $w_0 = w$ is the whole text, $p = |w_0| - |v_0|$ is the period length, and all implicit nodes whose length is at least p are external. Specifically, $k > |w| - 2p$ and $|v_i| < 2p$, $i = \max\{0, k-p\}, \ldots, k-1$.

Proof. Let $w'_0 = w_0$ and let w'_i be the suffix of w'_{i-1} obtained after chopping off the first symbol of w'_{i-1}, i.e., following w'_{i-1}'s suffix link. Let $\kappa = \min\{p, k\}$. By Lemma 1, since $p = |w_0| - |v_0|$, the external implicit nodes are $v_i = w'_{i+p}$, $0 \leq i \leq k-1$, with $|v_i| = |v_0| - i$. Since v_i is an ancestor of w'_i, the longer $w_i = w'_i$, $0 \leq i \leq \kappa - 1$, are leaves and $|w_i| = |w_0| - i$. The leaf w_i at the edge containing v_i, $p \leq i \leq k-1$, must be the same as that at the edge containing $v_{i-p} = w'_i$. Thus $w_i = w_{i-p}$, $p \leq i \leq k-1$, establishing (1) and (2). Observe that the implicit nodes on the leaf edge to w_i are v_{i+jp}, for $0 \leq i \leq p-1$

and $0 \le i + jp \le k - 1$. Recalling that $k = r + pm$, we get $0 \le j \le m$ when $0 \le i < r$ and $0 \le j < m$ when $r \le i < p$, proving (3) since v_{i+jp} has length $|v_0| - i - jp = |w_0| - p - i - jp = |w_i| - (j+1)p$. We next turn to (4). Since v_0 is an ancestor of w_0 both in the suffix tree and in the suffix link trie, by Lemma 2, $p = |w_0| - |v_0|$ is a period length of w_0, and by the maximal length of v_0, p is the smallest such period length. The same holds for any other leaf w_i and its longest border v_i, $i = 1, \ldots, \kappa - 1$.

We finally turn to (5). Let u be an implicit node and let z_0 be the longest leaf descendant of u. We first prove that if $z_0 = u_0 u$ has period u_0, $|u_0| \le |u|$, then u must be external. Assume it is not: then u is also ancestor of another shorter leaf $z_1 = u_1 u$ with period u_1, such that $|u_1| < |u_0|$. But z_1 is a suffix of z_0 with periods of length $|u_0|$ and $|u_1|$, such that $|u_0| + |u_1| \le |u| + |u_1| \le |z_1|$, and therefore by Theorem 1, z_1 must have a period of length $\gcd(|u_0|, |u_1|)$, and z_0 also must have period length $\gcd(|u_0|, |u_1|) < |u_0|$, clearly a contradiction. Let x be the period of the text, $w = xv$. Then by Lemma 2, v is an implicit node that is ancestor of w and since the text is periodic, $|x| \le |v|$. Therefore, $v_0 = v$ must be an external implicit node, $w_0 = w$ its leaf, and $p = |w_0| - |v_0|$ the text period length. Let v_i be the suffix of v_0 of length $|v_i| = |v_0| - i$, and let w'_i be the longest leaf descendant of v_i. Since the period length of w'_i is at most p, if $|v_i| \ge p$ then v_i must be an external implicit node and $w_i = w'_i$ its leaf. This holds for all v_i such that $|v_i| = |w| - p - i \ge p$, i.e., for $i = 0, \ldots, |w| - 2p$. Thus, it must be that $k > |w| - 2p$, and the last p implicit nodes in the sequence are such that $|v_i| < 2p$, $i = \max\{0, k - p\}, \ldots, k - 1$. □

We remark that irregular long periods (small overlap borders) are permitted by Theorem 1. However, by Theorems 2 and 3, these irregular periods must be separated from each other and from the arithmetic progression of large overlap periods by branching suffix tree nodes. Multiple implicit floating nodes within an external edge also indicate the eventual formation of new maximal repetitions in the sense of [17,21]. Theorem 3 also shows that an implicit suffix tree that is enhanced by the external implicit nodes, which are easy to represent, provides all suffix nodes except possibly for those in the last period of the text, missing fewer suffixes than half of the text's length. An algorithm may choose to represent the external implicit text suffixes by an arithmetic progression, or to start pro-actively inserting such implicit nodes into the tree anticipating their suffix tree locations when they eventually branch out.

4 Maintaining Implicit Nodes Efficiently

We push Ukkonen's approach one step further and allow implicit nodes to "float" within suffix tree edges. Since implicit nodes always represent suffixes of the text, implicit nodes that do not branch out of the suffix tree as the text grows can be envisioned to be floating along the tree edges, and need to be updated only when they move from one edge to the next. This is analogous to having "bounded ended edges" leading to branching nodes as opposed to "open ended" edges leading to leaves, and updates are required to skip to the next edge only when

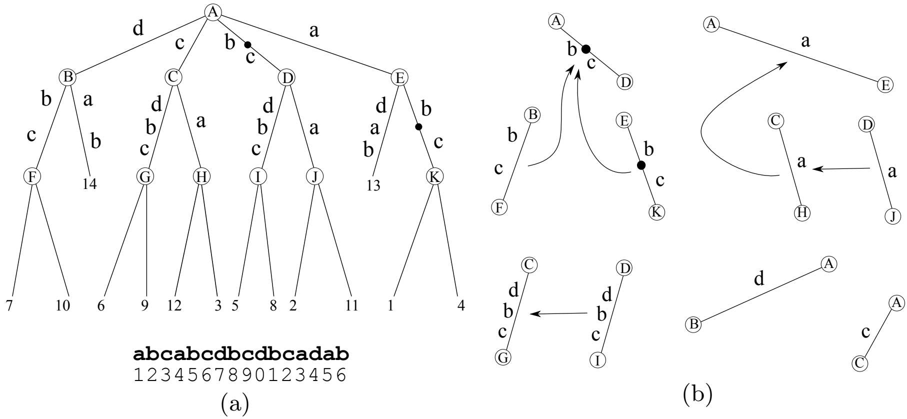


Fig. 2. (a) The suffix tree for "*abcabcdbcdbcadab*". For brevity, the last characters of suffixes are omitted and the leaves are identified by numbers giving the start position of the corresponding suffix. (b) Band trees of the internal edges in (a).

the edge boundary is reached at branching suffix tree nodes. Unfortunately, even with this proposed modification that eliminates the need to maintain implicit nodes inside a suffix tree edge, the number of updates might still be too large, as shown by the example $a^n ba^n c$, which requires $\Theta(n^2)$ updates. Note that an implicit floating node within an open ended edge that is leading to a leaf will never reach the end of the edge since both the implicit node and the leaf at the end of the edge keep getting extended to the current end of the text. Such external implicit nodes will therefore remain implicit nodes until they branch out to become suffix tree leaves.

To reduce the number of updates, we partition the suffix tree edges into *bands* and maintain only one representative for the implicit nodes within each band. Considering a suffix tree edge $e = (au, auv)$ from the node au to the node auv, the suffix link to u must be an ancestor of the suffix link to uv. If $e' = (u, uv)$ is an edge in the suffix tree, we say that e is in the same *band* as e' and connect the suffix tree edge e to e' by an edge in the *band forest*, partitioning all edges of the suffix tree into a collection of band trees (see Figure 2). For any suffix tree edge e the root in e's band tree is the *band representative* of e. We only maintain representative implicit nodes within these band representatives. Since all the beginning nodes and end nodes of all edges in a band are connected by a suffix link path, an implicit node enters and exits the band together with the band representative, allowing us to update only band representatives. This definition leads to reduce the number of implicit node representative updates to $O(n)$.

Theorem 4. *An on-line algorithm has to update band representatives at most* $2n$ *times.*

Proof. We give an implementation of Ukkonen's algorithm that maintains implicit nodes representatives in each band. We maintain a stack of the active

bands, observing that bands on the stack are nested, because their endpoints are suffix tree nodes that have suffix link paths all the way up to the suffix link trie root. We first pop from the stack all the bands whose end is reached, keeping the active node that is the end of the last popped edge, or the suffix tree root if no bands were popped from the stack (this active node is the first explicit node on the suffix chain after the active point in Ukkonen's algorithm, whereas all the longer implicit nodes are in the middle edges). We then check whether the current input symbol does not branch out of the suffix tree at the deepest band (the active point in Ukkonen's algorithm). If it does, then the stack is reset, new suffix tree nodes are inserted and the band tree structure is updated accordingly. In either eventuality, we repeatedly take the suffix tree edge that starts at the active node and the current input symbol and find the end of its band, the edge at the root of its band tree. That band is pushed onto the stack and the active node is set to the next band, the suffix link of the node at the beginning of the band tree root, repeatedly, stopping only after the suffix tree root. Because the bands on the stack are all nested, each band popped from the stack must have started or must end at a different text position. This implies that at most $2n$ bands can be popped from the stack while processing a string of length n. □

Theorem 4 can be used in an on-line implementation of Ukkonen's algorithm to maintain the band representatives and to produce implicit nodes upon queries.

Corollary 1. *An on-line algorithm can maintain band representatives using auxiliary data structures in $O(n)$ amortized time. Queries returning the implicit nodes within a specified suffix tree edge take worst-case $O(1)$ time.*

Proof. The algorithm maintains two dynamic auxiliary data structures. The first data structure maintains the various band trees under operations that insert band tree leaves or delete band tree roots (splitting up the tree) in amortized $O(n)$ time, supporting queries that return the band tree's root, which is the top suffix tree edge in the band tree, in worst-case $O(1)$ time. This can be achieved with the help of data structures for dynamic nearest marked ancestors in trees [1,31]. The second data structure maintains the active part between the first and last active band edges in a band tree, supporting queries that check if a specific suffix tree edge that is represented by a band tree node is currently active or not, with tree updates and queries taking worst-case $O(1)$ time. This can be done by testing whether the query edge is on the band tree path between the first band edge and the last band edge, i.e., the band tree root, and can be implemented efficiently by checking for ancestor relationship [4,10,28].

The stack in Theorem 4 is then implemented using these data structures, where each band that is pushed on the stack has its band representative, the band tree root, updated and the active band part set, leading to an $O(n)$ on-line implementation of Ukkonen's suffix tree algorithm. Insertions in each batch are effectuated from shallow to deep, from the shortest inserted suffix towards the longest (reverse of Ukkonen's algorithm using reverse suffix links), so that only band tree roots are deleted at each step and the split edge's parts are inserted as leaves into their appropriate band trees. Queries about some specified suffix tree

edge first locate the root of its band tree, and then check if the edge is in the active part of the band tree. If the edge is part of an active band, the band tree root gives the single implicit node in the band representative for internal suffix tree bands (Theorem 2) or the arithmetic progression representing the implicit nodes in external suffix tree bands (Theorem 3). Bands of external leaf edges are optional due to the global properties of Theorem 3. □

5 Conclusions

We proved some new combinatorial properties about suffix trees that appear to be mostly of theoretical interest; we are curious whether they can be used in new applications. For example, the intermediate suffix trees could be used as an on-line index to report all occurrences of a pattern in the text in time proportional to the pattern length and the number of reported occurrences. However, one could use instead the suffix tree of the reverse text [30] that undergoes only $O(n)$ structural changes while being updated on line.

There exist off-line linear-time algorithms for suffix tree and suffix array construction over larger integer alphabets [12,13,18,19,20,27] (DAWG and Aho-Corasick Automata by reductions [6,11]). We are curious whether linear-time on-line suffix tree algorithms exist over large integer alphabets.

Acknowledgments. We thank Maxime Crochemore, Roberto Grossi, Gadi Landau and Filippo Mignosi for several discussions about this work and the anonymous reviewers for their useful comments.

References

1. Amir, A., Farach, M., Idury, R., Poutré, J.L., Schäffer, A.: Improved Dynamic Dictionary-Matching. Information and Computation 119, 258–282 (1995)
2. Apostolico, A.: The Myriad Virtues of Subword Trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO ASI Series F, vol. 12, pp. 85–96. Springer, Berlin (1985)
3. Apostolico, A., Preparata, F.: Optimal Off-line Detection of Repetitions in a String. Theoret. Comput. Sci. 22, 297–315 (1983)
4. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two Simplified Algorithms for Maintaining Order in a List. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002)
5. Berstel, J.: Transductions and Context-Free Languages. Teubner-Verlag (1979) Revised version is available electronically as, http://www-igm.univ-mlv.fr∼berstel/LivreTransductions/LivreTransductions14dec2009.pdf
6. Breslauer, D., Hariharan, R.: Optimal Parallel Construction of Minimal Suffix and Factor Automata. Parallel Processing Letters 6(1), 35–44 (1996)
7. Cohen, H., Porat, E.: Range Non-overlapping Indexing. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1044–1053. Springer, Heidelberg (2009)
8. Crochemore, M.: Transducers and Repetitions. Theoret. Comput. Sci. 12, 63–86 (1986)

9. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, Oxford (1994)
10. Dietz, P.F., Sleator, D.D.: Two Algorithms for Maintaining Order in a List. In: STOC, pp. 365–372. ACM, New York (1987)
11. Dori, S., Landau, G.M.: Construction of Aho Corasick Automaton in Linear Time for Integer Alphabets. Inf. Process. Lett. 98(2), 66–72 (2006)
12. Farach, M.: Optimal Suffix Tree Construction with Large Alphabets. In: FOCS, pp. 137–143 (1997)
13. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the Sorting-Complexity of Suffix Cree construction. J. ACM 47(6), 987–1011 (2000)
14. Fine, N., Wilf, H.: Uniqueness Theorems for Periodic Functions. Proc. Amer. Math. Soc. 16, 109–114 (1965)
15. Giegerich, R., Kurtz, S.: From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. Algorithmica 19(3), 331–353 (1997)
16. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
17. Gusfield, D., Stoye, J.: Linear Time Algorithms for Finding and Representing all the Tandem Repeats in a String. J. Comput. Syst. Sci. 69(4), 525–546 (2004)
18. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear Work Suffix Array Construction. J. ACM 53(6), 918–936 (2006)
19. Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing Suffix Arrays in Linear Time. J. Discrete Algorithms 3(2-4), 126–142 (2005)
20. Ko, P., Aluru, S.: Space Efficient Linear Time Construction of Suffix Arrays. J. Discrete Algorithms 3(2-4), 143–156 (2005)
21. Kolpakov, R.M., Kucherov, G.: Finding Maximal Repetitions in a Word in Linear Time. In: FOCS, pp. 596–604 (1999)
22. Kosaraju, S.: Computation of Squares in a String. In: Crochemore, M., Gusfield, D. (eds.) CPM 1994. LNCS, vol. 807, pp. 146–150. Springer, Heidelberg (1994)
23. Manber, U., Myers, E.W.: Suffix Arrays: A New Method for On-line String Searches. SIAM J. Comput. 22(5), 935–948 (1993)
24. McCreight, E.: A Space Economical Suffix Tree Construction Algorithm. J. Assoc. Comput. Mach. 23, 262–272 (1976)
25. Salson, M., Lecroq, T., Léonard, M., Mouchard, L.: A Four-Stage Algorithm for Updating a Burrows-Wheeler Transform. Theor. Comput. Sci. 410(43), 4350–4359 (2009)
26. Salson, M., Lecroq, T., Léonard, M., Mouchard, L.: Dynamic Extended Suffix Arrays. J. Discrete Algorithms 8(2), 241–257 (2010)
27. Shibuya, T.: Constructing the Suffix Tree of a Tree with a Large Alphabet. In: Aggarwal, A.K., Pandu Rangan, C. (eds.) ISAAC 1999. LNCS, vol. 1741, pp. 225–236. Springer, Heidelberg (1999)
28. Tsakalidis, A.K.: Maintaining Order in a Generalized Linked List. Acta Inf. 21, 101–112 (1984)
29. Ukkonen, E.: On-line Construction of Suffix Trees. Algorithmica 14(3), 249–260 (1995)
30. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. 14th Symposium on Switching and Automata Theory, pp. 1–11 (1973)
31. Westbrook, J.: Fast Incremental Planarity Testing. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 342–353. Springer, Heidelberg (1992)

COCA Filters: Co-occurrence Aware Bloom Filters*

Kamran Tirdad, Pedram Ghodsnia, J. Ian Munro, and Alejandro López-Ortiz

Cheriton School of Computer Science
University of Waterloo

Abstract. We propose an indexing data structure based on a novel variation of Bloom filters. Signature files have been proposed in the past as a method to index large text databases though they suffer from a high false positive error problem. In this paper we introduce COCA Filters, a new type of Bloom filters which exploits the co-occurrence probability of words in documents to reduce the false positive error. We show experimentally that by using this technique we can reduce the false positive error by up to 21.6 times for the same index size. Furthermore Bloom filters can be replaced by COCA filters wherever the co-occurrence of any two members of the universe is identifiable.

Keywords: Information Retrieval, Bloom Filters, Signature Files, Locality Sensitive Hash Functions.

1 Introduction

Inverted indices and variants thereof are the preferred data structure currently in use in search engines. However in environments that are very sensitive to index size this method becomes impractical since they require approximately 50% of the size of the corpus for the index file. By compressing the index file and pruning of the less frequent query terms we can reduce the size of inverted indexes down to 10% of the corpus size [30]. In areas where false positive errors are acceptable a more space efficient method called signature files is applicable. With this method it is possible to reduce the size of index file significantly at the cost of precision. Another key advantage of this method is that it can be used in optimizing intersection queries in distributed inverted indices[20,26]. Parallelizability and the simplicity of the insertion are two other benefits of this method that make it a suitable choice for certain environments.

When using signature files a signature is maintained for each document. A signature is basically a sequence of bits. There are several different methods for computing the signature of a document. One of the most common methods is to use a randomized data structure called Bloom filter. In Bloom filter-based signature files it is implicitly assumed that every pair of words is equally likely to appear in the same document while in practice this assumption is not true.

* This work was supported by NSERC of Canada and the Canada Research Chairs program.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 313–325, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In this paper we introduce a new variant of Bloom filters named co-occurrence aware Bloom filters or COCA Filters for short. COCA Filters utilize the probability of the co-occurrences of the words in documents to improve the false positive error. We show through experiments that COCA Filters can reduce the space by up to 75% for the same false positive error or equivalently reduce the false positive error by up to 21 times for the same index size.

Reducing the size of the signature file index or equivalently its false positive error makes COCA Filters ideally suited for applications which are extremely sensitive to the size of the storage.

The rest of the paper is organized as follows: Section 2 reviews related work and background. Section 3 describes the details of our approach and our proposed methods. Section 4 presents the evaluation and analysis of our proposed methods and finally we conclude our work in section 5.

2 Previous Work and Background

Inverted file indices and signature files are two well established indexing methods which have been proposed for large text databases [11,15,30]. Although using the inverted files is more favourable because of its wide range of useful properties in comparison to signature files, Carterette and Can in [11] showed that signature file indexes can be as good as inverted file indexes in special environments where memory is scarce and a given false positive rate is acceptable. Library catalogues, multimedia files with many attributes, medical cross references, and a large lexicon or lists of streets for a GPS system are examples of text databases in which signature file can work faster with less storage. However, the high false positive error rate is one of the critical problems of the signature file method which makes it impractical for many applications.

Signature files are a forward index method which stores a signature for every document. Hashing every single term of a document and concatenating the hash values of the terms can be considered as a simple signature for that document. Alternatively, superimposed coding can be used to create a signature of a document. In this method, hashing every word of the document yields a bit pattern of size m, with k bits set to 1, in which m and k are design parameters. The bit patterns are superimposed (OR-ed) together to form the document signature. Searching for a set of words is handled by creating the signature of each word and OR-ing them together to build the query signature and returning all documents with a matching pattern.

To avoid having document signatures that are flooded with 1s, long documents are divided into smaller blocks, that is, pieces of text that contain a constant number of unique words. Each block of a document gives a block signature and block signatures are concatenated to form the document signature.

Although not explicitly stated in the literature, superimposed coding is a variation of Bloom filters, a well-known randomized data structure first suggested in [5]. A Bloom filter is a probabilistic data structure used to check whether an element belongs to a set with possible false positive error but zero false negative

error. It consists of a bit vector of size m and k independent hash functions $h_1, h_2, ..., h_k$ with ranges of $1, ..., m$. All the bits are initially set to zero. These hash functions can be interpreted as uniform random number generators over the range of $1, ..., m$. For every element of the set, say x, the bits $h_i(x)$ for $(1 \leq i \leq k)$ are set to 1. Some bits of the array might be turned on more than once, but this will not affect the status of the array. To check if an item, say y, is a member of S, the k positions of $h_i(y)$ for $1 \leq i \leq k$ in the array should be checked and if one or more of the k positions are set to 0, it can be assured that y has not been inserted to this array and consequently is not a member of S. If all k positions are set to 1, it is assumed that y is in S. However, there is some probability that this assumption is wrong. Therefore, a Bloom filter may result in a false positive error, also known as false drop error.

Bloom filters have been used in a wide variety of applications in recent years. They are used as spell-checkers [23], as a means of succinctly storing a dictionary of unsuitable passwords for security purposes [28], to speed up semi-join operations [22], for Web cache sharing [16] and in many other areas. In order to support multi-sets, Cohen and Matias introduced spectral Bloom filters [14]. Chazelle et al. in [13] introduce a similar data structure which is called a Bloomier filter in order to approximate functions.

Bloom [5] proved that the false positive probability of the Bloom filter is about $f = (1 - \frac{e^{-kn}}{m})^k$. Recently Bose et al. in [6] showed that the Bloom's formula for false positive is not accurate and gave a proper formula. They also demonstrated that for large enough values of m (size of Bloom filter) with small values of k (number of hash functions), the difference between Bloom's formula and the actual false positive rate is negligible.

To obtain an estimate of the efficiency of Bloom filters, it is good to know the information theoretic lower bound of the size of any data structure that can represent all sets of n elements from a universe with false positive for at most a fraction t of the universe but allows no false negative. Broder et al. [7] showed that to achieve a false positive rate less than t, we must have $m > n\ lg(\frac{1}{t})$ bits. Furthermore, they showed that this lower bound in Bloom filters is $m > n\ lg(e) \cdot lg(\frac{1}{t})$ and consequently argued that space-wise Bloom filters need more than a $lg(e) \simeq 1.44$ factor of the information theoretic lower bound. In [25] Pagh et al. introduced a more complicated data structure that achieved this lower bound.

One of the key observations in both of the aforementioned proofs is the assumption that there is no correlation between any two members of the universe, and any subset of the universe with cardinality of n is equally likely. Now assume that some members of the universe are strongly correlated (i.e. given that one of them belongs to a subset, the probability that the other is also a member of that set is very likely). Intuitively this property of possible subsets can be exploited by using special hash functions which produce more "similar" bit patterns (i.e. with smaller hamming distances) for more correlated members of our set and vice versa. For doing so, we use locality sensitive hash (LSH) functions which are customized to hash similar items to the same hash value with high probability [12].

The LSH algorithm has been used in numerous applied settings from bio-sequence similarity search [10] to audio similarity identification [29] and many other areas [17,19,24]. Min-wise independent permutations is a locality sensitive hashing scheme for a collection of subsets with the similarity function defined as follows:

$$Pr_{h \in F}[h(A) = h(B)] = sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In this setting, hash of a set A is defined as $h(A) = min_{a \in A} \pi(a)$ where π is a permutation which was chosen randomly from a min-wise independent permutation family F. A permutation family F (subset of all n factorial permutations) is *min-wise independent* if for any subset X of $[1...n]$, and any $x \in X$, when π is chosen randomly from F we have $Pr(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}$ [9].

One of the applications of min-wise independent hash functions was suggested by Broder in [8] to detect near duplicate documents over a large set of documents. Broder suggested to consider a set of shingles (contiguous subsequences of words) for each document and choose a set of t independent random permutations $\pi_1, \pi_2, \pi_3, ..., \pi_t$. For each document D, calling its set of shingles SD, he defined the sketch of Document D as $(min_{a \in SD} \pi_1(a); min_{a \in SD} \pi_2(a); ...; min_{a \in SD} \pi_t(a))$. Then, he argued that the sketch of two documents can be used to estimate their resemblance by computing how many corresponding elements in their sketches are equal. In the next section a similar approach is taken in order to reduce the false positive errors of signature files.

3 COCA Filters

Considering the concept of signature files over human readable texts, some terms (members of the universe) are more likely to exist in the same document (set).

In order to exploit this non randomness, it is preferable to modify the k hash functions of the Bloom filter such that "similar" words (i.e. with high co-occurrence ratio) have "similar" bit patterns (i.e. with less hamming distance). For example if two words occur in almost the same set of documents their bit patterns can be designed such that they differ in a few places. More importantly by using these bit patterns, after inserting these two words there would be more bit positions available for the rest of the words in the Bloom filter causing reduction in the average false positive error. This observation can be formalized as the following optimization problem.

Consider two keywords of x and y from the universe of all the words W with corresponding posting lists of X and Y. Furthermore assume that the k bit positions of each of these two terms are stored in the sets of $H(x)$ and $H(y)$. Rather than having k random numbers between 1 to m, the proposed objective is to design hash functions such that:

$$\forall x, y \in W, \frac{|H(x) \cap H(y)|}{|H(x) \cup H(y)|} = \frac{|X \cap Y|}{|X \cup Y|}$$

Note that the right hand side is determined by the corpus and so is fixed. So this problem can be characterized as given N^2 rational numbers p_{ij} design a bipartite graph with m vertices on one side and N vertices the other side such that for every two vertices V_i and V_j where $0 < i, j \leqslant N$ the following constraints hold:

$$\frac{|Neighbour(V_i) \cap Neighbour(V_j)|}{|Neighbour(V_i) \cup Neighbour(V_j)|} = p_{ij}$$

We conjecture that this problem is NP-hard when we are given p_{ij} as a pair I_{ij} (intersection size) and U_{ij} (union size). Here we propose the following ad-hoc probabilistic approach. Define k co-occurrence-aware hash functions of x to be k of the min-wise independent permutations over the set of X. So, the probability that each hash of two distinct terms x and y be equal to each other is $\frac{|X \cap Y|}{|X \cup Y|} = sim(x, y)$. This new data structure is named co-occurrence-aware Bloom filters or in short *COCA filter*.

Assuming that the probability that two different hash functions produce the same bit position for two different words is negligible, the expected value of the difference between the left and right hand side of the objective function for every two term can be calculated as follows:

$$E\left[\left|\frac{|H(x) \cap H(y)|}{|H(x) \cup H(y)|} - \frac{|X \cap Y|}{|X \cup Y|}\right|\right] \simeq \tag{1}$$

$$\left|\frac{k \times sim(x,y)}{2k - k \times sim(x,y)} - sim(x,y)\right| = \tag{2}$$

$$\left|\frac{sim(x,y) \times (sim(x,y) - 1)}{2 - sim(x,y)}\right| \tag{3}$$

The approximation from (1) to (2) is based on the assumption that pairs of sets with large intersections on average have large unions but obviously this is not true in general. Since $sim(x, y) = \frac{|X \cap Y|}{|X \cup Y|}$ and is in $[0, 1]$, with simple algebraic calculations it can be shown that this value is less than 0.172 and more importantly for the pairs of x and y such that $sim(x, y)$ is close to 0 or 1 this value is close to 0. So over the sets that most of the members are strongly co-related or are not related to each other at all this approach can perform very well. Note that the reason this formula is not dependant on k, the number of hash functions, is the implicit assumption that the bit vector is large enough such that it is quite unlikely for two different hash functions to produce the same bit position for two different words.

Note that by doing so, the reduction in the false positive probability for all of the terms in documents happens at the cost of increasing the false positive probability over random terms which do not exist in any of the documents. In the next section we describe three experiments over three different English corpora comparing COCA Filters to traditional Bloom Filters.

In order to implement COCA filters, k min-wise independent permutations should be picked randomly from a min-wise independent family. Since min-wise

independent families are too big for practical applications (in fact it is known that their size is at least $lcm(1, 2, ..., n)$ [9]), variant notions of min-wise independence have been introduced in the literature [21,27].

In our experiments in order to keep the implementation relatively simple two-universal hash functions has been employed to replicate the behaviour of k independent permutations. For a large prime value of p, k random pairs of (a_i, b_i) are generated where $1 \leqslant i \leqslant k$. The hash of each document ID, say x can be calculated by $(a_i * x + b_i) mod p$. Each hash of the document IDs corresponds to one permutation over the set of all document IDs. This procedure is repeated for all the k random pairs so that there are k different permutations. Consequently, for each permutation the minimum value of each posting list is the hash of the corresponding keyword.

In algorithm 1, the pseudocode as explained in the last paragraph shows how to calculate the k hash functions of the COCA filter and store them in k hash tables $h_1, h_2, ..., h_k$.

Algorithm 1. Hash Calculator For COCA Filter

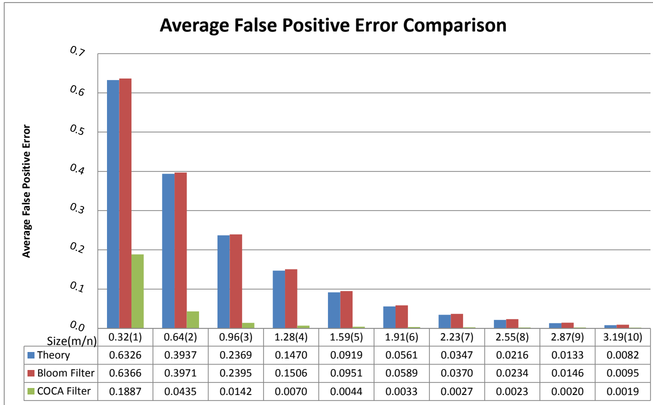
input: Assume documents are numbered from 1 to N and m is the size of the bit vector. The posting list of each term t can be accessed as a set by posting-list(t). k random pairs of (a_i, b_i) where $1 \leqslant i \leqslant k$ are generated.
Output: k hash tables $h_1, h_2, ..., h_k$

1: **for** $i = 1$ **to** N **do**
2:    **for** $j = 1$ **to** k **do**
3:       $perm[i][j] \leftarrow (a_j i + b_j) mod P$
4:    **end for**
5: **end for**
6: **for** every term t in the corpus **do**
7:    **for** $x = 1$ **to** k **do**
8:       $h_x[t] = [min_{num \in posting-list(t)}(perm[num][x])] mod(m)$
9:    **end for**
10: **end for**

4 Experimental Results

In order to evaluate the effectiveness of COCA Filters in reducing the false positive error, we test them experimentally on three collections. The first corpus is a collection of Wikipedia articles [2]. This collection consists of 2000 high quality pages selected by a team of volunteers. For indexing purposes we stripped all HTML tags, Java scripts, comments and other non-related elements from the html files and removed all numbers and words shorter than 3 characters. The total size of the html files is 244 Megabytes and after cleaning the files and removing the duplicates of the words in each file the total size is reduced to 20.4 Megabytes. According to the statistics provided in [4], in Wikipedia, the average number of words per document is about 400. Based on this assumption, each document of the test collection is divided into partitions of size 400 words. After

partitioning each document, the size of its last partition will be less than or equal to 400 words. To address this problem, the number of words in all fragments of each document has been balanced. For example, after partitioning a 700 word document, there will be 2 partitions of size 350 words. The output of this step is 7, 500 partitions with the average size of 350 words per document and 212568 unique terms in total.



Size(m/n)	0.32(1)	0.64(2)	0.96(3)	1.28(4)	1.59(5)	1.91(6)	2.23(7)	2.55(8)	2.87(9)	3.19(10)
Theory	0.6326	0.3937	0.2369	0.1470	0.0919	0.0561	0.0347	0.0216	0.0133	0.0082
Bloom Filter	0.6366	0.3971	0.2395	0.1506	0.0951	0.0589	0.0370	0.0234	0.0146	0.0095
COCA Filter	0.1887	0.0435	0.0142	0.0070	0.0044	0.0033	0.0027	0.0023	0.0020	0.0019

Fig. 1. The comparison of the average false positive error of the COCA filter method with the conventional Bloom filter and the theoretic formula for the sampled Wikipedia corpus. The x-axis shows the index size in Megabytes and the value in the parentheses indicates the $\frac{m}{n}$ ratio.

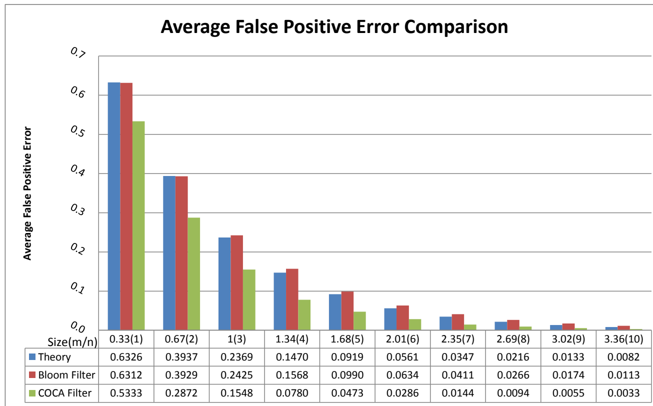
The goal of the experiment is to compare the average false positive error of the proposed hash function with that of the theoretic formula and the conventional Bloom filters. Let W be the set of all words. $FP(d)$ is defined as the number of words in W which are not in document d but its corresponding Bloom filter falsely claims that they are. For each document, the false positive error of its corresponding Bloom filter is defined as $FPE(d) = \frac{FP(d)}{|W|}$. Thus, the average false positive error of a signature file method is $\frac{\sum_{d \in D} FPE(d)}{|D|}$.

In figure 1, the x-axis shows the result of this experiment for different $\frac{m}{n}$ ratios and y-axis shows the corresponding average false positive error. In each method, for each $\frac{m}{n}$ ratio, only the result for the k value which minimizes the false positive error is shown. The total size of the signature file along with the average false positive error is also included in the table below the figure.

From this experiment the following key observations can be derived:

- For all values of $\frac{m}{n}$, the false positive error of conventional Bloom filters is just slightly more than that of the theoretic formula. It confirms the argument of Bose in [6] that Bloom's formula provides only a lower bound for false positive probability. The closeness of the average false positive error of the conventional Bloom filter and the theoretic formula justifies the validity of our implementation of the conventional Bloom filters as well.

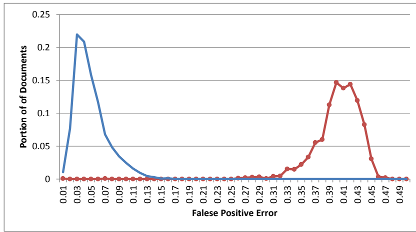
– For all values of $\frac{m}{n}$, the false positive error of our proposed methods is significantly better than the false positive ratio predicted by the conventional Bloom filter and the theoretic formula.
– In some cases there is about 21 times improvement in the average false positive error. For example, in $\frac{m}{n} = 6$, the average false positive error of the COCA filter is about 21 times better than the Bloom filter.
– In some cases there is up to a 75% reduction in the size of the index for the same average false positive error. For example if the objective is to achieve an average false positive error less than 0.19 in conventional Bloom filters the $\frac{m}{n}$ ratio should be at least 4 while in COCA filter with the $\frac{m}{n}$ ratio of 1 the average false positive error of 0.19 is achievable. In other words for every bit that is used in the COCA filter, 3 extra bits are required in a conventional Bloom filter. Note that when $\frac{m}{n} = 1$ the index size is only 1.6% of the polished corpus.



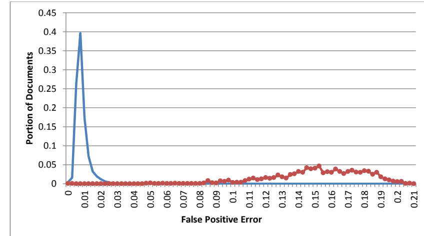
Size(m/n)	0.33(1)	0.67(2)	1(3)	1.34(4)	1.68(5)	2.01(6)	2.35(7)	2.69(8)	3.02(9)	3.36(10)
Theory	0.6326	0.3937	0.2369	0.1470	0.0919	0.0561	0.0347	0.0216	0.0133	0.0082
Bloom Filter	0.6312	0.3929	0.2425	0.1568	0.0990	0.0634	0.0411	0.0266	0.0174	0.0113
COCA Filter	0.5333	0.2872	0.1548	0.0780	0.0473	0.0286	0.0144	0.0094	0.0055	0.0033

Fig. 2. The comparison of the average false positive error of the COCA filter method with the conventional Bloom filter and the theoretic formula for the sampled Google corpus. The x-axis shows the index size in Megabyte and the value in the parentheses indicates the $\frac{m}{n}$ ratio.

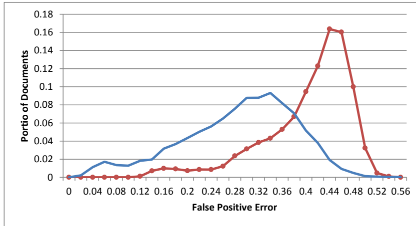
Our proposed approach is based on the co-occurrence of the words in documents and therefore is sensitive to the correlation of documents. In order to investigate the relationship between the degree of correlation among documents and the improvements in average false positive error, the previous experiment was repeated over a collection of weakly-correlated web pages. This collection is a random selection of 900,000 web pages released by Google in 2002 for a programming contest [1]. We chose about 13500 samples from this collection randomly and performed the previous experiment on the resulting collection. We used the same method as the first experiment for cleaning the documents and fragmenting them. Due to the smaller average size of documents in this



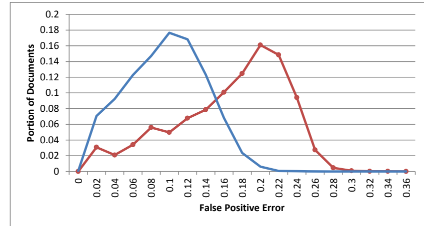
(a) Distribution of false positive error for $\frac{m}{n} = 2$ for the Wikipedia collection



(b) Distribution of false positive error for $\frac{m}{n} = 4$ for the Wikipedia collection



(c) Distribution of false positive error for $\frac{m}{n} = 6$ for the Google collection



(d) Distribution of false positive error for $\frac{m}{n} = 6$ for the Google collection

Fig. 3. Comparison of the distribution of the false positive error of the COCA Filter and the conventional Bloom filter. In each graph the left curve corresponds to the COCA filter and the right curve corresponds to the conventional Bloom filter.

collection, we divided the documents into partitions of size 100. After partitioning, the collection had about 35200 documents with the average size of 80 terms and 228715 terms in total.

Figure 2 shows the result of this experiment. Although COCA Filter is still better than conventional Bloom Filter, the improvement in this experiment is not as good as the first experiment. The result of this experiments confirms the sensitivity of our proposed method to the correlation among the terms of the documents.

In the first experiment we chose a collection of high quality articles of Wikipedia which are all coherent in writing and have a scientific theme. It is normal to encounter many synonyms of a word instead of a repetition and there is also a somewhat predictable set of antonyms to follow. On the contrary, in the second corpus documents are not coherent neither in meaning nor in the style which results to have lots of random terms from street names and addresses to gene sequences and peoples' names in them. Moreover the diversity of the topics that these terms are covering is higher than the first collection and this diversity reduces the probability of the co-occurrence of the words in documents and consequently reduces the effectiveness of our proposed method.

To ensure that the size of corpus does not have a negative effect on the quality of COCA Filters we repeated the first experiment with a similar but larger collection of Wikipedia documents [3]. This collection is a more comprehensive

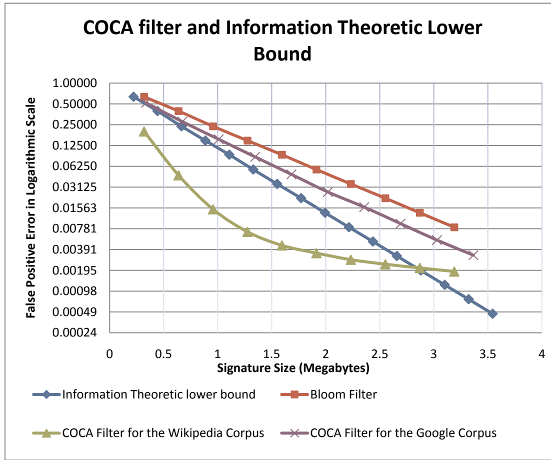


Fig. 4. The Comparison of Information Theoretic lower bound with COCA filters over two different corpora

Table 1. Comparison of the average false positive error of COCA filter over two wikipedia corpora with different sizes

m/n	1	2	3	4	5	6	7	8	9	10
Wikipedia(6500)	0.18913	0.04369	0.01193	0.00476	0.00264	0.00179	0.00142	0.00116	0.00099	0.00087
Wikipedia(2000)	0.18869	0.04346	0.01415	0.00703	0.00439	0.00331	0.00272	0.00234	0.00204	0.00185

version of the first collection and consists of 6500 high quality pages selected by a team of volunteers for school students. The size of this collection is more than 3 times the size of the first collection but it is very similar to the first collection in terms of coherency and writing style. We used the same method as the first experiment for cleaning the documents and fragmenting them. After partitioning, the collection had 21543 documents with the average size of 350 words per document and 321500 unique terms in total. Table 1 compares the result of this experiment with the result of the first experiment. It shows that increasing the size of the collection does not increase the average false positive error given that the coherency and style of the corpus remains the same. It confirms that the higher average false positive error of COCA Filters on Google collection is not because of the larger size of this collection and it is only due to the non-coherent, random nature and diversity of that collection.

One area of concern is whether in COCA Filters the average false positive error decreases at the cost of having many bloom filters with low false positive error and many with high false positive error (i.e. having a bimodal distribution). Figure 3 illustrates a comparison between the distribution of the false positive error in the COCA Filter and the conventional Bloom filter for $\frac{m}{n} = 2$, 4 over the Wikipedia and Google corpora. In each graph the left curve corresponds to the COCA filter and the right curve corresponds to the conventional Bloom filter. In all graphs, in both curves, by increasing the distance from the average,

the frequency of documents decreases rapidly. It shows that there are only a few documents with a false positive error significantly greater than (or smaller than) the average false positive error. It can be seen that in the Wikipedia corpus which has higher correlation even the worst false positive error of the COCA filter method is significantly better than the best false positive error of the conventional Bloom filter method while in the Google corpus this property does not hold. Moreover, in the Wikipedia corpus the deviation of the COCA filter curve from the average is much smaller than its corresponding Bloom filter curve while in the other corpus this is not easily observable.

Another interesting comparison is between the COCA filter and the information theoretic lower bound on the three corpora as suggested in [7]. In other words we want to compare our method in terms of space efficiency with the best possible randomized data structure which does not utilize the co-occurrence probability of the words. Figure [4] illustrates this comparison. Note that the y-axis is the average false positive error in logarithmic scale in order to demonstrate the difference more clearly. While the COCA filter for the Google corpus never beats the information theoretic lower bound, the COCA filter for the Wikipedia corpus beats it in most of the cases by a significant margin. Note that as the false positive error gets closer to zero the distance between the curves shows a smaller difference. Interestingly as the correlation among the terms of the corpus gets stronger the rate of the decrease in false positive error tends to be hyper-exponential (as in Wikipedia corpus) rather than exponential (in Google Corpus) but as the index size increases the improvement rate decreases until it becomes very close to Bloom filter. This shows that for these applications where the elements of the corpus are highly correlated, utilizing the extra information about this correlation can be very valuable.

5 Conclusion and Future Work

In this paper the problem of false positive error of Bloom filters has been addressed and a novel technique to reduce the false positive error is proposed. The effectiveness of this approach was evaluated by conducting two experiments and our experimental results showed that up to 21.6 times improvement in false positive error or equivalently up to 75% reduction in space is achievable. Although this improvement is surprisingly good it is important to note that this technique is very sensitive to the correlation among the terms of the documents in the corpus.

In the current definition of the similarity function the size of each posting list can not affect the similarity of any two words as long as the ratio of the intersection and the union of their corresponding posting list is the same. It would be interesting to investigate similarity functions which are sensitive to the size of the posting lists as well.

Finding the information theoretic lower bound for the minimum number of bits required for a Bloom filter, given the extra information of the co-occurrence probability of each pairs of the members of the universe is another avenue for research.

More recently a particular type of memory called ternary content addressable memory (TCAM) was used to replicate a set of Bloom filters in order to solve the subset query problem for small sets [18]. Another potential opportunity is to explore the possible positive effect of COCA filters in areas where TCAM is used as a group of Bloom filters.

References

1. http://www.google.com/programming-contest (2002) (accessed, January 2011)
2. http://www.wikipediaondvd.com/site.php (2007) (accessed, January 2011)
3. http://schools-wikipedia.org (2008) (accessed, January 2011)
4. http://en.wikipedia.org/wiki/Wikipedia:Words_per_article (2009) (accessed, January 2011)
5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 422–426 (1970)
6. Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M.H.M., Tang, Y.: On the false-positive rate of bloom filters. Inf. Process. Lett. 108(4), 210–213 (2008)
7. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. In: Internet Mathematics, pp. 636–646 (2002)
8. Broder, A.: Identifying and filtering near-duplicate documents. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 1–10. Springer, Heidelberg (2000)
9. Broder, A.: Min-wise independent permutations: Theory and practice. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, p. 808. Springer, Heidelberg (2000)
10. Buhler, J., Tompa, M.: Finding motifs using random projections. Journal of Computational Biology 9(2), 225–242 (2002)
11. Carterette, B., Can, F.: Comparing inverted files and signature files for searching a large lexicon. Inf. Process. Manage. 41(3), 613–633 (2005)
12. Charikar, M.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388 (2002)
13. Chazelle, B., Kilian, J., Rubinfeld, R., Tal, A.: The bloomier filter: an efficient data structure for static support lookup tables. In: SODA, pp. 30–39 (2004)
14. Cohen, S., Matias, Y.: Spectral bloom filters. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD 2003, pp. 241–252. ACM, New York (2003)
15. Faloutsos, C., Christodoulakis, S.: Signature files: an access method for documents and its analytical performance evaluation. ACM Trans. Inf. Syst. 2, 267–288 (1984)
16. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. 8, 281–293 (2000)
17. Georgescu, B., Shimshoni, I., Meer, P.: Mean shift based clustering in high dimensions: A texture classification example. In: ICCV, pp. 456–463 (2003)
18. Goel, A., Gupta, P.: Small subset queries and bloom filters using ternary associative memories, with applications. SIGMETRICS Perform. Eval. Rev. 38, 143–154 (2010)
19. Haveliwala, T.H., Gionis, A., Indyk, P.: Scalable techniques for clustering the web. In: WebDB (Informal Proceedings), pp. 129–134 (2000)

20. Li, J., Loo, B., Hellerstein, J., Kaashoek, M., Karger, D., Morris, R.: On the feasibility of peer-to-peer web indexing and search. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 207–215. Springer, Heidelberg (2003)
21. Matousek, J.: On restricted min-wise independence of permutations (2002)
22. Mullin, J.: Optimal semijoins for distributed database systems. IEEE Transactions on Software Engineering 16(5), 558–560 (1990)
23. Mullin, J.K., Margoliash, D.J.: A tale of three spelling checkers. Softw. Pract. Exper. 20, 625–630 (1990)
24. Ouyang, Z., Memon, N.D., Suel, T., Trendafilov, D.: Cluster-based delta compression of a collection of files. In: WISE, pp. 257–268 (2002)
25. Pagh, A., Pagh, R., Rao, S.S.: An optimal bloom filter replacement. In: SODA 2005, pp. 823–829 (2005)
26. Reynolds, P., Vahdat, A.: Efficient peer-to-peer keyword searching. In: Endler, M., Schmidt, D.C. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 21–40. Springer, Heidelberg (2003)
27. Saks, M., Srinivasan, A., Zhou, S., Zuckerman, D.: Low discrepancy sets yield approximate min-wise independent permutation families. Information Processing Letters 73(1-2), 29–32 (2000)
28. Spafford, E.H.: Opus: Preventing weak password choices. Computers & Security 11(3), 273–278 (1992)
29. Yang, C.: Macs: music audio characteristic sequence indexing for similarity retrieval. In: 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics, pp. 123–126 (2001)
30. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38(2) (2006)

On-Line Construction of Position Heaps

Gregory Kucherov

Université Paris-Est & CNRS, Laboratoire d'Informatique Gaspard Monge,
Marne-la-Vallée, France
Gregory.Kucherov@univ-mlv.fr

Abstract. We propose a simple linear-time on-line algorithm for constructing a position heap for a string [EMOW11]. Our definition of position heap differs slightly from the one proposed in [EMOW11] in that it considers the suffixes ordered in the descending order of length. Our construction is based on classic suffix pointers and resembles the Ukkonen's algorithm for suffix trees [Ukk95]. Using suffix pointers, the position heap can be extended into the augmented position heap that allows for a linear-time string matching algorithm [EMOW11].

1 Introduction

The theory of string algorithms developed beautiful data structures for string matching and text indexing. Among them, *suffix tree* and *suffix array* are most widely used structures, providing efficient solutions for a wide range of applications [CR94, Gus97]. The DAWG (*Directed Acyclic Word Graph*) [BBH+85], also known as *suffix automaton* [Cro86], is another elegant structure that can be used both as a text index [BBH+85] or as a matching automaton [Cro88, CR94].

Recently, a new *position heap* data structure was proposed [EMOW11]. Similar to the suffix tree, DAWG or suffix array, position heap allows for a preprocessing of a text string in order to efficiently search for patterns in it. As for the above-mentioned data structures, a position heap for a string of length n can be constructed in time $O(n)$. Then all locations of a pattern of length m can be found in time $O(m + occ)$, where occ is the number of occurrences.

The construction algorithm of [EMOW11] processes the string from right to left, like the Weiner's algorithm does for suffix trees [Wei73]. Moreover, the construction requires a so-called dual heap, which is an additional trie on the same set of nodes. The position heap and its dual heap are constructed simultaneously.

To obtain a linear-time pattern matching algorithm of [EMOW11], the position heap should be post-processed in order to add some additional information, resulting in the *augmented position heap*. Most importantly, this information includes so-called *maximal-reach pointers* assigned to certain nodes. Computing these pointers makes use of the dual heap too.

In this paper, we propose a different construction of the position heap. First, we change the definition of the position heap by reversing the order of suffixes and thus allowing for a left-to-right traversal of the input string. The modified definition, however, preserves good properties of the position heap and does not

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 326–337, 2011.
© Springer-Verlag Berlin Heidelberg 2011

affect the string matching algorithm proposed in [EMOW11]. For this modified definition, we propose an *on-line* algorithm for constructing the position heap. Our algorithm does not use the dual heap, replacing it by classic *suffix pointers* used for constructing suffix trees by the Ukkonen's algorithm [Ukk95] or for constructing the DAWG [BBH+85]. Our algorithm is simple and has some similarity with the Ukkonen's algorithm for suffix trees, as opposed to the Weiner's algorithm. We deliberately use some terminology of the Ukkonen's algorithm to underline this similarity.

We further show that the augmented position heap can be easily constructed using suffix pointers. Thus, we completely eliminate the use of the dual heap, replacing it by suffix pointers for constructing both the position heap and its augmented version. Even if this replacement does not provide an immediate improvement in space or running time, we believe that our construction is conceptually simpler and more natural.

Throughout the paper, we assume given a constant-size alphabet A. Positions of strings over A are numbered from 1, that is, a string w of *length* k is $w[1]\dots w[k]$. The length k of w is denoted by $|w|$. $w[i..j]$ denotes substring $w[i]\dots w[j]$.

A *trie* (term attributed to Fredkin [Fre60]) is a simple natural data structure for storing a set of strings. It is a tree with edges labeled by alphabet letters, such that for any internal node, the edges leading to the children nodes are labeled by distinct letters. In this paper, we assume the edges to be directed towards leaves, and call an edge labeled by a letter a an a-edge. A *label* of a node (*path label*) is the string formed by the letters labeling the edges of the path from the root to this node. Given a trie, a string w is said to be represented in the trie if it is a path label of some node. The corresponding node will then be denoted by $\overline{w}$.

2 Definition of Position Heap

To define position heaps, we first need to introduce the *sequence hash tree* proposed by Coffman and Eve back in 1970 [CE70] as a data structure for implementing hash tables. Assume we are given an ordered set of strings $W = \{w_1, \dots, w_n\}$ and assume for now that no w_i is a prefix of w_j for any $j < i$. The sequence hash tree for W, denoted $SHT(W)$, is a trie defined by the following iterative construction. We start with the tree $SHT_0(W)$ consisting of a single root node *root*[1]. We then construct $SHT(W)$ by processing strings $w_1, \dots, w_k$ in this order and for each w_i, adding one node to the tree. By induction, assume that $SHT_i(W)$ is the sequence hash tree for $\{w_1, \dots, w_i\}$. To construct $SHT_{i+1}(W)$, we find the shortest prefix v of w_{i+1} which is *not* represented in $SHT_i(W)$. Note that by our assumption, such a prefix always exists. Let $v = v'a$, $a \in A$, i.e. v' is the longest prefix of w_{i+1} represented in $SHT_i(W)$. Then $SHT_{i+1}(W)$ is

[1] This definition agrees with the definition of [CE70] but is slightly different from that of [EMOW11] which defines the root to store w_1. The difference is insignificant, however.

obtained from $SHT_i(W)$ by adding a new node as a child of v' connected to v' by an a-edge and pointing to w_{i+1}. After inserting all strings of W, we obtain $SHT(W)$, that is $SHT(W) = SHT_k(W)$. Thus, $SHT(W)$ is a trie of $n+1$ nodes such that a node pointing to w_i is labeled by some prefix of w_i. Note that the size of the sequence hash tree depends only on the number of strings in the set and does not depend on the length of these words. An example of sequence hash tree is given on Figure 1.

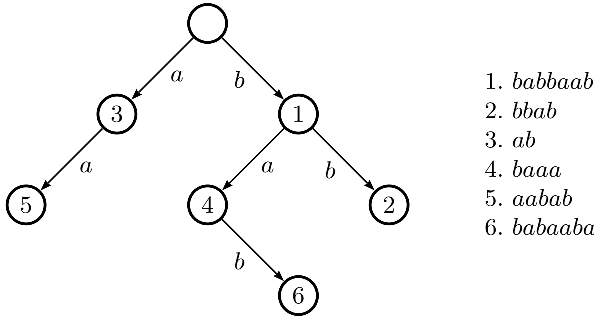


Fig. 1. Sequence hash tree for the set of words shown on the right. Each node stores the rank of the corresponding word in the set.

We now define the *position heap* of a string T. In [EMOW11], the position heap for T is defined as the sequence hash tree for the set of suffixes of T, where *the suffixes are ordered in the ascending order of length*, i.e. from right to left. This insures, in particular, the condition that no suffix is a prefix of a previously inserted suffix, and then no suffix is already represented in the position heap at the time of its insertion.

In this paper, we define the position heap of T to be the sequence hash tree for the set of suffixes of T, where *the suffixes are ordered in the descending order of length*, i.e. from left to right. From now on, we stick to this order. An immediate observation is that the assumption of the suffix hash tree does not hold anymore, and it may occur that an inserted suffix is already represented in the position heap by an existing node. One easy way to cope with this is to systematically assume that T is ended by a special sentinel symbol \$, like it is generally assumed for the suffix tree.

On the other hand, as we will be interested in an on-line construction of the position heap, we will still need to construct the position heap for strings without the ending sentinel symbol. For that, we have to slightly change the definition of sequence hash tree of a set W, by allowing one node to point to several strings of W. The definition of the position heap extends then to any string, with the only difference that inserting a suffix may no longer lead to the creation of a node, but to inserting a new pointer to an existing node. This feature, however, will be used in a very restricted way, as the following observation shows.

Lemma 1. *Let W be a set of distinct strings. Then every node of $SHT(W)$ points to at most two strings of W.*

Proof. Straightforward from the definition of $SHT(W)$ and the fact that all strings are distinct. □

As a consequence of Lemma 1, a position heap contains two types of nodes, pointing respectively to one and two suffixes of T. The former will be called *regular nodes* and the latter *double nodes*. We naturally assume that a pointer to a suffix is simply the starting position of that suffix, therefore regular and double nodes store one and two string positions respectively. Hereafter we interchangeably refer to "suffixes" and "positions" when the underlying string is unambiguously defined.

Figure 2 provides an example of a position heap.

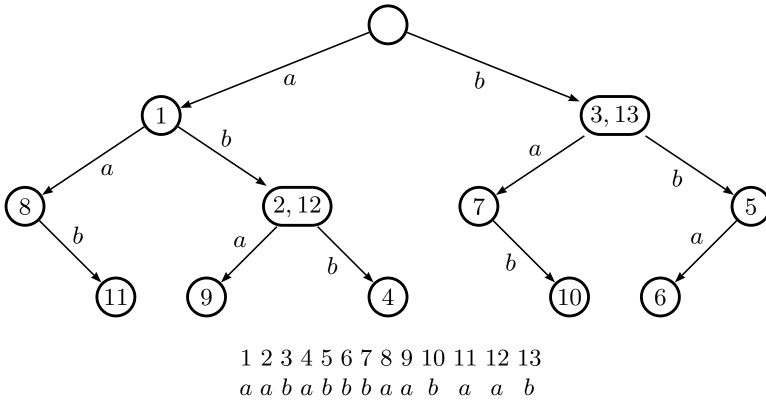


Fig. 2. Position heap for string *aababbbaabaab*. Double nodes store pairs of positions.

3 Properties of Position Heap

Denote by $PH(T)$ the position heap for a string $T[1..n]$ constructed as defined in the previous section. In the following theorem, we summarize some key properties of the position heap. Property *(i)* is a straightforward from the definition, and properties *(ii)*-*(iv)* have been established in [EMOW11] but remain valid for our definition of position heap when suffixes are inserted from left to right.

Theorem 1 ([EMOW11]). *Consider $PH(T[1..n])$. The following properties hold.*

(i) *A substring $T[i..j]$ is represented in $PH(T)$ iff there exist in T occurrences of strings $T[i]$, $T[i..i+1]$, $T[i..i+2]$, ..., $T[i..j]$, appearing in this order.*

(ii) *The labels of all nodes of $PH(T)$ form a factorial set. That is, if a string is represented in $PH(T)$, all its substrings are represented too.*

(iii) The depth of $PH(T)$ is no more than $2h(T)$, where $h(T)$ is the length of the longest substring w of T which occurs $|w|$ times in T (possibly with overlap).

(iv) If a string w occurs in T at least $|w|$ times, then w is represented in $PH(T)$. Inversely, if w is not represented in $PH(T)$ and w' is the longest prefix of w which is represented, then w cannot occur in T more than $|w'|$ times.

Properties *(iii)* and *(iv)* show that the position heap of a string "adapts" to frequencies of its substrings. In particular, if a string is "frequent" (occurs as many times as it is long), then it is necessarily represented in the position heap. On the other hand, if it is not represented, it has less occurrences than its length. The latter property is crucial for obtaining a linear-time string matching algorithm of [EMOW11].

4 On-Line Construction Algorithm

Let us have a closer look at the properties of double nodes of a position heap $PH(T)$. Each such node stores two positions i, j of T. Assume $i < j$, then positions i and j will be called the *primary* and the *secondary* positions respectively.

Lemma 2. *If j is the secondary position of some node of a position heap, then so is $j+1$.*

Proof. Consider $PH(T)$ for some string $T[1..n]$. Assume i, j, $i < j$, are respectively primary and secondary positions of some node. This means that by the time the suffix $T[j..n]$ is inserted into $PH(T)$ during its construction, node $\overline{T[j..n]}$ already exists. By Theorem 1*(ii)*, node $\overline{T[j+1..n]}$ exists too. *A fortiori*, node $\overline{T[j+1..n]}$ exists when $T[j+1..n]$ is inserted into $PH(T)$. Therefore, $j+1$ becomes the secondary position of that node after the insertion of suffix $T[j+1..n]$. $\square$

Lemma 2 implies that all positions of $T[1..n]$ are split into two intervals: primary positions $[1..s-1]$, for some position s, and secondary positions $[s..n]$. Position s will be called *active secondary position*, or *active position* for short.

Assume we have constructed the position heap $PH(T[1..k])$ for some prefix $T[1..k]$ of the input string $T[1..n]$. Let us analyze the differences between $PH(T[1..k])$ and $PH(T[1..k+1])$ and the modifications that need to be made to transform the former into the latter.

Let s be the active position of $T[1..k]$. First observe that for suffixes $1, \ldots, s-1$, no changes need to be made. Inserting each suffix $T[i..k]$ for $1 \leq i \leq s-1$ into $PH(T[1..k])$ led to the creation of a new node. This means that by the time this suffix was inserted into $PH(T[1..k])$, some prefix $T[i..\ell]$ of $T[i..k]$, $\ell \leq k$, was not represented in the position heap, which led to the creation of a new node $\overline{T[i..\ell]}$ with the minimal such ℓ. This shows that inserting suffixes $1, \ldots, s-1$ involve completely identical steps in the construction of both $PH(T[1..k])$ and $PH(T[1..k+1])$.

The situation is different for the secondary positions $s, \ldots, k$. Each suffix $T[i..k]$ for $s \le i \le k$ was already represented in $PH(T[1..k])$ at the moment of its insertion, and then resulted in the addition of the secondary position i to the node $\overline{T[i..k]}$. When inserting the corresponding suffix $T[i..k+1]$ into the position heap $PH(T[1..k+1])$, two cases arise. In the *first case*, inserting the suffix $T[i..k+1]$ leads to the creation of the new node $\overline{T[i..k+1]}$ if this node does not exist yet. Position i then becomes the primary position of this new node. Observe that this only occurs when $PH(T[1..k])$ does not contain an $T[k+1]$-edge outgoing from the node $\overline{T[i..k]}$. It is easily seen that such an edge cannot appear by the time of insertion of $T[i..k+1]$ into $PH(T[1..k+1])$ if it is not already present in $PH(T[1..k])$. In the *second case*, node $\overline{T[i..k]}$ has an outgoing $T[k+1]$-edge in $PH(T[1..k])$, and in the construction of $PH(T[1..k+1])$, the secondary position i stored in this node should be "moved" to the child node $\overline{T[i..k+1]}$. It becomes then the secondary position of this node.

Observe now that if for a secondary position i, the corresponding node $\overline{T[i..k]}$ has an outgoing $T[k+1]$-edge, then so does the node $\overline{T[i+1..k]}$ storing the secondary position $i+1$. This can again be seen from the factorial property of the position heap (Theorem 1*(ii)*). This shows that the above two cases split the interval of secondary positions $[s..k]$ into two subintervals $[s..t-1]$ and $[t..k]$, such that node $\overline{T[i..k]}$ does not have an outgoing $T[k+1]$-edge for $i \in [s..t-1]$ and does have such an edge for $i \in [t..k]$.

The above discussion is summarized in the following lemma specifying the changes that have to be made to transform $PH(T[1..k])$ into $PH(T[1..k+1])$.

Lemma 3. *Given $T[1..n]$, consider $PH(T[1..k])$ for $k < n$. Let s be the active secondary position, stored in the node $\overline{T[s..k]}$. Let $t \ge s$ be the smallest position such that node $\overline{T[t..k]}$ has an outgoing $T[k+1]$-transition. To obtain $PH(T[1..k+1])$, $PH(T[1..k])$ should be modified in the following way:*

(i) for every node $\overline{T[i..k]}$, $s \le i \le t-1$, create a new child linked to $\overline{T[i..k]}$ by a $T[k+1]$-edge. Delete secondary position i from the node $\overline{T[i..k]}$ and assign it as a primary position to the new node $\overline{T[i..k+1]}$,
(ii) for every node $\overline{T[i..k]}$, $i \ge t$, move the secondary position i from node $\overline{T[i..k]}$ to node $\overline{T[i..k+1]}$.

We describe now the algorithm implementing the changes specified by Lemma 3. We augment $PH(T)$ with *suffix pointers* f defined in the usual way:

Definition 1. *For each node $\overline{T[i..j]}$ of $PH(T)$, a* suffix pointer *is defined by $f(\overline{T[i..j]}) = \overline{T[i+1..j]}$.*

Note that the definition is sound, as the node $\overline{T[i+1..j]}$ exists whenever the node $\overline{T[i..j]}$ exists, according to Theorem 1*(ii)*. For the root node, it will be convenient for us to define $f(root) = \perp$, where $\perp$ is a special node such that there is an a-edge between $\perp$ and *root* for every $a \in A$ (cf [Ukk95]). Figure 3 shows the position heap of Figure 2 supplemented by suffix pointers.

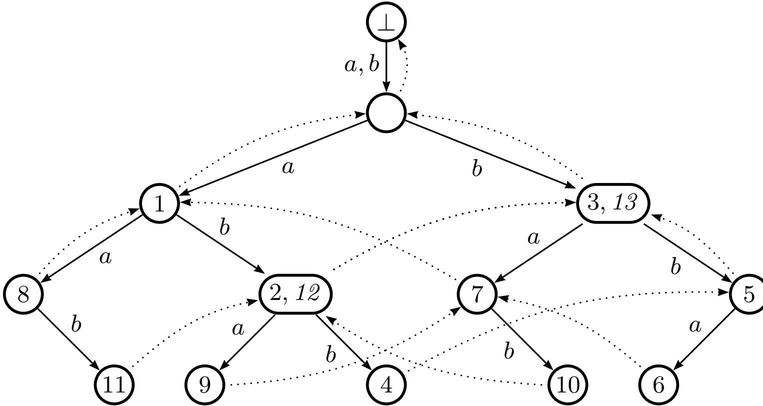


Fig. 3. Position heap for string *aababbbaabaab* with suffix pointers (dotted arrows). Secondary positions are shown in italic.

We now begin to describe the on-line construction algorithm for $PH(T)$, given a text $T[1..n]$. Consider the node $\overline{T[s..k]}$ of $PH(T[1..k])$ storing the active secondary position s, that we call the *active node*. If the active secondary position does not exist (i.e. there is no secondary positions at all), then the active node is *root* and the active position is set to $k+1$. Observe that the nodes storing the other secondary positions $s+1, s+2, \ldots, n$ can be reached, in order, by following the chain of suffix pointers $f(\overline{T[s..n]}), f(f(\overline{T[s..n]})), \ldots$ until the root node is reached. Figure 3 provides an illustration.

This leads us to the main trick of our construction: *we will not store secondary positions at all, but only memorize the active secondary position and the active node*. The secondary positions can be easily recovered by traversing the chain of suffix pointers starting from the active node and incrementing the position counter after traversing each edge. Note also that if the input string T is ended by a unique sentinel symbol, the resulting position heap does not contain any secondary nodes and there is no need to recover them.

Having in mind that the secondary positions are not stored explicitly, the transformation of $PH(T[1..k])$ into $PH(T[1..k+1])$ specified by Lemma 3 is done by the following simple procedure. Starting from the active node, the algorithm traverses the chain of suffix pointers as long as the current node does not have an outgoing $T[k+1]$-edge. For each such node, a new node is created linked by a $T[k+1]$-edge to the current node. A suffix pointer to this new node is set from the previously created new node. Once the first node with an outgoing $T[k+1]$-edge is encountered, the algorithm moves to the node this edge leads to, sets the suffix pointer to this node, and assigns this node to be the active node for the following iteration. The correctness of the last assignment is stated in the following lemma.

Lemma 4. *Consider $PH(T[1..k])$ and let s be the active position, and $t \geq s$ be the smallest position such that node $\overline{T[t..k]}$ has an outgoing $T[k+1]$-edge. Then node $\overline{T[t..k+1]}$ is the active node of $PH(T[1..k+1])$.*

Proof. As it follows from Lemma 3, t is the largest secondary position of $T[1..k+1]$. □

Algorithm 1 provides a pseudo-code of the algorithm.

Algorithm 1. On-line construction of the position heap $PH(T[1..n])$

```
 1: create states root and ⊥
 2: f(root) ←⊥
 3: for all a ∈ A do
 4:    set an a-edge from ⊥ to root
 5: end for
 6: currentnode ← root
 7: currentsuffix ← 1
 8: for i = 1 to n do
 9:    lastcreatednode ← undefined
10:    while currentnode does not have an outgoing T[i]-edge do
11:       create a new node newnode pointing to currentsuffix
12:       set a T[i]-edge from currentnode to newnode
13:       if lastcreatednode ≠ undefined then
14:          f(lastcreatednode) ← newnode
15:       end if
16:       lastcreatednode ← newnode
17:       currentnode ← f(currentnode)
18:       currentsuffix ← currentsuffix + 1
19:    end while
20:    move currentnode to the target node of the outgoing T[i]-edge
21:    if lastcreatednode ≠ undefined then
22:       f(lastcreatednode) ← currentnode
23:    end if
24: end for
```

The correctness of Algorithm 1 follows from Lemmas 3, 4 and the discussion above. It is instructive, in addition, to observe the following:

- it is easily seen that the suffix pointers of $T[1..k+1]$ are correctly updated. Indeed, the algorithm assigns to $\overline{T[i..k+1]}$ a suffix pointer to $\overline{T[i+1..k+1]}$ which is obviously correct. Note that for the active position s of $T[1..k]$, the created node $\overline{T[s..k+1]}$ does not get pointed to by any suffix pointer, which is correct, as $T[s-1..k+1]$ is not represented in $PH(T[1..k+1])$: the position $s-1$ is primary in $T[1..k]$ and therefore the node $\overline{T[s-1..k]}$, if it exists in $PH(T[1..k])$, does not get extended by a $T[k+1]$-edge (cf Lemma 3).
- since the depth of $\overline{T[s..k]}$ (s is the active position) in $PH(T[1..k])$ is $k+1-s$ and a traversal of a suffix link decrements the depth by 1 and increments the current position by 1, it follows that if the traversal of the suffix chain

reaches the root node, the active position value becomes $k+1$, which is exactly what we need to start processing the next letter $T[k+1]$. This shows why Algorithm 1 correctly maintains *currentsuffix* and never needs to reset it at the beginning of the **for**-loop iteration.

It is easy to see that the running time of Algorithm 1 is linear in the length n of the input string. Since each iteration of the **while**-loop creates a node, this loop iterates exactly n times over the whole run of the algorithm. Trivially, the **for**-loop iterates n times too, and all the involved operations are constant time. Thus, the whole algorithm takes $O(n)$ time. The following theorem concludes the construction.

Theorem 2. *For an input string $T[1..n]$, Algorithm 1 correctly constructs $PH(T)$ on-line in time $O(n)$.*

5 Augmented Position Heap

Assume we have a text $T[1..n]$ for which we constructed the position heap $PH(T)$. We don't assume that T is ended by a unique letter, and therefore some nodes of $PH(T)$ are double nodes and store two positions of T, one primary and one secondary. Here we assume that the secondary positions *are* actually stored (or can be retrieved in constant time for each node). As explained in Section 4, even if the secondary positions are not stored during the construction of $PH(T)$, they can be easily recovered once the construction is completed.

[EMOW11] proposed a linear-time string matching algorithm using $PH(T[1..n])$, i.e. an algorithm that computes all occurrences of a pattern string in T in time $O(m + occ)$, where m is the pattern length and occ the number of occurrences. Describing this elegant algorithm is beyond the scope of this paper, we refer the reader to [EMOW11] for its description. Note that the algorithm itself applies without changes to our definition of position heap, as it does not depend in any way on the order that the suffixes of T are inserted.

However, the algorithm of [EMOW11] runs on $PH(T)$ enriched with some additional information. Let $\bar{i}$ denote the node of $PH(T)$ storing position i, $1 \leq i \leq n$. The extended data structure, called the *augmented position heap*, should allow the following queries to be answered in constant time:

- given a position i, retrieve the node $\bar{i}$,
- given two nodes $\bar{i}$ and $\bar{j}$, is $\bar{i}$ a (not necessarily immediate) ancestor of $\bar{j}$?
- given a position i of T, retrieve the node $\overline{T[i..i+\ell]}$, where $T[i..i+\ell]$ is the longest substring of T starting at position i and represented in $PH(T)$.

To answer the first query, [EMOW11] simply introduces an auxiliary array storing, for each position i, a pointer to the node $\bar{i}$. Maintaining this array during the construction of $PH(T)$ by Algorithm 1 is trivial: once a position is assigned to a newly created node (line 11 of Algorithm 1), a new entry of the array is created. If T is not ended by a unique symbol and then the final $PH(T)$ has secondary positions, those are easily recovered by traversing the chain of suffix pointers at the very end of the construction.

The second query can be also easily answered in constant time after a linear-time preprocessing of $PH(T)$. A solution proposed in [EMOW11] consists in traversing $PH(T)$ depth-first and storing, for each node, its discovery and finishing times [CLR99]. Then node $\bar{i}$ is an ancestor of node $\bar{j}$ if and only if the discovery and finishing time of $\bar{i}$ is respectively smaller and greater than the discovery and finishing time of $\bar{j}$.

A more space-efficient solution would be to use a balanced parenthesis representation of the tree topology of $PH(T)$, taking $2n$ bits, and link each node to the corresponding opening parenthesis. Then the corresponding closing parenthesis can be retrieved in constant time by the method of [MR01] using $o(n)$ auxiliary bits. This allows ancestor queries to be answered in constant time.

The third type of queries is answered by a precomputed function, called *maximal-reach pointer* [EMOW11]: for a position i of $T[1..n]$, define $mrp(i)$ to be the node $\overline{T[i..i+\ell]}$, where $T[i..i+\ell]$ is the longest prefix of $T[i..n]$ represented in $PH(T)$. Observe first that if i is a secondary position, then $mrp(i) = \bar{i}$. This is because a secondary position i is stored in node $\overline{T[i..n]}$, which trivially corresponds to the longest prefix starting at i. Therefore, as it is done in [EMOW11], mrp can be represented by pointers from node $\bar{i}$ to node $mrp(i)$ whenever these nodes are different. In our case, we have then to keep in mind that a maximal-reach pointer from a double node applies to the primary position of this node. Figure 4 provides an illustration.

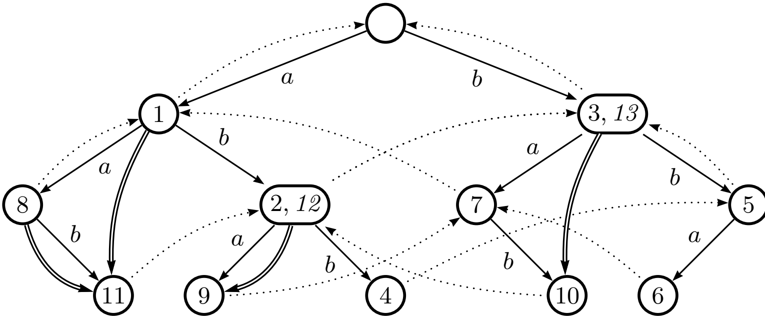


Fig. 4. Position heap for string *aababbbaabaab* with suffix pointers and maximal-reach pointers mrp (double arrows). Only values for which $mrp(i) \neq \bar{i}$ are shown, namely $mrp(1) = \overline{11}$, $mrp(8) = \overline{11}$, $mrp(2) = \bar{9}$, $mrp(3) = \overline{10}$. Note that maximal reach pointers outgoing from double nodes are unambiguous as for all secondary positions i, we have $mrp(i) = \bar{i}$.

In [EMOW11], maximal-reach pointers are computed by an extra traversal of $PH(T)$, using an auxiliary *dual heap* structure on top of it (see Introduction). Here we show that maximal-reach pointers can be easily computed using suffix pointers instead of the dual heap. Thus, we completely get rid of the dual heap for constructing the augmented position heap, replacing it with suffix pointers.

We compute $mrp(i)$ iteratively for $i = 1, 2, \ldots, s-1$, where s is the active secondary position of $T[1..n]$. Assume we have computed $mrp(i)$ for some i and

have to compute $mrp(i+1)$. Assume $mrp(i) = \overline{T[i..i+\ell]}$. It is easily seen that $T[i+1..i+\ell]$ is a prefix of the string represented by $mrp(i+1)$. To compute $mrp(k+1)$, we follow the suffix link $f(mrp(k))$ to reach $\overline{T[i+1..i+\ell]}$ and then keep extending the prefix $T[i+1..i+\ell]$ as long as it is represented in $PH(T)$. The resulting pseudo-code is given in Algorithm 2.

It is very easy to see that Algorithm 2 works in time $O(n)$: the **while**-loop makes exactly n iterations overall, as each iteration increments the *readhead* counter.

The following property of Algorithm 2 is useful to observe: as soon as *readhead* gets the value $n+1$ (line 6), the node *currentnode* gets assigned to the active node of $PH(T[1..n])$ (line 9); at the subsequent iterations, the algorithm simply traverses the chain of suffix links and sets the maximal-reach pointer for each secondary position to be the node storing this position (lines 8-9).

Algorithm 2. Linear-time computation of maximal-reach pointers $mrp(i)$

1: $currentnode \leftarrow root$
2: $readhead \leftarrow 1$
3: **for** $i = 1$ **to** n **do**
4:   **while** *currentnode* has an outgoing $T[readhead]$-edge **and** $readhead \leq n$ **do**
5:   **move** *currentnode* **to the target node of the outgoing** $T[readhead]$**-edge**

6: $readhead \leftarrow readhead + 1$
7:   **end while**
8: $mrp(i) \leftarrow currentnode$
9: $currentnode \leftarrow f(currentnode)$
10: **end for**

6 Concluding Remarks

We proposed a construction algorithm of a position heap of a string, under a modified definition of position heap compared to [EMOW11]. In contrast with the algorithm of [EMOW11] that processes the text right-to-left, our algorithm reads the string left-to-right and has the on-line property. Drawing a parallel to suffix trees, our algorithm can be compared to the Ukkonen's on-line algorithm [Ukk95], while the algorithm of [EMOW11] can be compared to the Weiner's algorithm [Wei73]. The similarity of our algorithm to the Ukkonen's algorithm goes beyond this parallel, as the two algorithms are also somewhat analogous in their design.

The $O(n)$ complexity bounds of both Algorithm 1 (Theorem 2) and Algorithm 2 are stated for a constant-size alphabet, otherwise a correcting factor $\log |A|$ should be introduced, similarly to the suffix tree construction.

The position heap is a smaller data structure than the suffix tree: it contains exactly $n+1$ nodes whereas the suffix tree has n leaves and then up to $2n$ nodes. Still, the position heap allows for a linear-time string matching. The authors of [EMOW11] proposed algorithms for updating the position heap when

the input string undergoes modifications (character insertions/deletions). These algorithms can be easily applied to our definition of position heap. Other interesting applications of position heap are still to be discovered.

It would be interesting to study further the properties of maximal-reach pointers. Note that their structure differs between our definition of position heap and the definition of [EMOW11]. It would be also interesting to exploit the "adaptiveness" of position heaps to substring frequencies, mentioned in Section 3.

References

[BBH+85] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science 40, 31–55 (1985)

[CE70] Coffman, E., Eve, J.: File structures using hash functions. Communications of the ACM 13, 427–432 (1970)

[CLR99] Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press, Cambridge (1999)

[CR94] Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press, Oxford (1994)

[Cro86] Crochemore, M.: Transducers and repetitions. Theoretical Computer Science 45, 63–86 (1986)

[Cro88] Crochemore, M.: String matching with constraints. In: Koubek, V., Janiga, L., Chytil, M.P. (eds.) MFCS 1988. LNCS, vol. 324, pp. 44–58. Springer, Heidelberg (1988)

[EMOW11] Ehrenfeucht, A., McConnell, R., Osheim, N., Woo, S.-W.: Position heaps: A simple and dynamic text indexing data structure. CPM 2009 Lille 9(1), 100–121 (2011); Preliminary version in Proc. 20th Anniversary Edition of the Annual Symposium on Combinatorial Pattern Matching (CPM 2009)

[Fre60] Fredkin, E.: Trie memory. Communications of the ACM 3(9), 490–499 (1960)

[Gus97] Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)

[MR01] Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)

[Ukk95] Ukkonen, E.: On-line construction of suffix-trees. Algorithmica 14(3), 249–260 (1995)

[Wei73] Weiner, P.: Linear pattern matching algorithm. In: 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Computing All Subtree Repeats in Ordered Ranked Trees

Michalis Christou[1], Maxime Crochemore[1,2], Tomáš Flouri[3], Costas S. Iliopoulos[1,4], Jan Janoušek[3], Bořivoj Melichar[3], and Solon P. Pissis[1]

[1] Dept. of Informatics, King's College London, UK
[2] Université Paris-Est, France
[3] Dept. of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Czech Republic
[4] Digital Ecosystems & Business Intelligence Institute, Curtin University, Perth, Australia

Abstract. We consider the problem of finding all subtree repeats in a given ordered ranked tree. Specifically, we transform the given tree to a string representing its postfix notation, and then propose an algorithm based on the bottom-up technique. The proposed algorithm is divided into two phases: the preprocessing phase, and the phase where all subtree repeats are computed. The linear runtime of the algorithm, as well as the use of linear auxiliary space, are important aspects of its quality.

1 Introduction

Tree pattern matching has been intensively studied over the past decades because of its various applications, among others, in mechanical theorem proving, term-rewriting, instruction selection, non-procedural programming languages, and computational biology [4, 8, 11–13].

In many applications, it is essential to extract the repeated patterns in a tree within a mathematical structure [3, 5, 10]. In particular, the *common subtrees* problem consists of finding all of the subtrees having the same structure and the same labels on the corresponding nodes of two ordered labeled unranked trees [7]. This problem of equivalence, which is strictly related to the *common subexpression* problem [3, 5], arises, for instance, in the code optimization phase of compiler design, or in saving storage for symbolic computations [1, 3, 5].

In this article, we consider a slightly different problem, and provide a completely different solution to what has been done so far. We focus on finding all subtree repeats – the subtrees occurring more than once – in a tree structure. This problem is analogous to the well-known problem of finding all the repetitions in a given word [2]. Notice that finding all subtree repeats can be solved by the algorithm presented in [7]. However, the presented solution requires the construction of a suffix tree, which is expensive in practical terms. Moreover, by analogy with standard suffix automata and repeats in strings, all subtree repeats in a tree can be directly computed by analyzing states of the deterministic subtree pushdown automaton, which represents a full index of a tree for all

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 338–343, 2011.
© Springer-Verlag Berlin Heidelberg 2011

subtrees [9]. This way of computing all subtree repeats in a tree can be found in [14], leading, however, to an $O(n \log n)$-time complexity.

The proposed algorithm is divided into two phases: the preprocessing phase and the phase where all subtree repeats are computed. The preprocessing phase transforms the given tree to a string representing its postfix notation, and then computes arrays that store the height of each node of the tree, the parent of each node, and an indicator showing whether a node is the leftmost (first) child of its parent or not. The second phase, for computing all subtree repeats, is done in a bottom-up manner, using a partitioning technique. The linear runtime of the algorithm, as well as the use of linear auxiliary space, are important aspects of its quality.

2 Preliminaries

We define an *alphabet* Σ as a finite, non-empty set of symbols. A string is a succession of zero or more symbols from an alphabet Σ. A string w is a *factor* (or substring) of x if $x = uwv$ for $u, v \in \Sigma^*$, and is represented as $w = x_i \dots x_j, 1 \leq i \leq j \leq |x|$. A *ranked alphabet* is a couple $\mathcal{A} = (\Sigma, \varphi)$, where Σ is an alphabet and φ is the mapping $\varphi : \Sigma \mapsto \mathbb{N}$. The *arity* (rank) of a symbol $x \in \Sigma$ is denoted by $\varphi(x)$, the number of children of the node that has x as a symbol.

We define an ordered tree as in [6]. A tree t is *unordered* if no ordering is given on the edge lists of its nodes. A tree t is *labeled* if every node $f \in N$ is labeled by a symbol $a \in \Sigma$, Σ a finite alphabet. A tree t is ranked if for every node $f \in N$ its out-degree is given. The number of nodes of a tree t is denoted by $|t|$. The height of a tree t, denoted by *Height(t)*, is defined as the maximal length of a path from the root of t to a leaf of t.

The *postfix notation* $post(t)$ of a labeled, ordered, ranked tree t is obtained by applying the following *Step* recursively, beginning at the root of t:
Step: Let this application of *Step* be node v. If v is a leaf, list v and return to the previous node. If v is an internal node having descendants $v_1, v_2, \dots, v_{\varphi(v)}$, apply *Step* to $v_1, v_2, \dots, v_{\varphi(v)}$ in that order and then list v.

Two ranked trees are equal iff their postfix notations are equal strings. A subtree p *matches* tree T at node v, if p is equal to the subtree of t rooted at v.

A *subtree repeat* p in a tree T, represented by its postfix notation $x = post(T)$, is a tuple: $M_{x,u} = (p; i_1, i_2, \dots, i_r), r \geq 2$, where $i_1 < i_2 < \dots < i_r$ and $u = x_{i_1} \dots x_{i_1+|p|-1} = x_{i_2} \dots x_{i_2+|p|-1} = \dots = x_{i_r} \dots x_{i_r+|p|-1} = post(p)$. If the tuple includes all the occurrences of u in x, then $M_{x,u}$ is said to be *complete* and is written $M^*_{x,u}$.

We formally define the problem of computing all subtree repeats in ordered ranked trees, as follows.

Problem 1 (All subtree repeats of an unlabeled ordered ranked tree). Find all complete subtree repeats of an unlabeled ordered ranked tree T consisting of n nodes.

3 Properties of Ranked Trees in Postfix Notation

Lemma 1. *Given a tree t and its postfix notation $post(t)$, the postfix notations of all subtrees of t are factors of $post(t)$.*

Definition 1. *Let $w = a_1 a_2 \dots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the* arity checksum $ac(w) = \sum_{i=1}^{m} \varphi(a_i) - m + 1$.

Lemma 2. *Let $post(t)$ and w be a tree t in postfix notation and a factor of $post(t)$, respectively. Then $w = w_1 \dots w_{|w|}$, $w_i \in \mathcal{A}$, is the postfix notation of a subtree of t iff:*

- *w is composed by one symbol of arity 0, or*
- *$|w| \geq 2$, $ac(w) = 0$ and w_1 is a symbol of arity 0 which corresponds to a leaf being the first child of some subtree of t.*

4 Algorithms

In this section we present an algorithm for solving Problem 1. It consists of two phases: the preprocessing phase and the searching phase.

4.1 Preprocessing

Given a tree t, its postfix notation is first computed using a simple postorder traversal of the tree. Let $post(t) = x_1 x_2 \dots x_n$ be the postfix notation of tree t. The preprocessing phase completes by computing 3 auxiliary arrays, which will be accessed during the searching phase.

1. The height of each subtree of t, having node x_i as its root, is stored in an array H, at position i, for $1 \leq i \leq n$.
2. An array P of n elements, with the *i-th* element having the value p if x_p is the parent of x_i.
3. A binary array *FC* consisting of 1's and 0's, with the *i-th* element being 1 in case x_i is the first (leftmost) child of its parent node $x_{P[i]}$.

4.2 Finding Subtree Repeats

We are now in a position to present Alg. 1 for solving Problem 1. The computation is based on a bottom-up traversal of the input tree t, described by its postfix notation $post(t) = x_1 x_2 \dots x_n$. The algorithm, at each step (level) i, locates and outputs all subtree repeats of height i by expanding the leaves of the tree as suggested by Lemma 2. We also introduce an auxiliary array called *level array* (*LA*), which keeps track of queues of triplets. These triplets describe factors of $post(t)$, and are of the form (S, ℓ, ac), where S is a set containing the starting positions of the occurrences of the factor, ℓ is the factor's length, and ac is its arity checksum, which, in case is 0, indicates that the factor corresponds to some subtree of t.

Algorithm 1. SUBTREE-REPEATS

Input : $post(t) = x_1 x_2 \dots x_n$ over ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$
Output: Sets of starting positions of factors of $post(t)$ and their lengths, representing subtrees from t

```
1  sc ← 1
2  for i ← 1 to n do
3      if φ(x_i) = 0 then
4          S ← S ∪ {i}
5          T[i] ← sc
6          TL[i] ← 1
7      else
8          T[i] ← 0
9          TL[i] ← 0
10 OUTPUT (S, 1)
11 ASSIGN-LEVEL((S, 1, 0))
12 for i ← 1 to H[n] do
13     while not empty LA[i] do
14         PARTITION(DEQUEUE(LA[i]))
```

The algorithm starts by constructing a triplet $(S, 1, 0)$, representing all leaves, i.e. subtrees of height 0 (lines 3-6 of Alg. 1), with its set S containing all positions of the unary symbol in $post(t)$. The triplet is then passed to the function ASSIGN-LEVEL, which splits the elements of S in several subsets, according to the height of the subtree specified by the parent (stored in $H[P[root]]$) of each element in S (line 4 of ASSIGN-LEVEL). Elements which do not correspond to subtrees being first children of their parent nodes are discarded. The resulting subsets are then wrapped into triplets and appended in the appropriates queues of LA according to $H[P[root]]$. Note, that the function ASSIGN-LEVEL takes as input only triplets describing factors that correspond to subtrees.

Function. ASSIGN-LEVEL(E)

Input : $E = (S, \ell, ac)$
Output: Partitioning of E in levels

```
1 for each i ∈ S do
2     root = i + TL[i] − 1
3     if FC[root] = 1 then
4         S_{H[P[root]]} ← S_{H[P[root]]} ∪ {i}
5         L ← L ∪ {H[P[root]]}
6 for i ∈ L do ENQUEUE(LA[i], (S_i, ℓ, 0))
```

At each step i of the algorithm, the triplets in $LA[i]$ are passed to the function PARTITION, which partitions a triplet according to the next factor, starting at position r (line 2 of PARTITION), that is to be concatenated with the factor described by the triplet. The next factor either represents a subtree of t in

Function. PARTITION(E)

Input : $E = (S, \ell, ac)$, $post(t) = x_1x_2 \dots x_n$
Output: Partitioning E in classes

```
1  for i ∈ S do
2      r = i + ℓ
3      if T[r] ≠ 0 then
4          S_{T[r]} ← S_{T[r]} ∪ {i}
5          E_{T[r]} ← (S_{T[r]}, ℓ + TL[r], ac − 1)
6          L ← L ∪ {E_{T[r]}}
7      else
8          S_{x_r} ← S_{x_r} ∪ {i}
9          E_{x_r} ← (S_{x_r}, ℓ + 1, ac − 1 + φ(x_r))
10         L ← L ∪ {E_{x_r}}
11 for each E_i = (S_i, ℓ_i, ac_i) ∈ L do
12     if ac_i = 0 then
13         OUTPUT(S_i, ℓ_i)
14         sc = sc + 1
15         for j ∈ S_i do
16             T[j] ← sc
17             TL[j] ← ℓ_i
18         ASSIGN-LEVEL(E_i)
19     else
20         PARTITION(E_i)
```

postfix notation, in case its first symbol marks the beginning of a subtree (lines 3-6), or a single symbol (lines 7-10). The triplets are then recursively partitioned (line 20) until they describe factors representing subtrees in postfix notation (lines 12-18). When a triplet finally describes a factor representing a subtree, starting positions of those factors are assigned an index (stored in array T), and their lengths are stored in an array TL (lines 16-17). This is to indicate that the factor starting at position i and having length $TL[i]$ was found to correspond to a subtree. Those triplets are then passed to ASSIGN-LEVEL, which partitions them in levels and stores them in the appropriate queues of LA. As stated before, we consider only the elements i of the set S of a triplet $E = (S, \ell, 0)$, such that the subtree corresponding to the factor $x_i \dots x_{i+\ell}$ is the first child of the subtree specified by its parent node (Lemma 2). The algorithm terminates when the level array LA is empty.

Theorem 1. *The algorithm* SUBTREE-REPEATS *computes all complete subtree repeats of a given tree* t *in* $\Theta(n)$ *time, where* $|t| = n$.

Proof. The preprocessing phase, i.e. the computation of $post(t)$, arrays P, H and FC, is done in $\Theta(n)$ time. During the expansion of the subtrees, performed in function PARTITION, the algorithm does not read a symbol more than once, but rather reads the previously expanded subtrees. Merging the subtrees is done in $n - 1$ operations (number of children of the tree). □

5 Conclusion

In this article, we have formally defined the problem of computing all subtree repeats in ordered ranked trees, and presented a new algorithm based on the bottom-up technique. The proposed algorithm runs in linear time and space. It is important to note that the proposed algorithm can be easily modified to compute all subtree repeats in labeled ordered ranked trees in linear time and space.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2006)
2. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inf. Process. Lett. 12(5), 244–250 (1981)
3. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM 27, 758–771 (1980)
4. Dubiner, M., Galil, Z., Magen, E.: Faster tree pattern matching. J. ACM 41, 205–213 (1994)
5. Flajolet, P., Sipala, P., Steyaert, J.M.: Analytic variations on the common subexpression problem. In: Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming, pp. 220–234. Springer-Verlag New York, Inc., New York (1990)
6. Flouri, T., Janoušek, J., Melichar, B.: Subtree matching by pushdown automata. Computer Science and Information Systems/ComSIS 7(2), 331–357 (2010)
7. Grossi, R.: On finding common subtrees. Theor. Comput. Sci. 108(2), 345–356 (1993)
8. Hoffmann, C.M., O'Donnell, M.J.: Pattern matching in trees. J. ACM 29, 68–95 (1982)
9. Janousek, J.: String suffix automata and subtree pushdown automata. In: Holub, J., Zdárek, J. (eds.) Stringology. pp. 160–172. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (2009)
10. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC 1972, pp. 125–136. ACM, New York (1972)
11. Kosaraju, S.R.: Efficient tree pattern matching. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, pp. 178–183. IEEE Computer Society, Washington, DC (1989)
12. Kuboyama, T.: Matching and Learning in Trees. Ph.D. thesis, University of Tokyo (2007)
13. Mauri, G., Pavesi, G.: Algorithms for pattern matching and discovery in rna secondary structure. Theor. Comput. Sci. 335, 29–51 (2005)
14. Melichar, B.: Arbology: Trees and pushdown automata. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 32–49. Springer, Heidelberg (2010)

Sparse Spatial Selection for Novelty-Based Search Result Diversification

Veronica Gil-Costa[1], Rodrygo L.T. Santos[2],
Craig Macdonald[2], and Iadh Ounis[2]

[1] Universidad Complutense de Madrid, Spain and Yahoo! Research Latin America
gvcosta@yahoo-inc.com
[2] University of Glasgow, UK
{rodrygo,craigm,ounis}@dcs.gla.ac.uk

Abstract. Novelty-based diversification approaches aim to produce a diverse ranking by directly comparing the retrieved documents. However, since such approaches are typically greedy, they require $O(n^2)$ document-document comparisons in order to diversify a ranking of n documents. In this work, we propose to model novelty-based diversification as a similarity search in a sparse metric space. In particular, we exploit the triangle inequality property of metric spaces in order to drastically reduce the number of required document-document comparisons. Thorough experiments using three TREC test collections show that our approach is at least as effective as existing novelty-based diversification approaches, while improving their efficiency by an order of magnitude.

1 Introduction

Search result diversification has emerged as an effective approach for tackling ambiguous queries. In particular, a diverse ranking aims to satisfy as many *aspects* of an ambiguous query as possible, and as early as possible. By satisfying multiple query aspects, a high *coverage* of these aspects is achieved. By having different aspects satisfied as early as possible, a high *novelty* is also attained [10].

Promoting coverage is typically more efficient than promoting novelty: while coverage can be estimated for different documents independently, the same is not true for novelty. In particular, the notion of novelty entails a dependence between the relevance of different documents—i.e., a novel document is one that covers aspects not covered by the other documents. As a result, novelty-based diversification becomes essentially the problem of finding a set of documents that together cover most of the aspects of a query at a given rank cutoff. In this general formulation, this is an NP-hard problem [1]. Most of the approaches proposed in the literature for this problem deploy a greedy approximation algorithm: at each iteration, the algorithm selects a document that covers the most aspects not yet covered by the documents selected in the previous iterations. In a typical case, after the system retrieves n documents to be diversified, this greedy algorithm performs $O(n^2)$ document-document comparisons—i.e., $O(n)$ similarity searches across n iterations [7]—which can severely impact the efficiency of these approaches.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 344–355, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In this paper, we propose to reduce the number of required similarity computations in novelty-based diversification approaches, by modelling novelty in a metric space [7]. Metric spaces have been used for similarity search in many modern database applications [2]. Similarity search in metric spaces focuses on retrieving objects which are similar to a query point, with a metric distance function measuring the objects' similarity. By representing the retrieved documents as m-dimensional vectors in a metric space, we can exploit the triangle inequality property of such spaces to dramatically reduce the number of similarity computations required to diversify these documents.

A number of metric space search algorithms have been proposed in the literature (e.g., [3, 16–18]). In this paper, we show that effective and efficient diversification can be obtained using a sparse spatial selection approach [3], which selects pivot documents from the result set at running time, in order to reduce the number of required similarity computations. Although metric spaces have been used for image search diversification [14], to the best of our knowledge, our approach is the first attempt to leverage the properties of such spaces for diversifying textual documents. Moreover, while the data structure used in [14] has a $O(n^2)$ construction time [15], the one used in this paper can be built in linear time.

The contributions of this paper are two-fold: (1) we propose to model novelty-based diversification as a sparse spatial selection over a metric space; (2) we thoroughly investigate the effectiveness and efficiency of our proposed approach, using three standard TREC test collections for diversity evaluation. Our experimental results attest both the effectiveness and the efficiency of our approach compared to existing novelty-based diversification approaches.

In Section 2, we review existing approaches for search result diversification and similarity search in metric spaces. Section 3 shows how novelty-based diversification can be modelled in a metric space. Sections 4 and 5 detail our experimental setup and evaluation, respectively. Conclusions follow in Section 6.

2 Background and Related Work

In this section, we review existing approaches to search result diversification (Section 2.1) and similarity search in metric spaces (Section 2.2).

2.1 Search Result Diversification

Diversification approaches can be broadly classified as *implicit* or *explicit*. Implicit approaches assume that different documents will cover different query aspects. As a result, these approaches promote novel documents as a means to indirectly cover multiple aspects. The definition of a 'novel' document is precisely what distinguishes the approaches in this family. For instance, Carbonell and Goldstein [4] proposed to compare documents based on their cosine similarity. Zhai et al. [23] proposed an extension of this idea, by comparing documents with respect to the divergence of their language models. More recently, Wang and Zhu [22] proposed to use the correlation of documents' relevance scores.

Instead of assuming that different documents cover different aspects, explicit diversification approaches directly model these aspects as part of their strategy. For instance, Agrawal et al. [1] proposed a diversification approach based on an explicit representation of query aspects as taxonomy classes, in order to promote documents that cover classes also covered by the query. A similar approach was proposed by Carterette and Chandar [5], but with query aspects represented as topic models built from the top retrieved results for the query. Finally, Santos et al. [20] proposed to represent the aspects underlying a query as 'sub-queries'. In their approach, documents are promoted according to their estimated relevance to multiple sub-queries, as well as to the estimated importance of each sub-query.

Although having the same goal, these two families of approaches deploy rather distinct strategies. While implicit diversification approaches are driven by novelty, explicit ones usually target coverage. Since coverage can be estimated independently for different documents, explicit approaches are generally more efficient than implicit ones. In this paper, we propose to reduce the overhead incurred by document-document comparisons in novelty-based diversification approaches. In particular, we model novelty seeking within a metric space, and exploit the properties of this space to efficiently identify novel documents.

2.2 Search in Metric Spaces

Metric spaces are useful to represent complex data objects, such as documents or images, in a searchable collection. Search queries are represented as objects of the same type as the objects in the collection wherein, for example, one is interested in retrieving the most similar objects to the query. Formally, a *metric space* $(\mathcal{U}, \delta)$ comprises a universe of objects $\mathcal{U}$ and a *distance function* $\delta : \mathcal{U} \times \mathcal{U} \rightarrow \mathcal{R}^+$, which determines the similarity between any pair of objects [7]. The definition of the distance function depends on the type of the objects being compared. In an m-dimensional vector space—a particular case of metric spaces in which every object is represented by a vector of m real coordinates—δ could be a distance function of the family $L_s(x, y) = (\sum_{1 \leq i \leq m} |x_i - y_i|^s)^{\frac{1}{s}}$. For example, $s = 2$ yields the Euclidean distance. For any $x, y, z \in \mathcal{U}$, the function δ holds several properties: non-negativity ($\delta(x, y) \geq 0$), reflexivity ($\delta(x, y) = 0$ iff $x = y$), symmetry ($\delta(x, y) = \delta(y, x)$), and the triangle inequality ($\delta(x, z) \leq \delta(x, y) + \delta(y, z)$). The latter property is of particular interest, as it can be used to improve efficiency by avoiding unnecessary similarity computations, as will be shown in Section 3.

The finite subset $\mathcal{X} \subseteq \mathcal{U}$, with $n = |\mathcal{X}|$, denotes the working set of objects where searches are performed (e.g., the top-n documents retrieved for a query). A type of similarity search of particular interest to this work involves *range queries* [15]. In this search type, the goal is to retrieve all objects within distance r to a query object q, where r denotes the *search range*. Fig. 1 shows how the triangle inequality property of metric spaces can be exploited to avoid unnecessary similarity computations for range queries. In particular, given the distance $\delta(x, y)$ between the objects x and y and the distance $\delta(q, x)$ between x and a query object q with search range r, we can avoid computing $\delta(q, y)$.

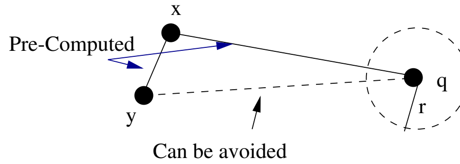


Fig. 1. The triangle inequality property. Once the distances $\delta(x, y)$ and $\delta(q, x)$ are computed, computing $\delta(q, y)$ for a query q with range r is unnecessary.

Most algorithms for similarity search in metric spaces fall into one of two categories: clustering and pivoting. Clustering techniques divide the working set of objects into groups (called clusters), such that similar objects fall into the same group [7]. Pivoting techniques select some objects as pivots, calculate the distance between every other object and each pivot, and apply the triangle inequality to avoid unnecessary similarity computations between objects. A key challenge for pivoting techniques is to determine the number of pivots needed to cover all objects in the working set. Moreover, the number of pivots tends to increase with the size of the working set. In the next section, we show how to adapt a pivot technique that avoids these problems in order to effectively and efficiently promote novel search results in the ranking.

3 Sparse Spatial Selection Diversification

Let $\mathcal{D}$ contain the documents initially retrieved for a query q. Existing diversification approaches typically re-score a document $d_i \in \mathcal{D}$ in light of the query q and the documents in $\mathcal{S} \subseteq \mathcal{D} \setminus \{d_i\}$, according to the following abstract model [21]:

$$\mathrm{score}(q, d_i) = (1 - \lambda)\,\mathrm{rel}(q, d_i) + \lambda\,\mathrm{div}(q, d_i, \mathcal{S}), \tag{1}$$

where relevance ($\mathrm{rel}(q, d_i)$) and diversity ($\mathrm{div}(q, d_i, \mathcal{S})$), as estimated by a given diversification approach, are traded off through the interpolation parameter λ.

In Equation (1), the relevance component $\mathrm{rel}(q, d_i)$ can be estimated using any standard retrieval model. In a novelty-based diversification approach, the diversity component $\mathrm{div}(q, d_i, \mathcal{S})$ is typically estimated in a greedy, iterative fashion. In particular, at any given iteration, every document $d_i \in \mathcal{D} \setminus \mathcal{S}$ is compared to every document $d_j \in \mathcal{S}$, where $\mathcal{S}$ comprises the documents selected in the previous iterations. This way, the document d_i that differs most from the already selected documents in $\mathcal{S}$ is itself included in $\mathcal{S}$. Such document-document comparisons are usually performed as distance computations in an m-dimensional term-frequency space, where m is the number of unique terms in the underlying document collection. As discussed in Section 2.1, these approaches differ mainly in their choice of a distance function (e.g., cosine [4], divergence [23], or correlation [22]). Regardless of the chosen distance function, these approaches require $O(n^2)$ distance computations to diversify a list of n documents. In particular, they perform an $O(n)$ similarity search across n iterations.

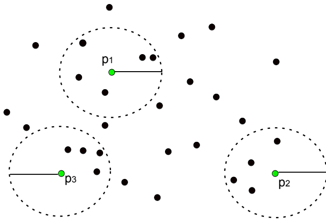


Fig. 2. Objects in the range of pivots p_1, p_2, and p_3 are considered redundant

SSSD1$[q, \mathcal{D} = \{d_1, \ldots, d_n\}, \delta, M, \phi]$

1 $\mathcal{P} \Leftarrow \{d_1\}$
2 **for all** $d_i \in \mathcal{D} \setminus \{d_1\}$ **do**
3 **if** $\delta(d_i, p_j) \geq \phi M \; \forall p_j \in \mathcal{P}$ **then**
4 $\mathcal{P} \Leftarrow \mathcal{P} \cup \{d_i\}$
5 **end if**
6 **end for**
7 $\mathcal{P} \Leftarrow \mathcal{P} \cup (\mathcal{D} \setminus \mathcal{P})$

Alg. 1. Single-Step Sparse Spatial Selection Diversification (SSSD1)

In order to reduce the quadratic number of distance computations incurred by the existing greedy novelty-based diversification approaches, we propose to exploit a key property of metric spaces, namely, the triangle inequality. Our approach is based on an efficient pivoting similarity search algorithm. As illustrated in Fig. 2, the Sparse Spatial Selection (SSS) algorithm [3] identifies a set of k "pivots" among the n objects in the search space. By pre-computing the distances between the k pivots and the n objects, the number of subsequent distance computations can be drastically reduced. For instance, suppose we want to find all objects similar to an object x within a range r. If, for some pivot p, it holds that $|\delta(x, p) - \delta(y, p)| > r$, then we know, from the triangle inequality, that $\delta(x, y) > r$. Therefore, we do not need to explicitly evaluate $\delta(x, y)$.

In this paper, we propose a novelty-based diversification approach inspired by the SSS pivoting search algorithm. Our novel Sparse Spatial Selection Diversification (SSSD) approach incorporates the notion of pivots to reduce the number of distance computations required to diversify a set of documents. In particular, we develop two variants of SSSD. Our first variant, SSSD1, builds upon the SSS algorithm to skip redundant documents in the ranking. As described in Alg. 1, SSSD1 takes as input a query q, an initial set of documents $\mathcal{D}$ retrieved for this query, a distance function δ with upper-bound M, and the search radius ϕ, with $0 \leq \phi \leq 1$, such that $r = \phi M$ determines the search range of each pivot.

The core of SSSD1 is the selection of pivots (lines 1-6 in Alg. 1). To this end, let $(\mathcal{U}, \delta)$ be a metric space, with $\mathcal{D} \subseteq \mathcal{U}$ comprising the documents retrieved for the query q. The pivot set $\mathcal{P}$ is initialised with the first retrieved document, i.e., $d_1 \in \mathcal{D}$. For each remaining document $d_i \in \mathcal{D} \setminus \{d_1\}$, d_i is chosen as a new pivot if its distance to every pivot in $\mathcal{P}$ is greater than or equal to the search range ϕM. Hence, a retrieved document becomes a new pivot if and only if it is located outside the search range of all current pivots. Moreover, documents within the search range of an already selected pivot are considered redundant and skipped, and are later added to the bottom of the ranking (line 7), in the same order as they were originally retrieved in the initial ranking $\mathcal{D}$.

Importantly, during the selection of pivots, it is not necessary that all documents in $\mathcal{D}$ be compared against all pivots. When a document d_i is compared against a pivot p_j and does not satisfy the condition $\delta(d_i, p_j) \geq \phi M$, this document is discarded and no additional comparisons are required. In the best case

scenario, documents are compared only with the first pivot when they are within the range of this pivot. Assuming that an unseen document d_i has a constant probability $\nu = f(\mathcal{U}, \delta, M, \phi)$ of lying outside the range of all pivots $p_j \in \mathcal{P}$ given the metric space $(\mathcal{U}, \delta)$ and the search range ϕM, it can easily be shown that Alg. 1 requires $\sum_{i=1}^{n-1} \nu^{i-1}(n-i)$ document-pivot comparisons to diversify n documents. In the worst case, when $\nu = 1$ (i.e., all documents are outside the range of all pivots and become themselves pivots), this algorithm exhibits the same quadratic complexity as the greedy novelty seeking approach. However, in practical deployments, $\nu < 1$, which results in a drastic reduction in the number of required document-pivot comparisons, as we will show in Section 5.

SSSD1 promotes novelty in an iterative fashion, by prospecting new pivots from the ranking $\mathcal{D}$. Although respecting the order that the documents were originally retrieved for the query q, this variant does not perform any explicit re-scoring of these documents, and in fact treats the non-pivot documents indistinctly. To investigate whether a more fine-grained re-ranking could be beneficial, we propose a second variant of our approach. In particular, the SSSD2 variant extends SSSD1, by performing a second step over the retrieved documents, in order to assign each document d_i a score with respect to the query, in light of the abstract diversification model defined in Equation (1). This variant is described in Alg. 2. SSSD2 is essentially equivalent to SSSD1, except for the introduced scoring step (lines 7-9 in Alg. 2), and the additional parameter β. In particular, this introduced step scores a document d_i as a linear combination of its estimated relevance to the query and its estimated novelty, as given by the inverse of the distance between d_i and its most similar pivot $p_j \in \mathcal{P}$. A balance between relevance and novelty is achieved through an appropriate setting of β. In a naive implementation, this step takes additional $O(kn)$ document-document comparisons, which is already substantially more efficient than existing novelty-based diversification approaches, if $k \ll n$. However, even when this is not the case—particularly for high-dimensional spaces—the second step can reuse all the comparisons performed for the pivot selection in the first step. Therefore, as we will show in Section 5, in contrast to SSSD1, SSSD2 can deploy a traditional diversification scoring scheme, at the cost of typically only a few additional comparisons.

SSSD2$[q, \mathcal{D} = \{d_1, \ldots, d_n\}, \delta, M, \phi, \beta]$

1 $\mathcal{P} \Leftarrow \{d_1\}$
2 **for all** $d_i \in \mathcal{D} \setminus \{d_1\}$ **do**
3   **if** $\delta(d_i, p_j) \geq \phi M \ \forall p_j \in \mathcal{P}$ **then**
4     $\mathcal{P} \Leftarrow \mathcal{P} \cup \{d_i\}$
5   **end if**
6 **end for**
7 **for all** $d_i \in \mathcal{D}$ **do**
8   $\text{score}(q, d_i) = (1-\beta)\,\text{rel}(q, d_i) + \beta\left[1 - \max_{p_j \in \mathcal{P}} \delta(d_i, p_j)\right]$
9 **end for**

Alg. 2. Two-Step Sparse Spatial Selection Diversification (SSSD2)

4 Experimental Setup

Our investigation aims to answer two major research questions:

1. How do SSSD and existing approaches compare in terms of effectiveness?
2. How do SSSD and existing approaches compare in terms of efficiency?

To evaluate our approach in different metric spaces, we experiment with three test collections for diversity evaluation, comprising both web and newswire documents. The first two are from the diversity task of the TREC 2009 and 2010 Web tracks [8, 9]—henceforth WT09 and WT10, respectively. WT09 includes 50 topics, while WT10 comprises 48 topics. Our third collection includes 20 topics from the Interactive track of TREC-6, TREC-7, and TREC-8 [12]—henceforth IT678. For WT09 and WT10, we index the TREC ClueWeb09 (cat. B) corpus, with 50 million web documents. For IT678, we index the Financial Times portion of TREC Disks 4&5, with 210,000 newswire documents. Both corpora are indexed using Terrier [19], with Porter's stemmer and standard stopword removal.

To retrieve an initial pool of documents to be diversified, we apply either BM25 or the Divergence from Randomness DPH model, as implemented in Terrier. On top of these adhoc retrieval baselines, we deploy two well-known novelty-based diversification approaches as diversification baselines: Maximal Marginal Relevance (MMR [4]) and Mean-Variance Analysis (MVA [22]). As these approaches compute novelty based on cosine or correlation estimations, respectively, we deploy both variants of our approach using both cosine and Pearson's correlation as instantiations of the distance function δ. To cope with the quadratic complexity of MMR and MVA while keeping a uniform setting across all approaches, both of these baselines as well as our SSSD variants are applied to diversify the top 100 documents retrieved by BM25 or DPH.

Effectiveness is assessed using the primary metrics in the diversity task of the TREC 2010 Web track, namely, ERR-IA [6] and α-nDCG [10]. To train the parameters of our approach (ϕ for SSSD1 and both ϕ and β for SSSD2), as well as the parameters for MMR (λ [4]) and MVA (σ and b [22]), we perform a simulated annealing [13] through a 5-fold cross validation. In particular, we train the parameters of all approaches to maximise α-nDCG@100 on the training folds, and report the results as an average across the corresponding separate test folds. As for efficiency, we report the number of document-document comparisons performed, as well as the time spent in performing such comparisons.

5 Experimental Results

In this section, we investigate whether novelty-based diversification approaches can be made efficient without compromising their effectiveness. Before investigating the efficiency of SSSD, we evaluate its effectiveness compared to MMR [4] and MVA [22] as baselines. Table 1 shows the diversification performance of both SSSD variants as well as these two baselines across the WT09, WT10, and IT678 settings. As the distance function δ, we consider both cosine (denoted c)

Table 1. Diversification performance across the WT09, WT10, and IT678 topics

	WT09		WT10		IT678	
	ERR-IA @20	α-nDCG @20	ERR-IA @20	α-nDCG @20	ERR-IA @20	α-nDCG @20
BM25	0.1304	0.2290	0.1628	0.2349	0.1541	0.4703
+MMR	0.1341	0.2366	0.1652	0.2379	0.1573	**0.4806**
+MVA	0.1336	0.2369	0.1654	0.2343	0.1547	0.4708
+SSSD1(c)	**0.1429**$^{\vartriangle}$	**0.2526**$^{\vartriangle}$	0.1688$^{\vartriangle}$	**0.2447**	**0.1600**	0.4764
+SSSD1(ρ)	0.1242	0.2234	0.1585	0.2324	0.1500	0.4577
+SSSD2(c)	0.1237	0.2178	0.1628	0.2356	0.1483	0.4481
+SSSD2(ρ)	0.1279	0.2248	**0.1695**	0.2402	0.1532	0.4662
DPH	0.1430	0.2426	0.1952	0.2977	0.1658	0.4833
+MMR	0.1378	0.2363	0.1963	0.2889	0.1652	**0.4842**
+MVA	0.1314	0.2203	0.1908	0.2841	0.1636	0.4674
+SSSD1(c)	0.1474$^{\blacktriangle}$	0.2608$^{\blacktriangle}$	0.1952	**0.2981**	0.1620	0.4689
+SSSD1(ρ)	0.1333	0.2266	**0.1973**	0.2977	**0.1678**	0.4831
+SSSD2(c)	0.1344	0.2367	0.1944	0.2945	0.1639	0.4807
+SSSD2(ρ)	**0.1637**	**0.2646**$^{\vartriangle}$	0.1847	0.2796	0.1518	0.4692

and Pearson's correlation (denoted ρ). All approaches are applied on top of both BM25 and DPH. Significance between both SSSD1 and SSSD2 and the best between MMR and MVA is verified with the Wilcoxon matched-pairs test. In particular, the symbols $\vartriangle$ and $\triangledown$ denote a significant increase or decrease with $p < 0.05$, while $\blacktriangle$ and $\blacktriangledown$ denote significant increases or decreases with $p < 0.01$.

From Table 1, we note that both SSSD1 and SSSD2 can improve over MMR and MVA across several settings. Such improvements are significant for SSSD1(c) using BM25 (for WT09 and WT10) and DPH (for WT09), and for SSSD2(ρ) using DPH (for WT09). In all other cases, there is no significant difference between these approaches. This answers our first question, by showing that SSSD performs *at least as effectively* as existing novelty-based approaches. As for distance functions, when the initial ranking is given by BM25, cosine gives superior results for SSSD1, while Pearson's correlation is the most effective function for SSSD2. When DPH provides the initial ranking, there is no consistently best choice of function. As for the two SSSD variants themselves, SSSD1 performs generally better than SSSD2 (except for IT678 using DPH) when cosine is fixed as the distance function. For Pearson's correlation, SSSD2 is generally the best of the two variants for BM25, while SSSD1 is generally best for DPH. Overall, these results show that the choice of an SSSD variant depends on the considered metric space, as determined by the target test collection and the chosen distance function.

To answer our second question, we investigate how the pivot selection impacts the efficiency of our approach. In particular, the number of selected pivots is a function of both the dimensionality of the search space and the search radius ϕ. Hence, we analyse the efficiency of SSSD1 and SSSD2 over a range of ϕ values, as well as over the search spaces of the three considered test collections. For the WT09, WT10, and IT678 collections, Fig. 3 shows how the number of selected pivots (Figs. 3(a)-(c)), the number of document-document comparisons

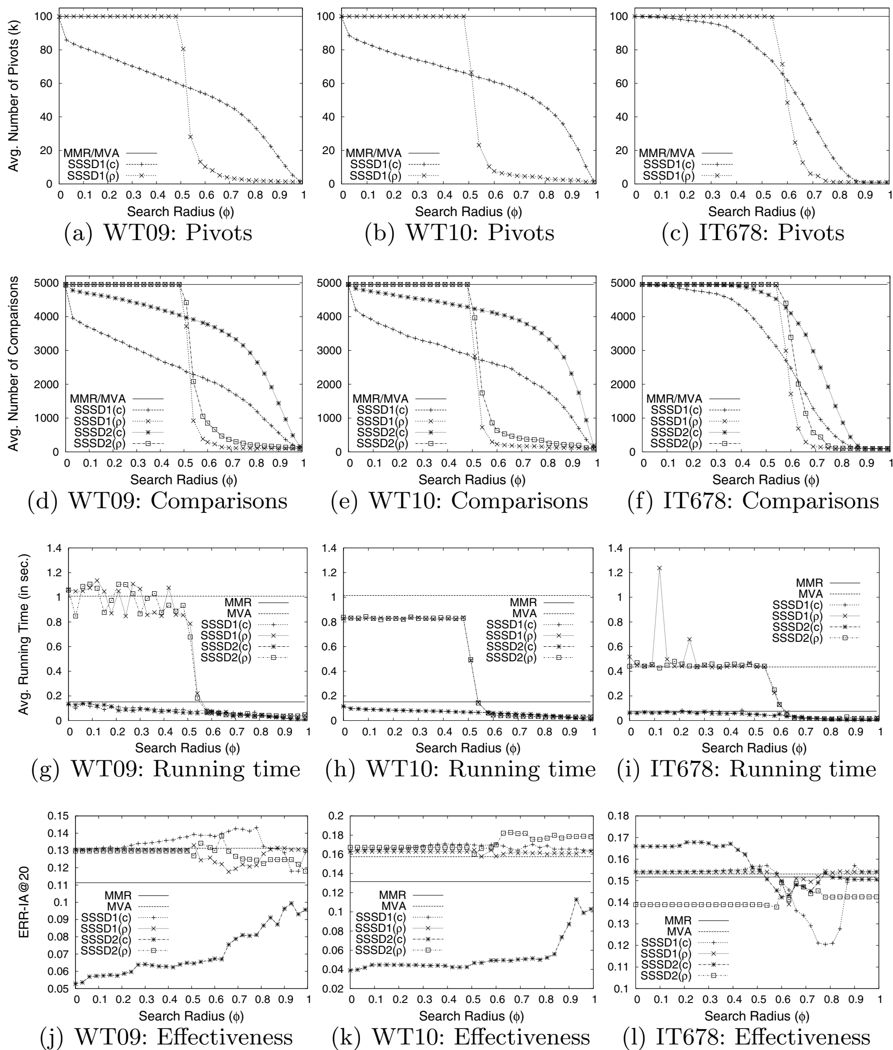
(Figs. 3(d)-(f)), and the running time[1] (Figs. 3(g)-(i)) of our approach are affected by the parameter ϕ. Additionally, to enable the analysis of efficiency in context, Figs. 3(j)-(l) show how ϕ impacts the effectiveness of our approach.

From Figs. 3(a)-(c), we first observe, as expected, that the number of pivots selected by SSSD1[2] decreases as ϕ increases, since the area covered by each pivot increases. However, while the number of selected pivots decreases smoothly for SSSD1(c), a more abrupt drop is observed for SSSD1(ρ), with an inflection around $\phi = 0.5$ for WT09 and WT10, and $\phi = 0.6$ for IT678. This suggests that correlation is more sensitive than cosine as a distance function. In particular, in such sparse spaces as those considered here, documents which share only a few but highly informative terms can exhibit negligible correlations, while still having a noticeable cosine. Next, we assess how ϕ (and consequently, the number pivots) impacts the number of comparisons and the running time of our approach.

Contrasting Figs. 3(a)-(c) and (d)-(f), we observe a similar shape between the number of selected pivots and that of performed comparisons. Indeed, there is an almost perfect linear correlation between the number of comparisons and of selected pivots (WT09: 0.993 for SSSD1(c), 0.998 for SSSD1(ρ); WT10: 0.993 for SSSD1(c), 0.999 for SSSD1(ρ); IT678: 0.992 for SSSD1(c), 0.997 for SSSD1(ρ)). This provides empirical evidence that SSSD1 has an average-case complexity of $O(k)$, where $k = O(n)$ is the number of selected pivots. Moreover, when SSSD2 is considered, only a constant number of additional comparisons is performed, hence leaving the asymptotic cost unchanged. In practice, this shows that our approach is *an order of magnitude faster* compared to the quadratic number of comparisons performed by both MMR and MVA (precisely, $n(n-1)/2$ comparisons), hence answering our second research question. This observation is further confirmed by Figs. 3(g)-(i), which show the running time of our approach, compared to both MMR and MVA, for a range of ϕ values, and averaged across the WT09, WT10, and IT678 topics, respectively. Although dominated by the number of comparisons, these figures exemplify another facet of the time complexity of all novelty-based approaches, namely, the unitary cost of a comparison. Indeed, computing the cosine between two documents is cheaper than computing their correlation, even though both are optimised to exploit the sparsity of the considered spaces. Nonetheless, the variants of SSSD using these distance functions are faster than MMR (which uses cosine) and MVA (which uses correlation), respectively, across the entire range of ϕ values, and for the three considered collections. To further test these approaches over a representative query stream, we select the first 1,000 queries from the MSN 2006 query log [11], after removing empty queries and queries with no results in the ClueWeb09 corpus. Figs. 4(a) and (b) show the results of this investigation in terms of number of comparisons and running time, respectively. These results closely match those shown in Figs. 3(d)-(f) and (g)-(i), hence further attesting the efficiency of our approach.

[1] Running times are based on a Linux Quad-Core Intel Xeon 2.4GHz 8GB, and denote the time spent to compare documents, as the cost to retrieve the initial documents and represent these documents in a vector space is the same for all approaches.

[2] SSSD2 uses the same pivot selection as SSSD1, and is hence omitted from the figures.



(a) WT09: Pivots (b) WT10: Pivots (c) IT678: Pivots

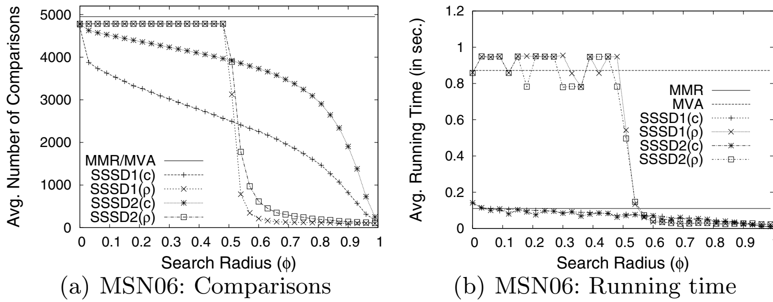
(d) WT09: Comparisons (e) WT10: Comparisons (f) IT678: Comparisons

(g) WT09: Running time (h) WT10: Running time (i) IT678: Running time

(j) WT09: Effectiveness (k) WT10: Effectiveness (l) IT678: Effectiveness

Fig. 3. Number of pivots, number of document-document comparisons, running time, and diversification performance for the WT09, WT10, and IT678 test collections (left, middle, and right columns, respectively), across a range of ϕ values. All figures are averages across the topics of the corresponding collection (50, 48, and 20, respectively).

Lastly, Figs. 3(j)-(l) bridge our two research questions, by showing the impact of increasing the search radius ϕ on the effectiveness of both variants of SSSD, in terms of ERR-IA@20. In general, we observe two distinct behaviours. Firstly, a steady improvement is observed for SSSD1(c) for WT09 (up to $\phi \approx 0.8$) and SSSD2(c) for both WT09 and WT10. With a higher search radius ϕ, these variants perform a more aggressive diversification, by creating fewer pivots and considering more documents as redundant. Secondly, a dual impact is observed



(a) MSN06: Comparisons (b) MSN06: Running time

Fig. 4. Number of document-document comparisons and running time across a range of ϕ values. All figures are averages over 1000 queries from the MSN 2006 query log.

for the SSSD2(ρ) variant for $\phi > 0.5$, which coincides with the inflection point in Figs. 3(a)-(c). In particular, while the effectiveness of SSSD2(ρ) decreases after this point for WT09, it increases for WT10. Likewise, a region of instability is observed for other variants after the inflection point (i.e., $0.5 \leq \phi \leq 0.9$. This is the case for SSSD1(ρ) for WT09 and IT678, and for SSSD1(c), SSSD2(c), and SSSD2(ρ) for IT678. Overall, these results show that even documents of a similar nature (e.g., web pages) can result in rather different spaces. Hence, carefully choosing a search radius for the test collection at hand is key for attaining a suitable trade-off between an effective and efficient diversification.

6 Conclusions

We have introduced a new approach for novelty-based search result diversification, by exploiting the properties of metric spaces. Our Sparse Spatial Selection Diversification (SSSD) approach selects a set of pivots from the space of documents retrieved for a query, and leverages the triangle inequality property of metric spaces to regard documents covered by a pivot as redundant. As an extended variant, we further score the retrieved documents with respect to their distance to the selected pivots, in order to perform a more fine-grained re-ranking.

In a thorough investigation across three standard TREC test collections for diversity evaluation, we have shown that both variants of our approach (SSSD1 and SSSD2) perform at least as effectively as well-known novelty-based diversification approaches in the literature, while improving their efficiency by an order of magnitude. Moreover, by evaluating our approach across metric spaces induced by different document collections and distance functions, we have shown that a careful selection of pivots is paramount for appropriately trading-off effectiveness and efficient in novelty-based search result diversification.

Acknowledgements. The work has been performed under the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission–Capacities Area–Research Infrastructures.

References

1. Agrawal, R., Gollapudi, S., Halverson, A., Ieong, S.: Diversifying search results. In: WSDM, pp. 5–14 (2009)
2. Barrios, J.M., Diaz-Espinoza, D., Bustos, B.: Text-based and content-based image retrieval on Flickr: DEMO. In: SISAP, pp. 156–157 (2009)
3. Brisaboa, N.R., Farina, A., Pedreira, O., Reyes, N.: Similarity search using sparse pivots for efficient multimedia information retrieval. In: ISM, pp. 881–888 (2006)
4. Carbonell, J., Goldstein, J.: The use of MMR, diversity-based reranking for reordering documents and producing summaries. In: SIGIR, pp. 335–336 (1998)
5. Carterette, B., Chandar, P.: Probabilistic models of ranking novel documents for faceted topic retrieval. In: CIKM, pp. 1287–1296 (2009)
6. Chapelle, O., Metlzer, D., Zhang, Y., Grinspan, P.: Expected reciprocal rank for graded relevance. In: CIKM, pp. 621–630 (2009)
7. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. 33(3), 273–321 (2001)
8. Clarke, C.L.A., Craswell, N., Soboroff, I.: Overview of the TREC 2009 Web track. In: TREC (2009)
9. Clarke, C.L.A., Craswell, N., Soboroff, I., Cormack, G.V.: Preliminary overview of the TREC 2010 Web track. In: TREC (2010)
10. Clarke, C.L.A., Kolla, M., Cormack, G.V., Vechtomova, O., Ashkan, A., Büttcher, S., MacKinnon, I.: Novelty and diversity in information retrieval evaluation. In: SIGIR, pp. 659–666 (2008)
11. Craswell, N., Jones, R., Dupret, G., Viegas, E. (eds.): Proceedings of the 2009 Workshop on Web Search Click Data (2009)
12. Hersh, W., Over, P.: TREC-8 Interactive track report. In: TREC (2000)
13. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
14. van Leuken, R.H., Garcia, L., Olivares, X., van Zwol, R.: Visual diversification of image search results. In: WWW, pp. 341–350 (2009)
15. Mamede, M., Barbosa, F.: Range queries in natural language dictionaries with recursive lists of clusters. In: ISCIS (2007)
16. Micó, L., Oncina, J., Carrasco, R.C.: A fast branch & bound nearest neighbour classifier in metric spaces. Pattern Recogn. Lett. 17(7), 731–739 (1996)
17. Navarro, G., Reyes, N.: Fully dynamic spatial approximation trees. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 254–270. Springer, Heidelberg (2002)
18. Navarro, G., Reyes, N.: Dynamic spatial approximation trees for massive data. In: SISAP, pp. 81–88 (2009)
19. Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., Lioma, C.: Terrier: a high performance and scalable information retrieval platform. In: OSIR (2006)
20. Santos, R.L.T., Macdonald, C., Ounis, I.: Exploiting query reformulations for Web search result diversification. In: WWW, pp. 881–890 (2010)
21. Santos, R.L.T., Macdonald, C., Ounis, I.: Selectively diversifying Web search results. In: CIKM (2010)
22. Wang, J., Zhu, J.: Portfolio theory of information retrieval. In: SIGIR, pp. 115–122 (2009)
23. Zhai, C., Cohen, W.W., Lafferty, J.: Beyond independent relevance: Methods and evaluation metrics for subtopic retrieval. In: SIGIR, pp. 10–17 (2003)

Candidate Document Retrieval for Web-Scale Text Reuse Detection*

Matthias Hagen and Benno Stein

Faculty of Media
Bauhaus-Universität Weimar, Germany
firstname.lastname@uni-weimar.de

Abstract. Given a document d, the task of text reuse detection is to find those passages in d which in identical or paraphrased form also appear in other documents. To solve this problem at web-scale, keywords representing d's topics have to be combined to web queries. The retrieved web documents can then be delivered to a text reuse detection system for an in-depth analysis. We focus on the query formulation problem as the crucial first step in the detection process and present a new query formulation strategy that achieves convincing results: compared to a maximal termset query formulation strategy [10, 14], which is the most sensible non-heuristic baseline, we save on average 70% of the queries in realistic experiments. With respect to the candidate documents' quality, our heuristic retrieves documents that are, on average, more similar to the given document than the results of previously published query formulation strategies [4, 8].

1 Introduction

The problem considered in this paper appears as an important sub-task of automatic text reuse detection. A text reuse detection system aims at finding passages within a given document which, in a similar form, are also contained in another document. The goal is not only to identify simple one-to-one copies but also cases of paraphrased text reuse. Note that plagiarism detection represents a special case whereas text reuse detection addresses a broader spectrum that also covers problems like information spread analysis (e.g., where are news stories reused?).

Usually, automatic detection systems find potential reuse passages via face-to-face comparisons of the given document against a set of "promising" documents. While for small document collections it is feasible to perform a complete comparison against every document, this is obviously not possible when the collection is large. The idea then is to compare only against documents that cover a topic similar to the given document, with the rationale that such documents are more likely to be the source (or "sink") of text reuse. A straightforward approach to find documents on similar topics is to extract keywords or longer components like head noun phrases from the given document and to retrieve other documents also containing these keywords.

Our contribution to this problem is a strategy of how to query a web search engine using the extracted keywords. However, we do not deal with the complete task of text

* Extended version of an ECDL 2010 poster paper [10].

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 356–367, 2011.
© Springer-Verlag Berlin Heidelberg 2011

reuse detection. We tackle the essential pre-computation step that finds promising candidate documents for the in-depth analysis (for which we in turn assume that state-of-the-art text reuse or plagiarism detection techniques are used [4, 9, 13, 16]). We focus on *web querying* to identify potential candidates since the web became the typical place of text reuse. However, a detection system usually is not given arbitrary access to a web search engine's index; moreover it has incomplete or even no knowledge about the engine's underlying retrieval model, implementation details, and the like. A web search engine appears as a black box and queries are not for free but entail costs—at the very least some non-negligible amount of time is consumed, and monetary charges come into play for larger contingents of queries.[1]

1.1 User over Ranking

The number of documents a detection system can consider for an in-depth text reuse analysis is constrained by a processing capacity k, which in turn depends on the desired answer time, the processing time per document, and machine usage cost. If the entire set of extracted keywords (typically about 10) from a given document is submitted as a single web query, this query will probably be *overspecific* (i.e., hardly returning more than a handful of documents) and thus wasting processing capacity. On the other hand, queries containing only few of the extracted keywords are likely to be *underspecific* (i.e., having very long result lists) and discard valuable information: from overlong result lists only a fraction, typically the top-ranked results, can be processed by the detection system. Notice that such queries put the burden of selecting the most promising text reuse candidates on the search engine's ranking algorithm; potential text reuse cases that are not among the top results will be missed. Hence, a set of promising web queries should avoid underspecificity and, combined, cover all extracted keywords in order to ensure a high similarity to the given document's topic. Altogether, we argue that the probability to find potential text reuse cases by exploring k results becomes maximum if the combined result list length of the set of promising queries is in the order of magnitude of the processing capacity k of the detection system. This is an implication of the recent User-over-Ranking hypothesis, which states that queries have a higher probability of satisfying a user's information need if they return about as many results as the user can consider [17]. Figure 1 illustrates the outlined connections.

Under the User-over-Ranking hypothesis the treated query formulation sub-problem of text reuse detection is defined as follows:

CAPACITY CONSTRAINED QUERY FORMULATION

Given: (1) Set W of keywords.
(2) Query interface for web search engine $\mathcal{S}$.
(3) Upper bound k on the number of desired documents.

Task: Find a family $\mathcal{Q} \subseteq 2^W$ of queries together returning at most k documents, while containing all keywords from W.

[1] E.g., \$0.40–\$0.80 per 1000 Yahoo! BOSS queries, http://www.ysearchblog.com/2011/02/08/latest-on-boss/ (accessed April 16th, 2011).

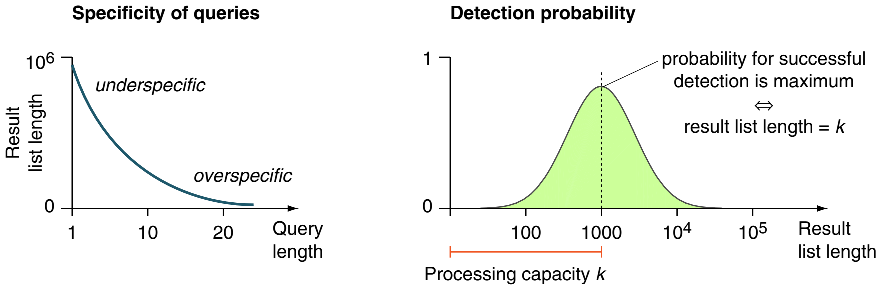


Fig. 1. Left: a query with few terms or many results is likely to be underspecific, queries with many terms or few results tend to be overspecific. Right: under the User-over-Ranking hypothesis a combined result list length of the detection system's capacity k maximizes the probability of finding potential text reuse cases [17].

1.2 Related Work

One of the earliest approaches on formulating queries respecting a bound on the number of returned results is the maximal termset query formulation of Pôssas et al. [14]. We use an adapted version of maximal termset query formulation as our baseline. With respect to runtime, our new system clearly improves upon the maximal termset baseline while retrieving basically the same candidate documents.

A recent paper by Bendersky and Croft also deals with the scenario of text reuse detection on the web [4]. However, Bendersky and Croft's problem setting is different from ours: they focus on single sentences and not on complete documents as input, and hence their querying strategy is quite elementary. Our setting is also more general in another respect: the passages from the document for which a reuse analysis is requested have not to be known a priori. Nevertheless, in our experiments we compare our query formulation strategy to Bendersky and Croft's querying approach.

In our setting it would be desirable to use the given document as a query itself ("query by document"). Yang et al. [19] focus on such a scenario in the context of analyzing blog posts. But, similar to us, they also try to derive a keyword query that reflects the document's (blog post's) content. Their approach extracts keyphrases from the document, but formulates only a single query from them. Since this would waste capacity in our setting and since their approach of manually selecting the number of "good" keywords for each document is not applicable in a fully automatic system, we do to not include Yang et al.'s approach in the experimental comparison.

A more applicable setting which is also related to ours is Dasdan et al.'s work on finding similar documents by using only a search engine interface [8]. Although Dasdan et al. focus on a search engine coverage problem (resolve whether a search engine's index contains a given document or some variant of it), their approach of finding similar documents using keyword interfaces is basically equivalent to our setting. Dasdan et al. propose two querying strategies and experimentally show that their approaches indeed find similar documents. In our experiments we also compare their strategies to our heuristic.

A very promising idea for our setting would be to predict a given query's performance before submitting it to a search engine [6, 7, 11, 12]. However, the evaluation of the best performing predictors needs access to knowledge that is not available at user site in a standard web search scenario. For example, the simplified query clarity predictor [12] needs the total keyword frequencies for the whole corpus—web search engines just return an estimation of the number of documents in the corpus that contain the keyword. The query scope predictor [12] needs the number of documents in the index—most web search providers stopped publishing it. Furthermore, there are studies suggesting to take care when interpreting the published evaluations of established predictors [15] such that we decided not to use quality prediction in our approaches.

2 Notation and Basic Definitions

Starting point of the query formulation process is a set $W = \{w_1, \ldots, w_n\}$ of keywords; allowing the w_i to be longer components like head noun phrases makes no difference. Subsets $Q \subseteq W$ can be submitted as web queries, with the notion that phrases are included in quotation marks. An engine's reply to a query consists of the beginning of an exhaustive, ranked list L_Q of snippets and URLs of documents relevant for Q, and an estimation ℓ_Q for the result list length $|L_Q|$. The task of CAPACITY CONSTRAINED QUERY FORMULATION is to find a family $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ of queries $Q_i \subseteq W$ having the following properties:

1. $\mathcal{Q}$ is *simple* in the sense that $Q_i \not\subseteq Q_j$ for any $Q_i, Q_j \in \mathcal{Q}$, with $i \neq j$. This avoids redundancy in the queries and the results.
2. Combined, $\mathcal{Q}$'s queries don't return more than k results. This respects the detection system's processing capacity.
3. Combined, $\mathcal{Q}$'s queries cover W's keywords if possible: ideally $\bigcup_{Q \in \mathcal{Q}} Q = W$. This ensures that the resulting documents cover all the topics contained in W.

With respect to the capacity k, we introduce a per-query upper bound $\ell_{\max}$ with the notion that a query Q is promising iff $\ell_Q \leq \ell_{\max}$ (i.e., it returns at most $\ell_{\max}$ results). How exactly this upper bound is derived will be explained in Section 3. A per-query lower bound $\ell_{\min}$ serves convenience purposes of ruling out queries with very few results (e.g., $\ell_{\min} = 1$ means that queries returning an empty result list are not tolerated). Applying both bounds, a query $Q \in \mathcal{Q}$ has to satisfy $\ell_{\min} \leq \ell_Q \leq \ell_{\max}$. Adopting notation from Bar-Yossef and Gurevich [2], we say that for $\ell_Q < \ell_{\min}$ the query Q is *underflowing*, whereas for $\ell_Q > \ell_{\max}$ it is *overflowing*. Queries that are neither under- nor overflowing are *valid*. A valid query Q is *minimal* iff dropping some keyword from Q results in an overflowing query.

As a solution to CAPACITY CONSTRAINED QUERY FORMULATION we suggest the family $\mathcal{Q}_{\text{lo}}$ of all minimal valid queries for an appropriate value of $\ell_{\max}$. Obviously, $\mathcal{Q}_{\text{lo}}$ is simple and depends on the value of $\ell_{\max}$. Our approach will adaptively determine values for $\ell_{\max}$, derive the corresponding $\mathcal{Q}_{\text{lo}}$ and, according to the combined number of $\mathcal{Q}_{\text{lo}}$'s results (more or less than k), output this $\mathcal{Q}_{\text{lo}}$ or re-iterate by setting $\ell_{\max}$ to a more appropriate value. Hence, an appropriate $\mathcal{Q}_{\text{lo}}$ respects the first two constraints of being simple and not returning more than k results.

That $\mathcal{Q}_{\text{lo}}$ also is a good choice with respect to the third constraint of covering as many keywords of W as possible can be seen as follows. We say that a query Q *covers* all its keywords. Analogously, a family $\mathcal{Q}$ of queries covers all keywords in $\bigcup_{Q \in \mathcal{Q}} Q$. Note that there are situations where it is not possible to cover W with a family of valid queries (e.g., when a single keyword itself is underflowing). A keyword $w \in W$ is *coverable* iff there is a valid query $Q \subseteq W$ with $w \in Q$.

Lemma 1. *Let $\mathcal{Q}$ be a family of valid queries covering the coverable keywords from a keyword set W for given $\ell_{\min}$ and $\ell_{\max}$. For every $Q \in \mathcal{Q}$ we have: there is a sub-family $\mathcal{Q}'_{\text{lo}} \subseteq \mathcal{Q}_{\text{lo}}$ such that $Q = \bigcup_{Q' \in \mathcal{Q}'_{\text{lo}}} Q'$.*

Proof. Assume we have $Q \in \mathcal{Q}$ but $Q \neq \bigcup_{Q' \in \mathcal{Q}'_{\text{lo}}} Q'$ for any $\mathcal{Q}'_{\text{lo}} \subseteq \mathcal{Q}_{\text{lo}}$. Since $\mathcal{Q}$ is a family of valid queries, Q must be valid. Assume that Q contains only coverable keywords from W. Now consider the family $\mathcal{Q}'$ of the $2^{|Q|} - 1$ subqueries of Q excluding the empty query. Let $\mathcal{Q}'' \subseteq \mathcal{Q}'$ be the sub-family of valid queries. Note that $\mathcal{Q}''$ is not empty since it contains Q. From $\mathcal{Q}''$ we remove all queries that are proper supersets of queries in $\mathcal{Q}''$ and obtain the family $\tilde{\mathcal{Q}}$ of minimal valid subqueries of Q. Note that $\tilde{\mathcal{Q}}$ is not empty since $\mathcal{Q}''$ is not empty and that $Q = \bigcup_{\tilde{Q} \in \tilde{\mathcal{Q}}} \tilde{Q}$. Since $\tilde{\mathcal{Q}}$ contains minimal valid queries only, we have $\tilde{\mathcal{Q}} \subseteq \mathcal{Q}_{\text{lo}}$; a contradiction to our assumption. Hence, Q contains a non-coverable keyword $w \in W$. Since w is not coverable by a valid query, Q cannot be valid. A contradiction again. □

Corollary 1. *For a given set W of keywords and given $\ell_{\min}$ and $\ell_{\max}$, the respective family $\mathcal{Q}_{\text{lo}}$ covers the coverable keywords. Furthermore, $\mathcal{Q}_{\text{lo}}$ contains the with respect to set inclusion minimal queries covering the coverable keywords.*

In the process of finding an appropriate $\mathcal{Q}_{\text{lo}}$ on input W, we count the overall number *cost* of queries that are submitted to the search engine. The underlying assumption is that a system is faster when it submits less queries.

3 Baseline: Maximal Termset Query Formulation

As a baseline query formulation process, we adapt the maximal termset approach by Pôssas et al. [14]. We refrain from using GenMax as a subroutine to enlarge promising keyword subsets (as proposed in [14]) but choose the classic Apriori algorithm instead, which also comes from the field of frequent itemset mining [1]. Apriori is considered as one of the top 10 data mining algorithms [18]; it traverses the search space of all possible queries in a level-wise manner. A basic pseudo-code listing of Apriori for fixed lower and upper bounds $\ell_{\min}$ and $\ell_{\max}$ is given as Algorithm 1. How the algorithm handles the adaptive adjustment to detect a reasonable $\ell_{\max}$ is explained below, after introducing the basic Apriori framework.

Apriori first checks which of the initial keywords itself are overflowing or valid. The overflowing keywords form the first level of candidate queries (variable C_1 in line 2) that can be further expanded. A second pre-check ensures that these remaining keywords from the first candidate level altogether are not overflowing (line 3). Otherwise no valid queries can be formulated from them. After the pre-checks, Apriori combines candidate queries (lines 5 to 11) in a level-wise manner. It is straightforward to show that Algorithm 1 finally outputs the desired $\mathcal{Q}_{\text{lo}}$ for given $\ell_{\min}$ and $\ell_{\max}$; just notice

Algorithm 1. The Apriori algorithm for query formulation

Input: a set W of keywords, $\ell_{\min}$, and $\ell_{\max}$
Output: the family $\mathcal{Q}_{\text{lo}}$

1: $\mathcal{Q} \leftarrow \{\{w\} : w \in W$ and $\{w\}$ is valid$\}$
2: $C_1 \leftarrow \{\{w\} : w \in W$ and $\{w\}$ overflows$\}$
3: **if** $\bigcup_{\{w\} \in C_1} \{w\}$ overflows **then stop and output** $\mathcal{Q}$
4: $i \leftarrow 1$
5: **while** $C_i \neq \emptyset$ **do**
6:     **for all** $Q, Q' \in C_i, |Q \cap Q'| = i - 1$ **do**
7:         $Q_{\text{cand}} \leftarrow Q \cup Q'$
8:         **if** $Q_{\text{cand}} \setminus \{w\} \in C_i$ for all $w \in Q_{\text{cand}}$ **then**
9:             **if** Q_{cand} overflows **then** $C_{i+1} \leftarrow C_{i+1} \cup \{Q_{\text{cand}}\}$
10:            **if** Q_{cand} is valid **then** $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{Q_{\text{cand}}\}$
11:     $i \leftarrow i + 1$
12: **output** $\mathcal{Q}$

that whenever a query becomes valid it is directly added to the output (thus being minimal) and that, due to the exhaustive search character, no minimal valid query will be missed. A query's validity (lines 1, 2, 3, 9, and 10) is checked via submission to the web search engine. We use the engine's estimations ℓ_Q, although they often overestimate the correct result list lengths. However, they usually respect monotony (queries containing additional keywords have smaller ℓ-value) and the shorter the result list, the more accurate the estimations.

We adopt Algorithm 1 as our baseline—and even tighten this baseline by applying the following Lemma as a means to reduce the number of web queries Apriori submits.

Lemma 2. *Let $Q_1, Q_2, Q_3 \subseteq W$ be queries. Assuming the ℓ-estimations to be reasonable we have* $\ell_{Q_1 \cup Q_2 \cup Q_3} \geq \ell_{Q_1 \cup Q_2} + \ell_{Q_1 \cup Q_3} - \ell_{Q_1}$.

Proof.
$$\begin{aligned}\ell_{Q_1 \cup Q_2 \cup Q_3} &= |L_{Q_1 \cup Q_2} \cap L_{Q_1 \cup Q_3}| = |L_{Q_1 \cup Q_2} \setminus (L_{Q_1 \cup Q_2} \setminus L_{Q_1 \cup Q_3})| \\ &\geq |L_{Q_1 \cup Q_2} \setminus (L_{Q_1} \setminus L_{Q_1 \cup Q_3})| \geq |L_{Q_1 \cup Q_2}| - |L_{Q_1} \setminus L_{Q_1 \cup Q_3}| \\ &= |L_{Q_1 \cup Q_2}| - (|L_{Q_1}| - |L_{Q_1 \cup Q_3}|) = \ell_{Q_1 \cup Q_2} + \ell_{Q_1 \cup Q_3} - \ell_{Q_1} \quad \square\end{aligned}$$

Let Q and Q' be the queries that are merged to get Q_{cand} (line 7 of Apriori). Lemma 2 then is applied as follows: $Q_1 \cup Q_2 = Q$, $Q_1 \cup Q_3 = Q'$, and $Q_1 = Q \cap Q'$. The rationale is: if the Lemma 2 estimation $\ell_{Q_{\text{cand}}} \geq \ell_Q + \ell_{Q'} - \ell_{Q \cap Q'}$ is larger than $\ell_{\max}$, we do not have to submit Q_{cand} to an engine but can add it to the current candidate set C_{i+1} immediately.

A remaining problem is to adaptively set $\ell_{\max}$. The algorithm starts with $\ell_{\max} = k$, computes $\mathcal{Q}_{\text{lo}}$ using Apriori and checks the number of results returned by $\mathcal{Q}_{\text{lo}}$. Usually this will be too many since $\mathcal{Q}_{\text{lo}}$ contains more than one query. The algorithm then sets $\ell_{\max} = \lfloor \ell_{\max}/2 \rfloor$ and computes the corresponding $\mathcal{Q}_{\text{lo}}$. A naïve approach would restart the entire Apriori computation from scratch and repeat all steps from the previous run, resulting in a bad overall practical performance. However, a nice feature of Apriori is that it can be easily modified to continue computation on the reached state of the previous $\ell_{\max}$ setting such that re-computations and re-submissions of web queries are

avoided. If at one intermediate step the algorithm notices that the current $\mathcal{Q}_{\text{lo}}$ returns not more but approximately k results (we set the bound to at least 90%), it stops and outputs the current $\mathcal{Q}_{\text{lo}}$. If eventually too few results are returned, the algorithm sets $\ell_{\max} = \lfloor 3/2 \cdot \ell_{\max} \rfloor$. Altogether, this implements a kind of binary search for a good value of $\ell_{\max}$. Note that whenever the algorithm enlarges $\ell_{\max}$ for the first time, all of the currently needed queries have already been examined during the previous step such that no further queries have to be submitted.

4 Heuristic Search Strategy

Preliminary tests revealed that the savings due to Lemma 2 in the Apriori algorithm are often negligible: the sum $\ell_Q + \ell_{Q'}$ usually is too small compared to $\ell_{Q \cap Q'}$ and hence the query $Q_{\text{cand}} = Q \cup Q'$ has to be submitted. Nevertheless, the performance of the baseline Apriori framework can be significantly improved. We propose a heuristic that mimics Apriori's workflow in the second of the following two phases. The first phase of our heuristic can be seen as a pre-processing step although it submits exactly the same queries as Apriori does on the first two levels (while $i < 2$). However, for a keyword set W co-occurrence information obtained from the estimations of the first and second Apriori levels are stored in a matrix M in form of the—here called—*yield factors* $\gamma(w, w') = \ell_{\{w,w'\}}/\ell_{\{w\}}$. A yield factor $\gamma(w, w')$ multiplied by $\ell_{\{w\}}$ gives the yield of web results when the keyword w' is added to the query $\{w\}$. Note that the yield factors are not symmetric (i.e., usually $\gamma(w', w) \neq \gamma(w, w')$) such that M also is not symmetric.

The second phase of our heuristic then starts an Apriori-like candidate generation on the third level (queries containing three keywords). Hence, our technique does not save queries on the first two levels compared to our baseline Algorithm 1 but from Level 3 on the heuristic uses the yield factors to internally estimate a query and only submit it as a web query if necessary. Assume we are on some level $i \geq 3$ and that all processed queries Q from lower levels have a stored value est_Q, indicating an estimation of the length of their result lists, and a value age_Q, indicating the elapsed Apriori levels from the last time a subset of Q was submitted as a web query. Hence, for $age_Q = 0$ we have $est_Q = \ell_Q$. Let the current candidate query Q_{cand} be obtained by merging Q and Q' (line 7 of Apriori). Before submitting a web query, we now internally compute $est_{Q_{\text{cand}}}$ as follows. Let $age_Q \leq age_{Q'}$ and $Q' \setminus Q = \{w'\}$. We set $est_{Q_{\text{cand}}} = est_Q \cdot \text{avg}\{\gamma(w, w') : w \in Q\}$, where avg denotes the mean value. Submitting Q_{cand} as a web query and storing the engine's $\ell_{Q_{\text{cand}}}$ as $est_{Q_{\text{cand}}}$ is done iff $est_{Q_{\text{cand}}} < adj \cdot \ell_{\max}$ for a given adjustment factor adj. If however $est_{Q_{\text{cand}}} \geq adj \cdot \ell_{\max}$ we do not submit a web query but store the internally derived $est_{Q_{\text{cand}}}$ and set $age_{Q_{\text{cand}}} = age_Q + 1$.

The rationale for using the factor adj in the above inequalities is as follows. An experimental in-depth analysis revealed that $\ell_Q \geq est_Q$ holds for most queries Q, though there are rare cases where Q is valid or underflowing while $est_Q > \ell_{\max}$ (i.e., even the tendency of the internal estimation is wrong). For this reason, the informed heuristic does not blindly follow the internal estimations but only trusts them when $est_{Q_{\text{cand}}} \geq adj \cdot \ell_{\max}$ for an adjustment factor adj. The rationale is that as long as the internal estimations are sufficiently above the validity bound $\ell_{\max}$, the probability for a wrong validity check based on the internal estimation is negligible. Only when the internal estimation $est_{Q_{\text{cand}}}$ is close to or below the validity bound $\ell_{\max}$, the current

query is submitted to the search engine in order to "adjust" the internal estimation with the search engine's $\ell_{Q_{\text{cand}}}$. Larger values of adj enlarge the adjustment range and thus guarantee to catch more of the rare cases where Q is valid but $est_Q > \ell_{\max}$. However, this comes with a larger amount of submitted web queries. Moreover, only huge values of adj can guarantee the heuristic to return the same family $\mathcal{Q}_{\text{lo}}$ as the baseline. We compare different realistic settings of adj, and the fine-tuning shows good conformity of the output with the baseline's $\mathcal{Q}_{\text{lo}}$ while saving lots of queries (cf. Section 5).

5 Experimental Analysis

In a first experiment, we compare the uninformed Apriori baseline to our yield factor informed heuristic with respect to the number of submitted queries. In a second experiment, we then compare our heuristic to Bendersky and Croft's and Dasdan et al.'s query formulation strategies [4, 8] with respect to the quality of the retrieved documents. The experimental setting for both experiments is inspired by the observation that scientific publications often follow an evolutionary process from a technical report / workshop / poster / or short paper level to a full conference paper and sometimes to a journal paper. Although the different versions of the same publication have a potentially different and more complete presentation at more mature levels, they still deal with the same topic—such that we assume a significant amount of text reuse among them. To model the described scenario, we crawled computer science papers from major conferences and journals available on the web and tried to find a previous version for each. The document pairs were manually checked to ensure that they really are different versions of the same paper; we obtain 257 such verified pairs, all written in English. We verified that both versions are retrievable using the Bing API that we use in our experiments.

5.1 Number of Submitted Queries

For each of the 257 document pairs we extract a number of keywords from the more mature paper (e.g., conference vs. workshop) and then formulate queries using these keywords. For the keyword extraction we use an implementation of the head noun extractor [3]. We set $\ell_{\min} = 1$ to foreclose queries with no results. Furthermore, we set $k = 1000$ since current state-of-the-art automatic plagiarism detection techniques against a collection of 1000 potential source documents run in about 10 minutes [9], which we consider as a reasonable answer time for most text reuse detection scenarios. For each keyword set extracted from a document of our test collection, we run the Apriori baseline and our heuristic with the first 4, 5, ..., 10 extracted keywords against the Bing API during November 07–20, 2010.

Table 1 contains the results of this experiment. Different settings of the heuristic's adjustment factor correspond to different rows. Note that especially for small numbers of extracted keywords even the complete query containing all keywords is often overflowing. Because $\mathcal{Q}_{\text{lo}}$ cannot be computed in such cases and the first 1000 results of the complete query should be used instead, we filter out the corresponding documents and derive the statistics just for the remaining ones. For those inputs where the computation of $\mathcal{Q}_{\text{lo}}$ is possible, all four approaches always find a $\mathcal{Q}_{\text{lo}}$. We report the average number $cost$ of web queries the approaches submitted to obtain the output $\mathcal{Q}_{\text{lo}}$ and the

Table 1. Results of the number-of-queries experiment

	Number of extracted keywords							
	3	4	5	6	7	8	9	10
Number of documents where								
Complete query overflows	238	207	177	146	124	102	93	81
$\mathcal{Q}_{\mathrm{lo}}$ computation possible	19	50	80	111	133	155	164	176
Average cost (number of submitted queries)								
Heuristic, $adj = 1$	4.91	6.69	9.35	13.30	20.20	32.58	53.13	95.86
Heuristic, $adj = 3$	5.81	7.88	10.85	16.48	26.16	43.44	70.77	125.56
Heuristic, $adj = 5$	5.91	8.73	13.55	20.41	33.30	53.13	87.70	159.16
Uninformed baseline	6.09	10.65	19.08	34.60	61.66	106.19	178.98	302.87
Average cost ratio (basis: uninformed baseline)								
Heuristic, $adj = 1$	**0.80**	**0.63**	**0.49**	**0.38**	**0.33**	**0.31**	**0.30**	**0.32**
Heuristic, $adj = 3$	0.95	0.74	0.57	0.48	0.42	0.41	0.40	0.41
Heuristic, $adj = 5$	0.97	0.82	0.71	0.59	0.54	0.50	0.49	0.53
Average $\lvert\mathcal{Q}_{\mathrm{lo}}\rvert$								
Heuristic, $adj = 1$	1.33	1.85	2.69	3.76	5.25	7.44	10.65	14.72
Heuristic, $adj = 3$	1.32	1.84	2.69	3.75	5.28	7.50	10.88	14.85
Heuristic, $adj = 5$	1.33	1.85	2.72	3.83	5.38	7.61	11.01	14.97
Uninformed baseline	1.33	1.87	2.75	3.88	5.49	7.78	11.12	15.18
Average number of retrievable result URLs (without duplicates)								
Heuristic, $adj = 1$	71	104	157	221	315	428	555	679
Heuristic, $adj = 3$	68	101	158	223	314	422	553	682
Heuristic, $adj = 5$	70	99	161	228	325	439	562	686
Uninformed baseline	69	98	162	231	328	445	569	690
Average ratio of common result URLs with baseline								
Heuristic, $adj = 1$	0.95	0.92	0.92	0.93	0.93	0.94	0.94	0.95
Heuristic, $adj = 3$	0.92	0.97	0.98	0.98	0.97	0.98	0.95	0.97
Heuristic, $adj = 5$	0.98	0.98	0.99	0.99	0.98	0.99	0.98	0.98

average ratio of submitted queries compared to the baseline (smaller *cost* and smaller ratio indicate better approaches). Note that using very few keywords results in fewer retrievable documents using the $\mathcal{Q}_{\mathrm{lo}}$ queries. We also observed that we needed at least 6 keywords to guarantee the retrieval of the original document and its previous version among $\mathcal{Q}_{\mathrm{lo}}$'s web results. Hence, we suggest to use about 10 extracted keywords to obtain a meaningful set of documents from our approaches.

With respect to the runtime, the possible savings in the number of submitted queries are substantial compared to the baseline. For 7 or more keywords our heuristics save 70% of the queries. For all approaches the internal computation time to formulate the queries is never larger than several hundred milliseconds, while a typical web query against the API takes about 300ms–550ms. Hence, the fastest algorithm always is the one that submits the fewest queries. With respect to the quality of the heuristics' $\mathcal{Q}_{\mathrm{lo}}$, we compare the average size of the generated $\mathcal{Q}_{\mathrm{lo}}$ and the ratio of retrieved result URLs common with the baseline's results. The small differences are due to some rare overestimations using the internal expectations that hide some of the queries the baseline finds. It can be observed that larger values of the adjustment factor adj are able to compensate

for more of these overestimations. Additional spot checks show that the difference of the baseline's results to the heuristic with $adj = 1$ is rather small, such that for larger keyword sets the by far better running time should be favored. For keyword sets of size 10 the fastest heuristic with $adj = 1$ computes $\mathcal{Q}_{\text{lo}}$ in about 38 seconds compared to 50 seconds for $adj = 3$, 64 seconds for $adj = 5$, and about 2 minutes for the baseline. This is a saving of 70%. Hence, a near real time text reuse detection service can safely extract 10 keywords.

5.2 Candidate Document Quality

The first experiment shows the heuristic with $adj = 1$ to outperform the other approaches with respect to runtime (while retrieving basically the same set of documents). On the same corpus, we now compare this variant to other previously published query formulation strategies with respect to the quality of the retrieved documents. Important competitors in this regard are Bendersky and Croft's and Dasdan et al.'s query formulation approaches [4, 8].

Bendersky and Croft submit $2n - 1$ queries for a set of n keywords [4]; the first n queries are submitted to obtain the search engine estimations ℓ_w for each keyword w. The keywords are ordered by descending ℓ-value and submitted as a single query containing all n keywords. The approach then iteratively drops the last keyword and submits the resulting queries until 1000 documents are retrieved.

Dasdan et al. describe two approaches [8] that are slightly different right from the keyword extraction. Their first approach (LFT) extracts from a document the terms that are least frequent on the web, using a dictionary with web frequencies like 1-grams from the Google 5-gram corpus [5]. The strategy submits 10 queries: the first one contains the 10 least frequent keywords, the second one contains the next 10 least frequent keywords, and so forth. Dasdan et al.'s second approach (RST) builds 10 queries each of which containing a random sequence of 10 words from the given document. Note that this approach aims to their original problem setting of search engine coverage analysis, where the task is to find near-duplicates of a document in a search engine index. However, such strategies of using a random string from a document are also often suggested as an "intelligent" strategy to manually detect plagiarism. Hence, we decided to employ RST in our candidate document quality experiment. We adjust LFT and RST to only retrieve the top 100 results of each of the constructed queries to ensure for $k = 1000$.

Bendersky and Croft's approach uses the same 10 extracted keywords as our heuristic. Note that for the 81 documents from our collection, where the 10 keywords as one query already overflow, our heuristic and Bendersky and Croft's approach retrieve the same result documents, since both use the first 1000 documents from the all-keywords query. To detect the keywords for LFT, we indexed the Google 1-grams in a big hash table and for our 257 documents in a pre-processing detected the 100 keywords with lowest frequencies. As for the RST approach, a pre-processing sampled 10 random sequences of 10 consecutive words from the documents. Our heuristic obviously submits more queries (65.96 on average: just one query for the 81 documents where $\mathcal{Q}_{\text{lo}}$ is not possible and an average of 95.86 for the other 176 documents) than Bendersky and Croft's approach (10.45 queries on average; note that 19 is the worst case but often 1000 results are retrieved earlier, for 81 documents even with the first query) or LFT

Table 2. Average cosine similarity (tf weights) of the retrieved documents to the given document

Measure	Approach			
	Our heuristic	[4]	LFT	RST
10 most similar documents	0.55	0.55	**0.56**	**0.56**
100 most similar documents	**0.39**	0.37	0.35	0.29
all retrieved documents	**0.29**	0.25	0.22	0.21

and RST (10 queries). With respect to the quality of the retrieved documents, we first check whether the different approaches have the two paper versions among their results. For our heuristic, for Bendersky and Croft's approach as well as for LFT this is always the case. However, RST missed the previous version 8 times. This is probably due to the fact that random sequences help to find nearly-identical versions of a document but that scientists also often rewrite a paper in different versions. Since RST was primarily developed to find near-duplicates, the few misses are no surprise.

The aim of our approach is not only to retrieve the documents from our corpus but also to retrieve similar documents as good candidates: the assumption for text reuse detection is that more similar documents are more likely to contain the assumed text reuse. Hence, we downloaded the results the different approaches retrieve and compare the approaches with respect to similarity of the retrieved documents. As similarity measure we use cosine similarity with tf weights. Table 2 contains the results of this experiment. With respect to the retrieved 10 most similar documents, all approaches are somehow on par. However, this behavior changes significantly when one checks the average similarity for the 100 most similar or even all retrieved documents: then our approach outperforms the other approaches. The gap to Bendersky and Croft's approach is only due to the given documents for which $\mathcal{Q}_{\mathrm{lo}}$ could be computed, since on the other documents our heuristic and Bendersky and Croft's approach return exactly the same documents (the top 1000 results of the query containing all keywords). The gap to LFT and RST is probably due to the slightly different use case as LFT and RST were mainly designed to retrieve a few near-duplicate instances of a given document. That goal is achieved as shown by Dasdan et al.'s experiments [8] and the slightly worse performance in our experiments is probably mainly due to the different scenario that our experiments address.

6 Conclusion and Outlook

We developed a new strategy to formulate promising queries from a given set of keywords. In our scenario a text reuse detection system "plays" against a retrieval system (the web search engine) in order to find promising queries that help to detect text reuse in or from a given document. Our formalization forms the ground for both to define the problem Capacity Constrained Query Formulation and to develop a heuristic search strategy that tackles a query-cost-oriented optimization variant. The analysis of our heuristic shows (1) that it drastically outperforms a maximal termset query formulation baseline system, and (2) that it finds candidate documents which are more similar to the original document than other approaches for related problems. If, however, a method is needed that returns few very similar web results for a given document (like it is the case in Dasdan et al.'s scenario), using our heuristic would probably be an

overhead since it requires more queries. But, if the aim is to retrieve a larger collection of documents all of which are similar to a given document (like it is the case in the text reuse detection scenario), the rather small 20 second overhead of our method can be regarded as a worthwhile investment for a better average similarity. A straightforward extension of the above similarity experiment is an analysis of the retrieved documents with state-of-the-art text reuse or plagiarism detection techniques [4, 9, 13, 16], and to compare the candidate document sets with respect to the number of found text reuse cases (and not just the similarity). However, this is beyond the scope of this paper: having shown the potential of our heuristic for the retrieval step, we leave the examination of text reuse cases as an interesting task for future work.

References

[1] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. of VLDB 1994, pp. 487–499 (1994)

[2] Bar-Yossef, Z., Gurevich, M.: Random sampling from a search engine's index. JACM 55(5) (2008)

[3] Barker, K., Cornacchia, N.: Using noun phrase heads to extract document keyphrases. In: Proc. AI 2000, pp. 40–52 (2000)

[4] Bendersky, M., Croft, W.B.: Finding text reuse on the web. In: Proc. of WSDM 2009, pp. 262–271 (2009)

[5] Brants, T., Franz, A.: Web 1T 5-gram Version 1. LDC2006T13 (2006)

[6] Carmel, D., Yom-Tov, E., Darlow, A., Pelleg, D.: What makes a query difficult? In: Proc. of SIGIR 2006, pp. 390–397 (2006)

[7] Cronen-Townsend, S., Zhou, Y., Croft, W.B.: Predicting query performance. In: Proc. of SIGIR 2002, pp. 299–306 (2002)

[8] Dasdan, A., D'Alberto, P., Kolay, S., Drome, C.: Automatic retrieval of similar content using search engine query interface. In: Proc. of CIKM 2009, pp. 701–710 (2009)

[9] Grozea, C., Gehl, C., Popescu, M.: ENCOPLOT: Pairwise Sequence Matching in Linear Time Applied to Plagiarism Detection. In: Proc. of PAN 2009, pp. 10–18 (2009)

[10] Hagen, M., Stein, B.M.: Capacity-constrained query formulation. In: Lalmas, M., Jose, J., Rauber, A., Sebastiani, F., Frommholz, I. (eds.) ECDL 2010. LNCS, vol. 6273, pp. 384–388. Springer, Heidelberg (2010)

[11] Hauff, C., Hiemstra, D., de Jong, F.: A survey of pre-retrieval query performance predictors. In: Proc. of CIKM 2008, pp. 1419–1420 (2008)

[12] He, B., Ounis, I.: Inferring query performance using pre-retrieval predictors. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 43–54. Springer, Heidelberg (2004)

[13] Kasprzak, J., Brandejs, M.: Improving the Reliability of the Plagiarism Detection System: Lab Report for PAN at CLEF 2010. In: Proc. of PAN 2010 (2010)

[14] Pôssas, B., Ziviani, N., Ribeiro-Neto, B.A., Meira Jr., W.: Maximal termsets as a query structuring mechanism. In: Proc. of CIKM 2005, pp. 287–288 (2005)

[15] Scholer, F., Garcia, S.: A case for improved evaluation of query difficulty prediction. In: Proc. of SIGIR 2009, pp. 640–641 (2009)

[16] Seo, J., Croft, W.B.: Local text reuse detection. In: Proc.of SIGIR 2008, pp. 571–578 (2008)

[17] Stein, B., Hagen, M.: Introducing the user-over-ranking hypothesis. In: Clough, P., Foley, C., Gurrin, C., Jones, G.J.F., Kraaij, W., Lee, H., Mudoch, V. (eds.) ECIR 2011. LNCS, vol. 6611, pp. 503–509. Springer, Heidelberg (2011)

[18] Wu, X., Kumar, V.: The Top Ten Algorithms in Data Mining. CRC Press, Boca Raton (2009)

[19] Yang, Y., Bansal, N., Dakka, W., Ipeirotis, P.G., Koudas, N., Papadias, D.: Query by document. In: Proc. of WSDM 2009, pp. 34–43 (2009)

A Multi-faceted Approach to Query Intent Classification*

Cristina González-Caro[1,3] and Ricardo Baeza-Yates[2,3]

[1] Universidad Autónoma de Bucaramanga
Avenida 42 No. 48, Bucaramanga, Colombia
cgonzalc@unab.edu.co
[2] Yahoo! Research Barcelona
Barcelona, Spain
rbaeza@acm.org
[3] Web Research Group
Dept. of Information and Communication Technologies
Universitat Pompeu Fabra, Barcelona

Abstract. In this paper we report results for automatic classification of queries in a wide set of facets that are useful to the identification of query intent. Our hypothesis is that the performance of single-faceted classification of queries can be improved by introducing information of multi-faceted training samples into the learning process. We test our hypothesis by performing a multi-faceted classification of queries based on the combination of correlated facets. Our experimental results show that this idea can significantly improve the quality of the classification. Since most of previous works in query intent classification are oriented to the study of single facets, these results are a first step to an integrated query intent classification model.

1 Introduction

As the Web continues to increase both in size and complexity, Web search is a ubiquitous service that allows users to find information, resources, and activities. However, as the Web evolves so do the needs of the users. Nowadays, users have more complex interests that go beyond to the traditional *informational queries*. For example, many users may want to perform a particular commercial transaction, locate a special service, etc. Thus, it is important for Web-search engines, not only to continue answering effectively informational and navigational queries, but also to be able to identify and provide accurate results for new types of queries.

Based on the premise above, Web-search engines try to improve the quality of their results by adopting a number of different strategies. For instance, diversification of search results aims at providing a list of diversified results that

* This research was partially funded by the Coordinated Research Grant TIN2009-15536-C02-1 of the Spanish Ministry of Science and Technology.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 368–379, 2011.
© Springer-Verlag Berlin Heidelberg 2011

cover different interpretations of ambiguous queries [15]. The objective of diversification is to identify ambiguous queries and present the best results for each meaning of those queries. The first step towards this goal, is to identify the type of the query. In this respect, all the recent efforts to describe and identify the intent of the user's query are of great value [1,7,6]. Although there is a lot of work in the topic of identifying query intent, most of it is based on the analysis of only one possible facet of the query. The most common of these facets are the *topic category* and the *type of query intent*; mainly based on Broder's taxonomy [3]. However, one can argue that classifying a query with respect to one facet may improve the classification with respect to another facet. For example, knowing that a query topic is *art* increases the prior that the query intent is *informational*. Similarly, knowing that a query topic is *electronics* increases the prior that the query intent is *transactional*. Hence, we argue that identification of the query intent is a multi-faceted problem. We show that by treating the problem as such we can significantly improve the accuracy of the classification problem.

In this paper, we explore the automatic classification of queries in a wide set of facets that are useful to the identification of query intent. We also investigate whether combining multiple facets can improve the predictability of the facets. As result, our contributions to query-intent classification include:

- We show the feasibility automatic faceted-classification of a large set of queries with a comprehensive set of facets, for the prediction of query intent.
- We propose a multi-faceted classification of queries based on the combination of correlated facets. We evaluate the performance of each combination of facets and its impact on each individual facet.
- We compare the results of multi-faceted classification with the conventional faceted classification to determine if the combination of related facets can improve the identification of the intent of the user queries.
- We provide an extensive experimental evaluation showing that the combination of facets proposed in this paper can significantly improve the quality of the classification results.

To the best of our knowledge, this is the first work that explores automatic multi-faceted classification of user's query intent. Previous work has considered the multi-faceted classification of the query intent an open research problem [7].

2 Related Work

According to Cool and Belkin [5], users engage in multiple information seeking behavior within the context of accomplishing a single task. Therefore, it is important to have Web-search systems able to support multiple information seeking behaviors, and multiple interactions with the information. In this direction, many efforts have been devoted in trying to understand the intent of user queries. The understanding of user queries has been conducted from different facets. The first approaches to identify query intent include classifying queries based on the

topic [2,17,10]. Topical associations of queries are important because they allow to place the queries in a particular context. A second line of work has focused on the classification of queries regarding the type of intent. In this case, the intent of the query refers to the type of resource associated with the query. The first taxonomy of query intent, proposed by Broder [3], defines three types of query intent: informational, navigational and transactional. Rose and Levinson [16] extended Broder's taxonomy by adding hierarchical sub-categories for informational and transactional queries. Based on these early taxonomies many works have attempted automatic classification of queries [9,11,7,6]. Other approaches to identifying query intent consider facets like geographic locality [8], time sensitivity [12], ambiguity [19] and specificity through ambiguity levels [18]. However, each of these works has been restricted to the analysis of only one facet. Meanwhile, there have been few works attempting to classify query intent in more than one facet. Baeza-Yates et al. [1] presented an approach for automatic classification of queries into topic and query intent (informational, not informational and ambiguous). Nguyen and Kan [13] analyzed a set of four facets (ambiguity, authority sensitivity, temporal sensitivity and spatial sensitivity). However, despite that they presented a query log analysis of the four facets, they only provided automatic classification results for one of the facets: authority sensitivity.

In summary, none of studies mentioned implements multi-faceted classification of query intent. They only implement automatic classification of query intent based on the information of individual facets.

3 Experimental Design

In this work, we use a data set composed of 4,726 unique queries extracted from a query-log of a vertical search engine. The queries were randomly in a way that allowed us to work with queries with different level of popularity according to the Zipf's law distribution, considering a sample of popular, normal and long tail queries. Each query was represented as a weighted term vector restricted to the terms appearing in the most popular Web pages for that query. The queries were manually classified into a set of nine facets, that can be used for the identification the query intent. These facets are:

Genre{*News, Business, Reference, Community*}: this facet provides a generic context to the user's query intent, and can be thought as a meta-facet.

Topic {*Adult, Arts & Culture, Beauty & Style, Cars & Transportation, Charity, Computers & Internet, Education, Entertainment, Music & Games, Finance, Food & Drink, Health, Home & Garden, Industrial Goods & Services, Politics & Government, Religion & belief systems, Science & Mathematics, Social Science, Sports, Technology & Electronic, Travel, Undefined, Work*}: a list of topics built from the first level of categories offered by ODP (www.dmoz.org), Yahoo! (www.yahoo.com), and Wikipedia (en.wikipedia.org).

Task {*Informational, Not Informational, Ambiguous*}[1]: this facet is related with the type of resource associated with the query.

Table 1. Performance evaluation of automatic prediction for Task

Training/Testing	50%/50%			70%/30%		
Task	Precision	Recall	F-measure	Precision	Recall	F-measure
Informational	0.7037	**0.9889**	**0.8223**	0.7227	**0.9915**	**0.8360**
Not Informational	0.8408	0.2670	0.4053	**0.8917**	0.2948	0.4431
Ambiguous	**0.9167**	0.0550	0.1038	0.8571	0.0526	0.0992
Average	0.8204	0.4370	0.4438	0.8238	0.4463	0.4594

Table 2. Performance evaluation of automatic prediction for Objective

Training/Testing	50%/50%			70%/30%		
Objective	Precision	Recall	F-measure	Precision	Recall	F-measure
Action	**0.9451**	0.1673	0.2843	**0.9375**	0.2007	0.3306
Resource	0.8116	**0.9973**	**0.8949**	0.8235	**0.9964**	**0.9017**
Average	0.8783	0.5823	0.5896	0.8805	0.5985	0.6162

Objective {*Resource, Action*}: represents if the query is aimed to do some action or to obtain a resource.

Specificity {*Specific, Medium, Broad*}: this facet describes how specialized is a query.

Scope {*Yes, No*}: the scope aims at capturing whether the query contains polysemic words or not.

Authority Sensitivity {*Yes, No*}: through this facet it is possible to determining whether the query is designed to retrieve authoritative and trusted answers [13]

Spatial Sensitivity {*Yes, No*}: this facet reflects the interest of the user to get a resource related to an explicit spatial location.

Time Sensitivity {*Yes, No*}: this facet captures the fact that some queries require different results when posed at different times [13].

The facets genre, objective, specificity and scope are considered for first time for query intent classification. The rest of the facets have been considered in previous works for query intent classification (see Section 2).

4 Predicting Individual Facets

In this section we use the labeled data-set described in Section 3 to train user intent prediction models. We trained a single support vector machine (SVM) classifier for each facet. The software used to implement SVM was LIBSVM [4]. We selected the one-against-one multi-class strategy using majority voting to decide the final output. We used the radial basis function kernel for the SVM algorithm. For the experiments, three metrics were considered: recall, precision and the F-measure.

The results for the automatic prediction of the facets are reported in Tables 1 to 5. As we can see from these tables, in general we obtain good results for estimating the facets. There is not too much difference between the results obtained

with the automatic classifiers based on training-sets of 50% of queries and those classifiers based on training-sets with 70% of queries. That is, the prediction's performance is good for different quantities of training data.

The best results are for the facets task and objective (see Tables 1 and 2). The average precision for these facets is 0.822 and 0.879 respectively. In the case of task, the classifier was most useful for predicting *Informational* and *Not Informational* queries, where we can observe a good balance of precision and recall (see F-measure values). For queries in the *Ambiguous* category, we maintain high precision (0.88 on average) at the expense of recall. In the case of objective, the classifier is very effective, specially to distinguish *Resource* queries (F-measure is 0.89 on average). For *Action* queries the precision is also very good (0.94 on average), although the F-measure is not as high as for *Resource* queries. In general, the automatic classification results for task and objective show the feasibility of the prediction of these facets. This is important, as being able to correctly identify the task and the objective of queries give us insight into the query intent and the type of resource associated to it. In Figure 1 (left) we show the distribution of queries along the task and the objective. As we can observe, *Informational* queries have a clear orientation towards *Resource*-objective and *Not Informational* queries are more oriented towards *Action*-objective. An interesting point is that *Ambiguous* queries are also oriented towards *Resource*-objective. This finding suggest that *Action*-objective queries are less ambiguous than *Resource*-objective queries. Since most of the queries with an *Action*-objective belong to the *Not Informational*- task we can say that the ambiguity of *Not Informational* queries is low.

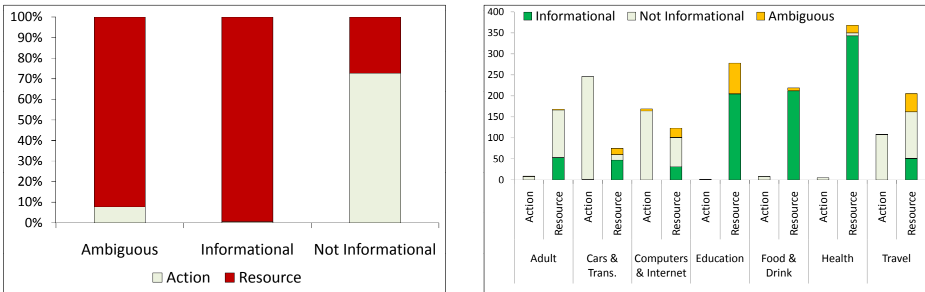


Fig. 1. Distribution of queries into facets Task and Objective (left) and distribution of queries into facets Topic, Task and Objective (right)

Table 3 shows the performance results for the facet genre. We obtain good genre identification performance for *Business*, *Community* and *Reference*, as SVM classifiers yielded overall precision around 0.76 (F-measure is 0.46 on average). These three genre categories are the most representative categories of the facet; they group together 97.4% of the total of queries. For the category *News* the classifier obtains a F-measure of zero, which may be caused by the small number of queries belonging to this category (56 queries in the training set, 122

Table 3. Performance evaluation of automatic prediction for Genre

Training/Testing	50%/50%			70%/30%		
Genre	Precision	Recall	F-measure	Precision	Recall	F-measure
Business	**0.9146**	0.2884	0.4386	**0.8984**	0.3159	0.4675
Community	0.5649	**0.9867**	**0.7185**	0.5765	**0.9836**	**0.7269**
Reference	0.8033	0.1038	0.1839	0.8537	0.1203	0.2108
Average	0.7609	0.4597	0.4470	0.7762	0.4733	0.4684

Table 4. Performance evaluation of automatic prediction for Topic (values above 0.8 are highlighted)

Training/Testing	50%/50%			70%/30%		
Topic	Precision	Recall	F-measure	Precision	Recall	F-measure
Adult	0.7692	0.2000	0.3175	0.7000	0.2414	0.3590
Arts & Culture	0.5769	0.0938	0.1613	0.5556	0.1020	0.1724
Cars & Transportation	**0.9107**	0.3778	0.5340	0.8800	0.2857	0.4314
Computers & Internet	**0.8600**	0.3308	0.4778	0.8261	0.3167	0.4578
Education	0.8043	0.2483	0.3795	**0.8529**	0.3222	0.4677
Entertainment	0.1502	**0.8743**	0.2564	0.5474	0.4685	0.5049
Finance	0.6622	0.3858	0.4876	0.2149	**0.8052**	0.3393
Food & Drink	**0.8667**	0.2364	0.3714	**0.9500**	0.2923	0.4471
Health	0.6383	0.6667	0.6522	0.7347	0.4800	0.5806
Home & Garden	0.7692	0.1538	0.2564	0.5600	0.1667	0.2569
Politics & Government	0.4483	0.4437	0.4460	0.4903	0.5549	0.5206
Religion & Belief Systems	**1**	0.1224	0.2182	**1**	0.1515	0.2632
Science & Mathematics	0.4279	0.6394	0.5127	0.3846	0.6765	0.4904
Social Science	0.4375	0.0753	0.1284	0.3636	0.0690	0.1159
Travel	0.7083	0.2252	0.3417	0.7333	0.2472	0.3697
Average	0.6687	0.3382	0.3694	0.6529	0.3453	0.3851

in total). Apart of the *news* category, the performance of prediction for the facet genre is good. For the facet topic, the classifier provides predictions for fifteen of the twenty topics that were considered for the classification, as shown in Table 4. Overall, the precision is good, and over 0.6 for most of the topics. The best precision values are for the topics: *Adult, Cars & Transportation, Computers & Internet, Education, Food & Drink* and *Health*. These topics group an important number of queries and have interesting connections with the other facets (see Figure 1 right).

Table 5 shows the performance of the prediction for the facets authority sensitivity, spatial sensitivity, time sensitivity, scope and specificity. The results for this group of facets is good. Some of the facets obtain better results than the others, but the overall results are balanced. The facets spatial sensitivity and authority sensitivity show the best precision results. To be able to identify correctly these two facets is important because the search results for spatially-sensitive and authority-sensitive queries, must be both relevant to the query and valid for the

Table 5. Performance evaluation of automatic prediction for Authority Sensitivity, Spatial Sensitivity, Time Sensitivity, Scope and Specificity

Training/Testing	50%/50%			70%/30%		
Facet	Precision	Recall	F-measure	Precision	Recall	F-measure
Authority Sen.	**0.8570**	0.5193	0.4865	0.8441	0.5245	0.4961
Spatial Sen.	0.7526	**0.6140**	**0.5471**	**0.7618**	**0.6494**	**0.5913**
Time Sen.	0.4884	0.50	0.4941	0.4873	0.50	0.4936
Scope	0.4945	0.50	0.4972	0.4958	0.50	0.4979
Specificity	0.5126	0.3412	0.3101	0.5060	0.3444	0.3160

associated location and for the authoritative requirement. This is the typical case while searching in a mobile device. The performance for time sensitivity, scope and specificity is similar in average.

5 Combining Multiple Facets

Although the set of facets that we are studying here are different dimensions of the user's intent and not all of them are necessarily correlated, we are interested to study how the combination of multiple facets in the classification process can improve the prediction performance. We selected two groups of correlated facets and performed a multi-faceted classification of the queries with these facets.

We address the problem of classifying a query into a set of relevant facets as a multi-label classification problem. The traditional *single-label* classification, also known as multi-class classification, is the common machine learning task where an instance is assigned a single label ℓ, that is chosen from a previously known finite set of labels L. A data-set D of n instances is composed of instance-classification pairs $(x_0, \ell_0), (x_1, \ell_1), ..., (x_n, \ell_n)$. The *multi-label* classification task is an extension of this problem, where each instance is associated with a subset of labels $S \subseteq L$. A multi-label data-set D of n instances is composed of instance-classification pairs $(x_0, S_0), (x1, S_1), ..., (x_n, S_n)$. Learning a multi-label model can be achieved through one of two approaches: problem transformation methods, and algorithm adaptation methods. The problem-transformation methods, transform the multi-label classification problem into one or more single-label classification or regression problems. The adaptation methods, extend specific learning algorithms in order to handle multi-label data directly [20]. In this work we explore a problem-transformation method, which reduces the multi-label classification problem to a multi-class classification problem by treating each distinct label set as a unique multi-class label. This transformation explicitly captures overlaps between facets, which is one of our goals. Since SVMs have shown good generalization ability in different single-label multi-class problems, is also one of the most used techniques to resolve multi-label classification problem [14]. We used the multi-label classification tool of LIBSVM (available at http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/multilabel/) to build multi-label classifiers for our group of facets. We selected the label combination option as the transformation method.

Table 6. Multi-label classification results based on the combination genre-objective for the facet genre (50% training and 50% testing)

Genre	Precision	Recall	F-measure
Business	**0.7620**	0.7052	0.7325
Community	0.7204	**0.7874**	**0.7524**
News	0.2727	0.2143	0.2400
Reference	0.5561	0.4936	0.5230
Average	0.5778	0.5501	0.5620

Table 7. Multi-label classification results based on the combination genre-objective for the facet objective (50% training and 50% testing)

Objective	Precision	Recall	F-measure
Action	0.6762	0.6420	0.6587
Resource	**0.9019**	**0.9145**	**0.9082**
Average	0.7890	0.7783	0.7834

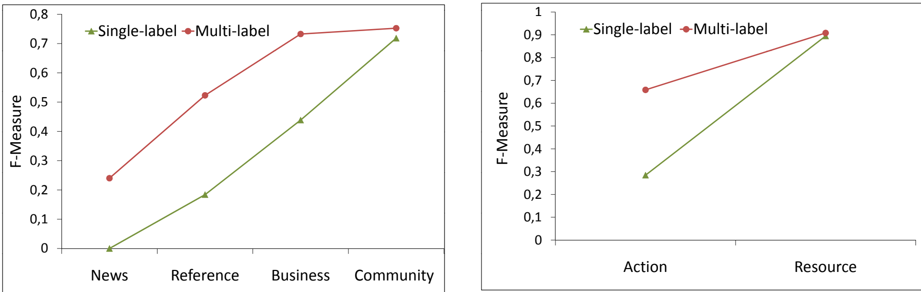


Fig. 2. Comparison of single-label and multi-label F-measure results for the facets genre (left) and objective (right)

5.1 Genre-Objective Combination

Two of the most descriptive and correlated facets are genre and objective; for this reason we selected this combination of facets to test multi-label classification. In order to obtain comparable results, the training and test sets used for multi-label classification are the same data-sets used to perform single-label classification. That is, we used the same group of queries and the only variation are the training-set labels. In this case, each query was marked with two values, genre and objective, respectively.

Table 6 shows the multi-label classification results for the facet genre. Overall, multi-label classification outperforms the single-label classification for all categories of genre. Specially, we note important improvements in recall for small categories (i.e., categories with less representation of queries in the sample data) like *News* and *Reference*. For *Reference*, single-label classification yielded good

precision result (0.82 on average) but the recall value is low (0.11 on average). With the multi-label classification the results for *Reference* are more balanced, the recall improves to near 0.5 and the precision is still good (above 0.55). In the case of *News*, the overall performance is not so good, but it is noticeable that the multi-label classifier provides predictions for this category, given that the single-label classifier did not report any results for this case, because was too small. When we use multi-label classification, the additional information provided with the multi-label training set allows the classifier to predict small categories. For the large categories the multi-label classification also improved the results. In Figure 2 (left) we can see the comparison of the performance (F-measure values) of single-label and multi-label classification for the facet genre. The results of the multi-label classification for the facet objective are shown in Table 7, being the results for objective even better. With respect to the single-label classification results, the major improvements are for the category *Action*, as seen in Figure 2 (right), where recall improves dramatically from 0.18 to 0.64. For the category *Resource* the multi-label classifier maintains the high recall and precision obtained with the single-label classification (F-measure is 0.908).

Overall, the combination genre-objective is positive for multi-label classification, obtaining better performance than the traditional single-label classification.

5.2 Genre-Task-Topic Combination

Three of the most important facets we are evaluating are genre, task and topic. The combination of these facets might influence the prediction results of the queries. Analyzing the data set used, we find that there are some topics that are oriented to specific genre categories. For instance, the topic *Cars & Transportation* is related with the genre-category *Business* and the topic *Politic & Government* is related with the genre-category *Community*. These relations suggest that knowing that a query belongs to a particular topic, could also indicate that the query belongs to a particular genre-category. Hence, we test this hypothesis through performing multi-label classification with these facets.

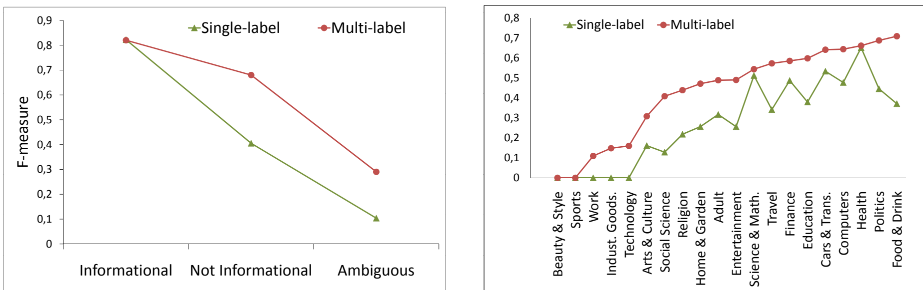


Fig. 3. Comparison of single-label and multi-label F-measure results for the facets task (left) and topic (right)

Table 8. Multi-label classification results based on the combination genre-task-topic for the facet genre (50% training and 50% testing)

Genre	Precision	Recall	F-measure
Business	**0.7170**	**0.7068**	0.7119
Community	**0.7116**	**0.7807**	**0.7446**
News	0.1538	0.3214	0.2081
Reference	0.5908	0.3792	0.4619
Average	0.5433	0.5471	0.5316

Table 9. Multi-label classification results based on the combination genre-task-topic for the facet task (50% training and 50% testing)

Task	Precision	Recall	F-measure
Informational	**0.8235**	**0.8170**	**0.8202**
Not Informational	0.6923	0.6682	0.6801
Ambiguous	0.2692	0.3150	0.2903
Average	0.5950	0.6001	0.5969

Table 8 shows the performance evaluation of the multi-label classification for the facet genre. In general, we obtain very similar results to the multi-label classification based on the combination genre-objective (see Table 6). The difference between the results of the two multi-label classifications is not high, the combination genre-objective has slightly better results, but the improvements are similar in both cases. Since genre is the facet that has more positive correlations with other facets, it could be combined in different ways with several facets. The results of the multi-label classification for the facet task are shown in Table 9. The multi-label classification maintain the good results of the large categories and improve the results of the small categories. In this case, the large category is *Informational*. For this category, the multi-label classifier yields similar results with the single label classifier (F-measure is 0.82). For the other two categories, *Not Informational* and *Ambiguous*, the multi-label classifier yields much better results than the single-label classifier. In Figure 3 (left) we can see a comparison between the overall results (F-measure values) of the multi-label classification and the single-label classification for the facet task.

Finally, in Table 10 we show the multi-label classification results for the facet topic. The multi-label classifier provides predictions for eighteen of the twenty considered topics, while with the single-label classification, five topics obtained F-measure values of zero. Hence, when we use combinations of facets to train automatic classifiers, the probability to predict small categories increases. We observe this effect of multi-label classification in facets like topic and genre. In general, the coverage (recall) of topics is substantially increased. The results of the multi-label classification are more balanced (precision-recall) than the results of the single-label classification. Figure 3 (right) shows the F-measure values for the two types of classifications. As we can observe, multi-label

Table 10. Multi-label classification results based on the combination genre-task-topic for the facet topic (50% training and 50% testing)

Topic	Single-Label			Multi-Label		
	Precision	Recall	F-measure	Precision	Recall	F-measure
Adult	0.7692	0.2000	0.3175	0.5500	0.4400	0.4889
Arts & Culture	0.5769	0.0938	0.1613	0.2313	0.4625	0.3083
Cars & Transportation	0.9107	0.3778	0.5340	0.7333	0.5704	0.6417
Computers & Internet	0.8600	0.3308	0.4778	0.6214	0.6692	0.6444
Education	0.8043	0.2483	0.3795	0.6560	0.5503	0.5985
Entertainment	0.1502	**0.8743**	0.2564	0.4944	0.4863	0.4904
Finance	0.6622	0.3858	0.4876	0.5927	0.5787	0.5857
Food & Drink	0.8667	0.2364	0.3714	**0.7742**	0.6545	**0.7094**
Health	0.6383	0.6667	**0.6522**	0.6194	**0.7111**	0.6621
Home & Garden	0.7692	0.1538	0.2564	0.5455	0.4154	0.4716
Industrial Goods and Services	–	–	–	0.5000	0.0870	0.1481
Politics & Government	0.4483	0.4437	0.4460	0.7245	0.6553	0.6882
Religion & Belief Systems	**1**	0.1224	0.2182	0.5455	0.3673	0.4390
Science & Mathematics	0.4279	0.6394	0.5127	0.6293	0.4796	0.5443
Social Science	0.4375	0.0753	0.1284	0.3431	0.5054	0.4087
Technology and Electronic	–	–	–	0.2000	0.1333	0.1600
Travel	0.7083	0.2252	0.3417	0.6074	0.5430	0.5734
Work	–	–	–	0.0606	0.5714	0.1096

classification outperforms the single classification in all the topics. In some of them with remarkable improvements, like *Food & Drink* and *Politics & Government* categories.

In summary, the combination of the facets genre, task and topic is good for the automatic prediction of each facet. Specially, we show that the coverage of the facets increases considerably and the precision is more balanced.

6 Conclusions

In this paper we explored the feasibility of multi-faceted query intent prediction. We first perform individual classification processes for nine facets and then we perform multi-label classification based on the combination of correlated facets. Our experimental evaluation show that the combination of correlated facets can effectively improve the prediction of the query intent. Some previous works have analyzed several facets, but they have not shown how these facets can be combined to automatically identify the intent of the query. As we can see from the results of our experiments, the best performance is obtained with multi-label classification (multi-faceted classification). There are several potential areas for future work, including the study of the optimal combination of facets that could be implemented through multi-faceted classification.

References

1. Baeza-Yates, R., Calderón-Benavides, L., González-Caro, C.N.: The Intention Behind Web Queries. In: Crestani, F., Ferragina, P., Sanderson, M. (eds.) SPIRE 2006. LNCS, vol. 4209, pp. 98–109. Springer, Heidelberg (2006)
2. Beitzel, S.M., Jensen, E.C., Frieder, O., Lewis, D.D., Chowdhury, A., Kolcz, A.: Improving automatic query classification via semi-supervised learning. In: ICDM 2005, pp. 42–49. IEEE, Los Alamitos (2005)
3. Broder, A.: A taxonomy of web search. SIGIR Forum 36, 3–10 (2002)
4. chung Chang, C., Lin, C.J.: Libsvm: a library for support vector machines (2001)
5. Cool, C., Belkin, N.J.: A classification of interactions with information. In: Proceedings of the Fourth International Conference on Conceptions of Library and Information Science, pp. 1–15. Libraries Unlimited, Greenwood Village (2002)
6. Herrera, M.R., de Moura, E.S., Cristo, M., Silva, T.P., da Silva, A.S.: Exploring features for the automatic identification of user goals in web search. Inf. Process. Manage. 46, 131–142 (2010)
7. Jansen, B.J., Booth, D.L., Spink, A.: Determining the informational, navigational, and transactional intent of web queries. Inf. Process. Manage. 44(3), 1251–1266 (2008)
8. Jones, R., Zhang, W.V., Rey, B., Jhala, P., Stipp, E.: Geographic intention and modification in web search. Int. J. Geogr. Inf. Sci. 22, 229–246 (2008)
9. Lee, U., Liu, Z., Cho, J.: Automatic identification of user goals in web search. In: WWW 2005, pp. 391–400. ACM, New York (2005)
10. Li, X., Wang, Y.Y., Shen, D., Acero, A.: Learning with click graph for query intent classification. ACM Trans. Inf. Syst. 28, 12:1–12:20 (2010)
11. Liu, Y., Zhang, M., Ru, L., Ma, S.: Automatic query type identification based on click through information. In: Ng, H.T., Leong, M.-K., Kan, M.-Y., Ji, D. (eds.) AIRS 2006. LNCS, vol. 4182, pp. 593–600. Springer, Heidelberg (2006)
12. Metzler, D., Jones, R., Peng, F., Zhang, R.: Improving search relevance for implicitly temporal queries. In: SIGIR 2009, pp. 700–701. ACM, New York (2009)
13. Nguyen, V.B., Kan, M.Y.: Functional faceted web query analysis. In: Amitay, E., Murray, C.G., Teevan, J. (eds.) Query Log Analysis: Social And Technological Challenges. A Workshop at WWW 2007 (2007)
14. Qin, Y.-p., Wang, X.-k.: Study on multi-label text classification based on svm. In: FSKD 2009, vol. 01, pp. 300–304. IEEE Computer Society, Los Alamitos (2009)
15. Rafiei, D., Bharat, K., Shukla, A.: Diversifying web search results. In: WWW 2010, pp. 781–790. ACM, New York (2010)
16. Rose, D.E., Levinson, D.: Understanding user goals in web search. In: WWW 2004, pp. 13–19. ACM, New York (2004)
17. Shen, D., Sun, J.T., Yang, Q., Chen, Z.: Building bridges for web query classification. In: SIGIR 2006, pp. 131–138. ACM, New York (2006)
18. Song, R., Luo, Z., Nie, J.Y., Yu, Y., Hon, H.W.: Identification of ambiguous queries in web search. Inf. Process. Manage. 45, 216–229 (2009)
19. Teevan, J., Dumais, S.T., Liebling, D.J.: To personalize or not to personalize: modeling queries with variation in user intent. In: SIGIR 2008, pp. 163–170. ACM, New York (2008)
20. Tsoumakas, G., Katakis, I.: Multi-label classification: An overview. Int. J. Data Warehousing and Mining 2007, 1–13 (2007)

Navigating the User Query Space

Ronan Cummins[1], Mounia Lalmas[2], Colm O'Riordan[3], and Joemon M. Jose[1]

[1] School of Computing Science, University of Glasgow, UK
[2] Yahoo! Research, Barcelona, Spain
[3] Dept. of Information Technology, National University of Ireland, Galway, Ireland
ronan.cummins@nuigalway.ie

Abstract. Query performance prediction (QPP) aims to automatically estimate the performance of a query. Recently there have been many attempts to use these predictors to estimate whether a perturbed version of a query will outperform the original version. In essence, these approaches attempt to navigate the space of queries in a guided manner.

In this paper, we perform an analysis of the *query space* over a substantial number of queries and show that (1) users tend to be able to extract queries that perform in the top 5% of all possible user queries for a specific topic, (2) that *post-retrieval* predictors outperform *pre-retrieval* predictors at the high end of the query space. And, finally (3), we show that some post retrieval predictors are better able to select high performing queries from a group of user queries for the same topic.

1 Introduction

Query performance prediction (QPP) (or estimating query difficulty) has become a vibrant research area in the last decade. Predicting the performance of a query is a useful task for many reasons. For example, search engines may wish to augment queries in different ways depending on their estimated performance. In fact, if query performance prediction becomes good enough [6], the space of all possible queries for a given topic may be able to be navigated efficiently, so that an initial query can be perturbed effectively. Furthermore, such techniques might be effective for creating good queries when a large number of terms are available. Query performance predictors can be used in conjunction with information extraction techniques to be able to extract good queries from these longer information needs. These approaches may ultimately help in shifting the cognitive load of query creation from the user to the system.

In this paper, we analyse the space of possible user queries (under some assumptions) over a range of topics and collections. In particular, we show that (1) while there are a number of queries which are extremely effective, humans create queries which peform within the top 5% of all possible user queries that can be extracted from a given *information need* (IN) under certain assumptions. Furthermore, (2) we show that post retrieval predictors are more effective than pre-retrieval predictors for predicting the performance of user queries (i.e. high

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 380–385, 2011.
© Springer-Verlag Berlin Heidelberg 2011

performing queries for a topic). Finally, (3) we demonstrate that some post-retrieval predictors are very successful at selecting high performing queries from a set of user queries (for the same topic).

The remainder of the paper is organised as follows: Section 2 presents background and related research that is relevant to this work. Section 3 comprises three parts. In section 3.1, we perform an analysis of the query space for a number of topics and collections. In section 3.2, we conduct a study which outlines the correlation of numerous pre- and post-retrieval predictors on sets of user queries for the same topic. Section 3.3 demonstrates a practical application of using predictors to select good user queries. Finally, section 4 outlines our conclusions.

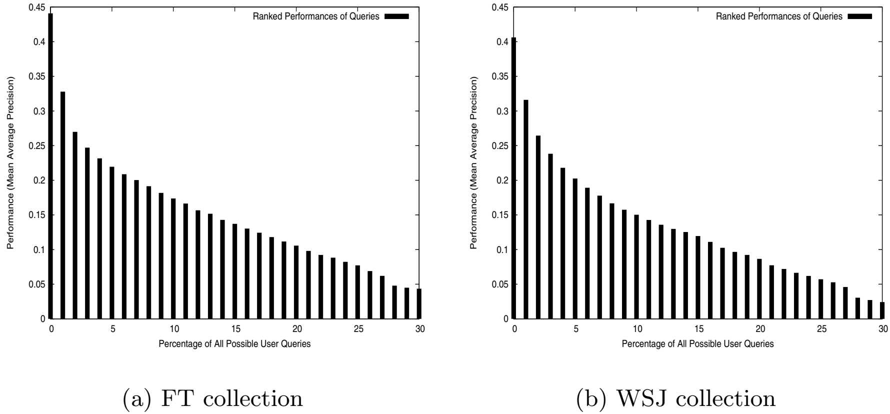
2 Background and Related Research

Fundamentally, retrieval predictors can be divided into two classes: *pre-retrieval* [7,6,12] and *post-retrieval* [3,10,11] predictors. Pre-retrieval predictors use features from the query, document and collection before a query has been processed in order to ascertain its performance. Conversely, post-retrieval predictors analyse the result list, scores and complex features to create predictors that have a higher overhead in terms of computation [2]. One of the earliest approaches to QPP has been that of the clarity score [3], which measures the KL-divergence between the query and collection model in a language modelling framework. Recent research has shown that the standard deviation (σ) of scores in a ranked list is a good predictor of query performance [10,11] for the traditional QPP task. It has also been shown [10] that even better prediction can be obtained if a variable cut-off point is used (i.e. different cut-off points for different queries). A relatively new predictor has also been introduced where the standard deviation of the first N documents is calculated, where N is the number of documents in the head of the list that are within a certain a percentage (i.e. 50%) of the top score [5].

Recently work has been conducted into combining retrieval predictors with the aim of improving performance by reducing queries that may contain noisy terms (e.g. noisy terms in the description field of topics) [9,1]. Some work similar to the research outlined herein has been conducted [8]. However, we place the problem of selecting user queries in a query prediction framework, and review a substantial number of high performing pre-retrieval and post-retrieval methods. We also conduct an analysis of how effective users are at the task of query extraction.

3 Experimental Analysis

In this section, we conduct and analysis of the query space. Firstly, we show that the ranked performance of queries follows a power law distribution, and that user create queries that lie within the fifth percentile of such a distribution. Then, we perform an analysis of a number of pre-retrieval and post-retrieval predictors and show that post-retrieval predictors can more easily predict high performing queries.



(a) FT collection (b) WSJ collection

Fig. 1. Performance of All Queries

3.1 Sampling the User Query Space

First we outline two assumptions that constrain this work; (1) We assume user queries consist of queries of not longer than six terms (research has indicated that the vast majority of user queries are indeed less that this). And (2) we assume that user queries comprise terms that appear somewhere in the TREC topic statement (i.e. *title*, *desc*, and *narrative*), as these topic statements model actual *information needs*).

Now, to analyse the user query space in a thorough manner, we wish to sample a large number of the high performing queries that a user might possibly generate (when prompted with an *information need*). Given that there are 2^N possible queries for an *information need* of N terms, we cannot exhaustively evaluate and analyse all possible queries. Therefore, we create a sample of queries for a topic in the following manner; (1) We extract the top 20 (i.e. $N = 20$) most discriminative terms (*idf*) from the topic (this is all the terms for some topics) to be used in our sample user queries. (2) We submit *all* queries of length one and two terms, and record their performance (average precision). Then (3), for all other queries from length three to six terms (in that order), if a query has an estimated[1] performance within 66% of the best query found thus far for that information need, we submit it to the system and record its performance. Therefore, we are quite confident that by the end of the process we have a large selection of queries within the high end of the query space. Figure 1 shows that the distribution of queries when ranked by performance (mean average precision) follows a power law (i.e. there are few high performing queries and many poorly performing ones).

[1] When estimating a query of length Q, we find the performance of one of its sub-queries of length $Q - 1$ and aggregate this with the performance of the single query term remaining. This is very generous estimation of a query and is likely to over-estimate the performance of a query.

Table 1. Percentile Report (and Standard Deviation) for User Queries

Collection	Topic Range	# Topics	User1	User2	User3	User4
AP	051-200	149	4.0 (4.0)	3.4 (3.5)	3.0 (3.1)	3.3 (3.5)
FBIS	301-450	116	4.8 (7.8)	4.8 (7.8)	5.3 (7.6)	4.8 (7.6)
FT	250-450	188	5.2 (8.6)	5.3 (8.5)	5.7 (8.4)	5.7 (9.3)
WSJ	051-200	150	5.7 (7.8)	4.6 (6.6)	5.0 (7.6)	4.1 (6.0)

We asked human annotators to extract keyword type queries from the *desc* and *narr* fields in a topic (similarly to previous research [4]). This resulted in four sets of short keyword type queries for each topic. Table 1 shows the percentage of queries found in our sampling approach that outperform the actual user queries[2]. The analysis shows that users perform around the fifth percentile of all possible queries for the query extraction task. Another important point to note is that users do not simply extract the same *good* queries. We analysed all possible pairs of user queries (within each topic) and found that 89% of all possible query pairs are unique. This indicates that user queries for the same topic (even when prompted with the actual *desc* and *narr*) are quite varied.

3.2 Correlation of User Extracted Queries

In this section, we report the performance of a number of representative pre-retrieval and post-retrieval performance predictors from the literature[3] on the task of query performance prediction within each topic. We increase the number of queries per topics to five by including the original *desc* field. Although, we have only five queries for each topic, we have a large amount of topics across which to average the correlation coefficients. Furthermore, we know that the five queries for each topic are high performing queries, and we can confirm that for over 75% of the topics the full five queries are unique.

The best pre-retrieval predictors from the literature are the simplified clarity score (scs), the average idf of query terms (idf_{avg}), the maximum idf of the query terms (idf_{max}), the scq score, the normalised scq ($nscq$) score, and the maximum contributing term to the scq score (scq_{max}) [12]. Some of the highest performing post-retrieval predictors from the literature are query clarity ($clarity$), standard deviation at 100 documents (σ_{100}), a normalised version of standard deviation at 100 documents (i.e. the NQC predictor [11]), the maximum standard deviation in the ranked-list (σ_{max}) [10]. We also use two new predictors that calculate the standard deviation using a variable cut-off point ($\sigma_{50\%}$), and a query length normalised version of that predictor ($n(\sigma_{50\%})$) [5].

Table 2 shows the average correlation between the output of each predictor and the performance of the user queries (i.e. those at the high performing end of the query space). Firstly, we can see that the post-retrieval predictors (bottom

[2] We created another sample of the query space by exhaustively evaluating all queries of length one, two and three, and obtained nearly identical statistics.

[3] While we have not included, nor conducted experiments on, an exhaustive list of pre-retrieval and post-retrieval predictors, we have included the highest performing predictors from the literature.

Table 2. Correlations (ρ and r) for User Queries Averaged Over All Topics

Coll.	AP		FBIS		FT		WSJ	
Predictor	r	ρ	r	ρ	r	ρ	r	ρ
scs	0.073	0.086	0.069	0.108	0.133	0.142	0.000	0.035
idf_{avg}	0.063	0.067	0.086	0.109	0.163	0.153	0.035	0.051
idf_{max}	-0.022	-0.040	0.015	0.123	0.107	0.131	0.045	-0.042
scq	-0.033	0.017	-0.06	-0.018	-0.013	0.005	-0.015	0.001
scq_{max}	0.043	0.024	0.024	0.155	0.107	0.131	0.101	0.008
$nscq$	**0.122**	**0.093**	**0.128**	**0.193**	**0.139**	**0.159**	**0.073**	**0.106**
$clarity$	0.118	0.135	0.216	0.259	0.208	0.217	0.103	0.138
σ_{100}	0.185	0.157	0.196	0.237	0.265	0.211	0.263	0.253
σ_{max}	0.134	0.162	0.178	0.227	0.227	0.225	0.173	0.208
NQC	0.115	0.136	0.235	0.283	0.247	0.232	0.267	0.246
$\sigma_{50\%}$	0.250	0.238	0.188	0.211	0.215	0.214	0.271	0.260
$n(\sigma_{50\%})$	**0.368**	**0.328**	**0.280**	**0.340**	**0.255**	**0.269**	**0.405**	**0.398**

Table 3. MAP for Each Set of Topics Using Predictors to Select Queries

Collection	AP	FBIS	FT	WSJ
Avg. Qry per Topic	0.1698	0.2183	0.2294	0.2394
Best **Set** of User Qrys	0.1846	0.2325	0.2482	0.2669
idf_{avg}	0.1759	0.2353	0.2485	0.2371
$nscq$	0.1794	0.2338	0.2336	0.2427
$clarity$	0.1785	0.2311	0.2506†	0.2383
σ_{100}	0.1846 †	0.2463 †	0.2388†	0.2682†
NQC	0.1808	0.2483 †	0.2632†	0.2613†
$\sigma_{50\%}$	0.1881 †	0.2403 †	0.2511†	0.2679 ‡
$n(\sigma_{50\%})$	**0.1940**‡	**0.2523** †	**0.2623**†	**0.2859** ‡

half of the table) outperform the pre-retrieval predictors (top half of the table) for this part of the query space. For example, idf_{max}, a high performing predictor in other studies [6], performs poorly at the high end of the query space. This is because users will often choose the same highly discriminating term when creating a query for the same topic. Therefore, it should be noted that many proposed predictors (especially *pre-retrieval* predictors), may not be able to distinguish between high performing queries. The highest correlated pre- and post-retrieval predictors are outlined in bold.

3.3 Usability of Predictors for Query Selection

We now conduct an experiment to investigate the usefulness of the query performance predictors at selecting the best query among a group of high performing queries[4]. Within each topic, we use each predictor in turn to select the best query (as predicted by the predictor) and then measure the MAP of the set of queries chosen (i.e. the predictor selects one of five queries for each topic). Table 3 shows the performance (MAP) of each predictor for such a task. We deem a predictor to be useful when it consistently[5] outperforms the performance of the best single user. The best predictors tend to be the ones based on standard

[4] Such a scenario may have applications in an collaborative search scenario.

[5] † and ‡ denotes a significant increase over the average and best set of queries respectively, using a Wilcoxon test at the 0.05 level on the topics.

deviations (i.e. NQC, $\sigma_{50\%}$, and $n(\sigma_{50\%})$). Many of these predictors significantly outperform the average query for a topic. Overall, the best predictor for selecting good user queries are the $n(\sigma_{50\%})$ predictor [5]. The predictor can outperform the best single performing set of queries.

4 Conclusion

In this paper, we have shown that user queries lie in the top 5% of queries that a user could extract from an information need. We have shown that post retrieval predictors outperform pre-retrieval for actual user queries. Furthermore, we have shown that post retrieval predictors can be used to effectively choose between high performing queries. This has applications to systems that aim to automatically choose between queries of the same topic (e.g. collaborative IR systems).

References

1. Balasubramanian, N., Kumaran, G., Carvalho, V.R.: Exploring reductions for long web queries. In: SIGIR, pp. 571–578 (2010)
2. Carmel, D., Yom-Tov, E.: Estimating the Query Difficulty for Information Retrieval, 1st edn. Morgan and Claypool Publishers, San Francisco (2010)
3. Cronen-Townsend, S., Zhou, Y., Bruce Croft, W.: Predicting query performance. In: SIGIR, pp. 299–306 (2002)
4. Cummins, R., Lalmas, M., Jose, J.: The limits of retrieval effectiveness. In: Clough, P., Foley, C., Gurrin, C., Jones, G.J.F., Kraaij, W., Lee, H., Mudoch, V. (eds.) ECIR 2011. LNCS, vol. 6611, pp. 277–282. Springer, Heidelberg (2011)
5. Cummins, R., O'Riordan, C., Jose, J.: Improved query performance prediction using standard deviation. In: SIGIR 2011, ACM, New York (2011)
6. Hauff, C., Hiemstra, D., de Jong, F.: A survey of pre-retrieval query performance predictors. In: CIKM 2008, pp. 1419–1420. ACM, New York (2008)
7. He, B., Ounis, I.: Inferring query performance using pre-retrieval predictors. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 43–54. Springer, Heidelberg (2004)
8. Kumaran, G., Allan, J.: Selective user interaction. In: CIKM 2007, pp. 923–926. ACM, New York (2007)
9. Kumaran, G., Carvalho, V.R.: Reducing long queries using query quality predictors. In: SIGIR, pp. 564–571 (2009)
10. Pérez-Iglesias, J., Araujo, L.: Standard deviation as a query hardness estimator. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 207–212. Springer, Heidelberg (2010)
11. Shtok, A., Kurland, O., Carmel, D.: Predicting query performance by query-drift estimation. In: Azzopardi, L., Kazai, G., Robertson, S., Rüger, S., Shokouhi, M., Song, D., Yilmaz, E. (eds.) ICTIR 2009. LNCS, vol. 5766, pp. 305–312. Springer, Heidelberg (2009)
12. Zhao, Y., Scholer, F., Tsegay, Y.: Effective pre-retrieval query performance prediction using similarity and variability evidence. In: Macdonald, C., Ounis, I., Plachouras, V., Ruthven, I., White, R.W. (eds.) ECIR 2008. LNCS, vol. 4956, pp. 52–64. Springer, Heidelberg (2008)

Improved Compressed Indexes for Full-Text Document Retrieval*

Djamal Belazzougui[1] and Gonzalo Navarro[2]

[1] LIAFA, Univ. Paris Diderot - Paris 7, France
dbelaz@liafa.jussieu.fr
[2] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl

Abstract. We give new space/time tradeoffs for compressed indexes that answer document retrieval queries on general sequences. On a collection of D documents of total length n, current approaches require at least $|\mathsf{CSA}| + O(n\frac{\lg D}{\lg\lg D})$ or $2|\mathsf{CSA}| + o(n)$ bits of space, where CSA is a full-text index. Using monotone minimum perfect hash functions, we give new algorithms for document listing with frequencies and top-k document retrieval using just $|\mathsf{CSA}| + O(n \lg\lg\lg D)$ bits. We also improve current solutions that use $2|\mathsf{CSA}| + o(n)$ bits, and consider other problems such as colored range listing, top-k most important documents, and computing arbitrary frequencies.

1 Introduction and Related Work

Full-text document retrieval is the problem of, given a collection of D *documents* (i.e., general sequences over alphabet $[1, \sigma]$), concatenated into a text $T[1, n]$, preprocess T so as to later answer various queries of significance in IR. The problem has received much attention recently [16,22,24,11,8,7,4,12] as a natural evolution over plain full-text indexing (which just counts and locates all the individual occurrences of a pattern $P[1, m]$ in T) and for its applications in IR on Oriental languages, software repositories, and bioinformatic databases. As space is a serious problem in classical solutions [16,11], most of the focus has been on extending compressed full-text indexes to answer various document retrieval queries. The most studied queries, among several others, are the following.

Document listing: List the distinct documents where P appears.
Document listing with frequencies: List the distinct documents where P appears, and the frequency (number of occurrences) of P in each.
Top-k retrieval: List the k documents where P appears most times.

A compressed full-text index [17] is used as the base data structure. This is usually a compressed suffix array of T (we call this structure CSA and its bit space $|\mathsf{CSA}|$). The CSA simulates the suffix array $A[1, n]$ [13], where $A[i]$ points

* Partially funded by Fondecyt Grant 1-110066, Chile. First author also partially supported by the French ANR-2010-COSI-004 MAPPI Project.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 386–397, 2011.
© Springer-Verlag Berlin Heidelberg 2011

to the ith lexicographically smallest suffix in T. The CSA finds the interval $A[sp, ep]$ of occurrences of P in time t_{search}, usually $O(m \lg \sigma)$ or less [9,5]. It can also compute any cell $A[i]$, and even $A^{-1}[i]$, in time $O(t_{\mathsf{SA}})$, usually $O(\lg^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$. These indexes represent the text *and* the suffix array within as little as $nH_h(T) + o(n \lg \sigma)$ bits, for any $h \leq \alpha \lg_\sigma n$ and constant $\alpha < 1$. Here $H_h(T)$ is the empirical h-th order entropy of T [14], a lower bound on the bits-per-symbol a statistical order-h compressor may achieve on T.

In the rest of the section we describe our contributions in context. We introduce at this point the concepts of binary *rank*, *select*, and of a monotone minimum perfect hash function (*mmphf*). Given a bitmap $B[1, n]$ with m bits set, operation $rank_b(B, i)$ counts the number of occurrences of bit $b \in \{0, 1\}$ in $B[1, i]$, whereas $select_b(B, j)$ is the position of the jth occurrence of bit b in B. There exists a representation for B using $\lg \binom{n}{m} + O(\lg \lg m) + o(n) = m \lg \frac{n}{m} + O(m) + o(n)$ bits [21], solving both operations in constant time. As B can be reconstructed using operation $rank$, this space is asymptotically optimal. A mmphf can be seen as a weaker structure on B, able to answer only $rank_1(B, i)$ whenever $B[i] = 1$ and giving an arbitrary value otherwise (the mmphf is unable to tell whether $B[i] = 1$ or 0). As it cannot reconstruct B, the mmphf can be represented within less space than the previous lower bound: within $O(m \lg \lg \frac{n}{m})$ bits it answers the limited $rank$ query in constant time, and using $O(m \lg \lg \lg \frac{n}{m})$ bits it takes time $O(\lg \lg \frac{n}{m})$ [2].

1.1 Document Listing with Frequencies

The pioneering work in this area [16] defines a *document array* $E[1, n]$, where $E[i]$ tells the document to which suffix $A[i]$ belongs. As noted by Sadakane [22], a bitmap $B[1, n]$ marking the document boundaries in T is enough to find $E[i] = rank_1(B, A[i])$ in time $O(t_{\mathsf{SA}})$. The extra space for B is just $D \lg \frac{n}{D} + O(D) + o(n)$ bits [21]. This permits simulating Muthukrishnan's optimal document listing algorithm [16] within time $O(t_{\mathsf{SA}})$ per document reported, in addition to the time t_{search}. The total space is $|\mathsf{CSA}| + O(n)$, the latter coming from range minimum query (RMQ) data structures [6]. The space was made succinct by Hon et al. [11], by sparsifying the RMQ structures over array blocks of size $\lg^\varepsilon n$, so that the time raises to $O(t_{\mathsf{SA}} \lg^\varepsilon n)$ and the space decreases to $|\mathsf{CSA}| + o(n) + D \lg \frac{n}{D} + O(D)$.

We do not innovate on the plain document listing problem, but on the variant that computes frequencies. The solutions build over plain document listing and add extra data structures using two main approaches. A first one stores, in addition to the CSA of the whole collection, one CSA_d for each individual document d, for a total space of $2|\mathsf{CSA}| + O(n)$ [22] or $2|\mathsf{CSA}| + o(n) + D \lg \frac{n}{D} + O(D)$ [11]. This extra $|\mathsf{CSA}|$ space is used to compute document frequencies along the document listing. The times are as for document listing without frequencies.

A second approach [24,8] represents the document array directly, in the form of a wavelet tree [9]. This data structure makes the document listing times independent of t_{SA} and enables algorithms that do not derive from Muthukrishnan's [8], listing each document in $O(\lg D)$ time. The space, however, is at least $n \lg D + o(n)$ (by using a recent encoding of the redundancy [20]).

Gagie et al. [7] abstracted this problem in terms of representing a sequence E providing support for accessing any element of E, enumerating each distinct element in a range of E, and computing sequence $rank_d(E, i)$ (the number of occurrences of document d in $E[1, i]$), so that each document can be listed within the sum of these three times. The abstraction enabled new space/time tradeoffs for document listing with frequencies, achieving times as good as $O(\lg\lg D)$.

An interesting observation of Gagie et al. was that one could use *succinct indexes* over a given sequence representation, for example in order to support the $rank_d$ operation on top of just the B bitmap. These "weaker" representations that need an auxiliary mechanism to compute the cells of E are able to reduce space. For example, they achieved $O(n\frac{\lg D}{\lg\lg D})$ bits with $O(t_{\mathsf{SA}} \lg\lg D)$ time by using a succinct index by Grossi et al. [10]. The very same lower bounds on sequence *rank* given by Grossi et al. show that this tradeoff is optimal.

Our first major contribution improves upon this apparent lower bound. We obtain a succinct index on top of the B bitmap that enables us to carry out document listing with frequencies within *less time and space* than the best previous succinct index. We achieve $O(n \lg\lg D)$ bits extra space and $O(t_{\mathsf{SA}})$ time, or $O(n \lg\lg\lg D)$ bits space and $O(t_{\mathsf{SA}} + \lg\lg D)$ time per reported document.

Our solution is based on mmphfs. As we can solve only a limited case of *rank*, we cannot follow Gagie et al.'s framework [7]. Instead, we simulate Sadakane's method [22] using mmphfs instead of a second CSA. Our space/time results are incomparable with those of Sadakane. Compared to the methods that represent directly the document array, we obtain the least space, while the time comparison depends on t_{SA} (e.g., there are full-text indexes where $t_{\mathsf{SA}} = O(\lg^{\varepsilon} n \lg^{1-\varepsilon} \sigma)$ for any $\varepsilon > 0$, yet they require $O((1 + \frac{1}{\varepsilon})nH_h(T))$ bits [9]).

Actually our solution is general enough to solve the *colored range listing* problem, that is, finding the distinct colors (and their frequencies) of any range in an array $E[1, n]$ of D possible colors. Our solution is the *first* in achieving *optimal time* (i.e., $O(1)$ time per color reported) within *succinct space* (i.e., $n \lg D + n\, o(\lg D)$ bits). Achieving this optimal time involves solving in linear time a particular sorting problem, which can be of independent interest.

Table 1 summarizes our results on this part.

1.2 Top-k Document Retrieval

The pioneering work of Hon et al. [11] uses a sampled suffix tree [1] of $o(n)$ extra bits to reduce this problem to that of accessing $E[i]$ and computing *arbitrary* frequencies (document listing with frequencies turns out to be a simpler problem). They achieve time $O(t_{\mathsf{search}} + k \lg^{4+\varepsilon} n)$ using $2|\mathsf{CSA}| + o(n)$ bits.

Our *second major contribution* is the reduction of their time to $O(t_{\mathsf{search}} + k \lg k \lg^{2+\varepsilon} n)$. First, we show that by choosing better the block sizes one can reduce one $\lg n$ to $\lg k$ (in practice k is much smaller than n, and this improvement applies to many previous solutions). The other $\lg n$ is removed thanks to an improved algorithm to compute arbitrary frequencies, that reduces the time from their $O(t_{\mathsf{SA}} \lg n)$ to $O(t_{\mathsf{SA}} \lg\lg n)$. While both ideas are simple, their impact on performance is large and general.

Table 1. Current and new results on document listing with frequencies (left side) and colored range listing with frequencies (right side). On the left, the extra space is on top of the $|\mathsf{CSA}|$ bits of the full-text index. The time complexities are in addition to t_{search}, and per each of the *ndoc* elements returned. They are valid for any constant $\varepsilon > 0$. On the right we give total space, and total time per each of the *ncol* results reported.

Source	Extra space	Extra Time	Space (colors)	Time (colors)
[16]	$O(n \lg n)$	$O(1)$	$O(n \lg n)$	$O(1)$
[22]	$\|\mathsf{CSA}\| + O(n)$	$O(t_{\mathsf{SA}})$	n/a	n/a
[11]	$\|\mathsf{CSA}\| + o(n)$ $+D \lg \frac{n}{D} + O(D)$	$O(t_{\mathsf{SA}} \lg^{\varepsilon} n)$	n/a	n/a
[8,4]	$n \lg D + o(n)$	$O(\lg \frac{D}{ndoc})$	$n \lg D + o(n)$	$O(\lg \frac{D}{ncol})$
[7]	$n \lg D + O(n)$	$O(\frac{\lg D}{\lg \lg n})$	$n \lg D + O(n)$	$O(\frac{\lg D}{\lg \lg n})$
[7]	$n \lg D + O(n)$	$O(\lg \lg D)$	$n \lg D + O(n)$	$O(\lg \lg D)$
[7]	$O(n \frac{\lg D}{\lg \lg D})$	$O(t_{\mathsf{SA}} \lg \lg D)$	$n \lg D + O(n \frac{\lg D}{\lg \lg D})$	$O(\lg \lg D)$
Ours	$O(n \lg \lg D)$	$O(t_{\mathsf{SA}})$	$n \lg D + O(n \lg \lg D)$	$O(1)$
Ours	$O(n \lg \lg \lg D)$	$O(t_{\mathsf{SA}} + \lg \lg D)$	$n \lg D + O(n \lg \lg \lg D)$	$O(\lg \lg D)$

When representing the document array with support for *rank* operations, arbitrary document counting is easy. Gagie et al. [7], apart from improving the time achieved by Hon et al., gave several new space/time tradeoffs by replacing the second $|\mathsf{CSA}|$-bit space by *rank*-capable representations of E.

Replacing the document array by a weak representation based on mmphfs is not straightforward, as mmphfs do not support general *rank*s. Our *third main contribution* is a technique that modifies Hon et al.'s sampled suffix tree [11] so as to achieve the least space among the methods that represent the document array, while increasing their time by an $O(\lg n)$ factor with respect to the most space-consuming variant. The solution owes in part to the observation that there are not too many candidates around a sampled suffix tree node to replace its precomputed top-k documents. This observation can be useful in other scenarios.

Table 2 summarizes the state of the art and our contribution to the top-k problem. As noted by Hon et al. [11], the bounds apply to the frequency mining problem (list all documents with frequency over f), by running top-k queries with $k = 2^j$ for consecutive j values. Our final contribution is to reduce the time to report the k *most important* documents (i.e., they have a fixed priority) where P appears, from $O(t_{\mathsf{search}} + k \lg^{3+\varepsilon} n)$ [11] to $O(t_{\mathsf{search}} + k \lg k \lg^{1+\varepsilon} n)$.

2 Range Color Listing with Frequencies

We solve the following abstract problem: preprocess an array $E[1,n]$ over D colors so as to answer queries of the form: given i and j, list all the *ncol* distinct colors in $E[i,j]$ and their number of occurrences. The connection with the document listing problem with frequencies is obvious.

Muthukrishnan [16] solved this problem without reporting frequencies. He builds an array $F[1,n]$ where $F[i] = \max\{j < i, E[j] = E[i]\}$. Then, using a data structure that answers RMQ queries on F ($rmq(i,j) = \arg\min_{i \le r \le j} F[r]$)

Table 2. Current and new results on top-k retrieval, using the same conventions of Table 1. The last column assumes $t_{\mathsf{SA}} = O(\lg^{1+\varepsilon} n)$, as in optimal-space CSAs [5].

Source	Extra space	Extra Time	Simplified time
[11]	$\lvert\mathsf{CSA}\rvert + o(n) + D \lg \frac{n}{D} + O(D)$	$O(t_{\mathsf{SA}} \lg^{3+\varepsilon} n)$	$O(\lg^{4+\varepsilon} n)$
[7]	$\lvert\mathsf{CSA}\rvert + o(n) + D \lg \frac{n}{D} + O(D)$	$O(t_{\mathsf{SA}} \lg D \lg(D/k) \lg^{1+\varepsilon} n)$	$O(\lg^{4+\varepsilon} n)$
Ours	$\lvert\mathsf{CSA}\rvert + o(n) + D \lg \frac{n}{D} + O(D)$	$O(t_{\mathsf{SA}} \lg k \lg(D/k) \lg^{\varepsilon} n)$	$O(\lg k \lg^{2+\varepsilon} n)$
[7]	$n \lg D + o(n)$	$O(\lg D \lg(D/k) \lg^{\varepsilon} n)$	$O(\lg^{2+\varepsilon} n)$
[7]	$O(n \frac{\lg D}{\lg\lg D})$	$O(t_{\mathsf{SA}} \lg D \lg(D/k) \lg^{\varepsilon} n)$	$O(\lg^{3+\varepsilon} n)$
Ours	$n \lg D + o(n)$	$O(\lg k \lg(D/k) \lg^{\varepsilon} n)$	$O(\lg k \lg^{1+\varepsilon} n)$
Ours	$O(n \frac{\lg D}{\lg\lg D})$	$O(t_{\mathsf{SA}} \lg k \lg(D/k) \lg^{\varepsilon} n)$	$O(\lg k \lg^{2+\varepsilon} n)$
Ours	$O(n \lg\lg\lg D)$	$O(t_{\mathsf{SA}} \lg k \lg^{1+\varepsilon} n)$	$O(\lg k \lg^{2+\varepsilon} n)$

in constant time (e.g., Fischer's [6] takes $2n + o(n)$ bits and does not access F), he finds the leftmost occurrences of all distinct colors in $E[i, j]$ in time $O(ncol)$.

For computing frequencies, Sadakane [22] finds also the rightmost occurrences of the colors by building another RMQ structure on the array $\overline{F}$ built on the reverse sequence $\overline{E}$. The colors could be reported in different order when listing their rightmost or leftmost occurrences. He does not represent F nor $\overline{F}$, and as a consequence needs to mark the colors found in an array $V[1, D]$. The rest of Sadakane's solution is particular of document retrieval; we instead build on it to obtain an improved solution to the general problem.

Theorem 1. *We can augment a sequence of n colors in $[1, D]$ with a structure using $O(n \lg\lg D)$ bits, so that range color listing with frequencies can be solved in $O(1)$ time per color reported, or using $O(n \lg\lg\lg D)$ bits and $O(\lg\lg D)$ time.*

The theorem assumes $D = O(n)$; otherwise a mapping to the colors actually occurring in the sequence, using $O(n \lg \frac{D}{n}) + o(D)$ bits [21], must be added.

To achieve the result, for each color c we store in a mmphf f_c the positions i such that $E[i] = c$ (i.e., $f_c(i) = rank_c(E, i)$ if $E[i] = c$). Let n_c be the frequency of color c in E, then this structure occupies $\sum_c O(n_c \lg\lg \frac{n}{n_c})$ bits, which by the log-sum inequality is $O(n(\lg H_0(E)+1)) = O(n \lg\lg D)$ bits. The two RMQ data structures will add just $O(n)$ bits. Then a query proceeds in four steps:

1. Use the RMQ on (virtual array) F to get the leftmost occurrences of the $ncol$ colors appearing in the interval. This step takes time $O(ncol)$.
2. Use the RMQ on (virtual array) $\overline{F}$ to get the rightmost occurrences of the $ncol$ colors appearing in the interval. This step also takes time $O(ncol)$.
3. Match the left and right occurrences of the $ncol$ colors. This can be done via sorting, but we show how to do it in time $O(ncol)$.
4. For each color with leftmost and rightmost occurrences l_i and r_i, report the color and its frequency $f_c(r_i) - f_c(l_i) + 1$ in constant time.

To avoid the sorting in step 3, we will slightly modify steps 1 and 2. We will store V and the following additional structures:

1. A vector $R[1, \frac{D}{\lg n}]$, where each cell occupies $\lg D$ bits; R uses at most D bits.
2. A dynamic vector Q storing triplets (c_i, l_i, r_i) and taking $O(ncol \lg n)$ bits.
3. A dynamic vector S storing leftmost positions (c_i, l_i), in $O(ncol \lg n)$ bits.
4. A counter C.

Initially the bits in V and R are set to zero[1], Q and S are empty, and C is set to 1. We then run step 1, setting the bits in V as we progress, and appending the unique colors and their leftmost positions (c_i, l_i) in array S.

We now traverse S and, for each color c_i, compute $g = \lfloor c_i / \lg n \rfloor$. Then, if $R[g] = 0$, we set $R[g] = C$ and $c = rank_1(V[g \lg n + 1, (g+1) \lg n], \lg n)$, which can be computed in constant time in the RAM model [15]. Then we append c copies of the dummy triplet $(\#, \#, \#)$ at the end of vector Q and finally update counter $C = C + c$. At the end of this process array Q will be of size $ncol$ and each distinct color in $E[i, j]$ will have an allocated position into Q.

We now retraverse S and write each pair (c_i, l_i) in the triplet $Q[R[g] + p]$, where $p = rank_1(V[g \lg n + 1, g \lg n + r], r)$, $g = \lfloor c_i / \lg n \rfloor$, and $r = c_i - g \lg n$. So V and R simulate pointers to array Q, where we have already the information on leftmost positions, and now are prepared to write the rightmost positions.

Now we run step 2, but instead of using V to check if we have already reported a color c_i, we compute g and p as before and check whether $Q[R[g] + p] = (c_i, l_i, \#)$. If the third component is a $\#$, then we had not seen the color before and can set the component to r_i. Otherwise we have already seen it.

Now Q has the input to step 4, and step 3 is avoided. Note our working space $O(ndoc \lg n)$ bits of the query is of the same order needed to store the output.

Let us now consider the case where our mmphfs use $O(n_c \lg \lg \lg \frac{n}{n_c})$ bits. By the log-sum inequality these add up to $O(n \lg \lg \lg D)$ bits. The time to query f_c is $O(\lg \lg \frac{n}{n_c})$. To achieve $O(\lg \lg D)$ worst case, we use constant-time mmphfs when $\frac{n}{n_c} > D \lg \lg D$. This implies that on those arrays we spend $O(n_c \lg \lg \frac{n}{n_c}) = O(\frac{n}{D \lg \lg D} \lg \lg D) = O(n/D)$ bits, as it is increasing with n_c and $n_c < \frac{n}{D \lg \lg D}$. Adding over all the possible colors c, we have at most $O(n)$ bits.

By applying the algorithm to document retrieval, where accesses to E are through the CSA, we have the following result.

Theorem 2. *We can augment a* CSA *on* $T[1, n]$ *containing* D *documents with a data structure using* $O(n \lg \lg D)$ *bits, so that document listing with frequencies can be solved in time* $O(t_{\mathsf{SA}})$ *per document reported, or one using* $O(n \lg \lg \lg D)$ *bits and time* $O(t_{\mathsf{SA}} + \lg \lg D)$*. The* $\lg D$ *in the space complexities can be replaced by* $\lg(H) + 1$*, where* $H = \sum \frac{n_d}{n} \lg \frac{n}{n_d}$ *and* n_d *is the length of document* d*.*

3 Faster Top-k Retrieval

In this section we considerably improve the time complexities of Hon et al.'s scheme [11] for top-k retrieval. Their solution partitions the suffix array into

[1] This is done at indexing time. After a query returns the $ncol$ results and sets those $ncol$ bits, we reset them to 0 one by one, leaving V and R ready for the next query.

chunks of $b = k\ell$ bits. A suffix tree [1] on T is built and all the suffix tree nodes that are lowest common ancestors (lca) of consecutive chunk endpoints are represented in a *sampled* suffix tree, which contains $O(n/b)$ nodes. At each sampled node they store the top-k solution of its subtree.

When a pattern is mapped to the suffix array interval $A[sp, ep]$, it is shown that there exists a sampled node covering an area $A[sp', ep']$, where both $sp' - sp$ and $ep - ep'$ are less than b. Therefore one can simply collect the k precomputed candidates and the (at most $2b$) distinct documents mentioned in these remaining intervals, compute their frequencies in $A[sp, ep]$, and take the k highest frequencies. By using y-fast tries [25] on the identifiers and on the frequencies, the process takes time $O(t_{\mathsf{op}} b)$, where $t_{\mathsf{op}} = t_{\mathsf{SA}} + t_{\mathsf{count}} + \lg\lg n$ and t_{count} is the time to count an arbitrary frequency (the $\lg\lg n$ will be absorbed by a $\lg^{\varepsilon} n$ later).

Since k is unknown at indexing time, this structure is built for all k powers of 2 (i.e., $\lg D$ sampled trees), and at query time the next power of 2 is used. By storing the top-k identifiers in increasing order [7] a node uses $O(k \lg(D/k))$ bits, and the total space is $O((n/b)k \lg D \lg(D/k)) = O((n/\ell) \lg D \lg(D/k))$ bits. This allows using $b = k\ell = k \lg D \lg(D/k) \lg^{\varepsilon} n$, which determines the query time.

Something that is not properly considered by Gagie et al. [7] is that if the trees are stored using pointers, then there is a component of $O((n/b) \lg n)$ bits for $k = 1$, and thus ℓ must be at least $\lg^{1+\varepsilon} n$.

To avoid this we store the sampled tree in succinct form [23] using just $2 + o(1)$ bits per node and supporting in $O(1)$ time many operations, including lca, $preorder$ (whose consecutive values are used to index an array storing the top-k candidate data on each node), and $preorder^{-1}$. For each pair of consecutive chunk endpoints p_i and p_{i+1} we store the preorder x_i of the sampled tree node $lca(p_i, p_{i+1})$. As $x_i \geq x_{i-1}$, values $x_i + i$ are increasing, and thus can be stored in a structure of $(n/b) \lg \frac{2n}{n/b} + O(n/b)$ bits that retrieves any x_i in constant time [19][2]. This space is $O((n/b) \lg b) = O(n \frac{\lg k + \lg\lg n}{k \lg D \lg(D/k) \lg^{\varepsilon} n}) = o(n)$. Now we can find in constant time the lowest sampled node covering chunk interval $[L, R]$ as $lca(preorder^{-1}(x_L), preorder^{-1}(x_{R-1}))$. We will omit $preorder^{-1}$ for simplicity.

3.1 Lowering the $\lg D$ Factor to $\lg k$

The fact that we wish to answer queries for any $k \leq D$ translates into a $\lg D$ factor in ℓ, and into the time complexities. If we set a limit k^* on the maximum k allowed at queries, this $\lg D$ becomes $\lg k^*$. We show now that, by carefully choosing ℓ, we can convert the time to $\lg k$.

Instead of choosing $\ell = \lg D \lg(D/k) \lg^{\varepsilon} n$ so that all the sampled suffix trees have the same size, we reduce it to the slightly increasing $\ell = \lg k \lg(D/k) \lg^{\varepsilon} n$. Then the space for a given k is $(n/b)k \lg(D/k) = (n/\ell) \lg(D/k) = \frac{n}{\lg k \lg^{\varepsilon} n}$. Added over all the $k = 2^j$ values this gives $\sum_{j=1}^{\lg D} \frac{n}{j \lg^{\varepsilon} n} = O(\frac{n \lg\lg D}{\lg^{\varepsilon} n}) = o(n)$.

Therefore we obtain times $O(t_{\mathsf{op}} b) = O(t_{\mathsf{op}} k \lg k \lg(D/k) \lg^{\varepsilon} n)$. Note this applies also to previous solutions [7], as shown in Table 2.

[2] Using a constant-time $rank/select$ implementation on their internal bitmap H [15].

3.2 Computing Arbitrary Frequencies

We additionally remove an $O(\lg n)$ factor from Hon et al.'s top-k retrieval query time [11], while using the same asymptotic space. The following theorem states the result building on the improved variant of Gagie et al. [7] and on Section 3.1.

Theorem 3. *Given a concatenation $T[1, n]$ of D documents, the top-k retrieval problem can be solved in time $O(t_{\mathsf{search}} + t_{\mathsf{SA}} k \lg k \lg(D/k) \lg^{\varepsilon} n)$ while using $2|\mathsf{CSA}| + o(n) + D \lg \frac{n}{D} + O(D)$ bits of space, where t_{search} is the time to find the suffix array interval of pattern P in the* CSA *of T, t_{SA} is the time to compute a position of the suffix array or its inverse, and $\varepsilon > 0$ is any constant.*

The theorem is obtained just by noting that time $t_{\mathsf{count}} = O(t_{\mathsf{SA}} \lg n)$ in Hon et al.'s algorithm comes from a binary search for the ep_d such that an interval $[sp_d, ep_d]$ inside a local CSA_d is mapped to a given interval $[sp, ep]$ in the global CSA. This binary search can be sped up by sampling every $\lg^2 n$ positions in CSA_d and storing their corresponding position in the global CSA. This sampled array stores $\lfloor n_d / \lg^2 n \rfloor$ entries and thus takes $O(n_d / \lg n)$ bits of space for each document d of length n_d. The overall space is thus $O(n/ \lg n) = o(n)$.

We store that array of increasing values in a y-fast trie [25] so that a predecessor query takes $O(\lg \lg n)$ time. Then the binary search for ep can be done by first querying the y-fast trie in time $O(\lg \lg n)$, which will delimit an interval of size $\lg^2 n$, and then with a binary search within that interval in time $t_{\mathsf{count}} = O(t_{\mathsf{SA}} \lg \lg n)$. They also need to find sp_d given ep_d, which is similar. With the optimum-space CSA used by Hon et al. [11] this time is $O(\lg^{1+\varepsilon} n)$, and the overall time reduces from $O(\lg^{4+\varepsilon} n)$ per element returned, to $O(\lg k \lg^{2+\varepsilon} n)$.

4 Using Mmphfs for Top-k Retrieval

We now use mmphfs f_c as in Section 2, instead of the local CSA_d's. This would give $t_{\mathsf{count}} = \lg \lg D$ using $O(n \lg \lg \lg D)$ bits. Then the time would be $O((t_{\mathsf{SA}} + \lg \lg D + \lg \lg n) k\ell) = O(t_{\mathsf{SA}} k\ell)$, as the $\lg \lg n$ term is absorbed by the $\lg^{\varepsilon} n$ in ℓ.

The problem is that mmphfs do not give a way to compute arbitrary frequencies. We could only do so if the document appeared *both* in $A[sp, sp' - 1]$ and $A[ep' + 1, ep]$. In such a case we could easily find its leftmost (l_i) and rightmost (r_i) occurrence in $A[sp, ep]$ and compute the frequency as $f_c(r_i) - f_c(l_i) + 1$.

The candidates can be divided into four groups: (1) Appearing only inside $A[sp', ep']$; (2) appearing both in $A[sp, sp' - 1]$ and $A[ep' + 1, ep]$; (3) appearing only in $A[sp, sp' - 1]$; and (4) appearing only in $A[ep' + 1, ep]$.

The only interesting candidates of group (1) are those in the precomputed top-k list, for which we must store the frequencies, as we will have no other way to compute them. This raises the $\lg(D/k)$ time of Section 3 to $\lg n$. Candidates of group (2) are found by scanning both subintervals, finding the documents that appear in both, and their leftmost and rightmost positions. This is easily done in time $O(b \lg \lg n)$ with y-fast tries. Then we compute their frequencies using the corresponding mmphf. How to handle the other two groups is considered next.

4.1 Bounding the Number of Valid Candidates

We show that the number of documents that can make it to the top-k list if they appear only to the left (or, similarly, to the right) chunk of the precomputed interval, is $O(k\sqrt{\ell})$. This allows us to store all those potentially relevant documents within the nodes. By storing their frequency in $A[sp', ep']$, we can complete the frequency computation in $A[sp, ep']$ by just traversing the area $A[sp, sp' - 1]$ and increasing the frequencies of the documents found (we omit this step on documents that have already been found in both tails, as explained).

In order for a document to be out of the top-k list, but able to make it to the list by scanning the chunk to the left of the sampled node, its frequency must be betwen $f - b + 1$ and f, where f is the frequency of the kth most frequent candidate stored. Therefore its frequency can be stored using $O(\lg b) = O(\lg k + \lg\lg n)$ bits. Moreover each document with frequency under $f - \ell + 1$ must appear at least ℓ times in the chunk in order to have a chance, thus there are at most $b/k = \ell$ such nodes. The rest need only $O(\lg \ell)$ bits. Therefore the total space per node will be $O(k \lg n + k \lg b + k\sqrt{\ell} \lg \ell)) = O(k \lg n + k\sqrt{\ell} \lg\lg n)$ (note we are *not* storing the document identifiers of these extra candidates), and the overall space for a given $k = 2^j$ will be $O((n/b)k(\lg n + \sqrt{\ell} \lg\lg n))$. For the sum of spaces over j to be $o(n)$ we need that $\ell = \lg k \lg^{1+\varepsilon} n$ for some $\varepsilon > 0$.

To know which documents are indeed candidates (i.e., can make it to the top-k list so we have stored their frequency inside the node) we set up a bitmap of length b marking the rightmost occurrence of such candidates, and their position in the array of frequencies is obtained with $rank_1$ on that bitmap (a second bitmap distinguishes $\lg b$-bit from $\lg \ell$-bit candidates). As it has at most $k\sqrt{\ell}$ bits set, the bitmap can be stored within $O(k\sqrt{\ell} \lg \sqrt{\ell}) = O(k\sqrt{\ell} \lg\lg n)$ bits. Thus we traverse $A[sp, sp' - 1]$ right to left. When we find a 1 in this bitmap, this is the first time we see a relevant candidate. We compute its identity in $O(t_{\mathsf{SA}})$ time and find its $A[sp', ep']$ frequency using $rank_1$ as explained. Now we have the data to insert it (increasing its frequency by 1) into the y-fast trie. The next occurrences (when the bitmap has value 0) correspond to candidates that either have already been found (and thus are already inserted in the y-fast trie) or candidates that cannot make it to the top-k list (and thus are not present in the y-fast trie and we must not care about them).

The missing piece is to prove that there are sufficiently few candidates.

Lemma 1. *Let $top_k(s, e)$ be k most frequent colors in an array $E[s, e]$. Then there is a choice of $top_k(\cdot, \cdot)$ sets in case of frequency ties such that, for any b, $C(b) = |\cup_{r=0}^{b} top_k(s - r, e)| < k + \sqrt{2bk}$.*

Proof. Let us call $s_t < s$ the position where $k \cdot t$ new elements have made it in top_k at some point, i.e., $C(s - s_t) = C(0) + kt = k + kt$. Let us call f_r the kth highest frequency in $E[r, e]$. Since all elements not in $top_k(s, e)$ have frequency at most $f = f_s$ in $E[s, e]$, a new element must appear at least once in $E[r, s - 1]$ to reach frequency $f + 1$ and force us choose it for $top_k(r, e)$. Hence $s_1 \leq s - k$.

Now, as k distinct elements have entered in $top_k(s_1, e)$, it must hold that $f_{s_1} \geq f + 1$, as we have seen k distinct elements reaching frequency $f + 1$. Thus the $(k + 1)$th *distinct* element appearing in $top_k(r, e)$ must appear at least twice

in $E[r, s-1]$, to jump from frequency at most f to at least $f+2$. Thus we need $2k$ occurrences of elements that are incompatible with the previous k occurrences in order to have k new distinct elements, thus $s_2 \leq s - 3k$.

Once these new k distinct elements enter in $top_k(s_2, e)$, it holds that $f_{s_2} \geq f + 2$, and thus we need $3k$ incompatible occurrences for the next k occurrences, and so on. Iterating the argument, it holds $s_t \leq s - \frac{t(t+1)}{2} k$ for all $t \geq 1$.

Thus as long as $s_t \geq s - b$ we have $\frac{t(t+1)}{2} k \leq b$, and thus $t < \sqrt{2b/k}$. Hence the number of new elements entering into some $top_k(s - r, e)$ for $1 \leq b \leq r$ is $C(b) < k(t+1) < k + \sqrt{2bk}$. □

In our case $b = k\ell$ so the bound is $C(b) = O(k\sqrt{\ell})$. We have proved the main result. The time simplifies to $O(t_{\mathsf{search}} + k \lg k \lg^{2+\varepsilon} n)$ when $t_{\mathsf{SA}} = \lg^{1+\varepsilon} n$.

Theorem 4. *Given a concatenation $T[1, n]$ of D documents, the top-k retrieval problem can be solved in time $O(t_{\mathsf{search}} + t_{\mathsf{SA}} k \lg k \lg^{1+\varepsilon} n)$ using $O(n \lg \lg \lg D)$ extra bits, where t_{search} is the time to find the suffix array interval of pattern P in the* CSA *of T, t_{SA} is the time to compute a position of the suffix array or its inverse, and $\varepsilon > 0$ is any constant.*

5 Top-*k* Most Important Document Retrieval

A particular variant of top-k document retrieval, somewhat easier than the one that seeks for the highest frequencies, is one where the documents have a fixed *importance* or priority. An example would be the PageRank value of Web pages. A way to handle this problem is to sort the documents by importance, so that document i is the ith most important in the collection. Then the problem becomes that of finding the k smallest distinct values in $E[sp, ep]$. While methods based on range quantile queries on wavelet trees [8] naturally report the documents in sorted order and thus automatically solve this problem in $O(k \lg D)$ time by pruning the process after reporting k results, the situation is not that easy for the other approaches that use potentially less space.

A solution comes from the same top-k retrieval technique of Hon et al. [11]. This time one stores the k smallest document values within each sampled node, and traverses the tails of the interval looking for smaller document identifiers. No frequencies need to be computed, which allows for an $O(t_{\mathsf{SA}} k \lg k \lg(D/k) \lg^{\varepsilon} n)$ time solution, e.g., $O(k \lg k \lg^{2+\varepsilon} n)$. This seems unimportant now that we have reduced the complexity of the more difficult top-k retrieval problem to the same level. Yet, we show that this particular problem can be solved faster, removing the $\lg(D/k)$ factor. When $t_{\mathsf{SA}} = \lg^{1+\varepsilon} n$, this gives time $O(t_{\mathsf{search}} + k \lg k \lg^{1+\varepsilon} n)$.

Theorem 5. *Given a concatenation $T[1, n]$ of D documents, the top-k most important retrieval problem can be solved in time $O(t_{\mathsf{search}} + t_{\mathsf{SA}} k \lg k \lg^{\varepsilon} n)$ while using $|\mathsf{CSA}| + o(n) + D \lg \frac{n}{D} + O(D)$ bits of space, where t_{search} is the time to find the suffix array interval of pattern P in the* CSA *of T, t_{SA} is the time to compute a position of the suffix array or its inverse, and $\varepsilon > 0$ is any constant.*

The result of Hon et al. [11] is achieved by using chunks of $b = k\ell$ positions for $\ell = \lg^{2+\varepsilon} n$ (for the more refined complexity we use $\ell = \lg k \lg(D/k) \lg^{\varepsilon} n$). Our idea is to further divide those chunks into $\lg(D/k)$ buckets of size $b' = k \lg k \lg^{\varepsilon} n$.

For each chunk we build a small local sampled suffix tree. A query will then span at most one global node, two local nodes, and two tail buckets.

Consider the endpoints $p_1 \ldots p_r$ of the buckets inside a given chunk, and call $v = lca(p_1, p_r)$ the lowest sampled global suffix tree node that covers the chunk. Just as for the global scheme, find in the suffix tree the lca nodes of each pair of consecutive endpoints, $lca(p_i, p_{i+1})$. All those lca nodes are below v or are v.

There are overall $O(n/b')$ local sampled nodes. Moreover, if some node $u = lca(p_i, p_{i+1})$ covers the whole chunk $[p_1, p_r]$, then it must be an ancestor of $v = lca(p_1, p_r)$, but since it is also a descendant of v, we have $u = v$. That is, the local sampled suffix tree nodes (that are not already global sampled suffix tree nodes) cannot cover a chunk and hence span less than $2b$ positions.

Instead of storing the top-k document identifiers using $O(k \lg(D/k))$ bits, for these local sampled nodes we will store the *positions* of some occurrence of those identifiers within the local sampled node, sorted by increasing position. The identifier must be obtained with an access to that position, which will not change the complexity. Since local positions span less than $2b$, they require $O(k \lg(b/k)) = O(k \lg \ell) = O(k \lg \lg n)$ bits. The tree topology itself will require $2 + o(1)$ bits per node, as for the global tree. The total space for a given $k = 2^j$ is $O((n/b')k \lg \lg n) = O(n \frac{\lg \lg n}{\lg k \lg^{\varepsilon} n})$, which added over all $k = 2^j$ values gives $o(n)$ bits overall. We also must store a local node identifier $y_i = preorder(lca(p_i, p_{i+1}))$ for each bucket, which requires $O((n/b') \lg b) = O(n \frac{\lg k + \lg \lg n}{k \lg k \lg^{\varepsilon} n}) = O(\frac{n \lg \lg n}{k \lg^{\varepsilon} n})$, which added over all $k = 2^j$ values gives $o(n)$ bits as well.

To query, we determine the interval $A[sp, ep]$ of P and the covered chunk $[L, R]$, the covered bucket $[l_1, r_1 = Lb'/b]$ to the left of chunk L, and the covered bucket $[l_2 = Rb'/b, r_2]$ to the right of chunk R. Then we find the global sampled node $v = lca(x_L, x_{R-1})$, and the local sampled nodes $u_1 = lca(y_{l_1}, y_{r_1 - 1})$ and $u_2 = lca(y_{l_2}, y_{r_2 - 1})$. If u_1 or u_2 are equal to v we discard them. Now we take the at most $3k$ candidates from v, u_1 and u_2, and also consider the elements in $E[sp, r_1 b' - 1]$ and $E[b' l_2 + 1, ep]$. The time is $O(t_{\mathsf{SA}}(k + b'))$ to extract all the candidate identifiers, plus $O(k \lg \lg n)$ to maintain a heap of the smallest k values seen in the process using a y-fast trie [25]. The time adds up to $O(t_{\mathsf{SA}} k \lg k \lg^{\varepsilon} n)$.

6 Final Remarks

A natural next step is to implement these solutions. Many of our improvements are easy to implement, and practical implementations of mmphfs exist [3]. A recent empirical work [18] shows that the individual CSA_d's pose much space overhead, at least if implemented naively. Instead, they compress wavelet trees to 7-17 bpc (bits per text character), compared to the 4.5-6.0 bpc of the global CSA. Over their same collections, our mmphf implementation takes 3-5 bpc and gives sub-microsecond times. This shows that the alternative of using mmphfs is very appealing compared to both using CSA_d's or wavelet trees.

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer, Heidelberg (1985)

2. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In: SODA, pp. 785–794 (2009)
3. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Theory and practise of monotone minimal perfect hashing. In: ALENEX (2009)
4. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 194–205. Springer, Heidelberg (2010)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), art. 20 (2007)
6. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
7. Gagie, T., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 67–81. Springer, Heidelberg (2010)
8. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
9. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
10. Grossi, R., Orlandi, A., Raman, R.: Optimal trade-offs for succinct string indexes. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 678–689. Springer, Heidelberg (2010)
11. Hon, W.-K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: FOCS, pp. 713–722 (2009)
12. Karpinski, M., Nekrich, Y.: Top-k color queries for document retrieval. In: SODA, pp. 401–411 (2011)
13. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
14. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
15. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
16. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: SODA, pp. 657–666 (2002)
17. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), art. 2 (2007)
18. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical compressed document retrieval. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 193–205. Springer, Heidelberg (2011)
19. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/ select dictionary. In: ALENEX (2007)
20. Pătraşcu, M.: Succincter. In: FOCS, pp. 305–313 (2008)
21. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA, pp. 233–242 (2002)
22. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discr. Alg. 5(1), 12–22 (2007)
23. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA, pp. 134–149 (2010)
24. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
25. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\theta(n)$. Inf. Process. Lett. 17(2), 81–84 (1983)

ESP-Index: A Compressed Index Based on Edit-Sensitive Parsing

Shirou Maruyama[1], Masaya Nakahara[2],
Naoya Kishiue[2,⋆], and Hiroshi Sakamoto[2,3,⋆⋆]

[1] Kyushu University, 744 Motooka, Nishi-ku, Fukuoka-shi, Fukuoka 819-0395
[2] Kyushu Institute of Technology, 680-4 Kawazu, Iizuka-shi, Fukuoka, 820-8502
[3] PRESTO JST, 4-1-8 Honcho Kawaguchi, Saitama 332-0012, Japan
shiro.maruyama@i.kyushu-u.ac.jp, m_nakahara@donald.ai.kyutech.ac.jp,
kishiue.n@gmail.com, hiroshi@ai.kyutech.ac.jp

Abstract. We propose a compressed self-index based the edit-sensitive parsing (ESP). Given a string S, its ESP tree is equivalent to a context-free grammar deriving just S, which can be represented as a DAG G. Finding pattern P in S is reduced to embedding P into G. Succinct data structures are adopted and G is then decomposed into two LOUDS bit strings and a single array for permutation, requiring $(1+\varepsilon)n\log n + 4n + o(n)$ bits for any $0 < \varepsilon < 1$ where n corresponds to the number of different symbols in the grammar. The time to count the occurrences of P in S is in $O(\frac{\log^* u}{\varepsilon}(m\log n + occ_c(\log m \log u)))$, where $m = |P|$, $u = |S|$, and occ_c is the number of occurrences of a maximal common subtree in ESP trees of P and S. Using an additional array in $n\log u$ bits of space, our index supports locating P and displaying substring of S. Locating time is the same as counting time and displaying time for a substring of length m is $O(m + \log u)$.

1 Introduction

We propose a compressed index based on the *edit-sensitive parsing* (ESP), which was introduced to approximate a variant of string edit distance where a moving operation for any substring with unit cost is permitted. For instance, $a^n b^n$ is transformed to $b^n a^n$ by a single operation. This problem called edit distance *with move* is NP-hard, and the distance was proved to be $O(\log u)$-approximable [9] for strings of length u. Moreover, the harder problem, edit distance *matching* with move, was also proved to be approximable within almost $O(\log u)$ ratio by embedding of string into L_1 vector space using ESP [3].

When we consider *tighter* embedding, i.e. a string is embedded into another one as a substring, this problem becomes the pattern matching. In this work we use ESP to represent a grammar which is transformed to a compressed index based on our theoretical results for efficient pattern matching and data structures

⋆ He moved to Hitachi Solutions, Ltd.
⋆⋆ This work was partially supported by JST PRESTO program.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 398–409, 2011.
© Springer-Verlag Berlin Heidelberg 2011

on ESP. The following outlines the proposed data structures and its operations. We first preprocess the text S and build ESP tree T_S. Given a pattern P, we construct T_P and decomposes it into a sequence of subtrees, whose roots are labeled by $x_1, \ldots, x_k$. This sequence is an *evidence* of occurrence of P in T: S contains an occurrence of P iff there is a sequence $v_1, \ldots, v_k$ of nodes in T_S such that the label of v_i is x_i and the subtrees rooted by v_i, v_{i+1} are *adjacent* in this order. We should note that P itself is always evidence of P. We design algorithms to extract as short evidence as possible by analysis of ESP, and to embed the evidence in T_S.

Another contribution is to develop compact data structures for the proposed algorithms. An ESP is represented by a restricted CFG, and is equivalent to a DAG G where every internal node has its left and right children. G is then decomposed into two *in-branching* spanning trees. The one called the left tree is constructed by the left edges, whereas the other, called the right tree, is constructed by the right edges. Both the left and right trees are encoded by LOUDS [4], one of the succinct data structures for ordered trees. Further, correspondence among the nodes of the trees is stored in a single array. Adding the data structure for the permutation [5] over the array makes it possible to traverse G. The size of such data structures is at most $(1+\varepsilon)n \log n + 4n + o(n)$ bits of space for arbitrary $0 < \varepsilon < 1$, where n is the number of variables in G.

On the other hand, the compression algorithm should refer to a function, called *reverse dictionary* to get a name of the variable associated with a digram. For example, if a production rule $Z \to XY$ exists, any occurrence of the digram XY in P, which is determined to be replaced, should be replaced by the same Z. Taking up the hash function $H(XY) = Z$ for preprocessing P compels that index size be increased. Thus our algorithm obtains the names of variables for P directly from the compressed G using binary search.

The time to count all occurrences of a pattern P in S is $O(\frac{\log^* u}{\varepsilon}(m \log n + occ_c(\log m \log u)))$ time, where $m = |P|$, $u = |S|$, and occ_c is the number of occurrences of a *core* in T_S, which is the maximal common subtrees of T_S and T_P. The other two operations to locate/display are supported by an additional array requiring $n \log u$ bits of space to store the length of the substring encoded by each variable. This array can be reduced by level-wise sampling because the ESP tree is *balanced*. The time to display $S[i,j]$ is $O(j - i + \log u)$.

We compare the performance of ESP-index with other practical self-indexes [6–8] under several reasonable parameters. Beyond that, we give an experimental result for maximal common substring detection. These common substrings are obtained to find common variables in compressed T_S and T_P. In these experiments, we conclude that the proposed index is efficient enough for cases where the pattern is long.

2 Pattern Matching on ESP

The set of all strings over an alphabet Σ is denoted by Σ^*. The length of a string $w \in \Sigma^*$ is denoted by $|w|$. A string a^k $(k \geq 1)$ is also denoted by a^+, and is called

a repetition, denoted by a^{++}, if $k \geq 2$. $S[i]$ and $S[i,j]$ denote the i-th symbol of S and the substring from $S[i]$ to $S[j]$, respectively. We let $\log^{(1)} u = \log u$, $\log^{(i+1)} u = \log \log^{(i)} u$, and $\log^* u = \min\{i \mid \log^{(i)} u \leq 1\}$, i.e. $\log^* u \leq 5$ for $u \leq 2^{65536}$. Thus We can treat any $\log^* u$ as constant in a practical sense.

We assume that any context-free grammar G is *admissible*, i.e., G derives just one string and for each variable X, exactly one production rule $X \to \alpha$ exists. The set of variables is denoted by $V(G)$, and the set of production rules, called dictionary, is denoted by $D(G)$. We also assume that for any $\alpha \in (\Sigma \cup V(G))^*$ at most one $X \to \alpha \in D(G)$ exists. We use V and D instead of $V(G)$ and $D(G)$ when G is omissible. The string derived by D from a string $S \in (\Sigma \cup V)^*$ is denoted by $S(D)$. For example, when $S = aYY$ and $D = \{X \to bc, Y \to Xa\}$, we obtain $S(D) = abcabca$.

2.1 Edit-Sensitive Parsing (ESP)

We start with the outline of ESP. For any string, it is uniquely partitioned to $w_1 a_1^{++} w_2 a_2^{++} \cdots w_k a_k^{++} w_{k+1}$ by maximal repetitions, where each a_i is a symbol and w_i is a string containing no repetition. Each a_i^{++} is called Type1 metablock, w_i is called Type2 metablock if $|w_i| \geq \log^* n$, and other short w_i is called Type3 metablock, where if $|w_i| = 1$, this is attached to a_{i-1}^{++} or a_i^{++}, with preference a_{i-1}^{++} when both are possible. Any metablock is longer than or equal to two.

Let S be a metablock and D be a current dictionary starting with $D = \emptyset$. We set $ESP(S, D) = (S', D \cup D')$ for $S'(D') = S$ and S' described as follows:

1. In case S is Type1 or Type3 of length $k \geq 2$,
   (a) If k is even, let $S' = t_1 t_2 \cdots t_{k/2}$, and make $t_i \to S[2i-1, 2i] \in D'$.
   (b) If k is odd, let $S' = t_1 t_2 \cdots t_{(k-3)/2}\, t$, and make $t_i \to S[2i-1, 2i] \in D'$, $t \to S[k-2]t'$, and $t' \to S[k-1, k] \in D'$, where t_0 denotes the empty string for $k = 3$.
2. In case S is Type2,
   (c) for the partitioned $S = s_1 s_2 \cdots s_k$ $(2 \leq |s_i| \leq 3)$ by *alphabet reduction*, let $S' = t_1 t_2 \cdots t_k$, and make $t_i \to XY \in D'$ if $s_i = XY$ and make $t_i \to XY', Y' \to YZ \in D'$ if $s_i = XYZ$.

Case (a) and (b) denote a typical *left aligned parsing*. For example, in case $S = a^6$, $S' = x^3$ and $x \to a^2 \in D'$, and in case $S = a^9$, $S' = x^3 y$ and $x \to a^2, y \to ay', y' \to aa \in D'$. In Case (c), we omit the description of alphabet reduction [3] because the details are unnecessary in this paper.

Finally, we define ESP for any string $S \in (\Sigma \cup V)^*$ that is partitioned to $S_1 S_2 \cdots S_k$ by k metablocks; $ESP(S, D) = (S', D \cup D') = (S'_1 \cdots S'_k, D \cup D')$, where D' and each S'_i satisfying $S'_i(D') = S_i$ are defined in the above.

Iteration of ESP is defined by $ESP^i(S, D) = ESP^{i-1}(ESP(S, D))$. In particular, $ESP^*(S, D)$ denotes the iterations of ESP until $|S| = 1$. After computing $ESP^*(S, D)$, the final dictionary represents a rooted ordered binary tree deriving S, which is denoted by $ET(S)$. We refer to several characteristics of ESP, which are bases of our study.

Lemma 1. (Cormode and Muthukrishnan [3]) The height of $ET(S)$ is $O(\log|S|)$ and $ET(S)$ can be computed in time $O(|S|\log^*|S|)$ time.

Lemma 2. (Cormode and Muthukrishnan [3]) Let $S = s_1 s_2 \cdots s_k$ be the partition of a Type2 metablock S by alphabet reduction. For any $1 \leq j \leq |S|$, the block s_i containing $S[j]$ is determined by at most $S[j - \log^*|S| - 5, j + 5]$.

2.2 Pattern Embedding Problem

We focus on the problem to find occurrences of P by embedding of P into a parsing tree. Given a parsing tree $T_S = ET(S)$ by D_S and a pattern P, the key idea is to compute $ESP(P, D_S)$ and find an embedding of resulting tree T_P into T_S. The label of node v is denoted by $L(v) \in \Sigma \cup V$, and $L(v_1 \cdots v_k) = L(v_1) \cdots L(v_k)$. Let $yield(v)$ denote a substring of S derived by $L(v)$, and $yield(v_1 \cdots v_k) = yield(v_1) \cdots yield(v_k)$. We note that T_S and T_P are ordered binary trees.

Let us define some notations for ordered binary tree. The parent and left/right child of node v are denoted by $parent(v)$ and $left(v)/right(v)$, respectively. For an internal node v, edges $(v, left(v))$ and $(v, right(v))$ are called left edge and right edge. Node v is called the *lowest right ancestor* of x, denoted by $lra(x)$, if v is the lowest ancestor satisfying that the path from v to x contains at least one left edge. If x is a rightmost descendant, $lra(x)$ is undefined. Otherwise, $lra(x)$ uniquely exists. The lowest left ancestor of x, denoted by $lla(x)$, is similarly defined. Let v_1, v_2 be different nodes. If $lra(v_1) = lla(v_2)$, we call that v_1, v_2 are *adjacent* in this order, and we also call that v_1 is left adjacent to v_2 (or v_2 is right adjacent to v_1). We can derive the following characterization immediately.

fact 1. *v_1 is left adjacent to v_2 iff v_2 is a leftmost descendant of $right(lra(v_1))$, and v_2 is right adjacent to v_1 iff v_1 is a rightmost descendant of $left(lla(v_2))$.*

Using this adjacency, we define an embedding of sequence of nodes, $v_1, \ldots, v_k$ ($k \geq 2$), as follows: if v_i, v_{i+1} are adjacent in this order ($1 \leq i \leq k-1$) and z is the lowest common ancestor of v_1 and v_k denoted by $z = lca(v_1, v_k)$, we state that the sequence is embedded in z and it is denoted by $(v_1, \ldots, v_k) \prec z$. For such z, if $yield(v_1 \cdots v_k) = P$, z is called an *occurrence node of* P. We introduce a special type of strings to guarantee occurrences of P in S.

Definition 1. A string $Q \in (\Sigma \cup V)^*$ of length k satisfying the following condition is called an *evidence of* P: node z in T_S is an occurrence node of P iff there is a sequence $v_1, \ldots, v_k$ such that $(v_1, \ldots, v_k) \prec z$, $yield(v_1 \cdots v_k) = P$, and $L(v_1 \cdots v_k) = Q$.

For any T_S and P, at least one evidence of P exists because P itself is an evidence of P. We propose an algorithm to find as short evidence as possible for given T_S and P; we also propose another algorithm to find all occurrences of P in S using obtained evidence.

```
Find_evidence(P, D_S)
let D' ← ∅, Q_p = Q_s be the empty string;
while(|P| > 1){ /* appending prefix and suffix of P to Q */
    let P = αβγ for the first/last metablock α/γ;     /* possibly |βγ| = 0 */
    (P', D_S ∪ D') ← ESP(P, D_S), where
        P' = α'β'γ', α'(D') = α, β'(D') = β, γ'(D') = γ;

    if(α is Type1 or 3) {Q_p ← Q_p α, remove the prefix α' of P';}
    else{
        let α = α_1 ··· α_ℓ, α' = p_1 ··· p_ℓ, where p_i → α_i ∈ D';
        Q_p ← Q_p α_1 ··· α_j, remove the prefix p_1 ··· p_j of P' for j = min(log* u + 5, ℓ);
    }/* the bound j for prefix is guaranteed by Lemma 2 */

    if(γ is Type1 or 3) {Q_s ← γ Q_s, remove the suffix γ' of P';}
    else{
        let γ = γ_1 ··· γ_r, γ' = q_1 ··· q_r, where q_i → γ_i ∈ D';
        Q_s ← γ_{r−j} ··· γ_r Q_s, remove the suffix q_{r−j} ··· q_r of P' for j = min(5, r);
    }/* the bound j for suffix is also from Lemma 2 */
    P ← P', D_S ← D_S ∪ D', D' ← ∅;     /* update */
} if(|Q_p Q_s| > 0) return Q ← Q_p Q_s; else return P;
```

Fig. 1. Algorithm to find evidence of pattern P using dictionary D_S from $ESP^*(S, D)$

3 Algorithms and Data Structures

We propose two algorithms: one generates an evidence Q of pattern P from preprocessed $ESP^*(S, D) = (S', D_S)$, and the other algorithm finds the occurrence node z of P such that $(v, v_1 \ldots, v_k) \prec z$ for given Q and v in T_S satisfying $L(v) = Q[1]$. Finally, we propose data structures to access the next node v' satisfying $L(v') = L(v)$ for each v in T_S. By this, all occurrences of P are found to check if $(v, v_1 \ldots, v_k) \prec z$ for all candidates v satisfying $L(v) = Q[1]$.

3.1 Finding Evidence of Pattern

The algorithm to generate evidence Q of pattern P is described in Fig. 1. An outline follows. Input is a pair of pattern P and final dictionary D_S from $ESP^*(S, D)$. P is partitioned to $P = \alpha\beta\gamma$ for the first metablock α and the last metablock γ. Depending on the type of metablocks, P is further partitioned to $P = \alpha_p\alpha_s\beta\gamma_p\gamma_s$. The algorithm then updates current $Q = Q_pQ_s$ by $Q_p \leftarrow Q_p\alpha_p$ and $Q_s \leftarrow \gamma_sQ_s$, and $P \leftarrow P'$ such that P' is the string produced by $ESP(\alpha_s\beta\gamma_p, D_S)$. This is continued until P is entirely deleted.

Lemma 3. Let Q be an output string of *Find_evidence*(P, D_S) and let $Q = Q_1 \cdots Q_k$, $Q_i \in \{q_i^+\}$ for some symbol q_i and $q_i \neq q_{i-1}, q_{i+1}$. Then Q is an evidence of P satisfying $k = O(\log m \log^* u)$ for $m = |P|$.

Proof. For $P = \alpha\beta\gamma$ by the first/last metablock α/γ, if α, γ are Type1 or 3, $\alpha\beta\gamma$ is clearly an evidence of P. In this case, any occurrence of β inside $S[n, m] = \alpha\beta\gamma$ is transformed to a same β' in this while loop. Thus, $\alpha\beta'\gamma$ is an evidence of P.

If α, γ are Type2, by alphabet reduction, the prefix α of P is partitioned to $\alpha = \alpha_1 \cdots \alpha_\ell$ $(2 \leq |\alpha_i| \leq 3)$. Then $j = \min(\log^* u + 5, \ell)$ is determined and $\alpha_1 \cdots \alpha_j$ is appended to current Q, and a short suffix of γ is similarly appended to Q. By Lemma 2, for any $S = x\beta y$ $(|x| \geq \log^* u + 5, |y| \geq 5)$, any occurrence of β inside $S[n, m] = x\beta y$ is transformed to a same β' in this while loop. The selected j for α satisfies either $|\alpha_1 \cdots \alpha_j| \geq \log^* u + 5$ or $\alpha_1 \cdots \alpha_j = \alpha$ and the selected j for β similarly satisfies either $|\gamma_{r-j} \cdots \gamma_r| \geq 3$ or $\gamma_{r-j} \cdots \gamma_r = \gamma$. Thus we can obtain an evidence $\alpha_1 \cdots \alpha_j \beta' \gamma_{r-j} \cdots \gamma_r$ of P, and the other cases, one of α, γ is Type1 or 3 and the other is Type2, are similarly proved.

Applying the above analysis to β' until its length becomes one, we can finally obtain Q as an evidence of P. The number of iterations of $ESP(P, D) = (P', D \cup D')$ is $O(\log m)$ because $|P'| \leq |P|/2$. In i-th iteration, if current $Q = Q_p Q_s$ is updated to $Q_p \alpha_p \gamma_s Q_s$, $\alpha_p \gamma_s$ contains $O(\log^* u)$ different symbols. Therefore we conclude $k = O(\log m \log^* u)$. □

3.2 Finding Pattern Occurrence

The algorithm to find an occurrence node of P is described in Fig. 2. Using *Find_evidence*(P, D_S) as subroutine, the algorithm *Find_pattern*(Q, D_S, v, T_S) finds an embedding $(v, v_1, \ldots, v_\ell) \prec z$ satisfying $yield(vv_1 \cdots v_\ell) = P$ for fixed v having the label $Q[1]$. By Lemma 3, such z exists iff z is an occurrence node of P. We show the correctness of this algorithm and the time complexity.

Lemma 4. *Find_pattern*(Q, D_S, v, T_S) *outputs node* z *in* T_S *iff* z *is an occurrence node of* P *satisfying* $(v, v_1, \ldots, v_\ell) \prec z$ *for some* $v_1, \ldots, v_\ell$ *and fixed* v *in* T_S*. The time complexity is* $O(\log m \log u \log^* u)$ *for* $m = |P|$ *and* $u = |S|$*.*

Proof. We outline the proof. For any node v in T_S and $q \in \Sigma \cup V$, we can check if $(v, v') \prec z$ and $L(v') = q$ for some nodes v', z in $O(\log u)$ time because such v' must be a leftmost descendant of $right(lra(v))$ and the height of T_S is $O(\log u)$.

Let $Q = Q_1 \cdots Q_k$ and $Q_i \in \{q_i^{++}\}$ for some $q_i \in \Sigma \cup V$. If Q contains no repetition and we assume that $(v_1, \ldots, v_j) \prec z$ is found for $Q_1 \cdots Q_j = q_1 \cdots q_j$. From $(v_j, v_{j+1}) \prec z'$ and $L(v_{j+1}) = q_{j+1}$, we obtain $(v_1, \ldots, v_{j+1}) \prec lca(z, z')$ in $O(\log u)$ time because z, z' must be in a same path. Thus an embedding of length at most $O(\log m \log^* u)$ from v is computed in $O(\log m \log u \log^* u)$ time.

Let $Q_j = q^\ell$ for some symbol q and $\ell \geq 2$. In ESP, any repetition is transformed to a shorter string by the left aligned parsing, and this transformation is continued as long as the resulting string contains a repetition. Thus, by T_S, an occurrence $S[s, t] = q^\ell$ is partitioned to $S[s, t] = S[s_1, t_1] S[s_2, t_2] \cdots S[s_k, t_k]$ such that $|S[s_i, t_i]| = 2^{\ell_i} \geq 1$, v_i in T_S is the root of the complete binary tree deriving $S[s_i, t_i]$, and $k = O(\log \ell)$. Let $S[s_i, t_i]$ be the longest segment. We note that all symbols in current string are replaced by the next iteration of ESP. By this characteristic, when $S[s_i, t_i]$ is transformed to $S'[j]$, the adjacent digram

```
Find_pattern(Q, D_S, v, T_S) /* L(v) = Q[1] */
let Q = Q_1 ··· Q_k, Q_i ∈ {q_i^+}, q_i ∈ Σ ∪ V, q_i ≠ q_{i−1}, q_{i+1};
initialize j ← 1, z ← v;      /* current block Q_j and embedding in z */
if(|Q| = 1) return z;
while(j ≤ k){
    if(|Q_j| = 1){ /* block Q_j is just one symbol */
        if((v, v') ≺ z', L(v') = q_{j+1} for some v', z' in T_S)
            v ← v', z ← lca(z, z'), j ← j + 1;
        else return 0;
    }
    else{ /* block Q_j is a repetition */
        ℓ ← |Q_j|;
        while(ℓ > 0){ /* find maximal complete binary tree parsing q_j^{++} */
            if((v, v') ≺ z', L(v') = q_j for some v', z' in T_S){
                let v_a be the highest ancestor of v' satisfying X_0 = L(v_a), X_d = q_j,
                    X_0 → X_1^2, ..., X_{d−1} → X_d^2 ∈ D_S, 1 ≤ 2^d ≤ ℓ;
                v ← v_a, z ← lca(z, z'), ℓ ← ℓ − 2^d;
            }/* next complete binary tree until whole q_j^{++} is covered */
            else return 0;
        }j ← j + 1;
    }
}return z;
```

Fig. 2. Algorithm to find occurrence node of P starting with a given node v

$S'[j-2, j-1]$ derives a string containing $S[s_1, t_1] \cdots S[s_{i-1}, t_{i-1}]$ as its suffix. Thus, we can check if $(v_1, \ldots, v_{i-1}) \prec v_p$ for some v_p in $O(\log \ell + \log u) = O(\log m + \log u)$ time. The time to check if $(v_p, v_i) \prec z$ is $O(\log u)$. Hence, the time to embed q^ℓ is $O(\log m + \log u)$. Therefore, we find the occurrence of P in $O(\log m \log^* u(\log m + \log u)) = O(\log m \log u \log^* u)$ time. □

3.3 Data Structures

We develop compact data structures for *Find_pattern*(Q, D_S, v, T_S) to access a next occurrence of v satisfying $L(v) = q$ for any $q \in \Sigma \cup V$. These improvements are achieved by two techniques: one is decomposition of DAG into *left tree and right tree*; the other is simulation of the *reverse dictionary* for pattern compression. First we treat decomposition of DAG G, which is a graph representation of D_S. Introducing a super sink v_0 together with left and right edges from any sink of G to v_0, G can be modified to have the unique source/sink.

fact 2. *Let G be a DAG with single source/sink such that any node except the sink has exactly two children. For any in-branching spanning tree of G, the graph defined by the remaining edges is also an in-branching spanning tree of G.*

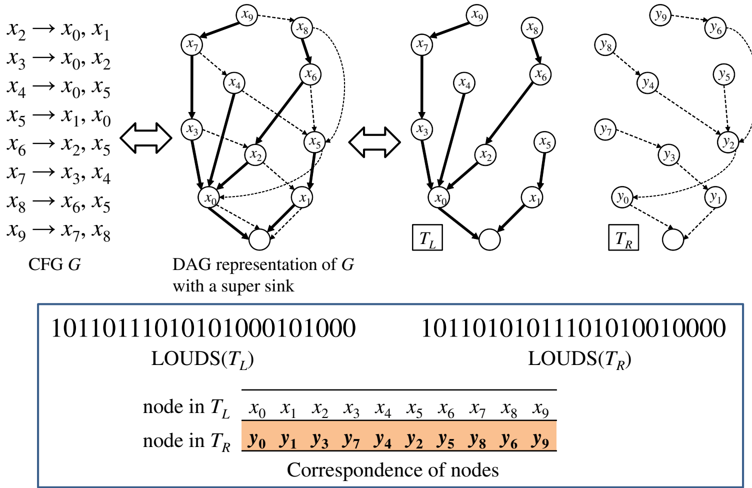


Fig. 3. Succinct representation of a CFG by left tree and right tree in LOUDS bit-string with a permutation array

In-branching spanning tree of G constructed by left edges only is called *left tree* of G and denoted by T_L. Thus the complementary tree is called *right tree* of G and denoted by T_R. An example of G and its left/right tree is shown in Fig. 3 with its succinct representation proposed hereafter.

When a DAG is decomposed into T_L and T_R, the two are represented by succinct data structures for ordered trees and permutations. The bit-string by LOUDS in [4] for an ordered tree is defined as shown below. We visit any node in level-order from the root. As we visit a node v with $d \geq 0$ children, we append 1^d0 to the bit-string beginning with the empty string. Finally, we add 10 as the prefix corresponding to an imaginary root, which is the parent of the root of the tree. For the n-node tree, LOUDS uses $2n + o(n)$ bits to support constant time access to the parent, the i-th child, and the number of children of a node.

To traverse G equivalent to T_S, we also need the correspondence of nodes in one tree to the other. For this purpose, we employ the succinct data structure for permutation in [5]. For a given permutation π of $N = (0, \ldots, n-1)$, using $(1+\varepsilon)n \log n + o(n)$ bits of space, this data structure supports access to $\pi[i]$ in $O(1)$ time and $\pi^{-1}[i]$ in $O(1/\varepsilon)$ time. For instance, if $\pi = (2, 3, 0, 4, 1)$, then $\pi[2] = 0$ and $\pi^{-1}[4] = 3$; that is, $\pi[i]$ is the i-th member of π and $\pi^{-1}[i]$ is the position of the member i. For each node i in $LOUDS(T_L)$ and the corresponding node j in $LOUDS(T_R)$, we can get the relation by $\pi[i] = j$ and $\pi^{-1}[j] = i$.

We introduce another preprocess for G. For each iteration $ESP(S, D) = (S', D \cup D')$, we rename new variables in D' and S' by sorting[1] all production rules $X \to X_i X_j \in D'$ by (i, j): If the rank of $X \to X_i X_j$ is k, all occurrences of X in D' and S' are renamed to X_k. For example, $D' = \{X_1 \to ab,$

[1] A similar technique was proposed in [2] but variables are sorted by encoded strings.

$X_2 \to bc, X_3 \to ac, X_4 \to aX_2\}$ and $S' = X_1X_2X_3X_4$ are renamed to $D' = \{X_1 \to ab, X_2 \to ac, X_3 \to aX_4, X_4 \to bc\}$ and $S' = X_1X_4X_2X_3$. Thus, variable X_i in G coincides with node i in LOUDS of T_L because they are both named in level-order. The G in Fig. 3 is already renamed. By this improvement, the size of the array required for node correspondence is reduced to $n \log n$ bits of space.

Finally we simulate the reverse dictionary using G for pattern compression. When $ESP^*(S, D)$ is computed, the naming function $H_S(XY) = Z$ defined by $Z \to XY \in D$ is realized by a hash function. However, because our index does not contain this data structure, we must simulate H_S by only G to obtain an evidence of P. By preprocessing, variable X_k corresponds to the rank of its left hand side X_iX_j for $X_k \to X_iX_j$. Conversely, given X_i, the children of X_i in T_L are already sorted by the ranks of their parents in T_R. Because LOUDS supports referring to the number of children and i-th child, $H_S(X_iX_j) = X_k$ is obtained by binary search in the following time complexity.

Lemma 5. The function $H_S(XY) = Z$ is computable in $O(\frac{1}{\varepsilon}\log k) = O(\frac{1}{\varepsilon}\log n)$ time, where k is the maximum degree and n is the number of nodes of T_L.

Theorem 1. The size of ESP-index for string S is $(1+\varepsilon)n\log n + 4n + o(n)$ bits of space, where n is the number of variables in T_S. With pattern P, the number of its occurrence in S is computable in $O(\frac{\log^* u}{\varepsilon}(m \log n + occ_c(\log m \log u)))$ time for any $0 < \varepsilon < 1$, where $u = |S|$, $m = |P|$, and occ_c is the number of occurrences of a maximal common subtree in T_S and T_P.

Proof. We can modify *Find_pattern* to find $(v_1, \ldots, v_k) \prec z$ from v_k to v_1. Thus, starting with v_ℓ labeled by q which encodes a longest string, we can find z by $(v_1, \ldots, v_\ell) \prec z_1$, $(v_\ell, \ldots, v_k) \prec z_2$, and $(v_1, \ldots, v_k) \prec lca(z_1, z_2) = z$. This derives the time bound. □

Locating and displaying are realized by an additional array to store the length of each variable. Since ESP tree is balanced, we obtain the bound below.

Corollary 1. With additional $n \log u + o(n)$ bits of space, ESP-index supports locating P and displaying $S[i, j]$. The time to locate is the same as the case of counting, and the time to display a substring of length m is $O(m + \log u)$.

4 Experiments

The environment is OS:CentOS 5.5 (64-bit), CPU:Intel Xeon E5504 2.0GHz (Quad)×2, Memory:144GB RAM, HDD:140GB, and Compiler:gcc 4.1.2. Datasets of English texts and DNA sequences of 200MB each are obtained from the text collection in Pizza&Chili Corpus.[2]

We first show how a long string is encoded by evidence of pattern in Fig. 4. This figure shows the maximum length of a string encoded by a symbol in evidence Q according to the pattern length. We call this symbol in Q a *core*. By

[2] http://pizzachili.dcc.uchile.cl/texts.html

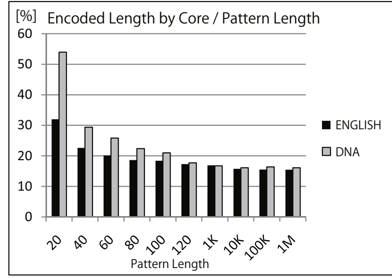


Fig. 4. Encoded String Length by Core / Pattern Length

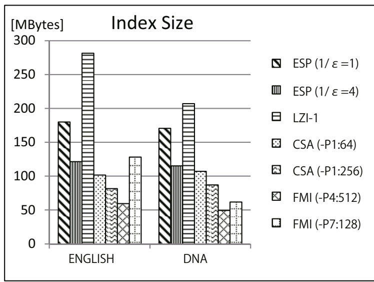


Fig. 5. Index Size

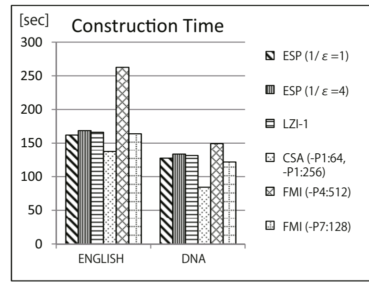


Fig. 6. Construction Time

this result, sufficiently long common substrings in S_1 and S_2 are extracted as common variables in $ET(S_1)$ and $ET(S_2)$.

We next compare our ESP-index with other compressed indexes referred to as LZ-index (LZI)[3], Compressed Suffix Array, and FM-index (CSA and FMI)[4]. These implementations are based on [6–8]. Due to the trade-off between construction time and index size, the indexes referred to above are examined with respect to reasonable parameters.

For ESP-index, we set $\varepsilon = 1, 1/4$ for permutation. In CSA, the option (-P1:L) means that the ψ function is encoded by the gamma function and L specifies the block size for storing ψ. In FMI, (-P4:L) means that BW-text is represented by a Huffman-shaped wavelet tree with compressed bit-vectors and L specifies the sampling rate for storing rank values; (-P7:L) is the uncompressed version. In addition, these CSA and FMI do not make indexes for occurrence position.

The result of index size is shown in Fig. 5, where the space for locating is removed in all indexes except LZI; total index size including the space for locating/displaying is shown in the last two tables. Fig. 5 reveals that ESP-index is compact enough and comparable to CSA(-P1:64). The result of construction time is shown in Fig. 6. It is deduced from this result that ESP-index is

[3] http://pizzachili.dcc.uchile.cl/indexes/LZ-index/LZ-index-1

[4] http://code.google.com/p/csalib/

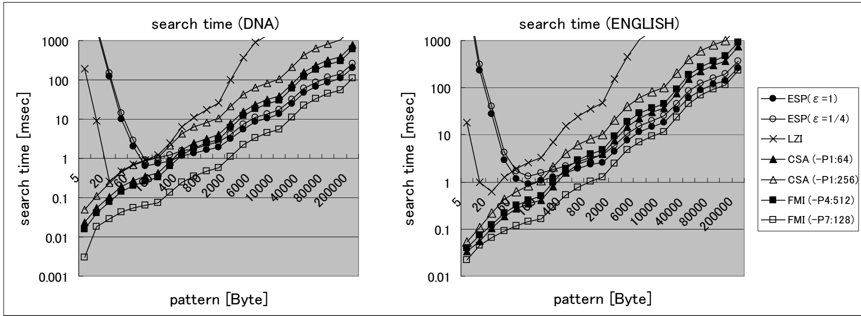


Fig. 7. Counting Time

comparable with FMI and CSA in the parameters in construction time, and slower than LZI. Further, a conspicuous difference is not seen in construction time.

Fig. 7 shows the time to count the occurrences of pattern for both types of English and DNA. Random[5] selection of pattern from the text was made 1000 times for each fixed pattern length; the search time indicates the average time. In this implementation, we modified our search algorithm so that the core q is extracted by preprocessing a short prefix of P. And an occurrence of P in S is examined by finding q and the exact match of the remaining substrings by partial decoding of the compressed S. By preliminary experiments, we determine the length of preprocessed prefix to be 1% of $u = |P|$ in practice. In DNA and ENGLISH, our method is faster than LZI and CSA in case of long patterns.

Finally, we show the results of locating and displaying in Table 1 and Table 2 respectively. Locating is achieved with an additional array to store the length of the encoded string for each variable. Because ESP tree is balanced, the size of this array is reduced by level-wise sampling; in this experiment, the array is developed only for variables produced in odd level of ESP. In CSA/FMI, option -I:{D}:{D2} indicates sampling parameter D for suffix array and D2 for the reverse array. We get better results for $m = 100$ than for $m = 1000$. This is because the frequency of longer pattern becomes one and the search time is thus proportional to m. For comparison of theoretical time/space bound, see e.g. [2].

5 Discussion

We have another motivation to apply our data structures to practical use. Originally, ESP was proposed to solve a difficult variant of the edit distance by finding the maximal common substrings of two strings. Thus, our method will exhibit its ability if both strings are sufficiently long. Such situations are found in the framework of normalized compression distance [1] to compare two long strings directly. Improving $\log m \log u$ in time complexity is also an important work.

[5] The result for random *generated* pattern is omitted because the search time immediately converges under milliseconds in ESP, CSA, and FMI due to its rare occurrence.

Table 1. Locating Time

	ENGLISH				DNA			
	index size	locating time [sec]			index size	locating time [sec]		
	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$
ESP ($1/\varepsilon = 4$)	223292	84.27	0.93	2.44	212847	1923.45	0.63	1.73
ESP ($1/\varepsilon = 1$)	282646	60.76	0.67	1.81	269424	1271.55	0.46	1.33
ESP sparse ($1/\varepsilon = 4$)	181756	88.91	0.97	2.47	170954	1701.10	0.77	1.78
ESP sparse ($1/\varepsilon = 1$)	241110	73.09	0.67	1.81	227532	1400.90	0.46	1.32
LZI-1	290915	0.61	2.00	30.12	214161	7.23	0.77	16.34
CSA (-P1:64 -I:4:0)	308927	0.81	0.67	3.36	314529	1.79	0.66	3.56
CSA (-P1:64 -I:256:0)	107327	53.65	0.96	3.76	112929	168.83	1.02	3.17
CSA (-P1:256 -I:4:0)	288307	1.21	1.32	8.35	293865	3.02	1.41	8.67
CSA (-P1:256 -I:256:0)	86707	82.41	1.94	7.81	92265	314.05	1.43	8.81
FMI (-P4:512 -I:4:0)	265706	2.24	0.55	3.46	255483	3.94	0.36	2.54
FMI (-P4:512 -I:256:0)	64106	180.51	1.22	4.26	53883	412.22	0.64	2.59
FMI (-P7:128 -I:4:0)	336193	0.80	0.33	1.43	268264	1.17	0.19	0.67
FMI (-P7:128 -I:256:0)	134593	55.48	0.78	1.77	66664	96.22	0.25	0.59

Table 2. Displaying Time

	ENGLISH				DNA			
	index size	displaying time [sec]			index size	displaying time [sec]		
	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$
ESP ($1/\varepsilon = 4$)	223292	0.09	0.37	1.63	212847	0.12	0.21	1.08
ESP ($1/\varepsilon = 1$)	282646	0.07	0.25	1.18	269424	0.05	0.16	0.80
ESP sparse ($1/\varepsilon = 4$)	181756	0.16	0.47	2.58	170954	0.14	0.34	2.20
ESP sparse ($1/\varepsilon = 1$)	241110	0.10	0.37	1.99	227532	0.09	0.30	1.80
LZI-1	290915	0.01	0.04	0.27	227532	0.01	0.03	0.20
CSA (-P1:64 -I:0:4)	308927	0.04	0.28	1.20	314529	0.03	0.27	1.16
CSA (-P1:64 -I:0:256)	107327	0.30	0.47	1.22	112929	0.31	0.34	1.28
CSA (-P1:256 -I:0:4)	288307	0.04	0.31	1.83	293865	0.05	0.21	1.69
CSA (-P1:256 -I:0:256)	86707	0.25	0.53	2.17	92265	0.22	0.60	2.01
FMI (-P4:512 -I:0:4)	265706	0.09	0.39	2.52	255483	0.05	0.27	1.41
FMI (-P4:512 -I:0:256)	64106	0.09	0.41	2.27	53883	0.04	0.27	1.67
FMI (-P7:128 -I:0:4)	336193	0.05	0.30	0.94	268264	0.05	0.18	0.58
FMI (-P7:128 -I:0:256)	134593	0.08	0.37	1.10	66664	0.05	0.16	0.44

References

1. Cilibrasi, R., Vitanyi, P.M.B.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (2005)
2. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 235–246. Springer, Heidelberg (2009)
3. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. ACM Trans. Algor. 3(1), Article 2 (2007)
4. Delpratt, O., Rahman, N., Raman, R.: Engineering the LOUDS succinct tree representation. In: Àlvarez, C., Serna, M. (eds.) WEA 2006. LNCS, vol. 4007, pp. 134–145. Springer, Heidelberg (2006)
5. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 345–356. Springer, Heidelberg (2003)
6. Navarro, G.: Indexing text using the ziv-lempel tire. Journal of Discrete Algorithms 2(1), 87–114 (2004)
7. Navarro, G., Makinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article 2 (2007)
8. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms 48(2), 294–313 (2003)
9. Shapira, D., Storer, J.A.: Edit distance with move operations. J. Discrete Algorithms 5(2), 380–392 (2007)

Compressed Indexes for Aligned Pattern Matching*

Sharma V. Thankachan

Department of CS, Louisiana State University, USA
thanks@csc.lsu.edu

Abstract. In many situations like protein sequences, the primary protein sequence is associated with secondary structure labels [6]. This can be treated as two sequences aligned character by character. Many other DNA and RNA sequences involve linkages which are aligned across or in the same or different strands. In this paper, we consider the most natural characterization of aligned string data.

The *aligned pattern matching problem* is to index two input texts $T_1[1...n]$ and $T_2[1...n]$, each having n characters taken from an alphabet set Σ of size $\sigma = \text{polylog}(n)$, such that the following query can be answered efficiently: given two query patterns P_1 and P_2, find all the text positions i such that P_1 matches with $T_1[i...(i+|P_1|-1)]$ and P_2 matches with $T_2[i...(i+|P_2|-1)]$. Our objective is to design a compressed space index for this problem and we obtained the following main results: when the query patterns are sufficiently long ($|P_1|, |P_2| > \alpha = \Theta(\log^{2+2\epsilon} n)$, where $\epsilon > 0$), we can design an index which takes $nH'_k + nH''_k + o(n \log \sigma)$ bits space and $O(|P_1| + |P_2| + \log^{4+4\epsilon} n + t)$ query time, where H'_k and H''_k denotes the empirical kth-order entropy ($k = o(\log_\sigma n)$) of T_1 and T_2 respectively, t represents the number of outputs and $\epsilon > 0$. Further we show that designing a compressed/succinct space index with poly-logarithmic query time, which works for query patterns of all lengths is at least as hard as designing a linear space index for 3-dimensional orthogonal range reporting with poly-logarithmic query time. However, we introduce another compressed index of $nH'_k + nH''_k + O(n) + o(n \log \sigma)$ bits space requirement with a query time of $O(|P_1| + |P_2| + \sqrt{nt} \log^{2+\epsilon} n)$ which works without any restriction on the length of the patterns.

1 Introduction and Related Work

In many situations like protein sequences, the primary protein sequence is associated with secondary structure labels [6]. This can be treated as two sequences aligned character by character. Many other DNA and RNA sequences involve linkages which are aligned across or in the same or different strands. In this paper, we consider the most natural characterization of aligned string data.

Given two input texts $T_1[1...n]$ and $T_2[1...n]$, each having n characters taken from an alphabet set Σ of size $\sigma = \text{polylog}(n)$, the *aligned pattern matching*

* This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 410–419, 2011.
© Springer-Verlag Berlin Heidelberg 2011

problem is to index T_1 and T_2 such that the following query can be answered efficiently: given two query patterns P_1 and P_2, find all the text positions i such that P_1 matches with $T_1[i...(i+|P_1|-1)]$ and P_2 matches with $T_2[i...(i+|P_2|-1)]$. That is, let L_1 be the list of positions where P_1 matches T_1 and let L_2 be the list of positions where P_2 matches T_2, then the output is the intersection of L_1 and L_2. Hon et. al. [11] studied this problem and gave an algorithm based on a heuristic for fast intersection of lists which requires to access the elements in the lists in some order. However, intersection algorithms will not be efficient when the size of output is much smaller when compared to the size of lists to be intersected. Our objective is to design a compressed space index for this problem and we obtain the following main results:

1. For sufficiently long query patterns ($|P_1|, |P_2| > \alpha = \Theta(\log^{2+2\epsilon} n)$, where $\epsilon > 0$), we can design an index which takes $nH'_k + nH''_k + o(n \log \sigma)$ bits space and $O(|P_1|+|P_2|+\log^{4+4\epsilon} n+t)$ query time, where H'_k and H''_k denotes the empirical kth-order entropy ($k = o(\log_\sigma n)$) of T_1 and T_2 respectively, t represents the number of outputs and $\epsilon > 0$.
2. *Hardness of the problem:* Designing a compressed/succinct space index ($O(n \log \sigma)$ bits) with poly-logarithmic query time, which works for query patterns of all lengths is at least as hard as designing a linear space index ($O(n \log n)$ bits) for 3-dimensional orthogonal range reporting with poly-logarithmic query time.
3. We propose a compressed index of $nH'_k + nH''_k + O(n) + o(n \log \sigma)$ bits space with a query time of $O(|P_1| + |P_2| + \sqrt{nt} \log^{2+\epsilon} n)$ which works without any restriction on the length of the patterns.

2 Preliminaries

2.1 Suffix Trees and Suffix Arrays

Suffix trees [19] and suffix arrays [14] are two classic data structures for online pattern matching queries. For a text $T[1...n]$, substring $T[i..n]$, with $i \in [1, n]$, is called a suffix of T. The suffix tree for T is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. For any node v, the string formed by concatenating the edge labels from the root to v is called $path(v)$. The locus node v of a pattern P is defined as the node closest to the root, such that P is a prefix of $path(v)$.

Suffix array $SA[1...n]$ is an array of length n, such that $SA[i]$ is the starting position of the ith lexicographically smallest suffix of T. The suffix range of a pattern P in SA is defined as the the maximal range $[L, R]$ such that for all $j \in [L, R], SA[j]$ is the starting point of a suffixes of T with P as a prefix. Both suffix trees and suffix arrays take $O(n \log n)$ bits space and can perform pattern matching in $O(|P| + occ)$ and $O(|P| + \log n + occ)$ time respectively, where occ is the number of occurrences of P in T.

A space efficient version of suffix tree is called compressed suffix tree (CST) [18] and that of suffix array is called compressed suffix array (CSA) [9,8,7,16], which takes space close to the size of text T. In version of CSA by Ferragina et al. [7] for σ = polylog(n), $SA[i]$ and $SA^{-1}[j]$ can be computed in $O(\log^{1+\epsilon} n)$ time, and the suffix range of a pattern P can be computed in $O(|P|)$ time for any $\epsilon > 0$. The total space is $nH_k + o(n \log \sigma)$ bits, where H_k denotes the empirical kth-order entropy ($k = o(\log_\sigma n)$) of T.

2.2 Sparse Suffix Trees

Let α be a *sampling factor*. For a text $T[1..n]$, we call every substring $T[(1+i\alpha)...n]$ an α-sampled suffix of T, and every substring $T[1...i\alpha]$ an α-sampled prefix of T. Let Δ be a compact trie (of size $O(n/\alpha)$ nodes = $O((n/\alpha)\log n)$ bits) of all α-sampled suffixes of T. Note that Δ is not a self index, hence we need to maintain the original text T. For our purpose we maintain the text in the form of CSA [7] in $nH_k + o(n \log \sigma)$ bits, which is capable of retrieving any substring of T of length ℓ in $O(\ell + \log^{1+\epsilon} n)$ time (assuming σ =polylog(n)).

Lemma 1. *Given the suffix range of a pattern in suffix tree (or CSA), its suffix range in Δ can be obtained in constant time by maintaining an additional structure of size $\frac{n}{\alpha}\log(\alpha) + 1.44\frac{n}{\alpha} + o(n)$ bits.*

Proof: Let B be a bit vector such that $B[i] = 1$, if $SA[i]$ is an α-sampled suffix of T, else $B[i] = 0$. For a given pattern P, this gives a mapping between the suffix range $[L, R]$ in CSA and the suffix range $[L', R']$ in Δ as follows: $L' = 1 + rank_B(L-1)$ and $R' = rank_B(R)$, where $rank_B(i) = \sum_{j=1}^{i} B[j]$. Note that B can be maintained in $\frac{n}{\alpha}\log(\alpha) + 1.44\frac{n}{\alpha} + o(n)$ bits[1] and can perform this mapping in constant time [17]. □

Lemma 2. *The suffix ranges of all suffixes of P in Δ can be computed in $O(|P|)$ time by maintaining an index of size $nH_k + o(n \log \sigma) + O((n/\alpha)\log n)$ bits.*

Proof: The suffix ranges of all the suffixes of P in CSA (of size $nH_k + o(n \log \sigma)$ bits) can be computed in $O(|P|)$ time using backward search [8,7]. Using lemma 1, each of this suffix range in CSA can be mapped to Δ in constant time. Here $O((n/\alpha)\log n)$ bits is the size of Δ. □

2.3 Orthogonal Range Reporting

In [1], Alstrup et. al. showed that orthogonal range reporting of a set of n points in an $n \times n$ grid can be performed in $O(\log\log n + t)$ time by maintaining an $O(n \log^{1+\epsilon} n)$ bits index, where t represents the number of outputs. They also showed that for n points in an $n \times n \times n$ grid, orthogonal range reporting can be performed in $O(\log n + t)$ time using an $O(n \log^{2+\epsilon} n)$ bits index [1].

[1] Let $B[1...n]$ be a bit vector of length n with m 1's. Then B can be maintained in $\lceil\log\binom{n}{m}\rceil + o(n)$ bits, where $\lceil\log\binom{n}{m}\rceil \leq m\log(\frac{n}{m}) + 1.44m$ and can perform rank/select operations in constant time [17]. In our case $m = n/\alpha$.

2.4 A Simple Framework

In this subsection, we show a simple index for aligned pattern matching problem, which consists of two suffix trees (ST_1 and ST_2) of T_1 and T_2 and a two dimensional orthogonal range searching structure on n points of the form (x, y) such that $SA_1[x] = SA_2[y]$. Now the aligned pattern matching query can be answered by reporting those points (x, y) such that $L_1 \leq x \leq R_1$ and $L_2 \leq y \leq R_2$, where $[L_1, R_1]$ is the suffix range of P_1 in ST_1 and $[L_2, R_2]$ is the suffix range of P_2 in ST_2. By using the two dimensional orthogonal range searching structure by Alstrup et. al. [1], the space of this index can be bounded by $O(n \log^{1+\epsilon} n)$ bits and the query time will be $O(|P_1| + |P_2| + \log\log n + t)$, where t is the number of outputs.

2.5 Interleaving Technique

When the query patterns are of equal length ($|P_1| = |P_2|$), we can easily design a compressed space index. Let $T_1 = T_1[1...n]$ and $T_2 = T_2[1...n]$, then $T^* = T_1 \oplus T_2 = T_1[1]T_2[1]T_1[2]T_2[2]...T_1[n]T_2[n]$. That is $T^*[2i-1] = T_1[i]$ and $T^*[2i] = T_2[i]$. We call $\oplus$ as the interleaving function. Now, the query can be answered by retrieving all those occurrences of $P = P_1 \oplus P_2$ in odd positions of T. By maintaing the CSA [7] of T^* along with a bit vector (with constant time rank/select structures) marking all odd-suffixes, we can design an $2nH_k^* + 2n + o(n \log \sigma)$ bits index with $O(|P_1| + |P_2| + t \log^{1+\epsilon} n)$ query time, where H_k^* is the kth-order empirical entropy ($k = o(\log_\sigma n)$) of T^* and t represents the number of outputs and the size of alphabet set σ is polylog(n). Even though this index is derived for a very specialized case, we use this technique as a building block for designing compressed index for general case of different pattern lengths.

3 A Compressed Index for Long Query Patterns

In this section, we describe a compressed space solution, provided both the query patterns are longer than a sampling factor $\alpha = \Theta(\log^{2+2\epsilon} n), \epsilon > 0$. The index consists of the following components.

- CSA_1: Compressed suffix array of T_1
- CSA_2: Compressed suffix array of T_2
- Δ_1 : Trie of α-sampled suffixes of T_1
- Δ_2 : Trie of α-sampled suffixes of T_2
- Δ_I : Trie of all strings S of the form $S = S_1 \oplus S_2$, where S_1 and S_2 represents the reverse of α-sampled prefixes (of equal length) of T_1 and T_2 respectively and $\oplus$ is the interleaving function.
- A three dimensional orthogonal range searching structure $RS3D$.

The $RS3D$ consists of $\lceil n/\alpha \rceil$ points, such that a point (x, y, z) links the substrings $T_1[1...i\alpha], T_1[i\alpha + 1, ..., n], T_2[1...i\alpha]$ and $T_2[i\alpha + 1, ..., n]$ as follows:

- $T_1[i\alpha+1,...,n] = path(\ell_x)$, where ℓ_x is the xth leaf in Δ_1
- $T_2[i\alpha+1,...,n] = path(\ell_y)$, where ℓ_y is the yth leaf in Δ_2
- $S = S_1 \oplus S_2 = path(\ell_z)$, where ℓ_z is the zth leaf in Δ_I, where $S_1 =$ *reverse of* $T_1[1...i\alpha]$ and $S_2 =$ *reverse of* $T_2[1...i\alpha]$.

Now the given online query patterns P_1 and P_2 ($|P_1|$, $|P_2| > \alpha$) have an occurrence at some position j (i.e. $P_1 = T_1[j...j+|P_1|-1]$ and $P_2 = T_2[j...j+|P_2|-1]$) if the following conditions are satisfied for some $m \leq \alpha$,

- $P_1[1...m]$ is a suffix of an alpha sampled prefix $T_1[1...i\alpha]$ of T_1
- $P_2[1...m]$ is a suffix of an alpha sampled prefix $T_2[1...i\alpha]$ of T_2
- $P_1[m+1...|P_1|]$ is a prefix of an alpha sampled suffix $T_1[i\alpha+1...n]$
- $P_2[m+1...|P_2|]$ is a prefix of an alpha sampled suffix $T_2[i\alpha+1...n]$

Clearly $j = 1 + i\alpha - m$. Using our $RS3D$, all such aligned pattern matching occurrences can be obtained as follows: for $m = 1, 2, 3, ..., \alpha$,

1. Find the suffix range $[L_X, R_X]$ of $P_1[m+1...|P_1|]$ in Δ_1
2. Find the suffix range $[L_Y, R_Y]$ of $P_2[m+1...|P_2|]$ in Δ_2
3. Find the suffix range $[L_Z, R_Z]$ of $Q = Q_1 \oplus Q_2$ in Δ_I, where $Q_1 =$ *reverse of* $P_1[1...m]$ and $Q_2 =$ *reverse of* $P_2[1...m]$ and $\oplus$ is the interleaving function.
4. Perform the range reporting queries in $RS3D$ with the boundary $[L_X, R_X] \times [L_Y, R_Y] \times [L_Z, R_Z]$.

Let (x, y, z) be a reported point, then $i = SA_1[x] - 1$(or $i = SA_2[y] - 1$) and $j = 1 + \alpha(SA_1[x] - 1) - m$ will be a valid output. Note that the suffix corresponding to $SA_1[x]$ is an α-sampled suffix, whose value is directly stored in the trie Δ_1, hence we do not need to access CSA for computing $SA_1[x]$.

The space taken by CSA's of T_1 and T_2 is $nH'_k + nH''_k + o(n \log \sigma)$ bits, where H'_k and H''_k denotes the empirical kth-order entropy ($k = o(\log_\sigma n)$) of T_1 and T_2 respectively. By choosing $\alpha = \Theta(\log^{2+2\epsilon} n)$, ($\epsilon > 0$), the size of Δ_1 and Δ_2 can be bounded by $o(n)$ bits. The $RS3D$ structure takes $\Theta((n/\alpha)\log^{2+\epsilon} n) = o(n)$ bits space. The time for performing step 1 and step 2 for $m = 1, 2, 3, ..., \alpha$ can be bounded by $O(|P_1| + |P_2|)$ (from lemma 2). Step 4 takes $O(\alpha \log n + t) = O(\log^{3+2\epsilon} n + t)$ time (from Sec 2.3), where t represents the number of outputs. However, a question which is still remains unanswered is the space requirement of Δ_I and the query time for performing step 3. We conclude the result in the following lemma.

Lemma 3. *Step 3 in our aligned pattern matching algorithm can be performed in $O(\log^{4+4\epsilon} n)$ by maintaining an $o(n)$ bits index.*

Proof: The suffix range of the pattern Q in Δ_I can be computed by performing a binary search on the suffix array corresponding to Δ_I. Since, the text is not stored directly, retrieving a substring of length ℓ takes $O(\ell + \log^{1+\epsilon} n)$ time. Therefore we always retrieve $\log^{1+\epsilon} n$ characters from T_1 and T_2 and perform matching of Q. Since this is a binary search, there can be at most $O(\log n)$ mismatches (of $\log^{1+\epsilon} n$ characters). Thus the time for finding the suffix range

is $O((|Q| + \log^{2+\epsilon} n))$. Here we have α cases corresponding to each offset and $|Q| \leq 2\alpha$, hence the total time is $O(\alpha(\alpha + \log^{2+\epsilon} n)) = O(\log^{4+4\epsilon} n)$. The space taken by the suffix array corresponding to Δ_I is $O((n/\alpha)\log n) = o(n)$ bits. □

(We remark that this query time can be further improved to $O(\log^{3+3\epsilon} n)$ by maintaining a trie of reverse of all $\beta = \log^{1+\epsilon} n$ sampled prefixes of $T^* = T_1 \oplus T_2$ and make use of β-suffix links (a pointer from a node u to a unique node v, where $path(v)$ can be obtained by deleting first β characters of $path(u)$). Now the suffix range of a pattern in this trie can be mapped to the corresponding suffix range in Δ_I by maintaining an additional bit-vector.).

Thus by putting all together, we have the following theorem.

Theorem 1. *For query patterns of length* $> \alpha = \Theta(\log^{2+2\epsilon} n)$ *with* $\epsilon > 0$*, an index of size* $nH'_k + nH''_k + o(n \log \sigma)$ *bits can be maintained such that the aligned pattern matching queries can be answered in* $O(|P_1| + |P_2| + \log^{4+4\epsilon} n + t)$ *time, where* H'_k *and* H''_k *denote the empirical kth-order entropy* ($k = o(\log_\sigma n)$) *of* T_1 *and* T_2 *respectively and* t *represents the number of outputs.*

4 Hardness of the Problem

For all the compressed indexes derived so far, we assumed some conditions on the length of the query patterns (equal length, longer than α). It is interesting to know, if a compressed space index for aligned pattern matching without any constraints on query patterns can be derived. In [5] Chien et. al. showed a framework for proving lower bounds in pattern matching problems by reducing them to geometrical problems. We follow this technique to study the hardness of aligned pattern matching problems in compressed/succinct ($O(n \log \sigma)$ bits) space.

We first show the following reduction: a three dimensional orthogonal range reporting query on n points in an $n \times n \times n$ grid can be answered in $O(\log^3 n)$ aligned pattern matching queries. Note that each point of the form (x, y, z) can be represented using $h = O(\log n)$ bits. Let $\langle s \rangle$ be the binary representation of a string s and $\overleftarrow{\langle s \rangle}$ be the reverse of $\langle s \rangle$. Now we define two strings T_1 and T_2 as follows:

$$T_1 = \overleftarrow{\langle y_1 \rangle}\#\langle x_1 \rangle \star \overleftarrow{\langle y_2 \rangle}\#\langle x_2 \rangle \star ... \overleftarrow{\langle y_n \rangle}\#\langle x_n \rangle \star$$
$$T_2 = \langle w \rangle\#\langle z_1 \rangle \star \langle w \rangle\#\langle z_2 \rangle \star ... \langle w \rangle\#\langle z_n \rangle \star$$

Here $\langle w \rangle$ is a string of length h with all characters as 0. We show that, a succinct index ($(n \log \sigma)$ bits) on T_1 and T_2 for aligned pattern matching can be used as a linear index ($O(n \log n)$ bits) for three dimensional orthogonal range reporting. Note that the alphabet set is $\Sigma = \{0, 1, \#, \star\}$. Now from the orthogonal range query boundaries, our task is to generate the query patterns which can be fed to the aligned pattern matching index.

Lemma 4. *A given range* $[\ell, r]$ *can be represented by a set* $S = \{s_1, s_2, ..., s_k\}$ *of* $k \leq 2\log n$ *binary strings, such that none of these string is a prefix of another and any binary string of length* h *with prefix* $s_i \in S$ *is contained in* $[\ell, r]$.

Proof: Let Λ be a trie of binary representations of all integers $\in [1, n]$. Now any range $[\ell, r]$ can be split into $k \leq 2\log n$ non-overlapping sub-ranges such that each of these sub-range represents the complete sub-tree of some internal node u in Λ. Then S represents the set of *paths* from the root to all such nodes. □

Let S_X, S_Y and S_Z represents the set of binary strings (constructed using lemma 4) corresponding to the given ranges in each dimension. Then the set of query patterns P_1 and P_2 can be generated as follows: $P_1 = \overleftarrow{s_y}\#s_x$ and $P_2 = w_y\#s_z$, where $s_x \in S_X, s_y \in S_Y, s_z \in S_Z$ and w_y is a bit string of all $0'$s and of length $|s_y|$. Note that we can have $O(\log^3 n)$ combinations of s_x, s_y and s_z, hence $O(\log^3 n)$ aligned pattern matching queries and the length of each query pattern is $O(\log n)$. Therefore the total time for all aligned pattern matching query can be bounded by $O((t+1)\text{polylog}(n))$, where t is the number of outputs. Now a point (x_i, y_i, z_i) is a valid answer to the range reporting query, if and only if there is an aligned pattern matching across the ith $\#$ symbol.

Theorem 2. *Given an* $O(n\log\sigma)$ *bits index for aligned pattern matching problem with poly-logarithmic query time, we can design an* $O(n\log n)$ *bits index for 3-dimensional orthogonal range reporting problem (of* n *points in an* $n \times n \times n$ *grid) with poly-logarithmic query time.*

5 A Compressed Index for All Query Patterns

In this section, we propose a compressed index which works for all patterns, and the query answering time becomes $O(|P_1| + |P_2| + \sqrt{nt}\log^{2+\epsilon} n)$. We first show a sampling scheme over the suffix tree (ST) nodes. Let g be a parameter called *group size*. Starting from the left most leaf in ST, we combine every g contiguous leaves to form a group. Thus the first group consists of $\ell_1, ..., \ell_g$, the next group consists of $\ell_{g+1}, ..., \ell_{2g}$, and so on, where ℓ_j denotes the jth leaf from left. The total number of groups is $O(n/g)$. Now for each group we mark the lowest common ancestor (LCA) of the whole group of leaves. The marking continues as follows: if two nodes are marked, we mark their LCA also. It can be easily shown that the total number of marked nodes is $O(n/g)$ [12,13]. Now, suppose for a node u, the sub-tree rooted at u contains the leaves $\ell_x, \ell_{x+1}, \ldots, \ell_y$, we refer $[x, y]$ as the *suffix range* corresponding to u.

Lemma 5. *The suffix range* $[L, R]$ *of any pattern* P *can be split into a suffix range* $[L', R']$ *corresponding to some marked node* u^**, and two other suffix ranges* $[L, L'-1]$ *and* $[R'+1, R]$ *with* $L' - L < g$ *and* $R - R' < g$*. We call* u^* *the highest marked descendent (HMD) of* u *and those leaves* ℓ_i*, such that* $i \in [L, L'-1]$ *or* $i \in [R'+1, R]$*, the fringe leaves of* u.

Proof: By setting $L' = g\lceil L/g\rceil + 1$ and $R' = g\lfloor R/g\rfloor$, the LCA of ($\ell_{L'}$ and $\ell_{R'}$) gives the desired marked node u^*, and we also have $L' - L < g$ and $R - R' < g$

(i.e. the number of fringe leaves $< 2g$). Note that the existence of such a marked node is not necessary (in the case when suffix range is smaller than g). □

Let CST_1 and CST_2 be the compressed suffix trees (as described in Section 2.1) of T_1 and T_2 respectively. For any node u_1 in CST_1, let $L(u_1)$ represent the set of occurrences in the subtree of u_1. Similarly, for a node u_2 in CST_2, $L(u_2)$ represents the set of occurrences in the subtree of u_2 and $L(u_1, u_2)$ represent the intersection of $L(u_1)$ and $L(u_2)$. Based on the above notations, for any two patterns P_1 and P_2 with locus nodes u_1 and u_2 in CST_1 and CST_2 respectively, $L(u_1, u_2)$ represents the answer to the aligned pattern matching query. However, it is not space efficient to store $L(u_1, u_2)$ for all possible pairs of nodes, therefore we use the above described sampling scheme as follows:

- choose the group size $g = \sqrt{nk} \log^{1+\epsilon} n$
- Mark the nodes in CST_1 and CST_2 with the grouping factor g
- Between all pairs of marked nodes u_1^* and u_2^* in CST_1 and CST_2 respectively, if $|L(u_1^*, u_2^*)| \leq k$, then we store $|L(u_1^*, u_2^*)|$ explicitly, where $|L|$ is the number of elements in the list L.
- Maintain these information for $k = 1, 2, 4, 8, ..., n/\log^{2+2\epsilon} n$ (Note that $g = \sqrt{nk} \log^{1+\epsilon} n \leq n$, hence k can be at most $n/\log^{2+2\epsilon} n$).

For a particular k, the number of pairs of marked nodes is $O(n/g) \times O(n/g)$ and the amount of information stored is $O(n/(k \log^{2+2\epsilon} n) \times k \log n) = O(n/\log^{1+2\epsilon} n)$. Since k takes $\log n$ $(1, 2, 4, ...)$ different values, the total space for storing these precomputed answers is $o(n)$ bits. Hence the total index space is bounded by $nH'_k + nH''_k + O(n) + o(n \log \sigma)$ bits, where H'_k and H''_k denotes the empirical kth-order entropy ($k = o(\log_\sigma n)$) of T_1 and T_2 respectively.

5.1 Query Answering

The query answering can be performed as follows: first find the locus nodes u_1 and u_2 of P_1 and P_2 in CST_1 and CST_2 respectively. Let t represent the number of outputs and t_k be the number of entries in $L(u_1^*, u_2^*)$, where u_1^* and u_2^* are the highest marked descendants of u_1 and u_2 respectively corresponding to the group size $g = \sqrt{nk} \log^{1+\epsilon} n$. Note that t_k for any k can be found in constant time (from lemma 4) using constant time LCA operations and $t_k \leq t$ for all values of k.

Now our task is simple: for $k = 1, 2, 4, ..$ check t_k and find the *smallest* $k(= k')$ such that $t_{k'} \leq k'$. Therefore we can safely write $k' = O(t)$. Note that $L(u_1^*, u_2^*)$ is a subset of $L(u_1, u_2)$, and those answers in $L(u_1^*, u_2^*)$ can be retrieved in $O(t_{k'}) = O(t)$ time. The remaining answers in $L(u_1, u_2)$ can come from only those suffixes corresponding to the fringe leaves. Therefore, by checking at most $2g$ fringe leaves in each CST, the remaining valid outputs can be retrieved as follows: let ℓ_x be a fringe leaf of u_1, then $SA_1[x]$ is a valid output if $SA_2^{-1}[SA_1[x]] \in [L_2, R_2]$, where $[L_2, R_2]$ is the suffix range of P_2. Similarly, if ℓ_y is a fringe leaf of u_2, then $SA_2[y]$ is a valid output if $SA_1^{-1}[SA_2[y]] \in [L_1, R_1]$, where $[L_1, R_1]$ is the suffix range of P_1.

Each of this checking takes $O(\log^{1+\epsilon} n)$ time (from Section 2.1), hence the total time for checking all fringe leaves is $O(g \log^{1+\epsilon} n) = O(\sqrt{nk'} \log^{2+2\epsilon} n) = O(\sqrt{nt} \log^{2+2\epsilon} n)$ (which is the main bottleneck in query time).

Theorem 3. *By maintaining an index of size $nH'_k + nH''_k + O(n) + o(n \log \sigma)$ bits, aligned pattern matching can be performed in $O(|P_1| + |P_2| + \sqrt{nt} \log^{2+2\epsilon} n)$ time, where H'_k and H''_k denotes the empirical kth-order entropy ($k = o(\log_\sigma n)$) of T_1 and T_2 respectively and σ =polylog(n).*

6 Concluding Remarks

In this paper, we studied the aligned pattern matching problem and proposed different indexes. An interesting open problem in this research direction is to design an index for aligned pattern matching with shift δ (which comes as an online parameter). That is, a position i is reported if and only if P_1 occurs at position i in T_1 and P_2 occurs at position $i + \delta$ in T_2.

References

1. Alstrup, S., Bordal, G.S., Rauhe, T.: New data structure for orthogonal range searching. In: FOCS, pp. 198–207 (2000)
2. Chazelle, B.: Lower bounds for orthogonal range searching: I. the reporting case. JACM 37, 200–212, 1990 (2005)
3. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
4. Burrows, M., Wheeler, D.J.: A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA (1994)
5. Chien, Y.F., Hon, W.-K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In: DCC 2008, pp. 252–261 (2008)
6. Eltabakh, M.Y., Hon, W.-K., Shah, R., Aref, W.G., Vitte, J.S.: The SBC-tree: an index for run-length compressed sequences. In: EDBT, pp. 523–534 (2008)
7. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. TALG 3(2) (2007)
8. Ferragina, P., Manzini, G.: Indexing Compressed Text. JACM 52(4), 552–581 (2005); A preliminary version appears in FOCS 2000
9. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM Journal on Computing 35(2), 378–407 (2005); A preliminary version appears in STOC 2000
10. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: SODA, pp. 841–850 (2003)
11. Hon, W.-K., Shah, R., Vitter, J.S.: Ordered Pattern Matching: Towards Full-Text Retrieval. Technical Report TR-06-008, Purdue University (March 2006)
12. Hon, W.-K., Shah, R., Vitter, J.S.: Space-Efficient Framework for Top-k String Retrieval Problems. In: FOCS 2009, pp. 713–722 (2009)

13. Hon, W.-K., Shah, R., Thankachan, S.V., Vitter, J.S.: String Retrieval for Multi-pattern Queries. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 55–66. Springer, Heidelberg (2010)
14. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
15. Munro, J.I., Raman, V.: Succinct Representation of Balanced Parentheses and Static Trees. SICOMP 31(3), 762–776 (2001)
16. Navarro, G., Mäkinen, V.: Compressed Full-Text Indexes. ACM Computing Surveys 39(1) (2007)
17. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. TALG 3(4) (2007)
18. Sadakane, K.: Compressed Suffix Trees with Full Functionality. In: TCS, pp. 589–607 (2007)
19. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. Switching and Automata Theory, pp. 1–11 (1973)

Reference Sequence Construction for Relative Compression of Genomes*

Shanika Kuruppu[1], Simon J. Puglisi[2], and Justin Zobel[1]

[1] National ICT Australia
Department of Computer Science & Software Engineering,
University of Melbourne, Australia
{kuruppu,jz}@csse.unimelb.edu.au
[2] Department of Informatics
King's College London, United Kingdom
simon.puglisi@rmit.edu.au

Abstract. Relative compression, where a set of similar strings are compressed with respect to a reference string, is an effective method of compressing DNA datasets containing multiple similar sequences. Moreover, it supports rapid random access to the underlying data. The main difficulty of relative compression is in selecting an appropriate reference sequence. In this paper, we explore using the dictionary of repeats generated by COMRAD, RE-PAIR and DNA-X algorithms as reference sequences for relative compression. We show that this technique allows for better compression, and allows more general repetitive datasets to be compressed using relative compression.

1 Introduction

Rapid advancements in the field of high-throughput sequencing have led to a large number of whole genome DNA sequencing projects. Some of these projects, like the Genome 10K project (www.genome10k.org) seek to obtain genomes of unsequenced species. Others, like the 1000 Genomes project (www.1000genomes.org) for humans and the 1001 Genomes project (www.1001genomes.org) for *Arabidopsis thaliana* plants, focus on resequencing, where individual genomes from a given species are sequenced to understand variation between individuals. The assembled sequences from these projects can range from terabytes to petabytes in size. Therefore, algorithms and data structures to efficiently store, access and search these large datasets are necessary. Some progress has already been made [2,7,12], but significant challenges remain.

DNA sequences contain repeated substrings, but, in a database of sequences, the most significant repeats occur *between* sequences, usually those of the same

* This work was supported by the Royal Society and the NICTA Victorian Research Laboratory. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Center of Excellence program.

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 420–425, 2011.
© Springer-Verlag Berlin Heidelberg 2011

or similar species. Compression algorithms that capture and efficiently encode this repeated information are employed to manage large genomic databases. Many compression algorithms are specific to DNA sequence compression [3,4,6], but most are unsuitable for compressing large multi-sequence datasets. More recently, algorithms that compress large repetitive datasets, which also support random access and search on the compressed dataset, known as *self-indexes*, have emerged; some are specific to DNA compression and support random access queries [8,9], while others can compress general datasets and allow search queries on the compressed sequences [7,12].

An effective way to compress a repetitive dataset containing multiple sequences from the same or similar species, or sequences serving the same biological function, is to compress each sequence with respect to a chosen reference [3,9,12]. Grumbach and Tahi [6] realised the need for such a compression method for DNA sequences, and `GenCompress` [4] and `XM` [3] implement this feature. Mäkinen et al. introduce more general methods to compress highly repetitive collections that also support searching in the compressed data [12].

The `RLZ` algorithm compresses each sequence using an `LZ77` parsing [14] with respect to a reference sequence chosen from the dataset [9]. Relative compression algorithms like `RLZ` produce good compression results with fast compression and decompression speeds. The reference sequence acts as a static "dictionary" that includes most of the repeats present in the input dataset. Once an index of the reference sequence is built, the sequences can be compressed in a single pass over the collection. The main drawback is the difficulty of selecting an appropriate reference sequence. Selecting a random reference sequence from a collection of similar sequences does not guarantee good compression. Figure 1 shows the compressed sizes produced by `RLZ` when each of 39 *S. cerevisiae* yeast genomes is used as a reference sequence to compress the remaining genomes. As can be seen, there is a significant variation in the compressed sizes.

Even if the best reference sequence for the dataset can be found, a single reference sequence still may not be representative of the repetitions present in the whole dataset. Sequences may form clusters, where a sequence is highly similar to only a few other sequences in the dataset. To test this hypothesis, we examined the position components of the factors that are generated by `RLZ` for the *S. cerevisiae* dataset that form alignments to the reference sequence. We found clusters of similar sequences, which could be partitioned according to where factors commenced.

Moreover, for datasets containing sequences from different strains or different species, selecting a reference sequence is not trivial. Grabowski and Deorowicz [5], in one of their many improvements to `RLZ`, address this issue. When substrings of a certain minimum length, which do not occur in the reference sequence, are encountered, they are appended to the reference sequence, so that later occurrences of those substrings can be encoded as references. This method provides a slight improvement to compression with no effects on the compression or decompression speed. However, the method may add more substrings to the reference sequence than necessary.

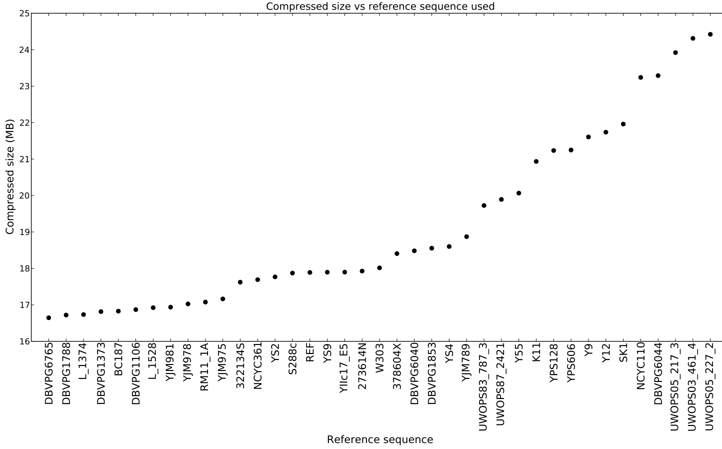


Fig. 1. The change in the compressed size of the *S. cerevisiae* dataset when the reference sequence is changed. The y-axis contains the compressed size, measured in Megabytes and the x-axis contains the reference sequence used.

In this paper we explore the artificial construction of reference sequences from the phrases built by popular dictionary compressors. Dictionary compressors find the repeated substrings in the dataset being compressed and stores the repeats in a dictionary. These dictionary entries can be used to construct a reference sequence consisting of significant repeats of the entire dataset. We show that reference sequences constructed in this manner produce superior compression results, while retaining the principle advantage of relative compression: fast random access to the collection.

2 Reference Sequence Construction

We choose three dictionary compression algorithms for generation of reference sequences; `RE-PAIR` [11], a well-known dictionary compression algorithm; our `COMRAD` [8]; and `DNA-X` [13], a DNA-specific implementation of the algorithm by Bentley and McIlroy [1]. We compress our test datasets with `RE-PAIR`, `COMRAD`, and `DNA-X`, and then use the dictionary of repeats to construct a reference sequence for relative compression as discussed below.

`RE-PAIR` compresses the input sequence in multiple iterations, where at each iteration, a symbol pair with the highest frequency is replaced with a new non-terminal symbol [11]. The algorithm outputs the input sequence with all its repeated substrings replaced by non-terminal symbols, and a dictionary of rules that map the non-terminals to the symbol pairs that they replaced. The substrings represented by the non-terminals in the `RE-PAIR` dictionary contains the repeated substrings of the input sequence, and these substrings can be concatenated to create a reference sequence. The `RE-PAIR` dictionary is hierarchical,

since a non-terminal can substitute a symbol pair containing at least one other non-terminal. For example, the substring for non-terminal Z in the set of rules $Z \leftarrow XY$, $X \leftarrow aA$, $Y \leftarrow CD$, includes the substrings for the non-terminals X, Y, A, C and D. Therefore, once the substring for Z is included in the reference sequence, it is redundant to include the substrings represented by each of the non-terminals X, Y, A, C and D. To ensure that only non-redundant substrings are included in the reference sequence, we start adding substrings to the reference sequence from higher in the hierarchy of rules.

COMRAD is a dictionary compression algorithm, similar to RE-PAIR, that detects repeated substrings in the input over multiple iterations, and encodes them efficiently to achieve compression [8]. Instead of replacing pairs of frequent symbols, COMRAD replaces repeated substrings of longer lengths to reduce the number of iterations. The algorithm is disk-based to allow large DNA datasets to be compressed. As with the RE-PAIR dictionary, we expand non-terminals and append them to create a reference sequence, ensuring to omit redundant non-terminals.

DNA-X is a single-pass dictionary compression algorithm [13]. As the input is read, the fingerprint of every B-th substring of length B is stored in a hash table. To encode the next substring, all overlapping B-mers in the so far unencoded part of the input are searched for in the hash table until there is a match. The hash table gives the positions of the earlier occurrences of the B-mer. Each of these occurrences is checked to find the longest possible match. Then the prefix until the matching substring, followed by the reference for the matching substring is encoded. Searching and encoding continues until no more symbols remain to be encoded. The longest matching substrings encoded by the algorithm are the repeated substrings we use to construct the reference sequence.

3 Experimental Results

To test the performance of the reference construction method, we use RLZ [10] as the relative compressor and three test datasets containing repetitive genomes: 39 strains of *S. cerevisiae* and 36 strains of the *S. paradoxus* species of yeast, and 33 strains of *E. coli* bacteria. We ran RE-PAIR, COMRAD and DNA-X on all three datasets. For RE-PAIR, we used the default parameters, which do not place any restrictions on the number or length of repeats that can be detected. For COMRAD, we used a starting substring length L of 16 and a threshold frequency F of 2. For DNA-X we set the substring length B to 16 to be consistent with COMRAD. The repeated substrings resulting from the dictionaries were used to generate the reference sequence as described above.

Compression results for the three datasets are presented in columns 2–4 of Table 1. The first row contains the number of megabases in each dataset. The second row contains the result produced by the optimised RLZ algorithm [10] when using the reference sequence in the dataset that gives the best compression result. The remaining rows contain the compression results produced by RLZ when using the reference sequences constructed by each of the dictionary compression algorithms. For all three datasets, using a COMRAD, RE-PAIR or DNA-X

Table 1. Compression results in Mbytes for using COMRAD, RE-PAIR and DNA-X generated reference sequences. The first line contains the dataset sizes in Mbases. The standard compression results for RLZ includes all optimisations. In the last three lines, RLZ was used with only the looking ahead and short factor encoding optimisations.

Dataset	Scerevisiae	Sparadoxus	Ecoli	Hemoglobin	Influenza	Mitochondria
Original	485.87	429.27	164.90	7.38	112.64	25.26
RLZ	9.33	13.23	18.69	2.97	38.93	8.36
RLZ(COMRAD)	6.77	8.10	8.06	1.17	3.00	6.05
RLZ(RE-PAIR)	6.48	7.70	7.72	1.19	2.82	6.20
RLZ(DNA-X)	7.51	8.80	9.14	1.21	3.63	6.05

Table 2. Compression times in seconds for COMRAD, RE-PAIR and DNA-X generated reference sequences. Times include generation of reference sequences, where necessary.

Dataset	Scerevisiae	Sparadoxus	Ecoli	Hemoglobin	Influenza	Mitochondria
RLZ	300	276	200	3	42	11
RLZ(COMRAD)	1775	1779	600	20	242	110
RLZ(RE-PAIR)	1672	1554	622	17	296	90
RLZ(DNA-X)	1529	1499	566	17	336	77

generated reference sequence produced better results than using a single reference sequence. The biggest improvement (a factor of two) is for *E. coli*.

Overall, using the RE-PAIR generated reference sequences led to slightly better compressed sizes than using the COMRAD generated reference sequences. The DNA-X-generated reference sequences produced less promising results due to the generated reference being somewhat large as a result of some redundant repeats being included. Our best results for *S. cerevisiae* and *S. paradoxus* of 7.64 Mbyte and 8.67 Mbyte, respectively, are comparable to those obtained by Grabowski and Deorowicz [5]. It may be possible to combine our improvement with theirs to acheive even better results.

Table 2 shows compression times. As expected, the compression time increases significantly when using a generated reference sequence, as the time taken to generate the reference is also part of the compression time. More importantly, decompression times were not affected. In general, using a COMRAD generated reference is slower than using a RE-PAIR generated reference. However, the main memory usage of RE-PAIR is much higher, with *S. cerevisiae* and *S. paradoxus* using approximately 12 Gb and 11 Gb, respectively, while COMRAD only requires 277 Mbyte and 554 Mbyte. DNA-X has the lowest memory usage.

Next we experiment with datasets which do not contain a specific reference. These were a *Hemoglobin* dataset containing 15,199 DNA sequences of proteins that are associated with Hemoglobin, an *Influenza* dataset containing 78,041 sequences of various strains of the Influenza virus and a *Mitochondria* dataset containing 1,521 mitochondrial DNA sequences from various species. Results are presented in the last three columns of Table 1. The second row contains the compression results of standard RLZ when the first sequence in the dataset is chosen to be the reference. Compression clearly improves for all three datasets.

4 Concluding Remarks

Relative compression is a powerful technique for compressing collections of related genomes, which are now becoming commonplace. In this paper we have shown that such collections can contain clusters of sequences which are more highly related than others and that impressive gains in compression can be achieved by exploiting these clusters. Our specific approach has been to detect repetitions across the dataset and build an artificial "reference sequence", relative to which the sequence is then compressed. This method retains the principle advantage of relative compression: fast random access. The drawback is slower compression time, as time must now be spent finding repeats with which to generate the reference. Future work will attempt to address this problem. It may also be fruitful to apply clustering algorithms to genomes to isolate strains.

References

1. Bentley, J., McIlroy, D.: Data compression using long common strings. In: Proc. Data Compression Conference (DCC 1999), pp. 287–295 (1999)
2. Brandon, M., Wallace, D., Baldi, P.: Data structures and compression algorithms for genomic sequence data. Bioinformatics 25(14), 1731–1738 (2009)
3. Cao, M.D., Dix, T., Allison, L., Mears, C.: A simple statistical algorithm for biological sequence compression. In: Proc. Data Compression Conference (DCC 2007), pp. 43–52 (2007)
4. Chen, X., Li, M., Ma, B., Tromp, J.: DNACompress: fast and effective DNA sequence compression. Bioinformatics 18(12), 1696–1698 (2002)
5. Grabowski, S., Deorowicz, S.: Engineering relative compression of genomes (2011), http://arxiv.org/abs/1103.2351v1
6. Grumbach, S., Tahi, F.: A new challenge for compression algorithms: Genetic sequences. Information Processing & Management 30(6), 875–886 (1994)
7. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 41–54. Springer, Heidelberg (to apppear, 2011)
8. Kuruppu, S., Beresford-Smith, B., Conway, T., Zobel, J.: Iterative dictionary construction for compression of large DNA datasets. IEEE/ACM Transactions on Computational Biology and Bioinformatics (to appear, 2011)
9. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)
10. Kuruppu, S., Puglisi, S.J., Zobel, J.: Optimized relative Lempel-Ziv compression of genomes. In: Proc. 34th Australasian Computer Science Conference (ACSC 2011), pp. 91–98 (2011)
11. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Proc. Data Compression Conference (DCC 1999), pp. 296–305 (1999)
12. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Computational Biology 17(3), 281–308 (2010)
13. Manzini, G., Rastero, M.: A simple and fast DNA compressor. Software: Practice and Experience 34(14), 1397–1411 (2004)
14. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)

Author Index

Abdul Nasir, Jamal 261
Alonso, Omar 26
Amir, Amihood 44, 168

Baeza-Yates, Ricardo 26, 368
Bannai, Hideo 278
Belazzougui, Djamal 55, 386
Beller, Timo 197
Bernhard, Delphine 221
Bonzanini, Marco 14
Boughanem, Mohand 117
Brejová, Broňa 144
Breslauer, Dany 156, 301

Carvalho, Cristiano 237
Chen, Kuan-Yu 81
Christou, Michalis 338
Claude, Francisco 185
Cristo, Marco Antônio 237
Crochemore, Maxime 338
Cummins, Ronan 380
Cuzzocrea, Alfredo 87

da Silva, Altigran Soares 237
Demaine, Erik D. 1
de Moura, Edleno Silva 237
Denecke, Kerstin 87
Dinarelli, Marco 221
Dupret, Georges 2

Egidi, Lavinia 32

Fisichella, Marco 87
Flouri, Tomáš 338

Gagie, Travis 295
Garcia-Fernandez, Anne 221
Gertz, Michael 26
Ghodsnia, Pedram 313
Gil-Costa, Veronica 344
Gog, Simon 197
González-Caro, Cristina 368
Goto, Keisuke 278
Gotthilf, Zvi 44

Hagen, Matthias 356
He, Meng 295
Henrique, Wallace Favoreto 237
Hon, Wing-Kai 267

Iliopoulos, Costas S. 338
Inenaga, Shunsuke 278
Italiano, Giuseppe F. 156, 301

Janoušek, Jan 338
Jones, Gareth J.F. 249
Jose, Joemon M. 380

Karim, Asim 261
Kärkkäinen, Juha 174
Kishiue, Naoya 398
Kopelowitz, Tsvi 67
Kopliku, Arlind 117
Ku, Tsung-Han 267
Kucherov, Gregory 326
Kuruppu, Shanika 420

Lalmas, Mounia 380
Landau, Gad M. 144
Lewenstein, Moshe 67, 135
Ligozat, Anne-Laure 221
López-Ortiz, Alejandro 313

Macdonald, Craig 104, 344
Manzini, Giovanni 32
Maruyama, Shirou 398
McCreadie, Richard 104
Melichar, Bořivoj 338
Mendoza, Marcelo 129
Min, Jinming 249
Munro, J. Ian 295, 313

Nakahara, Masaya 398
Navarro, Gonzalo 386
Nicholson, Patrick K. 185, 295

Ohlebusch, Enno 197
O'Riordan, Colm 380
Ounis, Iadh 104, 344

Paryenty, Haim 168
Pinel-Sauvagnat, Karen 117
Pissis, Solon P. 338
Poblete, Barbara 129
Porat, Ely 67
Puglisi, Simon J. 174, 420

Raffinot, Mathieu 55
Roditty, Liam 168
Roelleke, Thomas 14
Russo, Luís M.S. 290

Sakamoto, Hiroshi 398
Santos, Rodrygo L.T. 344
Schnattinger, Thomas 197
Seco, Diego 185
Shah, Rahul 267
Shalom, B. Riva 44
Spiliopoulou, Myra 129
Stein, Benno 356
Stewart, Avaré 87

Takeda, Masayuki 278
Thachuk, Chris 209
Thankachan, Sharma V. 267, 410
Tirdad, Kamran 313
Tischler, German 290
Tsatsaronis, George 261

Varlamis, Iraklis 261
Vinař, Tomáš 144
Vitter, Jeffrey Scott 267

Wang, Hung-Lung 81

Yahyaei, Sirvan 14

Ziviani, Nivio 237
Zobel, Justin 420