LNCS 5721

Jussi Karlgren
Jorma Tarhio
Heikki Hyyrö (Eds.)

String Processing and Information Retrieval

16th International Symposium, SPIRE 2009
Saariselkä, Finland, August 2009
Proceedings

Lecture Notes in Computer Science 5721

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
University of Dortmund, 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 of Computer Science, Saarbruecken, Germany

Jussi Karlgren   Jorma Tarhio
Heikki Hyyrö (Eds.)

String Processing and Information Retrieval

16th International Symposium, SPIRE 2009
Saariselkä, Finland, August 25-27, 2009
Proceedings

Springer

Volume Editors

Jussi Karlgren
Swedish Institute of Computer Science
Kista, Sweden
E-mail: jussi@sics.se

Jorma Tarhio
Helsinki University of Technology
Espoo, Finland
E-mail: tarhio@cs.hut.fi

Heikki Hyyrö
University of Tampere
Tampere, Finland
E-mail: heikki.hyyro@cs.uta.fi

Library of Congress Control Number: 2009932181

CR Subject Classification (1998): E.5, F.2.2, H.3.3, E.1, E.2, H.2.8, I.5

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

ISSN 0302-9743
ISBN-10 3-642-03783-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-03783-2 Springer Berlin Heidelberg New York

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.

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12738502 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 16th International Symposium on String Processing and Information Retrieval (SPIRE 2009), held during August 25–27, 2009 in Saariselkä, Finland.

The annual SPIRE conference provides researchers within fields related to string processing and/or information retrieval a possibility to present their original contributions and to meet and discuss with other researchers with similar interests. The Call for Papers invited submissions related to string processing (dictionary algorithms; text searching; pattern matching; text and sequence compression; automata-based string processing), information retrieval (information retrieval models; indexing; ranking and filtering; interface design; visualization; benchmarking), natural language processing (text analysis; text mining; machine learning; information extraction; language models; knowledge representation), search applications and usage (cross-lingual information access systems; multimedia information access; digital libraries; collaborative retrieval and Web-related applications; semi-structured data retrieval; evaluation), and interaction of biology and computation (DNA sequencing and applications in molecular biology; evolution and phylogenetics; recognition of genes and regulatory elements; sequence driven protein structure prediction).

The papers presented at the symposium were selected from 84 submissions written by authors from 28 different countries. Each submission was reviewed by at least two and on average 2.9 reviewers. The Committee accepted 34 papers ($\approx 40\%$): 22 papers for 25-minute and 12 papers for 15-minute presentations. In addition to these, SPIRE 2009 also featured invited talks by Mehryar Mohri (New York University, USA) and Kalervo Järvelin (University of Tampere, Finland).

The Program Committee voted to give the Best Paper Award to Amihood Amir and Haim Parienty for their paper "Towards a Theory of Patches."

We are especially thankful to the members of the Program Committee, who provided us with thorough and timely reviews despite the fact that the reviewing schedule was unusually tight due to the early date of the 2009 symposium.

Additional thanks go to the following people for their generous helpfulness and support throughout the process of planning and organizing SPIRE 2009: SPIRE Steering Committee coordinator Ricardo Baeza-Yates, the editorial office staff at Springer, the office personnel at the University of Tampere, and the staff at the conference hotel Riekonlinna in Saariselkä.

We are grateful for the financial support provided by the Department of Computer Sciences of University of Tampere, the Federation of Finnish Learned Societies, Yahoo! Research, the Department of Computer Science of University of Helsinki, and the Swedish Institute of Computer Science.

June 2009

Jussi Karlgren
Jorma Tarhio
Heikki Hyyrö

SPIRE 2009 Organization

Organizing Institution

Department of Computer Sciences, University of Tampere, Finland

Organizing Committee

Local Organization Chair

Heikki Hyyrö	University of Tampere, Finland

Program Committee Chairs

Jussi Karlgren	Swedish Institute of Computer Science, Sweden
Jorma Tarhio	Helsinki University of Technology, Finland

Program Committee

Tatsuya Akutsu	Kyoto University, Japan
Srinivas Aluru	Iowa State University, USA
Amihood Amir	Bar-Ilan University, Israel, and Johns Hopkins University, USA
Alberto Apostolico	Georgia Institute of Technology, USA, and University of Padova, Italy
Ricardo Baeza-Yates	Yahoo! Research, Spain and Chile, and CWR/DCC, University of Chile, Chile
Roberto Basili	University of Rome, Tor Vergata, Italy
Carlos Castillo	Yahoo! Research, Spain
Edgar Chávez	Universidad Michoacana, Mexico
Maxime Crochemore	King's College London, UK, and Paris-Est University, France
Paolo Ferragina	University of Pisa, Italy
Raffaele Giancarlo	University of Palermo, Italy
Costas Iliopoulos	King's College London, UK
Noriko Kando	National Institute of Informatics, Japan
Kimmo Kettunen	Kymenlaakso University of Applied Sciences, Finland
Dong Kyue Kim	Hanyang University, Korea
Mounia Lalmas	University of Glasgow, UK

Gad M. Landau	University of Haifa, Israel, and Polytechnic Institute of NYU, USA
Edgar Meij	University of Amsterdam, The Netherlands
Alistair Moffat	University of Melbourne, Australia
Veli Mäkinen	University of Helsinki, Finland
Gonzalo Navarro	University of Chile, Chile
Fredrik Olsson	Swedish Institute of Computer Science, Sweden
Marco Pennacchiotti	Yahoo! Inc., USA
Yoan J. Pinzón	National University of Colombia, Colombia
Mathieu Raffinot	LIAFA, CNRS-University Denis Diderot (Paris-7), France
Ian Ruthven	University of Strathclyde, UK
Wojciech Rytter	University of Warsaw, Poland
Marie-France Sagot	INRIA, France
Tetsuya Sakai	Microsoft Research Asia, China
Ayumi Shinohara	Tohoku University, Japan
Borkur Sigurbjornsson	Yahoo! Research, Spain
William F. Smyth	McMaster University, Canada, and Curtin University, Australia
Ian Soboroff	National Institute of Standards and Technology, USA
Wing-Kin Sung	National University of Singapore, Singapore
Masayuki Takeda	Kyushu University, Japan
Jeffrey Vitter	Texas A&M University, USA
Nivio Ziviani	Federal University of Minas Gerais, Brazil

Additional Reviewers

Cyril Allauzen
Humberto Almeida
Pavlos Antoniou
Diego Arroyuelo
Claudine Badue
Hideo Bannai
Guillaume Blin
Fabiano C. Botelho
Wladmir Brandao
Marc Bron
Manolis Christodoulakis
Krzysztof Ciebiera
Francisco Claude
Chiara Epifanio
Roberto Grossi
Jean-Loup Guillaume
Danny Hermelin
Bouke Huurnink
Shunsuke Inenaga
Takuya Kida
Shmuel Klein
Juha Kärkkäinen
Anisio Lacerda
Vincent Lacroix
Shir Landau
Avivit Levy
Sabrina Mantaci
Juan Mendivelso
Guilherme Vale Menezes
Spiros Michalakopoulos
Mirella M. Moro
Kazuyuki Narisawa

Laurence Park
Marcin Piatkowski
Solon Pissis
Benjamin Piwowarski
Simon Puglisi
Jakub Radoszewski
Stéphane Raux
Hiroshi Sakamoto
Riva Shalom
Jouni Sirén
Wilson Soto
Torsten Suel
German Tischler
Manos Tsagkias
Dekel Tsur
Oscar Täckström
Rossano Venturini
Anh Vo
Niko Välimäki
William Webber
Oren Weimann
Xiao Yang

SPIRE Steering Committee

Amihood Amir	Bar-Ilan University, Israel, and Johns Hopkins University, USA
Ricardo Baeza-Yates (Coordinator)	Yahoo! Research, Spain and Chile, and CWR/DCC, University of Chile, Chile
Fabio Crestani	University of Lugano, Switzerland
Paolo Ferragina	University of Pisa, Italy
Alistair Moffat	University of Melbourne, Australia
Berthier Ribeiro-Neto	Federal University of Minas Gerais, Brazil
Mark Sanderson	University of Sheffield, UK
Andrew Turpin	RMIT University, Australia
Nivio Ziviani	Federal University of Minas Gerais, Brazil

SPIRE Venues

SPIRE 2009 was the 16th edition of the Symposium on String Processing and Information Retrieval. The four first events concentrated mainly on string processing and were held in South America under the title South American Workshop on String Processing (WSP) in 1993, 1995, 1996 and 1997. WSP was transformed into SPIRE in 1998, when the scope of the conference was broadened to include also the area of information retrieval.

1993	Belo Horizonte, Brazil
1995	Valparaíso, Chile
1996	Recife, Brazil
1997	Valparaíso, Chile
1998	Santa Cruz de la Sierra, Bolivia
1999	Cancún, Mexico
2000	A Coruña, Spain
2001	Laguna de San Rafael, Chile
2002	Lisbon, Portugal
2003	Manaus, Brazil
2004	Padova, Italy
2005	Buenos Aires, Argentina
2006	Glasgow, UK
2007	Santiago, Chile
2008	Melbourne, Australia
2009	Saariselkä, Finland

Table of Contents

Algorithms on Trees

Range Quantile Queries: Another Virtue of Wavelet Trees 1
Travis Gagie, Simon J. Puglisi, and Andrew Turpin

Constant Factor Approximation of Edit Distance of Bounded Height Unordered Trees 7
Daiji Fukagawa, Tatsuya Akutsu, and Atsuhiro Takasu

k^2-Trees for Compact Web Graph Representation 18
Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro

On-Line Construction of Parameterized Suffix Trees 31
Taehyung Lee, Joong Chae Na, and Kunsoo Park

Compressed Indexes

Succinct Text Indexing with Wildcards 39
Alan Tam, Edward Wu, Tak-Wah Lam, and Siu-Ming Yiu

A Compressed Enhanced Suffix Array Supporting Fast String Matching 51
Enno Ohlebusch and Simon Gog

Compressed Suffix Arrays for Massive Data 63
Jouni Sirén

On Entropy-Compressed Text Indexing in External Memory 75
Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter

Compression

A Linear-Time Burrows-Wheeler Transform Using Induced Sorting 90
Daisuke Okanohara and Kunihiko Sadakane

Novel and Generalized Sort-Based Transform for Lossless Data Compression 102
Kazumasa Inagaki, Yoshihiro Tomizawa, and Hidetoshi Yokoo

A Two-Level Structure for Compressing Aligned Bitexts 114
Joaquín Adiego, Nieves R. Brisaboa, Miguel A. Martínez-Prieto, and Felipe Sánchez-Martínez

Directly Addressable Variable-Length Codes 122
Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro

Content Analysis

Identifying the Intent of a User Query Using Support Vector Machines 131
Marcelo Mendoza and Juan Zamora

Syntactic Query Models for Restatement Retrieval 143
Niranjan Balasubramanian and James Allan

Use of Co-occurrences for Temporal Expressions Annotation 156
Olga Craveiro, Joaquim Macedo, and Henrique Madeira

On-Demand Associative Cross-Language Information Retrieval 165
André Pinto Geraldo, Viviane P. Moreira, and Marcos A. Gonçalves

A Comparison of Data-Driven Automatic Syllabification Methods 174
Connie R. Adsett and Yannick Marchand

Indexing

Efficient Index for Retrieving Top-k Most Frequent Documents 182
Wing-Kai Hon, Rahul Shah, and Shih-Bin Wu

Fast Single-Pass Construction of a Half-Inverted Index 194
Marjan Celikik and Hannah Bast

Two-Dimensional Distributed Inverted Files 206
Esteban Feuerstein, Mauricio Marin, Michel Mizrahi, Veronica Gil-Costa, and Ricardo Baeza-Yates

Indexing Variable Length Substrings for Exact and Approximate Matching 214
Gonzalo Navarro and Leena Salmela

String Algorithms and Bioinformatics

Expectation of Strings with Mismatches under Markov Chain Distribution 222
Cinzia Pizzi and Mauro Bianco

Consensus Optimizing Both Distance Sum and Radius 234
Amihood Amir, Gad M. Landau, Joong Chae Na, Heejin Park, Kunsoo Park, and Jeong Seop Sim

Faster Algorithms for Sampling and Counting Biological Sequences 243
Christina Boucher

String Algorithms and Theory I

Towards a Theory of Patches 254
Amihood Amir and Haim Parienty

The Frequent Items Problem, under Polynomial Decay, in the Streaming Model 266
Guy Feigenblat, Ofra Itzhaki, and Ely Porat

Improved Approximation Results on the Shortest Common Supersequence Problem 277
Zvi Gotthilf and Moshe Lewenstein

String Algorithms and Theory II

Set Intersection and Sequence Matching 285
Ariel Shiftan and Ely Porat

Generalised Matching 295
Raphael Clifford, Aram W. Harrow, Alexandru Popa, and Benjamin Sach

Practical Algorithms for the Longest Common Extension Problem 302
Lucian Ilie and Liviu Tinta

Using and Understanding Usage

A Last-Resort Semantic Cache for Web Queries 310
Flavio Ferrarotti, Mauricio Marin, and Marcelo Mendoza

A Task-Based Evaluation of an Aggregated Search Interface 322
Shanu Sushmita, Hideo Joho, and Mounia Lalmas

Efficient Language-Independent Retrieval of Printed Documents without OCR 334
Walid Magdy, Kareem Darwish, and Motaz El-Saban

Sketching Algorithms for Approximating Rank Correlations in Collaborative Filtering Systems 344
Yoram Bachrach, Ralf Herbrich, and Ely Porat

Author Index 353

Range Quantile Queries: Another Virtue of Wavelet Trees*

Travis Gagie[1], Simon J. Puglisi[2,**], and Andrew Turpin[2]

[1] Research Group for Combinatorial Algorithms in Bioinformatics,
Bielefeld University, Germany
travis.gagie@gmail.com

[2] School of Computer Science and Information Technology,
Royal Melbourne Institute of Technology, Australia
{simon.puglisi,andrew.turpin}@rmit.edu.au

Abstract. We show how to use a balanced wavelet tree as a data structure that stores a list of numbers and supports efficient *range quantile queries*. A range quantile query takes a rank and the endpoints of a sublist and returns the number with that rank in that sublist. For example, if the rank is half the sublist's length, then the query returns the sublist's median. We also show how these queries can be used to support space-efficient *coloured range reporting* and *document listing*.

1 Introduction

If we are given a list of the closing prices of a stock for the past n days and asked to find the kth lowest price, then we can do so in $\mathcal{O}(n)$ time [1]. We can also preprocess the list in $\mathcal{O}(n \log n)$ time and store it in $\mathcal{O}(n)$ words such that, given k later, we can find the answer in $\mathcal{O}(1)$ time: we simply sort the list. However, we might also later face *range quantile queries*, which have the form "what was the kth lowest price in the interval between the ℓth and the rth days?". Of course, we could precompute the answers to all such queries, but storing them would take $\Omega(n^3 \log n)$ bits of space. In this paper we show how to use a balanced wavelet tree to store the list in $\mathcal{O}(n)$ words such that we can answer range quantile queries in $\mathcal{O}(\log \sigma)$ time, where σ is the number of distinct items in the entire list.

We know of no previous work on quantile queries[1], but several authors have written about *range median queries*, the special case in which k is half the length of the interval between ℓ and r. Krizanc, Morin and Smid [11] introduced the problem of preprocessing for median queries and gave four solutions, three

* This work was supported by the Sofja Kovalevskaja Award from the Alexander von Humboldt Foundation and the German Federal Ministry of Education and Research and by the Australian Research Council.

** Corresponding Author.

[1] Henceforth, for brevity, we will use "quantile query" to mean "range quantile query", and similarly with other types of range queries.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 1–6, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Table 1. Bounds for range median queries

	space (words)	time	restriction
Krizanc *et al.* [11]	$\mathcal{O}(n)$	$\mathcal{O}(n^\epsilon)$	$\epsilon > 0$
Krizanc *et al.* [11]	$\mathcal{O}(n \log_b n)$	$\mathcal{O}\big(b \log^2 n / \log b\big)$	$2 \leq b \leq n$
Krizanc *et al.* [11]	$\mathcal{O}\big(n \log^2 n / \log \log n\big)$	$\mathcal{O}(\log n)$	
Petersen and Grabowski [16]	$\mathcal{O}\big(n^2 (\log \log n)^2 / \log^2 n\big)$	$\mathcal{O}(1)$	
Theorem 1	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	

of which have worse bounds than using a balanced wavelet tree; their fourth solution involves storing $\mathcal{O}\big(n^2 \log \log n / \log n\big)$ words to answer queries in $\mathcal{O}(1)$ time. Bose, Kranakis, Morin and Tang [2] then considered approximate queries, and Har-Peled and Muthukrishnan [9] and Gfeller and Sanders [7] considered batched queries. Recently, Krizanc *et al.*'s fourth solution was superseded by one due to Petersen and Grabowski [15,16], who reduced the space bound to $\mathcal{O}\big(n^2 (\log \log n)^2 / \log^2 n\big)$ words. Table 1 shows the bounds for Krizanc *et al.*'s first three solutions, for Petersen and Grabowski's solution, and for using a balanced wavelet tree.

Har-Peled and Muthukrishnan [9] describe applications of median queries to the analysis of Web advertising logs. In the final section of this paper we show that our solution for quantile queries can be used to support *coloured range reporting*, that is, to enumerate the distinct items in a sublist. This result immediately improves Välimäki and Mäkinen's recent space-efficient solution to the *document listing problem* [13,18].

In the full version of this paper we will also discuss how to use a wavelet tree to answer range counting queries (see [12]), coloured range counting queries (returning the number of distinct elements in a range without enumerating them), and how to support updates at the cost of slowing queries down to take time proportional to the logarithm of the largest number allowed.

2 Wavelet Trees

Grossi, Gupta and Vitter [8] introduced wavelet trees for use in data compression, and Ferragina, Giancarlo and Manzini [5] showed they have myriad virtues in this respect. Wavelet trees are also important for compressed full-text indexing [14]. As we shall see, there is yet more to this intriguing data structure.

A wavelet tree T for a sequence s of length n is an ordered, strictly binary tree whose leaves are labelled with the distinct elements in s in order from left to right and whose internal nodes store binary strings. The binary string at the root contains n bits and each is set to 0 or 1 depending on whether the corresponding character of s is the label of a leaf in T's left or right subtree. For each internal node v of T, the subtree T_v rooted at v is itself a wavelet tree for the *subsequence* of s consisting of the occurrences of its leaves' labels. For example, if $s = \mathsf{a}, \mathsf{b}, \mathsf{r}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d}, \mathsf{a}, \mathsf{b}, \mathsf{r}, \mathsf{a}$ and the leaves in T's left subtree are

labelled a, b and c, then the root stores 00100010010, the left subtree is a wavelet tree for abacaaba and the right subtree is a wavelet tree for rdr. The important properties of the wavelet tree for our purposes are summarized in the following lemma.

Theorem 1 (Grossi et al. [8]). *The wavelet tree T for a list of n elements on alphabet σ requires $n \log \sigma (1 + o(1))$ bits of space, and can be constructed in $O(n \log \sigma)$ time.*

To see why the space bound is true, consider that the binary strings' total length is the sum over the distinct elements of their frequencies times their depths, which is $\mathcal{O}(n \log \sigma)$ bits. The construction time bound is easy to see from the recursive description of the wavelet tree given above.

We note as an aside that, while investigating data structures that support rank and select queries, Mäkinen and Navarro [12] pointed out a connection between wavelet trees and a data structure due to Chazelle [3] for two-dimensional range searching on sets of points.

3 Range Quantile Queries

We now describe how the wavelet tree can be used to answer quantile queries. Let s be the list of n numbers we want to query. We build and store the wavelet tree T for s and, at each internal node v, we store a small data structure that lets us perform $\mathcal{O}(1)$-time rank queries on v's binary string. A rank query on a binary string takes a position and returns the number of 1s in the prefix that ends at that position. Jacobson [10] and later Clark [4] showed we can support $\mathcal{O}(1)$-time rank queries on a binary string with a data structure that uses a sublinear number of extra bits, beyond those needed to store the string itself. It follows that the size of this preprocessed wavelet tree remains $\mathcal{O}(n \log \sigma)$ bits.

Given k, ℓ and r and asked to find the kth smallest number in $s[\ell..r]$, we start at the root of T and consider its binary string b. We use the two rank queries $\mathrm{rank}_b(\ell - 1)$ and $\mathrm{rank}_b(r)$ to find the numbers of 0s and 1s in $b[1..\ell - 1]$ and $b[\ell..r]$. If there are more than k copies of 0 in $b[\ell..r]$, then our target is a label on one of the leaves in T's left subtree, so we set ℓ to one more than the number of 0s in $b[1..\ell - 1]$, set r to the number of 0s in $b[1..r]$, and recurse on the left subtree. Otherwise, our target is a label on one of the leaves in T's right subtree, so we subtract from k the number of 0s in $b[\ell..r]$, set ℓ to one more than the number of 1s in $b[1..\ell - 1]$, set r to the number of 1s in $b[1..r]$, and recurse on the right subtree. When we reach a leaf, we return its label. An example is given in Figure 1. Since T is balanced and we spend constant time at each node as we descend (using the rank structures), our search takes $\mathcal{O}(\log \sigma)$ time. Thus, together with Theorem 1 we have the following.

Theorem 2. *There exists a data structure of size $\mathcal{O}(n \log \sigma)$ bits which can be built in $\mathcal{O}(n \log \sigma)$ time that answers range quantile queries on $s[1..n]$ in $\mathcal{O}(\log \sigma)$ time.*

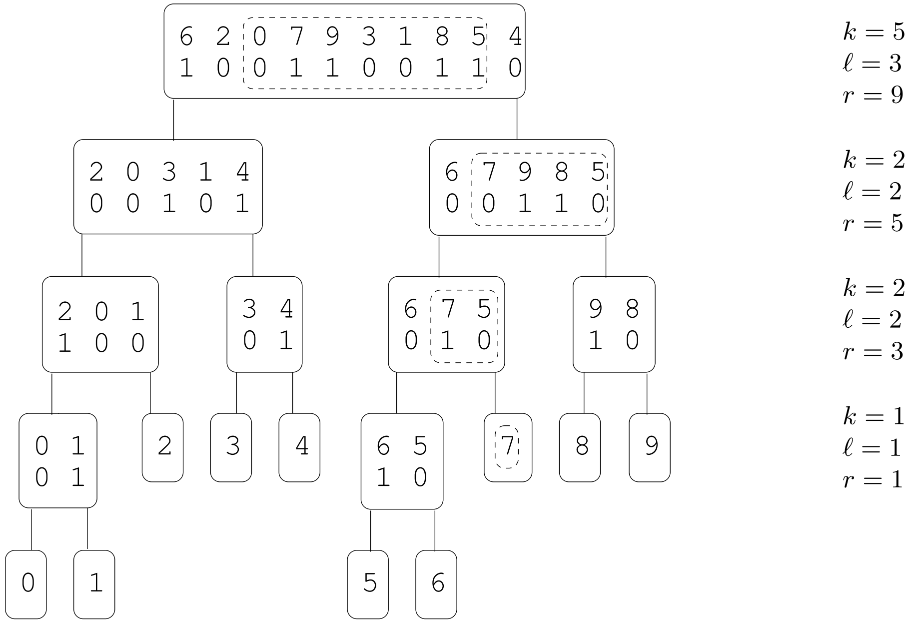


Fig. 1. A wavelet tree T (left) for $s = 6, 2, 0, 7, 9, 3, 1, 8, 5, 4$, and the values (right) the variables k, ℓ and r take on as we search for the 5th smallest element in $s[3..9]$. The dashed boxes in T show the ranges from which we recursively select.

Some comments on σ are in order at this point. Firstly, and obviously, if σ is constant, then so is our query time. If we represent the binary strings at each level of the wavelet tree with a more complicated rank/select data structure of Raman et. al [17] (instead of Clark [4], see [8,12]), the size of the wavelet tree is reduced to $nH_0(s) + \mathcal{O}(n \log \log n / \log_\sigma n)$ bits without affecting the query time, where $H_0(s)$ is the zeroth order entropy of s. Prior solutions for median queries do not make such *opportunistic* use of space.

At the other extreme, if σ is $\Omega(n)$ we can map the symbols in s to the range $[1..n]$, by first sorting the items in $\mathcal{O}(n \log n)$ time, and storing the mapping in $\mathcal{O}(n \log \sigma)$ bits of space. Preprocessing the array this way, and then using the wavelet tree approach above, allows us to match the $\Omega(n \log n)$ time lower bound for median queries [11], when the number of queries is $\mathcal{O}(n)$. This lower bound applies to any computational model which has an $\Omega(n \log n)$ time lower bound on sorting s. Still, the solution is not completely satisfying, and we leave an open question: Does an $\mathcal{O}(n \log n)$ preprocessing algorithm exist that allows quantile (or even just median) queries to be answered in $o(\log n)$ time when σ is $\Omega(n)$?

4 Application to Space Efficient Document Listing

The algorithm for quantile queries just described can, when coupled with another wavelet tree property, be used to enumerate the d distinct items in a given sublist

$s[\ell..r]$ in $\mathcal{O}(d \log \sigma)$ time as follows. Let $c_1, c_2, \ldots, c_d$ be the distinct elements in $s[\ell..r]$ and, without loss of generality, assume $c_1 < c_2 < \ldots < c_d$. Further, let m_i, $i \in 1..d$ be the number of times c_i occurs in $s[\ell..r]$. To enumerate the c_i, we begin by finding c_1, which can be achieved in $\mathcal{O}(\log \sigma)$ via a quantile query, as c_1 must be the element with rank 1 in $s[\ell..r]$. Observe now that c_2 must be the element in the range with rank $m_1 + 1$, and in general c_i is the element with rank $1 + \sum_{j=1}^{i-1} m_{j+1}$. Fortunately, each m_i can be determined in $\mathcal{O}(\log \sigma)$ time by exploiting a well known property of wavelet trees, namely, their ability to return, in $\mathcal{O}(\log \sigma)$ the number of occurrences of a symbol in a prefix of s (see [8]). Each m_i is the difference of two such queries.

The *document listing problem* [13] is a variation on the classical pattern matching problem. Instead of returning all the positions at which a pattern P occurs in the text T, we consider T as a collection of k documents (concatenated) and our task is to return the set of documents in which P occurs.

Muthukrishnan [13], who first considered the problem, gave an $\mathcal{O}(n \log n)$ bit data structure (essentially a heavily preprocessed suffix tree) that lists documents in optimal $\mathcal{O}(|P| + ndoc)$ time, where $ndoc$ is the number of documents containing P. Recently, Välimäki and Mäkinen [18] used more modern compressed and succinct data structures to reduce the space requirements of Muthukrishnan's approach at the cost of slightly increasing search to $\mathcal{O}(|P| + ndoc \log k)$ time. Their data structure consists of three pieces: the *compressed suffix array* (CSA) of T; a wavelet tree built on an auxilliary array, E (described shortly); and a succinct range minimum query data structure [6].

Central to both Muthukrishnan's and Välimäki and Mäkinen's solutions is the so-called "document array" $E[1..n]$, which is parallel to the suffix array $\mathrm{SA}[1..n]$: $E[i]$ is the document in which suffix $\mathrm{SA}[i]$ begins. Given an interval $\mathrm{SA}[i..j]$ where all the occurrences of a pattern lie, the document listing problem then reduces to enumerating the distinct items in $E[i..j]$. Without getting into too many details, Välimäki and Mäkinen use the *compressed suffix array* (CSA) of T to find the relevant sublist of E in $\mathcal{O}(|P|)$ time, and then a combination of E's wavelet tree and a range minimum query data structure [6] to enumerate the distinct items in that sublist in $\mathcal{O}(ndoc \log k)$ time. However, as we have described above, the wavelet tree of E alone is sufficient to solve this problem in the same $\mathcal{O}(ndoc \log k)$ time bound. In practice we may expect this new approach to be faster, as the avoidance of the minimum queries should reduce CPU cache misses. Also, because the wavelet tree of E is already present in [18] we have reduced the size of their data structure by $2n + o(n)$ bits, the size of the data structure for minimum queries.

Acknowledgements

Our thanks go to the three anonymous reveiwers whose helpful comments materially improved the paper, and to Meg Gagie for righting our grammar.

References

1. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. Journal of Computer and System Sciences 7, 448–461 (1973)
2. Bose, P., Kranakis, E., Morin, P., Tang, Y.: Approximate range mode and range median queries. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 377–388. Springer, Heidelberg (2005)
3. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17, 427–462 (1988)
4. Clark, D.: Compact PAT trees. PhD thesis, Waterloo University, Canada (1996)
5. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 560–571. Springer, Heidelberg (2006)
6. Fischer, J.: Efficient Data Structures for String Algorithms. PhD thesis, LMU, München (2007)
7. Gfeller, B., Sanders, P.: Towards optimal range medians. arXiv:0901.1761 (2009)
8. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the 14th Symposium on Discrete Algorithms, pp. 841–850 (2003)
9. Har-Peled, S., Muthukrishnan, S.M.: Range medians. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 503–514. Springer, Heidelberg (2008)
10. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th Symposium on Foundations of Computer Science, pp. 549–554 (1989)
11. Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nordic Journal of Computing 12, 1–17 (2005)
12. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387, 332–347 (2007)
13. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 657–666 (2002)
14. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Computing Surveys 39, Article 2 (2007)
15. Petersen, H.: Improved bounds for range mode and range median queries. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 418–423. Springer, Heidelberg (2008)
16. Petersen, H., Grabowski, S.: Range mode and range median queries in constant time and sub-quadratic space. Information Processing Letters 109, 225–228 (2009)
17. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 233–242 (2002)
18. 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)

Constant Factor Approximation of Edit Distance of Bounded Height Unordered Trees

Daiji Fukagawa[1], Tatsuya Akutsu[2], and Atsuhiro Takasu[1]

[1] National Institute of Informatics, Tokyo 101–8430, Japan
{daiji,takasu}@nii.ac.jp
[2] Bioinformatics Center, Institute for Chemical Research,
Kyoto University, Kyoto 611–0011, Japan
takutsu@kuicr.kyoto-u.ac.jp

Abstract. The edit distance problem on two unordered trees is known to be MAX SNP-hard. In this paper, we present an approximation algorithm whose approximation ratio is $2h + 2$, where we consider unit cost edit operations and h is the maximum height of the two input trees. The algorithm is based on an embedding of unit cost tree edit distance into L_1 distance. We also present an efficient implementation of the algorithm using randomized dimension reduction.

1 Introduction

The tree edit distance problem is important because of its wide range of applications which is not limited to but includes computational biology, XML databases, and image analysis [4,8,18].

For the ordered tree edit distance problem, Tai [15] first developed a polynomial time algorithm, from which several improvements followed [11,21]. Recently, Demaine et al. [6] developed an $O(n^3)$ time algorithm and showed that it is optimal under a reasonable computation model, where n is the maximum size of the input trees. Garofalakis and Kumar [7] developed an efficient method to embed ordered edit distance into L_1 normed vector space, although move operations are allowed in their definition of the edit distance.

On the other hand, the unordered edit distance problem is known to be MAX SNP-hard for trees of bounded height [20], whereas some polynomial time algorithms are known for restricted cases [9,19]. Halldórsson and Tanaka [9] gave a $2h$ approximation algorithm for the largest common subtree problem (LCST) for unordered trees of bounded height h, which was recently improved to a $1.5h$ approximation by us [3]. These results do not imply that the edit distance problem is $1.5h$-approximable, although the edit distance can be calculated from LCST. An intuitive explanation for this statement is as follows: let us consider trees each of which consists of n nodes. Assume that the input trees are very similar so that the size of LCST is large (e.g., $\mathrm{OPT}_{\mathsf{LCST}} = n - \Theta(\log n)$) and the edit distance between them is small (e.g., $\mathrm{OPT}_{\mathsf{TED}} = \Theta(\log n)$). Even if we can approximate the LCST by a small factor $\alpha (> 1)$, the corresponding edit distance becomes at

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 7–17, 2009.
© Springer-Verlag Berlin Heidelberg 2009

least $\text{APX}_{\text{TED}} \geq n - \text{APX}_{\text{LCST}} = n - (1/\alpha)\text{OPT}_{\text{LCST}} = n - (1/\alpha)(n - \Theta(\log n)) = (1 - 1/\alpha)n + \Theta(\log n) = \Theta(n)$, which may be much greater than $\text{OPT}_{\text{TED}} = \Theta(1)$. That is a motivation for us to develop an approximation algorithm for the tree edit distance problem.

In this paper, we propose an algorithm to embed the tree edit distance into the L_1 norm with worst case distortion $2h + 2$. Though similar representations have appeared in the literature [5,14,16,17], they did not prove any approximation ratios. Although the idea of embedding and its algorithm is not novel in the sense that linear (or close to linear) algorithms are already shown for the similar distances, our main interest of this paper is to guarantee an approximation ratio against the original edit distance using such a distance. For the embedding, we also show efficient representation of the vector by randomized dimension reduction. Our algorithm gives a $(2h + 2)$-approximation ratio for the edit distance problem on rooted, labeled and unordered trees, which implies a constant factor approximation for trees of bounded height. The handling of unordered trees of bounded height is important since the height of trees in XML databases tends to be low[1].

2 Preliminaries

2.1 Tree and Forest

A *forest* is a graph without cycles. A *tree* is a connected forest. For a forest F, $\mathcal{V}(F)$ denotes the set of nodes in F. In this paper, a tree T is *rooted*, that is, T has a special node called the *root* of T, denoted by $\mathsf{root}(T)$. Each connected component of a forest is a rooted tree. Every non-root node $v \in \mathcal{V}(F)$ has a *parent*, denoted by $\mathsf{parent}(v)$, and v is called a *child* of $\mathsf{parent}(v)$. Nodes sharing the same parent are *siblings*. In this paper, a forest F is *labeled*, that is, each node $v \in \mathcal{V}(F)$ has a label, denoted by $\mathsf{label}(v)$, from a finite alphabet Σ. In this paper, we assume that $|\Sigma| = O(n)$ where n is the number of nodes that we consider.

For a forest F, we call a forest F' a *subforest* of F if $\mathcal{V}(F') \subseteq \mathcal{V}(F)$ and the ancestor-descendant relation among $\mathcal{V}(F)$ is conserved in $\mathcal{V}(F')$. A subforest is a *subtree* if it is connected. A subforest F' of F is *complete* if for any node $v \in \mathcal{V}(F)$, $\mathsf{parent}(v) \in \mathcal{V}(F')$ implies $v \in \mathcal{V}(F')$. A (complete) subforest is a (complete) subtree if it is connected. We use $F(v)$ (resp. $T(v)$) to refer to the complete subtree of a forest F (resp. of a tree T) rooted at v. For a pair of subforests F_1 and F_2 of a forest F, $F_1 \cup F_2$, $F_1 \cap F_2$, and $F_1 - F_2$ denote the subforests of F induced by $\mathcal{V}(F_1) \cup \mathcal{V}(F_2)$, $\mathcal{V}(F_1) \cap \mathcal{V}(F_2)$, and $\mathcal{V}(F_1) - \mathcal{V}(F_2) = \{v \in \mathcal{V}(F_1) \mid v \notin \mathcal{V}(F_2)\}$, respectively.

In this paper, trees and forests are *unordered*; it implies that the ordering between siblings can be permuted arbitrarily. In what follows, we call an unordered, rooted, and labeled tree simply a tree.

[1] http://www.cs.washington.edu/research/xmldatasets/

We use $|F|$ to denote the size of F (i.e., $|F| = |\mathcal{V}(F)|$). The *depth* of a node v in a tree T, which is denoted by $\mathsf{depth}(v)$, is the length of the path from the root to v. The *height* of a tree T is $\mathsf{height}(T) = \max_{v \in \mathcal{V}(T)} \mathsf{depth}(v)$. For two isomorphic forests F_1 and F_2, we write $F_1 \approx F_2$.

2.2 Tree Edit Distance and Mapping

An *edit operation of tree* T is either a deletion, an insertion, or a substitution, which are defined as follows:

Deletion: Delete a non-root node v in T making the children of v become children of the parent of v.
Insertion: The complement of deletion. Insert a node v as a child of u in T making a subset of the children of u become the children of v.
Substitution: Change the label of a node v in T.

The *unit cost edit distance* (or just *edit distance*) between T_1 and T_2 is the minimum number of edit operations which transforms T_1 into T_2. We use $\mathsf{dist}(T_1, T_2)$ to denote the edit distance between T_1 and T_2. Note that $\mathsf{dist}(\cdot, \cdot)$ is a metric.

It is known that there exists a close relationship between the edit distance and the *edit distance mapping* (or just *mapping*) [4]. A set of pairs $M \subseteq \mathcal{V}(T_1) \times \mathcal{V}(T_2)$ is called a mapping if the following conditions are satisfied for any two pairs $(v_1, w_1), (v_2, w_2) \in M$: (i) $v_1 = v_2$ iff $w_1 = w_2$, (ii) v_1 is an ancestor of v_2 iff w_1 is an ancestor of w_2. For a mapping M and a pair of nodes $v_i \in \mathcal{V}(T_i)$, $i = 1, 2$, we denote $M(v_1) = v_2$ and $v_1 = M^{-1}(v_2)$ if $(v_1, v_2) \in M$.

3 A $(2h+2)$-Approximation Algorithm for Trees of Height h

The algorithm is quite simple. We construct a feature vector from each input tree and compute the L_1 distance between two feature vectors (see Fig. 1). Let h denote the maximum height of input trees, i.e., $h = \max_{i=1,2}\{\mathsf{height}(T_i)\}$.

3.1 Feature Vector for Trees

For a pair of trees t and T, $\#(T, t)$ denotes the number of $T(v)$'s isomorphic to t; in other words, $\#(T, t) = |\{v \in \mathcal{V}(T) \mid T(v) \approx t\}|$. Then, we consider the *feature vector* $\boldsymbol{\phi}(T)$ for a tree T which is defined by $\boldsymbol{\phi}(T) = (\phi_t(T))_{t \in \mathcal{T}}$ and $\phi_t(T) = \#(T, t)$ for a set $\mathcal{T} = \{t_1, t_2, \ldots\}$ of trees that we consider. Though we may consider feature vectors in infinite dimensions (i.e., we may consider all possible trees t), it is enough to consider feature vectors in $(|T_1| + |T_2|)$-dimensions (i.e., each coordinate corresponds to a complete subtree in T_1 or T_2) for the case of approximating the edit distance between T_1 and T_2.

Let $\mathbb{N}$ denote the set of non-negative integers and let $\mathbb{N}^{\mathcal{T}}$ denote the set of all possible functions from $\mathcal{T}$ to $\mathbb{N}$.

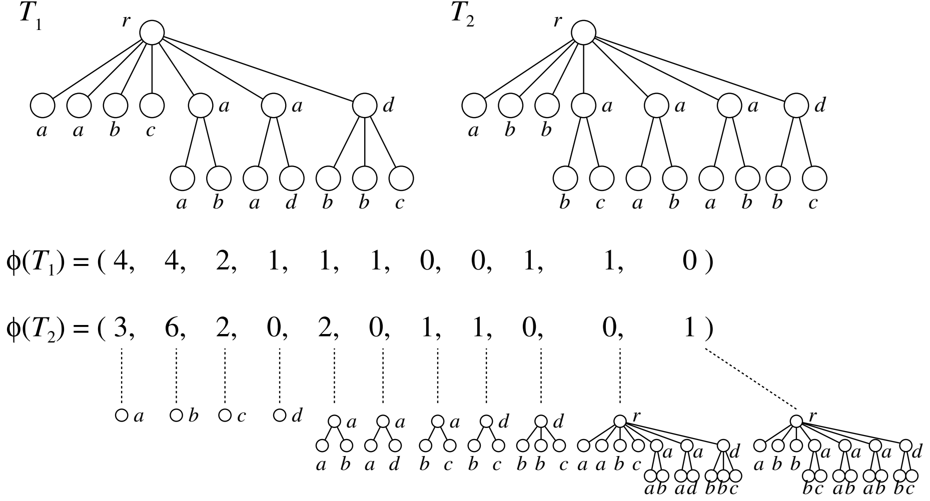


Fig. 1. Feature vectors for two trees. Only coordinates whose values are positive for at least one of T_1 and T_2 are shown.

Definition 1 (Feature vector for a tree). *For a tree T, the* feature vector *of T is the vector $\boldsymbol{\phi}(T) \in \mathbb{N}^{\mathcal{T}}$ which satisfies $\phi_t(T) = \#(T, t)$ for any $t \in \mathcal{T}$.*

In what follows, we assume that $\mathcal{T}$ is a set which covers all complete subtrees of the input trees.

The distance function $d_\phi(T_1, T_2) = \|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1$, which is naturally defined by the feature vector, is equivalent to the bottom-up distance introduced by Valiente [16]. As shown in that paper, the corresponding *bottom-up mapping* is a special case of the edit distance mapping and the time complexity of computing the bottom-up distance is linear in the number of nodes. However, he did not showed any approximation ratio of bottom-up distance against the original tree edit distance, which is the main interest of this paper.

3.2 Approximate Tree Edit Distance with Feature Vector

In this section, we prove that given two trees T_1 and T_2, the L_1 norm of their feature vectors $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1$ approximates $\mathsf{dist}(T_1, T_2)$ with distortion $2h + 2$. Recall that the L_1 norm $\|\boldsymbol{x}\|_1$ of a real vector $\boldsymbol{x} = (x_i)_{i \in X}$ in $\mathbb{R}^X$ is given by $\|\boldsymbol{x}\|_1 = \sum_{i \in X} |x_i|$.

Lemma 1. $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1 \leq (2h + 2) \cdot \mathsf{dist}(T_1, T_2)$ *holds.*

Proof. Let $e_1, \ldots, e_m$ denote an optimal sequence of edit operations which converts T_1 into T_2, and let $T_1^0 (= T_1), T_1^1, \ldots, T_1^m (= T_2)$ be the sequence of intermediate trees. That is, for $j = 1, 2, \ldots, m$, the tree T_1^j is obtained by applying an edit operation e_j to T_1^{j-1}. Note that $m = \mathsf{dist}(T_1, T_2)$ since the sequence of

operations is optimal. We assume without loss of generality that $\mathsf{height}(T_1^j) < h$, $j = 1, 2, \ldots, m$ by the following proposition.

Proposition 1. *Any optimal sequence of edit operations can be arranged so that all deletions precede all insertions.*

This proposition is apparent by considering the corresponding edit distance mapping (see e.g., [15]) and its cost. For an edit distance mapping M from T to T',

$$\mathsf{cost}(M) := \sum_{u \in I} \mathsf{cost}(u \to \emptyset) + \sum_{(u,v) \in M} \mathsf{cost}(u \to v) + \sum_{v \in J} \mathsf{cost}(\phi \to v) \quad (1)$$

where $I = \mathcal{V}(T) - M^{-1}(\mathcal{V}(T'))$ and $J = \mathcal{V}(T') - M(\mathcal{V}(T))$. This is equal to the cost of edit sequence in which we delete the nodes in I, change the labels of nodes $(u, v) \in M$ if it is necessary, and insert the nodes J.

Then, we prove the following equation.

$$\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1 \leq \sum_{j=1}^{m} \|\boldsymbol{\phi}(T_1^{j-1}) - \boldsymbol{\phi}(T_1^j)\|_1 \leq (2h + 2) \cdot \mathsf{dist}(T_1, T_2).$$

Since the first inequality clearly holds by the definition of T_1^j, it is sufficient to prove that $\|\boldsymbol{\phi}(T_1^{j-1}) - \boldsymbol{\phi}(T_1^j)\|_1 \leq 2h + 2$ holds for $j = 1, \ldots, m$. Let $\mathsf{anc}(v)$ denote the set of ancestors of v, including v itself. Note that $|\mathsf{anc}(v)| = \mathsf{depth}(v) + 1 \leq h + 1$.

Substitution: Let v and v' respectively be the nodes in T_1^{j-1} and T_1^j which are relevant to the substitution. Then, ϕ_t may decrease by one for each subtree $t = T_1^{j-1}(x)$, $x \in \mathsf{anc}(v)$ and $\phi_{t'}$ may increase by one for each subtree $t' = T_1^j(x')$, $x' \in \mathsf{anc}(v')$. Therefore, $\|\boldsymbol{\phi}(T_1^{j-1}) - \boldsymbol{\phi}(T_1^j)\|_1 \leq |\mathsf{anc}(v)| + |\mathsf{anc}(v')| \leq 2h + 2$ holds.

Deletion: Let v be the deleted node in T_1^{j-1}. For $w = \mathsf{parent}(v)$, let w' in T_1^j correspond to w. Then, the feature vector may decrease for the subtrees $t = T_1^{j-1}(x)$, $x \in \mathsf{anc}(v)$ and may increase for $t = T_1^j(x')$, $x' \in \mathsf{anc}(w')$. Therefore, $\|\boldsymbol{\phi}(T_1^{j-1}) - \boldsymbol{\phi}(T_1^j)\|_1 \leq |\mathsf{anc}(v)| + |\mathsf{anc}(w')| \leq 2|\mathsf{anc}(v)| \leq 2h + 2$ holds.

Insertion: Since insertion is the complement of deletion, T_1^{j-1} is obtained by deleting a node in T_1^j. Therefore, $\|\boldsymbol{\phi}(T_1^{j-1}) - \boldsymbol{\phi}(T_1^j)\|_1 \leq 2h + 2$ holds.

Since e_j is one of the above, it holds that $\|\boldsymbol{\phi}(T_1^{j-1}) - \boldsymbol{\phi}(T_1^j)\|_1 \leq 2h + 2$. □

Note that the ratio $2h + 2$ is tight; consider non-branching trees (that is, each non-leaf node has exactly one child) T_1 and T_2 of the same height h in which nodes are all labeled $\mathtt{a}$ except that the leaf of T_2 is labeled $\mathtt{b}$. Then we have $\mathsf{dist}(T_1, T_2) = 1$ and $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1 = 2h + 2$.

By Lemma 1, we have shown the upper bound of the embedded distance. Next, we show that the lower bound of the embedded distance exactly matches the edit distance. Before showing the lower bound, we show that our embedding

approximates the tree edit distance by computing the largest common complete subforest of the input trees. For two trees, a *common complete subforest* is a forest which is isomorphic to a complete subforest of each tree. It is obvious that the feature vectors of trees share the feature vector of a common subforest because the common subforest appears in each tree. The following lemma says that the opposite is true.

Lemma 2. *For a largest common complete subforest F of T_1 and T_2, $\phi_t(F) = \min\{\phi_t(T_1), \phi_t(T_2)\}$ holds for any $t \in \mathcal{T}$.*

Proof. Assume that there exists a tree t for which $\phi_t(F) \neq \min_{i=1,2}\{\phi_t(T_i)\}$ holds. Since F is a common complete subforest of T_1 and T_2, $\phi_t(F) \leq \phi_t(T_i)$, $i = 1, 2$ is obvious. By the assumption, we have $\phi_t(F) < \min_{i=1,2} \phi_t(T_i)$. For $i = 1, 2$, let $F_i \approx F$ be a complete subforest of T_i and let $M \subseteq \mathcal{V}(F_1) \times \mathcal{V}(F_2)$ be the corresponding mapping between nodes in F_1 and F_2. That is, $\{M(v) \mid v \in \mathcal{V}(F_1)\} = \mathcal{V}(F_2)$. Since $\phi_t(F) < \phi_t(T_i)$ $i = 1, 2$, T_i must include a complete subtree $T_i(w_i) \approx t$ for some node $w_i \in \mathcal{V}(T_i - F_i)$.

Now we prove that there is a common complete subforest F'' between T_1 and T_2 and that $|F''| > |F|$, which contradicts the assumption that F is the largest.

Let X_i be the complete subforest induced by $\mathcal{V}(F_i \cap T_i(w_i))$ (see Fig. 2). In what follows, we simply write $M(X_1)$ and $M^{-1}(X_2)$ to denote the subtrees induced by $\{M(v) \mid v \in \mathcal{V}(X_1)\}$ and $\{M^{-1}(v) \mid v \in \mathcal{V}(X_2)\}$, respectively. Let us divide F_i into four disjoint complete subforests of T_i:

$$\begin{cases} P_1 = X_1 - M^{-1}(X_2), \\ Q_1 = X_1 \cap M^{-1}(X_2), \\ R_1 = M^{-1}(X_2) - X_1, \\ F_1' = F_1 - X_1 - M^{-1}(X_2), \end{cases} \begin{cases} P_2 = M(X_1) - X_2, \\ Q_2 = M(X_1) \cap X_2, \\ R_2 = X_2 - M(X_1), \\ F_2' = F_2 - X_2 - M(X_1). \end{cases},$$

though some of them may be empty. Then, the mapping M satisfies $M(P_1) = P_2$, $M(Q_1) = Q_2$, and $M(R_1) = R_2$, where $M(X)$ denotes the subforest of T_2 induced by $\{M(v) \in \mathcal{V}(F_2) \mid v \in \mathcal{V}(X) \subseteq \mathcal{V}(F_1)\}$ for a subforest X of T_1. Therefore we have $M(F_1') = F_2'$ and $F_1' \approx F_2'$.

Furthermore, we expand construct larger subforests $F_i'' = F_i' \cup T_i(w_i)$, $i = 1, 2$. Since F_i' is a subforest of $F_i - X_i = F_i - (F_i \cap T_i(w_i)) = F_i - T_i(w_i)$, F_i' and $T_i(w_i)$ do not intersect. Thus F_1'' and F_2'' are isomorphic.

Finally, we prove that $|F_1''| = |F_2''| > |F|$. We assume without loss of generality that $|X_1 \cup M^{-1}(X_2)| \leq |T_1(w_1)|$ since we can modify the mapping M between F_1 and F_2 so that the mapping between their subforests X_1 and X_2 is *maximal*, that is, M maps all the component subtrees which are common between X_1 and X_2. Since F_1 includes $X_1 \cup M^{-1}(X_2)$ and $w_1 \in \mathcal{V}(F_1)$, the inequation is strict, that is, $|X_1 \cup M^{-1}(X_2)| < |T_1(w_1)|$. Therefore, we have $|F_1''| = |F_1| - |X_1 \cup M^{-1}(X_2)| + |T_1(w_1)| > |F_1| = |F|$ and the lemma follows. □

Lemma 3. *For any pair of trees T_1 and T_2, $\mathsf{dist}(T_1, T_2) \leq \|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1$.*

Proof. We assume without loss of generality that T_1 and T_2 are not isomorphic, since the lemma is obvious if they are isomorphic. Let F be a largest

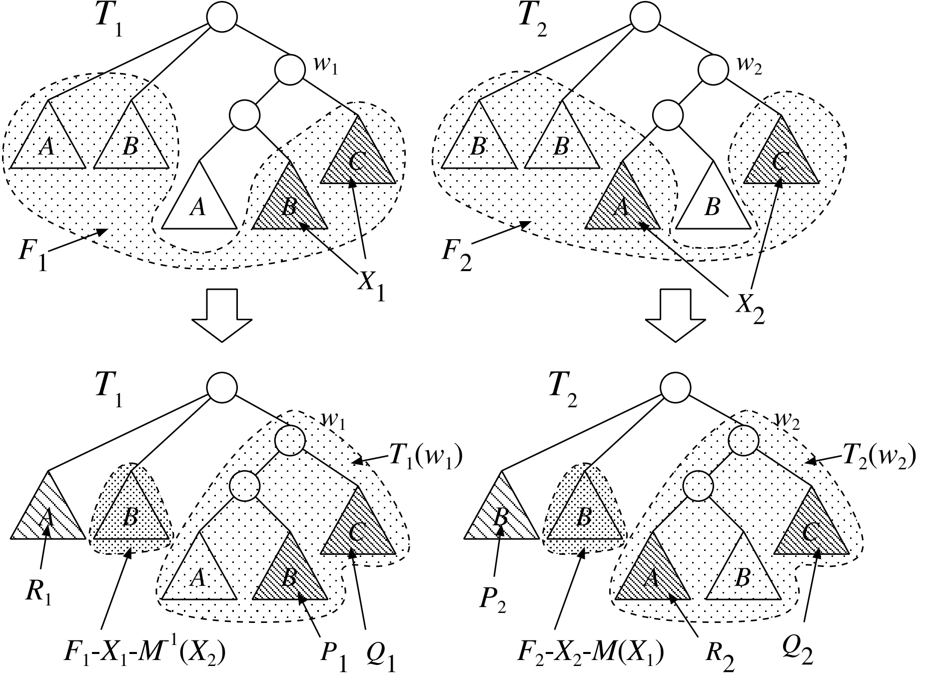


Fig. 2. An example for the proof of Lemma 2. In the upper row, the common complete subforest F of T_1 and T_2 (shown within dashed lines) consists of four components of complete subtrees $\{A, B, B, C\}$. The two complete subtrees $T_1(w_1)$ and $T_2(w_2)$ are isomorphic. For $i = 1, 2$, $T_i(w_i)$ and F_i share complete subtrees X_i (shown by shaded triangles). In the lower row, modified common subforests are shown (using dashed lines). The union of the modified common subforest $F_1' = F_1 - X_1 - M^{-1}(X_2)$ and the subtree $T_1(w_1)$ is strictly larger than F_1.

common complete subforest of T_1 and T_2. Then, we can edit T_1 to become isomorphic to T_2 by changing the label of $\mathsf{root}(T_1)$ into that of $\mathsf{root}(T_2)$, deleting $|T_1| - |F| - 1$ non-root nodes, and inserting $|T_2| - |F| - 1$ non-root nodes. Therefore, we have $\mathsf{dist}(T_1, T_2) \leq \sum_{i=1,2}(|T_i| - |F| - 1) + 1$. On the other hand, $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1 = \sum_{i=1,2}\sum_{t\in\mathcal{T}}(\phi_t(T_i) - \min\{\phi_t(T_1), \phi_t(T_2)\})$. Recall that $\|\boldsymbol{\phi}(T)\|_1 = |T|$ by the definition and $\sum_t \min\{\phi_t(T_1), \phi_t(T_2)\} = |F|$ by Lemma 2. Then, we have $\mathsf{dist}(T_1, T_2) \leq \sum_{i=1,2}|T_i| - 2|F| = \|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1$. □

Combining Lemma 1 and Lemma 3, we have:

Theorem 1. *For rooted, unordered, and labeled trees T_1 and T_2 of maximum height h, it holds that*

$$\frac{1}{2h+2}\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1 \leq \mathsf{dist}(T_1, T_2) \leq \|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1.$$

Procedure COMPUTESIGNATURE(F)
Let $L_k = \{v \mid \mathsf{height}(F(v)) = k\}$, $k = 0, 1, \ldots, h$;
Assign an integer $\sigma(v) \in [0, |\Sigma|)$ to each node $v \in L_0$ according to their labels ;
for $k = 1, \ldots, h$ **do**
For each node $v \in L_k$, let $\xi(v)$ denote the descending list of integers $\{\sigma(w) \mid w \in \mathsf{children}(v)\}$;
Sort L_k in lexicographical ascending order of $(\mathsf{label}(v), \xi(v))$, $v \in L_k$;
Let $m = \max_{v \in L_{k-1}} \sigma(v)$;
Assign an integer $\sigma(v) > m$ to each $v \in L_k$ so that $\sigma(v) = \sigma(v')$ iff $(\mathsf{label}(v), \xi(v)) = (\mathsf{label}(v'), \xi(v'))$;

Fig. 3. Algorithm to compute signatures $\sigma(v)$ for each complete subtree $F(v)$ in a forest F

Corollary 1. *Let T_1 and T_2 be trees of maximum height h. Then, $\mathsf{dist}(T_1, T_2)$ can be approximated within a factor of $2h + 2$ in $O(n \log n)$ time, where n is the number of nodes that we consider, that is, $n = |T_1| + |T_2|$.*

Note that computing $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1$ is usually much more efficient than computing $\mathsf{dist}(T_1, T_2)$. Computing $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1$ can be done efficiently by computing the *signatures* (see Section 4.1) of all complete subtrees T_1 and T_2. In Fig. 3, we show an algorithm to compute the signatures of all complete subtrees in a given forest F, which is similar to the well-known isomorphism test algorithm [1]. The time cost of our algorithm is $O(|F| \log(|F| + |\Sigma|))$ (or linear time in the RAM model)[2], although computing $\mathsf{dist}(T_1, T_2)$ is MAX SNP-hard for unordered trees and only an $O(n^3)$ algorithm is known even for ordered trees, where $n = \max_{i=1,2} |T_i|$. In the next section, we define the signature of trees and discuss its merits and drawbacks.

4 Tree Signature and Fingerprint

4.1 Coding of Trees

In this section, we discuss efficient data structures for the feature vectors of trees.

Recall that $\boldsymbol{\phi}(T) \in \mathbb{N}^{\mathcal{T}}$ where $\mathcal{T}$ is the set of all possible trees. Although $\mathcal{T}$ is infinite, the feature vector $\boldsymbol{\phi}(T)$ is so sparse that at most $|T|$ elements of $\boldsymbol{\phi}(T)$ become non-zero and we can use, for example, a set of pairs $\alpha(T) = \{(\mathsf{code}(t), \phi_t(T)) \mid \phi_t(T) > 0\}$ where $\mathsf{code}(t)$ is a bit-string which encodes a tree t.

For example, $\mathsf{code}(t)$ may be the *Euler string* [2,11] of a canonically ordered version of t (which can be obtained by using the algorithm in Fig. 3). In this case, the size of $\mathsf{code}(t)$ becomes $O(|t| \log |\Sigma|)$. This is optimal in the sense that

[2] Our algorithm is essentially the same as the $O(\Lambda |F| \log |F|)$ time algorithm by Vishwanathan and Smola [17] where they consider another setting in which a label of a node is a string of length at most Λ. Even in their setting, our algorithm works after a preprocess to encode strings by integers at most $|F|$, which does not affect the order of running-time.

there are at most $C_n|\Sigma|^n = O(4^n|\Sigma|^n)$ distinct labeled unordered trees of size n, where C_n denotes the Catalan number (see e.g., [12]). We further reduce the size of $\mathsf{code}(t)$ of a tree t by using hashing functions — for example, the *signature* of t. For a forest F, the signature of a complete subtree of F is an integer at most $O(|\Sigma| + |F|)$, which may be dealt with as one word in the RAM model. Furthermore, computing all signatures in a forest F is done efficiently by the algorithm shown in Fig. 3.

4.2 Karp-Rabin Fingerprint for Trees

In some applications, we want to deal with huge number of trees. In those cases, it is hard to apply the signature-based technique since the signature $\sigma(T)$ of a tree T depends on the whole set of input trees.

Let us consider a situation that we have a *text* of a set of trees $T_1, T_2, \dots$ and we want to perform nearest neighbor tasks for each query tree q which is not known beforehand. If we adopt the signature-based technique, we have to compute the signatures of all subtrees in q. If we directly use the algorithm ComputeSignature, we have to recompute almost all signatures in the text trees. To avoid running the algorithm against the whole trees, we may be able to use dictionary-based signatures, in which we maintain signature of a subtree t along with the label of $\mathsf{root}(t)$ and the list of signatures of $\mathsf{children}(\mathsf{root}(t))$. In both cases, however, we have to lookup each subtree of the query q, which causes expensive $|q|$ lookups to compute $\boldsymbol{\phi}(q)$.

Instead, we can design a randomized scheme to obliviously encode trees with small integers. By *obliviously* encoding trees, we can compute $\boldsymbol{\phi}(T)$ of a tree T by itself, that is, without the other trees.

Oblivious encoding is simply realized by using $\langle \mathsf{code}(T) \rangle$, though it may be too large. Since the size of $\mathsf{code}(T)$ is $O(|T| \log |\Sigma|)$, we can interpret it as an integer, denoted by $\langle \mathsf{code}(T) \rangle$, and can assume that $\log(\langle \mathsf{code}(T) \rangle) = O(|T| \log |\Sigma|)$. For example, a Σ-string $\boldsymbol{s} = (s_0, \dots, s_{|\boldsymbol{s}|-1}) \in \Sigma^*$ is interpreted as an integer $\langle \boldsymbol{s} \rangle = \sum_{i=0}^{|\boldsymbol{s}|} s'_i |\Sigma|^i$ where $s'_i = s_i$ for $0 \le i < |\boldsymbol{s}| - 1$ and $s'_{|\boldsymbol{s}|} = 1$.

To reduce the size of integers, we substitute $\langle \mathsf{code}(\cdot) \rangle$ by its modulo p for a randomly drawn prime number p. We call such a hashing function the *Karp-Rabin fingerprint* [10] and denote it by $\boldsymbol{\psi}^p : \mathcal{T} \to \mathbb{N}^p$, that is, $\boldsymbol{\psi}^p(T) = (\psi^p_k(T))_{k=0}^{p-1}$ and $\psi^p_k(T) = \sum_{t \in \mathcal{T}(k;p)} \#(T, t)$ where $\mathcal{T}(k;p) = \{t \in \mathcal{T} \mid \langle \mathsf{code}(t) \rangle \equiv k \bmod p\}$ for $k = 0, \dots, p-1$. By using modular arithmetic, two distinct complete subtrees may fall into the same hash; we call that a *collision*. Clearly, $\|\boldsymbol{\phi}(T_1) - \boldsymbol{\phi}(T_2)\|_1 = \|\boldsymbol{\psi}^p(T_1) - \boldsymbol{\psi}^p(T_2)\|_1$ holds if no collision occurs. If a collision occurs, the two distances may differ. In the rest of this section, let us estimate the probability that a collision occurs.

Let us assume that p is drawn uniformly at random from the set of prime numbers less than τ. Then, the following lemma is known [10,13]:

Lemma 4. *For any positive number N, the probability that p divides N is $O\big(\frac{\log N}{\tau / \log \tau}\big)$.*

Let $F = T_1 \cup T_2 \cup \cdots \cup T_m$ be the forest of concern. Let $n = \max_i |T_i|$ and $N = |F| = \sum_i |T_i|$. If we set the upper bound of the prime number p as $\tau = N^2 n^2 \log N \log |\Sigma|$, the probability of collision is at most:

$$\begin{aligned} &\Pr\big[\exists v, w \in \mathcal{V}(F)[F(v) \not\approx F(w) \wedge (\langle \mathsf{code}(F(v)) \rangle \equiv \langle \mathsf{code}(F(w)) \rangle \bmod p)]\big] \\ &\quad = N^2 \cdot O\left(\frac{\log(\max_{t \in F} \langle \mathsf{code}(t) \rangle)}{\tau / \log \tau}\right) = O\Big(\frac{N^2 n \log |\Sigma|}{\tau / \log \tau}\Big) = O(1/n) . \end{aligned} \tag{2}$$

Note that computation of $\boldsymbol{\psi}^p$ and modular arithmetic operations on p are done in time polynomial in N and in $\log |\Sigma|$. Using randomized prime number generation [13], our algorithm works in randomized polynomial time.

Theorem 2. *For a set of trees, denoted by $F = \{T_1, \ldots, T_m\}$, let us choose the upper bound of the random prime number as $\tau = N^2 n^2 \log N$, where $N = \sum_i |T_i|$ and $n = \max_i |T_i|$. Then, with probability $1 - O(1/n)$,*

$$\frac{1}{2h+2} \|\boldsymbol{\psi}^p(T_i) - \boldsymbol{\psi}^p(T_j)\|_1 \leq \mathsf{dist}(T_i, T_j) \leq \|\boldsymbol{\psi}^p(T_i) - \boldsymbol{\psi}^p(T_j)\|_1 \quad \textit{for all } i, j.$$

5 Concluding Remarks

In this paper, we have shown an $2h+2$ approximation algorithm for the tree edit distance problem for unordered trees of height at most h. The algorithm is based on embedding the tree edit distance into the L_1 norm of feature vectors. Although both the idea of the embedding and the algorithm to compute the feature vector are known in the literature, any approximation ratio had not been known for them.

Recently the simlar approximation ratio $1.5h$ is proved by the authors, however, the ratios are not compatible and the techniques used are completely different. The results complement each other, by which both problems are approximable when the height is small. This is interesting because both problems are known to be hard to approximate even if the height is constant. To develop an efficient approximation algorithm for the general tree edit distance problem is left as an open problem.

Acknowledgements

This work was partly supported by MEXT Grant-in-Aid No. 18049069 and by MEXT Grant-in-Aid No. 19650053. We would like to thank anonymous referees whose comments led the paper to significant improvements, including the accomplished tight example of Lemma 1.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston (1974)
2. Akutsu, T.: A relation between edit distance for ordered trees and edit distance for euler strings. Inf. Proc. Lett. 100, 105–109 (2006)

3. Akutsu, T., Fukagawa, D., Takasu, A.: Improved approximation of the largest common sub-tree of two unordered trees of bounded height. Inf. Proc. Lett. 109, 165–170 (2008)
4. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337, 217–239 (2005)
5. Collins, M., Duffy, N.: Convolution Kernels for Natural Language. In: Proc. NIPS, pp. 625–632 (2001)
6. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 146–157. Springer, Heidelberg (2007)
7. Garofalakis, M.N., Kumar, A.: XML stream processing using tree-edit distance embeddings. ACM Trans. Database System 30, 279–332 (2005)
8. Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Approximate XML joins. In: SIGMOD 2002 (2002)
9. Halldórsson, M.M., Tanaka, K.: Approximation and special cases of common subtrees and editing distance. In: Nagamochi, H., Suri, S., Igarashi, Y., Miyano, S., Asano, T. (eds.) ISAAC 1996. LNCS, vol. 1178, pp. 75–84. Springer, Heidelberg (1996)
10. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31, 249–260 (1987)
11. Klein, P.N.: Computing the edit-distance between unrooted ordered trees. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 91–102. Springer, Heidelberg (1998)
12. Knuth, D.E.: The Art of Computer Programming. Fascicle 4: Generating All Trees, vol. 4. Addison-Wesley Professional, Reading (2006)
13. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, New York (1995)
14. Müller-Molina, A.J., Hirata, K., Shinohara, T.: A tree distance function based on multi-sets. In: ALSIP 2008, PAKDD Workshops, pp. 90–100 (2008)
15. Tai, K.-C.: The tree-to-tree correction problem. J. ACM 26, 422–433 (1979)
16. Valiente, G.: An Efficient Bottom-Up Distance between Trees. In: Proc. Eighth Int'l Symp. String Processing Information Retrieval, pp. 212–219 (2001)
17. Vishwanathan, S.V.N., Smola, A.J.: Fast Kernels for String and Tree Matching. In: NIPS, pp. 569–576 (2002)
18. Yang, R., Kalnis, P., Tung, A.K.H.: Similarity evaluation on tree-structured data. In: SIGMOD (2005)
19. Zhang, K.: A constrained edit distance between unordered labeled trees. Algorithmica 15, 205–222 (1996)
20. Zhang, K., Jiang, T.: Some MAX SNP-hard results concerning unordered labeled trees. Inf. Proc. Lett. 49, 249–254 (1994)
21. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Computing 18, 1245–1262 (1989)

k^2-Trees for Compact Web Graph Representation*

Nieves R. Brisaboa[1], Susana Ladra[1], and Gonzalo Navarro[2]

[1] Database Lab., Univ. of A Coruña, Spain
{brisaboa,sladra}@udc.es
[2] Dept. of Computer Science, Univ. of Chile, Chile
gnavarro@dcc.uchile.cl

Abstract. This paper presents a Web graph representation based on a compact tree structure that takes advantage of large empty areas of the adjacency matrix of the graph. Our results show that our method is competitive with the best alternatives in the literature, offering a very good compression ratio (3.3–5.3 bits per link) while permitting fast navigation on the graph to obtain direct as well as reverse neighbors (2–15 microseconds per neighbor delivered). Moreover, it allows for extended functionality not usually considered in compressed graph representations.

1 Introduction

The World Wide Web structure can be regarded as a directed graph at several levels, the finest grained one being pages that point to pages. Many algorithms of interest to obtain information from the Web structure are essentially basic algorithms applied over the Web graph [11, 16].

Running typical algorithms on those huge Web graphs is always a problem. Even the simplest external memory graph algorithms, such as graph traversals, are usually non disk-friendly [24]. This has pushed several authors to consider *compressed graph representations*, which aim to offer memory-efficient graph representations that still allow fast navigation without decompressing. The aim of this research is to allow classical graph algorithms to be run in main memory over much larger graphs than those affordable with a plain representation.

The most famous representative of this trend is surely Boldi and Vigna's *WebGraph Framework* [6]. The WebGraph compression method is indeed the most successful member of a family of approaches to compress Web graphs based on their statistical properties [1, 5, 7, 20, 21, 23]. It allows fast extraction of the neighbors of a page while spending just a few bits per link (about 2 to 6, depending on the desired navigation performance). Their representation explicitly exploits Web graph properties such as: (1) the power-law distribution of indegrees and outdegrees, (2) the locality of reference, (3) the "copy property" (the set of neighbors of a page is usually very similar to that of some other page).

More recently, Claude and Navarro [10] showed that most of those properties are elegantly captured by applying Re-Pair compression [17] on the adjacency

* Funded in part (for the Spanish group) by MEC grant (TIN2006-15071-C03-03); and for the third author by Fondecyt Grant 1-080019, Chile.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 18–30, 2009.
© Springer-Verlag Berlin Heidelberg 2009

lists, and that *reverse navigation* (finding the pages that point to a given page) could be achieved by representing the output of Re-Pair using some more sophisticated data structures [9]. Reverse navigation is useful to compute several relevance ranking on pages, such as HITS, PageRank, and others. Their technique offers better space/time tradeoffs than WebGraph, that is, they offer faster navigation than WebGraph when both structures use the same space.

Asano et al. [2] achieve even less than 2 bits per link by explicitly exploiting regularity properties of the adjacency matrix of the Web graphs, but their navigation time is substantially higher, as they need to uncompress full domains in order to find the neighbors of a single page.

In this paper we also aim at exploiting the properties of the adjacency matrix, yet with a general technique to take advantage of clustering rather than a technique tailored to particular Web graphs. We introduce a compact tree representation of the matrix that not only is very efficient to represent large empty areas of the matrix, but at the same time allows efficient forward and backward navigation of the graph. An elegant feature of our solution is that it is symmetric, both navigations are carried out by similar means and achieve similar times. In addition, our proposal allows some interesting operations that are not usually present in alternative structures.

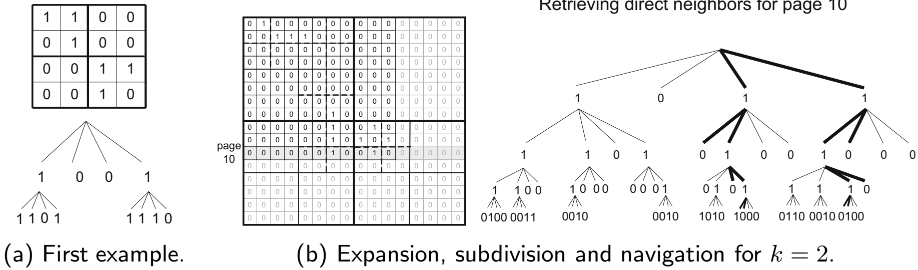
2 Our Proposal

The adjacency matrix of a Web graph of n pages is a square matrix $\{a_{ij}\}$ of $n \times n$, where each row and each column represents a Web page. Cell a_{ij} is 1 if there is a hyperlink in page i towards page j, and 0 otherwise. Page identifiers are integers, which correspond to their position in an array of alphabetically sorted URLs. This puts together the pages of the same domains, and thus locality of reference translates into closeness of page identifiers. As on average there are about 15 links per Web page, this matrix is extremely sparse. Due to locality of reference, many 1s are placed around the main diagonal (that is, page i has many pointers to pages nearby i). Due to the copy property, similar rows are common in the matrix. Finally, due to skewness of distribution, some rows and colums have many 1s, but most have very few.

We propose a compact representation of the adjacency matrix that exploits its sparseness and clustering properties. The representation is designed to compress large matrix areas with all 0s into very few bits.

We represent the adjacency matrix by a k^2-ary tree, which we call k^2*-tree*, of height $h = \lceil \log_k n \rceil$. Each node contains a single bit of data: 1 for the internal nodes and 0 for the leaves, except for the last level, where all are leaves and represent bit values of the matrix. The first level (numbered 0) corresponds to the root; its k^2 children are represented at level 1. Each child is a node and therefore it has a value 0 or 1. All internal nodes (i.e., with value 1) have exactly k^2 children, whereas leaves (with value 0 or at the last tree level) have no children.

Assume for simplicity that n is a power of k; we will soon remove this assumption. Conceptually, we start dividing the adjacency matrix following a MX-Quadtree strategy [22, Section 1.4.2.1] into k^2 submatrices of the same size, that



(a) First example. (b) Expansion, subdivision and navigation for $k = 2$.

Fig. 1. k^2-tree examples

is, k rows and k columns of submatrices of size n^2/k^2. Each of the resulting k^2 submatrices will be a child of the root node and its value will be 1 iff in the cells of the submatrix there is at least one 1. A 0 child means that the submatrix has all 0s and hence the tree decomposition ends there.

The children of a node are ordered in the tree starting with the submatrices in the first (top) row, from left to right, then the submatrices in the second row from left to right, and so on. Once the level 1, with the children of the root, has been built, the method proceeds recursively for each child with value 1, until we reach submatrices full of 0s, or we reach the cells of the original adjacency matrix. In the last level of the tree, the bits of the nodes correspond to the matrix cell values. Figure 1(a) illustrates a 2^2-tree for a 4×4 matrix.

A larger k induces a shorter tree, with fewer levels, but more children per internal node. If n is not a power of k, we conceptually extend our matrix to the right and bottom with 0s, making it of width $n' = k^{\lceil \log_k n \rceil}$. This does not cause a significant overhead as our technique is efficient to handle large areas of 0s.

Figure 1(b) shows an example of the adjacency matrix of a Web graph (we use the first 11×11 submatrix of graph CNR [6]), how it is expanded to an $n' \times n'$ matrix (n' power of $k = 2$) and its corresponding tree. Notice that its last level represents cells in the original adjacency matrix, but most cells in the original adjacency matrix are not represented in this level because, where a large area with 0s is found, it is represented by a single 0 in a smaller level of the tree.

2.1 Navigating with a k^2-Tree

To obtain the pages pointed by a specific page p, that is, to find direct neighbors of page p, we need to find the 1s in row p of the matrix. We start at the root and travel down the tree, choosing exactly k children of each node.

Example. We find the pages pointed by the first page in the example of Figure 1(a), that is, find the 1s of the first matrix row. We start at the root of the 2^2-tree and compute which children of the root overlap the first row of the matrix. These are the first two children, to which we move:

– The first child is a 1, thus it has children. To figure out which of its children are useful we repeat the same procedure. We compute in the corresponding

submatrix (the one at the top left corner) which of its children represent cells overlapping the first row of the original matrix. These are the first and the second children. They are leaf nodes and their values are 1 and 1.
- The second child of the root represents the second submatrix, but its value is 0. This means that all the cells in the adjacency matrix in this area are 0.

Thus, the Web page represented by the first row has links to itself and page 2. Figure 1(b) shows this navigation for a larger example.

Reverse neighbors. An analogous procedure retrieves the list of reverse neighbors. To obtain which pages point to page q, we need to locate which cells have a 1 in column q of the matrix. Thus, we carry out a symmetric algorithm, using columns instead of rows.

Summarizing, searching for direct or for reverse neighbors in the k^2-tree is completely symmetric. The only difference is the formula to compute the children of each node used in the next step. In either case we perform a top-down traversal of the tree. If we want to search for direct(reverse) neighbors in a k^2-tree, we go down through k children forming a row(column) inside the matrix.

3 Data Structure and Algorithms

Our data structure is essentially a compact tree of N nodes. There exist several such representations for general trees [4, 12, 14, 19], which asymptotically approach the information-theoretic minimum of $2N + o(N)$ bits. In our case, where there are only arities k^2 and 0, the information-theoretic minimum of $N + o(N)$ bits is achieved by a so-called "ultra-succinct" representation [15] for general trees. Our representation is much simpler, and close to the so-called LOUDS (level-ordered unary degree sequence) tree representation [14] (which would not achieve $N + o(N)$ bits if directly applied to our trees).

Our data structure can be regarded as a simplified variant of LOUDS for the case where arities are just k^2 and 0, which achieves the information-theoretic minimum of $N+o(N)$ bits, provides the traversal operations we require (basically move to the i-th child, although also parent is easily supported) in constant time, and is simple and practical.

3.1 Data Structure

We represent the whole adjacency matrix via the k^2-tree using two bit arrays:

T *(tree)*: stores all the bits of the k^2-tree except those in the last level. The bits are placed following a levelwise traversal: first the k^2 binary values of the children of the root node, then the values of the second level, and so on.

L *(leaves)*: stores the last level of the tree. Thus it represents the value of (some) original cells of the adjacency matrix.

We create over T an auxiliary structure that enables us to compute *rank* queries efficiently. Given an offset i inside a sequence T of bits, $rank(T, i)$ counts

the number of times the bit 1 appears in $T[1,i]$. This can be supported in constant time and fast in practice using sublinear space on top of the bit sequence [14, 18]. In practice we use an implementation that uses 5% of extra space on top of the bit sequence and provides fast queries, as well as another that uses 37.5% extra space and is much faster [13].

We do not need to perform *rank* over the bits in the last level of the tree; that is the practical reason to store them in a different bitmap (L). Thus the space overhead for *rank* is paid only over T.

Analysis. Assume the graph has n pages and m links. Each link is a 1 in the matrix, and in the worst case it induces the storage of one distinct node per level, for a total of $\lceil \log_{k^2}(n^2) \rceil$ nodes. Each such (internal) node costs k^2 bits, for a total of $k^2 m \lceil \log_k n \rceil$ bits. However, especially in the upper levels, not all the nodes in the path to each leaf can be different. In the worst case, all the nodes exist up to level $\lfloor \log_{k^2} m \rfloor$ (only since that level there can be m different internal nodes at the same level). From that level, the worst case is that each of the m paths to the leaves is unique. Thus, in the worst case, the total space is $\sum_{\ell=1}^{\lfloor \log_{k^2} m \rfloor} k^{2\ell} + k^2 m(\lceil \log_{k^2} n^2 \rceil - \lfloor \log_{k^2} m \rfloor) = k^2 m (\log_{k^2} \frac{n^2}{m} + O(1))$ bits.

This shows that, at least in a worst-case analysis, a smaller k yields less space occupancy. For $k = 2$ the space is $4m(\log_4 \frac{n^2}{m} + O(1)) = 2m \log_2 \frac{n^2}{m} + O(m)$ bits, which is asymptotically twice the information-theoretic minimum necessary to represent all the matrices of $n \times n$ with m 1s. In the experimental section we see that, on Web graphs, the space is much better than the worst case, as Web graphs are far from uniformly distributed.

Finally, the expansion of n to the next power of k can, in the horizontal direction, force the creation of at most k^ℓ new children of internal nodes at level $\ell \geq 1$ (level $\ell = 1$ is always fully expanded unless the matrix is all zeros). Each such child will cost k^2 extra bits. The total excess is $O(k^2 \cdot k^{\lceil \log_k n \rceil - 1}) = O(k^2 n)$ bits, which is usually negligible. The vertical expansion is similar.

3.2 Finding a Child of a Node

Our levelwise traversal satisfies the following property, which permits fast navigation to the i-th child of node x, $child_i(x)$ (for $0 \leq i < k^2$):

Lemma 1. *Let x be a position in T (the first position being 0) such that $T[x] = 1$. Then $child_i(x)$ is at position $rank(T,x) \cdot k^2 + i$ of $T:L$*

Proof. $T:L$ is formed by traversing the tree levelwise and appending the bits of the tree. We can likewise regard this as traversing the tree levelwise and appending the k^2 bits of the childred of the 1s found at internal tree nodes. By the time node x is found in this traversal, we have already appended k^2 bits per 1 in $T[1, x-1]$, plus the k^2 children of the root. As $T[x] = 1$, the children of x are appended at positions $rank(T,x) \cdot k^2$ to $rank(T,x) \cdot k^2 + (k^2 - 1)$.

Example. To represent the 2^2-tree of Figure 1(b), arrays T and L are:
$T = 1011\ 1101\ 0100\ 1000\ 1100\ 1000\ 0001\ 0101\ 1110$,
$L = 0100\ 0011\ 0010\ 0010\ 1010\ 1000\ 0110\ 0010\ 0100$.

Direct(n, p, q, x)	**Reverse**(n, q, p, x)
1. **If** $x \geq \|T\|$ **Then** // leaf	1. **If** $x \geq \|T\|$ **Then** // leaf
2. **If** $L[x - \|T\|] = 1$ **Then** output q	2. **If** $L[x - \|T\|] = 1$ **Then** output p
3. **Else** // internal node	3. **Else** // internal node
4. **If** $x = -1$ **or** $T[x] = 1$ **Then**	4. **If** $x = -1$ **or** $T[x] = 1$ **Then**
5. $y = rank(T, x) \cdot k^2 + k \cdot \lfloor p/(n/k) \rfloor$	5. $y = rank(T, x) \cdot k^2 + \lfloor q/(n/k) \rfloor$
6. **For** $j = 0 \ldots k - 1$ **Do**	6. **For** $j = 0 \ldots k - 1$ **Do**
7. **Direct**$(n/k, p \bmod (n/k),$ $q + (n/k) \cdot j,\ y + j)$	7. **Reverse**$(n/k, q \bmod (n/k),$ $p + (n/k) \cdot j, y + j \cdot k)$

Fig. 2. Returning direct(reverse) neighbors

In T each bit represents a node. First four bits represent nodes $0, 1, 2$ and 3, which are the children of the root. The following four bits represent the children of node 0. There are no children for node 1 because it is a 0, then the children of node 2 start at position 8 and those of node 3 start at position 12. The bit in position 4, the fifth bit of T, represents the first child of node 0, and so on.

3.3 Navigation

To find the direct(reverse) neighbors of a page $p(q)$ we need to locate which cells in row a_{p*} (column a_{*q}) of the adjacency matrix have a 1. We have already explained that these are obtained by a top-down tree traversal that chooses k out of the k^2 children of a node, and also gave the way to obtain the i-th child of a node in our representation. The only missing piece is the formula that maps global row numbers to the children number at each level.

Recall $h = \lceil \log_k n \rceil$ is the height of the tree. Then the nodes at level ℓ represent square submatrices of width $k^{h-\ell}$, and these are divided into k^2 submatrices of width $k^{h-\ell-1}$. Cell (p_ℓ, q_ℓ) at a matrix of level ℓ belongs to the submatrix at row $\lfloor p_\ell / k^{h-\ell-1} \rfloor$ and column $\lfloor q_\ell / k^{h-\ell-1} \rfloor$.

Let us call p_ℓ the relative row position of interest at level ℓ. Clearly $p_0 = p$, and row p_ℓ of the submatrix of level ℓ corresponds to children number $k \cdot \lfloor p_\ell / k^{h-\ell-1} \rfloor + j$, for $0 \leq j < k$. The relative position in those children is $p_{\ell+1} = p_\ell \bmod k^{h-\ell-1}$. Similarly, column q corresponds $q_0 = q$ and, in level ℓ, to children $j \cdot k + \lfloor q_\ell / k^{h-\ell-1} \rfloor$, for $0 \leq j < k$, with relative position $q_{\ell+1} = q_\ell \bmod k^{h-\ell-1}$.

The algorithms for extracting direct and reverse neighbors are described in Figure 2. For direct neighbors it is called **Direct**$(k^h, p, 0, -1)$, where the parameters are: current submatrix size, row of interest in current submatrix, column offset of the current submatrix in the global matrix, and the position in $T : L$ of the node to process (the initial -1 is an artifact because our trees do not represent the root node). Values T, L, and k are global. It is assumed that n is a power of k and that $rank(T, -1) = 0$. For reverse neighbors it is called **Reverse**$(k^h, q, 0, -1)$, where the parameters are the same except that the second is the column of interest and the third is the row offset of the current submatrix.

Analysis. Our navigation time has no worst-case guarantees better than $O(n)$, as a row $p-1$ full of 1s followed by p full of 0s could force a **Direct** query on p to go until the leaves across all the row, to return nothing.

However, this is unlikely. Assume the m 1s are uniformly distributed in the matrix. Then the probability that a given 1 is inside a submatrix of size $(n/k^\ell) \times (n/k^\ell)$ is $1/k^{2\ell}$. Thus, the probability of entering the children of such submatrix is (brutally) upper bounded by $m/k^{2\ell}$. We are interested in k^ℓ submatrices at each level of the tree, and therefore the total work is on average upper bounded by $m \cdot \sum_{\ell=0}^{h-1} k^\ell / k^{2\ell} = O(m)$. This can be refined because there are not m different submatrices in the first levels of the tree. Assume we enter all the $O(k^t)$ matrices of interest up to level $t = \lfloor \log_{k^2} m \rfloor$, and from then on the sum above applies. This is $O(k^t + m \cdot \sum_{\ell=t+1}^{h-1} k^\ell / k^{2\ell}) = O(k^t + m/k^t) = O(\sqrt{m})$ time. This is not the ideal $O(m/n)$ (average output size), but much better than $O(n)$ or $O(m)$.

Again, if the matrix is clustered, the average performance is indeed better than under uniform distribution: whenever a cell close to row p forces us to traverse the tree down to it, it is likely that there is a useful cell at row p as well.

3.4 Construction

Assume our input is the $n \times n$ matrix. Construction of our tree is easily carried o ut bottom-up in linear time and using the same space as the final tree. If, instead, we have an adjacency list representation of the matrix, we can still achieve the same time by setting up n cursors, one per row, so that each time we have to access a_{pq} we compare the current cursor of row p with value q.

In this case we could try to achieve time proportional to m, the number of 1s in the matrix. For this sake we could insert the 1s one by one into an initially empty tree, building the necessary part of the path from the root to the corresponding leaf. After the tree is built we can traverse it levelwise to build the final representation, or recursively to output the bits to different sequences, one per level, as before. The space could still be $O(k^2 m(1 + \log_{k^2} \frac{n^2}{m}))$, that is, proportional to the final tree size, if we used some dynamic compressed parentheses representation of trees [8]. The total time would be $O(\log m)$ per bit of the tree.

As we produce each tree level and traverse each matrix row (or adjacency list) sequentially, we can construct the tree on disk in optimal I/O time provided we have main memory to maintain $\log_k n$ disk blocks to output the tree, plus B disk blocks (B being the disk page size in bits) for reading the matrix.

4 A Hybrid Approach

As we can notice, the greater k is, the more space L needs, because even though there are fewer submatrices in the last level, they are larger. Hence we may spend k^2 bits to represent very few 1s. Notice for example that if $k = 4$ in Figure 1(b), we will store some last-level submatrices containing a unique 1, spending 15 more bits that are 0. On the contrary, when $k = 2$ we use fewer bits for that leaf level.

We can improve our structure if we use a larger k for the first levels of the tree and a small k for the last levels. This strategy takes advantage of the strong

points of both approaches. Using large values of k for the first levels, the tree is shorter, so we will be able to obtain the list of neighbors faster, as we have fewer levels to traverse. Using small values of k for the last levels we do not store too many bits for each 1 of the adjacency matrix, as the submatrices are smaller.

5 Experimental Evaluation

We ran several experiments over some Web crawls from the *WebGraph* project. Figure 3(a) gives the main characteristics of the graphs used: name (and version) of the graph, number of pages and links and the size of a plain adjacency list representation of the graphs (using 4-byte integers). The machine used in our tests is a 2GHz Intel®Xeon® (8 cores) with 16 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.4.22-15-generic SMP (64 bits). The compiler was gcc version 4.1.3 and `-O9` compiler optimizations were set. Space is measured in bits per edge (bpe), by dividing the total space of the structure by the number of edges (i.e., links) in the graph. Time results measure average CPU user time per neighbor retrieved: We compute the time to search for the neighbors of all the pages (in random order) and divide by the total number of edges in the graph.

5.1 Comparison between Different Alternatives

We first study our approach with different values of k. Figure 3(b) shows 8 different alternatives of our method over the EU graph using different values of k. All build on the *rank* structure that uses 5% of extra space [13]. The first column names the approaches as follows: $'2 \times 2'$, $'3 \times 3'$ and $'4 \times 4'$ stand for the alternatives where we subdivide the matrix into 2×2, 3×3 and 4×4 submatrices, respectively, in every level of the tree. On the other hand, we denote $'H - i'$ the hybrid approach where we use $k = 4$ up to level i of the tree, and then we use $k = 2$ for the rest of the levels. The second and third columns indicate the size, in bytes, used to store the tree T and the leaves L, respectively. The fourth column shows the space needed in main memory by the structures (e.g., including the extra space for *rank*), in bits per edge. Finally, the last two columns show the times to retrieve the direct (fifth column) and reverse (sixth) neighbors, measured in microseconds per link retrieved (μs/e). Note that, when we use a fixed k, we obtain better times when k is greater, because we are shortening the height of the tree, but the compression ratio worsens, as the space for L becomes dominant and many 0s are stored in there.

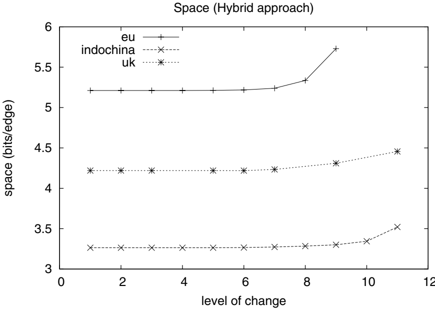
If we use a hybrid approach, we can maintain a compression ratio close to that obtained by the $'2 \times 2'$ alternative while improving the time, until we get close to the $'4 \times 4'$ alternative. The best compression is obtained for $'H - 3'$, even better than $'2 \times 2'$. Figure 3(c) shows similar results graphically, for the three larger graphs, space on the left and time to retrieve direct neighbors on the right. The space does not worsen much if we keep $k = 4$ up to a moderate level, whereas times improve consistently. A medium value, say switching to $k = 2$ at level 7, looks as a good compromise.

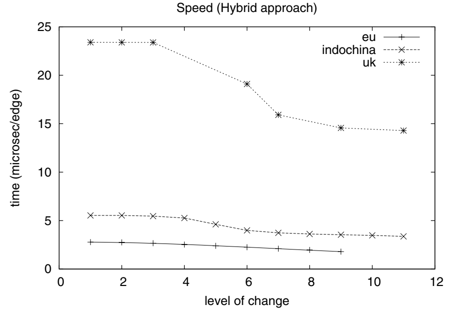
(a) Description of the graphs used.

File	Pages (millions)	Links (millions)	Size (MB)
CNR (2000)	0.325	3.216	14
EU (2005)	0.862	19.235	77
Indochina (2004)	7.414	194.109	769
UK (2002)	18.520	298.113	1,208

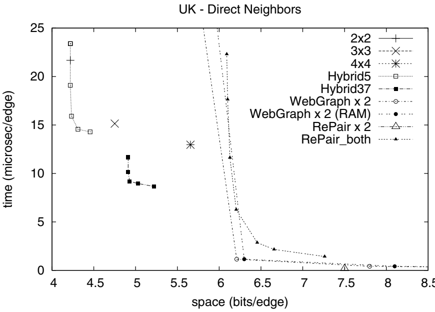
(b) Different approaches over graph EU.

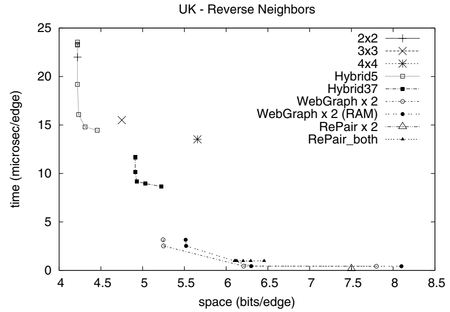
Variant	Tree (bytes)	Leaves (bytes)	Space (bpe)	Direct (μs/e)	Reverse (μs/e)
2×2	6,860,436	5,583,076	5.21076	2.56	2.47
3×3	5,368,744	9,032,928	6.02309	1.78	1.71
4×4	4,813,692	12,546,092	7.22260	1.47	1.42
$H-1$	6,860,432	5,583,100	5.21077	2.78	2.62
$H-3$	6,860,412	5,583,100	5.21076	2.67	2.49
$H-5$	6,864,404	5,583,100	5.21242	2.39	2.25
$H-7$	6,927,924	5,583,100	5.23884	2.10	1.96
$H-9$	8,107,036	5,583,100	5.72924	1.79	1.67



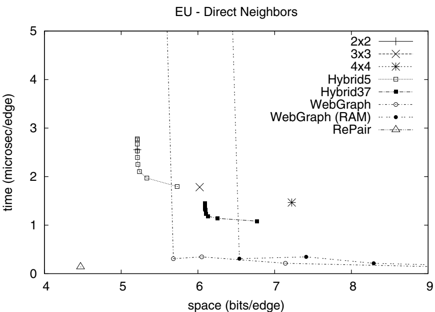


(c) Space/time behavior of the hybrid approach varying the level where we change k.





(d) Space/time to retrieve direct and reverse neighbors.



(e) Retrieving only direct neighbors.

(f) Comparison with approach *Smaller*.

Space (bpe)	*Smaller*	*Smaller* $\times$ *2*	*Hybrid5*
CNR	1.99	3.98	4.46
EU	2.78	5.56	5.21
Time (ms/p)			
CNR	2.34		0.048
EU	28.72		0.099

Fig. 3. Experimental evaluation

5.2 Comparison with Other Methods

We first compare graph representations that allow retrieving both direct and reverse neighbors. Figure 3(d) shows the space/time tradeoff for retrieving direct and reverse neighbors, over the larger graph (UK), as it is representative of the common behaviour of the other smaller graphs. We measure the average time efficiency in μs/e as before. Representations providing space/time tuning parameters appear as a line, whereas the others appear as a point.

We compare our compact representations with the proposal in [9, Chapter 7] that computes both direct and reverse neighbors (*RePair_both*), as well as the simpler representation in [10] (as improved in [9, Chapter 6], *RePair*) that retrieves just direct neigbors. In this case we represent both the graph and its transpose, in order to achieve reverse navigation as well (*RePair* $\times$ *2*). We do the same with Boldi and Vigna's technique [6] (*WebGraph*), as it also allows for direct neighbors retrieval only (we call it *WebGraph* $\times$ *2* when we add both graphs). As this technique uses less space on disk than what the process needs to run, we show in *WebGraph (RAM)* the minimum space needed to run (yet we keep the best time it achieves with sufficient RAM space). All the implementations were provided by their authors.

We include our alternatives 2×2, 3×3, 4×4, and *Hybrid5*, all of which use the slower solution for *rank* that uses just 5% of extra space [13], and *Hybrid37*, which uses the faster *rank* method that uses 37.5% extra space on top of T.

As we can see, our representations (particularly *Hybrid5* and 2×2) achieve the best compression (3.3 to 5.3 bpe, depending on the graph, 4.22 for graph UK) among all the techniques that provide direct and reverse neighbor queries. The only alternative that gets somewhat close is *RePair_both*, but it is much slower to retrieve direct neighbors. For reverse neighbors, instead, it is an interesting alternative. *Hybrid37* offers relevant tradeoffs in some cases. Finally, *WebGraph* $\times$ *2* and *RePair* $\times$ *2* offer very attractive time performance, but they need significantly more space. As explained, using less space may make the difference between being able of fitting a large Web graph in main memory or not.

If, instead, we wished only to carry out forward navigation, alternatives *RePair* and *WebGraph* become preferable in most cases. Figure 3(e), however, shows graph EU, where we still achieve significantly less space than *WebGraph*.

We also compare our proposal with the method in [2] (*Smaller*). As we do not have their code, we ran new experiments on a Pentium IV of 3.0 GHz with 4 GB of RAM, which resembles better the machine used in their experiments. We used the smaller graphs, on which they have reported experiments. Figure 3(f) shows the space and average time needed to retrieve the whole adjacency list of a page, in milliseconds per page. As, again, their representation cannot retrieve reverse neighbors, *Smaller* $\times$ *2* is *an estimation* of the space they would need to represent both the normal and transposed graphs.

Our method is orders of magnitude faster to retrieve an adjacency list, while the space is similar to *Smaller* $\times$ *2*. The difference is so large that it could be possible to be competitive even if part of our structure (e.g. L) was in secondary memory (in which case our main memory space would be similar to just *Smaller*).

6 Extended Functionality

While alternative compressed graph representations [2, 6, 10] are limited to retrieving the direct, and sometimes the reverse, neighbors of a given page, and we have compared our technique with those in these terms, we show now that our representation allows for more sophisticated forms of retrieval than extracting direct and reverse neighbors.

First, in order to determine whether a given page p points to a given page q, most compressed (and even some classical) graph representations have no choice but to extract all the neighbors of p (or a significant part of them) and see if q is in the set. We can answer such query in $O(\log_k n)$ time, by descending to exactly one child at each level of the tree. More precisely, at level ℓ we descend to child $k \cdot \lfloor p/k^{h-\ell-1} \rfloor + \lfloor q/k^{h-\ell-1} \rfloor$, if it is not a zero, and compute the relative position of cell (p, q) in the submatrix just as in Section 3.3. If we arrive at the last level and find a 1 at cell (p, q), then there is a link, otherwise there is not.

A second interesting operation is to find the direct neighbors of page p that are within a *range* of pages $[q_1, q_2]$ (similarly, the reverse neighbors of q that are within a range $[p_1, p_2]$). This is interesting, for example, to find out whether p points to a domain, or is pointed from a domain, in case we sort URLs in lexicographical order. The algorithm is similar to **Direct** and **Reverse** in Section 3.3, except that we do not enter all the children $0 \leq j < k$ of a row (or column), but only from $\lfloor q_1/k^{h-\ell-1} \rfloor \leq j \leq \lfloor q_2/k^{h-\ell-1} \rfloor$ (similarly for p_1 to p_2).

Yet a third operation of interest is to find all the links from a range of pages $[p_1, p_2]$ to another $[q_1, q_2]$. This is useful, for example, to extract all the links between two domains. The algorithm to solve this query indeed generalizes all of the others we have seen. This gives times of $O(n)$ for retrieving direct and reverse neighbors (we made a finer average-case analysis in Section 3.3), $O(p_2 - p_1 + \log_k n)$ or $O(q_2 - q_1 + \log_k n)$ for ranges of direct or reverse neighbors, and $O(\log_k n)$ for queries on single links.

7 Conclusions

We have introduced a compact representation for Web graphs that takes advantage of the sparseness and clustering of their adjacency matrix. Our representation enables efficient forward and backward navigation in the graph (a few microseconds per neighbor found) within compact space (about 3 to 5 bits per link). Our experimental results show that our technique offers an attractive space/time tradeoff compared to the state of the art. Moreover, we support queries on the graph that extend the basic forward and reverse navigation.

More exhaustive experimentation and tuning is needed to exploit the full potential of our data structure, in particular regarding the space/time tradeoffs of the hybrid approach. We also plan to research and experiment more in depth on the extended functionality supported by our representation.

The structure we have introduced can be of more general interest. It could be fruitful, for example, to generalize it to binary relations, such as the one relating

keywords with the Web pages, or more generally documents, where they appear. Then one could retrieve not only the Web pages that contain a keyword, but also the set of keywords present in a Web page, and thus have access to important summarization data without accessing the page itself. Our range search could permit searching within subcollections or subdirectories. Our structure could become a relevant alternative to the current state of the art in this direction, e.g. [3, 9]. Another example is the representation of discrete grids of points, for computational geometry applications or geographic information systems.

References

1. Adler, M., Mitzenmacher, M.: Towards compressing Web graphs. In: Proc. 11th DCC, pp. 203–212 (2001)
2. Asano, Y., Miyawaki, Y., Nishizeki, T.: Efficient compression of web graphs. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 1–11. Springer, Heidelberg (2008)
3. Barbay, J., He, M., Munro, I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proc. 18th SODA, pp. 680–689 (2007)
4. Benoit, D., Demaine, E., Munro, I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
5. Bharat, K., Broder, A., Henzinger, M., Kumar, P., Venkatasubramanian, S.: The Connectivity Server: Fast access to linkage information on the Web. In: Proc. 7th WWW, pp. 469–477 (1998)
6. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. 13th WWW, pp. 595–601. ACM Press, New York (2004)
7. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., Wiener, J.: Graph structure in the Web. Journal of Computer Networks 33(1–6), 309–320 (2000); also in Proc. 9th WWW
8. Chan, H.-L., Hon, W.-K., Lam, T.-W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2), article 21 (2007)
9. Claude, F.: Compressed data structures for Web graphs. Master's thesis, Dept. of Comp. Sci., Univ. of Chile, Advisor: Navarro, G., TR/DCC-2008-12 (August 2008)
10. Claude, F., Navarro, G.: A fast and compact Web graph representation. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 105–116. Springer, Heidelberg (2007)
11. Donato, D., Millozzi, S., Leonardi, S., Tsaparas, P.: Mining the inner structure of the Web graph. In: Proc 8th WebDB, pp. 145–150 (2005)
12. Geary, R., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theoretical Computer Science 368(3), 231–246 (2006)
13. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. 4th WEA, pp. 27–38 (2005)
14. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
15. Jansson, J., Sadakane, K., Sung, W.-K.: Ultra-succinct representation of ordered trees. In: Proc. 18th SODA, pp. 575–584 (2007)
16. Kleinberg, J., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.: The Web as a graph: Measurements, models, and methods. In: Asano, T., Imai, H., Lee, D.T., Nakano, S.-i., Tokuyama, T. (eds.) COCOON 1999. LNCS, vol. 1627, pp. 1–17. Springer, Heidelberg (1999)

17. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE 88(11), 1722–1732 (2000)
18. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
19. Munro, I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
20. Raghavan, S., Garcia-Molina, H.: Representing Web graphs. In: Proc. 19th ICDE, p. 405 (2003)
21. Randall, K., Stata, R., Wickremesinghe, R., Wiener, J.: The LINK database: Fast access to graphs of the Web. Technical Report 175, Compaq SRC (2001)
22. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc., San Francisco (2006)
23. Suel, T., Yuan, J.: Compressing the graph structure of the Web. In: Proc. 11th DCC, pp. 213–222 (2001)
24. Vitter, J.: External memory algorithms and data structures: dealing with massive data. ACM Computing Surveys 33(2), 209–271 (2001) (Version revised at 2007)

On-Line Construction of Parameterized Suffix Trees*

Taehyung Lee[1], Joong Chae Na[2,**], and Kunsoo Park[1]

[1] School of Computer Science and Engineering,
Seoul National University, Seoul 151-742, South Korea
{thlee,kpark}@theory.snu.ac.kr
[2] Department of Computer Science and Engineering,
Sejong University, Seoul 143-747, South Korea
jcna@sejong.ac.kr

Abstract. We consider on-line construction of a suffix tree for a parameterized string, where we always have the suffix tree of the input string read so far. This situation often arises from source code management systems where, for example, a source code repository is gradually increasing in its size as users commit new codes into the repository day by day. We present an on-line algorithm which constructs a parameterized suffix tree in randomized $O(n)$ time, where n is the length of the input string. Our algorithm is the first randomized linear time algorithm for the on-line construction problem.

1 Introduction

Parameterized pattern matching is a variant of traditional pattern matching in which some symbols are allowed to be consistently renamed into different symbols within a match. It was first introduced by Baker [1] and has been successfully applied to several application domains, such as software maintenance, program plagiarism detection [1,2], and RNA structural matching [3].

For general pattern matching problems, we usually preprocess a text and build an index data structure, e.g., a suffix tree or a suffix array, which enables us to answer pattern occurrence queries in time proportional to the length of the pattern but independent of the length of the text. For parameterized pattern matching, Baker adopted this idea and proposed an algorithm to answer all *pocc* parameterized matches of a pattern of length m in a text of length n in $O(m \log n + pocc)$ time, by using a variant of suffix trees, so called *parameterized suffix tree* (p-suffix tree) of the text.

* This work was supported by Korea Research Council of Fundamental Science and Technology. The ICT at Seoul National University provides research facilities for this study.

** The author was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2008-331-D00479).

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 31–38, 2009.
© Springer-Verlag Berlin Heidelberg 2009

A suffix tree of a string T is the compacted trie of all the suffixes of T [4,5,6]. A suffix tree is one of the most popular full-text index data structures, and it has been widely used in many applications for decades. There are several algorithms to construct the suffix tree of a string drawn from a constant-sized alphabet in $O(n)$ time. These include the algorithms by Weiner [4], McCreight [5] and Ukkonen [6]. We remark that all these algorithms exploit an important property of suffix trees, i.e., each node has an outgoing suffix link. For a general alphabet, where the alphabet size is not constant but some polynomial in n, however, these algorithms inherently incur $O(\log n)$ overhead due to the branching problem. To address this problem, Farach [7] proposed a divide-and-conquer approach, which constructs suffix trees in $O(n)$ time. This algorithm differs from the others above in that it is not sweep-based and it is not dependent on the existence of outgoing suffix links.

In contrast to suffix trees for strings, p-suffix trees lack the suffix link property, i.e., there could be nodes in the tree without an outgoing suffix link defined. In addition, the number of children in an internal node of the tree usually needs not to be bounded by a constant. These problems have been major hurdles in developing a linear time construction algorithm for p-suffix trees, and until recently, it seemed inevitable to bear the alphabet dependent $O(\log n)$ factor in the time complexity [1,8]. However, Cole and Hariharan [9] came up with a breakthrough that achieves a randomized $O(n)$ time construction algorithm. The algorithm uses dynamic perfect hashing and introduces additional types of nodes in the tree, so that it effectively deals with the above mentioned problems.

We consider an *on-line* construction of p-suffix trees, where we always have the p-suffix tree of the input string read so far. This situation often arises from source code management systems where, for example, a source code repository is gradually increasing in its size as users commit new codes into the repository day by day. If we want to use p-suffix trees for duplicate code detection [1], we have to rebuild the entire p-suffix tree index from scratch, every time a user commits a new code. On-line p-suffix tree construction, on the other hand, only requires to update a portion of the p-suffix tree, thus it can minimize the effort to manage the whole index system.

However, the above mentioned algorithms do not support on-line construction since they are based on McCreight's suffix tree construction algorithm. Recently, Shibuya [3] adapted Ukkonen's algorithm to [1] and [8], and proposed on-line construction algorithms that achieve the same time complexity bounds as their off-line counterparts (see Table 1). Still, to the best of our knowledge, an on-line construction algorithm which achieves linear time complexity has not been reported yet.

In this paper, we propose an on-line algorithm for constructing parameterized suffix trees in randomized linear time. Our algorithm can be regarded as the on-line counterpart to the Cole and Hariharan's off-line construction algorithm. Table 1 summarizes the time complexities of p-suffix tree construction algorithms in the literature.

Table 1. Time complexity of algorithms for constructing parameterized suffix trees. n is the length of an input parameterized string over $\Sigma \cup \Pi$ where Σ is a set of fixed symbols and Π is a set of parameters.

Off-line Algorithms		On-line Algorithms	
Algorithm	Time Complexity	Algorithm	Time Complexity
Baker [1]	$O(n(\|\Pi\| + \log \|\Sigma\|))$	Shibuya [3]	$O(n(\|\Pi\| + \log \|\Sigma\|))$
Kosaraju [8]	$O(n(\log \|\Pi\| + \log \|\Sigma\|))$	Shibuya [3]	$O(n(\log \|\Pi\| + \log \|\Sigma\|))$
Cole and Hariharan [9]	Randomized $O(n)$	This paper	Randomized $O(n)$

2 Preliminaries

Let $T = T[1..n]$ be a string of length n over a finite ordered alphabet. We denote the i-th symbol of T as $T[i]$. A substring starting at position i and ending at position j is denoted by $T[i..j] = T[i]T[i+1]\dots T[j]$, and if $i > j$, we regard $T[i..j]$ as an empty string. We denote the i-th *prefix* of T ending at position i as $T^i = T[1..i]$, and the j-th *suffix* of T starting at position j as $T_j = T[j..n]$.

2.1 Parameterized Matching

A *parameterized string* (*p-string*, in short) is a string over $\Sigma \cup \Pi$, where Σ is a set of fixed symbols and Π is a set of parameters. Two p-strings are said to be a *parameterized match* (*p-match*) if there exists a bijective mapping from Π to Π which maps a parameter in the first string into a parameter in the second string, while keeping the fixed symbols invariant. For example, suppose that we have a set of fixed symbols $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \cdots\}$ and a set of parameters $\Pi = \{\texttt{X}, \texttt{Y}, \texttt{Z}, \cdots\}$. Two p-strings `abXaXYb` and `abZaZXb` are a p-match since all fixed symbols, `a` and `b`, are identical and there exists the bijective mapping from `X` and `Y` of the first p-string into `Z` and `X`, respectively, of the second p-string.

In order to match two p-strings, we use an encoding *prev*, which chains together occurrences of the same parameter, to obtain a string in $(\Sigma \cup \mathbb{N})^*$ where $\mathbb{N}$ is the set of non-negative integers. For each parameter, the leftmost occurrence is represented by a 0, and each successive occurrence is represented by the difference in positions compared to the previous occurrence of the same parameter. A number representing such difference in positions is called a *parameter pointer*. For example, $prev(\texttt{abXaXYb}) = \texttt{ab0a20b} = prev(\texttt{abYaYZb})$.

Definition 1 (*prev* encoding). *We define* $prev : (\Sigma \cup \Pi)^* \to (\Sigma \cup \mathbb{N})^*$ *to be the function such that for any p-string T of length n, $prev(T) = S$ where, for $1 \le i \le n$,*

$$S[i] = \begin{cases} T[i] & \textit{if } T[i] \in \Sigma, \\ 0 & \textit{if } T[i] \in \Pi \textit{ and } T[i] \neq T[j] \textit{ for any } 1 \le j < i, \\ i - k & \textit{if } T[i] \in \Pi \textit{ and } k = \max\{j | T[j] = T[i] \textit{ and } 1 \le j < i\} \end{cases} .$$

From the above definitions, it is easily seen that matching two *prev* encoded strings is equivalent to p-matching two p-strings [1].

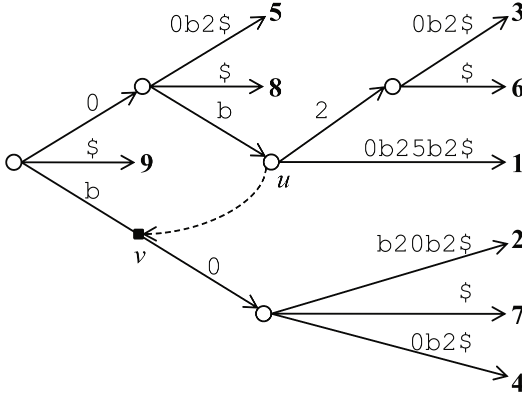


Fig. 1. A p-suffix tree of the p-string $T =$ `XbYbYXbX$` where $\Sigma = \{$`b`, `$`$\}$ and $\Pi = \{$`X`, `Y`$\}$

Theorem 1. *Two p-strings T and T' of the same length n are a p-match if and only if $prev(T) = prev(T')$.*

2.2 Parameterized Suffix Trees

Assume that a given p-string T ends with the sentinel symbol `$` $\in \Sigma$, which is lexicographically smaller than any other symbol in $\Sigma \cup \Pi$ and occurs nowhere else in T. For $1 \leq i \leq n$, we define the i-th *parameterized suffix* (p-suffix) of T, denoted by $\overline{T}_i$, as the *prev* encoded string of a suffix T_i, i.e., $prev(T_i)$.

A *parameterized suffix tree* (p-suffix tree) of T, $PST(T)$, is a compacted trie that represents all the p-suffixes of T. We refer to [1] for a formal description. An example of the p-suffix tree of $T =$ `XbYbYXbX$` is shown in Fig. 1. Each edge of the tree is labeled with a nonempty substring of suffix $\overline{T}_i$ for some i. Note that some edge labels are not exact substrings of $prev(T)$ as we will describe shortly. We frequently refer to any position, *locus*, in the tree as follows. The string for a locus u, denoted by $str(u)$, is the concatenation of the labels on the path from the root to that locus. If u is a node then we say that $str(u)$ occurs *explicitly*; otherwise, it occurs *implicitly*. For any locus u, we call u the locus of $str(u)$. If a locus u of a string ω lies in the middle of the edge from a node x to a node y, we call x and y the *contracted* and *extended* locus of u (or ω), respectively. An implicit locus can be specified by its contracted locus and the proper offset label. For any locus u, $pathlen(u)$ is defined to be the length of $str(u)$. We also denote the parent of a given node u as $parent(u)$. In Fig. 1, for example, we can see $str(u) =$ `0b` and $pathlen(u) = 2$ for explicit locus u. An implicit locus v can be specified by $(root, \text{b})$ in this example.

Due to the difference between normal and parameterized strings, we have to address the following issues to construct p-suffix trees.

Dynamic representation of edge labels. Note that, for some integers $1 \leq i < j \leq n$, a parameter symbol $T[j]$ can have different values in $\overline{T}_i$ and $prev(T)$, if the last occurrence of the same symbol lies before i. For example, observe that for $T = \texttt{aXaXb}$, the 4-th symbol X differs in $prev(T) = \texttt{a0a\underline{2}b}$ and $\overline{T}_3 = \texttt{a\underline{0}b}$. Since the algorithm frequently refers to the *prev* value of a symbol on any locus in the tree, it should be computed in constant time. By storing $prev(T)$ in an array of size n, we can compute the j-th symbol of $\overline{T}_i$ in constant time [1]. Let *eval* be the function of j and $b \in \Sigma \cup \mathbb{N}$ such that $eval(j, b) = 0$ if b is a non-negative integer larger than $j - 1$, and $eval(j, b) = b$, otherwise. Then, $\overline{T}_i[j] = eval(j, prev(T)[j + i - 1])$.

In order to store the p-suffix tree in space linear in the input size, each edge label is specified by a pair (k, p), where k and p are starting and ending positions of the corresponding substring of T. We store $pathlen(u)$ for each internal node u, so that we can dynamically determine the label on the edge. For example, if node u has an outgoing edge label (k, p), we evaluate the label as $\overline{T}_{k-pathlen(u)}[pathlen(u)+1..pathlen(u)+(p-k+1)]$. This can be done in time proportional to the length of the label.

Missing suffix links. We define *suffix links* in p-suffix trees as follows.

Definition 2 (Suffix link). *For a node u, we define a suffix link from a node u to a locus v, if $str(u)$ is a prefix of the $(i-1)$-th p-suffix $\overline{T}_{i-1}$ and $str(v)$ is a prefix of the i-th p-suffix $\overline{T}_i$ and $|str(u)| = |str(v)| + 1$. Let $link(u)$ denote this locus v.*

Unlike the suffix tree for strings, an internal node in the p-suffix tree does not necessarily have an outgoing suffix link defined as a node. This is due to the fact that the *distinct right context property* [10] does not hold for p-strings. Figure 1 shows that for node u, suffix link $link(u)$ points to the middle of an edge, which corresponds to implicit locus v. Observe that v is not defined as a *real* node, i.e., an internal node with at least two children. For this reason, dynamic data structure is used to maintain incoming suffix links for each edge and this incurs $O(\log n)$ factor in previous construction algorithms [1,8,3].

Node branching. Given a node u and a symbol α, *node branching* is to find the edge (u, v) whose label begins with α. Since an internal node in the tree has at most $|\Sigma| + |\Pi|$ children, branching may incur alphabet dependent $O(\log n)$ overhead.

3 Algorithm

We assume that p-string T is given *on-line* symbol by symbol and from left to right. At time r, the algorithm reads the r-th symbol of T and builds $PST(T^r)$ by updating $PST(T^{r-1})$. Note that the prefix of p-suffix $\overline{T}_i$ read so far at time r is $\overline{T}_i^r = \overline{T}_i[1..r - i + 1]$, where $1 \leq i \leq r$.

Algorithm 1. MAIN

1: Create $\perp$ and *root*, and set $link(root) \leftarrow \perp$;
2: Create an edge from $\perp$ to *root* and mark it as a don't care edge;
3: The result tree is denoted by $PST(T^0)$;
4: **for** $r = 1$ to n **do**
5: Compute $prev(T)[r]$;
6: Transform $PST(T^{r-1})$ into $PST(T^r)$;
7: **end for**

The outline of our algorithm for constructing the p-suffix tree of T is shown in Algorithm 1. Our algorithm is based on the Ukkonen's algorithm, but we need some tools to make our on-line algorithm run in linear time for p-strings.

- **Computing** *prev* **on-line.** As we have considered in the previous section, we need to determine the *prev* value of a newly introduced symbol in order to maintain an array of $prev(T)$. This can be done in $O(1)$ time by using a table of size $|\Pi|$ which holds the last occurrence of each parameter. In the case that $|\Pi|$ is a polynomial in n, we can use a dynamic perfect hashing scheme [9] instead. Thus we assume that at any time we can determine the symbol on the label in the p-suffix tree in constant time.
- **Implicit update.** For supporting implicit update of edges to leaf nodes u, we use an open transition (k, ∞) to represent the label on the edge from $parent(u)$ to u as described in [6].
- **Maintaining missing suffix links.** To maintain missing suffix links, three types of nodes, namely, real, imaginary and back-propagated nodes are used [9]. We call a node v an *imaginary* node, if v has only one child and there are some nodes u having suffix links to v, i.e., $v = link(u)$. We call a node u a *back-propagated* node, if u has one child and an outgoing suffix link to a node v, i.e., $v = link(u)$ exists. *Real* nodes are the other internal nodes, which are neither imaginary nor back-propagated.
- **Constant node branching.** Cole and Hariharan [9] showed that node branching can be done in randomized $O(1)$ time using dynamic perfect hashing with high success probability. By hashing a pair of the node number and the first symbol of the label, we can find the corresponding edge in constant time.

At time r, we need to update loci of the p-suffixes by extending the existing edges or creating new branches in $PST(T^{r-1})$. For $1 \le i \le r-1$, let ω denote $\overline{T}_i^{r-1}$. We can represent $\overline{T}_i^r$ as ωa, where $a \in \Sigma \cup \mathbb{N}$ is the *prev* value of the newly introduced r-th symbol on the corresponding locus. According to the occurrences of ω and ωa in $PST(T^{r-1})$, exactly one of the following cases holds:

- **Case 1.** ω **occurs only once in** $PST(T^{r-1})$**.**
  It implies that ωa does not occur in $PST(T^{r-1})$ and the locus of ω is a leaf in $PST(T^{r-1})$. In order to insert ωa to $PST(T^{r-1})$, we only update the locus of ω to point to ωa. However, this can be done implicitly thanks to the open transition [6].

- **Case 2.** ω **occurs more than once in** $PST(T^{r-1})$**, but** ωa **does not.**
  It implies that for some symbol $b \neq a$, ωb occurs in $PST(T^{r-1})$. Thus, we have to create a new leaf w and (possibly) an internal node v so that $str(w) = \omega a$ and $str(v) = \omega$.
- **Case 3.** ωa **occurs already in** $PST(T^{r-1})$**.**
  It implies that ωa already exists in $PST(T^{r-1})$. Thus, we do nothing but extend the current locus by one symbol to point to ωa.

We define an *active* point and an *end* point as follows. The end point at time r, denoted by End^r, is the locus of the longest p-suffix $\overline{T}_i^{r-1}$ for which Case 3 holds. The active point at time r, denoted by Act^r, is the locus of the longest p-suffix $\overline{T}_i^{r-1}$ for which Case 2 holds (if exists). If no such suffix for Case 2 exists, the active point is defined to be the same as the end point.

We now describe how the algorithm builds $PST(T^r)$ from $PST(T^{r-1})$. At time r, we denote $a \in \Sigma \cup \mathbb{N}$ as the *prev* value of the newly introduced r-th symbol. Note that we have to compute the proper value of a according to the corresponding locus. Let v be Act^r at the beginning of time r.

1. **(Case 3)** If we can follow an existing edge from v with a (i.e., $Act^r = End^r$),
   (a) follow the edge from v with a and find the locus w of ωa, where $\omega = str(v)$. By definition, w will be Act^{r+1}.
   (b) $r \leftarrow r + 1$.
2. Otherwise, we perform the following iterations until End^r is encountered.
   (a) **(Case 2)** While v does not have an a-transition
      i. Create an a-transition from v
      ii. Traverse upwards from v to find the nearest ancestor u which has an outgoing suffix link. Follow the suffix link from u to $u' = link(u)$.
      iii. Traverse downwards from u' to a locus v' such that $|str(v')| - |str(u')| = |str(v)| - |str(u)|$. Meanwhile, for every second node z' encountered on the path from u' to v' (excluding u' and v'), we create a new back-propagated node z (if not exists) on the path from u to v such that $|str(z')| - |str(u')| = |str(z)| - |str(u)|$. Set $link(z) \leftarrow z'$.
      iv. If v' is not a node, create a node at v' and $link(v) \leftarrow v'$.
      v. $v \leftarrow v'$.
   (b) **(Case 3)** Follow an existing edge from v with a and find the locus w of ωa, where $\omega = str(v)$. w will be Act^{r+1}.
   (c) $r \leftarrow r + 1$.

We remark that if a newly created node v' has only an a-transition, v' is an imaginary node by definition.

4 Analysis

We now discuss the time complexity of our algorithm.

First, we consider the total number of nodes created during the entire operations. Since we create at most one real node and one imaginary node per

suffix, the number of real and imaginary nodes is bounded by $O(n)$. By using a charging argument, the number of back-propagated nodes is also bounded by $O(n)$. We omit the details because it can be proven as in [9].

We now account for the time taken for traversing the p-suffix tree. At time r, the algorithm performs iterations until the end point is encountered. Note that the total number of iterations during the entire operations is bounded by $O(n)$ as in [6]. For each iteration, we process the following procedures:

1. Traverses up nodes from v to the nearest ancestor u with an outgoing suffix link.
2. Traverses down from $u' = link(u)$ to v' and create back-propagated nodes.
3. Performs the remaining operations, e.g., node branching, updating edge labels, computing $pathlen(v)$ for newly created nodes, etc.

The time to process Part 1 and 2 is proportional to the number of imaginary and back-propagated nodes, and it is bounded by $O(n)$. We refer to [9] for the details. Part 3 requires constant time to perform each operation and thus the total time is $O(n)$.

Theorem 2. *For a p-string T of length n, the parameterized suffix tree of T can be constructed on-line in randomized $O(n)$ time.*

References

1. Baker, B.S.: A theory of parameterized pattern matching: algorithms and applications. In: STOC 1993: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 71–80. ACM, New York (1993)
2. Baker, B.S.: Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM J. Comput. 26(5), 1343–1362 (1997)
3. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. Algorithmica 39(1), 1–19 (2004)
4. Weiner, P.: Linear pattern matching algorithms. In: SWAT 1973: Proceedings of the 14th Annual Symposium on Switching and Automata Theory, Washington, DC, USA, pp. 1–11. IEEE Computer Society, Los Alamitos (1973)
5. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
6. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
7. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM 47(6), 987–1011 (2000)
8. Rao Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In: FOCS 1995: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 631–637. IEEE Computer Society, Los Alamitos (1995)
9. Cole, R., Hariharan, R.: Faster suffix tree construction with missing suffix links. SIAM J. Comput. 33(1), 26–42 (2004)
10. Giancarlo, R.: A generalization of the suffix tree to square matrices, with applications. SIAM J. Comput. 24(3), 520–562 (1995)

Succinct Text Indexing with Wildcards

Alan Tam, Edward Wu, Tak-Wah Lam⋆, and Siu-Ming Yiu

Department of Computer Science, University of Hong Kong, Hong Kong
{sltam,mkewu,twlam,smyiu}@cs.hku.hk

Abstract. A succinct text index uses space proportional to the text itself, say, two times $n \log \sigma$ for a text of n characters over an alphabet of size σ. In the past few years, there were several exciting results leading to succinct indexes that support efficient pattern matching. In this paper we present the first succinct index for a text that contains wildcards. The space complexity of our index is $(3 + o(1))n \log \sigma + O(\ell \log n)$ bits, where ℓ is the number of wildcard groups in the text. Such an index finds applications in indexing genomic sequences that contain single-nucleotide polymorphisms (SNP), which could be modeled as wildcards.

In the course of deriving the above result, we also obtain an alternate succinct index of a set of d patterns for the purpose of dictionary matching. When compared with the succinct index in the literature, the new index doubles the size (precisely, from $n \log \sigma$ to $2n \log \sigma$, where n is the total length of all patterns), yet it reduces the matching time to $O(m \log \sigma + m \log d + occ)$, where m is the length of the query text. It is worth-mentioning that the time complexity no longer depends on the total dictionary size.

1 Introduction

Pattern matching is a fundamental problem. Consider a text T and a pattern P, the earliest work can solve the problem in $O(|T| + |P|)$ time. When the text remains relatively static (say, the text is the human genome), one would like to build an index of T so as to speed up pattern matching. Let n be the number of characters of T. The classical index suffix trees requires $O(n)$ words, or equivalently, $O(n \log n)$ bits, and can support pattern matching in $O(|P| + occ)$ time, where occ is the number of occurrences of P in T. Note that the space complexity has a natural lower bound of $n \log \sigma$ bits (i.e., worst-case text size), where σ is the alphabet size. Starting with the work of Ferragina and Manzini [6] and Grossi and Vitter [9], the past decade has witnessed a chain of works that make it feasible to build a succinct text index with size proportional to $n \log \sigma$ bits or even a compressed index (with size proportional to nH_k bits), while supporting efficient pattern matching, using $O(|P| + occ \log^{1+\epsilon} n)$ time for any $\epsilon > 0$ (see the survey by Navarro and Mäkinen [12] for a complete list of references).

This paper is concerned with pattern matching on text containing wildcards (or don't care characters). Specifically, a wildcard, denoted by ϕ, is a special character that matches any single character. Fischer and Paterson [8] were among

⋆ Part of the work is supported by RGC Grant HKU 714006E.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 39–50, 2009.
© Springer-Verlag Berlin Heidelberg 2009

the first to study wildcard matching. There are several results on text indexing for wildcard matching. In the simple setting where the text contains no wildcards, Rahman and Iliopoulos [15] and later Lam et al. [11] have each given an $O(n)$-word index for matching patterns with wildcards. Indexing a text containing wildcards is technically more challenging. It naturally arises in indexing genomic sequences, in which some base pairs are known to be single-nucleotide polymorphisms (SNP), that could be modeled as wildcards. The wildcard index by Cole et al. [5] uses $O(n \log^k n)$ words, where k is the number of wildcards. It takes $O(|P| + \log^k n \log\log n + occ)$ time to find the occurrences of a given pattern P without wildcards. Obviously, the size of the index implies a prohibitive amount of memory for applications involving more than a few wildcards. Lam et al. [11] have given another index, which requires only $O(n)$ words and also avoids a time complexity exponential in the number of wildcards. Precisely, the time required is $O(|P| \log n + \gamma + occ)$ time, where γ is defined as follows. Assume that the text T contains $\ell \geq 1$ groups of consecutive wildcards. I.e., $T = T_1\phi^{k_1}T_2\phi^{k_2} \ldots \phi^{k_\ell}T_{\ell+1}$, where $k_1, k_2, \ldots k_\ell \geq 1$, and each T_i contains no wildcards. Define γ to be the sum, over all T_i's, of the number of occurrences of T_i in P. Note that γ is upper bounded by $|P|(\ell + 1)$.[1] Both indexes can be extended to handle patterns with wildcards.

When we index long genomic sequences (e.g., the human genome which has about three billion characters), even an $O(n)$-word or $O(n \log n)$-bit data structure is still too large. In this paper, we give a succinct index for a text containing wildcard characters. Precisely, assume that T has $\ell \geq 1$ wildcard groups, the space complexity is $(3 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits. For practical applications, the last two terms can often be absorbed into $o(n \log \sigma)$, and the pattern matching time of the new index compares favorably with the previous indexes. It is useful to define $\hat{\ell}$ to be the number of distinct wildcard-group lengths (i.e., the number distinct elements in the set $\{k_1, k_2, \ldots, k_\ell\}$; e.g., if $k_i = 1$ for all i, then $\hat{\ell} = 1$). Given a pattern P, our new index can find all occurrences of P in $O(|P|(\log \sigma + \min(|P|, \hat{\ell}) \log \ell) + \gamma \log_\sigma \ell + occ \log^{1+\epsilon} n)$ time for any $\epsilon > 0$.

In the course of deriving the above solution for indexing wildcards, we have also obtained a succinct index for the dictionary matching problem, which is another classical matching problem not involving wildcards. In this problem, we are required to index a set of patterns $P_1, P_2, \ldots, P_d$ with total length n. Given a query text T, the index is required to locate the occurrences of all P_i in T. Aho and Corasick [1] were the first to give an $O(n)$-word index for the dictionary matching problem. Chan et al. [3] have improved the space complexity to $O(n\sigma)$ bits, and recently Hon et al. [10] gave a succinct index using $(1 + o(1))n \log \sigma + O(d \log n)$ bits. The matching time for any text T is $O(|T|(\log^\epsilon n + \log d) + occ)$. In this paper we present a different way to derive a succinct index for the dictionary

[1] [11] has given a more practical upper bound of γ. Define the prefix complexity of the T_i's to be the maximum number of T_j's that are prefixes of the same T_i. Then γ is at most $|P|$ times the prefix complexity. In practice, wildcards are sparse and the prefix complexity is often a small constant.

matching problem. The new index increases the space to $(2 + o(1))n \log \sigma + O(d \log n)$ bits, but reducing the matching time to $O(|T|(\log \sigma + \log d) + occ)$.

Organization of the paper. In Section 2, we will review several data structures in the literature for indexing text (without wildcards), as well as for indexing geometric data on a two-dimensional plane. In Section 3, we describe the core elements of our succinct index, which include BWT and a new solution to the dictionary matching problem. In Section 4, we present the details of matching with wildcards in the text.

2 Preliminaries

Throughout this paper, we consider texts and patterns with characters chosen from an alphabet Σ of size σ. The text can contain one or more wildcard character ϕ, which is a special character not in Σ, and which can match any character in Σ. Our data structures would make use of two additional symbols \$ and # not in Σ. We assume that \$ is lexicographically smaller than all characters in Σ, and # greater than all characters in Σ. Below we review several data structures for text indexing (without wildcards), as well as points and rectangles in a two dimensional plane.

2.1 Suffix Array

Let $T[1..n]$ be a text that does not contain wildcard character and ends with a special character \$. A suffix of T is a substring $T[j..n]$ where $1 \leq j \leq n$. We sort all suffixes of T in lexicographical order and store their starting positions in an integer array $SA[1, n]$. Intuitively, $SA[i]$ gives the starting position of the i-th smallest suffix of T, or equivalently, the suffix with rank i.

Consider a pattern X. Inside SA, all the suffixes of T that contain X as a prefix appear in consecutive entries. We define the SA range X to be $[s, r]$ if there are $s' = s - 1$ suffixes lexicographically smaller than X, and r suffixes smaller than or equal to X. If X does not appear in T, then $s - 1 = r$ and the SA range has a right boundary (r) smaller than the left boundary (s). In this case, we say that the SA range of X is *empty*.

2.2 Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform (BWT) was first proposed as a compression technique [2]. Later it was found that BWT can support pattern matching efficiently when equipped with auxiliary data structures. Let $T[1..n]$ be a text (containing no wildcard). Assume $T[n] = \$$. The BWT of T is a sequence of n characters such that the i-th character is the character in T just preceding the rank-i suffix of T. Precisely, $BWT[i] = T[j - 1]$ where $j = SA[i]$ and $SA[i] \neq 1$. If $SA[i] = 1$, $BWT[i] = \$$.

BWT can be used to compute the SA range of any pattern if it is equipped with auxiliary data structures to compute the functions $Count(c)$ and $Appear(i, c)$. For

any character c, $Count(c)$ gives the number of characters in T that are lexicographically smaller than c, and $Appear(i, c)$ returns the number of times c appears in the prefix $BWT[1..i]$. Suppose that the SA range $[s, r]$ of a string X is given. Then, for any character c, we can find the SA range of cX as $[Count(c) + Appear(s-1, c) + 1, Count(c) + Appear(r, c)]$ [6].

A straightforward implementation of the *Appear* function requires $O(n\sigma \log n)$ bits. To reduce the space requirement, we use the *wavelet tree* implementation proposed by Ferragina et al. [7]. It only uses $n \log \sigma + o(n \log \sigma)$ bits, but it is slower, taking $O(\log \sigma)$ time to serve each function call. On the other hand, with the wavelet tree implementation, we no longer need to store T or BWT explicitly, since it supports retrieving any single character of BWT in $O(\log \sigma)$ time. In summary, BWT together the auxiliary data structures occupy $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits and can support pattern matching efficiently, as stated in the following lemma.

Lemma 1. *Let P be a pattern of m characters. The SA ranges of all suffixes of P can be computed in $O(m \log \sigma)$ time.*

2.3 Orthogonal Range Search

Consider a set G of ℓ points on a two-dimensional plane. Given a rectangle $R = (x_1, y_1) \times (x_2, y_2)$, we want to find all the points in G that are enclosed by R.

Lemma 2. *[13] Given ℓ points with coordinates in $[1..n]$, we can build an $O(\ell \log n)$-bit data structure such that given a query rectangle R, all the points enclosed by R can be reported in $O(\log \ell + t \log^{\epsilon} \ell)$ time, where t is the number of answers and $\epsilon > 0$.*

2.4 Point Enclosure Problem

Consider a set H of ℓ rectangles on a two-dimensional plane. Given a query point $q = (x, y)$, we want to find efficiently all the rectangles in H that enclose q.

Lemma 3. *[4] Given ℓ rectangles on a 2-D plane, we can build an $O(\ell)$-word data structure such that given a query point q, all the rectangles enclosing q can be reported in $O(log\ell + t)$ time, where t is the number of answers.*

3 Succinct Representation of Non-wildcard Characters

Consider a text T of n characters. Suppose $T = T_1\phi^{k_1}T_2\phi^{k_2} \ldots T_\ell\phi^{k_\ell}T_{\ell+1}$, where ϕ^{k_i} denotes a group of k_i consecutive wildcards, and each T_i does not contain any wildcard. Below, each T_i is called a text segment. In this section we show how to index the T_i's. We make use of BWT and the point enclosure data structure. The former allows us to determine whether each T_i is a prefix of a given pattern X in constant time. This prefix matching capability, together with the point enclosure data structure, allow us to have a faster index for dictionary matching, i.e., to find out the occurrences of every T_i in a given pattern X. In Section 4, we will show how to make use of these indexes to perform wildcard matching.

3.1 BWT and Prefix Matching

Define $TS = T_1\$T_1\#T_2\$T_2\# \ldots T_{\ell+1}\$T_{\ell+1}\#$, where \$ and # are new symbols assumed to be lexicographically smaller than and greater than all symbols in T, respectively. We construct the BWT index (including the necessary auxiliary data structures) to support pattern matching for TS. We denote this index as $BWT\text{-}TS$. Note that with $BWT\text{-}TS$, we no longer need to store the text TS explicitly as the index can support pattern matching TS. $BWT\text{-}TS$ uses $(2+o(1))n \log \sigma + O(\sigma \log n)$ bits. Furthermore, we explicitly store the SA range of each T_i (with respect to the suffixes of TS), using $(\ell+1)\log n$ bits.

Below, an SA range always makes reference to the suffixes of TS. By Lemma 1, for any pattern $P[1..m]$, we can use $BWT\text{-}TS$ to find the SA ranges of the suffixes $P[m..m], P[m-1..m], \ldots, P[1..m]$ in $O(m \log \sigma)$ time. In the rest of this section, we show how to exploit the SA ranges of a suffix $X = P[j..m]$ and a text segment T_i to determine whether X is a prefix of T_i, and more importantly, whether T_i is a prefix of X.

Note that X may or may not appear in any T_i, and the SA range $[s, r]$ of X may be empty ($s - 1 = r$) or non-empty ($s \le r$). When X has a non-empty SA range, it is straightforward to determine whether X is a prefix of a text segment T_i, or vice versa. See the following lemma. The duplicate structure of TS is needed to handle the case when X has an empty SA range.

Lemma 4. *Suppose that the text segment T_i has SA range $[p, q]$. For any string X, if the SA range $[s, r]$ of X is non-empty, then* **(i)** *X is a prefix of T_i if and only if $s \le p \le q \le r$; and* **(ii)** *T_i is a prefix of X if and only if $p \le s \le r \le q$. Both conditions can be determined in constant time.*

Proof. We only prove (i), as (ii) is symmetric. Suppose X is a prefix of T_i. The SA range of X encloses all suffixes with prefix X, so the SA range of T_i must be enclosed by the SA range of X. Hence, $s \le p \le q \le r$. Conversely, suppose $s \le p \le q \le r$. The SA range of X encloses all suffixes with the prefix X. Since the SA range of T_i is a subrange of $[s, r]$, all suffixes with the prefix T_i must also have X as the prefix. Thus, X is a prefix of T_i.

It remains to consider the case when X has an empty SA range. In this case, X does not occur anywhere in TS, and X is not a prefix of any text segment T_i. However, T_i can still be a prefix of X. To determine this case is no longer straightforward. The following lemma exploits the duplicate structure of each T_i in TS to derive a simple condition.

Lemma 5. *Suppose that the text segment T_i has SA range $[p, q]$. For any string X, if the SA range $[s, r]$ X is empty (i.e., $s - 1 = r$), then T_i is a prefix of X if and only if $p \le r < s \le q$. This can be determined in constant time.*

Proof. Suppose that T_i is a prefix of X. Since X has an empty SA range and T_i has a non-empty one, T_i is a proper prefix of X. Recall that \$ is smaller than any character in Σ, and hence $T_i\$$ is lexicograpically smaller than X. Similarly, $T_i\#$ is lexicograpically greater than X. If $[s, r]$ is an empty range, $s - 1 = r$ and $r < s$. It remains to prove the other two inequalities: (1) $p \le r$; (2) $s \le q$.

(1) By definition of $[p, q]$, the p-th smallest suffix of TS contains $T_i\$$ as a prefix. This prefix is smaller than X, and hence there are at least p suffixes of TS smaller than X. Therefore, $s - 1 \geq p$ and $r = s - 1 \geq p$.
(2) By definition of $[p, q]$ and $\#$, the q-th smallest suffix of TS contains $T_i\#$ as a prefix, and this prefix is greater than X. There are at most $q - 1$ suffixes of TS smaller than or equal to X. Therefore, $r \leq q - 1$ and $s = r + 1 \leq q$.

Conversely, if $p \leq r < s \leq q$, we can prove that T_i is a prefix of X. Let X' be the prefix comprising the first $|T_i|$ characters of X (or equal to X if X is shorter than T_i). For the sake of contradiction, we assume that T_i is not a prefix of X and consider the scenarios when X' is larger than T_i or smaller than T_i. If $X' > T_i$, TS contains at least q suffixes smaller than X, and $s - 1 \geq q$. It contradicts that $s \leq q$. If $T_i > X'$, then there are at most $p - 1$ suffices that are smaller than X, and $s - 1 \leq p - 1$. It contradicts that $r = s - 1 \geq p$.

3.2 Dictionary Matching

Given the text segments $T_1, T_2, \ldots, T_{\ell+1}$ and a pattern $P[1..m]$, the dictionary matching problem is to report the occurrences of all T_i that appear in P. In this section, we show how to make use of *BWT-TS* (defined in the previous section) and a point enclosure index to perform dictionary matching in a more efficient way than the existing indexes in the literature. The overall space requirement is $(2 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits, and the dictionary query can be answered in $O(m \log \sigma + m \log \ell + \gamma)$ time, where γ denotes the total number of occurrences. This result, when compared with the work of Hon et al. [10], doubles the space requirement, but improves the dominating term of the time complexity from $m \log^{\epsilon} n$ to $m \log \sigma$.

Suppose that a text segment T_i appears in P. Then T_i must be a prefix of some suffix of P. To find out such occurrences, we consider each suffix $P[j..m]$ of P separately and find all T_i's that are a prefix of $P[j..m]$. First of all, we use Lemma 1 to compute the SA ranges (with respect to TS) of every suffix $P[j..m]$. Using Lemmas 4(ii) and 5, we can check whether T_i, for all i in $[1, \ell + 1]$, is a prefix of $P[j..m]$ in $O(\ell)$ time. We can speed up this checking process for each $P[j..m]$ to $O(\log \ell)$ time by a reduction to a point enclosure problem defined as follows.

> For each T_i with SA range $[p, q]$, we consider the rectangle $(p, p) \times (q, q)$ in the two-dimensional plane. Let H be the set of all the $\ell + 1$ rectangles associated with the T_i's. We build an $O(\ell \log n)$-bit index for point enclosure query. For each $P[j..m]$, we transform its SA range $[s, r]$ to a query point $x_j = (s, r)$. By Lemmas 4(ii) and 5, T_i is a prefix of $P[j..m]$ if and only if the rectangle of T_i encloses x_j.

Lemma 6. *We can build an index for $T_1, T_2, \ldots, T_{\ell+1}$ using $(2 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits. Then, given a pattern P, the occurrences of all T_i in P can be computed in $O(m \log \sigma + m \log \ell + \gamma)$ time, where γ is the total number of occurrences.*

Proof. First, we can find the SA ranges of all suffixes of P. By Lemma 1, it takes $O(m \log \sigma)$ time. H contains $\ell + 1$ rectangles. By Lemma 3, we can build an $O(\ell \log n)$-bit data structure to answer the point enclosure query of each suffix of P in $O(\log \ell + t)$ time, where t is the number of answers. In summary, the total time required to find the occurrences of all T_i in P is $O(m \log \sigma + m \log \ell + \gamma)$.

Repeated Dictionary Matching. Given a pattern P, after we have computed the γ occurrences of the text segments in P, we want to store these results in a compact way so that they can be retrieved altogether in $O(\gamma)$ time. It is indeed relatively simple to derive a scheme using only $O(m \log \ell)$ bits, i.e., independent of the size of γ. Details are as follows.

First, we observe a relationship between all text segments T_i that are a prefix of a particular suffix $P[j..m]$ of P. For any $1 \leq j \leq m$, let D_j be the set containing all such T_i's. Let $Longest(D_j)$ denote the longest T_i in the set D_j. Note that a text segment T_i is in D_j if and only if T_i is a prefix of $Longest(D_j)$. Therefore, for each T_i, we maintain a set of text segments that are each a prefix of T_i. Then, for each $P[j..m]$, we only need to store $Longest(D_j)$. The space required to store all $Longest(D_j)$ for all j is $O(m \log \ell)$ bits. To re-generate the γ answers of the dictionary matching for P, we report all T_i's that are each a prefix of $Longest(D_j)$ for all j.

It remains to show how to maintain the list of prefix text segments for each T_i. There are several possible ways. Below we make use of a compact trie, which requires $O(\ell \log \ell)$ bits. First, we build a compact trie CT for all text segments $\{T_1\$, T_2\$, .., T_\ell\$\}$. Each T_i is associated with a leaf in CT. If text segments are identical, they are associated with the same leaf. Consider any node u in CT, we denote $path(u)$ as the concatenation of all edge labels from the root to u. For each T_i, we mark the node v of CT such that $path(v) = T_i$. Then, for all nodes, we store a link to its closest marked ancestor. The space required by CT is $O(\ell \log \ell)$ bits. Given any T_i, we can recover the text segments that are a prefix of T_i by traversing the marked nodes from the leaf associated with T_i towards the root. To conclude, the space requirement is dominated by BWT-TS and the SA ranges of all T_i's, which is $(2 + o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits.

Lemma 7. *Using CT, we can retrieve, for any T_i, all the text segments that are each a prefix of T_i in $O(t)$ time, where t is the number of results.*

4 Matching with Wildcards

Finally we come to the discussion of matching a text T containing wildcards. Assume $T = T_1\phi^{k_1}T_2\phi^{k_2}..T_\ell\phi^{k_\ell}T_{\ell+1}$, where $k_1, k_2, \ldots, k_\ell \geq 1$, and each T_i contains no wildcards. The basic data structure is BWT-TS (as defined in Section 3.1), which indexes all the text segments T_i of T. Furthermore, we index the reverse of each T_i, which is denoted $\overleftarrow{T_i}$ below. Let $TP = \overleftarrow{T_1}\$\overleftarrow{T_2}\$\ldots\overleftarrow{T_\ell}\$$, and denote BWT-TP as the index comprising the BWT of TP and the required auxiliary data structures (as stated in Section 2.2). BWT-TS and BWT-TP together occupy $(3 + o(1))n \log \sigma + O(\sigma \log n)$ bits. Note that TP doesn't have the

duplicate structure of TS. We only need $BWT\text{-}TP$ to support constant-time checking whether a string X is a prefix of some $\overleftarrow{T_i}$, but not vice versa. We also store the SA ranges of all T_i's with respect to TS, as well as the SA ranges of all $\overleftarrow{T_i}$'s with respect to TP. They require $O(\ell \log n)$ bits.

Additional auxiliary data structures (such as for indexing the SA ranges of the T_i's) will be given in the discussion below; they only use $o(n \log \sigma) + O(\ell \log n)$ bits.

Let P be a given pattern of m characters. Following Lam et al. [11], we divide the problem of matching P with T into three cases.

Type 1: P is a substring of some T_i, where $1 \leq i \leq \ell + 1$.
Type 2: P occurs in $T[u..v]$ which contains exactly one wildcard group ϕ^{k_j}.
Type 3: P occurs in $T[u..v]$ which contains two or more wildcard groups.

Below we show how to make use of $BWT\text{-}TS$, $BWT\text{-}TP$ and some auxiliary data structures to match the pattern efficiently in each case.

4.1 Type 1 Matching

This is the simplest case and it does not involve any wildcards. We simply search for P in $BWT\text{-}TS$. The required SA range can be computed in $O(n \log \sigma)$ time. The only technical difficulty is how to retrieve the occurrences of P given the SA range of P with respect to TS. The problem becomes trivial if we can keep a suffix array of TS, which requires $O(n \log n)$ bits. Below we show that with a suitable sampling of the suffix array, we can reduce the space to $o(n \log \sigma)$, while allowing each occurrence to be retrieved in $O(\log^{1+\epsilon} n)$ time for any $\epsilon > 0$.

Lemma 8. *We can build an $o(n \log \sigma)$-bit auxiliary data structure such that, given the SA range of a pattern P, the occurrences of P in TS can be reported in $O(occ_1 \log^{\epsilon+1} n)$ time, where occ_1 is the number of type-1 occurrences.*

Proof. Let β be the sampling factor. We show that an index of $O(\frac{n}{\beta} \log n)$ bits would allow us to access an value in the suffix array of TS in $O(\beta \log \sigma)$ time.

Let M be a bit vector of length $|TS|$. Initially, $M[i] = 0$ for all i. Then we mark every $M[i] = 1$ where $SA[i] = k\beta$ and $0 \leq k \leq \left\lceil \frac{n}{\beta} \right\rceil$. We store the tuple $(i, SA[i])$ where $M[i]$ is marked with 1 in ascending order of i. Suppose we want to retrieve $SA[j]$ which has not been stored up. Let $j_0 = j$. We will have to find an index j_y such that the tuple $(j_y, SA[j_y])$ is stored and $SA[j_0] - SA[j_y] < \beta$. In general, we can find the index j_x by backward searching $BWT\text{-}TS$ with character $BWT\text{-}TS[j_{x-1}]$. We recurively obtain $j_1, j_2, j_3..$ until we find j_y such that the tuple $(j_y, SA[j_y])$ is stored. Tuple can be retrieved in constant time if a rank and select data structure has been built on M. Then, we report $SA[j] = SA[j_y] + y$. The searching time for a character in $BWT\text{-}TS$ is $O(\log \sigma)$. Since $y < \beta$, we can compute $SA[j]$ in $O(\beta \log \sigma)$ time.

Let $\beta = \lceil \log^{\epsilon} n \log_{\sigma} n \rceil$ for some $\epsilon > 0$. The space requirement of the sampled SA plus the rank and select index is $o(n \log \sigma)$. The access time of an entry in the suffix array becomes $O(\log^{1+\epsilon} n)$.

4.2 Type 2 Matching

For type 2 matching, we are interested in matching a given pattern $P[1..m]$ with $T_i\phi^{k_i}T_{i+1}$ for all $1 \leq i \leq \ell$. More specifically, we want to find out whether, for some $1 \leq a \leq m$, $P[1..a]$ is a suffix of T_i, and $P[a+k_i+1..m]$ is a prefix of T_{i+1}. The first condition can be rewritten as $\overleftarrow{P[1..a]}$ is a prefix of $\overleftarrow{T_i}$. In other words, both conditions involve prefix matching, so we can exploit BWT-TP and the SA ranges of $\overleftarrow{T_i}$'s, as well as and BWT-TS and the SA ranges of T_i's. By Lemma 4(i), we would first compute the SA ranges of the suffixes of P and $\overleftarrow{P}$; then for any fixed i and a, it takes constant time to check whether $\overleftarrow{P[1..a]}$ is a prefix of $\overleftarrow{T_i}$, and $P[a+k_i+1..m]$ is a prefix of T_{i+1}. Finding all the SA ranges requires $O(m\log\sigma)$ time, and then the naive implementation of type-2 matching requires $O(m\ell)$ time.

For genomic sequences, we observe that the number of wildcard groups (i.e., ℓ) is usually not a small constant, but the number of distinct wildcard group sizes k_i's is a small constant. Recall that the latter is denoted by $\hat{\ell}$. In fact, it is often the case that most groups contain only one wildcard. This motivates us to further improve the time complexity to something depending on $\hat{\ell}$ instead of ℓ. Below we show how to index the SA ranges of the T_i's using an orthogonal range search index. Then the time complexity can be reduced to to $O(m(\hat{\ell})\log\ell + occ_2\log^{\epsilon}\ell)$ time, where occ_2 is the number of type-2 occurrences of P.

Consider any integer b which is equal to some wildcard group size k_i. Let $W(b)$ denote all the wildcard groups that have size b, i.e., $W(b) = \{i \mid k_i = b \text{ and } 1 \leq e \leq \ell\}$. We want to conduct type-2 matching for all the wildcard groups in $W(b)$ together. Given a position a of P, we want to find, for all i in $W(b)$, whether $P[1..a]$ is a suffix of T_i and $P[a+b+1..m]$ is a prefix of T_{i+1}.

Lemma 9. *We can build an $O(\ell\log n)$-bit data structure to store the SA ranges of $\overleftarrow{T_i}$'s and the SA ranges of T_i's. Then, for any wildcard group size b, given a pattern $P[1..m]$ and a position $1 \leq a \leq m$, we can find in $O(\log\ell)$ time the number of $i \in W(b)$ such that $\overleftarrow{P[1..a]}$ is a prefix of $\overleftarrow{T_i}$ and $P[a+b+1..m]$ is a prefix of T_{i+1}. Furthermore, if there are t such i's, we can report them in $O(t\log^{\epsilon}\ell)$ time for some $\epsilon > 0$.*

Proof. We make use of orthogonal range search on a two-dimensional plane. Consider any wildcard group size b. We define a set G_b of points as follows. For each wildcard group $i \in W(b)$, let the SA range of $\overleftarrow{T_i}$ on TP be (s', r') and the SA range of T_{i+1} on TS be (s, r). We add the point (s', s) into G_b. Given any position a on P, let R_a be the rectangle $(x_1, y_1) \times (x_2, y_2)$ where (x_1, y_1) and (x_2, y_2) are the SA range of $\overleftarrow{P[1..a]}$ on TP and the SA range of $P[a+b+1..m]$ on TS, respectively. We find all the points on G_b that is enclosed by the query range R_a. A point in G_b represents a wildcard group k_i, it is enclosed by R_a if and only if the SA range of $\overleftarrow{P[1..a]}$ encloses the SA range of $\overleftarrow{T_i}$ and the SA range of $P[a+b+1..m]$ encloses the SA range of T_{i+1}. By Lemma 2, an $O(\ell\log n)$-bit data structure can be built for all $\hat{\ell}$ distinct wildcard group sizes, then we can

determine the number of points enclosed by R_a in $O(\hat{\ell}+\log \ell)$ time, and retrieve each point in $O(\log^{\epsilon} \ell)$ time.

There are ℓ wildcard groups. The total number of points in all $\hat{\ell}$ orthogonal range search indexes is ℓ. Therefore, the total space required by the orthogonal range search indexes is $O(\ell \log n)$ bits.

Let us summarize the computation for type-2 matching for any given pattern P of m characters. We compute the SA ranges of all $\overleftarrow{P[1..a]}$ with respect to TP and the SA ranges of all $P[a..m]$ with respect to TS in $O(m \log \sigma)$ time. There are $\min(m, \hat{\ell})$ different b's for P. For each b, we consider every position a on P. All type-2 matches can be found in $O(m \log \ell + t \log^{\epsilon} \ell)$ time, where t is the number of occurrences. Summing over possible wildcard group size, we obtain the following lemma.

Lemma 10. *Given a pattern of m characters, all type-2 matches can be located in $O(m(\log \sigma + \min(m, \hat{\ell}) \log \ell) + occ_2 \log^{\epsilon} \ell)$ time, where occ_2 is the number of type-2 occurrences and ϵ is an arbitrary positive constant.*

4.3 Type 3 Matching

Type-3 matching occurs when a pattern P matches with a substring $T[j..j+m]$ which contains at least two groups of wildcards. In this case, P contains at least a whole text segment T_i. Therefore, we will first find out all T_i completely included in P, and then verify whether each such T_i can be extended to form a type-3 matching.

The first step is equivalent to performing a dictionary matching to report all T_i that occurs in P. By Lemma 6, we could find all T_i that occurs on P in $O(m \log \sigma + m \log \ell + \gamma)$ time, where γ is the total number of occurrences of the T_i's in P. If T_i occurs in P with starting position x, then it is possible that P occurs in T with starting position $y = t_i - x + 1$, where t_i is the starting position of T_i in T. Using *BWT-TS* and *BWT-TP*, we can apply Lemma 4(i) to verify each candidate position y in constant time. Details are as follows.

First of all, we collect all the γ candidate positions y in an array $A[1..n]$ as follows. Initially, all entries of A are set to zero. We employ the constant time initialization technique[14] on A. The access time to any cell in A remains constant. Each time we find a candidate position y of P, we increment $A[y]$ by 1. The working space required by A is $O(n \log \ell)$ bits.

Consider each y with $A[y] > 0$. We want to verify whether P matches $T[y..y+m-1]$. Let T_f be the first text segment whose starting position $t_f \geq y$. Let T_g be the last text segment that ends at or before $y+m-1$ (i.e., $t_g \leq y+m-|T_i|$). Note that $g \geq f$. A position y defines a type-3 matching of P if and only if the following three conditions hold.

(1) $A[j] = g - f + 1$.
(2) If $y < t_f - k_{f-1}$ (i.e., the wildcard group $\phi^{k_{f-1}}$ starts after $T[y]$), then $P[1..t_f - k_{f-1} - y]$ is a suffix of T_{f-1}, or equivalently, $\overleftarrow{P[1..t_f - k_{f-1} - y]}$ is a prefix of $\overleftarrow{T_{f-1}}$.
(3) If $t_{g+1} \leq y + m - 1$ (i.e., the wildcard group ϕ^{k_g} ends before $T[y+m-1]$), then $P[t_{g+1} - y + 1..m]$ is a prefix of T_{g+1}.

Suppose that we have computed the SA ranges of all suffixes of P with respect to TS, as well as the SA ranges of all the suffixes of $\overleftarrow{P}$ with respect to TP. Then, by Lemma 4(i), we can make use of BWT-TP and BWT-TS to verify condition (2) and condition (3) in constant time. We conclude with the following lemma.

Lemma 11. *All type-3 matches can be located in $O(m \log \sigma + m \log \ell + \gamma)$ time. The working space required is $O(n \log \ell + m \log n)$ bits.*

Theorem 1. *Combining the results on type-1, type-2 and type-3 matching, we can find all occurrences of a given pattern P of m characters in $O(m(\log \sigma + \min(m, \hat{\ell}) \log \ell) + occ_1 \log^{\epsilon+1} n + occ_2 \log^{\epsilon} \ell + \gamma)$ time, where occ_1 and occ_2 denote the number of type-1 and 2 occurrences respectively, and γ is the occurrences of all text segments in P. The index space required is $(3+o(1))n \log \sigma + O(\sigma \log n) + O(\ell \log n)$ bits. The working space required is $O(n \log \ell + m \log n)$ bits.*

Reducing the Working Space. The solution to the type-3 matching demands a working space $O(n \log \ell + m \log n)$ bits. The first term is way too much. Below we show how to trade the running time for a solution that requires less working space. At the end, we obtain a solution that requires only $O(n \log \sigma + m \log n)$-bit working space, but the verification time for the γ candidates would increase to $O(\gamma \log_\sigma \ell)$. Intuitively, the idea is to split the array A into a number of subarrays. Then, we parse the γ dictionary matching results several times to cover all candidate positions.

By Lemma 7, we could retrieve the γ matching results for multiple times. Precisely, we could retrieve the γ matching results for d times in $O(d\gamma)$ time. Now, we split the entries in the array A into a number of groups. In each group, there are $\rho = \lfloor \frac{n \log \sigma}{\lceil \log \ell \rceil} \rfloor$ consecutive entries of array A. No entry in A is contained in more than one group. Therefore, there are $O(\log_\sigma \ell)$ groups of entries in total. Each group corresponds to a range of entries in A.

Let $B[1..\rho]$ be an array of integers. The space required by B is $O(\rho \log \ell) = O(n \log \sigma)$ bits. We repeat the process to mark the candidate positions, however, we mark the candidate positions on array B instead. We set $b = 1, \rho + 1, 2\rho + 1, \ldots, \rho \log_\sigma \ell + 1$. For each b, we mark on the array B by increasing the entry $B[j']$ by one if the candidate position $j = t_i - k + 1$ falls between b and $b + \rho - 1$, where $j' = j - b + 1$. We ignore all candidate positions that do not fall between b and $b + \rho - 1$. After we have marked array B for all γ dictionary matching occurrences, for each $B[j'] > 0$, it indicates an candidate position $j = j' + b - 1$. Then, we verify the candidate position j as mentioned in previous section. We repeat the marking process for another b until all positions on T are covered. The process marks the array for $\log_\sigma \ell$ times.

Lemma 12. *Type 3 matches can be located in $O(m \log \sigma + m \log \ell + \gamma \log_\sigma \ell)$ time. The working space required is $O(m \log n + n \log \sigma)$ bits.*

References

1. Corasick, M., Aho, A.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Burrow, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California (1994)
3. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
4. Chazelle, B.: Filtering search: a new approach to query answering. SIAM J. Comput. 15(3), 703–724 (1986)
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of Symposium on Theory of Computing, pp. 91–100 (2004)
6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of Symposium on Foundations of Computer Science, pp. 390–398 (2000)
7. 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)
8. Fischer, M.J., Paterson, M.S.: String matching and other products. Technical Report MAC TM 41, Massachusetts Institute of Technology, Cambridge, MA, USA (January 1974)
9. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proceedings of Symposium on Theory of Computing, pp. 397–406 (2000)
10. Hon, W.K., Shah, R., Vitter, J.S., Lam, T.W., Tam, S.L.: Compressed index for dictionary matching. In: IEEE Data Compression Conference, pp. 23–32 (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. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. 39(1) (2007)
13. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Computational Geometry: Theory and Applications 42(4), 342–351 (2009)
14. Torczon, L., Briggs, P.: An efficient representation for sparse sets. In: ACM Letters on Programming Languages and Systems 2, pp. 59–69 (1993)
15. Rahman, M.S., Iliopoulos, C.S.: Pattern matching algorithms with don't cares. In: Proceedings of 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), vol. 2, pp. 116–126 (2007)

A Compressed Enhanced Suffix Array Supporting Fast String Matching

Enno Ohlebusch and Simon Gog

Institute of Theoretical Computer Science, University of Ulm, D-89069 Ulm
Enno.Ohlebusch@uni-ulm.de, Simon.Gog@uni-ulm.de

Abstract. Index structures like the suffix tree or the suffix array are of utmost importance in stringology, most notably in exact string matching. In the last decade, research on compressed index structures has flourished because the main problem in many applications is the space consumption of the index. It is possible to simulate the matching of a pattern against a suffix tree on an enhanced suffix array by using range minimum queries or the so-called *child table*. In this paper, we show that the Super-Cartesian tree of the LCP-array (with which the suffix array is enhanced) very naturally explains the child table. More important, however, is the fact that the balanced parentheses representation of this tree constitutes a very natural compressed form of the child table which admits to locate all *occ* occurrences of pattern P of length m in $O(m \log |\Sigma| + occ)$ time, where Σ is the underlying alphabet. Our compressed child table uses less space than previous solutions to the problem. An implementation is available.

1 Introduction

The suffix tree of a string S is an index structure that can be computed and stored in $O(n)$ time and space [1], where $n = |S|$. Once constructed, it can be used to efficiently solve a "myriad" of string processing problems [2,3]. Although being asymptotically linear, the space consumption of a suffix tree is quite large. This is a drawback in actual implementations. Thus, nowadays many string algorithms are based on suffix arrays and not on suffix trees. The suffix array specifies the lexicographic ordering of all suffixes of S, and it was introduced by Manber and Myers [4]. They showed that all *occ* occurrences of a pattern P of length m can be found in $O(m \log n + occ)$ time by binary search. Using additional information, this worst-case time complexity can be improved to $O(m + \log n + occ)$; see [4]. The suffix array can be compressed; see e.g. [5,6].

In another line of research, Abouelhoda et al. [7] introduced the concept of lcp-intervals in the LCP-array (the LCP-array stores the lengths of longest common prefixes of consecutive suffixes in the suffix array; it can also be compressed [8,9]) and showed that these form a virtual tree (called lcp-interval tree) which directly corresponds to the suffix tree of the string under consideration. To simulate the string matching of pattern P against a suffix tree, one must be able to solve the following problem efficiently: Given an lcp-interval $[i..j]$, find its child interval

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 51–62, 2009.
© Springer-Verlag Berlin Heidelberg 2009

(if it exists) that "starts" with a certain character a. The solution of Abouelhoda et al. [7] uses a so-called *child table*, which can be precomputed in linear time. Exact pattern matching then takes $O(m|\Sigma|+occ)$ time, where Σ is the underlying alphabet. Other researchers improved this result:

– Kim et al. [10] showed that pattern matching can be done in $O(m \log|\Sigma| + occ)$ time. This time can even be achieved with a compressed child table; see Kim and Park [11].
– Fischer and Heun [12] pointed out that the child table can be replaced by constant time range minimum queries, yielding an $O(m|\Sigma| + occ)$ time pattern search algorithm. Very recently, they showed that an improvement to $O(m \log|\Sigma| + occ)$ time is possible by finding range medians of minima queries, building on the new data structure *Super-Cartesian tree*; see [13].

In this paper, we show that the Super-Cartesian tree of the LCP-array naturally explains the child table (i.e., the introduction of the child table in [7] was not as simple as it could have been). More important, however, is the fact that the balanced parentheses representation of this tree constitutes a very natural compressed form of the child table which admits to locate all *occ* occurrences of pattern P in $O(m \log|\Sigma| + occ)$ time. Our compressed child table requires only $2n + o(n)$ bits, while Fischer and Heun's [13] approach takes $2.54n + o(n)$ bits and Kim and Park's [11] compressed child table requires $5n + o(n)$ bits.

As a matter of fact, the combination of a compressed enhanced suffix array (i.e., both the suffix array [5,6] and the LCP-array are compressed [8,9]) with the balanced parentheses representation of the Super-Cartesian tree of the LCP-array yields yet another compressed suffix tree with full functionality; see e.g. [8,9] and [14] for an overview of this field.

2 Preliminaries

Let S be a string of length n over the alphabet Σ. For every i, $1 \leq i \leq n$, S_i denotes the i-th suffix $S[i..n]$ of S. 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, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$. As already mentioned, the suffix array was introduced by Manber and Myers [4]. In 2003, it was shown independently and contemporaneously by Kärkkäinen & Sanders [15], Kim et al. [16], and Ko & Aluru [17] that a direct linear time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see the taxonomy by Puglisi et al. [18].

The *inverse suffix array* SA^{-1} is an array of size n such that for any q with $1 \leq q \leq n$ the equality $\mathsf{SA}^{-1}[\mathsf{SA}[q]] = q$ holds. Moreover, ψ is defined by $\psi[i] = \mathsf{SA}^{-1}[\mathsf{SA}[i] + 1]$ for all i with $1 \leq i \leq n$ if $\mathsf{SA}[i] \neq n$ and $\mathsf{SA}^{-1}[1]$ otherwise.

Let $\mathsf{lcp}(u, v)$ denote the longest common prefix between two strings u and v. The suffix array is often enhanced with the so-called LCP-array containing the lengths of longest common prefixes between consecutive suffixes in SA. Formally, the LCP-array is an array such that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n + 1]$ and $\mathsf{LCP}[i] =$

						CLD	
i	SA	LCP	PSV	NSV	$S_{\mathsf{SA}[i]}$	L	R
1	3	−1			*aaacatat*		11
2	4	2	1	3	*aacatat*		
3	1	1	1	7	*acaaacatat*	2	5
4	5	3	3	5	*acatat*		
5	9	1	1	7	*at*	4	6
6	7	2	5	7	*atat*		
7	2	0	1	11	*caaacatat*	3	9
8	6	2	7	9	*catat*		
9	10	0	1	11	*t*	8	10
10	8	1	9	11	*tat*		
11		−1				7	

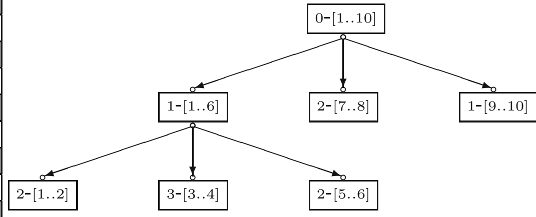


Fig. 1. The enhanced suffix array of the string $S = acaaacatat$ consists of the arrays SA and LCP. The corresponding lcp-interval tree is shown on the right.

$|\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$. Kasai et al. [19] showed that the LCP-array can be computed in linear time from the suffix array and its inverse.

According to [7], an interval $[i..j]$, where $1 \leq i < j \leq n$, in an LCP-array is called an *lcp-interval of lcp-value* ℓ (denoted by ℓ-$[i..j]$) if

1. $\mathsf{LCP}[i] < \ell$,
2. $\mathsf{LCP}[k] \geq \ell$ for all k with $i + 1 \leq k \leq j$,
3. $\mathsf{LCP}[k] = \ell$ for at least one k with $i + 1 \leq k \leq j$,
4. $\mathsf{LCP}[j + 1] < \ell$.

Every index k, $i + 1 \leq k \leq j$, with $\mathsf{LCP}[k] = \ell$ is called ℓ-index. Note that each lcp-interval has at most $|\Sigma| - 1$ many ℓ-indices.

An lcp-interval m-$[p..q]$ is said to be *embedded* in an lcp-interval ℓ-$[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq p < q \leq j$) and $m > \ell$. The interval $[i..j]$ is then called the interval *enclosing* $[p..q]$. If $[i..j]$ encloses $[p..q]$ and there is no interval embedded in $[i..j]$ that also encloses $[p..q]$, then $[p..q]$ is called a *child interval* of $[i..j]$. This parent-child relationship constitutes a tree which we call the *lcp-interval tree* (without singleton intervals). An interval $[k..k]$ is called *singleton interval.* The parent interval of such a singleton interval is the smallest lcp-interval $[i..j]$ which contains k.

The child intervals of an lcp-interval can be determined as follows. If $i_1 < i_2 < \ldots < i_k$ are the ℓ-indices of an lcp-interval ℓ-$[i..j]$ in ascending order, then the child intervals of $[i..j]$ are $[i..i_1 - 1]$, $[i_1..i_2 - 1], \ldots, [i_k..j]$ (note that some of them may be singleton intervals); see [7] for details. With range minimum queries on the LCP-array, ℓ-indices can be computed easily [12]: $\mathsf{RMQ}_{\mathsf{LCP}}(i + 1, j)$ returns the smallest index k such that $\mathsf{LCP}[k] = \min\{\mathsf{LCP}[q] \mid i + 1 \leq q \leq j\}$. Therefore, it returns the first ℓ-index i_1. Analogously, $\mathsf{RMQ}_{\mathsf{LCP}}(i_1 + 1, j)$ yields the second ℓ-index i_2, etc. In this way, one can simulate a top-down traversal of the lcp-interval tree, and exact string matching takes $O(m|\Sigma|)$ time in the worst case [7,12] because an array can be preprocessed in linear time so that range minimum queries can be answered in constant time [12,20,21].

The next lemma states how the parent interval of an lcp-interval can be determined; cf. [9]. For any index $2 \leq i \leq n$, define

$$\mathsf{PSV}[i] = \max\{j \mid 1 \leq j < i \text{ and } \mathsf{LCP}[j] < \mathsf{LCP}[i]\}$$
$$\mathsf{NSV}[i] = \min\{j \mid i < j \leq n+1 \text{ and } \mathsf{LCP}[j] < \mathsf{LCP}[i]\}$$

Lemma 1. *Let* ℓ-$[i..j]$ *be an lcp-interval with* $\mathsf{LCP}[i] = p$ *and* $\mathsf{LCP}[j+1] = q$. *If* $p \geq q$, *then the parent of* $[i..j]$ *is the lcp-interval* $[\mathsf{PSV}[i]..\mathsf{NSV}[i]-1]$. *Otherwise, if* $p < q$, *the parent of* $[i..j]$ *is the lcp-interval* $[\mathsf{PSV}[j+1]..\mathsf{NSV}[j+1]-1]$.

Given an lcp-interval $[i..j]$, the smallest lcp-interval $[p..q]$ satisfying $p \leq \psi[i] < \psi[j] \leq q$ is called the *suffix link interval* of $[i..j]$. For every lcp-interval ℓ-$[i..j]$ the suffix link interval exists and it has lcp-value $\ell - 1$; see [7] for details.

3 Finding Child Intervals without Range Minimum Queries

In the top-down traversal of the lcp-interval tree we actually do not need the rather complex machinery of constant-time range minimum queries. To see this, we first recall the definition of the Super-Cartesian tree from [13].

Definition 1. *Let* $A[l..r]$ *be an array of elements of a totally ordered set* $(S, <)$ *and suppose that the minima of* $A[l..r]$ *appear at positions* $p_1, p_2, \ldots, p_k$ *for some* $k \geq 1$. *The* Super-Cartesian tree $\mathcal{C}^{sup}(A[l..r])$ *of* $A[l..r]$ *is recursively constructed as follows:*

- *If* $l > r$, *then* $\mathcal{C}^{sup}(A[l..r])$ *is the empty tree.*
- *Otherwise create* k *nodes* $v_1, v_2, \ldots, v_k$, *label each* v_j *with* p_j, *and for each* j *with* $1 < j \leq k$ *the node* v_j *is the right sibling of node* v_{j-1} *(in Fig. 2, node* v_{j-1} *is connected with* v_j *by a* horizontal *edge). Node* v_1 *is the root of* $\mathcal{C}^{sup}(A[l..r])$. *Recursively construct* $\mathcal{C}_1 = \mathcal{C}^{sup}(A[l..p_1-1])$, $\mathcal{C}_2 = \mathcal{C}^{sup}(A[p_1+1..p_2-1])$, $\ldots$, $\mathcal{C}_{k+1} = \mathcal{C}^{sup}(A[p_k+1..r])$. *For each* j *with* $1 \leq j < k$, *the left child of* v_j *is the root of* $\mathcal{C}_j$. *The left and right children of* v_k *are the roots of* $\mathcal{C}_k$ *and* $\mathcal{C}_{k+1}$, *respectively.*

We would like to emphasize that a node in a Super-Cartesian tree has either a right sibling or a right child but not both. The Super-Cartesian tree $\mathcal{C}^{sup}(A)$ of an array A can be build incrementally in $O(n)$ time; see [13] for details. As an example, consider the enhanced suffix array of the string $S = acaaacatat$ in Fig. 1. The Super-Cartesian tree of this LCP-array is depicted in Fig. 2.

We store the Super-Cartesian tree in an additional table CLD, which we call *child table* because it can be used to determine child intervals. For didactic reasons, we will first store the child table CLD in two arrays CLD.L and CLD.R. We shall see in a moment, however, that one array suffices. By definition, a node in a Super-Cartesian tree has either a right sibling or a right child but not both. Therefore, for each node i, we store its left child in CLD$[i]$.L and its right

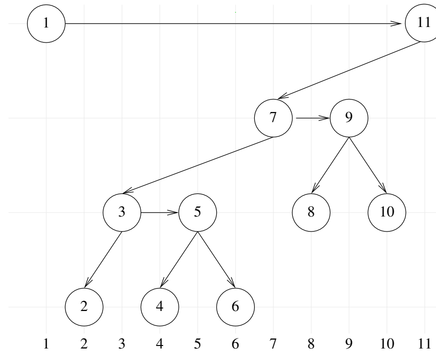


Fig. 2. The Super-Cartesian tree of the LCP-array from Fig. 1

sibling/right child in $\mathsf{CLD}[i].R$. For example, the child table CLD of the string $S = acaaacatat$ is depicted in Fig. 1.

As already mentioned, finding all child intervals of an lcp-interval ℓ-$[i..j]$ boils down to finding all ℓ-indices of that interval. The following theorem shows where the first ℓ-index of an lcp-interval ℓ-$[i..j]$ can be found in the child table.

Proposition 1. *For every lcp-interval ℓ-$[i..j]$ we have:*

1. *If $\mathsf{LCP}[i] \leq \mathsf{LCP}[j+1]$, then $\mathsf{CLD}[j+1].L$ stores the first ℓ-index of the lcp-interval $[i..j]$.*
2. *If $\mathsf{LCP}[i] > \mathsf{LCP}[j+1]$, then $\mathsf{CLD}[i].R$ stores the first ℓ-index of the lcp-interval $[i..j]$.*

Now we have all ingredients to realize a top-down traversal of the lcp-interval tree without range minimum queries. Proposition 1 tells us where the first ℓ-index, say i_1, of $[i..j]$ can be found. Using the child table, we find the second ℓ-index i_2 by $i_2 = \mathsf{CLD}[i_1].R$, the third ℓ-index i_3 by $i_3 = \mathsf{CLD}[i_2].R$, and so on. The index i_k is the last ℓ-index if $\mathsf{LCP}[i_{k+1}] \neq \ell$. Algorithm 1 implements this approach. The procedure *getChildIntervals* applied to an lcp-interval $[i..j]$ returns the list of all child intervals of $[i..j]$.

Of course, the Super-Cartesian tree is only conceptual, i.e., we can construct the child table without it. Algorithm 2 uses a stack to do this. The procedures *push* (pushes an element onto the stack) and *pop*() (pops an element from the stack and returns that element) are the usual stack operations, while *top*() provides a pointer to the topmost element of the stack. Moreover, *top*().*idx* denotes the first component of the topmost element of the stack, while *top*().*lcp* denotes the second component.

To reduce the space requirement of the child table, only one array is used in practice. As a matter of fact, the memory cells of $\mathsf{CLD}[i].R$, which are unused, can store the values of the $\mathsf{CLD}.L$ array. To see this, note that $\mathsf{CLD}[i+1].L \neq \perp$ if and only if $\mathsf{LCP}[i] > \mathsf{LCP}[i+1]$. In this case, however, we have $\mathsf{CLD}[i].R = \perp$. In other words, $\mathsf{CLD}[i].R$ is empty and can store the value $\mathsf{CLD}[i+1].L$; see

Algorithm 1. *getChildIntervals* applied to an lcp-interval $[i..j]$.

```
intervalList = [ ]
k ← i
if LCP[i] ≤ LCP[j + 1] then
   m ← CLD[j + 1].L
else m ← CLD[i].R
ℓ ← LCP[m]
repeat
   add(intervalList, [k..m − 1])
   k ← m
   m ← CLD[m].R
until m = ⊥ or LCP[m] ≠ ℓ
add(intervalList, [k..j])
```

Algorithm 2. Construction of the child table.

```
push(⟨1, −1⟩)          /* an element on the stack has the form ⟨idx, lcp⟩ */
for k ← 2 to n + 1 do
   while LCP[k] < top().lcp do
      last ← pop()
      while top().lcp = last.lcp do
         CLD[top().idx].R ← last.idx
         last ← pop()
      if LCP[k] < top().lcp then
         CLD[top().idx].R ← last.idx
      else CLD[k].L ← last.idx
   push(⟨k, LCP[k]⟩)
```

Fig. 1. Finally, for a given index i, one can decide whether $\mathsf{CLD}[i].R$ contains the value $\mathsf{CLD}[i+1].L$ by testing whether $\mathsf{LCP}[i] > \mathsf{LCP}[i+1]$. To sum up, although the child table conceptually uses two arrays, only space for one array is actually required.

4 Balanced Parentheses Representation of the Tree

The Super-Cartesian tree of the LCP-array can be represented by a sequence of balanced parentheses; see Fig. 3. Again, it turns out that the Super-Cartesian tree is only conceptual. To be precise, its balanced parentheses representation can be obtained solely based on the LCP-array; see Algorithm 3.

Each node k, $1 \leq k \leq n$, in the Super-Cartesian tree is represented by the k-th opening parenthesis (and the matching closing parenthesis). Node $n+1$ is not represented. Consequently, the sequence of balanced parentheses has $2n$ parentheses, and it can be represented with $2n$ bits.

(	(	)	(	(	)	(	(	)	)	)	(	(	)	(	(	)	)	)	)
1	2	2	3	4	4	5	6	6	5	3	7	8	8	9	10	10	9	7	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Fig. 3. Balanced parentheses representation of the Super-Cartesian tree of Fig. 2

Algorithm 3. Construction of the balanced parentheses representation of the Super-Cartesian tree of the LCP-array.

```
push(−1)          /* LCP[1] = −1 */
write an opening parenthesis
for k ← 2 to n + 1 do
   while LCP[k] < top() do
      pop() and write a closing parenthesis
   if k ≠ n + 1 then
      push(LCP[k]) and write an opening parenthesis
   else
      write a closing parenthesis
```

Given a balanced parentheses sequence, the following operations can be supported in constant time with only $o(n)$ bits of extra space [22,23,24,25]:

- $rank_($$(i)$: returns the number of opening parentheses up to and including position i; see Jacobson [22]. $rank_)(i)$ is defined analogously.
- $select_($$(i)$: returns the position of the i-th opening parenthesis; see Clark [23]. $select_)(i)$ is defined analogously.
- $findclose(i)$: returns the position of the closing parenthesis matching the opening parenthesis at position i; see Munro & Raman [24]. $findopen(i)$ is defined analogously.
- $enclose(i)$: given a parenthesis pair whose opening parenthesis is at position i, it returns the position of the opening parenthesis corresponding to the closest matching parenthesis pair enclosing i; see Munro & Raman [24].

Geary et al. [25] provide a simpler $o(n)$ extra space solution for $findclose$, $findopen$, and $enclose$. In our implementation, we use a data structure which is similar to that of [25]. That is, our implementation needs $2n + o(n)$ bits to support all operations in constant time.

As we have seen, determining the child intervals of an ℓ-interval $[i..j]$ boils down to finding the ℓ-indices of $[i..j]$ in ascending order. On the balanced parentheses representation of the Super-Cartesian tree of the LCP-array these can be found as follows.

Lemma 2. *With the balanced parentheses representation, the first ℓ-index k of an lcp-interval $[i..j]$ can be determined in constant time by*

$$k = \begin{cases} rank_((findopen(select_((j+1)-1)) & , \text{ if } \mathsf{LCP}[i] \le \mathsf{LCP}[j+1] \\ rank_((findopen(findclose(select_((i))-1)), & \text{ if } \mathsf{LCP}[i] > \mathsf{LCP}[j+1] \end{cases}$$

Proof. If $\mathsf{LCP}[i] \leq \mathsf{LCP}[j+1]$, then k is the left child of $j+1$ in the Super-Cartesian tree of the LCP-array. In the balanced parentheses representation, the closing parenthesis matching the k-th opening parenthesis is directly followed by the $(j+1)$-th opening parenthesis. Therefore, $k = rank_{(}(findopen(select_{(}(j+1)-1))$. If $\mathsf{LCP}[i] > \mathsf{LCP}[j+1]$, then k is the right child of i in the Super-Cartesian tree of the LCP-array. In the balanced parentheses representation, the closing parenthesis matching the k-th opening parenthesis is directly followed by the closing parenthesis matching the i-th opening parenthesis. Thus, $k = rank_{(}(findopen(findclose(select_{(}(i)) - 1))$.

If we know an ℓ-index k of an lcp-interval $[i..j]$, then the next ℓ-index m (if it exists) is the right sibling of k in the Super-Cartesian tree of the LCP-array. In the balanced parentheses representation, the closing parenthesis matching the m-th opening parenthesis is directly followed by the closing parenthesis matching the k-th opening parenthesis. So $m = rank_{(}(findopen(findclose(select_{(}(k)) - 1))$.

Given an ℓ-index k of an lcp-interval $[i..j]$, an algorithm that computes the next ℓ-index m (if it exists) works as follows. First it tests whether $select_{(}(k) \neq findclose(select_{(}(k)) - 1$. If this is the case, then k is not a leaf in the Super-Cartesian tree of the LCP-array (i.e., the k-th opening parenthesis is not directly followed by a closing parenthesis) and the algorithm further computes $m = rank_{(}(findopen(findclose(select_{(}(k)) - 1))$. If $\mathsf{LCP}[k] = \mathsf{LCP}[m]$, then it returns m as the next ℓ-index of $[i..j]$. Otherwise, there is no next ℓ-index. It follows as a consequence that all child intervals of an lcp-interval $[i..j]$ can be determined in $O(|\Sigma|)$ time solely based on the balanced parentheses representation of the Super-Cartesian tree of the LCP-array. That is, one neither needs range minimum queries nor the child table.

To exemplify our method, we search for the first ℓ-index k of the lcp-interval 1-$[1,6]$ (see Fig. 1): As $\mathsf{LCP}[1] = -1 \leq \mathsf{LCP}[7] = 0$, k is the left child of node 7 in the Super-Cartesian tree of Fig. 2. In the balanced parentheses sequence, we obtain the position of the 7th opening parenthesis by $select_{(}(7) = 12$; see Fig. 3. The left child of node 7 is represented by the opening parenthesis matching the closing parenthesis at position $12-1 = 11$, and this opening parenthesis is found at position $findopen(11) = 4$. Since $rank_{(}(4) = 3$, we conclude that $k = 3$. The next ℓ-index (if it exists) corresponds to the right sibling of node $k = 3$ in the Super-Cartesian tree. If node $k = 3$ is not a leaf, then the parenthesis directly left to the closing parentheses corresponding to k is also a closing parenthesis. In our example this is the case and therefore $rank_{(}(findopen(10)) = 5$ tells us that node 5 is either a sibling or a child of node $k = 3$. Because $\mathsf{LCP}[5] = \mathsf{LCP}[3]$, node 5 is the right sibling. Hence 5 is the next ℓ-index.

In string matching, we search for a specific child interval. To be precise, if $[i..j]$ is an lcp-interval that represents a string ω, we wish to compute the lcp-interval $[i'..j']$ that represents the string ωa for some character $a \in \Sigma$. Clearly, we can enumerate all child intervals $[i'..j']$ of $[i..j]$ until the one with $S[\mathsf{SA}[i'] + |\omega|] = \ldots = S[\mathsf{SA}[j'] + |\omega|] = a$ is found. This takes $O(|\Sigma|)$ time in the worst case. As a matter of fact, the balanced parentheses representation allows us to determine such an interval in $O(\log |\Sigma|)$ time. This goes as follows. The left

boundary of the interval we are searching for is either i or one of the ℓ-indices of the lcp-interval $[i..j]$. Thus, if $S[\mathsf{SA}[i] + |\omega|] = a$, we are done. Otherwise, one determines the first ℓ-index k as in Lemma 2. If $S[\mathsf{SA}[k] + |\omega|] = a$, we are done. If not, we determine the position p of the closing parenthesis matching the k-th opening parenthesis by $p = findclose(select_((k))$. The key observation is that the remaining ℓ-indices of $[i..j]$ (there are at most $|\Sigma| - 2$ many) are the siblings of node k in the Super-Cartesian tree and—by construction—the closing parentheses immediately preceding the closing parenthesis at position p correspond to these ℓ-indices. Therefore, a binary search on the matching opening parentheses of the first $|\Sigma| - 2$ closing parentheses immediately preceding the closing parenthesis at position p (if there are so many closing parentheses at all), can be used to find the desired interval. First check whether the index m under consideration satisfies $\mathsf{LCP}[m] = \mathsf{LCP}[k]$. If not, we have to search in the right half. If so, m is another ℓ-index and one further compares $S[\mathsf{SA}[m] + |\omega|]$ with the character a. If the characters coincide, then m is the left boundary of the interval we are searching for. If $S[\mathsf{SA}[m] + |\omega|] < a$, we have to search in the right half, and if $S[\mathsf{SA}[m] + |\omega|] > a$, we have to search in the left half.

5 Full Functionality

The balanced parentheses representation of the Super-Cartesian tree of the LCP-array supports all operations of a suffix tree as listed e.g. in [8,9]. Here we show how the following two crucial operations can be implemented (the other operations are rather straightforward; cf. [9]):

– $parent([i..j])$: returns the parent interval of the lcp-interval $[i..j]$.
– $slink([i..j])$: returns the suffix link interval of the lcp-interval $[i..j]$.

According to Lemma 1, the parent interval of an lcp-interval can be determined with the help of PSV and NSV-values. The next lemma shows how these values can be computed on the balanced parentheses representation.

Lemma 3. *Let i be an index with $\mathsf{LCP}[i] \neq -1$. With the balanced parentheses representation of the Super-Cartesian tree of the* LCP*-array,* $\mathsf{NSV}[i]$ *can be determined in constant time by* $\mathsf{NSV}[i] = rank_((findclose(select_((i))) + 1$ *and* $\mathsf{PSV}[i]$ *can be computed in $O(|\Sigma|)$ time by*

$j \leftarrow rank_((enclose(select_((i)))$
while $\mathsf{LCP}[j] = \mathsf{LCP}[i]$ **do**
  $j \leftarrow rank_((enclose(select_((j)))$
$\mathsf{PSV}[i] \leftarrow j$

It is also possible to compute $\mathsf{PSV}[i]$ in $O(\log |\Sigma|)$ time by a binary search on the balanced parentheses sequence (similar to the method described above). Consequently, the parent interval of an lcp-interval can be found in $O(\log |\Sigma|)$ time by Lemma 1.

The suffix link interval $[p..q]$ of an lcp-interval ℓ-$[i..j] \neq 0$-$[1..n]$ can be determined as follows: First, the range minimum query $\mathsf{RMQ}_{\mathsf{LCP}}(\psi[i] + 1, \psi[j])$ yields

an $(\ell-1)$-index of $[p..q]$ (see [7] for details), say index k, and then the boundaries of the suffix link interval are determined by $p = \mathsf{PSV}[k]$ and $q = \mathsf{NSV}[k]-1$ (this is a direct consequence of the definition of an lcp-interval). Thus, $[p..q]$ can be computed in $O(\log|\Sigma|)$ time provided that range minimum queries can be answered in constant time. However, the ability to answer range minimum queries in constant time requires a different data structure, and this is disadvantageous in practice. To compute suffix links on our data structure, we introduce the new operation *range-restricted-enclose* (*rr-enclose* for short) on balanced parentheses sequences: Given two opening parentheses at positions i and j such that $findclose(i) < j$, the operation $rr\text{-}enclose(i,j)$ returns the smallest position, say k, of an opening parenthesis such that $findclose(i) < k < j$ and $findclose(j) < findclose(k)$. If such a k does not exist, it returns $\perp$.

Theorem 1. *Given the balanced parentheses representation of the Super-Cartesian tree of the* LCP*-array and two indices i and j with $1 \leq i \leq j \leq n$, let $i' = select_{(}(i)$ and $j' = select_{(}(j)$. Then*

$$\mathsf{RMQ_{LCP}}(i,j) = \begin{cases} i, & \text{if } j' < findclose(i') \\ j, & \text{if } findclose(i') < j' \text{ and } rr\text{-}enclose(i',j') = \perp \\ rank_{(}(rr\text{-}enclose(i',j')), & \text{otherwise} \end{cases}$$

Proof. We use a case differentiation. If $j' < findclose(i')$, then $i' < j' < findclose(j') < findclose(i')$, i.e., the parenthesis pair with opening parenthesis at position i encloses the other parenthesis pair. This, in turn, means that $\mathsf{LCP}[i] \leq \mathsf{LCP}[q]$ for all q with $i < q \leq j$. Hence $\mathsf{RMQ_{LCP}}(i,j) = i$. Otherwise, we have $findclose(i') < j'$. If $rr\text{-}enclose(i',j') = \perp$, then there is no parenthesis pair with opening parenthesis at a position $> findclose(i)$ that encloses the parenthesis pair with opening parenthesis at position j. This means that the parenthesis pairs are "siblings". In other words, $\mathsf{LCP}[i]$ is a successor of $\mathsf{LCP}[j]$ on a "left path" in the Super-Cartesian tree of the LCP-array. It follows as a consequence that $\mathsf{LCP}[q] > \mathsf{LCP}[j]$ for all q with $i \leq q < j$. Thus, $\mathsf{RMQ_{LCP}}(i,j) = j$. In the last case $rr\text{-}enclose(i',j') = k$, where k is the smallest position of an opening parenthesis such that $findclose(i') < k < j'$ and $findclose(j') < findclose(k)$. So we have $i' < findclose(i') < k < j' < findclose(j') < findclose(k)$. In the Super-Cartesian tree of the LCP-array, this corresponds to $\mathsf{LCP}[q] > \mathsf{LCP}[k]$ for all q with $i \leq q < k$ and $\mathsf{LCP}[k] \leq \mathsf{LCP}[q]$ for all q with $k < q \leq j$. Therefore, $\mathsf{RMQ_{LCP}}(i,j) = k$.

6 Conclusions

The methods described above have been implemented in *C++*, and the implementation is available under the GNU General Public License. In our opinion, the main advantage of the balanced parentheses representation of the Super-Cartesian tree of the LCP-array is that child intervals can be computed efficiently. Thus, it would be natural to compare our implementation experimentally with

the related methods of Kim and Park [11] and of Fischer and Heun [13]. Unfortunately, for both methods no implementation is available.

On the other hand, our compressed enhanced suffix array has full functionality, i.e., it supports all operations of a suffix tree and it is natural to compare it experimentally with implementations of compressed suffix trees with such a full functionality. Välimäki et al. [26] implemented Sadakane's compressed suffix tree [8], and to the best of our knowledge this is the sole implementation which is available. For a fair comparison, however, both implementations should use the same compressed suffix array and the same compressed LCP-array. We are currently working on our own implementation of Sadakane's compressed suffix tree [8] and an experimental comparison is forthcoming.

First experiments with texts of size 50MB show that the $O(\log|\Sigma|)$-time version of the method that determines a specific child interval is two times faster than the $O(|\Sigma|)$-time version for $20 < |\Sigma| < 30$ and up to 10 times faster for $90 < |\Sigma| < 230$. Unsurprisingly, for $|\Sigma| = 4$ (DNA-alphabet) the $O(\log|\Sigma|)$-time version is (slightly) slower than the $O(|\Sigma|)$-time version.

References

1. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Annual Symposium on Switching and Automata Theory, pp. 1–11 (1973)
2. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words, pp. 85–96. Springer, Heidelberg (1985)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
4. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
5. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. ACM Symposium on the Theory of Computing, pp. 397–406. ACM Press, New York (2000)
6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
7. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
8. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41, 589–607 (2007)
9. Fischer, J., Navarro, G., Mäkinen, V.: An(other) entropy bounded compressed suffix tree. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 152–165. Springer, Heidelberg (2008)
10. Kim, D., Jeon, J., Park, H.: An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 138–149. Springer, Heidelberg (2004)
11. Kim, D., Jeon, J., Park, H.: A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 33–44. Springer, Heidelberg (2005)

12. 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)
13. Fischer, J., Heun, V.: Range median of minima queries, super-cartesian trees, and text indexing. In: Proc. 19th International Workshop on Combinatorial Algorithms, pp. 239–252. College Publications (2008)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39 (2007)
15. 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)
16. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
17. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
18. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
19. Kasai, T., Lee, G., 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)
20. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13, 338–355 (1984)
21. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing 17, 1253–1262 (1988)
22. 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)
23. Clark, D.: Compact Pat Trees. PhD thesis, University of Waterloo (1996)
24. Munro, J., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
25. Geary, R., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
26. Välimäki, N., Gerlach, W., Dixit, K., Mäkinen, V.: Engineering a compressed suffix tree implementation. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 217–228. Springer, Heidelberg (2007)

Compressed Suffix Arrays for Massive Data

Jouni Sirén*

Department of Computer Science, University of Helsinki, Finland
jltsiren@cs.helsinki.fi

Abstract. We present a fast space-efficient algorithm for constructing compressed suffix arrays (CSA). The algorithm requires $O(n \log n)$ time in the worst case, and only $O(n)$ bits of extra space in addition to the CSA. As the basic step, we describe an algorithm for merging two CSAs. We show that the construction algorithm can be parallelized in a symmetric multiprocessor system, and discuss the possibility of a distributed implementation. We also describe a parallel implementation of the algorithm, capable of indexing several gigabytes per hour.

1 Introduction

Self-indexing [23] is a new approach for storing sequence data. The main idea is to combine the data and its index in a compressed structure, which provides random access to the data and supports various pattern matching queries. Some of the most relevant self-indexes are the *compressed suffix array (CSA)* [12] and the *FM-index* [8], both offering suffix array-like functionality.

With the explosive growth of sequential data in many applications such as genome browsers, version control systems, and online document collections, good search capabilities are becoming more and more important every day. This trend is making the self-indexes, combining small size with full-text searching, a promising approach for indexing large and massive data sets.

Obviously we need efficient practical algorithms for constructing these self-indexes, if we want them to truly live up to their promises. Unfortunately all the experiments reported so far have been performed with data sets at most a few gigabytes in size [5, 7, 14, 16, 17, 24], telling that the construction algorithms have trouble scaling up for massive data sets.

The typical way to construct a compressed self-index has been to use a regular suffix array construction algorithm [24]. While these algorithms are fast, they must store the data and the suffix array in main memory, making the memory requirements many times the size of the data. This is a major problem, especially with highly repetitive collections [21, 26], where the final index can be more than a hundred of times smaller than the suffix array.

Other alternatives have been to use secondary memory suffix array construction algorithms [4, 5], dynamic indexes [3, 11, 19, 20, 25], or algorithms for constructing the compressed index directly [13, 15, 22]. While these algorithms are

* Funded by the Academy of Finland under grant 119815. Part of the work was done while visiting NICTA Neville Roach Laboratory.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 63–74, 2009.
© Springer-Verlag Berlin Heidelberg 2009

often memory efficient, they are also slow. Experiments have reported throughputs in the order of 100 kilobytes/second, which is more than an order of magnitude slower than the regular suffix array construction algorithms, and clearly too slow for data sets of tens of gigabytes or more.

The most promising algorithms are the distributed suffix array construction algorithm by Kulla et al. [17] and the space-efficient Burrows-Wheeler transform construction algorithm by Kärkkäinen [16]. Still, we must store either the suffix array in distributed memory or the entire data set in local memory, making both of the algorithms unsatisfactory for highly compressible data sets.

In this paper, we present a fast and space-efficient algorithm for direct CSA construction. The algorithm is related to the incremental suffix array construction algorithm by Gonnet et al. [10], as well as to the incremental CSA construction algorithm by Hon et al. [13]. Alternatively our algorithm can be thought of as replacing a dynamic CSA with a static structure and batch updates.

We start by some basic definitions in Sect. 2. Section 3 describes an algorithm for merging two compressed suffix arrays. Section 4 builds upon it, describing a parallelizable incremental CSA construction algorithm. The details of our implementation of the algorithm are discussed in Sect. 5. In Sect. 6, we validate the effectiveness of our algorithm experimentally. Finally, we discuss the possibility of a distributed implementation in Sect. 7.

2 Background Information

A *string* $S = S_{1,n} = s_1 s_2 \cdots s_n$ is a *sequence* of *symbols* (characters, letters). Each symbol is an element of an *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written as $S_{i,j} = s_i \cdots s_j$. A substring of type $S_{1,j}$ is called a *prefix*, while a substring of type $S_{i,n}$ is called a *suffix*. We often assume that the string is an array, and refer to its symbols as $S[i]$, and to its substrings as $S[i, j]$. A *text* string $T = T_{1,n}$ is a sequence terminated by $t_n = \$ \notin \Sigma$ smaller than any symbol in Σ. The *lexicographic order* "$<$" among strings is defined in the usual way.

We call a set $\mathcal{C}$ of texts $T^1, T^2, \ldots, T^r$ a *collection*. The collection can be represented as a string $\mathcal{T} = T^1 T^2 \cdots T^r$. We denote the length of each text T^i as n_i, and the total length of the collection as $|\mathcal{C}| = |\mathcal{T}| = n$. Lexicographic order among such strings is defined in the usual way, except that each of the end markers \$ is considered a different symbol, so that every suffix of every string will be unique in the collection. If $\mathcal{T}[i] = \mathcal{T}[j] = \$$ and $i < j$, we define $\mathcal{T}[i] < \mathcal{T}[j]$. We informally call a collection *highly repetitive*, if most of its texts are highly similar to some other text in the collection. Examples of highly repetitive collections include individual genomes and different versions of a document.

The *suffix array* $SA[1, n]$ of a string S is an array of pointers to the suffixes of S in lexicographic order. As an abstract data type, a suffix array is any data structure providing similar functionality as the concrete suffix array. This can be defined by the following operations: (a) *count* the number of occurrences of a *pattern* in the string; (b) *locate* these occurrences (or more generally retrieve a suffix array value); and (c) *display* any substring of S.

The compressed suffix arrays discussed in this paper support these operations. Their compression is based on the *Burrows-Wheeler transform (BWT)* [2], a permutation created by sorting *cyclical strings*. The cyclical strings corresponding to string S are all strings of the form $CS_i = S_{i,n}S_{1,i-1}$, including $CS_1 = S$. The BWT is a sequence L such that $L[i] = CS_j[n]$, where CS_j is the ith cyclical string in lexicographic order. If S is a text or a collection, sorting cyclical strings is the same as sorting suffixes, as the first end marker encountered will end the comparison between two cyclical strings. In that case the BWT can be defined as $L[i] = S[SA[i] - 1]$, where $S[0] = S[n]$.

The Burrows-Wheeler transform is reversible. The reverse transform is based on a function called *LF-mapping* [2, 8] that is also used extensively in compressed self-indexes. The mapping is usually described by using an array $C[1, \sigma]$ such that $C[c]$ is the number of characters in $\{\$, 1, 2, \ldots, c - 1\}$ occurring in the collection. With this array and the sequence L, we can define LF-mapping as $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$, where $rank_c(L, i)$ is the number of occurrences of character c in the prefix $L[1, i]$. We leave $LF(i)$ undefined when $L[i] = \$$, as each \$ is actually a different character. This is not a problem, as LF-mapping is not used for these positions in CSA operations.

LF-mapping and its inverse function Ψ [12] form the backbone of many compressed self-indexes. As $SA[LF(i)] = SA[i] - 1$ [8] and hence $SA[\Psi(i)] = SA[i] + 1$, we can use these functions to move the suffix array position backward and forward in the sequence. Both of the functions can be efficiently implemented by adding some extra information to a compressed representation of the BWT. Standard techniques [23] to support suffix array operations by using these functions include *backward searching* [8] for *count*, and adding a sample of suffix array values for *locate* and *display*.

The regular BWT is based on the cyclical strings of a single string. In this paper, we generalize the transform by allowing multiple strings, each of which can be a concatenation of several texts [9]. This makes it easier to merge the BWTs of two collections. We call the way the texts of a collection $\mathcal{A}$ are concatenated to form strings the *structure* of $\mathcal{A}$. Collection $\mathcal{B}$ contains the structure of $\mathcal{A}$, if $\mathcal{A} \subseteq \mathcal{B}$ and the texts of $\mathcal{A}$ are concatenated to form the same strings in the structures of $\mathcal{A}$ and $\mathcal{B}$.

The position of a character $T^i[j] \neq \$$ in the BWT is determined by the cyclical string of the string containing it starting at $T^i[j + 1]$. As the first end marker encountered ends any comparison, we only need the suffix $T^i[j + 1, n_i]$ to determine the position, as with the regular BWT. The position of the end marker $T^i[n_i]$ is determined by the text $T^{i'}$ following T^i in the cyclical strings. As each text is used to determine the position of exactly one end marker in the BWT, the structure of a collection does not affect its BWT.

3 Merging Compressed Suffix Arrays

Consider the collection $\{T^1, T^2\}$, where $T^1 = ababbaa\$$ and $T^2 = abbaa\$$. The BWTs of the texts are $aab\$bbaa$ and $aab\$ba$, respectively. Figure 1 shows a

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathcal{T}$	a	b	a	b	b	a	a	$\$$	a	b	b	a	a	$\$$
SA	8	14	7	13	6	12	1	3	9	5	11	2	4	10
I	0	1	0	1	0	1	0	0	1	0	1	0	0	1
L	a	a	a	a	b	b	$\$$	b	$\$$	b	b	a	a	a
Ψ_a	1	1	1	1	0	0	0	0	0	0	0	1	1	1
Ψ_b	0	0	0	0	1	1	0	1	0	1	1	0	0	0

Fig. 1. A generalized Burrows-Wheeler transform L and the suffix array SA of collection $\{T^1, T^2\}$. Note that T^1 and T^2 are separate strings in the structure of the collection. One can get the regular BWT by changing which \$ belongs to which text.

generalized Burrows-Wheeler transform of the collection, where the characters of text T^2 are marked with 1-bits in bit vector I.

We see that the marked characters form the BWT of T^2, while the other characters form the BWT of T^1. This is true in general as well. Assume we have two collections $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{B}$ contains the structure of $\mathcal{A}$. As the position of each character of $\mathcal{A}$ is determined by the same cyclical string in the BWTs of $\mathcal{A}$ and $\mathcal{B}$, the BWT of $\mathcal{A}$ is a *subsequence* of the BWT of $\mathcal{B}$.

Now let us turn our attention to the bit vectors Ψ_c marking the occurrences of character $c \in \Sigma$ in L. These vectors completely describe the BWT of the collection. We can perform the $rank_c(L, i)$ used in LF-mapping as $rank_1(\Psi_c, i)$. We can also compute $\Psi(j)$ as $select_1(\Psi_c, j - C[c])$, where $C[c] < j \leq C[c+1]$ and $select_1(\Psi_c, i)$ returns the position of the ith 1-bit in Ψ_c. Hence we can implement a self-index by compressing the bit vectors Ψ_c.

In fact, that is exactly what compressed suffix arrays [12], based on the function Ψ, already do. As the values of Ψ form an increasing sequence in the region of the suffix array corresponding to a character c, any representation of that part of Ψ is also a representation of the bit vector Ψ_c.

This gives us an idea for an algorithm to merge two compressed suffix arrays. If we have the CSAs of collections $\mathcal{C}_1$ and $\mathcal{C}_2$ and the bit vector I, we can use them to build a CSA for the combined collection. For each $c \in \Sigma$, we simply take the Ψ_c vectors of the two CSAs and merge them. Vector I is used to indicate how to interleave the bits from the two vectors. Sampled suffix array positions can be merged in a similar manner.

Let $n_i = |\mathcal{C}_i|$ and $n = n_1 + n_2$. With a suitable representation of the bit vectors (as in Sect. 5), we can merge the CSAs in-place in $O(|\text{CSA}| + n_2\sigma)$ time, where $|\text{CSA}|$ is the size of the resulting CSA. The $O(n_2\sigma)$ part comes from the fact that we have to scan the bit vector I once for every pair of vectors merged. This is not very efficient for large alphabets.

In such situations, it is better to merge the BWTs instead of the bit vectors. We can read the BWT of collection $\mathcal{C}_i$ from its CSA in $O(n_i)$ time by using a

buffer of $\Omega(\sigma)$ characters. If we decompress both of the BWTs simultaneously, merge the buffers, and write the results immediately to the combined CSA, we can perform the merge in $O(n)$ time and $O(\sigma \log n)$ extra space for the buffer and bookkeeping.

The remaining question is, how to construct the bit vector I, denoting the ranks of every suffix of $\mathcal{B}$ in the combined suffix array. As the rank of a suffix is the sum of its ranks among the suffixes of $\mathcal{A}$ and $\mathcal{B}$ [13], we get the following algorithm for merging two CSAs:

1. **Search** for the ranks of the suffixes of $\mathcal{B}$ among the suffixes of $\mathcal{A}$ by backward searching [13]. Store the ranks in an integer array in any order.
2. **Sort** the array. Increment the values by their positions in the array (by the ranks of the suffixes of $\mathcal{B}$ among themselves) to get I.
3. **Merge** the BWTs of the two CSAs.

Searching takes $O(n_2 t_\Psi)$ time, where t_Ψ is the cost of one access to Ψ. We are not aware of any upper bounds better than $t_\Psi = O(\log n_1)$ (as in Sect. 5 with a logarithmic value for B) for CSAs that allow efficient merging. Array I requires $O(n_2 \log n)$ bits of space, and sorting it takes $O(n_2 \log n_2)$ time. Hence the entire algorithm takes $O(|\text{CSA}| + n_2(\sigma + \log n))$ or $O(n + n_2 \log n)$ time, and works in $|\text{CSA}| + O(n_2 \log n)$ or $|\text{CSA}| + O((n_2 + \sigma) \log n)$ bits of space, respectively, with regular and BWT-based merging.

A similar algorithm can be used to remove sequences from the collection. We search for the positions of the suffixes to be removed, marking them on a bit vector I. Then we scan the bit vectors Ψ_c, removing bits as indicated by I.

4 CSA Construction

The algorithm for merging two compressed suffix arrays can be used as a building block for a CSA construction algorithm. The basic idea is to divide the collection into smaller ones, each of which can be indexed in limited memory, build CSAs for the parts, and merge the resulting partial indexes by using the algorithm in the previous section. For each part of the input, we first execute the *build* phase:

1. **Build** a CSA for the current input collection.

Then we merge the resulting partial index to the existing CSA by executing the *search*, *sort*, and *merge* phases.

Assume a collection of size n has been split into p parts of size n/p. Then, with any $O(n \log n)$ time and space suffix array construction algorithm, the *build* phases take a total of $O(n \log(n/p))$ time and require $|\text{CSA}| + O((n/p) \log(n/p))$ bits of space. By using BWT-based merging, the other three phases require $O(pn + n \log n)$ time and $|\text{CSA}| + O((n/p + \sigma) \log n)$ bits of space. If we assume $p = \Theta(\log n)$ and $\sigma = O(n/p)$, we get an algorithm requiring $O(n \log n)$ time and $|\text{CSA}| + O(n)$ bits of space.

The algorithm can be parallelized with the following modifications:

1. **Build.** We can either build indexes for multiple input collections in parallel, increasing memory usage, or use a parallel suffix array construction algorithm such as [17].
2. **Search.** The ranks of the suffixes of a text are independent from the other texts in the input collection. Hence we can perform the search for multiple texts in parallel. If there are too few texts to distribute the searches evenly, we can try to split a search into multiple smaller ones. Assume we are searching for the ranks of the suffixes of text T backwards from position $T[j]$. If we find a substring $T[i, j]$ with no occurrences in the index, we can start reporting the ranks, as the symbols after $T[j]$ do not affect them.
3. **Sort.** Use a parallel sorting algorithm.
4. **Merge.** Multiple bit vector pairs can be merged in parallel. If there are more processors than bit vectors, work can be divided by splitting the vectors into multiple parts.

5 Implementation

We have implemented a sequential version of the algorithm, as well as a parallel version for *symmetric multiprocessor (SMP)* systems.[1] The implementation is written in in C++. The input is assumed to be divided into a number of files, each of them consisting of concatenated C-style 0-terminated strings. Each string is considered a separate text, with the trailing 0 interpreted as an end marker. The *build* phase is executed for all input files in the beginning of the construction to save memory. The resulting partial indexes as well as unused parts of the input are stored in secondary memory until needed.

We use two kinds of bit vectors in the implementation: gap encoded and run-length encoded. In gap encoding, the vector is encoded as a sequence of integers denoting the distances between the successive 1-bits, while in run-length encoding each run of 1s is encoded as the gap after the previous run followed by the length of the run. In both cases, δ codes [6] are used to encode the integers.

The compressed bit vectors are divided into blocks of B bytes. For each block, we sample the first 1-bit in the block, writing down its rank and position in the vector. Each sample takes $2 \log u$ bits, where u is the length of the vector. By using these samples, we can determine, which block to decompress to answer bit vector operations such as *rank* and *select*.

As a binary search among the samples is quite slow, we speed up the search by constructing secondary indexes for *rank* and *select* when the vector is loaded into memory. Both indexes consist of about $b/5$ integers of $\log b$ bits, where b is the number of blocks in the vector. For *rank*, the ith value is the number of the the block storing the first 1-bit at or after position $i \cdot 5u/b$. For *select*, the jth value is similarly the number of the block storing the 1-bit of rank $j \cdot 5n_o/b$,

[1] The implementation is available at
`http://www.cs.helsinki.fi/group/suds/rlcsa/`

where n_o is the number of 1s in the vector. By using these indexes, we can limit the search to a (typically) small number of samples.

Instead of a compressed bit vector, we use a simpler structure as the indicator vector I in the *merge* phase. This structure is just an array of native 32-bit or 64-bit integers in increasing order, each of them indicating a 1-bit in the vector.

Our implementation of CSA is based on the Run-Length Compressed Suffix Array [21, 26]. We use a run-length encoded bit vector to represent each Ψ_c. This makes the index most suitable for highly repetitive collections, while some compression is lost on other types of collections.

Suffix array samples are marked in a gap encoded bit vector and stored as $\log(n/d)$-bit integers, where n is the size of the collection and d is the sample rate. Inverse suffix array samples used in *display* are constructed when the index is loaded, and stored as another array of $\log(n/d)$-bit integers. The end points of all sequences in the collection are marked in a gap encoded bit vector E.

The implementation supports multiple parallel queries. Each thread using the CSA maintains separate state information, while large arrays, such as samples and bit vector blocks, are shared between the threads. Large queries are not automatically split into smaller ones, but must be parallelized manually.

Locate queries are optimized for retrieving multiple occurrences simultaneously [21]. This greatly reduces the required number of accesses to Ψ and suffix array samples on highly repetitive collections.

We use the suffix array construction algorithm by Larsson and Sadakane [18] in the *build* phase because of its robustness with highly repetitive collections. The algorithm supports large alphabets, making it possible to use a different character value for each $ in the collection. By limiting the size of the input files to less than 2 gigabytes, we can build the CSA for a file of size n_i in about $8n_i$ bytes. We build the indexes for multiple files in parallel, making this phase the most memory intensive one in the algorithm.

When the partial indexes have been built, we take one of them as the initial index, and begin merging the other indexes with it one at a time. We distribute the sequences in the input file dynamically between the threads, and report the ranks of the suffixes as either 32-bit or 64-bit integers. When all threads have finished searching, we sort the resulting array, and increment each value by its position in the array to get the bit vector I used in merging.

We merge the bit vectors instead of the BWTs in our implementation. Suffix array samples, bit vector E, and each of the bit vectors Ψ_c are merged as separate subtasks that are dynamically allocated to available threads. Large subtasks are not divided into smaller ones, which can be a problem with small alphabets, or when merging a large number of suffix array samples. In-place merging is not implemented, doubling the memory usage of the bit vectors being merged.

6 Experiments

We tested the performance of our new algorithms experimentally. The experiments were performed on a 16-core SMP system running Ubuntu Linux. The system had 128 gigabytes of memory and four quad-core Intel Xeon X7350

processors running at 2.93 GHz. All programs were compiled with GCC version 4.2.4. OpenMP was used for parallelization. MCSTL[2] was used to parallelize `std::sort`, as the GCC version in use did not support libstdc++ parallel mode.

Three data sets were used to test our construction algorithms: genome, enwiki, and fiwiki. Genome is the human reference genome (NCBI build 34), with 25 sequences as individual files for a total of 2.88 gigabytes. Enwiki and fiwiki are larger text collections downloaded from Wikipedia.[3] Enwiki contains a dump of the current versions of all English language Wikipedia articles (as of 2009-03-13), while fiwiki is a highly repetitive collection containing all Finnish language Wikipedia articles with their full version histories (as of 2009-01-22).

The Wikipedia data sets were in XML format, and had to be preprocessed before indexing. In the enwiki collection, we considered the lines between tags `<page>` and `</page>` as one sequence. In fiwiki, each sequence was contained between tags `<revision>` and `</revision>`. The extracted sequences were written into 500-megabyte input files. In this final form, enwiki contains 16080833 sequences in 85 files for a total of 41.48 gigabytes, while fiwiki contains 5849111 sequences in 87 files for a total of 42.03 gigabytes.

We tested our construction algorithm on the three data sets. The sequential implementation was used on the smaller genome data set, while the larger enwiki and fiwiki collections were indexed using the parallel implementation. Index parameters were mostly set to default ones. We used 32-byte block size on the run-length encoded Ψ_c vectors, and 16-byte block size on the gap encoded vectors. Suffix array sample rate was set to 64 on genome and enwiki data sets, and to 512 on the highly repetitive fiwiki data set. With these parameters, the final index sizes for genome, enwiki, and fiwiki were 2.18 GB, 17.37 GB, and 2.13 GB, respectively. Table 1 summarizes the construction.

Table 1. Results for index construction. The construction times are in hours, and the peak memory usage is in gigabytes. Throughput is measured in megabytes / second to make comparisons with earlier results easier.

			Construction Times					
Collection	**Threads**	**Memory**	**Build**	**Search**	**Sort**	**Merge**	**Total**	**MB/s**
genome	1	2.9	0.75	0.86	0.08	0.74	2.43	0.34
enwiki	8	36–37	3.25	1.88	0.37	3.42	9.00	1.31
	16	64	2.97	1.17	0.37	3.35	7.92	1.49
fiwiki	8	32	5.33	1.75	0.36	2.16	9.60	1.24
	16	64	5.01	1.22	0.38	1.99	8.62	1.39

We were able to index the human genome in about 145 minutes using less than 3 gigabytes of memory. Even considering the improvements in processor speeds and cache sizes, this is clearly better than the 24 hours and 3.6 gigabytes

[2] http://algo2.iti.uni-karlsruhe.de/singler/mcstl/

[3] http://download.wikipedia.org/

Table 2. Query times on the three data sets with a different number of (T)hreads. *(C)ount* and *(D)isplay* times are in microseconds / character, while *(L)ocate* times are in microseconds / occurrence.

	genome			**enwiki**			**fiwiki**		
T	**C**	**L**	**D**	**C**	**L**	**D**	**C**	**L**	**D**
1	1.532	34.155	0.611	2.272	10.603	0.992	1.704	37.081	0.829
8	0.238	4.816	0.082	0.337	1.475	0.137	0.262	5.163	0.116
16	0.211	2.914	0.044	0.221	0.899	0.080	0.177	4.091	0.070

on a 1.7 GHz Pentium 4 system reported by Hon et al. [14]. By using in-place merging, we should be able to reduce our memory consumption by the final size of the largest bit vector (almost 500 megabytes). Further memory would be saved by replacing the run-length encoded bit vectors with gap encoded ones.

On the the enwiki and fiwiki collections, there was no significant speedup from 8 to 16 threads. Only the *search* phase that involves relatively complex operations on small pieces of data shows major improvement. This behavior is probably caused by cache and memory bus issues in the other phases that process large amounts of data sequentially. Another thing to note is that while merging the small indexes of the highly repetitive fiwiki collection was fast, building the partial indexes for it was much slower than for the enwiki collection.

We also tested our implementation by performing a large number of *count*, *locate*, and *display* queries using 1, 8, and 16 threads. We generated a set of random patterns for *count* and *locate* queries for each of the three collections. On the genome data set, this was 1000 patterns of length 10, with about 15.15 million total occurrences. We modified one pattern with over 2 million occurrences, as it dominated the query times in *locate*. For enwiki and fiwiki, we generated 40 random patterns of length 15 per input file, for a total of 3400 and 3480 patterns, respectively. Patterns with more than 10^5 occurrences were ignored in locate, making the total number of reported occurrences 16.89 million and 14.07 million, respectively.

Display queries consist of 10000 random prefixes of at most 10000 characters each. The total size of the extracted prefixes was 95.37 megabytes for genome, 17.64 megabytes for enwiki, and 38.90 megabytes for fiwiki. Table 2 shows the average query times. The results are mostly comparable with those in [21].

The time required for one random access to the CSA is similar in all three collections. We got a significant improvement from 8 to 16 threads for the same reasons, as in the *search* phase of index construction. *Locate* performance was similar on genome and fiwiki with different sample rates, because of the optimizations for retrieving multiple occurrences. Enwiki was significantly faster, as it benefited both from the low sample rate and the optimizations.

We could not directly compare the performance of our algorithm to other similar algorithms. Of the few known implementations, the one by Hon et al. [14] is not generally available. While Kärkkäinen's space-efficient BWT construction algorithm [16] is available, we could not compile it in a 64-bit environment.

Table 3. Construction times for Burrows-Wheeler transform. The BWT is not included in the memory consumption, as both implementations write it directly to disk.

	Our Algorithm			**Kärkkäinen's**		
	5	**10**	**20**	**128**	**1024**	**4096**
Time (minutes)	35	41	46	29	46	68
Memory (GB)	1.86	1.29	1.02	2.00	1.45	1.28

Finally, the dynamic *FM-index* by Gerlach [9] is outperformed by Kärkkäinen's algorithm in BWT construction, making comparisons to it redundant.

With this in mind, we compared our sequential algorithm to Kärkkäinen's algorithm on BWT construction. The comparison was performed on a 2.66 GHz Intel Core 2 Duo E6750 desktop system with 4 GB of memory (3.2 GB visible to OS). We downloaded the 1.10 GB protein sequence collection from the Pizza & Chili Corpus [7], and split it into 5, 10, and 20 parts for our algorithm. We used parameter values $v = 128$ (default), 1024, and 4096 for Kärkkäinen's algorithm. The results can be seen in Table 3. While Kärkkäinen's algorithm was faster with default parameters, our algorithm performed better with limited memory. We also achieved a reasonable speed while using less memory than the input size, which is impossible with Kärkkäinen's algorithm.

7 Discussion

We have presented a parallel algorithm for constructing compressed suffix arrays, and demonstrated its practical effectiveness by indexing tens of gigabytes with a throughput of about 4–5 gigabytes / hour. When the collection is highly repetitive, this can be done in memory available on today's high-end desktop systems, except for the *build* phase of the algorithm. Hence if we distribute the building of partial indexes to multiple systems, it should be feasible to index collections of hundreds of gigabytes in size with the current implementation.

We actually considered indexing the German language Wikipedia with full version history – a 933 GB highly repetitive collection. The plan was to use two older SMP systems (both with 8 cores and 32 GB of RAM) to index 10-gigabyte parts, and to merge the partial indexes on the larger system. Extrapolating from the results with the Finnish language Wikipedia, this should have taken about four days. However, due to the need for exclusive access to the systems, the experiment had to be postponed.

This naturally leads to the question, whether a true distributed implementation of the algorithm is possible. The answer seems to be yes. In addition to the *build* phase, *sort* and *merge* phases are also relatively easy to distribute. Sorting is one of the fundamental operations in distributed computing, with many efficient practical solutions, as is made evident by the Sort Benchmark.[4] On

[4] `http://www.hpl.hp.com/hosted/sortbenchmark/`

the other hand, merging can easily be split into as many independent tasks as necessary, making its distribution straightforward.

Search phase is the hardest one to distribute, as solving it will probably require a distributed CSA. As long as the CSA fits into the memory of a single node, things are easy. We can just have a copy of the CSA in each node, and distribute the sequences between the nodes. When the index grows larger, we must either store it in secondary memory, or distribute it among the nodes (or both in case of very large collections).

Using secondary memory yields a major performance loss, as we need one random access to the CSA for each character inserted. While a sequential search can process more than 1 MB/s, hard disks allow at most a few hundred random accesses per second. Although modern solid-state drives are much faster, allowing tens of thousands of random accesses per second, they are still about 30–40 times slower than the CPU. With one solid-state drive, one might get a 100 megabytes / hour throughput, so with many drives reasonable speeds could be attained.

Storing the CSA in distributed memory creates different performance problems. Network latency becomes the main factor in sequential search speed, as nodes must communicate with each other to access different parts of the CSA. On the other hand, large bandwidth makes it possible to search for many sequences in parallel, alleviating the problem. If many queries directed to the same node are grouped into one packet, a 5–10 MB/s ($\log n$ bits / character) data stream should be enough for one CPU core.

With this algorithm, distributed construction of CSAs seems feasible for multi-terabyte collections. Much of the work can even be performed on a production system, as new data arrives. Significant resources are only required for the final merging of the indexes. The real question is, can the algorithm be extended for the other structures required for suffix tree functionality [1]. If the answer is positive, it could make compressed suffix arrays the data structure of choice for many applications, such as large-scale analysis of genome data.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal on Discrete Algorithms 2(1), 53–86 (2004)
2. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
3. Chan, H.-L., Hon, W.-K., Lam, T.-W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2), 21 (2007)
4. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica 32(1), 1–35 (2002)
5. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. Journal of Experimental Algorithms 12, article no. 3.4 (2008)
6. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21(2), 194–203 (1975)

7. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. Journal of Experimental Algorithms 13, article no. 1.12 (2009)
8. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52(4), 552–581 (2005)
9. Gerlach, W.: Dynamic FM-index for a collection of texts with application to space-efficient construction of the compressed suffix array. Master's thesis, Bielefeld University (2007)
10. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: PAT trees and PAT arrays. In: Information retrieval: data structures and algorithms, pp. 66–82. Prentice-Hall, Englewood Cliffs (1992)
11. González, R., Navarro, G.: Improved dynamic rank-select entropy-bound structures. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 374–386. Springer, Heidelberg (2008)
12. 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)
13. Hon, W.-K., Lam, T.-W., Sadakane, K., Sung, W.-K., Yiu, S.-M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica 48(1), 23–36 (2007)
14. Hon, W.-K., Lam, T.-W., Sung, W.-K., Tse, W.-L., Wong, C.-K., Yiu, S.-M.: Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In: ALENEX 2004, pp. 31–38. SIAM, Philadelphia (2004)
15. Hon, W.-K., Sadakane, K., Sung, W.-K.: Breaking a time-and-space barrier in constructing full-text indices. SIAM Journal on Computing 38(6), 2162–2178 (2009)
16. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science 387(3), 249–257 (2007)
17. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Computing 33(9), 605–612 (2007)
18. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science 387(3), 258–272 (2007)
19. Lee, S., Park, K.: Dynamic rank-select structures with applications to run-length encoded texts. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 95–106. Springer, Heidelberg (2007)
20. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3), 32 (2008)
21. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
22. Na, J.C., Park, K.: Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space. Theoretical Computer Science 385(1-3), 127–136 (2007)
23. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2 (2007)
24. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 4 (2007)
25. Salson, M., Lecroq, T., Léonard, M., Mouchard, L.: Dynamic extended suffix arrays. Accepted to Journal of Discrete Algorithms
26. Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-length compressed indexes are superior for highly repetitive sequence collections. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 164–175. Springer, Heidelberg (2008)

On Entropy-Compressed Text Indexing in External Memory*

Wing-Kai Hon[1], Rahul Shah[2], Sharma V. Thankachan[2], and Jeffrey Scott Vitter[3]

[1] Department of Computer Science, National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw

[2] Department of Computer Science, Louisiana State University, LA, US
{rahul,svt}@csc.lsu.edu

[3] Department of Computer Science, Texas A & M University, TX, USA
jsv@tamu.edu

Abstract. A new trend in the field of pattern matching is to design indexing data structures which take space very close to that required by the indexed text (in entropy-compressed form) and also simultaneously achieve good query performance. Two popular indexes, namely the FM-index [Ferragina and Manzini, 2005] and the CSA [Grossi and Vitter 2005], achieve this goal by exploiting the Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994]. However, due to the intricate permutation structure of BWT, no locality of reference can be guaranteed when we perform pattern matching with these indexes. Chien *et al.* [2008] gave an alternative text index which is based on sparsifying the traditional suffix tree and maintaining an auxiliary 2-D range query structure. Given a text T of length n drawn from a σ-sized alphabet set, they achieved $O(n \log \sigma)$-bit index for T and showed that this index can preserve locality in pattern matching and hence is amenable to be used in external-memory settings. We improve upon this index and show how to apply entropy compression to reduce index space. Our index takes $O(n(H_k + 1)) + o(n \log \sigma)$ bits of space where H_k is the kth-order empirical entropy of the text. This is achieved by creating variable length blocks of text using arithmetic coding.

1 Introduction

Given a text T and a pattern P, finding all occurrences of P in T is the most fundamental problem in the field of pattern matching. In the data-structural sense, an index is built over T, and later some pattern P comes as a query; our target is to solve the above problem more quickly with the help of the index. Suffix trees [20,16] and suffix arrays [15] are the most popular indexes which can answer the query in $O(p + occ)$ time and $O(p + \log n + occ)$ time respectively, where $n = |T|$, $p = |P|$, and occ is the number of places where P occurs in T.

* This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082-MY3 (W. Hon) and US NSF Grant CCF–0621457 (R. Shah and J. S. Vitter).

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 75–89, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Historically, these two data structures are considered to consume "linear" space. However, the notion of space measure here was in terms of memory words. When measured in terms of bits, these indexes take $O(n \log n)$ bits which is asymptotically higher than the $n\lceil \log \sigma \rceil$ bits required to store the text in plain form; here, σ denotes the size of the common alphabet set Σ from which characters of T and P are drawn. Practically when we are indexing DNA texts (with $\sigma = 4$), these indexes are reported to take 15 to 50 times more space than the original data. Furthermore, the text T can often be compressed into nH_k bits by entropy-compression methods like gzip or bzip, where $H_k \leq \log \sigma$ denotes the kth-order empirical entropy of the text. Thus, the actual gap between the indexing space and the storage space is even larger.

A longstanding open question was to develop a text index which takes "truly" linear space. Grossi and Vitter [10] presented the first text index taking $O(n \log \sigma)$ bits. Simultaneously, Ferragina and Manzini [6] presented an index based on Burrows-Wheeler transform (BWT) [3] which took $O(nH_k)$ bits. Both indexing schemes were further refined [18,9,7] to take $nH_k + o(n \log \sigma)$ bits, and various space-time trade-offs are also obtained (see [17] for an excellent survey). One of the main approach in designing all these indexes is to permute the text according to the BWT. However, a short-coming of this approach is that BWT permutation completely shatters the locality of text characters. Each next character of the pattern being matched can occur at a random location in the BWT. Hence, no efficient external memory results were possible with such an approach. Chien *et al.* [4] took a different approach of *sparsifying* the suffix tree to achieve space reduction. The main idea was to combine a few contiguous characters from the text to create a block, where each block in turn is treated like a new alphabet symbol (or a meta-character). The index structures then includes the suffix tree of this blocked text as a component, which is effectively a miniature of the suffix tree of the original text but with fewer suffixes. This leads to an alternative $O(n \log \sigma)$-bit index when we set each block to contain roughly $d = 0.5 \log_\sigma n$ characters.

In this paper, we show the first entropy-compressed index in external memory which can effectively exploit locality in pattern matching. Our technique is to improve the blocking technique of Chien *et al.* [4]. We first introduce a variable-length blocking technique which is combined with arithmetic coding scheme. Using this we improve the space from $O(n \log \sigma)$ bits to $O(nH_k) + o(n \log \sigma)$ bits when $k = o(\log_\sigma n)$ and $\sigma = O(n^{1-\epsilon})$ for any fixed $\epsilon > 0$. We first present an index that works efficiently in the RAM model. Then, we show how to convert it to work in the external-memory model, and show that by maintaining an $O(n^\epsilon)$-bit table in RAM, pattern matching queries can be answered in $O((p \log n)/B + \log^3 n/(\log \sigma \log B) + occ \log_B n)$ I/Os; here, B denotes I/O block size in terms of memory words. This result is further improved to $O(p/(B \log_\sigma n) + \log^4 n/ \log \log n + occ \log_B n)$ I/Os by using $O(n)$-bit extra space.

On a related note, there were several attempts at designing compressed indexes in secondary memory based on LZ-indexes. In [2], Arroyuelo and Navarro

proposed an index whose space is $O(nH_k) + o(n \log \sigma)$, but the I/O bounds for pattern searching were not given. Their work is practical in nature and claims to answer pattern matching queries in about 20–60 disk accesses. In [8], González and Navarro provided an index which achieves $O(p + occ/B)$ I/Os for answering pattern matching query. However, their space usage is $O((n \log n) \times H_k \log(1/H_k))$ bits, which is an $O(\log n)$ factor more in terms of the optimal space complexity. Our techniques of blocking text and encoding blocks (meta-characters) using arithmetic coding are similar to the ones used in the above LZ-index line of work [2,8]. The key difference is in the way how the size of the blocks is controlled to achieve the desired theoretical bounds.

2 Preliminaries

This section introduces a few existing data structures for text indexing and orthogonal range searching which form the building blocks of our compressed text indexes. We will briefly explain their roles in our indexes, while a more detailed description is deferred in later sections. We also give a brief summary of the external-memory model of [1].

Throughout the paper, we use T to denote the text to be indexed, and $n = |T|$ to denote its length. We use P to denote the pattern which comes as an online pattern matching query, and $p = |P|$ to denote its length. Further, we assume the characters of T and P are both drawn from the same alphabet set Σ whose size is σ.

2.1 Suffix Trees, Suffix Arrays, and Burrows-Wheeler Transform

Suffix trees [20,16] and suffix arrays [15] are two well-known and popular text indexes that support online pattern matching queries in optimal (or nearly optimal) time. For text $T[1...n]$ to be indexed, each 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. Each edge e in the suffix tree is labeled by a series of characters, such that if we examine each root-to-leaf path, the concatenation of the edge labels along the path is exactly equal to the corresponding suffix represented by the leaf.

Suffix array $SA[1...n]$ is an array of length n, where $SA[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T. An important property of SA is that the starting positions of all suffixes with the same prefix are always stored in a contiguous region in SA. Based on this property, we define the *suffix range* of a pattern P in SA to be the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, $SA[j]$ is the starting point of a suffix of T with P as a prefix. Note that SA can be obtained by traversing the leaves of suffix tree in a left-to-right order, and outputting the starting position of each leaf (i.e., a suffix of T) along this traversal. In particular, we have the following technical lemma about suffix trees, suffix arrays, and suffix ranges.

Lemma 1. *Given a text T of length n, we can index T using suffix tree and suffix array in $\Theta(n \log n)$ bits such that the suffix range of any input pattern P can be obtained in $O(p)$ time.*

Suffix trees or suffix arrays maintain relevant information of all n suffixes of T such that on given any input pattern P, we can easily search for the occurrences of P *simultaneously* in each position of T. However, a major drawback is the blowup in space requirement, from the original $\Theta(n \log \sigma)$ bits of storing the text in plain form to the $\Theta(n \log n)$ bits of maintaining the indexes. In our compressed text indexes, we apply a natural and very simple idea to achieve space reduction, as suggested in [4] by maintaining only a fraction of these suffixes. The consequence is that we can no longer search all positions of T in a single pass. Instead, we need multiple passes, thus causing some inefficiency in the query time. On the other hand, we gain much space reduction by storing fewer suffixes.

The Burrows-Wheeler transform of a text T is an array BWT of characters such that $BWT[i]$ is the character preceding the ith lexicographically smallest suffix of T. That is, $BWT[i] = T[SA[i] - 1]$.

2.2 External-Memory Model

The external-memory model [1] or I/O model was introduced by Aggarwal and Vitter in 1988. In this model, the CPU is connected directly to an internal memory of size M, which is then connected to a much larger and slower disk. The disk is divided into blocks of B words (i.e., $B \log n$ bits). The CPU can only operate on data inside the internal memory. So, we need to transfer data between internal memory and disk through I/O operations, where each I/O may transfer a block from the disk to the memory (or vice versa). Since internal memory (RAM) is much faster, operations on data inside this memory are considered free. Performance of an algorithm in the external-memory model is measured by the number of I/O operations used.

2.3 String B-Tree

String B-tree (SBT) [5] is an index for a text T that supports efficient online pattern matching queries in the external-memory setting. Basically, it is a B-tree over the suffix array SA of T but with extra information stored in each B-tree node to facilitate the matching. The performance of SBT is summarized as follows.

Lemma 2. *Given a text T of length n characters , we can index T using a string B-tree in $\Theta(n/B)$ blocks or $\Theta(n \log n)$ bits such that the suffix range of any input pattern P of length p can be obtained in $O(p/(B \log_\sigma n) + \log_B n)$ I/Os.* □

In our compressed text index for the external-memory setting, we again achieve space reduction by maintaining fewer suffixes. Thus, our index includes a sparsified version of the SBT as the main component.

2.4 Orthogonal Range Searching in 2D Grid Using Wavelet Tree

In our compressed text index, in addition to the suffix trees or SBT, another key component is a data structure to represent some integer array $A[1...m]$, with each integer drawn from $[1, n]$, which can efficiently support online 4-sided queries of the following form:

Input: A position range $[\ell, r]$ and a value bound $[y, y']$
Output: All those z's in $[\ell, r]$ such that $y \leq A[z] \leq y'$

The above problem can easily be modeled as a geometric problem as follows. First, for each $i \in [1, m]$, generate a point $(i, A[i])$ in the 2-dimensional grid $[1, m] \times [1, n]$. This forms the representation of the array A. Then, for any input query with position range $[\ell, r]$ and value bound $[y, y']$, the desired output corresponds to all points in the grid that are lying inside the rectangle $[\ell, r] \times [y, y']$.

Such a query is called an *orthogonal range query* in the literature, and many indexing schemes are devised that have different tradeoffs between index space and query time. In our compressed text indexes, we will require an index for A which takes $O(m \log n)$ bits of space, so we select the wavelet tree [14,12,21] as our choice, whose results are summarized in the following lemma.

Lemma 3. *Given an integer array A of length m with values drawn from $[1, n]$, we can index A in $O(m \log n)$ bits such that the 4-sided query of any position range $[\ell, r]$ and any value bound $[y, y']$ can be answered in $O((occ + 1) \log n / \log \log n)$ time in the RAM model and $O((occ + 1) \log_B n)$ I/Os in the external-memory model.* □

3 The Framework of Our Indexing Scheme

This section first describes the general framework of our index design, which consists of a combination of the building block data structures mentioned in Section 2. Afterwards, we will look at the general approach to perform pattern matching based on our index. The following two sections details with the design and the analysis of the index performance.

3.1 The Framework of the Index Design

To obtain our compressed index, we perform the following three key steps:

Step 1: Given a text T, we first transform T into an equivalent text T' such that T' consists of at most $O((nH_k + o(n \log \sigma))/ \log n)$ meta-characters, where each meta-character represents at most d consecutive characters in the original text for some threshold d. In addition, we also require that each meta-character can be described in $O(\log n)$ bits, so that T' can be described in $O(nH_k) + o(n \log \sigma)$ bits.

Step 2: We maintain the suffix tree or String B-Tree for T', where we consider each meta-character of T' as a single character from a new alphabet.

Step 3: We perform the Burrows-Wheeler transform on T' to obtain an array A. Then we maintain the wavelet tree for A.

3.2 The Framework of the Pattern Matching Algorithm

The suffix tree or SBT in our index will maintain only the suffixes of T', which correspond to only a fraction of the original suffixes. Then, when a pattern P occurs in T, it will in general match the corresponding meta-characters of T' in the following way:

> The first part of P, say $P[1..i]$, matches the suffix of a meta-character $T'[j]$ and the remainder of P, say $P[i+1..p]$, matches the prefix of $T'[j+1..|T'|]$. We shall call such an occurrence of P an *offset-i occurrence* of P in T.

Our pattern matching algorithm is to find the offset-i occurrences of P separately for each relevant i. In our design, each meta-character of T' represents at most d original characters of T. It is therefore sufficient to consider only those i in $[0, d-1]$. This leads to the following pattern matching algorithm, which consists of two major steps:

Step 1: Compute the suffix range of $P[i+1..n]$ in the suffix array SA' of T' for each $i \in [0, d-1]$ using the suffix tree (ST') or String B-Tree (SBT') of T'.

Step 2: For each $i \in [0, d-1]$, use the suffix range of $P[i+1..n]$ to issue a 4-sided query in the wavelet tree of A to find all offset-i occurrences of P. (Details of how to issue the corresponding 4-sided query are given in the next section.)

4 Index for Internal Memory Model

In this section, we show a simple index based on variable length meta-character blocking and sparse suffix tree in the internal memory model. Later, in section 5, we shall show how to extend our results to the external memory model.

4.1 Index Design

In the index given by Chien *et al.* [4], the given text T is converted to an equivalent text T' by blocking every $d = 0.5 \log_\sigma n$ characters. Each block, called a meta-character, contains fixed number of characters. The transformed text T' consists of $O(n/\log_\sigma n)$ meta-characters. Hence, the suffix tree of T' takes $O(n \log \sigma)$ bits space.[1] The new index we propose in this paper improves the space complexity to $O(nH_k) + o(n \log \sigma)$ bits. Here, instead of having each meta-character contain a fixed number of characters, we allow a variable number of characters. Each meta-character is encoded in such a way that, its first k characters are written explicitly (using fixed length encoding) and the rest using kth-order arithmetic coding. The number of characters within a meta-character is restricted by the following two conditions.

[1] Assuming each integer and each pointer is at most $\log n$ bits long.

– The number of characters should not exceed a threshold $d = \log^2 n / \log \sigma$.
– After encoding, the total length should not exceed $0.5 \log n$ bits.[2]

In our new index, the transformation of T into T' can be performed as follows. Start encoding T from $T[1]$ and get its longest prefix $T[1...j]$, which satisfies the conditions of a meta-character. Hence, $T[1...j]$ in its encoded form is our first meta-character. After that the remainder of T is encoded recursively. (Note that the strings corresponding to distinct meta-characters are not required to be prefix-free.) The starting position of each meta-character is stored in an array M such that $M[i]$ corresponds to the starting position of ith meta-character in T. In other words, the substring $T[M(i)...(M[i+1]-1)]$ corresponds to the ith meta-character. For instance, $M[1] = 1$ and $M[2] = j + 1$. By concatenating all these meta-characters (in the order in which the corresponding block appears in T), we obtain the desired string T'.

Since each meta-character corresponds to a maximal substring of T without violating the two conditions, a meta-character corresponds either to (i) exactly d characters of T, or (ii) its encoding is just below $0.5 \log n$ in which case the encoding is of $\Theta(\log n)$ bits and corresponds to $\Theta(\log_\sigma n)$ characters of T.[3] Note that in both cases each meta-character corresponds to $\Omega(\log_\sigma n)$ characters.

Direct entropy compression of T would have resulted in $nH_k + o(n \log \sigma)$-bit space for T'. But in our scheme, the first k characters are written explicitly in each block. This results in an overhead of $O((n/\log_\sigma n) \times k \log \sigma) = o(n \log \sigma)$ bits to encode T', assuming $k = o(\log_\sigma n)$.[4] Thus, the number of meta-characters from (i) cannot exceed $n/d = o(n \log \sigma / \log n)$, while the number of meta-characters from (ii) is bounded by $O((nH_k + o(n \log \sigma))/\log n)$. In summary, the length of $T' = nH_k + o(n \log \sigma)$ bits, and there is a total of $O((nH_k + o(n \log \sigma))/\log n)$ meta-characters in T'.

By considering each meta-character as a single character from the new alphabet set, we construct the suffix tree ST' of T'. As the length of T' is given by $O((nH_k + o(n \log \sigma))/\log n)$, so is the number of nodes in ST'. Thus, ST' takes $O((nH_k + o(n \log \sigma))/\log n \times \log n) = O(nH_k) + o(n \log \sigma)$ bits of space.

Lemma 4. *The total number of distinct meta-characters is* $O(\sqrt{n})$.

Proof. Each meta-character has an encoding between 1 and $0.5 \log n$ bits. Thus, the number of distinct meta-character is at most $\sum_{r=1}^{0.5 \log n} 2^r = O(\sqrt{n})$. □

[2] Without loss of generality, we assume here that $\sigma < n^{1/4}$. The parameters can be appropriately adjusted for the more general case when $\sigma = O(n^{1-\epsilon})$ for any fixed $\epsilon > 0$.

[3] Here, we make a slight modification that one extra bit is spent for each meta-character, such that if our kth-order encoding of the next $o(\log_\sigma n)$ characters already exceeds $0.5 \log n$, we shall instead encode the next $0.5 \log_\sigma n$ characters (i.e., more characters) in its plain form. The extra bit is used to indicate whether we use the plain encoding or the kth-order encoding.

[4] As mentioned, there is also an extra bit overhead per meta-character; however, we will soon see that the number of meta-characters $= O((nH_k + o(n \log \sigma))/\log n)$ so that this overhead is negligible.

We also construct an auxiliary trie-structure Π which can be used to rank each of the meta-characters among all the meta-characters that are constructed from the text. Let B be a block in T which corresponds to a meta-character C in T', and let $\overleftarrow{B}$ denote the string obtained by reversing the characters of B. We maintain a string L which is the concatenation of all distinct $\overleftarrow{B}$'s in the uncompressed form and we construct a compact trie Π storing all distinct $\overleftarrow{B}$'s. The edges of Π are represented using two pointers, which are the starting and ending points of the corresponding substring in L. String L takes $O(\sqrt{n} \times (\log^2 n/\log\sigma) \times \log\sigma) = o(n)$ bits and Π takes $O(\sqrt{n} \times \log n) = o(n)$ bits of space.

Let $\Pi(i)$ represent the ith leftmost leaf of Π. Now we shall show how to obtain an array A from which we construct the wavelet tree. For this, we first compute BWT of T'. Let $BWT[i] = C$, where C is a meta-character and B is its corresponding character block. Now, search for $\overleftarrow{B}$ in Π and reach a leaf node $\Pi(j)$; then we set $A[i] = j$. That is, $A[i]$ is the leaf-rank of $\overleftarrow{B}$ in Π. Finally, we maintain a wavelet tree of A based on Lemmas 3 and 4, whose space takes $O((nH_k + o(n\log\sigma))/\log n) \times \log(O(\sqrt{n})) = O(nH_k) + o(n\log\sigma)$ bits. The total space requirement for our index is $O(nH_k) + o(n\log\sigma)$ bits.

4.2 Pattern Matching Algorithm

The suffix tree ST' maintains only the suffixes of T'. Therefore navigating through ST' can only report those occurrences of the query pattern P which start at a meta-character boundary. But in general, P can start anywhere inside T, where $P[1...i]$ matches to the suffix of a meta-character $T'[j]$ and the remaining of P, $P[i+1...p]$ matches the prefix of $T'[j+1...|T'|]$. We call such an occurrence of P an offset-i occurrence of P in T. We need to check for all possible offset occurrences. Since the number of characters inside a meta-character is at most d, it is sufficient to check for those offsets i where $i = 0, 1, 2, \ldots, d-1$.

To find offset-i occurrences, we let P_{pre} represent the prefix $P[1...i]$ and P_{suf} represent the suffix $P[i+1...p]$ of the pattern P. We first convert P_{suf} into P'_{suf} by blocking this into meta-characters. Following our convention, we use $\overleftarrow{P_{pre}}$ to denote the reverse of P_{pre}. Next, we search for $\overleftarrow{P_{pre}}$ in the compact trie Π to reach a position u^* (if exists); note that u^* may be an internal node, or within an edge, rather than a leaf. In any case, we use $\Pi(i_{left})$ and $\Pi(i_{right})$ to denote, the leftmost and rightmost leaves in the subtree of u^*.

We are now ready to show how to search for the desired offset-i occurrences of P:

1. Search for P'_{suf} in ST' and obtain its suffix range $SA'[\ell...r]$. Here P'_{suf} is of length at most $p\log\sigma$, hence by assuming standard word length of $O(\log n)$ bits, this matching step can be performed in $O(p/\log_\sigma n)$ time. But for matching an ending portion of a pattern, which may be smaller than the length of a meta-character, we need to perform a "predecessor search" in order to get the range. Therefore, in general the suffix range can be obtained in $O(p/\log_\sigma n + \log n)$ time.[5]

[5] More precisely, we maintain the SBT data structure for short patterns as suggested by Hon *et al.* [11] to accomplish the task. We defer the details in the full paper.

2. We need to find out those text positions in $SA'[\ell...r]$, such that P_{pre} occurs before those positions. This is equivalent to finding all z's in $[\ell, r]$, such that $i_{left} \leq A[z] \leq i_{right}$.
3. Now the search for offset-i occurrences is reduced to an orthogonal range searching problem in 2D grid. We use the wavelet tree structure of A to solve this query. According to Lemma 3, this will take $O((occ(i)+1) \log n / \log \log n)$ time, where $occ(i)$ represents the number of offset-i occurrences.

Lemma 5. *Based on ST' and the wavelet tree of A, all the offset-i occurrences of a pattern P in T, which cross at least one meta-character boundary, can be reported in $O(p/\log_\sigma n + \log n + occ(i) \log n / \log \log n)$ time, where $occ(i)$ is the number of offset-i occurrences of P in T.* □

The above steps need to be performed for all possible offsets i, where $i = 0, 1, \ldots, d-1$. For each offset i we need to convert P_{suf} into P'_{suf}. Assuming the conversion is done independently for each offset, it will in total take $O(p \log n + d \log n)$ time. This gives the following lemma.

Lemma 6. *A given text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits such that all the occurrences of a pattern P in T, which crosses at least one meta-character boundary in T, can be reported in $O(p \log n + \log^3 n / \log \sigma + occ \log n / \log \log n)$ time.* □

4.3 Index for Short Patterns

The methods described before will work only for those occurrences of a pattern that cross a meta-character boundary. To find those short patterns which start and end inside the same meta-character, we rely on an auxiliary data structure which is a generalized suffix tree Δ of all the *distinct* meta-characters that appear. Considering Lemma 4, the space for Δ can easily be bounded by $o(n)$.

The search begins by matching the pattern P in Δ to obtain the list L of all the distinct meta-characters in which P occurs (along with the relative positions of pattern occurrences inside a given meta-character. Now, on top of this, for each distinct meta-character C appearing in the text, we maintain the list H_C of all the positions in T' where the meta-character C occurs. These lists overall take $\log n$ bits per meta-character and hence the total space for the H structure is bounded by $O(nH_k)+o(n \log \sigma)$ bits. Once the list L of meta-characters (along with the internal positions) is obtained from Δ we use H as the de-referencing structure to obtain the final set of positions.

Lemma 7. *A given text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits such that all the occurrences of pattern P in T, which starts and ends inside the same meta-character in T, can be reported in $O(p + occ)$ time.* □

The following theorem concludes our result.

Theorem 1. *A text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits space, such that all the occurrences of a pattern P in T can be reported in $O(p \log n + \log^3 n / \log \sigma + occ \log n / \log \log n)$ time.* □

5 Extension to External Memory Model

In this section, we extend our results in the RAM model to the external memory model.[6] For this, we replace each data structure in internal memory model with its external memory counterpart. The sparse suffix tree ST' will be replaced by a sparse string B-tree SBT' of T'. The wavelet tree of array A will be replaced with its external memory version [12,14]. By performing a similar analysis, and setting the threshold d to be $\log^2 n / \log \sigma$, the searching for a pattern P in T will take $\sum_{i=0}^{d-1} O(p/(B \log_\sigma n) + \log_B n + occ(i) \log_B n) = O((p \log n)/B + (\log_B n)(\log^2 n / \log \sigma) + occ \log_B n)$ I/Os, where $occ(i)$ represents the number of offset-i occurrences that cross at least one meta-character boundary and occ represents the total number of such occurrences. The generalized suffix tree for short patterns will be replaced by string B-tree, which can perform pattern matching in $O(p/B + \log_B n + occ)$ I/Os. Immediately, we have the following theorem.

Theorem 2. *A text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits in the external memory, such that all occurrences of pattern P can be reported in $O((p \log n)/B + \log^3 n / (\log \sigma \log B) + occ \log_B n)$ I/Os.*

Indeed, we can reduce the $O((p \log n)/B)$ term to $O(p/(B \log_\sigma n))$, if we allow slightly more index space. This is done by combining our index with Sadakane's Compressed Suffix Tree (CST) [19]. Our goal is to avoid repeated pattern matching for various offsets, which is done by using the "suffix link" functionality provided by CST. The main idea is that if some part of the pattern is matched during the offset-k search then we avoid re-matching it for offset-$(k+1)$ search and onwards; instead we rely on the suffix link to provide information for the subsequent search.

In the remainder of this section, we sketch how the pattern matching algorithm can be sped up by storing the CST. Firstly, for any internal node u inside the suffix tree, let $path(u)$ denote the string obtained by concatenation of edge labels from root to u. The *suffix link* of u is defined to be the (unique) internal node v such that the removal of the first character of $path(u)$ is exactly the same as $path(v)$. However, suffix link with respect to the original suffix tree may not exist in the sparse suffix tree or the sparse string B-tree (simply because some suffixes are missing).

In our algorithm, the full (non-sparse) suffix tree on T must be used, so that we can follow the original suffix links. To stay within our space bounds of $O(nH_k)$ we cannot afford to use the regular suffix tree. This explains why we choose the CST of [19], which provides *all* suffix tree functionalities in compressed space.

---

[6] Recall that the block size parameter B is measured in terms of memory words while the pattern length p is measured in terms of characters. Here, we further assume that the decoding table for arithmetic coding fits in the internal memory. By choosing appropriate parameters and with the condition that $k = o(\log_\sigma n)$, we can ensure that the decoding table size is $O(n^\epsilon)$ bits.

5.1 Compressed Suffix Tree

Let us assume we have stored Compressed Suffix Tree CST of the text T. In addition, all the nodes in CST which are also in the sparse suffix tree ST' are marked. For this marking, a bit-vector is maintained in addition to CST. The nodes in CST are considered in pre-order fashion and whenever a marked node is visited we write "1" or else we write "0". Thus, this bit-vector $\mathcal{B}$ stores marking information on the top of CST.

We shall need the following functionalities provided by the recent CST of [19] together with our bit-vector $\mathcal{B}$:

Suffix link: Given a node u (by its pre-order rank) in CST, return the suffix link node v (by its pre-order rank). This function can be done in $O(\log \sigma)$ I/Os.

Highest marked descendant: Given a node u in CST, its highest marked descendant is defined to be the node v such that v is in the subtree of u, v is marked, and no nodes between u and v is marked. Such a node v (if exists) is unique. This is due to the fact that the least common ancestor of two marked nodes (i.e., the least common ancestor of two sparse suffix tree nodes) is also marked. Note that this functionality is not directly provided by CST of [19] but can easily be implemented in $O(1)$ I/Os by storing a rank/select data structure over the bit-vector $\mathcal{B}$ along with the parentheses encoding of CST.

Lowest marked ancestor: Given a node u in CST, report its lowest marked ancestor (if exists). This can be done in $O(1)$ I/Os based on $\mathcal{B}$ and its the rank/select data structure.

Leftmost leaf: Given a node u in CST, locate its leftmost (rightmost) leaf node in its subtree. This can be done in $O(1)$ I/Os.

String-depth: Given a node u, report the length of $path(u)$. This can be done in $O(\log^2 n / \log\log n)$ I/Os.

Weighted level ancestor: Given a leaf ℓ and string-depth w, report the (unique) node u such that u is the first node on the path from root to ℓ with string-depth $\geq w$. This node u must be a lowest common ancestor between ℓ and some other leaf ℓ', so that we can find u if ℓ' is determined. Such ℓ' can be found by binary searching all leaves to the right of ℓ, and examine the string-depth of lowest common ancestor of ℓ and the leaf. The process can be done in $O(\log^3 n / \log\log n)$ I/Os.

5.2 Sparse String B-Tree

Our explanation below shall refer to both the sparse suffix tree and the sparse string B-tree. However, the sparse suffix tree is never stored and is just for the sake of notation and the identification of nodes. Firstly, the following two functionalities of the sparse string-B tree SBT' will be used. The I/O complexity for both functions follows directly from the searching strategy of SBT in the original paper [5].

1. Given a pattern P, let $lcp(P, ST')$ be the length of the longest common prefix of P with any suffix stored in SBT': we can use $O(lcp(P, ST')/B + \log_B n)$ I/Os to find the node u (by its pre-order ranking in the suffix tree ST') such that u is the node with smallest string-depth in ST' and $lcp(P, ST') = lcp(P, path(u))$.
2. If we are given a node u in ST' such that the pattern P is guaranteed to match up to some length x on $path(u)$, then the above lcp search can be done in $O((lcp(P, ST') - x)/B + \log_B n)$ I/Os.

5.3 Pattern Matching Algorithm

Now, we are ready to show how we match a pattern P in this combination of sparse string B-tree and CST. First we start with finding offset-0 occurrences, then we find offset-1 occurrences, then offset-2 occurrences and so on. Let P_i denote the pattern P with the first i characters deleted. Thus we have to match $P_0, P_1, P_2, \ldots, P_{d-1}$ in the string B-tree. Corresponding to each offset i we find the range $[\ell_i, r_i]$ in the sparse string B-tree.

We start matching the pattern $P = P_0$ in SBT'; this allows us to find the node u in ST', such that u is the closest node from root such that $lcp(path(u), P) = lcp(P, ST')$. If the pattern is matched entirely, then we call this offset a success and output its range. In this case we set $lcp = p$, and also obtain the range $[\ell_0, r_0]$. If not, we set $lcp = lcp(P, ST')$ and follow the "suffix link". Let's first define the notion of suffix link in the sparse suffix tree ST' (or SBT').

Definition 1. *Given the pair (u, lcp), let pair (v, lcp') be such that (1) $lcp' = lcp - t$, (2) $path(u)[t + 1..lcp] = path(v)[1..lcp']$ and (3) t is the smallest integer ≥ 1 for which such a node v exists in ST'. If more than one v exists in ST', we set v to be the highest node among them. Then (v, lcp') as is called t-suffix link of (u, lcp).*

Now, we show how to compute t-suffix link for pair (u, lcp) in $O(t \log^3 n/ \log \log n)$ I/Os. This is done by using the suffix link functionality provided by CST. First, we use the pre-order rank of u to find the corresponding node in CST. Then, inside CST, we can find u's ancestor y such that string-depth of y is just more than lcp. This can be done by the weighted level ancestor query in $O(\log^3 n/ \log \log n)$ I/Os. The node y represents the location where P stops in the CST if P were matched with the CST instead. To proceed for the next offset, we follow the suffix link from y and reach node w (and increment t by 1). Now, we first find the lowest marked ancestor m of w in $O(1)$ I/Os and check if its string-depth is at least $lcp - t$. If so, we come back to its corresponding node v in ST' and set $lcp' = lcp - t$. Note that (v, lcp') is the desired t-suffix link of (u, lcp), so that we can proceed with the pattern matching in SBT'.[7] Otherwise, if m does not exist or its string-depth is too small, we find in the subtree of w and try the highest marked descendant m' of w in $O(1)$ I/Os. If m' exists, we come

[7] Note that when we switch back to a node in SBT', we choose the top-most node in SBT' corresponding to the node v.

back to its corresponding node v' in ST' and set $lcp' = lcp - t$, while it follows that (v', lcp') is the desired t-suffix link of (u, lcp) so that we can again proceed with the pattern matching in SBT'. If there is no such marked descendant m', we follow further the suffix link from w (and increment t), and keep following suffix links until we reach either a node m or m' using the above procedure. In this case, we can be sure that none of the offsets between 1 and $t-1$ would produce any results. Consequently the corresponding (v, lcp') or (v', lcp') will be the desired t-suffix link and we can directly jump to offset-t match. This procedure gives us all the ranges $[\ell_i, r_i]$ for all the possible offsets (up to at most d of them).

5.4 Analysis

The space taken by both CST and string B-tree is $O(nH_k + n) + o(n \log \sigma)$ bits. For matching the pattern P, there are d phases. In each phase, we match some *distinct* part of P and then spend $O(\log^3 n / \log\log n)$ I/Os in CST plus an extra $O(\log_B n)$ I/Os (apart from matching characters of P) in SBT'. Thus, in total, we spend $O(d \log^3 n / \log\log n)$ in addition to the I/O in which the pattern is matched with the actual text inside the SBT'. On the other hand, since the characters of P are accessed once and are accessed sequentially, the total I/Os for matching characters of P can be bounded by $O(p/(B \log_\sigma n) + d \log_B n)$. For the conversion of the characters in P into the corresponding meta-characters, we assume that it is done in RAM so that it does not incur additional I/Os. Overall, this gives us $O(p/(B \log_\sigma n) + d \log^3 n / \log\log n)$ I/Os for finding out all the ranges $[\ell_0, r_0], [\ell_1, r_1], ..., [\ell_{d-1}, r_{d-1}]$.

Once these ranges are ready, we can use the external memory wavelet tree to find out the actual occurrences (which cross a meta-character boundary). The short patterns are handled as before using the generalized suffix tree approach (except we are using a SBT instead). Since the space of CST is $O(nH_k + n)$ bits which is the bottleneck, we may reduce the blocking factor to be $d = 0.5 \log n$ (thus having the effect of more meta-characters in T' but faster query) without affecting the space. The following theorem captures our new result.

Theorem 3. *A text T can be indexed in $O(nH_k + n) + o(n \log \sigma)$ bits in external memory, such that all occurrences of a pattern P in T can be reported in $O(p/(B \log_\sigma n) + \log^4 n / \log\log n + occ \log_B n)$ I/Os.*

6 Conclusion

We show the first entropy compressed text index in external memory. Our index is based on the paradigm of using sampled suffixes [13], and achieves locality while matching pattern which was lacking in other BWT based indexes. The main idea here is to partition the text into variable length block according to their compressibility and then compress each block using arithmetic coding. We show how this idea can be combined with the notion of suffix links by using CST of Sadakane[19].

We achieve optimal query I/O performance with respect to the length p of the input query pattern, taking $O(p/(B \log_\sigma n))$ I/Os. As noted by Chien *et al.* [4], the lower bounds in range searching data structures suggest that the last term $O(occ \log_B n)$ cannot be improved to $O(occ/B)$. But, it may be possible to improve the middle term of polylog(n). Another possible improvement could be in reducing space term from $O(nH_k)$ to strictly nH_k.

Acknowledgments

We would like to thank the anonymous reviewers for their careful reading and constructive comments, and for pointing out a potential flaw in the paper. We would also like to express our gratitude to Kunihiko Sadakane for clarifying the functionalities of the CST in his recent paper [19].

References

1. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1998)
2. Arroyuelo, D., Navarro, G.: A Lempel-Ziv Text Index on Secondary Storage. In: Proceedings of Symposium on Combinatorial Pattern Matching, pp. 83–94 (2007)
3. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA (1994)
4. Chien, Y.-F., Hon, W.-K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In: Proceedings of Data Compression Conference, pp. 252–261 (2008)
5. Ferragina, P., Grossi, R.: The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. Journal of the ACM 46(2), 236–280 (1999)
6. Ferragina, P., Manzini, G.: Indexing Compressed Text. Journal of the ACM 52(4), 552–581 (2005); A preliminary version appears in FOCS 2000
7. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed Representations of Sequences and Full-Text Indexes. ACM Transactions on Algorithms 3(2) (2007)
8. González, R., Navarro, G.: A Compressed Text Index on Secondary Memory. In: Proceedings of IWOCA, pp. 80–91 (2007)
9. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: Proceedings of Symposium on Discrete Algorithms, pp. 841–850 (2003)
10. 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
11. Hon, W.-K., Lam, T.-W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed Index for Dictionary Matching. In: Proceedings of Data Compression Conference, pp. 23–32 (2008)
12. Hon, W.K., Shah, R., Vitter, J.S.: Ordered Pattern Matching: Towards Full-Text Retrieval. Technical Report TR-06-008, Department of CS, Purdue University (2006)
13. Kärkkäinen, J., Ukkonen, E.: Sparse Suffix Trees. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 219–230. Springer, Heidelberg (1996)

14. Mäkinen, V., Navarro, G.: Position-Restricted Substring Searching. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 703–714. Springer, Heidelberg (2006)
15. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
16. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
17. Navarro, G., Mäkinen, V.: Compressed Full-Text Indexes. ACM Computing Surveys 39(1) (2007)
18. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003); A preliminary version appears in ISAAC 2000
19. Sadakane, K.: Compressed Suffix Trees with Full Functionality. Theory of Computing Systems, 589–607 (2007)
20. Weiner, P.: Linear Pattern Matching Algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
21. Yu, C.C., Hon, W.K., Wang, B.F.: Efficient Data Structures for Orthogonal Range Successor Problem. In: Ngo, H.Q. (ed.) COCOON 2009. LNCS, vol. 5609, pp. 97–106. Springer, Heidelberg (2009)

A Linear-Time Burrows-Wheeler Transform Using Induced Sorting

Daisuke Okanohara[1] and Kunihiko Sadakane[2]

[1] Department of Computer Science, University of Tokyo,
Hongo 7-3-1, Bunkyo-ku, Tokyo 113-0013, Japan
hillbig@is.s.u-tokyo.ac.jp
[2] National Institute of Informatics (NII),
Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo 101-8430, Japan
sada@nii.ac.jp

Abstract. To compute Burrows-Wheeler Transform (BWT), one usually builds a suffix array (SA) first, and then obtains BWT using SA, which requires much redundant working space. In previous studies to compute BWT directly [5,12], one constructs BWT incrementally, which requires $\mathrm{O}(n \log n)$ time where n is the length of the input text. We present an algorithm for computing BWT directly in linear time by modifying the suffix array construction algorithm based on induced sorting [15]. We show that the working space is $\mathrm{O}(n \log \sigma \log \log_\sigma n)$ for any σ where σ is the alphabet size, which is the smallest among the known linear time algorithms.

1 Introduction

A Burrows-Wheeler Transform (BWT) [1] is a transformation from a text to a text, which is useful for many applications including data compression, compressed full-text indexing, pattern mining to name a few.

To compute BWT, one usually builds a suffix array (SA) first, and then obtains BWT from SA. Although both steps can be done in linear time in the length of the text [8,9,10], it requires large working space. Although the space for the result of BWT is $n \lg \sigma$ bits [1] where n is the length of the text, and σ is the alphabet size, that for SA is $n \lg n$ bits. For example, in the case of human genomes, $n = 3.0 \times 10^9$ and $\sigma = 4$, the size of BWT is about 750 MB, and that of SA is 12 GB. Therefore, the working space is about 16 times larger than that for BWT.

Previous studies [5,12] showed that one can compute BWT without SA by implicitly adding suffixes from the shortest ones to longest ones. However, these algorithms are slow due to the large constant factor, and their computational cost are $\mathrm{O}(n \log n)$ time. There also exist other types of algorithms. Hon et al. [6] gave an algorithm using $\mathrm{O}(n \log \sigma)$ space and $\mathrm{O}(n \log \log \sigma)$ time. Na and Park [13] gave one using $\mathrm{O}(n \log \sigma \log_\sigma^\alpha n)$ space and $\mathrm{O}(n)$ time where $\alpha = \log_3 2$.

In this paper, we present an algorithm for computing BWT directly in linear time using $\mathrm{O}(n \log \sigma \log \log_\sigma n)$-bit space. Our algorithm is based on the suffix array construction algorithm based on induced sorting [15]. In original SA algorithm, the whole

[1] $\lg x$ denotes $\lceil \log_2 x \rceil$.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 90–101, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Table 1. Time and space complexities. H_0 is the order-0 empirical entropy of the string and $H_0 \leq \log \sigma$. $\alpha = \log_3 2$.

Time	Space (bits)	References
$O(n)$	$O(n \log n)$	[2,8,9,10,15] (compute SA)
$O(n \log n)$	$O(nH_0)$	[5]
$O(n \log \log \sigma)$	$O(n \log \sigma)$	[6]
$O(n)$	$O(n \log \sigma)$	[6] ($\log \sigma = O((\log \log n)^{1-\epsilon})$)
$O(n)$	$O(n \log \sigma \log_\sigma^\alpha n)$	[13]
$O(n)$	$O(n \log \sigma \log \log_\sigma n)$	This paper

SA are induced from the carefully sampled SA. Our algorithm simulates this algorithm by using BWT only, and induces whole BWT from the sampled BWT. Our algorithm works in linear time and requires the working space close to that for input and output. Moreover our algorithm is simple and easy to implement. Table 1 gives a comparison with other algorithms. Our algorithm uses the smallest space among the linear time algorithms.

2 Preliminaries

Let $T[1,n]$ be an input text, n its length, and Σ its alphabet set, with $\sigma = |\Sigma|$. We denote the i-th character of T by $T[i]$, and the substring from i-th character to j-th character for $i \leq j$ by $T[i,j]$. We assume that T is followed by a special character $T[n] = \$$, which is lexicographically smaller than any other characters in T, and do not appear in T elsewhere. We also assume $\sigma \leq n$ because otherwise $n \log \sigma = \Omega(n \log n)$.

2.1 Suffix Arrays and Burrows-Wheeler Transform

A suffix of T is $T_i = T[i,n]$ $(i = 1, \ldots, n)$. Then, a suffix array of T, $SA[1,n]$ is defined as an integer array $SA[1,n]$ of length n such that $T_{SA[i]} < T_{SA[i+1]}$ for all $i = 1, \ldots, n-1$ where $<$ between strings denotes the lexicographical order of them. SA requires $n \lg n$ bit of space.

A Burrows-Wheeler Transform (BWT) of a text T, $B[1,n]$ is defined as follows;

$$B[i] = \begin{cases} T[SA[i]-1] & (SA[i] > 1) \\ T[n] & (SA[i] = 1). \end{cases} \tag{1}$$

We will denote BWT not only as the transformation, but also the result of the transformation.

BWT has several characteristics; First, BWT is a reversible transformation. That is, the original text can be recovered from BWT without any additional information [1]. Second, BWT is often easy to compress. For example, by using the compression boosting technique [3], we can compress BWT in k-th order empirical entropy for any k by using simple compression algorithms, which does not consider the context information. Third, BWT can be used for constructing compressed full-text indexes. For example, we can build a compressed suffix array [4], and a FM-index from BWT in $O(n)$ time [5].

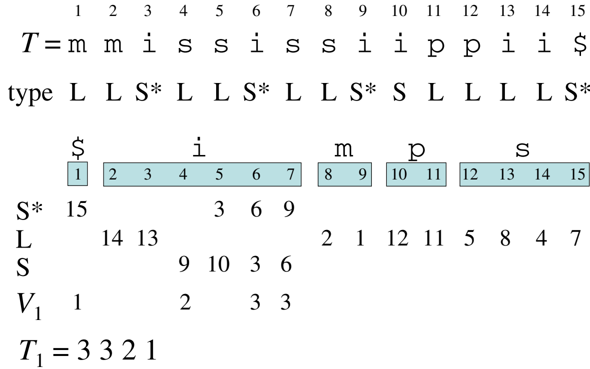


Fig. 1. An example of induced sorting for $T = mmississiippii\$$. S*substrings are located at positions $(15, 3, 6, 9)$ in T, and their positions are stored in SA. From them, L-type suffixes $(14, 13, 2, 1, 12, 11, 5, 8, 4, 7)$ are induced. Then S-type suffixes $(9, 10, 3, 6)$ are induced from L-type suffixes. We obtain names V_1 of the S*substrings, and finally obtain the shortened string $T_1 = 3321$.

Forth, BWT is also useful for many other applications, such as data compression [1], compressed full-text indexes [14], and pattern mining [11].

Since BWT is the result of the shuffled input text, the space of BWT is $n \lg \sigma$ bits. Therefore, the space of BWT is much smaller than that for the suffix array when $\sigma \ll n$, such as genome sequences.

To compute BWT, one usually constructs SA first, and then obtains BWT using (1). Although the total computational time is linear in the length of the input text [8,9,10], its working space is $n \lg n$ bits and is much larger than that for BWT. Previous studies [5,12] show how to compute BWT without SA. These algorithms incrementally build BWT by implicitly adding the suffixes from the shortest ones. Although these algorithms are remarkably simple, they require $O(n \log n)$ time and also slow in practice due to the large constant for keeping dynamic data structures. In another study [7], one divides input into the small blocks according to their first characters in suffixes so that the working space would be small. However it requires $O(n \log n)$ time and relies on the complex handling of long repetitions. Therefore, no previous work can compute BWT in linear time using small working space.

2.2 Storing Increasing Sequences

Let $s_1, s_2, \ldots, s_n$ be a strictly increasing sequence of integers such that $0 \le s_1 < s_2 < \cdots < s_n < U$. A naive representation of the sequence uses $n \lg U$ bits of space. Instead we can represent it succinctly by using a clever encoding with the following properties, which is rephrased from [6].

Lemma 1. *A sequence $s_1, s_2, \ldots, s_n$ of n integers such that $0 \le s_1 < s_2 < \cdots < s_n < U$ can be stored in a bit-stream of $n(2 + \lg \frac{U}{n})$ bits. The bit-stream can be*

constructed incrementally in $O(n)$ time in the sense that the integers can be given in arbitrary order, provided that both the value s_i and its index i are given. Furthermore, after $O(n)$ time preprocessing to the bit-stream to construct an auxiliary data structure of $O(n \log\log n / \log n)$ bits, the i-th smallest integer s_i ($1 \leq i \leq n$) is obtained in constant time.

Proof. The bit-stream consists of two parts: upper stream and lower stream. Each integer s_i is originally encoded in $\lg U$ bits, and its lower $\lg \frac{U}{n}$ bits are stored in the lower stream as it is. The upper $\lg n$ bits are stored in the upper stream after converting its original binary encoding to the following one. The upper stream is represented by a $0, 1$ vector $B[1, 2n]$, and the i-th number s_i is encoded by setting $B[i + \lfloor \frac{s_i}{n} \rfloor] = 1$. It is easy to show that each bit of B corresponds to at most one number in the sequence, and the bit position for s_i does not depend on other numbers. Therefore the upper stream can be constructed for any input order of the numbers.

The element s_i is obtained from the bit-streams as follows. The upper $\lg n$ bits of the binary encoding of s_i are computed by $\mathbf{select}(B, i) - i$, where $\mathbf{select}(B, i)$ is the position of i-th 1 in B and it is computed in constant time using an auxiliary data structure of $O(n \log\log n / \log n)$ bits [16] which is constructed by $O(n)$ time preprocessing. The lower $\lg \frac{U}{n}$ bits of the binary encoding of s_i are obtained directly from the lower stream in constant time. By concatenating the upper and the lower parts, we obtain s_i. □

3 Constructing SA Based on Induced Sorting

Our novel algorithm for computing BWT is based on the liner-time suffix array construction algorithm using purely induced sorting [15]. We will explain their algorithm here again for the sake of clarity. We call this algorithm *SAIS* (Suffix Array construction algorithm based on Induced Sorting).

First, we classify suffixes into two types; S-type, and L-type as follows.

Definition 1. *A suffix T_i is called S-type if $T_i < T_{i+1}$, and called L-type if $T_i > T_{i+1}$. The last suffix is defined as S-type.*

We also classify a character $T[i]$ to be S- or L-type if T_i is S- or L-type, respectively. We can determine the type of each suffixes in $O(1)$ time by scanning T once from right to left as follows. First, $T[n]$ is defined as S-type. Next, for i from $n-1$ to 1, we classify a suffix by using the following rule;

- T_i is S-type if ($T[i] < T[i+1]$) or ($T[i] = T[i+1]$ and T_{i+1} is S-type).
- T_i is L-type otherwise.

Obviously, in SA, the pointers for all the suffixes starting with a same character must span consecutively. Let's call a sub-array in SA for all the suffixes with a same character as a bucket. Specifically, we call c-bucket a bucket starting with a character c. Further, in the same bucket, all L-type suffixes precede to the S-type suffixes due to

their definition. Therefore, each bucket can be split into two sub-bucket with respect to the types of suffixes inside: we call them L- and S-type buckets each.

We also introduce S*-type suffixes.

Definition 2. *A suffix T_i is called S*-type if T_i is S-type and T_{i-1} is L-type (called Left-Most-S type in [15]). A character $T[i]$ is called S*-type if T_i is S*-type.*

Then, given sorted S*-type suffixes, we can induce the order of L-type and S-type suffixes as follows. These steps can be done in linear time.

- Given sorted S*suffixes, put all of them into their corresponding S-type buckets in SA, with their relative orders unchanged.
- Scan SA from the head to the end. For each item $SA[i]$, if $c = T[SA[i] - 1]$ is L-type, then put $SA[i] - 1$ to the current head of the L-type c-bucket and forward the current head one item to the right.
- Scan SA from the end to the head. For each item $SA[i]$, if $c = T[SA[i] - 1]$ is S-type, put $SA[i] - 1$ to the current end of the S-type c-bucket and forward the current end one item to the left.

Next, we explain how to obtain sorted S*suffixes in linear time.

We introduce S*substring.

Definition 3. *An S*substring is (i) a substring $T[i, j]$ with both $T[i]$ and $T[j]$ being S*characters, and there is no other S*character in the substring, for $i \neq j$; or (ii) $T[n]$.*

Let we denote these S*substrings in T as $R_1, R_2, \ldots, R_{n'}$ where R_i is the i-th S* substring in T. Let σ_1 be the number of different S*substrings in T. Then we assign names $V_i \in [1, \sigma_1]$ to R_i, $(i = 1, \ldots, n')$ so that $V_i < V_j$ if $R_i < R_j$ and $V_i = V_j$ if $R_i = R_j$. Finally, we construct a new text $T_1 = V_1, V_2, \ldots, V_{n'}$ whose length is n' and the alphabet size is σ_1.

We recursively apply the linear-time suffix array construction algorithm to T_1 and obtain the order of S*suffixes. Since the relative order of any two S*suffixes in T is the same for corresponding suffixes in T_1 [15], we can determine the order of the S*suffixes by using the result of the recursive algorithm.

To compute the names of S*substrings, we again use the induced algorithm modified that input are *unsorted* S*suffixes; we place unsorted S*suffixes at the end of S-type buckets, and apply inducing procedure; induce the order of L-type suffixes from S*suffixes, and the order of S-type suffixes from L-type suffixes. As a result, we obtain the sorted S*substrings. Then, we can assign names to each S*substring in linear time by checking their suffixes from the beginning to the ending. An example is shown in Figure 1.

Finally, to obtain the total computational cost, we use the following lemma;

Lemma 2. *[15] The length of T_1 is at most half of that of T.*

The SAIS algorithm for an input of length n requires $O(n)$ time and the time required to solve the same problem of half the length. Therefore, the total time complexity is $O(n)$.

4 Direct Construction of BWT

We explain our algorithm to obtain BWT without SA by modifying SAIS. We use the same definitions of S-, L-, S*-type suffixes/characters, and S*-substrings as in the previous section. In addition, we call BWT character $B[i] = T[SA[i]-1]$ ($B[i] = T[n]$ if $SA[i] = 1$) S-, L-, S*-type if $T[SA[i]]$ is S-, L-, S*-type character. Our idea is that we can simulate SAIS by using only S*substrings, in that we can induce L-type BWTs from sorted S*-BWTs and induce S-type BWTs from sorted L-type BWTs. Similarly, we can determine the order of S*substrings by using the inducing algorithm.

In our algorithm, we do not store SA, but keep S*substrings directly. Specifically, we keep following arrays, each of which stores the list of substrings for each character $c \in \Sigma$.

- S^*_c : Store substrings whose last character is c and S*-type.
- L_c : Store substrings whose last character is c and L-type.
- LS_c : Store substrings whose last character is c and S-type, and the next to the last character in the original text is L-type.
- S_c : Store substrings whose last character is c and S-type and the next to the last character in the original text is S-type.

These arrays support the following operations.

- $A.push_back(q)$: Add the substring q at the end of the array A.
- $A.pop_front()$: Return the substring at the front of the array A, and remove it.
- $A.reverse()$: Reverse the order in the array A.

For example, after the operation $A.push_back$(“abc”), $A.push_back$(“bcd”), and $A.push_back$(“cde”), $A = \{$“abc”, “bcd”, “cde”$\}$. The operation results are $A.pop_front()$ = “abc” and $A.pop_front()$ = “bcd”. We will discuss how to store these substrings in the section 5.

Note that, since our algorithm only uses these FIFO operations (and reverse operations.), we can implement this on external memory architecture easily.

In addition to these arrays, we keep following three arrays to store the result of BWT.

- $E[1, n']$: Store the end characters of S*substrings.
- BL_c : Store the result of L-type BWT for a c bucket.
- BS_c : Store the result of S-type BWT for a c bucket.

The overall algorithm is shown in the algorithm 1. All S*substrings are placed in S^*_c, and then moved to L_c, LS_c, and S_c in turn, and this is almost the same as in SAIS.

First, an input text $T[1, n]$ is decomposed into S*substrings $R_1, R_2, \ldots, R_{n'}$. Let σ_1 be the number of different S*substrings. Then we assign names V_i to R_i, $(i = 1, \ldots, n')$ from $1 \ldots \sigma_1$ so that $V_i < V_j$ if $R_i < R_j$ and $V_i = V_j$ if $R_i = R_j$. Then, we recursively call BWT-IS to determine the BWT of T_1. Let $B_1[1, n']$ be the result of BWT of T_1. Then, we induce the B from B_1. All steps except the assigning of names and induce are obviously done in linear time. We will see these steps in the following sections.

Algorithm 1. BWT-IS(T, n, k): The algorithm for computing BWT for T

Input: $T[1, n]$: An input text
n: An input length
k: A number of alphabets
Scan T once to classify all characters in T as S- or L-type (Also S*type).
Decompose T into S*substrings $R[1, \dots n']$ ($R[i] \in \{1, \dots, k\}^*$)
Name S*substrings by using the result of Induce(R), and get a new shortened string $T_1[1, n']$, $T_1[i] \in \{1, \dots, k'\}$.
if Each character in T_1 is unique **then**
  Directly compute B_1 from T_1
else
  B_1 = BWT-IS(T_1, n', k') // recursive call
end if
Decode S*strings B_1 into $R'[1, \dots, n']$
B = Induce(R')
Output: B

4.1 Induce BWT

We explain how to induce B, the BWT of original substring, from B_1, the BWT of S*substrings. The overall algorithm is shown in the algorithm 2.

First we lookup the original S*substrings, and keep the reversed ones. We reverse it because all operations are represented by pop_front and $push_back$ only. We do not require the reverse operation if we replace pop_front and $push_back$ with pop_back and $push_front$. Each substring is appended the character $c = k - 1$ which denotes the sign of the end of the string. We place these substring at S^*_c where c is the first character of the substring.

Second, for each character i from 1 to σ, we lookup L_i one by one. and check whether the first character c (Since S*strings are reversed, this corresponds to the last character in S*strings) is L-type or not. Particularly if $c \geq i$ then it is L-type and append it to the array L_c. If not, we place it at LS_i. After enumerating all the elements in L_i, we next lookup the substrings in S^*_i, and move it to L_c where c is the first character of each substring. Note that we can omit the check of L-type here because all the last characters in S^*_i should be L-type.

Third, for each character i from σ to 1, we lookup the substrings in the array S_i, and check whether it is empty or not. If so, we place the last character of E (we determine the position of S*substring) at the end of BS_i. After seeing all elements in S_i, we check LS_i similarly. In this case, we can omit the check whether it is empty because all substrings in this arrays should not be empty.

After obtaining the L-type and S-type BWTs, we just append these substrings in order and return it as the result of BWT.

4.2 Assigning Names to S*Strings

Let we explain how to compute the names of S*substrings. This is almost the same as in the induced algorithm in the previous section. As in SAIS algorithm, we apply

Algorithm 2. Induce(R): The algorithm for inducing BWT from the S*substrings

```
Input: R[1, n'] : A list of S* substrings.
for i = 1 to n' do
  U := Reverse(R_i)
  U.push_back(k − 1) // Sentinel
  c := U.pop_front().
  S*_c.push_back(U)
end for
for i = 1 to k do
  while U := L_i.pop_front() do
    c := U.pop_front()
    BL_i.push_back(c)
    if c < i then
      LS_i.push_back(c + U)
    else
      L_c.push_back(U)
    end if
  end while
  LS_i := Reverse(LS_i)
  while U := S*_i.pop_front() do
    c := U.pop_front()
    E.push_back(c)
    L_c.push_back(U)
  end while
end for
E := Reverse(E)
for i = k to 1 do
  while U := S_c.pop_front() do
    c := U.pop_front()
    if c < i then
      BS_i.push_front(c)
      S_c.push_back(U)
    else
      c2 := E_c.pop_front() // Reach the sentinel
      BS_i.push_back(c2)
    end if
  end while
  while U := LS_i.pop_front() do
    c := U.pop_front()
    BS_i.push_back(c)
    S_c.push_back(U)
  end while
end for
for i = 1 to k do
  BS_i := Reverse(BS_i)
  B := B + BL_i + BS_i
end for
Output: B
```

the algorithm to unsorted S*substrings, and as a result, we obtain sorted S*substrings. At this time, we do not place the BWT characters, and we keep S*substrings without removing. This is achieved by changing $pop_front()$ operations for the substring U in the algorithm 2 by the following $cycle()$ operation.

– $A.cycle()$: Return the character at the front of the array A, and moves it to the end of A.

For example, for $A =$ "abcd", $A.cycle() =$ "a" and after this operation $A =$ "bcda". This cycle operation can be done in $\mathrm{O}(1)$ time when the length of substring is $\mathrm{O}(\log n)$ bits. Otherwise, we keep pointers and simulate the cycle operation, which is also done in $\mathrm{O}(1)$ time.

Next, given sorted S*substrings, we calculate the names of them, which is trivial, and we place them in an original order to obtain the shortened string $T_1 = V_1, V_2, \ldots, V_{n'}$. However, since we don't have SA, we cannot find the original positions of each names directly.

First, we store positions $p_1, \ldots, p_{n'}$ of S*substrings in T using the data structure of Lemma 1. Namely, we define $q_i = cn + p_i$ for $i = 1, \ldots, n'$ where $c = T[p_i]$, and store all q_i by using Lemma 1 using $n(2 + \log \sigma) + o(n)$ bits. From q_i, c and p_i are obtained in constant time by $c = \lfloor q_i/n \rfloor$ and $p_i = q_i \bmod n$. We call this data structure Φ. To compute Φ, for each $c \in \Sigma$ we count the number of occurrences of c in T. This is done in $O(n)$ time for all c by using an integer array of $\sigma \log n \le n \log \sigma$ bits because $\sigma \le n$. Then we scan T from right to left to determine the positions $p_{n'}, p_{n'-1}, \ldots, p_1$ of S*substrings in this order. For each p_i we compute q_i and store it in Φ. We also store the S*substring located at p_i in a bucket. Namely, we obtain the head character $c = T[p_i]$, and store the substring into bucket for c. Each bucket stores the concatenation of S*substrings with the same head character. We store a pointer to indicate the position to append a new S*substring for each bucket. The space for storing all the pointers is $\mathrm{O}(\sigma \log n) = \mathrm{O}(n \log \sigma)$.

If the length of an S*substring is at most $\log_\sigma n$, it is encoded in at most $\log n$ bits, and therefore it takes constant time to append it to the end of a bucket. Otherwise, instead of storing the S*substring itself, we store the index p_i and the length of the S*substring. Because there exist at most $n \log \sigma / \log n$ S*substrings of length more than $\log_\sigma n$, we can store the indexes and lengths in $\mathrm{O}(n \log \sigma)$ bits.

The S*substrings are sorted by this modified induced-sorting in $\mathrm{O}(n)$ time and $\mathrm{O}(n \log \sigma)$-bit working space. During the modified induced-sorting for sorting S*substrings, we compute the following function Ψ and store it by the data structure of Lemma 1. Assume that in the original algorithm a suffix $SA[j]$ is induced from $SA[i]$, that is, $T[SA[i] - 1] = c$ and $SA[j]$ belongs to the bucket for c. In our modified algorithm corresponding this, we define $\Psi[j] = cn + i$. Because this induce-sorting scans SA from left to right and buckets are sorted with c, Ψ is strictly increasing. Therefore we can store Ψ in $n(2 + \log \sigma) + \mathrm{o}(n)$ bits by Lemma 1. We also store a bit-vector $P_{S^*}[i]$ indicating that the i-th suffix in the sorted order corresponds to an S*substring.

After the induced-sorting, we obtain $\Psi[i]$ for $i = 2, \ldots, n$ ($\Psi[1]$ is not defined because $SA[1]$ is the last suffix.). To compute names of S*substrings, we use another bit-vector $V[i]$ indicating that the S*substring corresponding to $V[i]$ is different from its left neighbor. To compute V, we scan P_{S^*} to enumerate S*substrings, and if $P_{S^*}[i] = 1$,

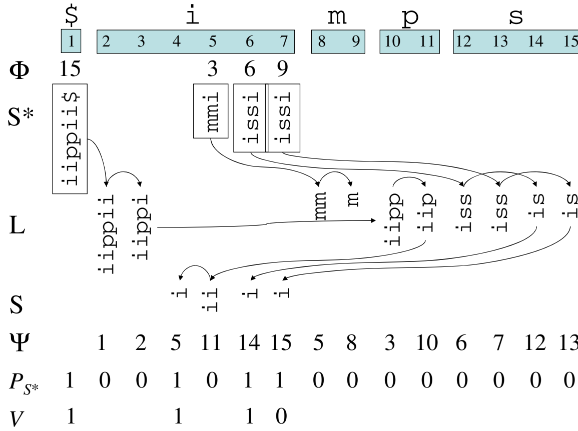


Fig. 2. An example of our modified induced-sorting corresponding to the original one in Figure 1. Positions of S*substrings are stored in Φ. We also store S*substrings in queues. Then L-type suffixes are induced from them. We actually move the substrings among queues, which are illustrated by arrows in the figure. The inverse of the movements are memorized as Ψ. The bit-vector P_{S^*} represents where are the S*substrings stored, and the bit-vector V encodes the names of them.

we compute $i := \Psi[i]$ repeatedly until we reach the position i that corresponds to the head of an S*substring, whose position in T is computed by $\Phi[i]$. We can determine if two adjacent S*substrings in sorted order are different or not in time proportional to their lengths. Therefore computing V takes $\mathrm{O}(n)$ time because the total length of all the S*substrings is $\mathrm{O}(n)$. If V is ready, we can compute the name of the S*substring corresponding to $P_{S^*}[i]$ by $\mathbf{rank}(P_{S^*}, i)$ which returns the number of 1's in $P_{S^*}[0, i]$. The **rank** function is computed in constant time using an $\mathrm{O}(n \log\log n / \log n)$-bit auxiliary data structure [16].

5 Succinct Representation of Substring Information

We explain the data structure for storing the list of (prefix of) S*strings. As noted in the previous section, these arrays should support $push_back(q)$, $pop_front()$, and $cycle()$ operations. Note that in our algorithm, $A.pop_front()$ is not called when A is empty. If the length of an S*substring is at most $\log_n \sigma$ bits, we directly store it and the operations $push_back(q)$ and $pop_front()$ are done in constant time. Otherwise, instead of storing the S*substring itself, we store the index p_i and the length of the S*substring. Because there exist at most $n \log \sigma / \log n$ S*substrings of length more than $\log_\sigma n$, we can store the indexes and lengths in $\mathrm{O}(n \log \sigma)$ bits.

Next, we estimate the size for BWTs in the recursive steps. It seems that in the worst case $n' = n/2$, the naive encoding of names will cost $n' \log n' = \Omega(n \log \sigma)$ bits. To guarantee that the string $T_1 = V_1, V_2, \ldots, V_{n'}$ is encoded in $\mathrm{O}(n \log \sigma)$ bits, we use the following encoding of the names, which is summarized as follows.

Lemma 3. *For a string T of length n with alphabet size σ, the shortened string $T_1 = V_1, V_2, \ldots, V_{n'}$ is encoded in $O(n \log \sigma)$ bits, and given i, the name of V_i and consecutive $O(\log n)$ bits at any position of the S*substring are computed in constant time. This encoding can be done in $O(n)$ time during the modified induced-sorting.*

Proof. The encoding consists of two types of codes; one is for S*strings whose lengths are at most $\frac{1}{2}\log_\sigma n$, and the other is for the rest. We call the former *short S*strings* and the latter *long S*strings*. For short S*strings, the code is its binary encoding itself, and for long S*strings the code is their names. To distinghish the type, we use a bit-vector $F[1, n']$ such that $F[i] = 1$ indicates V_i is a long S*string.

For computing the name V_i of a short S*string R_i, we obtain $\frac{1}{2}\log_\sigma n$ bits of T whose position is the beginning of R_i. To compute the name from the $\frac{1}{2}\log_\sigma n$ bits, we construct a decoding table such that for all bit patterns of $\frac{1}{2}\log_\sigma n$ bits which begin with the code of R_i we store the name V_i. This table can be constructed in $O(n)$ time in the modified induced-sorting. We scan S*strings in lexicographic order, and for each one we obtain its name and its position in T. From T we obtain the code of R_i, and fill a part of the table with the name. The size of the table is $O(\sigma^{\frac{1}{2}\log_\sigma n} \log n) = O(\sqrt{n} \log n)$ bits.

For long S*strings, we first construct the bit-vector F and the auxiliary data structure for $\mathbf{rank}$. Then during the modified induced-sorting, if there is a long S*substring R_i, we store its name in an array entry $W[\mathbf{rank}(F, i)]$. Because there exist at most $\frac{n}{\frac{1}{2}\log_\sigma n}$ long S*substrings, we can store their names in $O(n \log \sigma)$ bits.

To obtain consecutive $O(\log n)$ bits at any position of the S*substring, we use another bit-vector $G[1, n]$ such that $G[i] = 1$ stands for $T[i]$ is the head of an S*substring. We construct the auxiliary data structure for $\mathbf{select}$. By $\mathbf{select}(G, i)$ we obtain the position of V_i in T. Then it is obvious that any consecutive $O(\log n)$ bits are obtained in constant time. □

6 Time and Space Analysis

Our algorithm for an input of length n requires $O(n)$ time and the problem with the half length. Obviously, the time complexity is $O(n)$.

Next we analyze the space complexity. At the recursive step, the input space is $O(n \log \sigma)$ bits using the lemma 3. In addition, at each step, we keep the mapping information from the name to the original S*substring. We keep this by using an array list, which requires $n \lg \sigma$ bits of space in the worst case.

After $\lg \lg_\sigma n$ steps, the input length becomes $n' = n/2^{\lg \log_\sigma n} = n/\log_\sigma n$ and the size of suffix array for this input is $n' \lg n' = n/\log_\sigma n(\lg n - \lg \log_\sigma n) < n \lg \sigma$. Therefore we can use SAIS using the space less than $n \lg \sigma$. Therefore the total space is $O(n \log \sigma \log \log_\sigma n)$ bits.

7 Conclusion

In this paper, we present an algorithm for BWT. Our algorithm directly computes BWT, and does not require suffix arrays. Our algorithm works in linear time, and requires

$O(n \log \sigma \log \log_\sigma n)$ bits of space for any alphabets where n is the length of an original input space, and σ is the alphabet size.

As a next step, we consider how to efficiently build the longest common prefix array, or compressed suffix trees from BWT only. And we are also interested in the problem, whether can we compute BWT in linear time using $2n \lg \sigma + o(n \log \sigma)$ bits only?

References

1. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation (1994)
2. Farach, M.: Optimal Suffix Tree Construction with Large Alphabets. In: Proc. of FOCS, pp. 137–143 (1997)
3. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Compression boosting in optimal linear time. Journal of the ACM 52(4), 688–713 (2005)
4. Grossi, R., Vitter, J.S.: Compressed suffix arryas and suffix trees with applications to text indexing and string matching. Computing 35(2), 378–407 (2005)
5. Hon, W.-K., Lam, T.-W., Sadakane, K., Sung, W.-K., Yiu, S.-M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica 48(1), 23–36 (2007)
6. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. SIAM Journal on Computing 38(6), 2162–2178 (2009)
7. Kärkkäinen, J.: Fast bwt in small space by blockwise suffix sorting. Theoretical Computer Science 387(3), 249–257 (2007)
8. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. ACM 53(6), 918–936 (2006)
9. Kim, D.-K., Sim, J.S., Park, H.-J., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
10. Ko, P., Aluru, S.W.: Space-efficient linear time construction of suffix arrays. Discrete Algorithm 3, 143–156 (2005)
11. Lippert, R.: Space-efficient whole genome comparisons with burrows-wheeler transforms. j. of computational biology. Computational Biology 12(4) (2005)
12. Lippert, R., Mobarry, C., Walenz, B.: A space-efficient construction of the burrows wheeler transform for genomic data. In: Computational Biology (2005)
13. Chae Na, J., Park, K.: Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space. Theoretical Computer Science 385(1-3), 127–136 (2007)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2–61 (2007)
15. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. of DCC (2009)
16. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. of SODA, pp. 233–242 (2002)

Novel and Generalized Sort-Based Transform for Lossless Data Compression

Kazumasa Inagaki, Yoshihiro Tomizawa, and Hidetoshi Yokoo

Department of Computer Science, Gunma University
Kiryu 376-8515, Japan
{inagaki,tomizawa,yokoo}@daisy.cs.gunma-u.ac.jp

Abstract. We propose a new sort-based transform for lossless data compression that can replace the BWT transform in the block-sorting data compression algorithm. The proposed transform is a parametric generalization of the BWT and the RadixZip transform proposed by Vo and Manku (VLDB, 2008), which is a rather new variation of the BWT. For a class of parameters, the transform can be performed in time linear in the data length. We give an asymptotic compression bound attained by our algorithm.

1 Introduction

The block-sorting data compression algorithm [4] has been analyzed and evaluated both theoretically and empirically by researchers from the fields of information theory and algorithms. Several extensions to this algorithm and applications have been developed for various purposes [1]. Most of these extensions are modifications and generalizations of the *BWT* (the *Burrows–Wheeler Transform*), which is the core component of the block-sorting data compression algorithm. Few transformations that are completely different from the BWT have been developed. One such recent example is the *RadixZip Transform* proposed by Vo and Manku [9], which can replace the BWT in the block-sorting data compression algorithm.

In this paper, we propose a parametric generalization of the following two different transforms: the BWT and the *permute transform* in RadixZip. The proposed transform, called the *generalized radix permute* transform, or the *GRP* transform, bridges the two existing transforms. It also includes some of the finite-order variations [8], [7] of the BWT as special cases.

Data compression methods based on these transforms do not perform any context modeling in an apparent way. They are not classified into the class of statistical methods that make use of contexts to predict the following symbols. Actually, however, the transforms gather those symbols that occur in the same or similar contexts in a source string. In effect, they can be regarded as context modeling methods, each of which is distinguished in the length, or the *order*, of contexts it considers. While the original BWT uses unlimited order contexts, RadixZip uses the contexts of orders from zero to a predetermined upperbound.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 102–113, 2009.
© Springer-Verlag Berlin Heidelberg 2009

RadixZip begins at the zeroth order context to gather the statistics of source strings, and it must inevitably include low-order contexts. It tends to fail in utilizing higher order contexts, on which any high-performance data compression method should rely.

In our GRP transform, the lowest order at which the encoder begins to obtain the statistics of source strings can be selected arbitrarily. The transform is more general than the finite-order variations of the BWT since both the highest and lowest orders of contexts can be controlled. It uses the contexts from the shortest to the longest cyclically on the source string to predict the following symbols. We show that as long as the lowest order remains constant, both the forward and inverse transformations run in time linear in the string length. Even if the lowest order is fixed, a compression method combining the GRP transform and an appropriate second-step encoder can attain an asymptotic compression bound similar to that obtained on the block-sorting data compression method.

For space reasons, we concentrate only on presenting the GRP transform itself and its asymptotic analysis in compression performance. The GRP transform can be applied to any data of any length. However, for simplicity we present a version, in which we require the data lengths to be integer multiples of a parameter.

2 GRP Transform

2.1 Preliminaries

Let

$$x[1:n] = x_1 x_2 \cdots x_n$$

be an n-symbol string over an ordered alphabet A of size $|A|$. The string $x[i:j]$ represents a substring $x_i \cdots x_j$ for $1 \leq i \leq j \leq n$, and the empty string λ for $i > j$. The string $x[i:j]$ will be denoted also as x_i^j in the later analysis section. Similarly, a two-dimensional $n \times m$ matrix M of symbols is denoted by $M[1:n][1:m]$.

Similar to the BWT, the GRP transform converts the input string $x[1:n]$ to another string $y[1:n] \in A^n$ and an integer L. The GRP transform has two integer parameters. The first parameter is called the *block* length, which is denoted by ℓ. For simplicity it is assumed that the string length n is an integer multiple of ℓ, that is, $n = b\ell$ for an integer b.

In our transform, the input string is divided into b non-overlapping blocks of length ℓ, and saved as the column vectors of a matrix as follows:

$$T[1:\ell][1:b] = \begin{bmatrix} x_1 & x_{\ell+1} & x_{2\ell+1} & \cdots & x_{(b-1)\ell+1} \\ x_2 & x_{\ell+2} & x_{2\ell+2} & \cdots & x_{(b-1)\ell+2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_\ell & x_{2\ell} & x_{3\ell} & \cdots & x_{b\ell} \end{bmatrix}. \quad (1)$$

The second parameter of the GRP transform is called the *context order*, or simply *order*, which is a non-negative integer less than or equal to ℓ. Let d denote

the order. We first perform a left-cyclic shift of the top d rows of $T[1:\ell][1:b]$ and insert the results as the bottom rows of $T[1:\ell][1:b]$. Thus, the GRP transform is applied to the initial configuration of the $(\ell+d)\times b$ matrix given below:

$$T[1:\ell+d][1:b] = \begin{bmatrix} x_1 & x_{\ell+1} & \cdots & x_{(b-1)\ell+1} \\ x_2 & x_{\ell+2} & \cdots & x_{(b-1)\ell+2} \\ \vdots & \vdots & \vdots & \vdots \\ x_\ell & x_{2\ell} & \cdots & x_{b\ell} \\ x_{\ell+1} & x_{2\ell+1} & \cdots & x_1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{\ell+d} & x_{2\ell+d} & \cdots & x_d \end{bmatrix}. \quad (2)$$

As an example, consider the string

$$x[1:15] = \texttt{hotspotstopshot}, \quad (3)$$

and let $\ell = 3$ and $d = 2$. Then, $b = 5$ and

$$T[1:5][1:5] = \begin{bmatrix} \texttt{h} & \texttt{s} & \texttt{t} & \texttt{o} & \texttt{h} \\ \texttt{o} & \texttt{p} & \texttt{s} & \texttt{p} & \texttt{o} \\ \texttt{t} & \texttt{o} & \texttt{t} & \texttt{s} & \texttt{t} \\ \texttt{s} & \texttt{t} & \texttt{o} & \texttt{h} & \texttt{h} \\ \texttt{p} & \texttt{s} & \texttt{p} & \texttt{o} & \texttt{o} \end{bmatrix}. \quad (4)$$

2.2 Forward Transformation

The forward transformation of the GRP transform proceeds as follows:

1. /* Initialization */
   Convert the input string $x[1:n]$ into a matrix $T = T[1:\ell+d][1:b]$;
   Set $\boldsymbol{v}$:= the rightmost column vector of T;
   Set $L := b$;
2. **for** $i := 1$ **to** d **do**
   (a) Sort the column vectors of T in a stable manner according to the symbols of the ith row;
   /* The vector $\boldsymbol{v}$ may have moved to another column. */
   (b) Set L := the current column number of $\boldsymbol{v}$;
   end for
3. **for** $i := d+1$ **to** $d+\ell$ **do**
   (a) Output the ith row of T;
   (b) **if** $i = d+\ell$ **then** break;
   (c) Sort the column vectors of T in a stable manner according to the symbols of the ith row;
   end for
4. Concatenate the outputs of Step 3 (a) to form $y[1:n] = y_1 y_2 \cdots y_n$. The string $y[1:n]$ with the value of L is an output of the GRP transform.

For the string given in (3), the above procedure works as follows:

Step 2: $i = 1$

Perform a stable sort on the columns of T using the first row as the key to yield

$$T = \begin{bmatrix} \mathtt{h} & \mathtt{h} & \mathtt{o} & \mathtt{s} & \mathtt{t} \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \\ \mathtt{t} & \mathtt{t} & \mathtt{s} & \mathtt{o} & \mathtt{t} \\ \mathtt{s} & \mathtt{h} & \mathtt{h} & \mathtt{t} & \mathtt{o} \\ \mathtt{p} & \mathtt{o} & \mathtt{o} & \mathtt{s} & \mathtt{p} \end{bmatrix}.$$

Now, the column $\boldsymbol{v}$ has shifted to the second column. Thus, we have $L = 2$.

$i = 2$

Perform a stable sort on the columns in T by using the second row. This does not change the value of T since the row was already sorted. Now, $L = 2$ is stored.

Step 3: $i = 3$

The third row of T, `ttsot`, is outputted. Then, perform a stable sort on the columns in T by using the third row to yield

$$T = \begin{bmatrix} \mathtt{s} & \mathtt{o} & \mathtt{h} & \mathtt{h} & \mathtt{t} \\ \mathtt{p} & \mathtt{p} & \mathtt{o} & \mathtt{o} & \mathtt{s} \\ \mathtt{o} & \mathtt{s} & \mathtt{t} & \mathtt{t} & \mathtt{t} \\ \mathtt{t} & \mathtt{h} & \mathtt{s} & \mathtt{h} & \mathtt{o} \\ \mathtt{s} & \mathtt{o} & \mathtt{p} & \mathtt{o} & \mathtt{p} \end{bmatrix}.$$

$i = 4$

The fourth row of T, `thsho`, is outputted. Then, perform a stable sort on the columns in T by using the fourth row to yield

$$T = \begin{bmatrix} \mathtt{o} & \mathtt{h} & \mathtt{t} & \mathtt{h} & \mathtt{s} \\ \mathtt{p} & \mathtt{o} & \mathtt{s} & \mathtt{o} & \mathtt{p} \\ \mathtt{s} & \mathtt{t} & \mathtt{t} & \mathtt{t} & \mathtt{o} \\ \mathtt{h} & \mathtt{h} & \mathtt{o} & \mathtt{s} & \mathtt{t} \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \end{bmatrix}.$$

$i = 5$

The fifth row of T, `oopps`, is outputted. Since $i = \ell + d$ $(= 5)$, the concatenation of the above three outputs and the value of L yield

$$\begin{aligned} &y[1:15] = \mathtt{ttsotthshooopps}, \\ &L = 2. \end{aligned} \tag{5}$$

This is the result of the GRP transform of the string given in (3).

2.3 Inverse Transformation

The GRP transform is reversible. The inverse transformation of the GRP transform is more complicated than the forward transformation. Actually, in its description below, we will introduce a couple of auxiliary matrices that have not appeared in the forward transformation. However, these matrices are used only for explaining the transformation and are not essential for the transformation. The values of the parameters ℓ and d, and the string length n are the same in both the forward and inverse transformations. Hence, the number of blocks of the string, $b = n/\ell$, is an integer.

1. /∗ Initialization ∗/
   Store the string $y[1 : n]$ in an $\ell \times b$ matrix $S = S[1 : \ell][1 : b]$ according to
   $$S[i][j] := y[(i-1)b + j] \quad \text{for } 1 \leq i \leq \ell, 1 \leq j \leq b;$$
   Set its ℓth row to the bottom row of an $(\ell + d) \times b$ matrix U;
   /∗ The top $\ell - 1$ rows of U are initialized to be empty. ∗/
2. **for** $j := 1$ **to** $\ell - 1$ **do**
   (a) Sort the symbols in the $(\ell - j)$th row of S alphabetically, and put the result into the $(\ell + d - j)$th row of U;
   (b) Sort the columns of U in a stable manner so that its $(\ell + d - j)$th row corresponds to the $(\ell - j)$th row of S;
   end for
3. (a) Copy the bottom d rows of U into a $d \times b$ matrix V;
   (b) Considering the bottom row of V to be a significant part of the key, perform a radix sort on the columns of V (that is, perform a stable sort on the columns of V using the first to dth rows as the keys in this order);
   (c) Stack the matrix V on U;
   /∗ Note that U is now identical to T which is obtained immediately after Step 2 in the forward transformation. ∗/
4. Let $\boldsymbol{w}$ be the Lth column of U;
   Copy $\boldsymbol{w}$ to the bth column of an $(\ell + d) \times b$ matrix T;
5. **for** $j := 1$ **to** $b - 1$ **do**
   (a) From the columns of U that have not been copied to T, select the leftmost column that has the same d top symbols as the bottom d-symbol column of $\boldsymbol{w}$;
   (b) Set $\boldsymbol{w} :=$ the selected column, and copy it to T as the jth column;
   end for
6. /∗ The matrix T in (2) has been reconstructed. ∗/
   Recover the original string by
   $$x[i + (j-1)\ell] := T[i][j] \quad \text{for } 1 \leq i \leq \ell, \ 1 \leq j \leq b.$$

Before giving the general explanation of the reversibility of the above inverse transformation, we show how it works for the example given in (5).

Step 1

$$S = \begin{bmatrix} \mathtt{t} & \mathtt{t} & \mathtt{s} & \mathtt{o} & \mathtt{t} \\ \mathtt{t} & \mathtt{h} & \mathtt{s} & \mathtt{h} & \mathtt{o} \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \end{bmatrix}, \quad U = \begin{matrix} \\ \text{5th} \end{matrix} \begin{bmatrix} & & \vdots & & \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \end{bmatrix}.$$

Step 2: $j = 1$

$$U = \begin{bmatrix} & & \vdots & & \\ \mathtt{h} & \mathtt{h} & \mathtt{o} & \mathtt{s} & \mathtt{t} \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \end{bmatrix} \longrightarrow U = \begin{bmatrix} & & \vdots & & \\ \mathtt{t} & \mathtt{h} & \mathtt{s} & \mathtt{h} & \mathtt{o} \\ \mathtt{s} & \mathtt{o} & \mathtt{p} & \mathtt{o} & \mathtt{p} \end{bmatrix}.$$

$j = 2$

$$U = \begin{bmatrix} & & \vdots & & \\ \mathtt{o} & \mathtt{s} & \mathtt{t} & \mathtt{t} & \mathtt{t} \\ \mathtt{t} & \mathtt{h} & \mathtt{s} & \mathtt{h} & \mathtt{o} \\ \mathtt{s} & \mathtt{o} & \mathtt{p} & \mathtt{o} & \mathtt{p} \end{bmatrix} \longrightarrow U = \begin{bmatrix} & & \vdots & & \\ \mathtt{t} & \mathtt{t} & \mathtt{s} & \mathtt{o} & \mathtt{t} \\ \mathtt{s} & \mathtt{h} & \mathtt{h} & \mathtt{t} & \mathtt{o} \\ \mathtt{p} & \mathtt{o} & \mathtt{o} & \mathtt{s} & \mathtt{p} \end{bmatrix}.$$

Step 3

$$V = \begin{bmatrix} \mathtt{s} & \mathtt{h} & \mathtt{h} & \mathtt{t} & \mathtt{o} \\ \mathtt{p} & \mathtt{o} & \mathtt{o} & \mathtt{s} & \mathtt{p} \end{bmatrix} \longrightarrow V = \begin{bmatrix} \mathtt{h} & \mathtt{h} & \mathtt{o} & \mathtt{s} & \mathtt{t} \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \end{bmatrix},$$

$$U = \begin{bmatrix} \mathtt{h} & \mathtt{h} & \mathtt{o} & \mathtt{s} & \mathtt{t} \\ \mathtt{o} & \mathtt{o} & \mathtt{p} & \mathtt{p} & \mathtt{s} \\ \mathtt{t} & \mathtt{t} & \mathtt{s} & \mathtt{o} & \mathtt{t} \\ \mathtt{s} & \mathtt{h} & \mathtt{h} & \mathtt{t} & \mathtt{o} \\ \mathtt{p} & \mathtt{o} & \mathtt{o} & \mathtt{s} & \mathtt{p} \end{bmatrix}.$$

Step 4

$$T = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \mathtt{h} \\ \cdot & \cdot & \cdot & \cdot & \mathtt{o} \\ \cdot & \cdot & \cdot & \cdot & \mathtt{t} \\ \cdot & \cdot & \cdot & \cdot & \mathtt{h} \\ \cdot & \cdot & \cdot & \cdot & \mathtt{o} \end{bmatrix}.$$

Step 5: $j = 1$ $\quad$ $j = 2$ $\quad$ $j = 3$

$$T = \begin{bmatrix} \mathtt{h} & \cdot & \cdot & \cdot & \mathtt{h} \\ \mathtt{o} & \cdot & \cdot & \cdot & \mathtt{o} \\ \mathtt{t} & \cdot & \cdot & \cdot & \mathtt{t} \\ \mathtt{s} & \cdot & \cdot & \cdot & \mathtt{h} \\ \mathtt{p} & \cdot & \cdot & \cdot & \mathtt{o} \end{bmatrix}, \quad T = \begin{bmatrix} \mathtt{h} & \mathtt{s} & \cdot & \cdot & \mathtt{h} \\ \mathtt{o} & \mathtt{p} & \cdot & \cdot & \mathtt{o} \\ \mathtt{t} & \mathtt{o} & \cdot & \cdot & \mathtt{t} \\ \mathtt{s} & \mathtt{t} & \cdot & \cdot & \mathtt{h} \\ \mathtt{p} & \mathtt{s} & \cdot & \cdot & \mathtt{o} \end{bmatrix}, \quad T = \begin{bmatrix} \mathtt{h} & \mathtt{s} & \mathtt{t} & \cdot & \mathtt{h} \\ \mathtt{o} & \mathtt{p} & \mathtt{s} & \cdot & \mathtt{o} \\ \mathtt{t} & \mathtt{o} & \mathtt{t} & \cdot & \mathtt{t} \\ \mathtt{s} & \mathtt{t} & \mathtt{o} & \cdot & \mathtt{h} \\ \mathtt{p} & \mathtt{s} & \mathtt{p} & \cdot & \mathtt{o} \end{bmatrix}.$$

$\boxed{j = 4}$

$$T = \begin{bmatrix} \mathtt{h} & \mathtt{s} & \mathtt{t} & \mathtt{o} & \mathtt{h} \\ \mathtt{o} & \mathtt{p} & \mathtt{s} & \mathtt{p} & \mathtt{o} \\ \mathtt{t} & \mathtt{o} & \mathtt{t} & \mathtt{s} & \mathtt{t} \\ \mathtt{s} & \mathtt{t} & \mathtt{o} & \mathtt{h} & \mathtt{h} \\ \mathtt{p} & \mathtt{s} & \mathtt{p} & \mathtt{o} & \mathtt{o} \end{bmatrix}.$$

Step 6

$$x[1:15] = \mathtt{hotspotstopshot}.$$

2.4 Reversibility and Complexity

In order to show the reversibility of the GRP transform, we first note the symmetric relation between Step 3 of the forward transformation and Step 2 of the inverse transformation, which can be stated in the following lemma.

Lemma 1. *For i and j such that $i + j = d + \ell$, at the end of the jth iteration of the loop in Step 2 of the inverse transformation, the bottom $j + 1$ rows of U are identical to the bottom $d + \ell - i + 1$ rows of T in Step 3 (a) of the forward transformation.*

The above lemma can be proved by induction on j. The case of $j = 0$ corresponds to the initial state of the loop in Step 2 of the inverse transformation. In this state, the bottom row of U is simply a copy of the last output of Step 3 of the forward transformation. From the condition of the lemma, we have $i = d + \ell$ when $j = 0$, which corresponds to the last iteration of Step 3 of the forward transformation. Therefore, the statement of the lemma holds for $j = 0$. Starting from this initial state, we can show the validity of the statement from $j = 1$ to $j = \ell - 1$, inductively. Finally, we can show that, at the end of Step 2 of the inverse transformation, the bottom ℓ rows of U are identical to the bottom ℓ rows of T that are obtained immediately after Step 2 of the forward transformation.

In the inverse transformation, the process then moves on to Step 3, which is essentially the same as Step 2 of the forward transformation. Thus, we can establish the fact written as the comment in Step 3 of the inverse transformation that U and T are identical. The rest of the inverse transformation, namely Steps 4 and 5, can be easily validated by the stability of the sorting process of Step 2 of the forward transformation. In this way, we can prove the reversibility of the GRP transform.

Here, we make a brief comment about the time complexity of the GRP transform. We assume that each stable sorting process can be performed linearly by using bucket sorting. Under this assumption, the forward transformation can be done in $O(b(\ell + d)) = O(n + bd)$ time.

The inverse transformation seems more time-demanding than the forward transformation since Step 5 of the inverse transformation requires string searching. Actually, however, we can perform this process of string searching in $O(bd)$

time by using the result of Step 3 (b). In Step 5, for every column, say $\boldsymbol{w}$, of U, we must find a column that has the same d top symbols as the d-symbol bottom column of $\boldsymbol{w}$. Step 3 has already established the correspondence between every $\boldsymbol{w}$ and at least one such column. Moreover, after the step, all columns are arranged in lexicographic order of the top d-symbols. Therefore, it is not so difficult to find the column that satisfies the condition of Step 5. The total time required in Step 5 is proportional to the total number of symbols in the top d rows in T. In summary, we can prove the following theorem.

Theorem 1. *For any string of length n, both the forward and inverse transformations run in $O(n+bd)$ time, where b is the number of blocks of the string, and d is the context order of the GRP transform. For any fixed order d, therefore, they run in time linear in the string length n.*

Remark: In this paper, we have presented only the case of $n = b\ell$. We have already succeeded in eliminating this assumption. The GRP transform can be modified to be applicable to any string of any length. We have also assumed that the order d satisfies $0 \leq d \leq \ell$. The transform can be extended for larger values of d than ℓ so that it includes existing transforms as special cases. Specific correspondences follow.

GRP with $\ell = 1$ and $d = n$: BWT;
GRP with $\ell = 1$ and $d < n$: ST transform [7],[8];
GRP with $d = 0$: Permute transform in RadixZip.

3 Information Theoretical Analysis

3.1 Second-Step Algorithm

Similar to the BWT, the GRP transform requires a second-step algorithm for actual compression. In addition to the same algorithms as those adopted in the block sorting compression algorithm [1], [5], we may incorporate new encoding methods that rely on the nature of the GRP transform. For example, the output string of the GRP transform is a concatenation of ℓ blocks; each block can be encoded by distinct encoding methods. In this paper, however, we consider only the simplest case for the analysis of asymptotic performance of the proposed transform.

We encode the output $y[1:n]$ of the GRP transform by using the Move-to-Front (MTF) encoding scheme [3], which produces a list of integers from 1 to the size $|A|$ of the source alphabet. Then, we encode each integer in the list using the δ code of Elias [6]. The codeword length for integer t is upperbounded by

$$f(t) = \log t + 2\log(\log t + 1) + 1 \quad \text{bit}, \tag{6}$$

where all logarithms in this paper are taken to base 2. We will ignore the codeword for the integer component L of the output, for simplicity.

3.2 Asymptotic Characterization

The following analysis is based mainly on the model in [2].

Although the order d can be extended to an arbitrary integer as mentioned above, we restrict its range to $0 \leq d \leq \ell$. We first shift the blocks of the input string by d symbols. That is, we assume $x[(j-1)\ell + d + 1 : j\ell + d]$ to be the jth block $(1 \leq j \leq b-1)$. Thus, we consider only the substring $x[d+1 : (b-1)\ell + d]$. We ignore $x[1 : d]$, which serves only as the context to the following symbols in the first column of matrix T in (2), and is encoded in the last column in a virtual context. We focus on the kth symbol $x[(j-1)\ell+d+k]$ in the jth block $(1 \leq k \leq \ell)$. We define the context of this kth symbol by $x[(j-1)\ell + 1 : (j-1)\ell + d + k - 1]$. The context of the kth symbol in the jth block is a substring of $d+k-1$ symbols that immediately precedes $x[(j-1)\ell + d + k]$. In the forward transformation, each kth symbol appears in the $(d+k)$th row of T, and is included somewhere in $y[(k-1)b + 1 : kb]$ of the transformed string. Note that when the kth symbols $\{x[(j-1)\ell + d + k]\}_{j=1}^{b-1}$ are transformed into $y[(k-1)b + 1 : kb]$ in Step 3 of the forward transformation, their contexts are lexicographically arranged as columns consisting of top $d+k-1$ rows of T. That is, the same contexts appear consecutively as columns in T (see Fig. 1).

In Fig. 1, $y_{1|c}$ is the ith symbol of $y[(k-1)b + 1 : kb]$ that appeared in context c. Thus, $y_{1|c}$, $y_{2|c}$, $y_{3|c}, \ldots, y_{N(c)|c}$ are the symbols that appear sequentially in this order in context c in the transformed string, where $N(c)$ is the number of blocks that have the same prefix c. In general, for an arbitrary string $a_1^i = a_1 a_2 \cdots a_i \in A^i$ $(0 \leq i \leq \ell)$, $N(a_1^i)$ represents the number of blocks appeared in the entire $b-1$ blocks that begin with the prefix a_1^i. For the empty string λ, $N(\lambda)$ equals $b-1$. Let $z_1^{N(c)} = z_1 z_2 \cdots z_{N(c)}$ be a sequence of positive integers that is obtained from $y_{1|c} y_{2|c} \cdots y_{N(c)|c}$ by using the MTF scheme. For every symbol $a \in A$, if $y_{i|c}$ equals a with $i = t_1, t_2, \ldots, t_{N(ca)}$, then we have

$$z_{t_1} \leq |A|, \tag{7}$$

$$z_{t_i} \leq t_i - t_{i-1} \quad \text{for } 2 \leq i \leq N(ca). \tag{8}$$

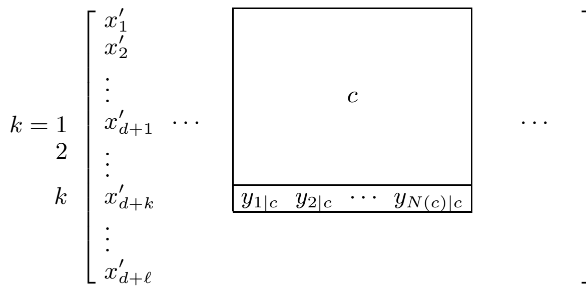


Fig. 1. Matrix T after transformation of kth symbols of blocks in Step 3 of the forward transformation

According to the proof of Theorem 1 in [2], the sum of the lengths of the codewords representing the symbol a in context c can be bounded by

$$f(|A|) + \sum_{i=2}^{N(ca)} f(t_i - t_{i-1}) \leq N(ca)\, f\Big(\frac{N(c)+|A|}{N(ca)}\Big). \tag{9}$$

The kth symbols $\{x[(j-1)\ell + d + k]\}_{j=1}^{b-1}$ are transformed into $b-1$ symbols in $y[(k-1)b+1 : kb]$, and then converted into a sequence of integers by the MTF scheme. Let $l_k(y_{(k)}^{b-1})$ denote the sum of the codeword lengths representing the $b-1$ kth symbols. Then, we have the following result, which is a direct consequence of the inequality (9).

Lemma 2. *For any fixed integer k in $[1, \ell]$, the kth symbols $\{x[(j-1)\ell + d + k]\}_{j=1}^{b-1}$ of $b-1$ blocks can be encoded with the length $l_k(y_{(k)}^{b-1})$, which satisfies*

$$\begin{aligned} l_k(y_{(k)}^{b-1}) &\leq \sum_{a_1^{d+k-1}} \sum_{a_{d+k}} N(a_1^{d+k}) f\Big(\frac{N(a_1^{d+k-1})+|A|}{N(a_1^{d+k})}\Big) \\ &= \sum_{a_1^i \in A^i} N(a_1^i)\, f\Big(\frac{N(a_1^{i-1})+|A|}{N(a_1^i)}\Big) \quad \textit{for} \quad i = d+k, \end{aligned} \tag{10}$$

where the second summation is taken over a_{d+k} so that $N(a_1^{d+k})$ is greater than zero.

Suppose that an input string is generated from a stationary and ergodic source $\{X_i\}_{i=1}^{\infty}$ with probability measure p and entropy rate H, where X_i takes values in the alphabet A. Let $p(a_1^m)$ denote the probability that X_1^m is equal to $a_1^m \in A^m$, and $p(a_m \mid a_1^{m-1})$ denote the conditional probability of $a_m \in A$ given $a_1^{m-1} \in A^{m-1}$. The conditional entropy is defined by

$$\begin{aligned} &H(X_m \mid X_1^{m-1}) \\ &= -\sum_{a_1^{m-1}} p(a_1^{m-1}) \sum_{p(a_m|a_1^{m-1})\neq 0} p(a_m \mid a_1^{m-1}) \log p(a_m \mid a_1^{m-1}). \end{aligned} \tag{11}$$

Similarly, $p(a_{d+1}^{d+\ell} \mid a_1^d)$ represents

$$p(a_{d+1}^{d+\ell} \mid a_1^d) = \prod_{i=1}^{\ell} p(a_{d+i} \mid a_1^{d+i-1}) = \frac{p(a_1^{d+\ell})}{p(a_1^d)}. \tag{12}$$

The conditional joint entropy $H(X_{d+1}^{d+\ell} \mid X_1^d)$ is defined by

$$H(X_{d+1}^{d+\ell} \mid X_1^d) = -\sum_{a_1^d \in A^d} p(a_1^d) \sum_{p(a_{d+1}^{d+\ell}|a_1^d)\neq 0} p(a_{d+1}^{d+\ell} \mid a_1^d) \log p(a_{d+1}^{d+\ell} \mid a_1^d).$$

The entropy rate of a stationary source can be characterized in multiple ways.

$$\begin{aligned} H &= \lim_{m\to\infty} H(X_m \mid X_1^{m-1}) && (13)\\ &= \lim_{d\to\infty} \frac{1}{\ell} H(X_{d+1}^{d+\ell} \mid X_1^d) \ \text{ for any } \ell && (14)\\ &= \lim_{\ell\to\infty} \frac{1}{\ell} H(X_{d+1}^{d+\ell} \mid X_1^d) \ \text{ for any } d. && (15) \end{aligned}$$

For arbitrary fixed integers $\ell > 0$ and $b > 1$, consider a prefix of length $(b-1)\ell$ that begins at the $(d+1)$th place of an infinite string x over A. Divide the prefix into $b-1$ blocks of non-overlapping substrings each of length ℓ, and let $N_x(a_1^i)$ represent the number of blocks whose prefix is equal to a_1^i, where $0 \le i \le \ell$. Define a set

$$\tilde{D}_b(a_1^i, \varepsilon) = \left\{ x \in A^\infty : \left| \frac{N_x(a_1^i)}{b-1} - p(a_1^i) \right| > \varepsilon p(a_1^i) \right\} \tag{16}$$

for fixed b and $\varepsilon > 0$. Moreover, we introduce the following set:

$$D_b(d, \ell, \varepsilon) = \bigcup_{i=d}^{d+\ell} \bigcup_{a_1^i} \tilde{D}_b(a_1^i, \varepsilon) \tag{17}$$

When we encode a $b\ell$-symbol prefix of x by using the proposed scheme, we represent the codeword length corresponding to the substring $x[d{+}1 : (b{-}1)\ell{+}d]$ by $l(y^{b-1})$. That is,

$$l(y^{b-1}) = \sum_{k=1}^{\ell} l_k(y_{(k)}^{b-1}). \tag{18}$$

We can now bound the codeword length for each source symbol in our encoding scheme in a series of theorems, which we will present without proofs.

Theorem 2. *For any fixed $\ell > 0$, $k \le \ell$, $d \le \ell$, and $\varepsilon_k > 0$, there exists a positive integer $B_k = B_k(d, \ell, \varepsilon_k)$ such that for any $b > B_k$ and $x \notin D_b(d, \ell, \varepsilon_k)$,*

$$\frac{l_k(y_{(k)}^{b-1})}{(b-1)\ell} \le \frac{1}{\ell} H(X_{d+k} \mid X_1^{d+k-1}) + \frac{2}{\ell} \log\Big(H(X_{d+k} \mid X_1^{d+k-1}) + 1 \Big) + 1 + \hat{\varepsilon}_k,$$

where $\hat{\varepsilon}_k \to 0$ as $\varepsilon \to 0$.

Theorem 3. *For any fixed $\ell > 0$, $d \le \ell$, and $\varepsilon > 0$, there exists a positive integer $B = B(d, \ell, \varepsilon)$ such that for any $b > B$ and $x \notin D_b(d, \ell, \varepsilon)$,*

$$\frac{l(y^{b-1})}{(b-1)\ell} \le \frac{1}{\ell} H(X_{d+1}^{d+\ell} \mid X_1^d) + 2\log\Big(\frac{1}{\ell} H(X_{d+1}^{d+\ell} \mid X_1^d) + 1 \Big) + 1 + \hat{\varepsilon}, \tag{19}$$

where $\hat{\varepsilon} \to 0$ as $\varepsilon \to 0$.

Theorem 4. *For any stationary and ergodic source with entropy rate* H*, the codeword length per symbol satisfies*

$$\lim_{\substack{b\to\infty \\ \ell\to\infty}} \frac{l(y^{b-1})}{(b-1)\ell} \le H + 2\log(H+1) + 1 \tag{20}$$

with probability one.

The above theorem shows that the symbolwise application of the MTF scheme followed by Elias' δ code simply yields the same bound as that obtained by the block-sorting data compression method when used under the same conditions. Thus, to eliminate the additive terms other than the entropy rate H in (20), we must incorporate such techniques as alphabet extension into our scheme. In addition to such theoretical techniques, more practical ones like run length encoding have been combined with the BWT to improve its actual compression performance. We have to introduce similar techniques to the proposed scheme to make it applicable to real data. Furthermore, although the bound in (20) can be attained by setting, e.g., $b = O(\sqrt{n})$ and $\ell = O(\sqrt{n})$, as $n \to \infty$, these parameters also have to be optimized from a practical viewpoint.

4 Conclusion

We have proposed a sort-based transform, called the GRP transform, which is a parametric generalization of the BWT. Future work includes efficient implementation of the transform for $d \ge \ell$ and evaluation of practical compression schemes based on it.

References

1. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer, Heidelberg (2008)
2. Arimura, M., Yamamoto, H.: Asymptotic optimality of the block sorting data compression algorithm. IEICE Trans. Fundamentals E81-A(10), 2117–2122 (1998)
3. Bentley, J.L., Sleator, D.D., Tarjan, R.E., Wei, V.K.: A locally adaptive data compression scheme. Comm. ACM 29(4), 320–330 (1986)
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto (1994)
5. Deorowicz, S.: Improvements to Burrows–Wheeler compression algorithm. Software—Practice and Experience 30(13), 1465–1483 (2000)
6. Elias, P.: Universal codeword sets and representations of the integers. IEEE Trans. Inform. Theory IT-21, 194–203 (1975)
7. Nong, G., Zhang, S., Chan, W.H.: Computing inverse ST in linear complexity. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 178–190. Springer, Heidelberg (2008)
8. Schindler, M.: A fast block-sorting algorithm for lossless data compression. In: DCC 1997, Proc. Data Compression Conf., 469, Snowbird, UT (1997)
9. Vo, B.D., Manku, G.S.: RadixZip: Linear time compression of token streams. In: Very Large Data Bases: Proc. 33rd Intern. Conf. on Very Large Data Bases, Vienna, pp. 1162–1172 (2007)

A Two-Level Structure for Compressing Aligned Bitexts*

Joaquín Adiego[1], Nieves R. Brisaboa[2], Miguel A. Martínez-Prieto[1], and Felipe Sánchez-Martínez[3]

[1] Dept. de Informática, Universidad de Valladolid, Spain
{jadiego,migumar2}@infor.uva.es
[2] Database Lab, Universidade da Coruña, Spain
brisaboa@udc.es
[3] Dept. de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, Spain
fsanchez@dlsi.ua.es

Abstract. A *bitext*, or *bilingual parallel corpus*, consists of two texts, each one in a different language, that are mutual translations. Bitexts are very useful in linguistic engineering because they are used as source of knowledge for different purposes. In this paper we propose a strategy to efficiently compress and use bitexts, saving, not only space, but also processing time when exploiting them. Our strategy is based on a two-level structure for the vocabularies, and on the use of *biwords*, a pair of associated words, one from each language, as basic symbols to be encoded with an ETDC [2] compressor. The resulting compressed bitext needs around 20% of the space and allows more efficient implementations of the different types of searches and operations that linguistic engineerings need to perform on them. In this paper we discuss and provide results for compression, decompression, different types of searches, and bilingual snippets extraction.

1 Introduction

The amount of multilingual texts is growing very fast due to multilingual digital libraries and legal requirements in countries and supra-national entities with more than one official language. Two texts that are mutual translations are usually referred to as a *bilingual parallel corpus* or, in short, as a *bitext*. The growing availability of bitexts has enabled the development on many natural language processing applications that use bitexts as source of knowledge.

Usually, bitexts get *aligned* before exploiting them; a standard *text alignment* process allows to establish word correspondences between the two texts of the bitext. Aligned bitexts can be used in applications involving both languages (machine translation, cross-language information retrieval, extraction of bilingual lexicons, etc) or in monolingual applications (syntactic parsing, word sense induction, word sense disambiguation, etc.) that use the bitexts as a bridge to project the linguistic knowledge available in one language to another one [11].

* Funded by Spanish projects TIN2006-15071-C03-01, TIN2006-15071-C03-02 and TIN2006-15071-C03-03. The work of Miguel A. Martínez-Prieto is supported by a fellowship granted by the Regional Government of Castilla y León and the European Social Fund.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 114–121, 2009.
© Springer-Verlag Berlin Heidelberg 2009

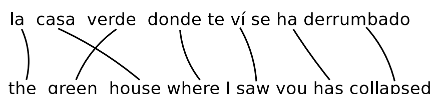


Fig. 1. Spanish–English word-aligned sentence

We present a strategy to compress bitexts that we called *Two-level Compressor for Aligned Bitexts* (2LCAB). Our strategy is designed to facilitate the use of the most interesting features of bitexts, because, in our compressed representation, obtaining the words in one language aligned with a word in the other language is simply done by using a vocabulary, instead of processing the whole aligned bitext. In addition, 2LCAB obtains compression ratios around 20% and allows a more efficient processing of the aligned bitexts that the uncompressed form.

2 Word-Aligned Bitexts

A *bitext* is a text written in two languages. In words of Melamed, “bitexts are one of the richest sources of linguistic knowledge because the translation of a text into another language can be viewed as a detailed annotation of what that text means” [10].

A bitext in which the translation relationship among the words in one text (left) and the words in the other text (right) has been established is usually referred to as a *word-aligned* bitext; the task of establishing such relationships is known as *word alignment*.

The *word alignment* task [15] connects words in the left sentence L with words in the right sentence R. The result is a bigraph for the words in L and the words in R with an arc between word $l \in L$ and word $r \in R$ if and only if they are mutual translations. Figure 1 shows an example of a Spanish–English word-aligned sentence.

For this research the bigraph representing a word-aligned bitext is stored as a sequence of pairs of two words, each one from a different language, that are mutual translations in the bitext. Therefore, for this research the word-aligned bitext of the example in Figure 1 is represented as the following sequence of pairs:
`(la,the) (,green) (casa,house) (verde,) (donde,where) (te,) (,I) (ví,saw) (,you) (se,) (ha,has) (derrumbado,collapsed)`

Notice that some words are associated to an “empty word”, e.g. `(te,)`. This is either because that word is not aligned with another word in the other text, or because its alignment has been discarded due to a crossing, e.g. `(,green)`. In this work we have used one-to-one word alignments obtained with the help of the open-source GIZA++ [15] toolkit.[1]

3 Compression of Natural-Language Texts

The key to the success of natural language text compression is the use of a *word-based* model, so that the text is regarded as a sequence of words. This poses the overhead of managing a large source alphabet, but in large text collections the vocabulary size is relatively

[1] `http://code.google.com/p/giza-pp/`

insignificant because of Heaps Law [5]. In order to be searchable, semi-static models have been used in compressed text databases, to ensure that the codeword assigned to a word does not change across the text. Thus, a pattern can be compressed and directly searched for in the compressed text without decompressing it. This is also essential to allow local decompression of text passages in order to present them to the final users.

End-Tagged Dense Code (ETDC) [2] is a *word-based* compression technique where the first bit of each byte is reserved to flag whether the byte is the last one of its codeword (*stopper*) or not (*continuer*); this flag is enough to ensure that the code is a prefix code regardless of the content of the other 7. The flag bit in ETDC permits Boyer-Moore-type searching [1] and random access. Simple encode and decode procedures can be used to obtain the codeword C_i corresponding to a position i in the sorted vocabulary ($C_i = \text{encode}(i)$) and, symmetrically, to obtain the position i corresponding to a specific codeword C_i ($i = \text{decode}(C_i)$).

3.1 Compression of Bitexts

Compression of bitexts is a subfield of natural language text compression. In spite of its relevance, only few previous works have been found in the literature. In [14] text compression methods are considered for its extension to bitext compression considering exact correspondences between two words, and synonymy relationships between the words in both texts (as given by a thesaurus). These parallel predictions are then combined with PPM [3] ones. The weighting of both models are carefully tuned improving PPM compression ratios on separate texts.

Text alignment is proposed in [4] as a way to enable multilingual text compression. The algorithm stores one of the texts (L) as it is, and the other one (R) as a collection of pointers to the translation of the substring in the L text. These relationships are determined by means of an alignment algorithm that uses some additional linguistic resources, such as a lemmata dictionary in L and a bilingual glossary, among others.

4 Two-Level Compressor for Aligned Bitexts (2LCAB)

Our strategy, called *Two-Level Compressor for Aligned Bitexts (*2LCAB*)*, is based on two main ideas: (i) the use of *biwords* [9], pairs of aligned words, as the basis of the model, that is, as the symbols to compress, and (ii) the use of a two level structure for the representation of the vocabularies, where the vocabulary of biwords, at the second level, is represented in compressed form using the vocabularies of the first level.

Figure 2 shows a conceptual description of this scheme. At the first level two vocabularies are stored, one for each language. Each of them stores the words of the corresponding language sorted by the number of biwords they take part in. The "empty word" is also represented in both dictionaries. On the second level, each pair of words (biword) is represented as the concatenation of the codewords assigned to each word in the pair using ETDC. That is, each biword is used as a single symbol in the *biword vocabulary*. In this second-level vocabulary the ranking of biwords is performed in accordance with their frequencies in the bitext.

Four strings are the output of the compression process. `Lv` and `Rv` contain sorted left and right vocabularies. `BWv` stores the biword vocabulary where biwords are represented in compressed form as explained above. These three strings constitute the header

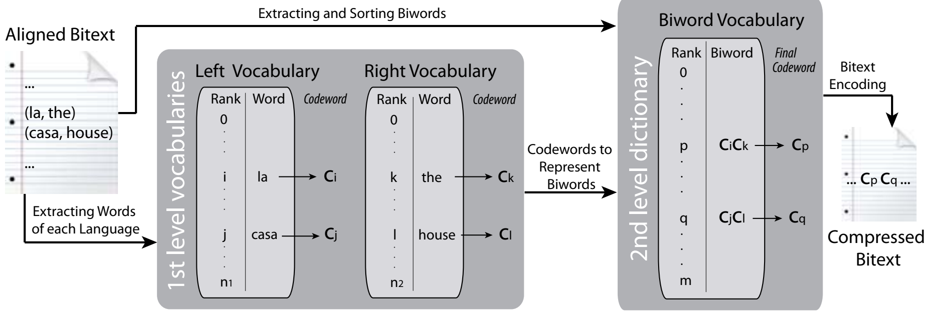


Fig. 2. Conceptual description of the 2LCAB strategy

of the compressed bitext. The fourth string contains the compressed bitext, where each pair of words is represented by the codeword corresponding to the ETDC codeword assigned to its biword from its position in the second level vocabulary.

4.1 Compression and Decompression

Our strategy is based on a semi-static approach; therefore, it is necessary to make two passes over the (aligned) bitext. In the first one, the aligned bitext is pair-to-pair parsed and, in addition to the three vocabularies aforementioned, a hash table of pairs (biword, codeword) is built. In the second pass, the compression process looks for each biword in the hash table and outputs its corresponding codeword. The compression process is completed in $O(n)$ time overall, where n is the number of biwords in the bitext.

The decompression process begins by loading the strings `Lv` and `Rv` to get the left and right vocabularies. These strings are stored in vectors V_l and V_r, respectively. Then, the string `BWv` is read and V_l and V_r are used to rebuilt the biword vocabulary, which is stored in vector V_b, where each biword is explicitly represented by its pair of words so as to improve the efficiency of the decompression process. Building V_b takes $O(b)$ time, where b is the number of entries in the `BWv` vocabulary. Then, the compressed bitext is processed by decoding each codeword. Given a codeword C_i, the simple decoding function of the ETDC is used to obtain the corresponding position $i = \text{decode}(C_i)$ in the biword vocabulary ($V_b[i]$). The decompression process is completed in $O(T)$ time, where T is the number of biwords in the bitext.

4.2 Processing the Compressed Bitext

Our representation allows to process the bitext without decompressing it. In fact, only decompressing small snippets is necessary for most applications, and only when they need to show the snippet to the user. Semantic relationships between languages in the bitext suggest specific search possibilities such as: (i) to find all the occurrences of a word in the bitext, that is, all the occurrences of biwords that include it; and (ii) to find all the possible translations of a word, that is, all the biwords for a specific word.

To process the compressed bitext we start loading and storing the strings Lv and Rv into V_l and V_r vectors, respectively. Also, two hash tables are built from Lv and Rv. Then, the string BWv is read and stored in main memory. To facilitate the searches, a *bitmap* with a bit for each byte in BWv is built. In this bitmap 0-bits correspond to *continuer* bytes in BWv whereas 1-bits correspond to *stopper* bytes.

Searching the Occurrences of a Word in the Bitext. This operation is useful to retrieve all the contexts (snippets) in which each biword appears, that is, to find all the occurrences of a specific biword and decode its snippets. Given a word, and the language in which it is represented, we first find it in the corresponding first-level vocabulary (left or right hash table depending of the language supplied); this process takes $O(1)$ time. Once the codeword is retrieved it is searched in BWv to find those biwords in which the word appears. This is carried out using any well-known exact pattern matching algorithm (such as KMP [7] or BM [1]) slightly modified to avoid possible false matchings due to the fact that ETDC codes are not suffix codes, and, therefore, a codeword can be a suffix of another one. This overhead in searches is negligible because checking the previous byte is only necessary when a matching occurs, which is infrequent [2].

To determine the language of a codeword found in the biwords vocabulary at position p a $\mathrm{rank}_1(p)$ operation on the bitmap is done. If an *even* value is obtained, the word belongs to the left vocabulary, whereas an *odd* value means that the word belongs to the right vocabulary. If the found matching corresponds to the adequate language, the codeword of the biword is computed as $C = \mathrm{encode}(\mathrm{rank}_1(p)/2)$. Then, that codeword is added to the trie of searched codewords that will be used by the multiple-pattern matching algorithm over the compressed bitext. The search of all the required biwords takes $O(b)$ time, where b is the size of BWv because the rank operation, to check the language correspondence, only takes $O(1)$ [12]. At the end of this process the codewords in the trie will encode all the biwords where the searched word appears. We choose Set Horspool [6,13], as search algorithm because it is an efficient choice for very small sets of searched patterns on large alphabets. Set Horspool outputs all the occurrences of the required biwords in the compressed text. This search takes $O(m)$ time, where m is the size in bytes of the compressed text.

To find the context where each specific translation (biword) of a word is found, it is only necessary to decompress a snippet around each occurrence. Doing this is straightforward by using the ETDC decode procedure.

Searching All the Possible Translations of a Word. This operation allows to find all the correspondences of a word in a language with words in the other language. One of the main advantages of our approach is that to find all the possible translations of a word in the bitext it is not necessary to read the bitext, because all the biwords (possible translations) are represented in the vocabulary of biwords. Therefore, it is only necessary to search for the codeword of that specific word in the BWv string using the strategy already explained.

5 Experimental Evaluation

All the experiments were performed on a Debian 4 Etch operating system, running on an AMD Athlon Dual Core processor at 2 GHz and with 2 GB of RAM. We used g++

4.1.2 compiler with full optimization. We used heterogeneous corpora with different languages pairs to evaluate the influence of the similarity between the two languages of the bitext in the compression ratio. Furthermore, we used bitexts of different size for each language pair. More precisely, we used bitexts of around 1, 10, and 100 MB where larger bitexts contained the smaller ones (for Spanish–Galician we only used bitexts of 1, and 10 MB). The following corpora were used:

- a Spanish–Catalan (`es-ca`) bitext from *El Periódico de Catalunya*,[2] a daily newspaper published both in Catalan and Spanish;
- a Spanish–Galician (`es-gl`) bitext from *Diario Oficial de Galicia*,[3] the bulletin of the Government of Galicia, published both in Galician and Spanish; and
- bitexts for German–English (`de-en`), Spanish–English (`es-en`) and French–English (`fr-en`) from the *European Parliament Proceedings Parallel Corpus* [8].

To evaluate the success of 2LCAB in obtaining a competitive compression ratio, we compare it with some well-known state-of-the-art compressors such as GZIP, BZIP2 and PPMDI [16], this last one as a representative PPM [3]. Moreover, to evaluate the effect of our strategy of using a biword-oriented model, we also implemented ETDC compression over the bitext using two different word-oriented models. In one case (1V) we just used one vocabulary to store the words of both languages. In the other case (2V) we used two different vocabularies, one for each language.

Table 1 summarizes some data about the bitexts and the compression ratios obtained by the different compressors. Notice that 2LCAB achieves very good compression ratios (some times the best one) when the size of the bitext is medium or large. However, GZIP, BZIP2, and PPM, not being semi-static, provide better results for small files. 2LCAB outperforms GZIP for 10 MB bitexts (except for `es-gl` which is a special case because Spanish and Galician are closely-related languages) and only PPM, as would be expected, can compete with 2LCAB for large bitexts. Nevertheless, GZIP, BZIP2, and PPM, as dynamic compressors, do not permit random access, nor direct searching.

We do not compare our compressor against those described in Section 3.1 because we have not found any available implementation. However, Conley and Klein [4] compare their TRANS approach with GZIP and BZIP2 and they conclude that TRANS is slightly better than BZIP2 (an improvement of 1% is reported). In any case, the authors do not consider the size of the auxiliary files that TRANS requires to decompress the bitext; thus, TRANS compression ratio would be worse than that of BZIP2.

Table 2 shows compression and decompression times (in seconds) for two bitext collections: `es-ca` and `es-en`. Similar times were obtained for the remaining bitexts. The times reported correspond to the average time obtained for 5 different executions. 2LCAB is always the fastest in compression, around 3-6 times faster than BZIP2 and PPM for medium-large file sizes. Only for small files GZIP shows a slightly better performance. When decompressing, only GZIP is slightly faster than our approach, which is much faster than BZIP2 (up to 9 times) and PPM (up to 35 times).

Table 3 shows the time required to retrieve all the occurrences of a specific word when searching the `es-en` bitext of 100 MB. Considering the number of biwords a

[2] http://www.elperiodico.com
[3] http://www.xunta.es/diario-oficial

Table 1. Compression ratios

BITEXT	SIZE (MB)	Words Left	Words Right	Biwords	GZIP	BZIP2	PPM	1V	2V	2LCAB
es-gl	1.09	6488	6543	7219	17.09%	11.21%	8.72%	32.09%	35.30%	25.35%
	10.55	24983	25284	29855	16.91%	10.58%	8.68%	28.67%	29.82%	18.53%
es-ca	1.18	15594	14939	19336	31.48%	23.09%	20.75%	47.89%	50.02%	42.78%
	11.53	54115	52256	78825	31.13%	22.18%	20.33%	36.68%	37.21%	26.72%
	105.36	161132	159216	292994	30.79%	21.95%	20.17%	32.75%	32.51%	19.97%
es-en	1.08	9169	6696	21301	31.83%	22.33%	20.07%	43.47%	42.85%	37.30%
	10.91	30486	19465	93544	31.67%	21.73%	19.98%	35.61%	34.54%	25.99%
	110.60	81868	51353	347866	31.26%	21.22%	19.48%	32.36%	31.19%	20.86%
fr-en	1.08	8211	6493	20491	31.64%	21.99%	19.78%	42.30%	41.97%	36.33%
	10.74	25536	19045	86353	31.43%	21.42%	19.74%	35.10%	34.11%	25.55%
	109.45	65877	50418	322618	31.26%	21.22%	19.52%	32.40%	31.21%	21.04%
de-en	1.08	9957	6514	21159	32.46%	22.76%	20.66%	44.69%	44.09%	39.16%
	10.94	39287	19305	90815	32.27%	22.13%	20.47%	36.38%	35.31%	27.44%
	110.86	139012	51018	357753	32.22%	22.05%	20.37%	32.75%	31.57%	22.01%

Table 2. Compression and decompression times

	es-ca								es-en							
SIZE	*Compression time (secs.)*				*Decompression time (secs.)*				*Compression time (secs.)*				*Decompression time (secs.)*			
(MB)	GZIP	BZIP2	PPM	2LCAB	GZIP	BZIP2	PPM	2LCAB	GZIP	BZIP2	PPM	2LCAB	GZIP	BZIP2	PPM	2LCAB
1	0.24	0.41	0.51	0.35	0.04	0.14	0.55	0.05	0.23	0.38	0.47	0.25	0.03	0.13	0.52	0.04
10	1.32	4.16	4.61	1.10	0.16	1.40	5.15	0.33	1.36	4.10	4.70	1.01	0.14	1.35	5.16	0.29
100	10.19	38.32	41.86	6.88	1.38	12.47	45.34	1.29	11.56	41.21	45.95	7.81	1.46	13.56	50.27	1.46

Table 3. Searching times. The values between brackets show the average time and standard deviation needed to locate all the biwords in which a given word occurs.

	Biwords	Occurrences	1V $\overline{time}$	1V σ	2V $\overline{time}$	2V σ	2LCAB $\overline{time}$	2LCAB σ
[A]	41.20	754.07	0.635	0.034	0.617	0.036	0.119 (*0.006*)	0.030 (*0.002*)
[B]	14.30	226.77	0.636	0.009	0.613	0.014	0.099 (*0.008*)	0.044 (*0.003*)
[C]	4.77	69.87	0.641	0.063	0.614	0.045	0.066 (*0.007*)	0.012 (*0.002*)
[D]	1.47	10.87	0.631	0.017	0.615	0.018	0.061 (*0.007*)	0.011 (*0.002*)

words is associated to, we defined four categories: [A]: $x_A \geq 25$; [B]: $7 \leq x_B \leq 24$; [C]: $3 \leq x_C \leq 6$; and [D]: $x_D \leq 2$, where $x_{category}$ indicates the number of biwords a word must be associated to, in order to belong to that category. Then, we built four groups of 30 words randomly chosen among those in each category.

We used 1V and 2V to compare the efficiency of 2LCAB in searching processes. Notice that, 1V performs a simple pattern-matching process to find the codeword of the desired word. However, 2V needs to check if the found codeword belongs to the appropriate side of the bitexts, that is, if it is in the desired language; this is achieved by calculating if the found codeword is in a even or an odd position.

2LCAB is the fastest choice in all the cases, improving 5-10 times both 1V and 2V compressed bitexts. Notice that the number of occurrences has a stronger influence in 2LCAB than in 1V and 2V. This is because when there are many biwords associated to a word the Set Horspool algorithm handles a more complex trie composed by all the codewords of those biwords. Finally, notice that 2V is always slightly better than 1V. This is mainly due to the fact that 2V gets better compression ratios than 1V.

6 Conclusions

2LCAB has been proposed as strategy to compress word-aligned bitexts. It provides very good compression ratios and it is the fastest option for compressing and decompressing large bitext. Its main property is that bitexts can be efficiently exploited because different kind of searches and local decompression can be effectively performed over the compressed bitext without needing to decompress it. Another interesting result of this research is how the similarity between the languages in the bitext affects the number of different biwords, the number of total biwords and, therefore, the compression ratio.

References

1. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Comm. of ACM 20(10), 762–772 (1977)
2. Brisaboa, N.R., Fariña, A., Navarro, G., Paramá, J.R.: Lightweight natural language text compression. Information Retrieval 10(1), 1–33 (2007)
3. Cleary, J.G., Witten, I.H.: Data Compression Using Adaptive Coding and Partial String Matching. IEEE Trans. on Communications COM-32(4), 396–402 (1984)
4. Conley, E.S., Klein, S.T.: Using alignment for multilingual text compression. Intl. J. of Foundations of Computer Science 19(1), 89–101 (2008)
5. Heaps, H.S.: Inf. Retrieval - Computational and Theoretical Aspects. Academic Press, London (1978)
6. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. & Exper. 10, 501–506 (1980)
7. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. of Computing 6(2), 323–350 (1977)
8. Koehn, P.: Europarl: A parallel corpus for statistical machine translation. In: Proc. of the 10th Machine Translation Summit, pp. 79–86 (2005), http://www.statmt.org/europarl/
9. Martínez-Prieto, M.A., Adiego, J., Sánchez-Martínez, F., de la Fuente, P., Carrasco, R.C.: On the use of word alignments to enhance bitext compression. In: Data Compres. Conf., p. 459 (2009)
10. Melamed, I.D.: Emplirical methods for exploting parallel texts. MIT Press, Cambridge (2001)
11. Mihalcea, R., Simard, M.: Parallel texts. Natural Language Eng. 11(3), 239–246 (2005)
12. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv., 39(1) (2007)
13. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, Cambridge (2002)
14. Nevill-Manning, C.G., Bell, T.C.: Compression of parallel texts. Information Processing & Management 28(6), 781–794 (1992)
15. Och, F.J., Ney, H.: A systematic comparison of various statistical alignment models. Comp. Linguistics 29(1), 19–51 (2003)
16. Shkarin, D.: PPM: One Step to Practicality. In: Data Compres. Conf., pp. 202–211 (2002)

Directly Addressable Variable-Length Codes⋆

Nieves R. Brisaboa[1], Susana Ladra[1], and Gonzalo Navarro[2]

[1] Universidade da Coruña, Spain
{brisaboa,sladra}@udc.es
[2] Dept. of Computer Science, Univ. of Chile
gnavarro@dcc.uchile.cl

Abstract. We introduce a symbol reordering technique that implicitly synchronizes variable-length codes, such that it is possible to directly access the i-th codeword without need of any sampling method. The technique is practical and has many applications to the representation of ordered sets, sparse bitmaps, partial sums, and compressed data structures for suffix trees, arrays, and inverted indexes, to name just a few. We show experimentally that the technique offers a competitive alternative to other data structures that handle this problem.

1 Introduction

Variable-length coding is at the heart of Data Compression [23,21]. It is used, for example, by statistical compression methods, which assign shorter codewords to more frequent symbols. It also arises when representing integers from an unbounded universe: Well-known codes like γ-codes and δ-codes are used when smaller integers are to be represented using fewer bits.

A problem that frequently arises when variable-length codes are used is that it is not possible to access directly the i-th encoded element, because its position in the encoded sequence depends on the sum of the lengths of the previous codewords. This is not an issue if the data is to be decoded from the beginning, as in many compression methods. Yet, the issue arises recurrently in the field of *compressed data structures*, where the compressed data should be accessible and manipulable in compressed form. A partial list of structures where the need to directly access variable-length codes arises includes Huffman and other similar encodings of text collections [14,15,1], compression of inverted lists [23,4], compression of suffix trees and arrays (for example the Ψ function [20] and the LCP array [7]), compressed sequence representations [19,6], partial sums [13], sparse bitmaps [19,18,3] and its applications to handling sets over a bounded universe supporting predecessor and successor search, and a long so on. It is indeed a common case that an array of integers contains mostly small values, but the need to handle a few large values makes programmers opt for allocating the maximum space instead of seeking for a more sophisticated solution.

⋆ Funded in part (for the Spanish group) by MEC grant (TIN2006-15071-C03-03); and for the third author by Fondecyt Grant 1-080019, Chile.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 122–130, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The typical solution to provide direct access to a variable-length encoded sequence is to regularly sample it and store the position of the samples in the encoded sequence, so that decompression from the last sample is necessary. This introduces a space and time penalty to the encoding that often hinders the use of variable-length coding in many cases where it would be beneficial.

In this paper we show that, by properly reordering the target symbols of a variable-length encoding of a sequence, direct access to any codeword (achieving constant time per symbol of the target alphabet) is easy and fast. This is a kind of *implicit* data structure that introduces synchronism in the encoded sequence without using asymptotically any extra space. We show some experiments demonstrating that the technique is not only simple, but also competitive in time and space with existing solutions in several applications.

2 Basic Concepts

Statistical encoding. Let $X = x_1 x_2 \dots x_n$ be a sequence of symbols to represent. A way to compress X is to order the distinct symbol values by frequency, and identify each value x_i with its position p_i in the ordering, so that smaller positions occur more frequently. Hence the problem is how to encode the p_is into variable-length bit streams c_i, giving shorter codewords to smaller values. Huffman coding [11] is the best code (i.e., achieving the minimum total length for encoding X) such that (1) assigns the same codeword to every occurrence of the same symbol and (2) is a prefix code.

Coding integers. In other applications, the x_is are directly the numbers p_i to be encoded, such that the smaller values are assumed to be more frequent. One can still use Huffman, but if the set of distinct numbers is too large, the overhead of storing the Huffman code may be prohibitive. In this case one can directly encode the numbers with a fixed prefix code that gives shorter codewords to smaller numbers. Well-known examples are γ-codes and δ-codes [23,21].

Vbyte coding. [22] is a particularly interesting code for this paper. In its general variant, the code splits the $\lfloor \log(p_i + 1) \rfloor$ bits needed to represent p_i by splitting it into blocks of b bits and storing each block into a *chunk* of $b + 1$ bits. The highest bit is 0 in the chunk holding the most significant bits of p_i, and 1 in the rest of the chunks. For clarity we write the chunks from most to least significant, just like the binary representation of p_i. For example, if $p_i = 25 = 11001_2$ and $b = 3$, then we need two chunks and the representation is $\underline{0}011\ \underline{1}001$.

Compared to an optimal encoding of $\lfloor \log(p_i + 1) \rfloor$ bits, this code loses one bit per b bits of p_i, plus possibly an almost empty final chunk. Even when the best choice for b is used, the total space achieved is still worse than δ-encoding's performance. In exchange, Vbyte codes are very fast to decode.

Partial sums are an extension of our problem when X is taken as a sequence of nonnegative *differences* between consecutive values of sequence $Y = y_1, y_2, \dots y_n$, so that $y_i = sum(i) = \sum_{1 \leq j \leq i} p_j$. Hence, X is a compressed representation of Y that exploits the fact that consecutive differences are small numbers. We are then

interested in obtaining efficiently $y_i = sum(i)$. Sometimes we are also interested in finding the largest $y_i \leq v$ given v, that is, $search(v) = \max\{i,\ sum(i) \leq v\}$. Let us call $S = sum(n)$ from now on.

3 Previous Work

From the previous section, we end up with a sequence of n concatenated variable-length codes. Being usually prefix, there is no problem in decoding them in sequence. We now outline several solutions to the problem of giving direct access to them, that is, extracting any p_i efficiently, given i. Let us call N the length in bits of the encoded sequence.

The classical solution samples the sequence and stores absolute pointers only to the sampled elements, that is, to each h-th element of the sequence. Access to the $(h+d)$-th element, for $0 \leq d < h$, is done by decoding d codewords starting from the h-th sample. This involves a space overhead of $\lceil n/h \rceil \lceil \log N \rceil$ bits and a time overhead of $O(h)$ to access an element, assuming we can decode each symbol in constant time. The partial sums problem is also solved by storing some sampled y_i values, which are directly accessed for *sum* or binary searched for *search*, and then summing up the p_is from the last sample.

A dense sampling is used by Ferragina and Venturini [6]. It represents p_i using just its $\lfloor \log(p_i + 1) \rfloor$ bits, and sets pointers to *every* element in the encoded sequence, giving the ending points of the codewords. By using two levels of pointers (absolute ones every $\Theta(\log N)$ values and relative ones for the rest) the extra space for the pointers is $O(\frac{n \log\log N}{\log N})$, and constant-time access is possible.

Sparse bitmaps solve the direct access and partial sums problems when the differences are strictly positive. The bitmap $B[1, S]$ has a 1 at positions y_i.

We make use of two complementary operations that can operate in constant time after building $o(S)$-bit directories on top of B [12,2,16]: $rank(B, i)$ is the number of 1s in $B[1, i]$, and $select(B, i)$ is the position in B of the ith 1 (similarly, $select_0(B, i)$ finds the ith 0). Then $y_i = select(B, i)$ and $search(v) = rank(B, v)$ easily solve the partial sums problem, whereas $x_i = select(B, i) - select(B, i-1)$ solves our original access problem. We can also accommodate zero-differences by setting bits $i + y_i$ in $B[1, S + n]$, so $y_i = select(B, i) - i$, $search(v) = rank(B, select_0(B, v))$, and $x_i = select(B, i) - select(B, i-1) - 1$.

A drawback of this solution is that it needs to represent B explicitly, thus it requires $S + o(S)$ bits, which can be huge. There has been much work on sparse bitmap representations that can lighten space requirements [19,10,18].

4 Our Technique: Reordered Vbytes

We make use of the generalized Vbyte coding described in Section 2. We first encode the p_is into a sequence of $(b+1)$-bit chunks. Next we separate the different chunks of each codeword. Assume p_i is assigned a codeword C_i that needs r

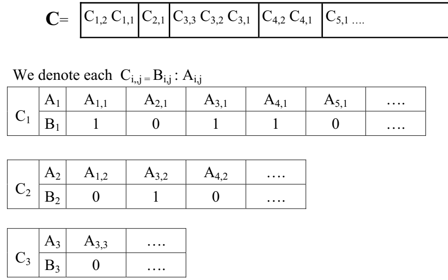


Fig. 1. Example of reorganization of the chunks of each codeword

chunks $C_{i,r}, \ldots, C_{i,2}, C_{i,1}$. A first stream, C_1, will contain the $n_1 = n$ least significant chunks (i.e., rightmost) of every codeword. A second one, C_2, will contain the n_2 second chunks of every codeword (so that there are only n_2 codewords using more than one chunk). We proceed similarly with C_3, and so on. As the p_is add up to S, we need at most $\lceil \frac{\log S}{b} \rceil$ streams C_k (usually less).

Each stream C_k will be separated into two parts. The lowest b bits of the chunks will be stored contiguously in an array A_k (of $b \cdot n_k$ bits), whereas the highest bits will be concatenated into a bitmap B_k of n_k bits. Figure 1 shows the reorganization of the different chunks of a sequence of five codewords. The bits in each B_k identify whether there is a chunk of that codeword in C_{k+1}.

We set up *rank* data structures on the B_k bitmaps, which answer *rank* in constant time using $O(\frac{n_k \log \log N}{\log N})$ extra bits of space, being N the length in bits of the encoded sequence[1]. Solutions to *rank* are rather practical, obtaining excellent times using 37.5% extra space on top of B_k, and decent ones using up to 5% extra space [8,18].

The overall structure is composed by the concatenation of the B_ks, that of the A_ks, and pointers to the beginning of the sequence of each k. These pointers need at most $\lceil \frac{\log S}{b} \rceil \lceil \log N \rceil$ bits overall, which is negligible. In total there are $\sum_k n_k = \frac{N}{b+1}$ chunks in the encoding (note N is a multiple of $b+1$), and thus the extra space for the *rank* data structures is just $O(\frac{N \log \log N}{b \log N})$.

Extraction of the i-th value of the sequence is carried out as follows. We start with $i_1 = i$ and get its first chunk $b_1 = B_1[i_1] : A_1[i_1]$. If $B_1[i_1] = 0$ we are done with $p_i = A_1[i_1]$. Otherwise we set $i_2 = rank(B_1, i_1)$, which sends us to the correct position of the second chunk of p_i in B_2, and get $b_2 = B_2[i_2] : A_2[i_2]$. If $B_2[i_2] = 0$, we are done with $p_i = A_1[i_1] + A_2[i_2] \cdot 2^b$. Otherwise we set $i_3 = rank(B_2, i_2)$ and so on[2].

Extraction of a random codeword requires $\lceil \frac{N}{nb} \rceil$ accesses; the worst case is at most $\lceil \frac{\log S}{b} \rceil$ accesses. Thus, in case the numbers to represent come from a statistical variable-length coding, and the sequence is accessed at uniformly

[1] This is achieved by using blocks of $\frac{1}{2} \log N$ bits in the *rank* directories [12,2,16].

[2] To avoid the loss of a value in the highest chunk we use in our implementation the variant of Vbytes we designed for text compression called ETDC [1].

distributed positions, we have the additional benefit that shorter codewords are accessed more often and are cheaper to decode.

4.1 Partial Sums

The extension to partial sums is as for the classical method: We store in a vector $Y[0, n/s]$ the accumulated sum at regularly sampled positions (say every hth position). We store in $Y[j]$ the accumulated sum up to p_{hj}. The extra space required by Y is thus $\lceil n/h \rceil \lceil \log S \rceil$ bits. With those samples we can easily solve the two classic operations $sum(i)$ and $search(v)$.

We compute $sum(i)$ by accessing the last sampled $Y[j]$ before p_i, that is $j = \lfloor i/h \rfloor$ and adding up all the values between p_{hj+1} and p_i. To add those values we first sequentially add all the values between $A_1[hj+1]$ and $A_1[i]$. We compute $s_1 = hj + 1$ and $e_1 = i$ and $Acc_1 = \sum_{s_1 \leq r \leq e_1} A_1[r]$; then we compute $s_2 = rank(B_1, s_1 - 1) + 1$ and $e_2 = rank(B_1, e_1)$ and again $Acc_2 = \sum_{s_2 \leq r \leq e_2} A_2[r]$; and so on for the following levels. The final result is $Y[j] + \sum Acc_k \cdot 2^{b(k-1)}$. Notice that for a sampling step h this operation costs at most $O(\frac{h \log S}{b})$.

To perform $search(v)$ we start with a binary search for v in vector Y. Once we find the sample $Y[j]$ with the largest value not exceeding v, we start a sequential scanning and addition of the codewords until we reach v. That is, we start with $total = Y[j]$, $b_1 = hj+1$, $b_2 = rank(B_1, b_1-1)+1$, $b_3 = rank(B_2, b_2-1)+1$ and so on. The value of each new codeword is computed using its different chunks at levels $k = 1, 2, \ldots$, adding $A_k[b_k] \cdot 2^{b(k-1)}$ and incrementing b_k, as long as $k = 1$ or $B_{k-1}[b_{k-1} - 1] = 1$. Once computed, the value is added to $total$ until we exceed the desired value v; then $search(v) = b_1 - 1$. Notice that we compute only one $rank$ operation per sequence B_k, as the next chunks to read in each B_k follow the current one. The total cost for a search operation is $O(\log \frac{n}{h})$ for the binary search in the samples array plus $O(\frac{h \log S}{b})$ for the sequential addition of the codewords following the selected sample $Y[j]$.

5 Applications and Experiments

We detail now some applications of our scheme, and compare it with the current solutions used in those applications. This section is not meant to be exhaustive, but rather a proof of concept, illustrative of the power and flexibility of our idea.

We implemented our technique with b values chosen manually for each level (in many cases the same b for all). We prefer powers of 2 for b, so that faster aligned accesses are possible. We implemented $rank$ using the 37.5%-extra space data structure by González et al. [8] (this is space over the B_k bitmaps).

Our machine is an Intel Core2Duo E6420@2.13Ghz, with 32KB+32KB L1 Cache, 4MB L2 Cache, and 4GB of DDR2-800 RAM. It runs Ubuntu 7.04 (kernel 2.6.20-15-generic). We compiled with gcc version 4.1.2 and the options `-m32 -O9`.

5.1 High-Order Compressed Sequences

Ferragina and Venturini [6] gave a simple scheme (FV) to represent a sequence of symbols $S = s_1 s_2 \ldots s_n$ so that it is compressed to its high-order empirical

Table 1. Space for encoding the 2-byte blocks and individual access time

Method	Space (% of original file)	Time (nanosec per extraction)
Dense sampling (FV, $c = 20$)	94.34%	298.4
Sparse sampling ($h = 14$)	68.44%	557.2
Vbyte ($b = 7$) sampling ($h = 14$)	75.90%	305.7
Ours ($b = 8$)	68.46%	216.1

entropy and any $O(\log n)$-bit substring of S can be decoded in constant time. This is extremely useful because it permits replacing *any* sequence by its compressed variant, and any kind of access to it under the RAM model of computation retains the original time complexity.

The idea is to split S into *blocks* of $\frac{1}{2}\log n$ bits, and then sort the blocks by frequency. Once the sequence of their positions p_i is obtained, it is stored using a dense sampling, as explained in Section 3. We compare their dense sampling proposal with our own representation of the p_i numbers, as well as a classical variant using sparse sampling (also explained in Section 3).

We took the first 512 MB of the concatenations of collections FT91 to FT94 (Financial Times) from TREC-2 (`http://trec.nist.gov`), and chose 2-byte blocks, thus $n = 2^{29}$ and our block size is 16 bits.

We implemented scheme FV, and optimized it for this scenario. There are 5,426 different blocks, and thus the longest block description has 12 bits. We stored absolute 32-bit pointers every $c = 20$ blocks, and relative pointers of $\lceil \log((c-1)\cdot 12)\rceil = 8$ bits for each block. This was the setting giving the best space, and let us manage pointers using integers and bytes, which is faster.

We also implemented the classical alternative of Huffman-encoding the different blocks, and setting absolute samples every h codewords. This gives us a space-time tradeoff, which we set to $h = 14$ to achieve space comparable to our alternative. In addition, we implemented a variant with the same parameters but using Vbyte-encoding, with $b = 7$ (i.e., using bytes as chunks).

We used our technique with $b = 8$, which lets us manipulate bytes and thus is faster. The space was almost the same with $b = 4$, but time was worse.

Table 1 shows the results. We measure space as a fraction of the size of the original 512 MB text, and time as nanoseconds per extraction, where we average over the time to extract all the blocks of the sequence in random order.

The original FV method poses much space overhead (achieving almost no compression). This, as expected, is alleviated by the sparse sampling, but the access times increase considerably. Yet, our technique achieves much better space and noticeable better access times than FV. When using the same space of a sparse sampling, on the other hand, our technique is three times faster. Sparse sampling can achieve 54% space (just the bare Huffman encoding), at the price of higher access times. The Vbyte alternative is both larger and slower than ours. In fact, the Vbyte-encoding itself, without the sampling overhead, occupies 67.8% of the original sequence, very close to our representation (which will be similar to an Vbyte encoding using $b = 8$).

Table 2. Space for encoding the differential Ψ array and individual *sum* time under different schemes. The b sequences refer to the (different) consecutive b values used in the arrays C_1, C_2, etc. "Ours*" uses 5% extra space for *rank* on the bitmaps.

Method	Space (% of original file)	Time (nanosec per Ψ computation)
Sadakane's	66.72%	645.5
Ours $b = 8$	148.06%	629.0
Ours $b = 4$	103.44%	675.6
Ours $b = 2$	85.14%	919.8
Ours $b = 0, 2, 4, 8$	73.96%	757.1
Ours* $b = 0, 2, 4, 8$	67.88%	818.7
Ours $b = 0, 4, 8$	76.85%	742.7

5.2 Compressed Suffix Arrays

Sadakane [20] proposed to represent the so-called Ψ array, useful to compress suffix arrays [9,17], by encoding its consecutive differences along the large areas where Ψ is increasing. A γ-encoding is used to gain space, and the classical alternative of sampling plus decompression is used in the practical implementation. We compare now this solution to our proposal, using the implementation obtained from *Pizza&Chili* site[3] and setting one absolute sample every 128 values.

We took TREC-2 collection CR, of about 47 MB, generated its Ψ array, and measured the time to compute $\Psi^i(x)$, for $1 \leq i < n$, where x is the suffix array position pointing to the first text character. This simulates extracting the whole text by means of function Ψ without having the text at hand.

As the differences are strictly positive, we represent in our method the differences minus 1 (so access to $\Psi[i]$ is solved via $sum(i) + i$). This time we use $b = 0$ for the first level of our structure, and other b values for the rest. This seemingly curious choice lets us spend one bit (in B_1, as A_1 is empty) to represent all the areas of Ψ where the differences are 1. This is known to be the case on large areas of Ψ for compressible texts [17], and is also a good reason for Sadakane to have chosen γ-codes. We set one absolute sample every 128 values for our *sum*. Apart from the usual *rank* version that uses 37.5% of space over the bitmaps, we tried a slower one that uses just 5% [8].

Table 2 shows the results. We measure space as a fraction of the size of the original text, and time as nanoseconds per *sum*, as this is necessary to obtain the original Ψ values from the differential version. We only show some examples of fixed b, and how using different b values per level can achieve better results.

This time our technique does not improve upon Sadakane's representation, which is carefully designed for this specific problem and known to be one of the best implementations [5]. Nevertheless, it is remarkable that we get rather close (e.g., same space and 27% slower, or 15% worse in space and time) with a general and elegant technique. It is also a good opportunity to illustrate the flexibility of our technique, which lets us use different b values per level.

[3] Mirrors `http://pizzachili.dcc.uchile.cl` and `http://pizzachili.di.unipi.it`.

6 Conclusions

We have introduced a data reordering technique that, when applied to a particular class of variable-length codes, enables easy and direct access to any codeword, bypassing the heavyweight methods used in current schemes. This is an important achievement because the need of random access to variable-length codes is ubiquitous in many sorts of applications.

We have shown experimentally that our technique competes successfully, in several immediate applications. We have also compared our proposal with the best solutions for sparse bitmaps [18], but we have omitted it in Section 5 due to space limitations: Except for the *search* operation, we achieved better space and time results when the distribution of the gaps was skewed, and comparable performance otherwise (uniform distribution).

We have used the same b for every level, or manually chose it at each level to fit our applications. This could be refined and generalized to use the best b at each level, in terms of optimizing compression. The optimization problem can be easily solved by dynamic programming in just $O(n \log S)$ time.

References

1. Brisaboa, N., Fariña, A., Navarro, G., Paramá, J.: Lightweight natural language text compression. Information Retrieval 10, 1–33 (2007)
2. Clark, D.: Compact Pat Trees. PhD thesis, University of Waterloo, Canada (1996)
3. 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)
4. Culpepper, J., Moffat, A.: Compact set representation for information retrieval. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 137–148. Springer, Heidelberg (2007)
5. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM JEA 13, article 12, 30 pages (2009)
6. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. In: Proc. 18th SODA, pp. 690–696 (2007)
7. Fischer, J., Mäkinen, V., Navarro, G.: An(other) entropy-bounded compressed suffix tree. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 152–165. Springer, Heidelberg (2008)
8. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. 4th WEA, pp. 27–38 (2005)
9. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. 32nd STOC, pp. 397–406 (2000)
10. Gupta, A., Hon, W.-K., Shah, R., Vitter, J.: Compressed dictionaries: Space measures, data sets, and experiments. In: Proc. 5th WEA, pp. 158–169 (2006)
11. Huffman, D.: A method for the construction of minimum-redundancy codes. Proceedings of the I.R.E. 40(9), 1090–1101 (1952)
12. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
13. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms (TALG) 4(3), 1–38 (2008)

14. Moffat, A.: Word-based text compression. Software Practice and Experience 19(2), 185–198 (1989)
15. Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM TOIS 18(2), 113–139 (2000)
16. Munro, I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
17. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
18. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. 9th ALENEX (2007)
19. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA, pp. 233–242 (2002)
20. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003)
21. Solomon, D.: Variable-length codes for data compression. Springer, Heidelberg (2007)
22. Williams, H.E., Zobel, J.: Compressing integers for fast file access. COMPJ: The Computer Journal 42(3), 193–201 (1999)
23. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes, 2nd edn. Morgan Kaufmann Publishers, New York (1999)

Identifying the Intent of a User Query Using Support Vector Machines

Marcelo Mendoza[1] and Juan Zamora[2]

[1] Yahoo! Research Latin America, Chile
[2] Applied Computational Intelligence Lab (INCA), Department of Informatics, Universidad Técnica Federico Santa María, Chile

Abstract. In this paper we introduce a high-precision query classification method to identify the intent of a user query given that it has been seen in the past based on informational, navigational, and transactional categorization. We propose using three vector representations of queries which, using support vector machines, allow past queries to be classified by user's intents. The queries have been represented as vectors using two factors drawn from click-through data: the time users take to review the documents they select and the popularity (quantity of preferences) of the selected documents. Experimental results show that time is the factor that yields higher precision in classification. The experiments shown in this work illustrate that the proposed classifiers can effectively identify the intent of past queries with high-precision.

1 Introduction

With advances in technology and world wide access to Internet, user needs have gone beyond simple informational needs. The web, growing in size and complexity, offers more resources and services that make it more difficult for search engines to find precise results for their users. In this sense, the attention of the web community has focused on identifying the intent of a user query. With this, the goal is for search engines to use different ranking functions depending on the type of intent detected.

To develop a coherent framework that points to the intent of a user query, also known as user goals, through the queries they make, several authors have risen to the challenge of defining web search taxonomies. In a first approximation, Broder [3], and later Rose and Levinson [15], have consolidated widely-accepted taxonomies of intents of user queries. These advances have concluded that the intents of user queries are related to the type of interaction or use the users wish to have with the selected resource.

Later, works based on these taxonomies have taken on the problem of constructing query classifiers. Using different information sources, among them, text [11,1], click-through data [14], or combinations of both sources [12], have shown results with diverse outcomes.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 131–142, 2009.
© Springer-Verlag Berlin Heidelberg 2009

1.1 Contributions

In this work we propose three vector representations of queries based on text and click-through data that allow for query classification by intent. We will consider the taxonomy proposed by Broder (informational, navigational, transactional) and for the classification process we will use techniques based on support vector machines (SVMs) that learn to classify past queries. We will show that the proposed methods achieve high-precision results for all the categories considered.

1.2 Outline

The rest of this work is organized as follows: In section 2 we review related work. In section 3 the vector representations of the queries presented in this work are introduced and the characteristics of the classification technique used are described in detail. In section 4 the experimental design is presented, along with analysis of the results obtained. Finally, conclusions are drawn and future work outlined in Section 5.

2 Related Work

When a user formulates a query in a search engine, their objective is not just to find information about a certain topic, but also to find a site or interact in some way with the suggested results (e.g. read a document, download files, purchase a book, among other interactions). Considering this, Broder [3] proposed a taxonomy of web searches consisting of three categories: *informational*, *navigational* and *transactional*. Broder understood that informational queries were those where the user's intents was to find information related to a specific topic. Navigational queries were those where the user's intents was to find a certain site. Finally, transactional queries were those where the user's intents was to complete some type of transaction on a website, that is, download a resource, or make a purchase, among others. Through an experiment based on the opinion of experts, a set of queries were classified using the proposed categories. As a result, the navigational, informational, and transactional categories were distributed in 20%, 50% and 30%, respectively. Later this taxonomy was extended by Rose and Levinson [15], who developed a framework for classification of search goals and illustrated how to use this framework to manually classify web queries from a search engine.

Once the categories were consolidated by Broder and subsequently by Rose and Levinson, the design and implementation of methods for the automatic classification of queries according to their intents became an important part of many investigations. Kang and Kim [11] proposed the construction of classifiers that characterized queries according to the distribution of query terms. To achieve this, over a set of queries classified by experts, they obtained two collections of terms frequently used in writing informational and navigational queries. By measuring mutual information between the two collections and features such as the distance between the terms in a query and the terms in the titles and

snippets of the selected documents, they were able to determine if a query is for general use or if it is informational or navigational. Despite the 80% precision rate this classifier reached, the results can not be considered conclusive because of the small volume of data used (only 200 queries were extracted and labeled from the TREC collection).

Lee *et al.* [12] enumerated bias levels in click distributions as classification features. Intuitively, an informational query should have more clicks concentrated in lower-ranked items, as opposed to navigational queries which are expected to have more clicks in the highest-ranking positions and, in general, they have only one click if they are successful. Using a set of queries classified by a group of experts, they evaluated the precision of a classifier based on these features and obtained a 54% precision for 50 queries manually labeled.

A similar focus was used by Liu *et al.* [14] to build a query classifier. They proposed two features based on click-through data that allowed query characterization: nRS (number of query sessions that register clicks before a position n in the ranking data) and nCS (number of query sessions registering less than n clicks). Using a decision tree as a classifier, their precision rate achieved was almost 80% for a set of 400 manually labeled queries.

Baeza-Yates *et al.* [1] proposed to analyze three categories: informational, equivalent to the category defined by Broder, non-informational, which considered Broder's navigational and transactional categories; and an ambiguous category that included queries whose intention was difficult to perceive based solely on the query terms, such as polysemic queries. Through an experiment based on expert opinion, a set of queries were classified using the proposed categories. As a result, the informational, non-informational, and ambiguous categories were distributed in 61%, 21% and 18% respectively. Later they establish the intent of a user query by analyzing the relationship between queries and 16 categories from the Open Web Directory (ODP). Using techniques such as Support Vector Machines (SVMs) and Probabilistic Latent Semantic Analysis (PLSA), they reached 60% precision (approximately) on a dataset of 6,000 queries semi-automatically classified into the Broder categories (the vector representations were clustered and then those clusters were labeled).

Recently, Jansen *et al.* [10], using a classifier based on query features drawn from logs, such as query terms, IP numbers, and length of the query, achieved a precision of 74% using a training data set of 400 queries classified by experts.

3 The Classifiers

In this section we introduce three vector representations of queries, based on a combination of two information sources: text and click-through data. The idea is to be able to represent queries in the vector space formed by the collection of query terms and the descriptive terms of the documents selected in the query sessions. The vector representations consider variables extracted from click-through data as influential factors for each term. Our idea is to model queries by means of the variables that influence user preferences. In order to do this we will consider only

the terms that the user read before their selections, considering also the time spent reading the selected documents and the clicks that the documents register in the query sessions. These variables are combined differently to see their effect on the precision of the classifiers. Finally, in this section we will detail the characteristics most relevant to the classification technique used.

3.1 Vector Representation Based on Descriptive Text and Clicks

Following Wen *et al.* [17], a *query session* consist of a query instance and the URLs the user clicked on. Thus:

$$\texttt{querySession} := < \texttt{query}, (\texttt{clickedURL})^* >.$$

In the strict sense of the definition, each query session represents a query instance formulated by an anonymous user in a defined point of time. As an extension to this idea, we will represent queries by means of the set of query sessions where the query was formulated.

The list of results shown by a search engine to a user describes each recommended page / site with the following three text components: the page / site title, the URL and the *snippet* or extract of the document content, often the header or dynamic summary. If at least one of the three components is related to the meaning of the query, this will be selected. According to this idea, we propose using a vector representation of queries based on a variation of the $\texttt{Tf} - \texttt{Idf}$ schema, where the vocabulary will be generated by the terms in the titles, URLs and snippets of the selected documents.

Given a query q, we use S_q to denote the set of sessions in which q has been formulated. Let D_S be the set of documents selected in S_q. The influence of each descriptive term on the vector representation of q will be proportional to the number of occurrences of that term in each document d of D_S (factor $\texttt{Tf}$). It will also be proportional to the fraction of clicks of each document d of D_S calculated over the clicks registered in S_q (factor $\texttt{Pop}_{d,q}$). Based on these facts, the component associated with the i-th vocabulary term in the vector representation of a query q will be given by:

$$q_{[i]} = \sum_{d \in D_S} \texttt{Pop}_{d,q} \cdot \frac{\texttt{Tf}_{i,d}}{\max_l \texttt{Tf}_{l,d}}, \tag{1}$$

where the second quotient is the factor $\texttt{Tf}$ normalized by the maximum frequency calculated for all the terms mentioned in the descriptive text of d, and $\texttt{Pop}_{d,q}$ is the fraction of clicks to d in the clicks registered in S_q. According to this vector representation, a term would have greater influence for q to the degree that the term has a greater number of occurrences in the descriptive text of the document (factor $\texttt{Tf}$) and the document registers more preferences in the sessions of q (factor $\texttt{Pop}_{d,q}$). That is, $\texttt{Pop}_{d,q}$ plays the role of *Idf* in the well-known $\texttt{Tf} - \texttt{Idf}$ weighting scheme for the vector model. Finally, the component $q_{[i]}$ is calculated considering all the documents selected in the sessions of q where the term was used.

3.2 Vector Representation Based on Descriptive Text and Reading Time

Claypool *et al.* [5], in the context of general web browsing, show how implicit interest measures are related to the interests of users. Comparing data about implicit interest indicators and explicit judgments of Web pages visited, they found that the time spent on a page has a strong correlation with explicit interest. In the same sense, Fox *et al.* [8] showed that the time spent on the search result page and the click-through data are the best predictors of user's satisfaction. Moreover, they proved that combinations of these implicit relevance feedback measures are useful to predict the quality of search results.

From the above, we will introduce a new vector representation of queries that combines both descriptive text and the time spent reading each selected document. In this section we will combine only descriptive terms and reading time. In the following section we will use these variables with the $\mathtt{Pop}_{d,q}$ factor introduced in Equation 1. The same as in the vector representation introduced previously, the vocabulary we will represent in the queries will be formed by all the terms that make up the page titles, URLs, and *snippets*.

Given a query q and the set of sessions S_q in which q has been formulated; D represents the collection of documents selected and registered in the log; N_D represents the size of D; and D_S is the set of documents selected in S_q. Let Q be the set of queries formulated and registered in the log and N_Q be the size of Q. With t_d we refer to the average reading time spent on document d calculated over S_q sessions. t_S represents the total duration of all the sessions in S_q. The $q_{[i]}$ component associated with the i-th term in the vocabulary of the vector representation of q will be given by:

$$q_{[i]} = \left(0.5 + 0.5\,\frac{\mathtt{Tf}_{i,q}}{\max_l\ \mathtt{Tf}_{l,q}}\right) \times \log\frac{N_Q}{n_{i,Q}} + \sum_{d\,\in\,D_S} \frac{\mathtt{Tf}_{i,d}}{\max_k\ \mathtt{Tf}_{k,d}} \times \frac{t_d}{t_S} \times \log\frac{N_D}{n_{i,D}}, \tag{2}$$

where $\mathtt{Tf}_{i,q}$ and $\mathtt{Tf}_{i,d}$ represent the number of occurrences of the term in query q and in document d, respectively, and $n_{i,Q}$ and $n_{i,D}$ represent the number of queries and the number of documents in which the term appears, respectively.

The first part of Equation 2 corresponds to the modified schema `Tf-Idf` for queries introduced by Salton and Buckley [16], which allow us to incorporate query terms such as descriptive text. The second part represents the effect of the page's descriptive text on reading time. Intuitively, a term will have greater influence on the vector representation of q as the term has more occurrences in the descriptive text of the selected document (factor `Tf`) and the user has invested more time in reading it (factor $\frac{t_d}{t_S}$). As the time spent in each query differs by query type (for example, the time spent viewing the answers of a closed-class question like "Barack Obama's 47th birthday" is less than a general information query like "History of United States"), we normalize t_d using t_S, calculating the time factor as the fraction of time spent in d over the time spent in the sessions of q.

Finally, in Equation 2, the inverse frequency of the term in the collection D (factor $\texttt{Idf}$) has been considered to give more or less relevance to the terms with greater or lesser frequency in the set. The influence $q_{[i]}$ is calculated considering all the documents selected in the sessions of q where the term is used.

3.3 Vector Representation Based on Descriptive Text, Reading Time and Clicks

This query representation corresponds to a combination of the factors considered in Equations 1 and 2. For this representation we have also considered the vocabulary formed by the page's descriptive text (titles, URLs and snippets). Regarding this set of terms, we have considered the variables of reading time and clicks. The same as in the representation of Equation 2, the descriptive text models the attraction effect that triggers the selection. The incorporation of factor $\texttt{Pop}_{d,q}$ aims to give greater relevance to the terms used in documents that have been selected in other sessions. Also, the reading time variable is still considered to give more influence to the terms used in documents that have attracted more user attention.

According to the above, the component $q_{[i]}$ associated to the i-th term in the vector representation of q is given by:

$$q_{[i]} = \left(0.5 + 0.5\,\frac{\texttt{Tf}_{i,q}}{\max_l\ \texttt{Tf}_{l,q}}\right) \times \log \frac{N_Q}{n_{i,Q}} + \sum_{d \in D_S} \frac{\texttt{Tf}_{i,d}}{\max_k\ \texttt{Tf}_{k,d}} \times \frac{t_d}{t_S} \times \texttt{Pop}_{d,q} \times \log \frac{N_D}{n_{i,D}}, \quad (3)$$

The second sum in the expression represents the influence of the term according to the attractiveness of the descriptive text in the document selections. According to this representation, a term will have greater influence on the vector representation of q in so far as it has more occurrences in the descriptive text of d, the average reading time of d is significant with respect to t_S, and d registers an important amount of preferences in the sessions of q.

3.4 Classification Technique

Since the vector representations in Equations 1, 2, and 3 are calculated based on large term collections, it is necessary to use a classification technique that behaves well with high-dimensional data. In this context the support vector machines (SVMs) [6] have proven to be useful in processing high-dimensional vectors (over 10^6 features) even when the vectors are sparse, such as in the case of text [2]. Leopold and Kindermann [13] showed that SVMs even perform well in categorizing documents without pre-selecting features (*pre-filtering*), which means they compare favorably to other techniques like Decision Trees (e.g. C4.5), Artificial Neural Networks (ANN) or Bayesian Networks. Due to the fact that we are using high dimensional term vectors we will use SVMs.

As has been argued in previous works [13,2,1], it is recommendable to use radial basis function as a kernel for the setup of the SVM function, since this has shown good performance in text categorization.

4 Experimental Results

4.1 Data Set

For this paper we have processed a commercial search engine log. The file corresponds to a period of 3 months from the year 2006 and contains 594,564 queries associated to 765,292 sessions. The log also contains 1,124,664 clicks on 374,349 different URLs. To construct the query vector representations based on text, the queries and descriptive text have been processed, eliminating accent marks, punctuation, and the top-50 stopwords. Using the log file we also estimate the time spent in each document visit, calculating the time gap between consecutive selections in the same query session. For the last click, we estimate the reading time as the average time spent in the query session.

Experts from our labs have manually classified a set of 2,000 queries considering the categories proposed by Broder. The queries considered are the top-2,000 most frequent queries of the query log analyzed. They are associated to 126,287 sessions. The experts had to respond to questions focused on identifying the intent of a user query, similar to those formulated in Broder's experiments. The questions asked were able to identify if the intent was to find a particular site or topic, if the desired results should be found in one website or in many, and finally, if the goal of the query was to read the results obtained or to interact in some other way with the resource, allowing later categorization in the taxonomies considered in this experiment.

The results obtained have been used in the following way: the definitive set of queries considered in the experiments is formed by those that have been classified in the same categories by all the experts. That is, those whose intent has been determined by consensus. Those queries that were classified in two or more categories were revised again by the experts in a second review. As a result of the classification process, 1,953 queries were labeled by consensus, distributed in the different categories as follows: the informational, navigational and transactional categories were distributed in 52%, 33% and 15%, respectively.

70% (1,367 queries) of the manually classified queries were considered as training data, leaving the remaining 30% for evaluation (586 queries). This partition was calculated using a simple random sample in each category in order to preserve the original distribution determined by the experts.

4.2 SVM Tuning

The SVM implementation called LIBSVM, developed by Chang & Lin [4], which is freely available, was used. The version of SVM used is that proposed by Cortes and Vapnik [6] known as C-SVM since the associated optimization problem is parameterized by a penalty factor C, $C > 0$. Since we will use radial basis functions ($K(x, y) = e^{-\gamma \|x-y\|^2}, \gamma > 0$) where the variables (x, y) represent labeled instances of the data (queries in our case), a second parameter is added to the problem, γ, which models the length of the radial basis.

The parameter tuning process proposed by Hsu *et al.* [9] was followed. This process consists in exhaustive cross-validation testing that yields the best (C, γ)

pair. As the query vector representations introduced in this paper are variants of the well known $\texttt{Tf} - \texttt{Idf}$ model, we will compare our performance results considering the $\texttt{Tf} - \texttt{Idf}$ query vector representation as a baseline. Table 1 shows the values found for each proposed query representation and for the baseline.

Table 1. SVM tuning results for the Broder's taxonomy

Method	c	γ
(0) `tf-idf`	2048	3.0517578e-05
(1) `tf-pop`	8192	3.0517578e-05
(2) `tf-idf-time`	2048	3.0517578e-05
(3) `tf-idf-pop-time`	2048	3.0517578e-05

4.3 Performance Evaluation

In a first analysis and in order to evaluate the overall performance of the classifiers, we compare the nominal and predicted categories for each query, tabulating error rates per category (the proportion of errors over the whole set of instances). These results are shown in Table 2.

Table 2. Overall performance for the classifiers (error rates). Bold fonts indicate the best error rate for each category.

Method	Inf.	Nav.	Tran.
`tf-idf`	19.8%	19.28%	**2.9%**
`tf-pop`	25.8%	12.8%	55.1%
`tf-idf-time`	**8.9%**	1.6%	59.5%
`tf-idf-pop-time`	24%	**0.5%**	28.6%

As we can see in Table 2, for the navigational category the best performance is reached by the $\texttt{tf} - \texttt{idf} - \texttt{pop} - \texttt{time}$ method. It seems that for the identification of transactional queries the text is the most useful information source, being the $\texttt{tf} - \texttt{idf}$ method the one which reaches the best results. We can observe also in Table 2 that the vector query representation defined from Equation 2 achieves the best result for the informational category. In general, the worst result is obtained in the transactional category of the Broder's taxonomy and the best result is obtained in the navigational category.

We expect that the third method outperforms the other methods for all the categories considered in the experiments because it considers all the factors (popularity, reading time and text) but this is not true for the informational category. We intend to illustrate with one example why the second method outperforms the third method in the informational category. The query "buyer of ingersoll rand compressed air dryer" was manually classified in the informational category. The predicted categories for the methods 2 and 3 were informational and transactional. The error of the third method suggests that the use of the $\texttt{Pop}_{d,q}$

variable in the vector query representation from Equation 3 introduces noise in the weight of the term "buyer", which matchs with instances classified in the transactional category. When we see the whole evaluation data set, we can observe that this kind of error is very frequent for the third method. The confusion between transactional and informational categories represents an error of 19.5% obtained by this classifier in the informational category (over the 90% of the error rate obtained by the third method in this category). On the other hand, this kind of error achieves only the 3% for the second method.

Following the analysis and in order to count the classification costs we consider the four possible cases presented when comparing the nominal class with the predicted class: true positives (*tp*), false positives (*fp*), false negatives (*fn*), and true negatives (*tn*). We calculate the following measures for each classifier of the evaluation set: Precision ($\frac{tp}{tp+fp}$), FP rate ($\frac{fp}{fp+tn}$), TP rate or Recall ($\frac{tp}{tp+fn}$), and F-measure ($\frac{\mathbf{2}\times \text{Precision}\times \text{Recall}}{\text{Precision}+\text{Recall}}$). In the case of the F-measure we have specifically considered the version F_1 equivalent to the harmonic average of the Precision and Recall measures given that it establishes a compromise between both criteria (it only has high values if both measures are high).

Given that this decision problem is multiclass, we will use the generalized version of the analysis based on two classes. Following Fawcett [7], we will consider each category as a reference class and will conduct the evaluation comparing it to the other classes. Let C be the set of all the classes considered in the decision problem and let c_i be the reference class upon which we will conduct the evaluation. Let P_i and N_i be the positive and negative classes of the performance analysis based on two classes. Thus for every $c_i \in C$ we will consider:

$$P_i = c_i,$$

$$N_i = \bigcup_{j \neq i} c_j \in C.$$

Then, for each reference class $c_i \in C$, we calculate the measures based on the analysis of both classes using the new classes P_i and N_i. This way, each classifier can be analyzed as a binary classifier according to its performance for the reference class. The results of this analysis can be revised in Table 3.

Table 3 shows the results obtained for the pairs of reference / other classes comparison, considering all the categories of the analyzed taxonomy. Using this we obtain the Informational / Other classes, Navigational / Other Classes, Transactional / Other classes for the Broder's taxonomy. The measures have been calculated for each classifier, with the baseline being identified as $\texttt{tf}-\texttt{idf}$, with the one from Equation 1 being identified as $\texttt{tf}-\texttt{pop}$, the one from Equation 2 as $\texttt{tf}-\texttt{idf}-\texttt{time}$ and that from Equation 3 as $\texttt{tf}-\texttt{idf}-\texttt{pop}-\texttt{time}$. As we can see in Table 3, the baseline reaches very good rates for false positives for the informational and transactional categories. On the other hand, the baseline reaches the worst FP-rate for the navigational category. Regarding the predictive capacity of the baseline, it reaches low TP Rates for the informational category being more competitive in the case of navigational queries. The classifier with

Table 3. Performance evaluation of the proposed classifiers. Bold fonts indicate the best result for each evaluation.

Method	Measures			
	TP Rate	FP Rate	Precision	F-Measure
Informational - Other				
(0) `tf-idf`	0.6538	**0.0292**	**0.9623**	0.7786
(1) `tf-pop`	0.58576	0.13281	0.84186	0.69084
(2) `tf-idf-time`	**0.92614**	0.05141	0.89071	**0.90808**
(3) `tf-idf-pop-time`	0.65000	0.23711	0.31138	0.42105
Navigational - Other				
(0) `tf-idf`	0.9655	0.2597	0.6109	0.7483
(1) `tf-pop`	0.92131	0.13475	**0.88088**	0.90064
(2) `tf-idf-time`	**0.99485**	0.06870	0.87727	**0.93237**
(3) `tf-idf-pop-time`	0.45455	**0.01603**	0.83333	0.58824
Transactional - Other				
(0) `tf-idf`	0.91	**0.0165**	0.9192	0.9146
(1) `tf-pop`	0.75692	0.05344	**0.94615**	0.84103
(2) `tf-idf-time`	**0.98438**	0.05316	0.90000	**0.94030**
(3) `tf-idf-pop-time`	0.70000	0.13153	0.41880	0.52406

the best performance in terms of TP Rate was $\texttt{tf} - \texttt{idf} - \texttt{time}$, which is the one that obtained the best proportion of positives over the total. The previous measure indicates that the classifier based on the $\texttt{tf} - \texttt{idf} - \texttt{time}$ representation has the most predictive capacity. It should be noted that the classifier $\texttt{tf} - \texttt{pop}$ obtains a precision greater than $\texttt{tf} - \texttt{idf} - \texttt{time}$ for the Navigational and Transactional reference classes, but in the global analysis poorer performance is registered because the values for TP Rate are considerably less that those reached by $\texttt{tf} - \texttt{idf} - \texttt{time}$ in these categories. Evaluating the Precision / Recall tradeoff, and considering the measurement F_1, the classifier based on the $\texttt{tf} - \texttt{idf} - \texttt{time}$ representation also obtained the best performance.

5 Conclusion

In this paper we have explored the use of query classifiers according to the intent of a user query. For this, we have proposed three vector representations for queries based on click-through data and descriptive text, identifying four relevant factors: frequency of terms, (`Tf`), inverse frequency in documents (`Idf`), user preferences (`Pop`), and reading time of selected documents (`Time`). Using SVMs we have evaluated the performance of the three representations over a set of queries categorized by experts.

The experimental results show that the third method reaches good results when we consider overall performance measures such as error rates. When we consider the cost of making wrong decisions incorporating to the analysis false positives and false negatives, the second method outperforms the other proposed methods. The most relevant reason that explains this performance is its ability

to identify informational and navigational queries, which represent a significant proportion of the whole data set. Finally, when we considerate costs into the analysis, the poor result obtained for the transactional category considering the plain error rate was minimized.

One of the most relevant characteristics of the second method is its high predictive / discriminative capacity. Furthermore, the quality of their results depends on the intent that their are identifying. Our approach has an advantange in this sense because we reach higher precision results for all the categories considered in the experiments. Among the factors that explain the success of the proposal we emphasize the incorporation of new factors drawn from click-through data such as `time`, which was considered for the first time in this problem.

Our classifiers consider only queries that appears in the query-log. We are working on extensions to deal with new queries submitted by users. In order to do this, we have to address the problem of determining distance functions that achieve good results measuring distances between queries with partial information (e.g. query terms) and our query vector representations. If this is possible, we could represent a new query with a close query registered in the query-log file.

Another problem is to conduct an analysis that will allow us to determine why the combination of factors $\texttt{tf} - \texttt{idf} - \texttt{time}$ outperforms $\texttt{tf} - \texttt{idf} - \texttt{pop} - \texttt{time}$. This work showed that the `Pop` factor allows for greater precision of the classifiers for the Navigational and Transactional classes and that the factor `Time` achieves the same but for the Informational class. This suggests that the user preferences are a relevant source of information for navigational and transactional queries, and that the time factor is more relevant for the informational class. This is intuitive in the sense that the reading time allows us to identify resources that capture attention based on content, usually pages, as opposed to clicks which would tend to concentrate more on sites than pages, making them more related to transactional and navigational queries.

Acknowledgments

Mr. Zamora was supported by a fellowship for scientific initiation of the Graduate School of the UTFSM, Chile. We would like to thank the contributions of Ricardo Baeza-Yates, Hector Allende, Hector Allende-Cid, Libertad Tansini and Katharine Sherwin.

References

1. Baeza-Yates, R., Calderón-Benavides, L., González-Caro, C.: 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. Basu, A., Watters, C.R., Shepherd, M.A.: Support vector machines for text categorization. In: HICSS 2003: Proceedings of the 36th Hawaii International Conference on System Sciences, Hawaii, USA, January 6- 9, p. 7. IEEE Computer Press, Los Alamitos (2003)

3. Broder, A.: A taxonomy of web search. SIGIR Forum 36(2), 3–10 (2002)
4. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines (2001), http://www.csie.ntu.edu.tw/~cjlin/libsvm
5. Claypool, M., Brown, D., Le, P., Waseda, M.: Inferring user interest. IEEE Internet Computing 5(6), 32–39 (2001)
6. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)
7. Fawcett, T.: An introduction to ROC analysis. Pattern Recognition Letters 27(8), 861–874 (2006)
8. Fox, S., Karnawat, K., Mydland, M., Dumais, S., White, T.: Evaluating implicit measures to improve web search. ACM Transactions on Information Systems 23(2), 147–168 (2005)
9. Hsu, C.W., Chang, C.C., Lin, C.J.: A practical guide to support vector classification. Department of Computer Science and Information Engineering, National Taiwan University (2003)
10. Jansen, B., Booth, D., Spink, A.: Determining the informational, navigational and transactional intent of Web queries. Information Processing and Management 44(3), 1251–1266 (2008)
11. Kang, I.-H., Kim, G.: Query type classification for web document retrieval. In: SIGIR 2003: Proceedings of the 26th International ACM SIGIR Conference, Toronto, Canada, July 28 - August 1, pp. 64–71. ACM, New York (2003)
12. Lee, U., Liu, Z., Cho, J.: Automatic identification of user goals in web search. In: WWW 2005: Proceedings of the 14th International Conference on World Wide Web, Chiba, Japan, May 10-14, 2005, pp. 391–400. ACM Press, New York (2005)
13. Leopold, E., Kindermann, J.: Text categorization with support vector machines. how to represent texts in input space? Machine Learning 46(1-3), 423–444 (2002)
14. 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)
15. Rose, D.E., Levinson, D.: Understanding user goals in web search. In: WWW 2004: Proceedings of the 13th International Conference on World Wide Web, May 17-20, pp. 13–19. ACM Press, New York (2004)
16. Salton, G., Buckley, C.: Term-weighting approaches in automatic retrieval. Information Processing and Management 24(5), 513–523 (1988)
17. Wen, J., Nie, J., Zhang, H.: Clustering user queries of a search engine. In: WWW 2001: Proc. of the 10th Int. Conf. on World Wide Web, Hong Kong, May 1-5, 2001, pp. 162–168. ACM Press, New York (2001)

Syntactic Query Models for Restatement Retrieval

Niranjan Balasubramanian and James Allan

Center for Intelligent Information Retrieval
University of Massachusetts Amherst, Amherst MA 01003
{niranjan,allan}@cs.umass.edu

Abstract. We consider the problem of retrieving sentence level restatements. Formally, we define restatements as sentences that contain all or some subset of information present in a query sentence. Identifying restatements is useful for several applications such as multi-document summarization, document provenance, text reuse and novelty detection. Spurious partial matches and term dependence become important issues for restatement retrieval in these settings. To address these issues, we focus on query models that capture relative term importance and sequential term dependence. In this paper, we build query models using syntactic information such as subject-verb-objects and phrases. Our experimental results on two different collections show that syntactic query models are consistently more effective than purely statistical alternatives.

1 Introduction

We describe and evaluate the task of finding restatements – sentences that match a query sentence, either in part or entirely. That is, starting from some sentence as a query, we define other sentences as relevant if they describe some or all of the same information units. Identifying sentences that contain overlapping information is a key challenge for several language applications such as tracking text reuse, summarization and novelty detection. Tracking information flow [14] and local text reuse [18] studied in the context of information provenance and plagiarism detection, focus on identifying sentence level information overlap. Also, extractive summarization techniques and novelty-based ranking often measure redundancy across sentences to avoid repeating information [1],[17],[23].

Cast as a sentence retrieval problem, the main challenge for restatement retrieval is that the query sentences are long: they usually contain multiple units of information (clauses), each of which could effectively be a query on its own. To effectively handle long queries, retrieval techniques must be able to identify key components of a query sentence [10],[11] and avoid spurious partial matches – accidentally flagging a match because a sentence includes partial information from different clauses. For document retrieval using keyword queries, the popular query likelihood (QL) model utilizes the frequency of query terms in the document, the length of the document and the collection frequency of the query terms. However, for sentence retrieval using long queries, the frequency of query

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 143–155, 2009.
© Springer-Verlag Berlin Heidelberg 2009

terms in a sentence or in the collection does not provide adequate information to discriminate between relevant and spurious partial matches. Table 1 shows examples of spurious and relevant partial matches. Figure 1 illustrates the partial matches problem for query likelihood (QL) retrieval. At the top of the ranked list (the left of the figure) we have complete restatements – sentences which include all information units in the query and which tend to be identical to the query. However, as we go down the ranked list of sentences (to the right), we can see that there are several cases where the non-relevant sentences match more query terms than the relevant sentences.

Table 1. Partial Matches and Term Dependence: Square brackets enclose noun phrases with modifiers. Italicized words in the partial matches indicate query term matches.

Query Sentence
[Jordanian security officials] on Sunday, announced the arrest of an *[Iraqi woman]*, closely linked to the *[terrorist leader Abu Musab al-Zarqawi]* as a *[fourth bomber]* in the *[Amman hotel attacks]* and they broadcast a *[taped confession]* showing her wearing a *[translucent suicide explosive belt]*, packed with *[ball bearings]* and describing how she had tried unsuccessfully to blow herself up.
Spurious partial match
Abu Musab al-Zarqawi briefly survived the *bomb attack* that killed him Wednesday night, a military spokesman said Friday, *describing* the al-Qaida leader turning away and mumbling when American troops approached the stretcher that he had been placed on by *Iraqi* police officers.
Relevant partial match
While a videotaped *confession showing* Rishawi *wearing* the disarmed *suicide belt* was being *broadcast* around the world, details about her life, motivation and role in the *attacks* that killed 57 people began to emerge in *Jordan* and *Iraq*.

Also, term independence assumptions can add to the partial matches problem. Consider the query shown in Table 1. Under term independence assumptions, non-relevant sentences that match many words in the pharases *Jordanian security officials* or *the terrorist leader, Abu Musab Al-Zarqawi* can receive higher query likelihood scores compared to relevant sentences which only match parts of these phrases. Addressing sequential term dependence – i.e., finding runs of adjacent terms that should be treated as a unit – can also help capture entities and concepts, moving us closer to semantic matching without actually trying to handle the semantics.

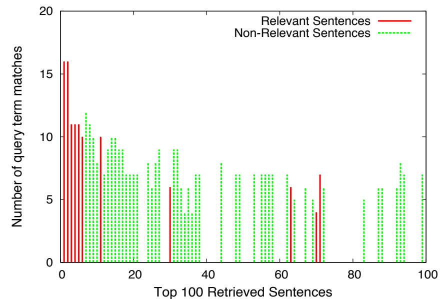


Fig. 1. Number of term matches for relevant and non-relevant sentences for an example query. Term matches for unjudged sentences in top 100 ranks are not shown.

In practice, spurious partial matches can lead to a large number of false alarms for identifying local text reuse. In extractive summarization and novelty detection, related but different events and news stories often use overlapping vocabulary inducing spurious partial matches that affect coverage of new information. In this paper, we investigate the partial term matches and term dependence problems in the restatement retrieval setting. In a sentence retrieval framework, we show that query models that emphasize the relative importance of query terms and capture sequential term dependence can improve retrieval effectiveness for our task. We compare syntactic query models to purely statistical alternatives and show that the syntactic models are more effective for restatement retrieval.

2 Related Work

The TREC Novelty Detection track focused on retrieval of novel on-topic sentences. The novelty detection techniques cannot be readily inverted to retrieve restatements; the typically poor precision of novelty detection [20], will lead to poor recall in retrieving restatements, especially the partial restatements. On the other hand, improvement in redundant information detection can help novelty detection by improving precision. Also, systems that used syntactic and semantic features for novelty detection improved precision. In a similar fashion, we believe that using syntactic features will help improve restatement retrieval.

Textual entailment is defined as the task of determining whether a given sentence entails an hypothesis [7]. Successful entailment systems often extract syntactic and semantic features from sentences and candidate hypotheses and utilize machine learning techniques to verify entailment [8]. While this task is similar to finding restatements, we focus on finding explicit restatements which are a subset of sentences that can entail a hypothesis. Moreover, the query sentences that we are interested in often contain multiple units of information unlike the sentences used for the entailment tasks.

In the context of document retrieval, Allan et al [2] studied the effect of extracting key components of TREC style queries and more recently, Bendersky et al [4] showed the utility of identifying key concepts in verbose description queries using a supervised learning approach to detect concepts – noun phrases – and weight them. Similarly, we believe the use of syntactic information will yield more effective query models for sentence retrieval.

For sentence matching, Metzler et al [14] showed that simple query likelihood (QL) model outperformed other word overlap based techniques such as TF-IDF for identifying sentences at various levels of similarity. Murdock [16] proposed translation based models for identifying sentence level similarity. Balasubramanian et al [3] showed that advanced language modeling techniques such as dependence models and relevance models can be combined to provide significant improvements over query likelihood baselines for finding redundant information. For sentence retrieval, Cai et al [5] used parse tree based features of candidate sentences to model term dependence in keyword queries for retrieving topically

related sentences. In addition to using syntactic dependencies, we also use the subject-verb-object information to build query models and avoid parsing candidate sentences.

3 Query Models

Statistical approaches such as relevance models (RM) [12] and sequential dependence models (SDM) [15], can be used to model term importance and term dependence, respectively. For sentence retrieval, Balasubramanian et al [3] used relevance models based on term frequencies in source documents and showed that a combination model (DMRM) using dependence model queries to perform initial retrieval and then building relevance model queries provides additional improvements over either method.

However, there are some known issues with these purely statistical approaches. First, relevance models add new words to the query and rely upon a good quality initial retrieval. Instead of relying on an initial retrieval to build query models, syntactic analysis can be used to directly estimate the relative importance of terms. Furthermore, a better query model will improve the quality of the initial results and in-turn improve the quality of the relevance models. Second, term dependencies often span multiple terms and for long queries a full dependence model does not scale. Also, the simple sequential term dependence model is a brute force enumeration of all possible pairs of adjacent query terms. For keyword queries in document retrieval, this indiscriminate enumeration does not result in too many spurious matches. However, for long query sentences, it can cause spurious matches adding to the partial matches problem. Using syntactic dependencies we can avoid the problem of brute force enumeration of spurious dependencies.

The query sentences that we consider are well formed grammatical sentences that are amenable to automatic natural language analysis such as parsing techniques [6, 13, 22]. One approach to finding restatements would be to rank candidate sentences by measuring the alignment of their parse trees with that of the query sentence. However, this approach involves parsing entire text collections and performing computationally intensive alignment and is less robust since parsing can sometimes fail on candidate sentences. We propose to parse the query sentences alone to build effective query models for sentence retrieval. Thus, we leverage the benefits of syntactic information through parsing and the robustness of a retrieval framework to improve effectiveness of sentence matching in an efficient manner.

4 Syntactic Query Models

We use syntactic information to build query models for query likelihood retrieval. Specifically, we propose three syntactic query models to a] emphasize the relative importance of terms and b] capture the syntactic dependencies between query terms. We also build a combination model that leverages the benefits of

these two models. Given a query sentence, we first parse the sentence using the Stanford Dependence parser [13] to obtain the syntactic dependencies. Then, we create *extended noun phrases* by grouping noun phrases, their adjectival modifiers and basic dependencies. We experimented with other types of modifiers and dependencies but found the adjectival modifiers to be most useful.

SVO Weighting (SW)

Given a query sentence Q, we identify the *extended noun phrases* and words $\{c_1, c_2, \cdots, c_m\} \in Q$, and extract the set of subjects and objects, the verbs and the remaining phrases. Then, we formulate the query weights as follows:

$$\forall c \in Q, w(c) = \begin{cases} \delta(|c|) + \lambda_{ph} & \text{if c} \in \text{Phrases(Q)} \\ \delta(|c|) + \lambda_{so} & \text{if c} \in \text{Subj-Objs(Q)} \\ \lambda_v & \text{if c} \in \text{Verbs(Q)} \\ 1.0 & \text{otherwise} \end{cases} \quad (1)$$

where $|c|$ is the number of words in unit c. For our experiments we chose a δ to be a simple increasing function, $\delta(n) = 0.1n$. Finally, the weighted Indri query [21] is composed as follows:

$$SW(Q) = \#weight(w(c_1)\ c_1\ w(c_2)\ c_2\ \cdots\ w(c_m)\ c_m) \quad (2)$$

Initially, we found that even a fixed choice of values greater than 1 for the λs provides improvements. However, learning the parameters from a corpus allows us to tune the weights on the single word verbs in relation to the subjects, objects and other phrases which often span multiple words.

Syntactic Phrase Matching (SD)

We use the syntactic dependencies obtained by parsing to effectively model sequential term dependencies of varying lengths. We create term dependence queries by modifying the sequential dependence model query described by Metzler et al [15]. Given the *extended noun phrases* extracted from the query sentence, $\{c_1, c_2, \cdots, c_m\}$ where $c_i = \{q^i_1, q^i_2, \cdots\}$, we construct an unordered window query for each extended noun phrase in the query and interpolate it with the original query sentence as follows:

$$SD(Q) = \beta(\text{Q}) \cup (1 - \beta) \left\{ \begin{array}{l} \#uw\delta_1(c_1) \\ \#uw\delta_2(c_2) \\ \cdots \\ \#uw\delta_m(c_m) \end{array} \right\} \quad (3)$$

where $\delta_i = |c_i| + 2$. $\#uw\delta(c)$ represents an un-ordered window query which estimates the likelihood of the words in the phrase c, occurring within a window of length δ. Figure 3 shows example Indri queries for the syntactic query models.

Syntactic Sub-Queries (SSQ)

We can extend the term dependence to larger units of information contained in the query sentence. We analyzed the parse trees of some query sentences and

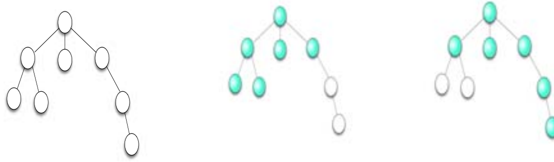


Fig. 2. Sub-queries: Tree on the left is an example parse of a query sentence. Colored nodes in the middle and right trees indicate words in the two possible sub-queries.

Query

The International Atomic Energy Agency and its chief Mohamed ElBaradei on Friday won the Nobel Peace Prize for 2005 for their work in stopping the spread of nuclear weapons.

SW

```
#weight(
0.90 #combine(
international atomic energy agency chief
mohamed elbaradei friday won nobel peace prize
2005 work stopping spread nuclear weapons)
0.10 #weight(
  1.70 #weight( 0.85 weapons  0.85 nuclear )
  1.00 #combine(spread)
  1.00 #combine(work)
  1.50 #combine(won)
  1.80 #weight( 0.6 elbaradei  0.6 chief  0.6 mohamed )
  2.65 #weight( 0.66 agency  0.66 energy  0.66 atomic
                0.66 international )
  1.00 #combine(friday)
  1.50 #combine(stopping)
  2.55 #weight( 0.85 prize  0.85 peace  0.85 nobel )
  1.00 #combine(2005)
 )
)
```

SD

```
#weight(

0.90 #combine(
international atomic energy agency chief
mohamed elbaradei friday won nobel peace prize
2005 work stopping spread nuclear weapons)

0.10 #combine(
  #uw4( weapons  nuclear )
  #uw5( elbaradei  chief  mohamed )
  #uw5( prize  peace  nobel )
  #uw6( agency  energy  atomic  international )
  spread work won friday stopping 2005
)
)
```

SSQ

```
#weight(
0.9 #combine( international atomic energy agency chief
mohamed elbaradei
friday won nobel peace prize 2005 work stopping spread
nuclear weapons)

0.033 #combine(stopping agency energy atomic international
prize peace nobel won spread weapons nuclear)
0.033 #combine(stopping agency energy atomic international
prize peace nobel won elbaradei chief mohamed friday)
0.033 #combine(stopping agency energy atomic international
prize peace nobel won work 2005)

)
```

Fig. 3. Example Syntactic Query Models

observed that the root and the first level of the parse tree often contained information central to the query sentence. The different units seemed to correspond to the different sub-trees of the nodes in the first level. Therefore, for each query we generate a list of sub-queries one for each node in first level of the parse tree as follows: Extract nodes from the node's sub-tree and add them to the core of the query (the root and all the nodes in the first level). Figure 2 shows the sub-queries generated by our heuristic from a parse.

Combination Models (SWD and SWD-RM)

We experimented with various combinations of the query models. A linear interpolation of SW and SD models turned out to be the most useful combination and we refer to this model as *Syntactic Combination* (SWD). We also created a combination of the *Syntactic Combination* and the relevance model queries (SWD-RM) by using *Syntactic Combination* for initial retrieval and adding the *Syntactic Combination* query to the relevance model query, similar to the combination of dependence and relevance models (DMRM) [3]. This combination will demonstrate the utility of combining syntactic and purely statistical methods.

5 Experiments

To evaluate restatement retrieval we constructed a test bed from english newswire documents. First, we extracted 50 query sentences from the collection. The query

sentences were chosen such that a) they could be potential responses to some actual query and b) many query sentences contained multiple units of information. Next, two annotators were asked to construct their own queries and find as many restatements as possible for each query sentence. Then, the restatements found in the first stage were used as surrogate queries and the results of a query likelihood sentence retrieval on the original query and the surrogate queries were used to create a pool of sentences for judging. Finally, we removed *easy* queries with QL MAP of 1.0 for the partial restatements category and *hard* queries that have no relevant judgments for the complete restatements category. Table 2 summarizes the details of the resulting collection (*Restatements Collection*). We also report results on a test collection consisting of 49 queries created for identifying redundant information [3](*Redundant Information Collection*).

Table 2. Collection Details

Collection	Documents	Sentences	Queries	Judged sentences	Restatements
English Newswire	79210	2,849,683	35	4340	647

We compare our syntactic approaches to statistical alternatives for building query models including the query likelihood baseline (QL) and the state-of-art document retrieval models – relevance models (RM) and sequential dependence models (DM). We also compare our combination model (SWD-RM) to the combination of relevance models and dependence models (DMRM) [3]. To train parameters for the query models, we performed grid search using cross-validation. For the Restatements collection we performed 5-fold cross validation and for the Redundant information collection we used 7-fold cross validation. Table 3 shows the training parameters for the various methods. DM and *Syntactic Phrase Matching* were tuned to optimize precision@10 to gain better generalization performance over MAP.

Table 3. Training parameters: a) λ - JM smoothing parameter b) β - Original query interpolation weight c) λ_{ph} - Phrase weight d) λ_{so} - Subject/Object weight e) λ_v - Verb weight f) γ_{svo} - SVO Query weight g) γ_{ph} - Syntactic Phrase Query weight h) fbt, fbd: Feedback terms and documents. i) w - interpolation weight.

QL	DM	RM	DMRM	SW	SD	SWD	SWD-RM
λ	λ, β	λ, β fbt, fbd	DM, RM + w	$\lambda, \beta, \lambda_{sv}, \lambda_v, \lambda_{ph}$	λ, β	$\lambda, \beta, \gamma_{svo}, \gamma_{ph}$	SW,SD+w

6 Results

6.1 Restatements collection

Table 4 shows results for the statistical and syntactic query models. The syntactic query models are consistently better than the statistical alternatives except for the *Syntactic Sub Queries* .

Table 4. 5-fold Cross Validation results on Restatements collection - Sig entries indicate significant improvements over the corresponding methods using paired t-test. (q:QL, r:RM, d:DM, dr:DMRM, sw:SW,sd:SD). Underlines indicate the best score.

Method	**P@5**	**P@10**	**P@20**	**MAP**	**Sig. (in MAP)**
QL	0.7086	0.5686	0.4514	0.6373	
DM	0.7086	0.5743	0.4429	0.6407	
RM	0.7086	0.5657	0.4443	0.6374	
DMRM	0.7200	0.5857	0.4500	0.6509	q,r
SW	0.7371	0.5971	0.4614	0.6660	q
SD	0.7200	0.5771	0.4471	0.6564	q
SWD	<u>0.7543</u>	0.6000	<u>0.4629</u>	0.6732	q,d,r
SSQ	0.7086	0.5714	0.4500	0.6358	
SWD-RM	0.7486	<u>0.6057</u>	0.4586	<u>0.6788</u>	q,d,r,dr, SW, SD

Statistical Query Models. In contrast to their known value for document retrieval, RM and DM do not provide any significant improvements in MAP over the query likelihood baseline. We observed that RM roughly helped half the queries and was detrimental to the other half. In addition to failures due to poor retrieval, even when baseline retrieval effectiveness was high, relevance models can add words from sentences that are only partial restatements of the original query sentence. This actually worsens the partial matches problem and lowers the overall effectiveness. For the DM, the poorly performing queries often overemphasize the match of a long noun phrase. For example, in the query fragment, *Iranian President Mahmoud Ahmadinejad*, the corresponding long noun phrase match *iran's president, mahmoud ahmadinejad* is amplified in the dependence model query by having multiple ordered window matches and a larger number of unordered window matches. The combination of dependence and relevance models, DMRM, provides good improvements over either model by leveraging their individual strengths [3]. Overall, the purely statistical approaches do not yield substantial improvements for restatement retrieval.

***SVO Weighting* and *Syntactic Phrase Matching*.** Intuitively, *SVO Weighting* weights the key components of the query sentence and balances the relative weights of these components. For query sentences with long noun phrases, the presence of the noun phrases alone can significantly boost the likelihood score of candidate sentences. Tuning the weights on phrases and verbs together provides a way to moderate the impact of matching long noun phrases alone and serves to improve the importance of verbs in candidate sentences. Table 5 displays a non-relevant sentence that matches the long noun phrase in the query sentence and some relevant sentences that only contain a portion of the long noun phrase but contain all the other key components of the query sentence. The difference in ranking according to dependence models and the *SVO Weighting* demonstrates the strength of the syntactic model in balancing the relative query term weights.

Syntactic Phrase Matching emphasizes the dependence between terms in the query sentence. By grouping noun phrases and their modifiers in an unordered

Table 5. Example illustrating benefits of SW compared to DM. Italicized words indicate term matches. DM and SW entries indicate ranks in corresponding methods.

Query: Iranian President Mahmoud Ahmadinejad called the holocaust a "myth"

DM	SW	Rel.	Text
21	38	NR	tel aviv, israel – with *iran's president, mahmoud ahmadinejad, call*ing for israel to be "wiped off the map," israeli officials have special reasons for concern now that iran has defied the west and said it will resume enriching uranium.
22	14	R	charlotte knobloch, president of the central council of jews in germany, noted that *ahmadinejad* has *call*ed the *holocaust* a "*myth*".
88	18	R	*ahmadinejad* produced outrage in the west last year when he threatened israel and *call*ed the holocaust a "*myth*".
97	19	R	*ahmadinejad* has generated international scorn by dismissing the *holocaust* as a *myth* and *call*ing for the destruction of israel.

window query it favors phrase matches thereby capturing entities and other important concepts in the query. Also, the syntactic grouping ensures that only valid dependencies are captured and spurious dependencies that are possible in a brute force enumeration are avoided. Compared to brute force DM queries, *Syntactic Phrase Matching* queries are more effective at modeling sequential term dependence and are also more efficient.

Syntactic Sub Queries. *Syntactic Sub Queries* did not provide any improvements over QL. This is mainly due to sub-queries that add only one or two unimportant words to the top level of the tree thus creating bad sub-queries that cause spurious matches. Furthermore, for some queries the top portion of the parse tree is not central to the query as assumed by our heuristic. However, we believe that by systematically capturing larger syntactic structures such as clauses, we can create more meaningful sub-queries.

***Syntactic Combination* Models.** The combination model SWD provides small improvements over either model used in the combination. However, the combination now consistently outperforms the DM and RM baselines. Similar to the combination method DMRM, the *SVO Weighting* and *Syntactic Phrase Matching* provide different types of evidences for relevance and their combination provides significant improvements. The SWD-RM provides minor improvements over SWD but it outperforms DMRM, *SVO Weighting* and *Syntactic Phrase Matching* thus showing that syntactic and statistical query models can be combined to obtain additional improvements. Finally, we note that the small improvements due to the combinations add up to an absolute 4 point improvement in MAP over the QL baseline.

6.2 Redundant Information

Table 6 shows results for this collection. The statistical query models follow the trends observed in [3]. Both *SVO Weighting* and *Syntactic Phrase Matching*

outperform the QL baseline and DM significantly and the combination models SWD and SWD-RM significantly outperform all the statistical models except for DMRM. The results clearly confirm the trends we observed in our collection but the actual improvements are smaller. We believe that this is due to the differences in the types of queries in the collections. In comparison to Redundant information collection , the query sentences in Restatements collection are more complex, in terms of average length (17 versus 21 average words per query) and number of SVO elements. Overall, we expect the syntactic query models to benefit complex query sentences more. Although we did not observe any solid trends in terms of complexity versus the benefit of the syntactic methods, we observed that the benefits of using syntactic query models were limited in another noisy collection with simple queries.

To summarize, we see that utilizing syntactic information to build query models consistently outperforms purely statistical methods for term weighting and the brute force sequential dependence models.

Table 6. Redundant information - 5-fold Cross Validation results on 49 queries. Sig column entries indicate significant improvements in MAP over the corresponding methods (q:QL, r:RM, d:DM, dr:DMRM, sw:SW,sd:SD) using paired t-test.

Method	P@5	P@10	P@20	MAP	Sig.
QL	0.6122	0.5041	0.3500	0.5207	
DM	0.6204	0.4878	0.3510	0.5253	
RM	0.6245	0.5082	0.3500	0.5308	q,d
DMRM	0.6122	0.5041	0.3561	0.5279	
SW	0.6204	0.5143	0.3643	0.5338	q,d
SD	0.6327	0.5082	0.3622	0.5376	q,d
SWD	0.6380	0.5082	0.3622	0.5380	q,d,SW
SWD-RM	0.6286	0.5020	0.3643	0.5387	q,d,r,SW

7 Syntactic Features for Discriminative Training

We also conducted experiments on the Restatements collection to investigate the utility of syntactic features in a discriminative setting for sentence retrieval. Nallapati et al devised some language modeling based features for a discriminative classifier to rank documents. We adopted the same set of text based features (T) and investigated the use of parser based features such as, presence of subject, verb, object and distance based measures on the parse tree (P). Our initial results (see Table 7) shows that ranking and binary SVM [9] using text only features (T) perform comparably to that of the traditional retrieval approaches. More importantly, we see that adding the parser based features improves the performance of both ranking and binary SVMs similar to the improvements we observed with the syntactic query models. These results clearly show that there is potential for investigating syntactic features in a discriminative setting. We

Table 7. SVM based learning for restatement retrieval on Restatements collection

Method	P@5	P@10	P@20	MAP
RankSVM (T)	0.6971	0.5829	0.4514	0.6349
BinarySVM (T)	0.6914	0.5686	0.4443	0.6304
BinarySVM (P)	0.5143	0.3771	0.2774	0.3966
RankSVM (P + T)	0.7086	0.5857	0.4571	0.6520
BinarySVM (P+ T)	0.7314	0.5829	0.4586	0.6528

plan to extend our work to directly incorporate the syntactic dependencies and model more complex relationships amongst the features for ranking.

8 Conclusions

In this paper, we described the problem of partial term matches and term dependence for a general restatement retrieval task. We showed that query models that address the partial matches and sequential term dependencies, provide consistent gains in effectiveness. We find that syntactic query models are consistently more effective than purely statistical alternatives. Avoiding spurious partial matches is a key challenge for several natural language applications and syntactic query models provide an effective and efficient query dependent solution. Using syntactic query models, we leverage the benefits of both syntactic information as well as the robustness and efficiency of a retrieval framework by avoiding pitfalls due to parser failures and inaccuracies. Natural language applications often use syntactic and semantic features in a machine learning framework. Our initial experiments with a discriminative approach for retrieving restatements also shows promise for integration in such settings.

For long queries, indiscriminately expanding all query words can hurt effectiveness. We believe that syntactic information can lead to some selective expansion techniques that leverage the term importance and dependence in the query. Sentence simplification [19] techniques use syntactic information to break query and candidate sentences into simpler sentences. Another extension to our work is to consider sentence simplification for matching query and candidate sentences. Also, we intend to explore the effect of modeling the term dependencies more directly in a machine learning framework for restatement retrieval.

Acknowledgements

This work was supported in part by the Center for Intelligent Information Retrieval, in part by the Defense Advanced Research Projects Agency (DARPA) under contract number HR0011-06-C-0023, and in part by UpToDate. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsor.

References

1. Allan, J., Wade, C., Bolivar, A.: Retrieval and novelty detection at the sentence level. In: Proceedings of the 26th annual international ACM SIGIR conference, pp. 314–321 (2003)
2. Allan, J., Callan, J., Croft, B., Ballesteros, L., Broglio, J., Xu, J., Shu, H.: Inquery at TREC-5. In: Fifth Text REtrieval Conference (TREC-5), pp. 119–132 (1997)
3. Balasubramanian, N., Allan, J., Croft, W.B.: A comparison of sentence retrieval techniques. In: SIGIR 2007: Proceedings of the 30th annual international ACM SIGIR conference, pp. 813–814. ACM Press, New York (2007)
4. Bendersky, M., Croft, W.B.: Discovering Key Concepts in Verbose Queries. In: Proceedings of ACM SIGIR 2008 conference, pp. 491–498 (2008)
5. Cai, K., Chen, C., Liu, K., Bu, J., Huang, P.: MRF based approach for sentence retrieval. In: SIGIR 2007: Proceedings of the 30th annual international ACM SIGIR conference, pp. 795–796. ACM Press, New York (2007)
6. Charniak, E.: A maximum-entropy-inspired parser. In: ACM International Conference Proceeding Series, vol. 4, pp. 132–139 (2000)
7. Dagan, I., Glickman, O., Magnini, B.: The PASCAL Recognising Textual Entailment Challenge. In: Quiñonero-Candela, J., Dagan, I., Magnini, B., d'Alché-Buc, F. (eds.) MLCW 2005. LNCS (LNAI), vol. 3944, pp. 177–190. Springer, Heidelberg (2006)
8. Giampiccolo, D., Magnini, B., Dagan, I., Dolan, B.: The Third PASCAL Recognizing Textual Entailment Challenge. In: Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing, pp. 1–9 (2007)
9. Joachims, T.: Making large scale SVM learning practical. Universität Dortmund (1999)
10. Kumaran, G., Allan, J.: A Case For Shorter Queries, and Helping Users Create Them. In: HLT 2007: NAACL Proceedings of the Main Conference, pp. 220–227 (2007)
11. Kumaran, G., Allan, J.: Effective and Efficient User Interaction for Long Queries. In: SIGIR 2008: Proceedings of the 31st annual international ACM SIGIR conference, pp. 11–18 (2008)
12. Lavrenko, V. and Croft, W.B.: Relevance based language models. In: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 120–127 (2001).
13. de Marneffe, M.C., MacCartney, B., Manning, C.D.: Generating typed dependency parses from phrase structure parses
14. Metzler, D., Bernstein, Y., Bruce Croft, W., Moffat, A., Zobel, J.: Similarity measures for tracking information flow. In: CIKM 2005: Proceedings of the 14th ACM international conference on Information and knowledge management, pp. 517–524. ACM, New York (2005)
15. Metzler, D., Croft, W.B.: A Markov random field model for term dependencies. In: SIGIR 2005: Proceedings of the 28th annual international ACM SIGIR conference, pp. 472–479 (2005)
16. Murdock, V.: Aspects of Sentence Retrieval. PhD Thesis. University of Massachussets Amherst (2006)
17. Radev, D., Jing, H., Stys, M., Tam, D.: Centroid-based summarization of multiple documents. In: Information Processing and Management, pp. 919–938 (2004)
18. Seo, J., Croft, W.B.: Local text reuse detection. In: Proceedings of the 31st annual international ACM SIGIR conference, pp. 571–578 (2008)

19. Siddharthan, A., Nenkova, A., McKeown, K.: Syntactic simplification for improving content selection in multi-document summarization. In: Proceedings of the COLING 2004, pp. 896–902 (2004)
20. Soboroff, I., Harman, D.: Novelty Detection: The TREC Experience. In: HLT/EMNLP, pp. 105–112 (2005)
21. Strohman, T., Metzler, D., Turtle, H., Croft, W.B.: Indri: A language model-based search engine for complex queries. In: Proceedings of the International conference on Intelligence Analysis (2004)
22. Taskar, B., Klein, D., Collins, M., Koller, D., Manning, C.: Max-margin parsing. In: Proc. EMNLP 2004, pp. 1–8 (2004)
23. Wang, D., Li, T., Zhu, S., Ding, C.: Multi-document summarization via sentence-level semantic analysis and symmetric matrix factorization. In: Proc. ACM SIGIR Conference, pp. 307–314 (2008)

Use of Co-occurrences for Temporal Expressions Annotation

Olga Craveiro[1,2], Joaquim Macedo[3], and Henrique Madeira[2]

[1] School of Technology and Management, Polytechnic Institute of Leiria, Portugal
[2] CISUC, Department of Informatics Engineering, University of Coimbra, Portugal
{marine,henrique}@dei.uc.pt
[3] Department of Informatics, University of Minho, Portugal
macedo@di.uminho.pt

Abstract. The annotation or extraction of temporal information from text documents is becoming increasingly important in many natural language processing applications such as text summarization, information retrieval, question answering, etc.. This paper presents an original method for easy recognition of temporal expressions in text documents. The method creates semantically classified temporal patterns, using word co-occurrences obtained from training corpora and a pre-defined seed keywords set, derived from the used language temporal references. A participation on a Portuguese named entity evaluation contest showed promising effectiveness and efficiency results. This approach can be adapted to recognize other type of expressions or languages, within other contexts, by defining the suitable word sets and training corpora.

1 Introduction

The Web is actually a key information source for our daily lives. Search engines are essential to use efficiently the information available at the Web. Therefore, there is an intensive academic and industrial research effort to improve the efficiency and effectiveness of underlying Web Information Retrieval (IR) models.

Temporal information is a key piece on most information system applications and, consequently, in Web based applications. Nevertheless, it has not been the focus of a systematic and deep work on IR applications. The temporal dimension is an important element of the user's information need context, and if used effectively it would improve the relevance of documents response set. The most effective temporal entities recognition programs on free (or semi-structured) text are heavily dependent on the natural language used in those texts. The typical approaches are based on intensive hand-crafted rules. Another set of solutions are based on natural language independent stochastic models which assigns probabilities to strings in a given language L allowed by the use of a training corpus. Between these two approaches there are a variety of mixed ones. The stochastic approaches are best suitable (and simpler) for multilingual context such as the Web. Additionally, another important requirement for huge Web applications is the efficiency, which can be achieved by improving simplicity of the used models.

In most applications, unigrams, bigrams or trigrams are used due to its simplicity and because they are hard to beat by more complex n-grams. However, the nature of

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 156–164, 2009.
© Springer-Verlag Berlin Heidelberg 2009

natural languages is such that many words combinations are infrequent and can even do not appear in a given training corpus. This point to the need for smoothing techniques to overcome zero probability strings on maximum likelihood estimation.

This work proposes a method for annotating temporal information to be included in a temporal aware Web IR model. The experimental scenario used is a Portuguese text collection but we believe that the proposed approach can be easily adapted to other languages. This method uses simple probabilistic based techniques to recognize temporal entities. Temporal entities are detected using temporal expression patterns derived from the higher probability temporal reference word co-occurrence from training corpora. Less frequent and unseen expressions are ignored. The main advantage of the proposed approach is the simplicity and efficiency improving, preserving a promising effectiveness.

The structure of the paper is as follows: section 2 presents the related work on temporal entities recognition, section 3 details the proposed model, section 4 discusses the results obtained from experimental evaluation and section 5 concludes the paper.

2 Related Work

Although a plethora of works exists for the area of temporal references extraction in English texts, to the best of our knowledge, none of them creates automatically a set of expression patterns and applies it for temporal entities recognition. Expression patterns matching require sentence-by-sentence processing. However, Natural Language Processing systems are mainly based on term-by-term processing, using term linguistic characteristics for its identification, such as techniques presented in [1]. An annotation scheme to represent dates and time, based on a variety of hand-crafted and machine-discovered rules, was proposed in [2]. This approach uses finite-state automata, a common technique in this area. A very different approach was proposed in [3]: the temporal expressions identification in French documents is based on a context-scanning strategy (CSS).

Unlike the English language, Portuguese text language extraction area has not been much explored. In particular, temporal information has not been the focus of any systematic work reported in the literature. PALAVRAS, for instance, is an automatic grammar and lexicon-based parser for unrestricted Portuguese text [4]. This system is an important tool for Portuguese text annotation, even though using a generic approach to handle temporal expressions. More recently, a temporal processor called XTM (XIP Temporal Module) was developed by Hagège and Tannier [5, 6] supporting Portuguese language processing, among others languages, such as, English and French. XTM is rule-based, relying on a word-by-word processing.

The novelty of our proposal relies on having lexical patterns automatically generated from Portuguese texts and follows an inductive empirical approach which starts from the data to the knowledge, unlike the work reviewed above.

3 Annotating Temporal Information

In this section, we present our approach for the recognition of temporal entities. Despite other possible applications for entities recognizing in other contexts and languages, Portuguese language is the focus of our experiments.

The method relies on a set of temporal patterns, based on regular expressions, used to identify and classify the temporal expressions found in Portuguese texts. The patterns are created using words co-occurrence, determined from a set of seed keywords, which are Portuguese temporal references. Our method is based on a two-stage approach, each stage being carried out by a different module: the first stage is executed by the *Co-occurrence processor* module (henceforth COP) and the second stage is carried out by the *Annotator* module. The modules work as follows. Firstly, the COP module creates the temporal patterns, based on the training corpora and on the set of reference words which are divided in two sets: lexical markers and grammatical markers. Then, these patterns are used by the Annotator module to perform the annotation of the Portuguese temporal expressions. Fig. 1 shows a diagram of the model architecture and module interconnection.

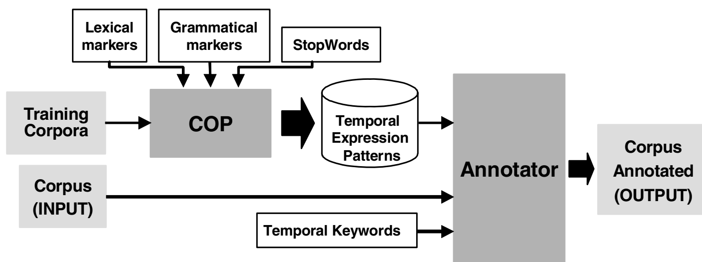


Fig. 1. Model architecture and module interconnection

3.1 Annotation Scheme

The temporal expressions are annotated accordingly as the temporal guidelines defined by the organization and the participants of the Second HAREM[1] [7]. The classification of temporal expressions defined in these guidelines was supported by the annotation scheme TimeML [8]. The annotation comprises a unique identification, a category which is TIME, a type (*calendar_ref*, *duration* or *frequency*) and a subtype only for the type CALENDAR_REF (*date*, *time* or *interval*). A detailed specification of the annotation scheme can be found in [7]. Some examples are presented below. The first sentence exemplifies a date expression and the second sentence represents a temporal expression which expresses a repetition in the time.

(1) I was in Berlin <EM ID="1" CATEG="**TIME**" TYPE="**CALENDAR_REF**" SUBTYPE="**DATE**">in 2008</EM>.
(2) I visit my parents<EM ID="2" CATEG="**TIME**" TYPE="**FREQUENCY**">every day</EM>.

3.2 Co-occurrence Processor

The task of the COP module is to create a set of temporal patterns that will be used by the Annotator module. COP can be easily executed over various corpora, yielding a

[1] Second evaluation contest of Named Entities Recognizer system, in Portuguese language document collections.

considerable number of patterns that enrich the annotation stage. It is worth noting that the COP module is only needed to get the set of patterns and once the patterns are established the COP module is not used anymore. Nevertheless, the patterns can be fine tuned later on to improve the identification of temporal expressions.

The COP module analyzes the input training corpora, determines the words combination and its frequency, and uses a statistical approach to decide which patterns must be created according to the co-occurrences found. COP module has several execution steps. The first step creates a list composed by the temporal expressions found and their frequency. These expressions were found using the lexical markers. These markers must be composed by all Portuguese words from which temporal expressions can be composed (e.g. months, seasons, weekdays, units of temporal measure like *day*, *week*, *month*, *year*, …). This set of words is used to detect their co-occurrences which are present in a maximum of *n* words before and/or *n* words after. An example of temporal expressions using the Portuguese temporal word *ano* (*year*) and *n=2*: *"No **ano** passado"* (*In the last year*), *"No próximo **ano** de 2010"* (*In the next year 2010*).

In the second step the list of expressions is pruned. Specifically, the expressions which do not make semantically sense in a language context are removed from the list, using the grammatical markers. However, the expressions that just contain lexical markers and grammatical markers are kept in the list, as long as no stopword exists in neighborhood. For example, in the sentence *"A rua 1 de Maio"* (*The 1st May street*) the expression *"1st May"* is not a temporal expression because it is the name of a street. As the word "*street*" is a stopword, it is excluded.

The next step aggregates temporal expressions found in the previous step according to the following rules. First, the temporal expressions are aggregated if they contain a date or time references. For example, *"Em Abril"* (*in April*) and *"Em Maio"* (*in May*) are aggregated in a single expression with a special tag *"Em tag_MONTH"* (*In tag_MONTH*). Second and last one, the temporal expressions are aggregated if they contain more than one co-occurrence with the same temporal word at the same position. For example, the expressions *"No **ano** passado"* (*In the last year*) and *"No **ano** seguinte"* (*In the following year*) are aggregated in *"No **ano** passado | seguinte"* (*In the last | following year*). The frequency of the aggregated expressions is the sum of the frequency of each expression. The resulting list is ordered by frequency (greater to less). Some expressions can be excluded by a previously defined minimum frequency threshold.

Finally, the patterns are defined by regular expressions. For each pattern is associated the classification according to the temporal guidelines of the Second HAREM (see section 3.1).

3.3 Annotator

The objective of the Annotator module is to identify and classify Portuguese temporal expressions with the relative annotation written in the original text, through the patterns defined by COP. After the text split into sentences, each of one is processed in five steps. The first step is introduced to improve performance by excluding all the sentences that cannot have a temporal expression. Only sentences with date and time references and/or temporal words from Portuguese language defined in a keyword list (see Fig. 1) are processed. For example, the sentence *'Lisbon is the capital of Portugal'* is not processed. However, the sentence *'**Today** is sunshine'* is processed.

The generation of candidate temporal expressions is done in second step. First, it identifies time expressions and date expressions which can be complete or incomplete dates. Then, these expressions are tagged with a "special tag" such as *tag_DATE*, *tag_MONTH*, *tag_YEAR*, *tag_WEEK*.

In the third step, the method verifies if the sentences match any temporal pattern. In this case, each sentence is annotated with semantic classification corresponding to the matching pattern (fourth step). Finally, the "special tags" are replaced by the original text.

4 Evaluation

Knowing there is a huge amount of documents to process in common application scenarios, one of the key decisions is to achieve the best tradeoff between efficiency and effectiveness in temporal expressions identification. As our option is to favor efficiency to some extent, with the used configuration the system may not find all temporal expressions (even for a trained human reader, it would be difficult to identify all the temporal expressions, as the notion of time is often subtly embedded in the text).

Our primary goal was to evaluate the performance of the method in a restricted environment. Therefore, the COP was configured only to create patterns of simple temporal expressions, expressions composed by only one temporal word or one date or time, and a maximum of *n* words before and/or *n* words after (*n*=*2*). The lexical markers were restricted to: months, seasons, weekdays, holidays (*Natal* (Christmas), *Páscoa* (Easter) and *Carnaval* (Carnival)) and the following words[2]: *década, século, ano, mês, semana, dia, hora, minuto, ontem, anteontem, amanhã, hoje, manhã, noite, tarde*. Furthermore, were included in the temporal patterns a set of limited grammatical markers[3] composed by prepositions {*à(s), de, em, durante, desde, pelas, no, naquele, (n)este, (n)esse*}, ordinal adjectives {*anterior, seguinte, próximo, passado, último*} and *haver* (to have) verb conjugations. Note that in the pruning step, the stopwords were not considered yet.

Using the prototype implementation for our method, we have carried out a set of experiments and participated in the evaluation contest Second HAREM with a promising effectiveness and efficiency for the first results obtained (72% precision and 53% recall).

The experiments were performed in a Personal Computer with 1GB RAM memory and an Intel Core 2 E6600 2.4GHz processor, running with Microsoft Windows XP Professional version 2002 SP 2.

We divided the experiments in two tasks: identification and classification. In the identification task, the goal was to obtain complete temporal expressions, while in the classification task the idea was to assign the type and subtype specification. In order to clarify this, we show below some examples with mistakes, accordingly as the temporal guidelines (see section 3.1). The expression *'1909-1955'* is correctly identified.

---

[2] English version: decade, century, year, month, week, day, hour, minute, yesterday, the day before, tomorrow, today, morning, night, afternoon.

[3] English version: prepositions {in, the, during, for, since, by, (in) this, (in) that}, ordinal adjectives {previous, following, next, past, last}.

However, the classification is wrong, as the subtype must be INTERVAL. The expression *'2009'* is incomplete. The correct identification must be *'in 2009'*, but the classification is right.

```
(1) CATEG="TIME" TYPE="CALENDAR_REF"
    SUBTYPE="DATE">1909-1955</EM>
(2) in CATEG="TIME" TYPE="CALENDAR_REF" SUBTYPE="DATE">2009</EM>
```

For efficiency purposes, we measure the time spent on the Annotator module to identify and classify the temporal expressions in test collection. For effectiveness, we calculate the three usual metrics: precision, recall, and the harmonic mean F (F-measure), using the evaluated collection. The formula used to calculate the classification was defined in [9].

4.1 The Collections

The Second HAREM Collection (2ndHC) was the corpus used in our experiments which texts are structured in different genres, such as journalistic, blog, FAQ, literary, etc., and are written in two Portuguese variants: Portuguese from Portugal and Portuguese from Brazil. The 2ndHC is the test collection that is composed by 1040 documents with 33,712 sentences and 668,817 words. The evaluation test is the Time Gold Collection (TGC), a subset of 2ndHC (30 documents, 622 sentences and 12,992 words) and their documents were manually annotated following time HAREM guidelines [7]. The training collection (TC) was another subset of 2ndHC which is composed by all documents of 2ndHC that do not belong to the TGC. The 2ndHC and TGC collections are available through Linguateca[4] and properly detailed in [9].

4.2 Results

The result of TC processing by COP was 289 patterns which can detect more than 289 different temporal expressions because some of them have more than one combination. Note also that about 17% of these patterns permit the identification of dates and times in different formats.

The execution time of the Annotator module was calculated in two scenarios. Scenario 1 – skipped the first step of the Annotator, but all the sentences are processed by every other steps of this module; Scenario 2 – all steps are executed, therefore, only sentences which we believe could indicate the presence of a temporal reference are processed (see section 3.3). In scenario 2, only 17,525 of 33,712 sentences (52%) proceed to the next step, the processing finishes here to the other sentences and the execution time decreases about 27.5% justified by the missing pattern matching step with the remaining sentences. This way, the performance was improved and the Annotator module processed the test collection with an output rate of about 22KB per second.

The effectiveness results are presented in Table 1. We can observe that the results of the two tasks obtained by our system do not have significant differences, which means that the Annotator module shows the same behavior in the two tasks. So, we

[4] Available at HAREM site http://www.linguateca.pt/HAREM

Table 1. Annotation results: our system versus XIP-L2F/Xerox system

	Identification Task		Classification Task	
	(1) *Our system*	(2) *XIP-L2F/ Xerox system*	(1) *Our system*	(2) *XIP-L2F/ Xerox system*
Precision	84,27%	75,31%	83,05%	73,76%
Recall	64,10%	77,59%	64,23%	75,80%
F-measure	72,82%	76,43%	72,44%	74,77%

can conclude that if this module identifies a given temporal expression, then it will achieve a good success in its subsequent classification. Table 1 also shows the results obtained by the XIP-L2F/Xerox system using the same collections. This system was ranked in the first place in the Second HAREM and its results are presented in [6]. Our approach matches the results of the top system concerning precision, but it shows lower recall. This is mainly due to the restricted set of lexical and grammatical markers used by COP to generate the patterns, which affects recall. However, we believe that we can improve recall by increasing the restricted set used by the COP module. We plan to exploit this in future work.

Although, the COP was configured with $n=2$, which means that the expressions was limited to 5 words, only approximately 12% of TGC expressions have more than two words before and/or after the lexical marker (see Table 2). Furthermore, the $n>4$ only exists about 1% of these expressions. It is our intention to carefully study the variation of the n value, since increasing this parameter makes the COP module more complex.

Table 2. TGC temporal expressions

	# words between temporal word and the expression begin/ending					
	n=2	n=3	n=4	n=5	n=6	n=7
# temporal expressions	205	18	8	1	1	0

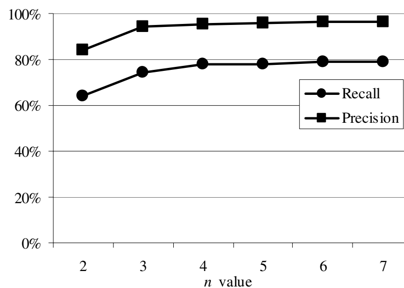


Fig. 2. Precision and Recall values with 2≤ n ≤7

In the precision calculation, the temporal expressions partially correct are not considered. Our system found 178 temporal expressions of which 150 are correctly identified. Indeed, the incorrect expressions are only 3 of 28; the others are incomplete because one or more words are missing in the annotated expression. We analyze the precision variation with *n* ranging from 3 to 7 (see Fig. 2). We observe that the precision improves with *n*, namely when *n* goes from 2 to 3. Improvement is still seen from with $n>3$, but at a lower rate. This means that having COP creating temporal patterns with $n>2$ and one more temporal word improves precision and recall. However, the recall achieved is about 80%. As we said above, the improvement of this metric can be done by increasing the restricted set of markers used by the COP module.

5 Conclusions

The main contribution of this paper is an original method for temporal named entities recognition. The approach creates semantically classified temporal patterns, based on regular expressions, using word co-occurrences obtained from training corpora and a pre-defined seed keywords set, derived from temporal references. The prototype implementation of the proposed method is composed by two modules and some configuration files including temporal reference words and a set of temporal keywords (only used to improve efficiency).

As this temporal named entities recognizer is intended for use in huge Web IR applications, the need for a careful tradeoff between effectiveness and efficiency is the justification for the deliberate simplification of the used approach. Even with a set of limitations and simplifications of a prototype implementation, our method has shown promising results in identification and classification of temporal named entities.

As further work, the most obvious research direction is the variation of used parameters: *n* (number of maximum words on the temporal expression) and low frequency threshold. Additionally, we plan to tune the lexical and grammatical markers and to improve the pruning step resorting to stopwords to lower the rate of false positives. The method can be also evaluated with foreign languages (English, for instance) and another application contexts (other kind of named entities recognition). To do this, a previous study of the chosen language and context, based on a careful statistical analysis, is needed to define the lexical and grammatical markers.

References

1. Mani, I.: Recent developments in temporal information extraction. In: RANLP 2003, Borovets, Bulgaria, pp. 45–60 (2004)
2. Mani, I., Wilson, G.: Robust temporal processing of news. In: ACL 2000: 38th Annual Meeting on Association for Computational Linguistics, Morristown, NJ, USA, p. 69–76 (2000)
3. Vazov, N.: A system for extraction of temporal expressions from French texts based on syntactic and semantic constraints. In: ACL 2001 workshop on temporal and spatial information processing, Toulouse, France (2001)

4. Bick, E.: The Parsing System, PALAVRAS: Automatic Grammatical Analysis of Portuguese in a Constraint Grammar Framework. Ph.D. thesis, Dept. of Linguistics, University of Aarhus, Denmark (2000)
5. Hagège, C., Tannier, X.: XTM: A Robust Temporal Text Processor. In: Gelbukh, A. (ed.) CICLing 2008. LNCS, vol. 4919, pp. 231–240. Springer, Heidelberg (2008)
6. Hagège, C., Baptista, J., Mamede, N.: Reconhecimento de entidades mencionadas com o XIP: Uma colaboração entre a Xerox e o L2F do INESC-ID Lisboa. In: Mota, C., Santos, D. (eds.) Desafios na avaliação conjunta do reconhecimento de entidades mencionadas: O Segundo HAREM. Linguateca (2008)
7. Hagège, C., Baptista, J., Mamede, N.: Apêndice B: Proposta de anotação e normalização de expressões temporais da categoria TEMPO para o HAREM II. In: Mota, C., Santos, D. (eds.) Desafios na avaliação conjunta do reconhecimento de entidades mencionadas: O Segundo HAREM. Linguateca (2008)
8. Pustejovsky, J., Ingria, B., Sauri, R., Castano, J., Littman, J., Gaizauskas, R., Setzer, A., Katz, G., Mani, I.: The Specification Language TimeML. In: Mani, I., Pustejovsky, J., Gaizauskas, R. (eds.) The Language of Time: A Reader. Oxford University Press, Oxford (2005)
9. Mota, C., Santos, D. (eds.): Desafios na avaliação conjunta do reconhecimento de entidades mencionadas: O Segundo HAREM. Linguateca (2008)

On-Demand Associative Cross-Language Information Retrieval

André Pinto Geraldo[1], Viviane P. Moreira[1], and Marcos A. Gonçalves[2]

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{andre.geraldo,viviane}@inf.ufrgs.br
[2] Dep. de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
CEP: 31270-010 – Belo Horizonte – MG – Brazil
mgoncalv@dcc.ufmg.br

Abstract. This paper proposes the use of algorithms for mining association rules as an approach for Cross-Language Information Retrieval. These algorithms have been widely used to analyse market basket data. The idea is to map the problem of finding associations between sales items to the problem of finding term translations over a parallel corpus. The proposal was validated by means of experiments using queries in two distinct languages: Portuguese and Finnish to retrieve documents in English. The results show that the performance of our proposed approach is comparable to the performance of the monolingual baseline and to query translation via machine translation, even though these systems employ more complex Natural Language Processing techniques. The combination between machine translation and our approach yielded the best results, even outperforming the monolingual baseline.

1 Introduction

Cross-Language Information Retrieval (CLIR) is the retrieval of documents in one natural language, based on a query formulated in another natural language, e.g. retrieval of documents written in English based on a set of keywords in Spanish. The need for exploring content in foreign languages has increased with the explosive growth of the Internet. The Web has content in many languages and the distribution of this content by language is very different from the distribution of the Internet access by language. While English is still dominant in terms of content (~66%) [7], the percentage of users that access the Internet in English is less than 30% [21].

In CLIR, terms in the query cannot be matched directly to terms in the documents. Thus, some mapping between languages is needed. It may be necessary to translate the queries, or the documents or even both. Throughout the years, many approaches for mapping terms across languages were proposed. Such approaches may include machine translation systems (MT), dictionaries or thesauri. Dictionaries and thesauri can be automatically constructed using techniques that analyse multilingual corpora to extract the information needed to map concepts between different languages.

All CLIR methods have drawbacks. MT systems typically pick only one translation for each term. Since queries tend to be very short, the MT system may not have

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 165–173, 2009.
© Springer-Verlag Berlin Heidelberg 2009

enough context to enable the choice of the correct sense of the keywords. Machine readable dictionaries (MRDs) are built for human use. Therefore, inflected word forms will most probably be missing from them. Multilingual thesauri are expensive to build; and corpus-based approaches tend to produce translation resources that are domain-specific. It is generally agreed that the combination of a corpus-based approach with MRD or MT yields better results.

The challenge in building a CLIR system based on multilingual corpora is to efficiently analyse large text collections and generate good mappings between concepts across languages. Algorithms for mining association rules (ARs) are widely used to analyse large databases of sales transactions. Their output is a set of implications with measures that reflect the degree of association between the sales items. Our proposal is to map the problem of finding ARs between items in a market-basket scenario to the problem of finding cross-linguistic equivalents between a pair of languages over a parallel corpus. To the best of our knowledge, this is the first work that proposes the use of ARs for CLIR. One of the main advantages of our proposed approach is its simplicity. Yet our results are superior to other co-occurrence based methods. In addition, experimental results show that the performance of our approach is not statistically different from the monolingual baseline. This shows that ARs can be effectively used to map concepts between languages.

2 Related Work

The first research on CLIR was done by Salton [16], who showed that CLIR systems could perform nearly as well as monolingual systems, using a good quality thesaurus.

According to Grefenstette [9], CLIR involves basically three problems: (*i*) knowing how a term expressed in one language might be written in another, i.e., crossing the language barrier; (*ii*) deciding which of the possible translations should be retained. Retaining more than one translation is useful in promoting recall. However, using wrong translations will reduce precision; and (*iii*) deciding how to properly weight the importance of translation alternatives when more than one is retained.

Many approaches have been proposed to solve these problems. These solutions typically use resources such as MT systems, MRD, thesauri or multilingual corpora.

The approach for CLIR we propose is statistical. Other statistical approaches have been previously presented. Nie et al. [12] and Kraaij et al. [10] propose a probabilistic translation model which extracts translation probabilities from parallel corpora mined from the web. Their results are comparable to query translation using Systran.

CLIR systems that achieve the best results do so by combining several techniques, such as good quality translation resources, stemming (or decompounding), more elaborate weighting schemes, query expansion and relevance feedback [3, 17].

3 Association Rules for CLIR

An association rule (AR) is an implication of the form $X \Rightarrow Y$, where $X = \{x_1,x_2,\ldots,x_n\}$, and $Y = \{y_1,y_2,\ldots,y_m\}$ are sets of items. The problem of mining ARs in market-basket data was firstly investigated by Agrawal et al. [1]. In the rule "90% of

customers that purchase bread also purchase milk", the antecedent is bread and the consequent is milk. The number 90% is the confidence factor (*conf*) of the rule, which is calculated according to equation 1. The confidence of the rule can be interpreted as the probability that the items in the consequent will be purchased given that the items in the antecedent are purchased. An AR also has a support level associated to it. The support (*sup*) of a rule refers to how frequently the sets of items $X \cup Y$ occur in the database. Equation 2 shows how the support of an AR is calculated.

$$conf(X \Rightarrow Y) = \frac{n(X \bigcup Y)}{n(X)} \tag{1}$$

$$\sup(X \Rightarrow Y) = \frac{n(X \bigcup Y)}{N} \tag{2}$$

where n is the number of transactions and N is the total number of transactions.

Mining ARs means to generate all rules that have support and confidence greater than predefined thresholds. We used the Apriori Algorithm [2] to mine the ARs.

Our proposal is to map the problem of finding ARs between items in a market-basket scenario to the problem of finding cross-linguistic equivalents on a parallel corpus. A parallel corpus provides documents in a language B that are exact translations of documents in a language A. The approach is based on co-occurrences and works under the assumption that cross-linguistic equivalents would co-occur a significant number of times over a parallel corpus. In this work, the transaction database is replaced by a text collection; the items that the customer buys correspond to the terms in the text; and the shopping transactions are represented by documents.

The proposed approach to use algorithms for mining ARs for CLIR is divided into the following five phases:

(*i*) **Pre-processing:** The inputs for this phase are a collection of parallel documents and the original query in the source language. During this phase, the original text and its equivalent in the other language are initially treated separately. We remove stop-words, apply stemming, and break the documents into sentences. The output of this phase is a set of pre-processed parallel sentences. During this phase, an inverted index containing all stems in the document collection and the list of sentences in which they appear is also built. The inverted index will be used in the next phase to enable selection of the sentences over which the Apriori algorithm is run.

(*ii*) **Mining ARs:** This step consists in generating ARs for the terms in the query. We run the Apriori Algorithm over the pre-processed parallel sentences. In order to speed up rule generation, only sentences that contain the query terms are considered. The output of this phase is a set of ARs for each query term.

(*iii*) **Rule Filtering:** The aim of this step is to keep the rules that most likely map a term to its translation. Rule filtering is based on the following heuristics:

a) Select the AR with the highest confidence. Such a rule will be called M and it has the greatest chance of being the correct mapping.
b) Select the ARs that have confidence of at least 80% of M.
c) Select ARs with confidence equal to ($100 - \text{M} \pm 0.1$). The rationale is that when word in language A is translated to two words in language B, the confidences of the ARs tend to be complementary to 100%.

(iv) **Query Translation:** Each term in the original query is replaced by all possible translations that remain after the filtering process.

(v) **Query Execution:** The last step is to execute the queries in a search engine. At this stage, the CLIR problem has been reduced to monolingual retrieval.

Some preliminary experiments we did for CLEF [6] showed encouraging results. Our approach was ranked amongst the top scoring methods. The test was done using collection of library catalogues in English and the query topics were in Spanish. The experiments described here use a different test collection and different languages for the query topics. In addition, we compare our proposed approach to MT and test the combination of the two. We also provide a deeper analysis of the results.

There are two basic strategies for generating the ARs to create a bilingual lexicon: (i) *eager* – mining rules for all terms in the collection a priori; and (ii) *lazy* – mining rules on demand for query terms only prior to query processing. Our approach mines the ARs on demand, according to a lazy strategy as advocated by Veloso et al. [20]. In their work, the lazy strategy brings improvements in terms of the quality of the rules that are generated. However, in our work the gain is in the number of ARs that are generated, as we only mine rules for the terms in the query. On the other hand, this strategy slightly delays querying. To speed up this process, we could build a cache of ARs. Only words that were not in the lexicon would need mining at query time.

The main advantage of our approach is that it is simpler than other co-occurrence based methods [10, 12, 13, 22] and yet the results are comparable or superior. The method does not require the generation of a term by document matrix, which is costly. The pruning of the itemsets that are below the thresholds for support and confidence, carried out by the Apriori algorithm, allows for efficiency in terms of memory management. It is also simpler than MT systems, which typically need more complex Natural Language Processing (NLP) capabilities.

4 Experiments

The aim of the experiments is to test the feasibility of our proposed approach for using ARs to map concepts between languages. We compare the quality of query translation using ARs to two MT systems and to the monolingual baseline.

4.1 Experimental Setup

We carried out standard ah-hoc IR experiments using the LA Times test collection which is part of the CLEF Test Suite and can be purchased from ELDA [4]. The collection contains 113,046 news articles from 1994. All articles are in English only.

We used the query topics from CLEF 2002. There are 50 queries in total. However, only 42 of those have relevant documents in the collection. Only the “Title” and “Description” fields were used. The original queries had on average 18.92 terms.

In order to show that our proposed approach is language-independent, we have chosen two languages with different morphologies to be used in the queries: Portuguese and Finnish.

The procedure used for the runs in which ARs are employed is the same as described in Section 3. Since our approach needs a sample of parallel documents and the LA Times collection is in English only, the following alternatives were implemented:

- Automatically translating a sample of the collection using Google Translator [8] to generate a synthetic parallel corpus. The sample size was 20% of the collection. One in every five editions of the newspaper was picked for translation into Portuguese and Finnish. We did not consider any relevance information in order to choose which documents to translate, thus not including any bias. The experimental runs that employ this procedure were tagged as **AR-LATimes**.
- Using distinct corpora for deriving the ARs and querying. We automatically translated a sample of the Glasgow Herald collection from English into Portuguese and Finnish to generate a synthetic parallel corpus. This collection contains 56,472 news articles from 1995, of which 11,437 were translated. Although it covers news from a different year and different geographical region, the Glasgow Herald is on the same domain as the corpus used for querying. These runs are tagged as **AR-GH.**
- Using a corpus from a different domain to generate the ARs. The idea is to test the feasibility of using a good quality (i.e. hand translated) corpus to derive the ARs. The corpus used was the EuroParl [5], which contains data extracted from the proceedings of the European Parliament. These runs are tagged as **AR-EuroParl**.

Aiming at having some means for comparison between the AR approach and MT-based approaches, we translated the topics from Finnish and Portuguese into English using Google translator [8] and Systran [19]. Since Systran does not support Finnish, we used it only for Portuguese. These runs are tagged **MT-Google,** and **MT-Systran**.

In order to test whether ARs and MT could be used in conjunction to improve results, we have also tested the combination of the best AR strategy (i.e. AR-LATimes) with the best MT run. The combination was done by performing a set union of the query terms generated from both strategies. These runs are tagged **MT+AR**.

So as to have a basis for comparison that enables us to assess how much is lost when our approach is used, we also executed a monolingual run. This was our baseline and was tagged as **Mono**.

We removed stop-words and used the Porter Stemmer[15, 18]. Zettair [23] was the IR system adopted. The similarity measure was Okapi BM25. The time taken to run all 50 queries is approximately 12 seconds including the mining of the ARs, rule filtering, query translation and processing by the search engine. The time taken varies according to the size of the corpus used to generate the ARs. Terms that had no translations were omitted from the query.

4.2 Results

The results of the experimental runs are summarised in Table 1, which contains the Mean Average Precision (MAP) for each run and two other statistics that are useful for analysis: the number of terms used in the queries and the number of terms that were missing from the parallel collection.

One way to evaluate a CLIR approach is to compare it to the performance of the equivalent monolingual baseline. Bilingual runs that employ algorithms for mining ARs to generate a bilingual lexicon achieve up to 88% of monolingual performance, which is comparable to the state-of-the art.

The best scoring AR runs were the ones in which the corpus used for mining the ARs is a sample of the corpus used for querying. A T-test has shown no significant difference between the monolingual run the bilingual runs (*p*-values of 0.08 for Portuguese and 0.06 for Finnish). The results also show that the performance of the AR-LATimes runs is consistent for different languages as the results for Portuguese and Finnish are very similar. A manual analysis of the rules generated for AR-LATimes showed that only about 8% of the terms are translated incorrectly.

AR runs that employ different corpora for training and querying yielded poorer results. AR-EuroParl was significantly inferior to the monolingual baseline for both Portuguese and Finnish. AR-GH was superior to AR-EuroParl, indicating that the domain of the documents plays a very important role.

Table 1. Summary of the results. Bold indicates no signifficant different from monolingual.

RunId	Source Language	MAP (%mono)	#Terms	MissingTerms
Mono	EN	0.4423	946	—
AR-LATimes	PT	**0.3787** (86%)	848	64
AR-GH	PT	0.3392 (77%)	869	80
AR-EuroParl	PT	0.1954 (44%)	632	67
MT-Google	PT	**0.4181** (95%)	956	—
MT-Systran	PT	0.3247 (73%)	914	—
MT+AR	PT	**0.4942** (112%)	1064	—
AR-LATimes	FN	**0.3895** (88%)	650	103
AR-GH	FN	0.3695 (84%)	628	118
AR-EuroParl	FN	0.2288 (52%)	497	21
MT-Google	FN	**0.3782** (86%)	868	—
MT+AR	FN	**0.3981** (90%)	910	—

MT runs using Google Translator have outperformed AR runs in Portuguese, but not in Finnish. However, in both cases, no statistical difference was found between AR-LAtimes and MT-Google for Portuguese (*p*-value = 0.36) and for Finnish (*p*-value = 0.58). Intuitively, MT systems were expected to perform better than ARs as they employ much more sophisticated NLP methods. This lack of significant difference favours our simpler proposal.

The fourth column in Table 1 shows the number of terms used in each query. When the number of query terms in the translation is smaller than number of original query terms, there is an indication that no translation was found for some of the terms. We found a correlation of 0.70 between MAP and the number of query terms, i.e. the more terms used in the query, the better the result. Looking at the results for AR-EuroParl, noticeably the scores for Finnish are better than the scores for Portuguese. This can be explained by the number of query terms that were missing from the English/Portuguese parallel collection (67) as opposed to the number of terms that were missing from the English/Finnish parallel corpus (21).

The best bilingual results were obtained when combining MT and ARs. For Portuguese, this run was significantly better than all other bilingual runs and even outperformed the monolingual baseline. This gain can be attributed to the query expansion effect brought by our approach. For Finnish, the gain in performance was significant only compared to AR-EuroParl. It is worth pointing out that for Finnish, the only runs

with performances comparable to the monolingual baseline were the ones involving our AR approach (AR-LATimes and MT+AR).

Performing a fair comparison between our results and other groups' requires that they have used the same test collection, the same query topics which should be written in the same language. We are aware of only two studies that satisfy these requirements: Orengo & Huyck [13] and McNamee [11]. ARs significantly outperform a Latent Semantic Indexing-based approach presented in [13]. They have also used a MT system to simulate a parallel corpus and achieved MAP scores of 0.2088 which represent 87% of the equivalent monolingual performance.

On a topic-by-topic analysis we observed that some topics were helped by our approach and others were harmed. Comparing AR-LATimes and MT-Google, we observed that the MT system achieves better results in 18 topics, ARs are superior for 15 topics and the approaches tie for 9 topics. One general tendency is that ARs tend to outperform MT-based approaches in topics with larger number of keywords.

Because we are not restricted to choosing a single translation, in some topics our performance was superior to MT and even to the monolingual baseline. For example, in topic 122 the term "*internacional*" was translated to "*international*" and "*global*". This had the effect of query expansion and improved MAP by 426% in relation to MT-Google and 288% in relation to the monolingual baseline.

The reason for the highest drops in performance was the choice of a wrong translation, e.g. in query 114 "*líder*", which means "*leader*", was translated to "*drive*".

Acronyms have also been a problem for our approach. Their equivalent in the other language was not always found in the parallel corpus, and if they were in their expanded form they were not recognised as a multiword expression since we are not treating phrases at the moment.

Another reason for poor performance was missing translations. In such cases the problem was that the corpus used for mining the ARs did not have any occurrences of the term. This happened mostly with proper names such as "Eurofighter" (topic 93) and "Ames" (topic 100). Our approach performs worse than MT in queries that have named entities. Currently, we are giving a uniform treatment to all terms. This could be solved by using a dictionary containing named entities and their translations.

5 Conclusions

This work proposes the use of algorithms for mining ARs to the problem of finding term translations for CLIR. Our approach requires a parallel corpus to serve as the basis for the mining process.

In the experimental analysis, the LATimes test collection was used to retrieve documents in response to queries in Portuguese and in Finnish. Different alternatives were tested for parallel corpora: (i) a synthetic parallel corpus created by translating a sample of the collection used for querying; (ii) a synthetic parallel corpus created by translating a sample of a collection on the same domain as the one used for querying; and (iii) a real parallel corpus from a different domain. Our results are comparable to the monolingual baseline and to the results of MT systems applied to the purpose of translating queries, even though MT systems apply much more complex NLP techniques. This indicates ARs can be effectively used as an approach to CLIR. We have

also investigated the combination of MT and our AR-based approach. The results were very positive, and even surpassed the monolingual baseline for the Portuguese topics.

Despite the encouraging results obtained with our approach, there is still room for many improvements. So far, we only considered the translation of single terms. However, often, a single term in one language translates into two (or more) terms in another language. This situation can also be addressed by generating rules with two (or more) terms in the antecedent.

Our biggest cause for poor performance was that there were no translations for some terms. An alternative to be examined is to treat untranslatable words as proposed by Pirkola et al. [14]. The idea is to decompose the word into semantic units and then using these units combined to the other query terms for retrieving the documents.

Acknowledgements

This work was supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brazil) and INCT-Web.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Mining Association Rules between Sets of Items in Large Databases. In: Proc. of the ACM SIGMOD Conference on Management of Data, Washington, D.C. (1993)
2. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proceedings of the 20th VLDB Conference, Santiago, Chile, pp. 487–499 (1994)
3. Aguirre, E., et al.: CLEF 2008: Ad Hoc Track Overview. In: Working Notes for the CLEF 2008 Workshop, Aarhus, Denmark (2008)
4. ELDA, http://www.elra.info/ (accessed on: 15-October 2008)
5. EuroParl, http://www.statmt.org/europarl/ (accessed on: 15-October 2008)
6. Geraldo, A.P., Orengo, V.M.: UFRGS@CLEF2008: Using Association rules for Cross-Language Information Retrieval. In: Borri, F., Nardi, A., Peters, C. (eds.) Working Notes of CLEF 2008, Aarhus, Denmark (2008)
7. Global Reach, http://global-reach.biz/globstats/refs.php3 (accessed on: 19-October 2007)
8. Google Translator, http://www.google.com/translate_t (accessed on: 29-October 2008)
9. Grefenstette, G.: Cross-Language Information Retrieval, p. 200. Kluwer Academic Publishers, Boston (1998)
10. Kraaij, W., Nie, J., Simard, M.: Embedding web-based statistical translation models in cross-language information retrieval. Computational Linguistics 29(3), 381–419 (2003)
11. McNamee, P., Mayfield, J.: Scalable Multilingual Information Access. In: Peters, C., Braschler, M., Gonzalo, J. (eds.) CLEF 2002. LNCS, vol. 2785. Springer, Heidelberg (2003)
12. Nie, J., et al.: Cross-Language Information Retrieval based on Parallel Texts and Automatic Mining of Parallel Texts from the Web. In: SIGIR, pp. 74–81 (1999)
13. Orengo, V.M., Huyck, C.R.: Portuguese-English Cross-Language Information Retrieval Using Latent Semantic Indexing. In: Peters, C., Braschler, M., Gonzalo, J. (eds.) CLEF 2002. LNCS, vol. 2785. Springer, Heidelberg (2003)

14. Pirkola, A., et al.: Dictionary-Based Cross-Language Information Retrieval: Problems, Methods, and Research Findings. Information Retrieval 4(3), 209–230 (2001)
15. Porter, M.F.: An Algorithm for Suffix Stripping. Program 14(3), 130–137 (1980)
16. Salton, G.: Automatic Processing of Foreign Language Documents. Journal of the American Society for Information Science 21(3), 187–194 (1970)
17. Savoy, J.: Combining Multiple Strategies for Effective Monolingual and Cross-Language Retrieval. Information Retrieval 7(1-2), 121–148 (2004)
18. Snowball Stemmers, `http://snowball.tartarus.org/texts/stemmersoverview.html` (accessed on: 28-October 2008)
19. Systran, `http://www.systransoft.com/` (accessed on: 22/01/2009)
20. Veloso, A., et al.: Learning to Rank at Query-Time using Association Rules. In: SIGIR 2008, Singapore, pp. 267–274 (2008)
21. World Internet Statistics, `http://www.internetworldstats.com/stats7.htm` (accessed on: 29- October 2008)
22. Yang, Y., et al.: Translingual Information Retrieval. In: 15th International Joint Conference on Artificial Inteligence (IJCAI), Nagoya, Japan (1997)
23. Zettair, `http://www.seg.rmit.edu.au/zettair/` (accessed on: 11/06/07)

A Comparison of Data-Driven Automatic Syllabification Methods

Connie R. Adsett[1,2,⋆] and Yannick Marchand[1,2,⋆⋆]

[1] Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia, Canada B3H 1W5
adsett@cs.dal.ca
[2] Institute for Biodiagnostics (Atlantic)
National Research Council Canada
1796 Summer Street, Suite 3900
Halifax, Nova Scotia, Canada B3H 3A7
yannick.marchand@nrc-cnrc.gc.ca

Abstract. Although automatic syllabification is an important component in several natural language tasks, little has been done to compare the results of data-driven methods on a wide range of languages. This article compares the results of five data-driven syllabification algorithms (Hidden Markov Support Vector Machines, IB1, Liang's algorithm, the Look Up Procedure, and Syllabification by Analogy) on nine European languages in order to determine which algorithm performs best over all. Findings show that all algorithms achieve a mean word accuracy across all lexicons of over 90%. However, Syllabification by Analogy performs better than the other algorithms tested with a mean word accuracy of 96.84% (standard deviation of 2.93) whereas Liang's algorithm, the standard for hyphenation (used in TEX), produces the second best results with a mean of 95.67% (standard deviation of 5.70).

Keywords: Natural language processing, machine learning, automatic syllabification.

1 Introduction

The capability to automatically determine the syllable boundaries in a word is useful for such applications as grapheme-to-phoneme (G2P) conversion and text-to-speech synthesis [1,2]. Data-driven methods are an attractive option to perform automatic syllabification because they are not language specific (unlike syllabification rules), simply requiring a lexicon of syllabified words for training. However, the best method for this task is undetermined.

⋆ The first author was funded by the National Sciences and Engineering Research Council of Canada (NSERC), the National Research Council (NRC) Graduate Student Supplement Program (GSSSP), an Izaak Walton Killam Predoctoral Scholarship, and the Eliza Ritchie Doctoral Scholarship for Women.
⋆⋆ The second author was funded by a discovery grant from NSERC.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 174–181, 2009.
© Springer-Verlag Berlin Heidelberg 2009

At best, recent work has only compared algorithms using one language [3,4]. This work compares data-driven automatic syllabification algorithms across nine European languages (Basque, Dutch, English, French, Frisian, German, Italian, Norwegian, and Spanish) in both the spelling and pronunciation domains.

2 Algorithms Compared

The problem of syllabification can be viewed as the task of determining whether or not a syllable boundary exists between each pair of contiguous symbols (letters or phonemes) in a word. A syllable boundary either exists between two symbols (a juncture) or it does not. The syllabification of the entire word can be understood as a structured classification problem because it is formed from the compilation of all juncture classifications. Therefore, the classification of individual junctures in a word are not independent. This is not taken into account in the IB1 and Look-up Procedure algorithms. In contrast, Liang's algorithm, Hidden Markov Support Vector Machines and Syllabification by Analogy incorporate structure information into training and testing.

2.1 Hidden Markov Support Vector Machines

This extension of the general structure SVM framework [5] was introduced by Altun, Tsochantaridis and Hofmann [6]. Intended for structured classification problems like syllabification, the Hidden Markov Support Vector Machine (HM-SVM) approach has produced better results than Syllabification by Analogy on English spelling domain words [3]. This work used the SVM^{hmm} 3.10 package[1].

This method uses a Hidden Markov approach by applying a Viterbi-like algorithm to determine the juncture classification, given previous classifications. The weight vector used in the function that discriminates between possible classification sequences is learned using a Support Vector Machine approach. The best classification sequence gives the maximal output for this function [6].

Training and testing require features for each word juncture. These are all possible substrings within a window of five characters on each side of the juncture. Features are binary values indicating the presence or absence of substrings at each window position. This formulation, along with the classification of junctures using numbered NB (non-boundary or boundary) tags, and the use of 0.1[2] and 0.5 for the C and ϵ parameters, respectively, were based on Bartlett's results [3].

2.2 IB1

When applied to syllabification, the IB1 algorithm (from the instance-based learning algorithm family) compares letter N-grams to determine the appropriate syllabification for each juncture. During training, an N-gram is stored for

[1] This is available from www.cs.cornell.edu/People/tj/svm_light/svm_hmm.html (last accessed 9 June, 2009).

[2] For SVM^{hmm} 3.10, the C parameter must be multiplied by the training set size.

each juncture in a word. An N-gram for each juncture in the testing word is then compared to those stored during training to infer the syllabification of the juncture (in the testing word) from the most similar.

A distance measure is used to determine how closely two N-grams match [7,8]. Three feature weighting functions: information gain (IB1-IG), gain ratio (IB1-GR)[3], and chi-squared (IB1-χ^2) can be used to set the values of the weights required for the distance function [8]. Each of these approaches assigns feature weights after training is complete. The values of features in the training data are used to compute the relevance of each feature to classification.

A variety of N-gram sizes were tested to determine which was best. From two characters to the left and right of the juncture (a 4-gram), the size was increased by two on each side up to a 20-gram (the limit of the TiMBL implementation[4]), keeping the number of characters equal on both sides of the juncture.

2.3 Liang's Hyphenation Algorithm

Since Liang formulated his TEX hyphenation algorithm, it has been a standard in the field [9]. Like syllabification, hyphenation is the segmentation of words into substrings.

During training, the `patgen` [9] program is used to generate a set of patterns for the hyphenation of new words[5]. These patterns differ from those used for IB1 and the Look Up Procedure; their length is not restricted and one pattern may contain information for multiple junctures. Once created, they are applied to the words to be syllabified [10].

The algorithm uses many parameters to create patterns, making it impossible to test all parameter settings. Specifically, these parameters are the number of training iterations (between 1 and 9) and, for each iteration, the minimum and maximum substring lengths of the generated patterns (from 1 to 15 characters) along with three values (`good`, `bad`, and `threshold`) used to determine desirable patterns (these may all range from 1 to ∞, in theory). According to Antoš [10], how to tune these parameters is an open problem.

Therefore, this work used parameters selected in previous studies on English hyphenation [9, Table 5], Czech hyphenation [11, Tables 4–9], German compound word hyphenation [12, Tables 4 and 5], and Thai segmentation [10, Table 7], [13, Tables 12.1 and 12.2] for syllabification. A freely available method developed by Ned Batchelder in July of 2007[6] was used to syllabify test words based on the generated patterns. The code was modified slightly to allow for the processing of lists of words using any set of patterns.

---

[3] This convention is not always followed in the literature and the name IB1-IG is sometimes used to refer to both the Information Gain and the Gain Ratio versions of this algorithm [8].

[4] This is available from `ilk.uvt.nl/timbl/` (last accessed 5 June, 2009).

[5] A version of the program is included by default in Linux distributions.

[6] This is available from `nedbatchelder.com/code/modules/hyphenate.html` (last accessed 9 June, 2009).

2.4 Look Up Procedure

Originally presented as a simpler and superior method to NETtalk for G2P conversion [14], the Look Up Procedure (LUP) has since been applied to automatic syllabification [7]. Except for feature weighting, it operates identically to IB1.

The Look Up Procedure weights are predetermined before testing and the weight set used determines the N-gram size. The 15 sets tested were the same weights used in previous syllabification studies [4] and were originally given by Weijters [14, Figure 2].

2.5 Syllabification by Analogy

Unlike other methods, Syllabification by Analogy (SbA) retains the training set in its entirety. Directed graphs compile the relevant syllabification information obtained lexical entries and are used to syllabify test words. Instances of all test word substrings from the training set are used to create the graph. Graph vertices and edges are labeled with the lexical syllabifications of the substrings (along with the corresponding number of occurences): the substring's initial and final characters form vertex labels and connecting edges are labeled with the intermediate information. The concatenation of the labels of all vertices and edges forming a complete path from the start to the end vertex thus provides a candidate syllabification for the test word.

Only the shortest pathes (without regard to edge weights) are considered and a combination of up to five scoring strategies is used to select the best shortest path. These strategies are the product of the edge weights (these correspond to the substring frequencies in the training data), the standard deviation of values associated with the path structure, the frequency of the same syllabification amongst the shortest paths, the number of differences between one candidate syllabification and all others, and the minimum edge weights. Each scoring strategy may be used independently or with other strategies to rank paths [15]. Because the best of the 31 possible scoring strategies combinations is unknown, all possible combinations were tested.

3 Languages and Lexicons Used

Nine languages were selected due to the availability of lexicons containing marked syllable boundaries. Entries which were clearly non-words, contained non-alphabetic characters or were in any way incomplete were removed. Additionally, proper nouns were removed and homophones and homographs with differing syllabifications to maintain consistency with previous automatic syllabification studies [4]. Because the number of similar entries was a concern, only the infinitive form of verbs and the singular form of nouns were retained in the Italian lexicon. Table 1 summarizes the sources and original and final sizes of the lexicons.

Table 1. The sources and sizes of the lexicons used in this work. S and P denote the domains for which lexicons were used (spelling and pronunciation).

		Number of Entries		
Language	Source	Original	Final	Domain
Basque	EuskalHitzak [16]	100,079	98,913	S,P
Dutch	CELEX [17]	124,136	115,182	S,P
English	CELEX [17]	52,447	31,467	S,P
French	Lexique3 [18]	138,175	31,156	S,P
Frisian	Jelske Dijkstra [19]	63,247	63,219	P
German	CELEX [17]	51,728	20,351	S,P
Italian	Italian Festival [20]	440,084	44,720	S,P
Norwegian	Terje Kristensen [21]	66,992	66,480	S
Spanish	BuscaPalabras [22]	31,491	31,364	S,P

4 Results

Training and testing were performed using 10-fold cross-validation using the same word length distribution in each lexicon fold. Results were computed using word accuracy: the percentage of words automatically syllabified according to the lexicon. After selecting the best parameter setting for each algorithm using the mean word accuracies over all lexicons, the results of each of the five algorithms were compared.

The IB1 algorithm obtained over 80% word accuracy on most lexicons. The highest average word accuracy was given by IB1_IG and the three largest N-gram sizes gave similar results with the best performance (94.36%) achieved using 12-grams.

The mean word accuracies from Liang's algorithm for each parameter set span a wide range of values; from 2.45% to 95.48% . Overall, the parameter settings previously used for Czech hyphenation [11, Table 9] give the best average word accuracy (95.48%).

Like the results of Liang's algorithm, those given by the Look Up Procedure range from 11.89% to 91.44%. Best average word accuracy across lexicons is achieved using the weights [1, 4, 16, 64, 16, 5, 1] which occur most in the top three for word accuracy for each lexicon (14 times). Average word accuracy using these weights is 91.44%.

In the case of HM-SVM, only one parameter setting was tested. These results give an average word accuracy of 95.17%.

All 31 scoring strategy combinations were tested for SbA. The best (using the frequency of the same syllabification and the largest minimum edge weight) achieved a mean word accuracy of 96.70%. At the individual lexicon level, this same combination again rose above the others with 11 occurrences in the top three scores for each lexicon.

Of the 16 lexicons tested, Liang's algorithm gave the best results for eight, and both IB1 and SbA performed best for four. Interestingly, although the IB1 algorithm treats each juncture classification as independent, it still produces the

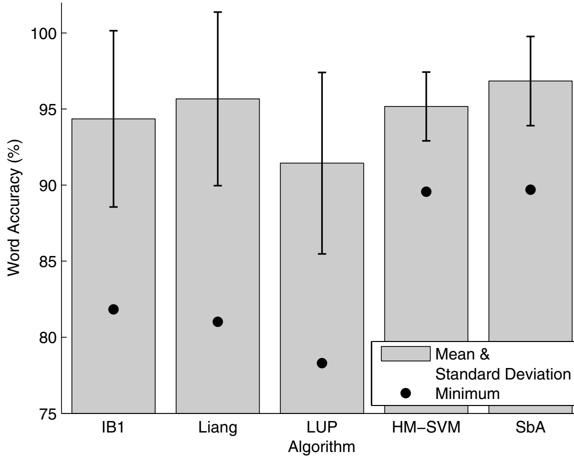


Fig. 1. Comparison of the minimum, mean and standard deviation word accuracy results of each algorithm

most accurate results for some lexicons. In fact, for nine of the 16 lexicons, the IB1 results are better than at least one of the algorithms that included structure information in the classification process. This difference in the algorithms does not seem to be the main key to accurate automatic syllabification.

These results point to Liang's algorithm as the best choice for automatic syllabification. However, for three lexicons, this approach provides the worst syllabifications. The poor performance of this algorithm on the French spelling domain (81.02%) and Spanish pronunciation domain (83.26%) point to why this algorithm does not give the overall best mean word accuracy (Figure 1).

Figure 1 also shows the minimum word accuracies and standard deviations of the mean word accuracies of the algorithms. Except for Liang's algorithm, which obtained minimum word accuracy in the French spelling domain, the English spelling domain lexicon was the source of the minimum word accuracies. This is not surprising, given that English syllabification is thought to be more complex (allowing a wider variety of syllabic structures) than other languages. HM-SVM and SbA have lower standard deviations and much higher minimum word accuracies than the other three algorithms because these methods are better able to model the syllabification task regardless of the differing language characteristics. Of the two, SbA gives the highest mean word accuracy which is also significantly better than Liang's algorithm (χ^2_{obt} = 1735.17, $p < 0.0001$).

These results differ from previous findings which showed that the HM-SVM method outperformed SbA for automatic syllabification in the English spelling domain [3]. This may be because training and test methods used were different (prior work used a training set of 14,000 words and a test set of 25,000 words from CELEX [17]).

Overall, these results point to SbA as a better choice for a syllabification algorithm.

5 Conclusions and Future Work

This work has shown, using a wide range of languages in both the spelling and pronunciation domains, that some data-driven algorithms automatically syllabify words better regardless of the lexicon characteristics. All algorithms achieve mean word accuracies over 90% but, from these results, the Syllabification by Analogy approach appears to be the best choice for automatic syllabification tasks. Although not producing as high word accuracy, Hidden Markov Support Vector Machines have also been shown to be more consistent in syllabification results. Liang's algorithm, although performing well for some lexicons, also syllabifies much less accurately for some lexicons. This same variability in results across lexicons is seen in the IB1 and Look-up Procedure algorithms.

Deeper investigation into how optimal parameters can be chosen is especially necessary for Liang's algorithm and the Look Up Procedure, given the large feature spaces of these two methods. Furthermore, by nature, data-driven algorithms require a lexicon of syllabified words in order to be used but the impact of lexicon size is unknown. Testing increasing lexicon sizes in each language and for each algorithm would provide insight into which algorithm is capable of learning syllable boundaries, given as little data as possible. Finally, it has been demonstrated that automatic syllabification information improves English grapheme-to-phoneme conversion accuracy [1]. Testing this with additional languages would clarify whether this finding generalizes to other languages.

References

1. Bartlett, S., Kondrak, G., Cherry, C.: Automatic syllabification with structured SVMs for letter-to-phoneme conversion. In: Proceedings of ACL 2008: HLT, Columbus, Ohio, pp. 568–576 (2008)
2. Libossek, M., Schiel, F.: Syllable-based text-to-phoneme conversion for German. In: Proceedings of the Sixth International Conference on Spoken Language Processing (ICSLP 2000), Beijing, China, pp. 283–286 (2000)
3. Bartlett, S.E.: A discriminative approach to automatic syllabification. Master's thesis, Department of Computing Science, University of Alberta (2007)
4. Marchand, Y., Adsett, C.R., Damper, R.I.: Automatic syllabification in English: A comparison of different algorithms. Language and Speech 52(1), 1–27 (2009)
5. Tsochantaridis, I., Hofmann, T., Joachims, T., Altun, Y.: Support vector machine learning for interdependent and structured output spaces. In: Proceedings of the 21st International Conference on Machine Learning (ICML 2004), Banff, Canada, pp. 104–112 (2004)
6. Altun, Y., Tsochantaridis, I., Hofmann, T.: Hidden markov support vector machines. In: Proceedings of the Twentieth International Conference on Machine Learning (ICML 2003), Washington, DC, pp. 3–10 (2003)
7. Daelemans, W., van den Bosch, A.: Generalization performance of backpropagation learning on a syllabification task. In: Drossaers, M.F.J., Nijholt, A. (eds.) TWLT3: Connectionism and Natural Language Processing, Enschede, The Netherlands, pp. 27–37 (1992)
8. Daelemans, W., Zavrel, J., van der Sloot, K., van den Bosch, A.: TiMBL: Tilburg Memory-Based Learner, 6.0th edn., Tilburg, The Netherlands (2007)

9. Liang, F.M.: Word Hy-phen-a-tion by Com-put-er. PhD thesis, Stanford University, Palo Alto, CA (1983)
10. Antoš, D.: PatLib, pattern manipulation library. Master's thesis, Faculty of Informatics, Masaryk University Brno (2001)
11. Sojka, P., Ševeček, P.: Hyphenation in TEX - quo vadis? TUGboat 16(3), 280–289 (1995)
12. Sojka, P.: Notes on compound word hyphenation in TEX. TUGboat 16(3), 290–297 (1995)
13. Sojka, P., Antoš, D.: Context sensitive pattern based segmentation: A Thai challenge. In: Proceedings of EACL 2003 workshop Computational Linguistics for South Asian Languages – Expanding Synergies with Europe, Budapest, Hungary, April 2003, pp. 65–72 (2003)
14. Weijters, A.J.M.M.: A simple look-up procedure superior to NETtalk? In: Proceedings of the International Conference on Artificial Neural Networks (ICANN 1991), Espoo, Finland, pp. 1645–1648 (1991)
15. Marchand, Y., Damper, R.I.: A multistrategy approach to improving pronunciation by analogy. Computational Linguistics 26(2), 195–219 (2000)
16. Perea, M., Urkia, M., Davis, C.J., Agirre, A., Laseka, E., Carreiras, M.: E-Hitz: A word frequency list and a program for deriving psycholinguistic statistics in an agglutinative language (Basque). Behavior Research Methods 38(4), 610–615 (2006)
17. Baayen, R.H., Piepenbrock, R., Gulikers, L.: The CELEX lexical database (CD-ROM). Technical report, Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA (1995)
18. New, B., Pallier, C.: Manuel de Lexique 3, France. 3.03 edn. (2005)
19. Dijkstra, J., Pols, L.C.W., Van Son, R.J.J.: Frisian TTS, an example of bootstrapping TTS for minority languages. In: Proceedings of the 5th ISCA Speech Synthesis Workshop, Pittsburgh, pp. 97–102 (2004)
20. Cosi, P., Tesser, F., Gretter, R., Avesani, C.: Festival speaks Italian? In: Proceedings of Eurospeech 2001, Aalborg, Denmark, pp. 509–512 (2001)
21. Kristensen, T.: A neural network approach to hyphenating Norwegian. In: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN 2000), Como, Italy, vol. 2, pp. 148–153. IEEE, Los Alamitos (2000)
22. Davis, C.J., Perea, M.: BuscaPalabras: A program for deriving orthographic and phonological neighborhood statistics and other psycholinguistic indices in Spanish. Behavior Research Methods 37(4), 665–671 (2005)

Efficient Index for Retrieving Top-k Most Frequent Documents*

Wing-Kai Hon[1], Rahul Shah[2], and Shih-Bin Wu[1]

[1] Department of Computer Science, National Tsing Hua University, Taiwan
[2] Department of Computer Science, Louisiana State University, LA, USA

Abstract. In the *document retrieval problem* [9], we are given a collection of documents (strings) of total length D in advance, and our target is to create an index for these documents such that for any subsequent input pattern P, we can identify which documents in the collection contain P. In this paper, we study a natural extension to the above document retrieval problem. We call this *top-k frequent document retrieval*, where instead of listing all documents containing P, our focus is to identify the top k documents having most occurrences of P. This problem forms a basis for search engine tasks of retrieving documents ranked with TFIDF metric.

A related problem was studied by [9] where the emphasis was on retrieving all the documents whose number of occurrences of the pattern P exceeds some frequency threshold f. However, from the information retrieval point of view, it is hard for a user to specify such a threshold value f and have a sense of how many documents will be outputted. We develop some additional building blocks which help the user overcome this limitation. These are used to derive an efficient index for top-k frequent document retrieval problem, answering queries in $O(P + \log D \log\log D + k)$ time and taking $O(D \log D)$ space. Our approach is based on novel use of the suffix tree called *induced generalized suffix tree* (IGST).

1 Introduction

String matching problems have been studied for more than three decades [2, 4, 5]. In the simplest form, we are given a relatively long character string, called *text*, and a relatively short character string, called *pattern*, in which our target is to locate all occurrences of the pattern within the text. In some applications, the text is given in advance, and we may preprocess it and create an auxiliary data structure—called an *index*—for the text, so that any subsequent pattern matching query can be answered more efficiently. For instance, if *suffix tree* [8, 12]—a linear-space index—for the text is created, locating all *occ* occurrences of

* This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082-MY3 (W. Hon) and US NSF Grant CCF–0621457 (R. Shah).

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 182–193, 2009.
© Springer-Verlag Berlin Heidelberg 2009

a pattern P of length $|P|$ can be done in *optimal* $O(|P| + occ)$ time, irrespective of the length of the text.

In string databases or in string retrieval systems, we have a collection Δ of multiple documents(strings) instead of just one text string. In this case, the basic problem is to retrieve all the documents in which the query pattern P occurs. This is known as *document retrieval* problem and has been studied by Matias et al [7] and Muthukrishan [9]. The main issue here is that there may be many occurrences of the pattern over the entire collection Δ, but the overall number of documents in which the pattern occurs might be much smaller. Thus, the naive method of finding all the occurrences first and then reporting unique documents is far from efficient. Muthukrishnan [9] gave an optimal $O(D)$ space data structure which answers the document retrieval query in $O(|P| + output)$, where *output* is the number of documents which have the pattern P. This has been a popular approach of many subsequent papers [10, 11] which attempted to derive succinct/compressed data structures for this problem.

A more interesting variant was also proposed in [9] where we need to retrieve only those documents which have more than K occurrences of the pattern. This was called *K-mine* problem. In terms of information retrieval this is a more interesting query because it attempts to obtain only those document which are highly relevant. The notion of relevance here is simply the term frequency. Sadakane [10] also gave a method to compute TFIDF scores of each retrieved document. However, what is lacking here is the notion of top-k highest TFIDF documents. In [10] one needs to retrieve all the documents first, and then only the scores are computed. Related to document retrieval, a more general problem of position-restricted substring matching was introduced by [3, 6]. Also, the study of top-k indexing and rank sensitive data structures was carried by [1]. None of these results are directly applicable here.

The problem considered in this paper is closely related to the *K-mine* problem. In our case, we directly find top-k documents having the maximum number of occurrences for the given pattern. We build novel primitives like *inverse document mine* query, which quickly allows us to find a threshold K which is the frequency of the pattern in kth frequent document. Based on this we can draw strong connections with the *K-mine* problem. The main component of our solution is called *induced generalized suffix tree* (IGST), which is structurally the same as the index proposed by [9]. However, we show novel application to the IGST where we "linearize" it and combined it with successor searching functionality to obtain the desired performance.

For the sake of completeness we mention here that if theoretical performance guarantees are not important then such problems are practically solved using inverted indexes [14]. However, inverted indexes either only allow efficient searching for certain predefined pattern, or they take a lot more space (if they were to answer for arbitrary patterns). Our solution provides theoretical guarantees but can also be seen as a modification of inverted index where the lists for the patterns (which are contained within some other patterns) are smartly combined to reduce space.

1.1 Our Problems

In this paper, we study two natural extensions to the above document retrieval problems. The first one is called the *inverse document mining* problem, which is defined as follows:

Problem 1: Inverse Document Mining Problem

Given: A collection Δ of documents, with total length D;
Target: Create an index for Δ to support the following query efficiently:
On given any input pattern P and any input integer k, find the frequency f such that $\rho_{f+1} < k \leq \rho_f$, where ρ_f denotes the number of documents in Δ containing at least f occurrences of P. In other words, we want to find the largest f such that there are k documents with f occurrences of P.
We denote the above query by $\mathit{inverse_mine}(P, k)$.

Example. Suppose there are five documents, T_1, T_2, T_3, T_4, and T_5, in the set Δ. Also, the number of times a pattern P occurring in these documents are $15, 24, 3, 3, 1$, respectively. On the query $\mathit{inverse_mine}(P, 2)$, we should return $f = 15$. □

The second extension is called the *top-k document retrieval* problem, which is motivated by the need of finding the "most relevant" documents in the output of a document retrieval query. Here, we assume that the relevance of a document with respect to a pattern P is measured by the number of times P occurring in the document. Our problem is then defined as follows:

Problem 2: Top-k Document Retrieval Problem

Given: A collection Δ of documents, with total length D;
Target: Create an index for Δ to support the following query efficiently:
On given any input pattern P and any input integer k, find the k documents in Δ which contain the most occurrences of P. We assume tie is broken arbitrarily in case two documents contain the same number of P.
We denote the above query by $\mathit{top_document}(P, k)$.

Example. Suppose there are five documents, T_1, T_2, T_3, T_4, and T_5, in the set Δ. Also, the number of times a pattern P occurring in these documents are $15, 24, 3, 3, 1$, respectively. On the query $\mathit{top_document}(P, 2)$, we should return $\{T_1, T_2\}$. However, on the query $\mathit{top_document}(P, 3)$, we may return either $\{T_1, T_2, T_3\}$ or $\{T_1, T_2, T_4\}$, as tie is broken arbitrarily among documents with the same number of occurrences of P. □

By adapting Muthukrishnan's $O(D \log D)$-space indexes [9], the queries in the above two problems can readily be supported in $O(|P| \log D)$ time and $O(|P| \log D + k)$ time, respectively. In this paper, we propose alternative indexes with the same space, so that the queries are supported in $O(|P| + \log D \log\log D)$ time and $O(|P| + \log D \log\log D + k)$ time, respectively. These indexes thus outperform

the naive extension of Muthukrishnan's index whenever P is sufficiently long; precisely, when $|P| = \omega(\log \log D)$.

The core of our indexes, called the *induced generalized suffix trees*, are structurally equivalent to the core of the indexes proposed in [9]; the major difference lies in the information being stored. Consequently, we are able to support the new types of query, and alter the searching methods to obtain the desired trade-off in query times.

1.2 Paper Organization

The remainder of the paper is as follows. Section 2 introduces two basic tools which form the building blocks of our indexes. Section 3 describes the induced generalized suffix trees (IGST), with which we can construct the index for the inverse document mining problem. In Section 4, we show how to adapt the IGST slightly to solve the top-k document retrieval problem. We conclude in Section 5 with some open problems.

2 Basic Tools

2.1 Generalized Suffix Tree

Let $\Delta = \{T_1, T_2, \ldots, T_m\}$ denote a set of documents. Each document is a character string with characters drawn from a common alphabet Σ whose size $|\Sigma|$ can be unbounded. For notation purpose, we assume that for each i, the last character of document T_i is marked by a special character $\$_i$, which is unique among all characters in all documents.[1] The *generalized suffix tree* [8, 12] (GST) for Δ is a compact trie storing all suffixes of each T_i. Precisely, each suffix of each document corresponds to a distinct leaf in the GST. Each edge is labeled by a sequence of characters, such that for each leaf representing some suffix s, the concatenation of the edge labels along the root-to-leaf path is exactly s. In addition, for any internal node u, the edges incident to its children all differ by the first character in the corresponding edge labels, so that the children of u are ordered according to the alphabetical order of such a first character. The following property of the GST is immediate:

Lemma 1. *Consider all suffixes of all documents in Δ. The jth smallest suffix corresponds to the jth leftmost leaf in GST.*

Next, we define an important concept called *suffix range*:

Definition 1. *Consider the subtree of a node u in the GST. Let v and w be the leftmost and the rightmost leaves in this subtree. Furthermore, let ℓ and r denote the rank of v and the rank of w among all leaves in the GST, respectively; precisely, v is the ℓth leftmost leaf and w is the rth leftmost leaf in GST. Then, the range $[\ell, r]$ is called the* suffix range *of u.*

[1] When the context is clear, we shall simply denote each $\$_i$ by the same character $\$$.

A simple observation is shown as follows:

Lemma 2. *Let u and v be two nodes in the GST, and let $[\ell_u, r_u]$ and $[\ell_v, r_v]$ be their suffix ranges, respectively. The two ranges are disjoint if and only if there is no ancestral-descendant relationship between u and v. In case u is an ancestor of v, we have $\ell_u \leq \ell_v \leq r_v \leq r_u$.*

For each node v, we use *path*(v) to denote the concatenation of edge labels along the path from root to v. Then, we define the concept of *locus* as follows:

Definition 2. *For any string Q, the* locus *of Q in the GST is defined to be the highest node v (i.e., nearest to the root) such that Q is a prefix of path(v). In case no v satisfies the condition, the locus of Q is* null.

Note that the locus of Q can be determined in $O(|Q|)$ time by traversing the GST and matching characters of Q from the beginning to the end.

It is easy to see that if a pattern P occurs at position j in a text T, P must be a prefix of the suffix of T which starts at position j. The converse is also true. Based on this, we have the following lemma which captures the power of GST in pattern matching:

Lemma 3. *A pattern P occurs in some document of Δ if and only if the locus of P is not null. In addition, each leaf in the subtree rooted at the locus of P corresponds to a distinct occurrence of P, and vice versa.*

2.2 Optimal Index for Colored Range Query

Let $A[1..n]$ be an array of length n, with each entry storing a color drawn from $C = \{1, 2, \ldots, c\}$. A *colored range query*, denoted by $CRQ(i, j)$, receives two input integers i and j with $1 \leq i \leq j \leq n$, and outputs the set of colors contained in the subarray $A[i..j]$. For instance, suppose A has seven entries, which are colored by $1, 3, 2, 6, 2, 4$, and 5, respectively. Then, the query $CRQ(2, 5)$ requests the set of colors in the subarray $A[2..5]$, which should return $\{2, 3, 6\}$.

An index is proposed in [9] for answering colored range query in an *output-sensitive* manner; the performance of the index is summarized in the following lemma:

Lemma 4. *We can create an index for the array A, using $O(n)$-space of storage, such that for any i and j, the colored range query $CRQ(i, j)$ can be answered in $O(\gamma)$ time, where γ denotes the number of (distinct) colors in the output set.*

2.3 Y-Fast Trie for Efficient Successor Query

Let S be a set of n distinct integers taken from $[1, D]$. Given an input x, a *successor query* on S reports the smallest integer in S that is greater than or equal to x. An efficient index for this query, called *y-fast trie*, was proposed by [13], whose performance is summarized as follows:

Lemma 5. *We can create an index for the set S of n integers, using $O(n)$-space of storage, such that for any input x, the successor query $succ(S, x)$ can be answered in $O(\log \log D)$ time, where D denotes the universe where integers of S are chosen from.*

3 Induced Generalized Suffix Tree

This section defines the *induced generalized suffix tree for frequency f*, or *IGST-f*, which can be used to count the number of documents with P occurring at least f times. Then, we give an array representation of IGST-f, and show how to support *inverse_mine*(P, k) query efficiently.

3.1 IGST-f: The IGST for Frequency f

First, we define a tree induced from the GST, called *pre-IGST-f*, as follows:

Definition 3. *Consider the GST for Δ and an integer f with $1 \leq f \leq D$. Suppose each leaf of GST is labeled by the origin (document) of the corresponding suffix. For each internal node v of the GST, we say v is f-*frequent *if in the subtree rooted at v, at least f leaves have the same label. The induced subtree of GST formed by retaining all f-frequent nodes is called the* pre-IGST-f.

Definition 4. *Let v be a node in the pre-IGST-f, so that v is an internal node in the GST. We use* count(f, v)[2] *to denote the number of distinct document with at least f leaves labeled by it in the subtree rooted at v in the GST.*

The following two lemmas demonstrate the pattern matching power of pre-IGST-f, which can both be proved easily based on Lemma 3:

Lemma 6. *A pattern P occurs at least f times in some document of Δ if and only if the locus of P in pre-IGST-f is not null.*

Lemma 7. *Let ρ_f denote the number of documents in Δ with pattern P occurring at least f times. If the locus of P in pre-IGST-f is null, then $\rho_f = 0$; otherwise, $\rho_f = count(v)$, where v is the locus of P.*

Next, we describe IGST-f, which is in fact a simplified version of pre-IGST-f.

Definition 5. *Consider the pre-IGST-f. For each internal node v, we say v is* redundant *if (i) v is a degree-1 node and (ii) $count(v) = count(\text{child}(v))$, where* child$(v)$ *denotes the unique child of v. The induced tree formed by contracting all redundant nodes in the pre-IGST-f is called the* IGST-f.

Observe that when a node v in the pre-IGST-f is redundant, the set of the $count(v)$ documents corresponding to v (where each of them has at least f labels in the subtree rooted at v in the GST) is exactly the same as the set of the $count(\text{child}(v))$ documents corresponding to child(v), so that the two counts are the same. This observation immediately leads to the following lemma:

[2] Or, we simply use *count(v)* instead when context is clear.

Lemma 8. *The locus of a pattern P in pre-IGST-f is not null if and only if the locus of P in IGST-f is not null. In case the locus is not null, let v and w denote the locus of P in pre-IGST-f and the locus of P in IGST-f, respectively. Then, $count(v) = count(w)$.*

The structure of our IGST-f is equivalent to the structure of the index proposed by Muthukrishnan [9] to solve the *document mining* problem. If we assume that the value of $count(v)$ is stored for each node v in the IGST-f (which is not required in [9]), we obtain the following theorem:

Theorem 1 (Adaptation of Muthukrishnan's Index). *By storing IGST-1, IGST-2, ..., and IGST-D, we can answer $inverse_mine(P, k)$ query, for any input pattern P and input integer k, in $O(|P| \log D)$ time.*

Proof. For any f, we can find the locus of P in IGST-f in $O(|P|)$ time, analogous to finding locus in the GST. Then, we can determine the number of documents containing at least f occurrences of P, based on Lemma 6. To answer $inverse_mine(P, k)$, it is sufficient to search for $O(\log D)$ IGSTs, based on a binary search of f, thus using $O(|P| \log D)$ time in total.

3.2 Array Representation of IGST-f

In Theorem 1, answering the *inverse_mine* query requires finding the locus of P in each IGST during the binary search. This could be time-consuming when P is long. In the following, we propose a simple alternative scheme that allows each locus-finding step to be done in $O(\log D)$ time instead of $O(|P|)$ time, thus giving a trade-off in the query time.

Our scheme is to make use of the suffix ranges. Firstly, suppose that the suffix range of the locus of P is already computed. Let $[\ell_P, r_P]$ denote this range if the locus exists. Next, suppose each node in IGST-f is associated with the suffix range of the corresponding node in the GST. Then, we have the following observation:

Lemma 9. *The locus of P in IGST-f, if exists, is the node v such that (i) the associated suffix range of any descendant of v (including v) is a subrange of $[\ell_P, r_P]$, and (ii) the associated suffix range of its parent node is not a subrange of $[\ell_P, r_P]$.*

Proof. By definition, P is a prefix of $path(v)$, so that by Lemma 2 and by the definition of IGST, the associated suffix range of v, and also any of its descendant, must be a subrange of $[\ell_P, r_P]$. On the other hand, the associated suffix range of the parent of v must not be a subrange of $[\ell_P, r_P]$, since otherwise, v is not the node nearest to the root having P as a prefix of $path(v)$, contradicting the definition of locus.

Next, consider performing a pre-order traversal on the IGST-f, and enumerating the associated suffix range of a node as it is visited. Let $[\ell(z), r(z)]$ denote the suffix range of the zth node enumerated during the traversal. Now, suppose that

the locus of P in IGST-f (assuming exists) is the jth node in the pre-order traversal of IGST-f. That is, the locus of P in IGST-f has associated suffix range $[\ell(j), r(j)]$. Further, suppose that we examine $[\ell(z), r(z)]$ for some z. The theorem below is the heart of our proposed index, which gives a simple way to determine the relationship between j and z: [3]

Theorem 2. *The following statements are true, and cover all possible relationship between $\ell_P, r_P, \ell(z)$, and $r(z)$:*

1. *if $r_P < \ell(z)$, then $j < z$;*
2. *if $\ell_P > r(z)$, then $j > z$;*
3. *if $[\ell(z), r(z)]$ is a subrange of $[\ell_P, r_P]$, then $j \le z$;*
4. *if $[\ell_P, r_P]$ is a subrange of $[\ell(z), r(z)]$, then $j \ge z$.*

Proof. All the four statements can be proven based on Lemma 8. For Statement 1, if $r_P < \ell(z)$, the associated suffix range of the zth node, and all nodes visited after the zth node in the pre-order traversal, must be disjoint with $[\ell_P, r_P]$, so that none of them can be the locus of P. Thus, the desired locus must be visited earlier, so that $j < z$. Similarly, for Statement 2, if $\ell_P > r(z)$, then $j > z$. For Statement 3, the desired locus must be an ancestor of the zth visited node, so that it is either the zth visited node itself, or a node visited earlier in the traversal. This implies $j \le z$. Similarly, for Statement 4, the desired locus must be a descendant of the zth visited node, so that $j \ge z$.

Let $c(z)$ denote the *count* value of the zth node visited during the pre-order traversal of IGST-f. Instead of storing the IGST-f as a tree structure in Theorem 1, we are going to represent it by an array I, such that the zth entry of I, $I[z]$, stores the 3-tuple $(\ell(z), r(z), c(z))$.

Based on the previous theorem, we can obtain the value j, using *binary search* on the array I, such that $[\ell(j), r(j)]$ is the associated suffix range of the locus of P. Then, the value $c(j)$ thus stores the number of documents with at least f occurrences of P.[4] Since the number of nodes in IGST-f is $O(D)$, the array I is of length $O(D)$, so that the binary search takes $O(\log D)$ time. This gives the following theorem:

Theorem 3. *By storing the GST, and the I-arrays for each IGST-1, IGST-2, ..., and IGST-D, we can answer inverse_mine(P, k) query, for any input pattern P and input integer k, in $O(|P| + \log^2 D)$ time.*

Proof. The GST is used to compute $[\ell_P, r_P]$ in $O(|P|)$ time. Then, we can determine the number of documents containing at least f occurrences of P in $O(\log D)$ time, by binary searching the I-array of IGST-f. To answer *inverse_mine*(P, k), it is sufficient to search for $O(\log D)$ I arrays of the IGSTs. The total time is thus $O(|P| + \log^2 D)$.

[3] Recall that $[\ell_P, r_P]$ denote the suffix range of the locus of P in the GST.

[4] If the locus of P in IGST-f does not exist, there is no z such that $[\ell(z), r(z)]$ is a subrange of $[\ell_P, r_P]$; consequently the binary search will correctly detect this.

Indeed, we can further speed up the query time by replacing each binary search in the IGST arrays with a single successor query in a slightly modified array. Consequently, the time spent in each visited IGST is reduced from $O(\log D)$ to $O(\log \log D)$ time. The idea is as follows. First, we observe that each node in the IGST has a natural correspondence in the original GST; precisely, a node u with suffix range $[\ell, r]$ exists in the IGST implies that a node u' with the same suffix range exists in the original GST. Next, we perform a pre-order traversal in the original GST, so that each node v receives the first time $\alpha(v)$ and the last time $\beta(v)$ visited during the traversal.[5] Then, each node u in the IGST is augmented with the information $\alpha(u')$ and $\beta(u')$ where u' is its correspondence in the GST. After that, for each IGST, we collect the set of α values of the nodes, and store a y-fast trie so that the successor query on the α values can be answered in $O(\log \log D)$ time.

Now, to perform searching, we first obtain the locus of P in the original GST, say u_P, whose pre-order traversal times are $\alpha(u_P)$ and $\beta(u_P)$. Since traversal times has a nice nested property, to search for the locus of P in IGST-f, it is equivalent to finding the successor of $\alpha(u_P)$ in the set of α values of IGST-f. Precisely, let u be the node in IGST-f with $\alpha(u)$ being the successor of $\alpha(u_P)$. It is easy to check that $\beta(u) \leq \beta(u_P)$ if and only if the locus of P exists in IGST-f, with u being the locus. After the above successor query, we can check the corresponding 3-tuple (ℓ, r, c) of u to determine how many documents contain at least f occurrences of P. This gives the following theorem.

Theorem 4 (Our Proposed Index). *By storing the GST, and the I-arrays for each IGST-1, IGST-2, ..., and IGST-D, we can answer inverse_mine(P, k) query, for any input pattern P and input integer k, in $O(|P| + \log D \log \log D)$ time.*

As shown in [9], the number of nodes in IGST-f is $O(D/f)$ for any f. Briefly speaking, each leaf or each degree-1 node in IGST-f corresponds to a disjoint set of at least f suffixes of the documents in Δ, so that there are $O(D/f)$ of them. On the other hand, the number of remaining nodes cannot exceed the total number of leaves and degree-1 nodes, so that there are $O(D/f)$ of them. Thus, the total number of nodes is $O(D/f)$. This gives the following space complexity result:

Theorem 5. *The total space of the indexes in Theorem 1, Theorem 3, or Theorem 4, is $O(D \log D)$.*

Proof. The theorem follows since $\sum_{f=1}^{D} D/f = O(D \log D)$.

Remarks. In the above discussion, we have focussed on the design of the index and have not mentioned the construction time. In fact, the index of Theorem 1, which is equivalent to the index in [9] with *count* value augmented to each node, can be constructed in $O(D \log^2 D)$ time by a simple adaptation of the construction algorithm in [9]. Once the count information in each node is available, the

[5] We assume a global time which is incremented by 1 whenever a node is visited.

index of Theorem 3 can be constructed in $O(D \log D)$ extra time to traverse each IGST and build the corresponding array. The index of Theorem 4 requires the construction of the y-fast trie, which in turn can be done in randomized $O(D \log D)$ extra time. Thus, although the query time by the index of Theorem 3 is slightly slower than that of Theorem 4, the former index has a slight advantage (worst-case guarantee) in its construction time.

4 Efficient Index for Top-k Document Retrieval Problem

Once we have obtained an index for answering the *inverse_mine* query, we can find the set of documents in $top_document(P, k)$, that is, those k documents with the most occurrences of P, by the following framework:

Solving $top_document(P, k)$**:**
1. Find f^* such that $f^* = inverse_mine(P, k)$;
`/* Consequently, there are at least` k `documents with at least` f^* `occurrences of` P`, but there are less than` k `documents with at least` $f^* + 1$ `occurrences of` P `*/`
2. Output all documents with at least $f^* + 1$ occurrences of P. Let k' be the number of such documents;
3. Output $k - k'$ extra documents, distinct from those obtained in Step 2, which have at least f^* occurrences of P.

One way to solve Step 2 is to augment each node v of IGST-f by the list of the associated $count(v)$ documents, each of which has at least f labels in the subtree rooted at v. Then, it is easy to see that in order to answer Step 2, we can just find the locus of P in IGST-$(f^* + 1)$, and output the list of documents in the locus. This method takes optimal $O(k')$ time in reporting the documents. Unfortunately, in the worst case, the extra space we need for the augmentation is $O(Dm \log D)$, where m denotes the number of documents in Δ. We refer this method as *Heuristic I*.

A better way to solve Step 2 is to apply Muthukrishnan's index for colored range query (Lemma 4 in Section 2) as an auxiliary data structure, as it is used in [9] for solving the document mining problem. The idea is that: For each node v in IGST-f, we only store the sublist of the associated $count(v)$ documents, where each such document does not appear in the list of the proper descendant of v in IGST-f. Next, we perform a pre-order traversal, and concatenate the list of the visited node into a single list L. Then, it is easy to check that each node v will correspond to a subrange in the list L, such that its associated $count(v)$ documents will correspond *exactly* to the $count(v)$ distinct documents in the subrange.

Thus, if we use Muthukrishnan's index for storing L, and for each node, we store the starting and ending positions in L for the associated subrange, Step 2

can also be answered optimally in $O(k')$ time, as in Heuristic I, but with a smaller $O(D \log D)$ space requirement.[6]

Step 3 can be solved similarly as in Step 2. We observe that any k documents with at least f occurrences of P, together with the k' documents obtained in Step 2, must contain a desired set of k documents for our *top_document*(P, k) query. So in Step 3, we will arbitrarily select a set of k documents with at least f occurrences of P, from which we further select $k - k'$ documents that are not obtained in Step 2. A simple way to solve the latter part is by sorting, taking $O(\min\{m, k \log k\})$ time. To speed up, we maintain an extra bit-vector of m bits, where the ith bit corresponds the document T_i, with all bits initialized to `0` at the beginning. When Step 2 is done, we mark each bit corresponding to the k' documents by `1`. Then, in Step 3, when a document is examined in the latter procedure, we can check this bit-vector to see whether a document has been obtained in Step 2 already, so that we can easily obtain the desired set of extra $k - k'$ documents. After Step 3, we can simply reset the bit-vector in $O(k)$ time by referring to the final output. Thus, we have completely solved the *top_document* query, and have obtained the following theorem:

Theorem 6 (Our Proposed Index). *We can maintain an $O(D \log D)$-space index for Δ, such that for any input pattern P and input integer k, top_document (P, k) query can be answered in $O(|P| + \log D \log \log D + k)$ time.*

Remark. Although Heuristic I does not guarantee good worst-case space bound, as our preliminary experiments shown, its space may be better than storing the color-range-query index in practice. We defer the details to the full paper.

5 Conclusion and Open Problems

We have proposed the *inverse document mining* and the *top-k document retrieval* problems, and provided indexes which support the required queries in near-optimal time.

The core of our indexes, called the *induced generalized suffix trees*, are adapted from the ones in [9], where we store new sets of data to support our desired queries. In addition, we devise a simple, yet novel, array representation to organize our data, which consequently leads to an interesting way to answer our queries.

In the future, we would like to conduct a thorough experimental studies for the practicality of our index and (the adapted version of) Muthukrishnan's indexes, and to run on queries obtained from the actual web searching applications. We would like to compare our performance with that of inverted index based approach for arbitrary patterns. Another interesting direction is to consider the case where k is fixed, and see if we can achieve better time/space bounds.

[6] It is easy to check that the list L in IGST-f has $O(D/f)$ entries, since each entry corresponds a distinct f labels from the same document. Thus, the total space is $\sum_f D/f = O(D \log D)$.

In addition, a main open question in this field is that: Is there a linear space index taking just $O(D)$ space or even better only the space close to the compressed form of the text? Many extensions of these basic functionalities can be considered, such as retrieving top-k documents based on TFIDF when there are multiple patterns involved.

Acknowledgments

The authors would like to thank the anonymous reviewers for their careful reading and constructive comments. In particular, one of the open problems listed is suggested by our reviewers.

References

1. Bialynicka-Birula, I., Grossi, R.: Rank-Sensitive Data Structures. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 79–90. Springer, Heidelberg (2005)
2. Boyer, R.S., Moore, J.S.: A Fast String Searching Algorithm. Communications of the ACM 20(10), 762–772 (1977)
3. Hon, W.K., Shah, R., Vitter, J.S.: Ordered Pattern Matching: Towards Full-Text Retrieval. Technical Report TR-06-008, Department of CS, Purdue University (2006)
4. Karp, R.M., Rabin, M.O.: Efficient Randomized Pattern-Matching Algorithms. Technical Report TR-31-81, Aiken Computational Laboratory, Harvard University (1981)
5. Knuth, D.E., Morris, J.H., Pratt, V.B.: Fast Pattern Matching in Strings. SIAM Journal on Computing 6(2), 323–350 (1977)
6. Mäkinen, V., Navarro, G.: Position-Restricted Substring Searching. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 703–714. Springer, Heidelberg (2006)
7. Matias, Y., Muthukrishnan, S., Sahinalp, S.C., Ziv, J.: Augmenting Suffix Trees, with Applications. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 67–78. Springer, Heidelberg (1998)
8. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
9. Muthukrishnan, S.: Efficient Algorithms for Document Retrieval Problems. In: Proceedings of Symposium on Discrete Algorithms, pp. 657–666 (2002)
10. Sadakane, K.: Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In: Proceedings of Symposium on Discrete Algorithms, pp. 225–232 (2002)
11. 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)
12. Weiner, P.: Linear Pattern Matching Algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
13. Willard, D.E.: Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)
14. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers, Los Altos (1999)

Fast Single-Pass Construction of a Half-Inverted Index*

Marjan Celikik and Hannah Bast

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{mcelikik,bast}@mpi-inf.mpg.de

Abstract. We show how a half-inverted index can be constructed twice as fast as an ordinary inverted index. As shown in a series of recent works, the half-inverted index enables very fast prefix search, which in turn is the basis for very fast processing of many other types of advanced queries. Our construction algorithm is truly single-pass in that every posting (word occurrence) is touched (read and written) only once in the whole construction by avoiding an expensive merge of the index. The algorithm has been carefully engineered, with special attention paid to cache-efficiency and disk cost. We compared our algorithm against the state-of-the-art index construction from Zettair.

1 Introduction

The inverted index is still the standard indexing data structure for full-text search, and for good reasons: it can be stored in little space compared to the size of the original text (20% - 60%, or more, depending on whether positional information is considered or not [1,2]), it can be constructed fast, and it enables very fast full-text search. It has one major shortcoming however, in that only basic keyword queries (find all documents that contain all or some of the given query words) are supported efficiently.

In [3], an alternative index data structure, called HYB, was proposed that was shown to be as compressible as the inverted index, called INV in the following, but provides efficient support for a certain kind of *prefix search* (see Sect. 1.2 for an example). It was also shown in that and a series of subsequent works (see [4] for a summary), how the special kind of prefix search supported, allows fast processing of a large class of advanced queries, including: faceted search, query expansion with a large number of synonyms, error-tolerant search and semantic search. It should be noted that prefix search and all these advanced types of queries are notoriously hard for INV.

1.1 Our Contribution

The one question that was left open in these works was that of an efficient construction of the HYB index. The construction described in [3] actually works by first constructing and then post-processing an ordinary inverted index, yielding a total index construction for HYB that is about twice as much as that of INV.

* This work was partially supported by DFG-SPP 1307, project *Efficient Search in Very Large Text Collections, Databases, and Ontologies.*

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 194–205, 2009.
© Springer-Verlag Berlin Heidelberg 2009

In this paper we show that with careful algorithm engineering the *HYB can actually be constructed twice as fast as INV*. This is remarkable in two respects. First, because HYB is much more powerful than INV with regard to efficient support of advanced queries. Second, because fast index construction for INV has been the subject of extensive research, leaving with little room for further improvement of the state of the art (see Sect. 2).

As we will see, our index construction is truly single-pass in the sense that every posting is read only once from disk in the whole process. The best construction algorithms for INV claim to be single-pass, too, but that is only true in the sense that they make a single pass over the *original* text data, whereas later passes of already (inverted and) compressed versions of that data are not counted as additional passes.

1.2 The HYB Index

We briefly recapitulate from [3] what is necessary to know about the HYB index for this paper. Both documents and words have contiguous ids. Word-ids are assigned to words in lexicographical order; this is key for the fast processing of prefix queries with HYB. A posting for HYB is a quadruple of document id, word-id, position, and score. HYB then consists of so-called *blocks* of postings, sorted by document id and position (not by word-id). The blocks are defined by a sequence of *block boundary words* $w_0, \ldots, w_k$ such that block i contains all words in the range $(w_{i-1}, w_i]$, where w_0 is some word smaller than all words in the collection. One of the key results from [3] is that if these blocks are of roughly equal volume $\varepsilon \cdot n$, where n is the number of documents and ε is some constant, then HYB can be stored in space $1 + \varepsilon$ times that of INV. Here is an example of a block that corresponds to the word range (abl, abt]:

abl - abt	(doc ids)	D401	D1701	D1701	D1701	D1892
	(word-ids)	ablaze	abroad	abnormal	abnormality	abscess
	(positions)	5	3	12	54	4
	(scores)	0.3	0.7	0.4	0.3	0.1

In this example, the first list entry says that the word “ablaze” occurs in a document with id D401 at position 5 with a score of 0.3. The basic queries that the HYB index can efficiently compute are the so called *context-sensitive auto-completion* queries, or more informally all *completions* of the query words that would lead to good hits (including the matching documents and scores). These lists are computed instantly, with every letter being typed. For example, promising completions for the query *algo engin* might be *algorithm engineering*, *algorithmic engine*, *algorithm engineer* etc.[1]

1.3 Overview of Our Construction Algorithm

Our algorithm poses three major challenges. The first challenge is to compute the block boundaries, so that at parsing time we can determine the block to which a given word-id belongs. This could be trivially done by a full pass over the data, counting the frequency of each word and then computing the prefix

[1] An on-line application of the HYB index (DBLP bibliography search with extended capabilities) can be found at http://dblp.mpi-inf.mpg.de/dblp-mirror/

sums. Inspired by parallel sorting algorithms, in Sect. 3 we show how to compute very good estimates of the optimal block boundaries by sampling only a logarithmic number of random passages in the given document collection.

The second challenge is a cache-efficient and truly single-pass construction of the index that does not require index merging. We will show these two, even though not pointed out in previous work, can be efficiency bottlenecks of the construction. This is dealt with in Sect. 4.

Note that the word-ids are part of the postings and have to be stored in the index. Unlike the inverted index construction, this requires a permanent in-memory word to word-id mapping. Section 4 makes the simplifying assumption that the vocabulary fits in main memory. The final challenge is to get rid of this requirement. In Sect. 4.6 we propose refinement of our basic algorithm that addresses this issue.

In Sect. 5, we experimentally compare our construction against the very fast and well engineered state-of-the art inverted index construction of Zettair [5], which will be described in more detail in the next section. A more complete on-line version of this paper, including proofs for some of the lemmas can be found at www.mpi-inf.mpg.de/~bast/papers/hyb-index-construction.pdf

2 Related Work

An extensive amount of research has been done on static inverted index construction with many inversion approaches proposed, however only few scalable in practice [1]. We compare ourselves against the state-of-the-art inverted index construction proposed in [5] (referred to as the *single-pass approach*) which improves the well known *sort-based* approach, considered as one of the most efficient approaches described in the literature [5,1].

Both approaches are *in-memory* (opposed to the inferior disk-based approaches [6,5]), as the inversion is done in main memory, and *single-pass*, as only one pass over the uncompressed data is required (note that out definition of single-pass is stricter). Two-pass approaches on the other hand are slow but memory efficient since the number of postings per indexed word is known and hence the sizes of all in-memory and on-disk vectors can be easily calculated and effective compression schemes applied [5,1]. An extensive amount of collected work on inversion approaches can be found in [1].

The sort-based approach consists of the following basic steps: (i) word to word-id mapping is maintained through hashing and the available memory is filled with postings that come from the incoming parsing stream; (ii) when the main memory limit is reached, the postings are sorted, compressed and a new run is written to disk; (iii) after the whole collection has been processed multi-way merge is performed to obtain the final index. The merging can be performed either in-place, for additional index permuting cost [7,1], or with a temporary file roughly twice larger than the size of the index.

The single-pass approach from [5] modifies steps (i) and (ii) as follows. First, a dynamic bit-vector that accumulates the postings from the posting stream is assigned to every index word; and second, words instead of word-ids are included in the runs and thus no word to word-id mapping is required. The advantage of the modified version is that sorting of large amount of in-memory postings is avoided and that the vocabulary can be flushed out whenever a new run gets written on disk. The reported improvement ranges from 15% up to 20%.

3 Computing Block Boundaries

In this section we show how to compute block boundaries by only a logarithmic number of accesses to the given collection, an idea related to splitter selection in the parallel sorting literature (Samplesort, [8]). Let n be the collection size in total number of occurrences and let k be the number of HYB blocks. The sampling lemma below shows how to compute block boundaries from a sample of word occurrences so that the resulting blocks are of size less than $a \cdot \frac{n}{k}$ with high probability ($a > 1$ and n/k is the ideal block size). Note, however, that sampling from disk can be still expensive as one random access per word occurrence is required. We provide an alternative proof of the lemma with somewhat tighter upper bound that for the same failure probability permits close to twice smaller sample than that known in the literature.

Lemma 1 (Sampling Lemma). *Pick $s \cdot k$ numbers from the range $1..n$ uniformly at random and independently from each other. Sort these numbers and consider the k integers $x_1, ..., x_k$ whose rank in the sorted sequence is a multiple of s. Let $b_1, ..., b_k$ be the block sizes induced by splitting the range $1...n$ according to $x_1, ..., x_k$. Let $b_{max} = \min\{b_1, ..., b_k\}$. Then*

$$Pr(b_{max} > a \cdot n/k) \leq n \cdot exp(-s \cdot K) \tag{1}$$

where $K \approx a - ln(a) - 1$.

Proof. Call the $s \cdot k$ random numbers picked in the beginning splitters. Let the maximum block size be b_{max}. We consider the event that b_{max} is larger than some b. Then there must be a sub-range of size b that contains strictly less that s splitters. There are $n-b+1 \leq n$ such sub-ranges in total which means that $Pr(b_{max} > b) \leq n \cdot p$, where p is the probability that a fixed sub-range of size b contain less than s splitters. This probability is equal to $\sum_{i=0}^{s-1} \binom{sk}{i} (b/n)^i (1 - b/n)^{sk-i}$. We will derive an upper bound on the probability $p(s)$ that exactly s splitters fall into a fixed range of size b and from there derive Equation 1. After plugging $b = a \cdot n/k$ into $p(s)$ and applying the inequalities $\binom{sk}{s} \leq (ek)^s$, $1-x < \exp(-x)$; by simple transformations we obtain $p(s) \leq \exp((1 + \ln(a) - a \cdot (k-1)/k) \cdot s)$ which can be written as $\exp(-C \cdot s)$. This inequality is satisfied for all practical values of k (e.g. $k > 1000$), provided that $a > 1$. To complete the proof we will use the inequality $p \leq s \cdot p(s)$ provided that $p(s) \geq p(s-1) \geq ... \geq p(0)$. For binomial distribution the latter holds if s is no larger than the mode of the distribution M as $p(s)$ is maximized when $s = M$. In our case this condition is satisfied as $M = \lfloor sk \cdot b/n \rfloor \geq sk \cdot a/k = s \cdot a$, which is larger than s if $a > 1$. By plugging in the obtained inequality for $p(s)$ in the latter inequality we attain an upper bound on p that can be written as $\exp(-K \cdot s)$. Again, $K > 0$ for small values of $a > 1$ and all practical values of k which concludes the proof.

Lemma 2 (Query Time). *The expected HYB query processing time with perfect boundaries is asymptotically equal to that with boundaries computed according to Lemma 1 with high probability.*

4 Block Building

Given the sequence of block boundaries, a straightforward approach to build the HYB blocks with a single-pass would be as follows. Maintain a dynamically

growing in-memory data structure of postings for each block (e.g. linked lists), and for each word parsed, append the corresponding posting to the array of the block to which it belongs. When all words have been parsed, compress the blocks one after the other, and write them to disk.

Likewise the inverted index construction, the first obvious problem is that the size of the in-memory data structures will by far exceed the total available memory. An obvious solution is to process the blocks in runs by imposing a limit on their in-memory size. However, unlike the inverted index construction, our algorithm avoids a merge of the temporary runs by writing the partial in-memory blocks to their corresponding positions on the fly, without any fragmentation. This is dealt in more details in Sect. 4.1 and Sect. 4.4.

An efficiency issue not considered in the previous work of [5] is the cache efficiency of the in-memory inversion. Namely, even though well approximated by a Zipfian distribution and hence with a good locality of reference, a significant fraction of the word occurrences will impose cache misses when appended to their inverted lists (HYB blocks). This is due to the fact that the number of inverted lists is typically much larger than the number of L1-cache lines. We show in Sect. 4.2 and experimentally confirm in Sect. 5.2 that this can significantly affect the inversion performance.

Since the word-ids are part of the postings, they have to be compressed as well. Note, that the word-ids cannot be gap-encoded as they come in random order and have to be entropy-encoded instead. Near entropy-optimal but inefficient compression could be, for example, achieved by arithmetic encoding [1]. In Sect. 4.3 we propose a fast two-pass compression scheme for the price of only slightly worsen compression ratio.

4.1 Posting Accumulation

Once a dynamic in-memory array to each HYB block has been assigned, the postings from the posting stream are accumulated and compressed on the fly with Elias-gamma code for the doc-gaps, position-gaps and word-frequencies and *Zipf compression* (see Sect. 4.3) for the word-ids. An interesting observation from [9] shows that by maintaining the dynamic array's growth, additional saving can be achieved close to that when the array size is known in advance.

To maintain the word to word-id mapping, a fast hash-table based vocabulary [10] is employed (an alternative to permanent vocabulary is discussed in Sect. 4.6). As assigning lexicographic word-ids on the fly is not easy, a non-lexicographic to lexicographic word-id permutation (obtained by sorting the vocabulary) is stored at the end of the construction. To determine the corresponding HYB block for each posting, a fast in-memory data structure is employed that computes the HYB block and then stores this information in the word's vocabulary entry so that the computation is done only once per distinct word.

Note that on the one hand, unlike the single-pass approach, the HYB construction does not require sorting of words and on the other the word boundaries are already precomputed and in fixed (lexicographic) order. Moreover, the postings that correspond to a certain HYB block are already in doc-id order and require no sorting either. This means that overall our algorithm requires almost no sorting.

4.2 Cache Efficiency

To address the cache inefficiency of the in-memory inversion we propose a multilevel posting accumulation scheme by grouping consecutive HYB blocks in sets

so that the total number of sets is close to the number of cache lines of the L1-cache. The postings from each set are further recursively assigned to next level of sets until each set comprises a single HYB block. The number of cache misses will be minimized if the number of sets per level is equal, i.e. $\sqrt[l]{k}$ (l is the number of levels). The price paid for this procedure is multiple copies of each posting. Note, however, that a cache hit can be from 5 up to 100 times faster than a cache miss. The best results in practice were achieved for $l = 2$ (two-level posting accumulation).

Let us for simplicity assume a fully associative cache of size c and equally sized HYB blocks. Then regardless of the replacement policy, the expected number of cache misses for $l = 1$ and $l = 2$ respectively is $n \cdot (1 - c/k)$ and $2n \cdot (1 - c/\sqrt{k})^+$. The number of cache misses for $l = 2$ will be less than the number of cache misses for $l = 1$ if $(k - c)/(2\sqrt{k} \cdot (\sqrt{k} - c)^+) > 1$. This inequality is almost always satisfied if $k < 4 \cdot c^2$, with number of cache misses many times smaller for $\sqrt{k} \sim c$ and essentially 0 for $\sqrt{k} \leq c$. The latter is in fact a realistic scenario given that the number of blocks is typically less than 10,000 and that todays L1 caches are larger than 8 KB (a typical 8 KB L1 cache with 64 B cache lines has $c = 128$, where $\sqrt{k}$ is usually less than 100).

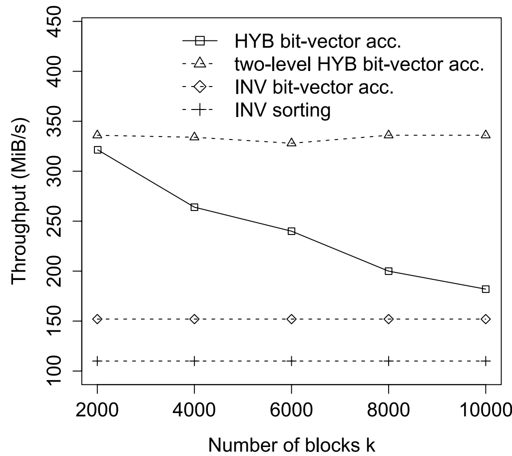


Fig. 1. Throughput (given in MiB/s) of HYB and inverted index posting accumulation approaches (defined in Sect. 4) to in-memory invert a run of 100 million occurrences

Figure 1 shows that the efficiency of the simple HYB block posting accumulation approaches that of the inverted index when the number of HYB block is large. The efficiency of two-level posting accumulation on the other hand is not affected. This is due to to the fact that the number of HYB sets remains smaller than the number of cache lines. Note that the above model even though simplified, matches Fig. 1.

4.3 Fast Word-id Compression

Our compression scheme is based on the assumption that the input (word-ids) have near Zipfian distribution. Given this, it is not hard to show that universal encoding [11] with $\sim \log x$ bits for number x of the ranks of the words sorted in order of descending frequency, is entropy optimal too. We refer to this scheme as *Zipf compression*. An obvious drawback here is that a full sort of a large number of word-ids is required to obtain the ranks. Assume that instead of sorting, the rank of each word is obtained by a MTF (move-to-front) transform over the input, with the intuition that frequent word-ids are likely to end up near the front of the list and thus get smaller ranks. This will allow us an efficient compression algorithm for the price of a very small loss in the compression ratio.

Lemma 3. *Given a Zipfian distribution of the input, the ranks obtained by a MTF-transform have expected values that are not far from the true ranks (determined by the skewness of the Zipfian distribution of the input).*

The loss in compression ratio on our two test collections in practice was surprisingly small: less than 1% on Wikipedia and about 1% on the TREC Terabyte (see Sect. 5.1). We note that this compression scheme is almost as fast as coding with gaps and Elias gamma code.

4.4 Writing the Blocks

Once the memory limit for the in-memory HYB blocks has been reached, each set of HYB blocks is decompressed and each individual HYB block restored, optionally re-compressed (this time the doc-ids are compressed with Golomb code) and written out on the fly to its corresponding position on disk. As the latter requires knowing the correct block sizes (in compressed format) in advance, we first compute an initial estimation for each block size by running an in-memory version of our algorithm on a random sample of documents. The problem of miss-estimated and potentially overflowing blocks is addressed in Sect. 4.5 by a procedure called *space-propagation*.

In the following we specify and compare the construction disk cost of the inverted and that of the HYB index. Let R_u be the cost to sequentially read the uncompressed collection, R_c be the cost to sequentially read the compressed temporary file, W_c be the cost to sequentially write the temporary compressed file, n be the total number of runs for both algorithms and t_s be the average disk seek time[2]. To reduce the number of disk seeks, a buffer of size B is allocated for each on-disk run. The total disk cost for the inverted index construction is then roughly equal to $R_u + W_c + \left(R_c + W_c + \max\left\{\frac{W_c}{B}, n\right\} \cdot t_s\right) + (R_c + W_c)$ where the second and the third word correspond to the cost to merge and permute the merged file. If twice more disk space than the size of the index file is allowed (merge is not *in-situ*), then the third term should be skipped (note that our approach does not require additional disk space, see Sect. 5.3). The total disk cost for the HYB construction is equal only to $R_u + (W_c + n \cdot k \cdot t_s)$. The second term in the brackets corresponds to the total seek time as each block requires a single disk seek for each run. Note that the above formula is pessimistic since it ignores the fact that the disk seek time depends on the track distance [12]. Our

[2] Note that the authors interchangeably use *disk seek time* and *disk access time*. In both cases we mean the full disk access time.

Table 1. Average random and equidistant seek time over 2000 disk seeks

File size	100 MB	1 GB	10 GB	100 GB
Average random seek	1.5 ms	5.2 ms	11.1 ms	14.5 ms
Average equidistant seek	0.8 ms	4.7 ms	5.6 ms	6.2 ms

algorithm makes equidistant jumps from one block to the adjacent, keeping this distance small e.g. not larger than 5 MB for a 10 GB collection. Table 1 shows this empirically (in the same time pointing out a slight inherent non-linearity of the disk cost, to some extent reflected in Table 2).

As an example, on a 50 GB collection (with 20 GB index), disk throughput of 50 MB/s with 5 ms seek time and a memory limit of 512 MB (= 40 runs), the disk cost for INV and HYB respectively is 3072 and 1833 secs. Note that writing the HYB blocks on the fly is possible since they are of roughly equal size and of reasonably large number. For the inverted index, where the number of inverted lists is in the order of millions, this is obviously impossible to achieve.

4.5 Space Propagation

We already mentioned the problem that blocks with initially underestimated sizes might overflow and overwrite the neighboring blocks. Note that no matter how good the initial estimation is, roughly half of the blocks will be initially underestimated. Also note that once the block writing started, further movement of the blocks is not possible. To address this problem we allow the underestimated blocks to make use of the space of the overestimated blocks, yet without splitting them in two parts. Below we give the basic description and provide theoretical evidence that the procedure fails with small probability given that certain assumptions are satisfied.

The idea is to "shift" the unused space of the overestimated blocks towards the underestimated by permitting an underestimated block to "borrow" some space from its neighbor. If the lender block is not large enough to fit its data and the data of the borrower in the same time, it becomes a borrower of the next neighbor. Hence a cascade of blocks borrowing space can be formed, stopping when blocks with enough free space to amortize for the propagated space demand are reached. To determine if a certain block will overflow, re-estimation of the block sizes takes place after significant fraction of the data has been processed (e.g. 70% - 80%). All blocks are initially sorted with respect to their initial size to increase the chance that neighboring blocks are of similar size and hence do not interrupt the propagation.

To provide theoretical evidence that supports this procedure we propose the following model. Let ϵ be the estimation error of each block size given in percents. Let assume that the initial estimated size of each block varies from its true size with Gaussian error with mean $\mu = 0$ and variance σ^2. Let the correct block size be B and let assume that $1 - r$ percents of the block size (equal to e.g. 70% - 80%) is enough to provide reliable estimation for the whole block. We also assume that there are enough runs so that the propagation "converges". Let consider k out of some larger number of blocks. In theory the propagation will fail if the total space demand of these blocks is larger than the available space, i.e. if $2 \cdot B \cdot r + \sum_{i=1}^{k} B \cdot \epsilon_i$ is negative, where the second term corresponds to

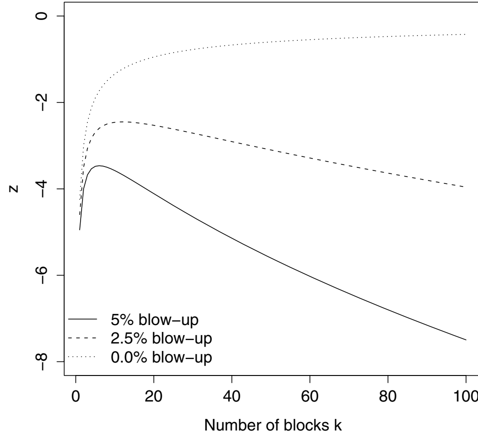


Fig. 2. Value of the z argument of the space propagation failure probability (defined in Sect. 4.5) for two directional propagation and 3 block space blow-up factors. (note that for z=-4, the propagation failure probability $p(z)$ is already less than 10^{-8}).

the available propagation space (either positive or negative) of the k blocks and the first term corresponds to the additional (borrowable) space of the first and the k-th block. To avoid propagation failure the above space should be positive, i.e. we are interested in $Pr(\sum_{i=1}^{k} \epsilon_i + 2 \cdot r < 0)$. Since $\sum_{i=1}^{k} \epsilon_i$ has Gaussian distribution with mean $k \cdot \mu$ and variance $k \cdot \sigma^2$, the latter probability can be written as $p(z) = \frac{1}{2}(1+\text{erf}(z(k)))$ where erf() is the Gaussian error function and $z(k) = \frac{-2r-k\mu}{\sigma\sqrt{2k}}$. Note that $p(z)$ is a strictly increasing function of $z(k)$, meaning that $p(z)$'s behaviour is solely determined by $z(k)$. Obviously if $\mu = 0$, then $z(k)$ strictly increases with k, resulting in a failure probability that approaches 1/2 for $k = 2000$ (see Fig. 2). However, if $\mu > 0$ (a small blow-up in the block sizes), then $z(k)$ starts to decrease for $k > \frac{2 \cdot r}{\mu}$ (~ 24), resulting in extremely small values of $p(z)$ for large k (e.g. 2000). Figure 2 plots the $z(k)$ value against the number of blocks k with different space blow-up factors.

4.6 Large Hash Keys

Our basic algorithm assumes that the entire vocabulary fits into main memory. Given that the vocabulary of TREC Terabyte takes roughly 600 MB, this is a realistic assumption. Still, for the case one wants to get rid of it, we propose the following refinement: instead of permanently storing (word, word-id) pairs, compute the word-ids with an additional hash function, making flushing to disk possible once the vocabulary size approaches the memory limit. At the end of the inversion the sorted vocabulary fractions[3] are merged into a final vocabulary (without fully reading them in memory). To avoid word-id collisions, by similar arguments as in the *birthday paradox*, one can show that by providing a universal family of hash-functions with large enough hash keys, the expected number of

[3] The vocabulary is sorted before flushing it to disk.

collisions can be kept below 1. Let V be the vocabulary size and $r = 2^{64}$ be the range size of the hash function (i.e. 64-bit hash keys). Then the expected number of collision is $\sum_{k=1}^{V} 1-((r-1)/r)^{k-1}$ which is equal to $V-r+r((r-1)/r)^{V}$, where the expression inside the sum is the probability that the hash of the k-th word collides with an earlier word. A simple calculation shows that the latter is less than 1 as long as V is smaller than ~ 6 billions (the TREC Terabyte collection has roughly 50 million distinct terms). The overhead imposed by this procedure in practice amounts to only about 10% of the total indexing time (see Table 3).

5 Experiments

We compare the performance of our HYB index construction algorithm to a state-of-the-art inverted index construction algorithm, namely the one implemented as part of Zettair[4], which essentially implements the ideas from [5], slightly varying from the original single-pass approach by using log-10 partitioning. According to the large study from [13], Zettair's index construction is indeed the fastest on the open source market to date.

All our code is written in C++ and compiled with GCC 4.1.2 with the -O3 flag. All experiments were performed on a machine with 16 GB of RAM (with contents flushed before every execution), 4 dual-core AMD Opteron 2.8 GHz processors (we used only one core at a time), operating in 32 bit mode and running Debian 4.1.1-19 with a standard ATA (Hitachi Deskstar) hard drive with 7200 RPM and reported average access time of 12.9 ms. We used the latest (0.9.3) version of Zettair.

5.1 Test Collections

Our experiments were carried out on two collections:

Wikipedia: A dump of the English Wikipedia, with a raw size of 12.7 GB, 2,874,500 million documents, 795 million word occurrences, and a vocabulary of 8 million distinct words.

TREC Terabyte: The TREC GOV2 corpus with a raw size of 426 GB, 25,204, 103 documents, 23 billion word occurrences, and a vocabulary of 57 million distinct words. To get a better picture on the scalability of both algorithms we run the experiments on subsets of size 25%, 50% and 100% of the full collection. To ensure that both parsers produce the same sequence of words we replaced all sequences of non-word characters in the collections by a single space each.

5.2 Index Construction Time

Compared to Zettair, our HYB index builder is faster by a factor of 1.6 (on Wikipedia) to 2.1 (on Terabyte). For both algorithms, we imposed a memory limit of 500 MB for the in-memory posting accumulation (we used the `--big-and-fast` option on Zettair) without taking into account the additional memory usage of the vocabulary and other auxiliary data structures. In all experiments we picked the HYB block sizes as $N/5$, where N here is the number of documents in the collection (see [3]).

[4] http://www.seg.rmit.edu.au/zettair/

Table 2. Elapsed index construction time in minutes to construct a word-level index with our HYB builder and Zettair on all our test collections

	Wikipedia	TREC Terabyte 25%	TREC Terabyte 50%	TREC Terabyte 100%
HYB	4.7 min	43.0 min	75.1 min	155.8 min
Zettair	7.3 min	92.3 min	146.4 min	288.9 min

Table 3. Increase in the HYB construction time on one quarter of the TREC Terabyte collection when different in-memory vocabulary size limits are used (see Sect. 4.6). The total number of distinct words was roughly 28.5 millions and the construction (without vocabulary limit) took 43 mins.

Vocabulary size limit in %	50%	33%	25%	20%
Increase in construction time	+7%	+10%	+12%	+11%

Table 4. Break-down of the total elapsed index construction time in four major steps for the TREC Terabyte collection

Parse & Look-up	Accumulate	Compress	Disk I/O	Sampling
32%	16%	20%	29%	3%

Table 4 shows a break-down of the running time of our HYB builder. The most expensive phase is the parsing and looking-up of word-ids which takes roughly one third of the total cost. While Zettair spends one third of its time merging runs, our index builder spends only one fifth of its time on block writing. This is due to the fact that Zettair has to fully read and fully write the temporary index file more than once. We note that the actual bottleneck in the index merging are not the disk seeks but rather reading and writing large amount of data. As a consequence, the index partitioning improvement proposed in the last section of [5] (aiming to reduce the number of disk seeks) showed only marginal speed-up in practice.

Only about 16% of the time of our index construction is spent on posting accumulation, that is, in-memory inversion. As suggested by both, Fig. 1 and the figures shown in [5], this cost for the inverted index is more than twice higher. The latter points out that cache misses seriously affect the efficiency of the in-memory inversion of the single-pass approach. For whatever reason, posting accumulation costs are not reported in the elapsed-time figures of [5].

5.3 Temporary Disk Space

The additional disk requirement of our HYB construction is very small and comes from the overflowed blocks. The total size of the overflowed space depends on the quality of the initial estimation of the block sizes (see Sect. 4.4). The peak disk space usage in our experiments varied from 100% of the size of the final index

file on small to medium-size collections, up to 103% on the full TREC Terabyte (i.e. up to 3% overhead).

The reported additional temporary disk usage in [5] is about 26% for document-level inverted index and about 8% for word-level inverted index. However, we found out that during the parsing and merging phase the peak space usage of Zettair on the TREC Terabyte was correspondingly about 165% and about 143% of the final size of the index file. We note that in a setting where the input data is very large and streamed, using significantly more space than the final index might be undesirable.

6 Conclusions

We have carefully designed and engineered a construction algorithm for the powerful HYB index from [3], that is twice as fast as the state-of-the-art inverted index construction from Zettair. Our algorithm is simple to implement, and, unlike for an inverted index, does not require additional data structures for external sorting. Our approach is truly single-pass in that the bulk of the word occurrences are touched only once each, it is cache-efficient, does not require in-memory or external sorting and during construction uses no more space than the final compressed index.

References

1. Witten, I.H., Moffat, A., Bell, T.C.: Managing gigabytes: Compressing and indexing documents and images (1999)
2. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. (2006)
3. Holger Bast, I.W.: Type less, find more: fast autocompletion search with a succinct index. In: SIGIR (2006)
4. Bast, H., Weber, I.: The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration. In: CIDR (2007)
5. Heinz, S., Zobel, J.: Efficient single-pass index construction for text databases. Jour. of the American Society for Information Science and Technology (2003)
6. Rogers, W., Gerald, C, Harman, D.: Space and time improvements for indexing in information retrieval. In: Proceedings of 4th Annual Symposium on Document Analysis and Information Retrieval (1995)
7. Moffat, A., Bell, T.A.H. In situ generation of compressed inverted files. Journal of the American Society for Information Science (1995)
8. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Reading (2003)
9. Buttcher, S., Clarke, C.L.A.: Memory management strategies for single-pass index construction in text retrieval systems. Technical report, School of Computer Science, University of Waterloo, Canada (2005)
10. Heinz, S., Zobel, J.: Performance of data structure for small sets of strings. In: Proc. of the Australasian conference on Computer Science (2002)
11. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst. (1996)
12. Popovici, F.I., Arpaci-dusseau, A.C., Arpaci-dusseau, R.H.: Robust, portable i/o scheduling with the disk mimic. In: USENIX Annual Technical Conference (2003)
13. Middleton, C., Baeza-Yates, R.: A comparison of open source search engines (2007), http://wrg.upf.edu/WRG/dctos/Middleton-Baeza.pdf

Two-Dimensional Distributed Inverted Files*

Esteban Feuerstein[1], Mauricio Marin[2], Michel Mizrahi[1], Veronica Gil-Costa[2], and Ricardo Baeza-Yates[2]

[1] Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina
[2] Yahoo! Research Latin America, Santiago, Chile

Abstract. Term-partitioned indexes are generally inefficient for the evaluation of conjunctive queries, as they require the communication of long posting lists. On the other side, document-partitioned indexes incur in excessive overheads as the evaluation of every query involves the participation of all the processors, therefore their scalability is not adequate for real systems. We propose to arrange a set of processors in a two-dimensional array, applying term-partitioning at row level and document-partitioning at column level. Choosing the adequate number of rows and columns given the available number of processors, together with the selection of the proper ways of partitioning the index over that topology is the subject of this paper.

1 Introduction

Inverted files [2] are used as index data structures to efficiently solve queries upon huge text collections. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the text collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get the set of documents associated with the query terms and then perform a ranking of these documents in order to select the top-K documents as the query answer. From the literature we can learn of a number of methods for distributing the inverted file onto P processors or computers and their respective query processing strategies [1,4,6,9,10,11,14,18]. Distributing an index consists of splitting the document collection and/or the index itself among the computers. There are different ways of doing this splitting, mainly variants of two basic dual approaches: document-based partition (a.k.a local indexes) and term-based partition (a.k.a global indexes). Variants of these two basic schemes have been proposed in [7,13,15].

The ranking of documents can be performed upon either intersection or union of posting lists. For queries requiring such intersection, the global indexes tend to be inefficient as they require sending complete posting lists among processors. To alleviate this problem, it has been proposed to concentrate together in the same

* This research was funded by a Yahoo! Research Alliance Grant.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 206–213, 2009.
© Springer-Verlag Berlin Heidelberg 2009

processor terms that usually appear together in queries, reducing the probability of having to transfer posting lists. Different methods have been proposed in [5,16,17]. On the other side, when using a local index, document IDs are assigned to unique processors and thereby the intersection of posting lists does not require communication. However, all processors must participate in the evaluation of each query: as the number of processors grows, the overhead associated with each query grows linearly, so the improvement in the throughput is not proportional to the number of processors. The reason is that each active query being processed is replicated P times on the P processors and, in each processor, they demand the use of hardware resources which do not come for free in terms of latency.

A natural idea to overcome the problems of these two approaches is to use a two-dimensional scheme trying to benefit from the advantages of the two extreme distributed indexes. The idea is to arrange a set of $P = R \times C$ processors as a matrix of R rows and C columns, applying term-partition at row level and document-partition at column level. In few words, the document collection is partitioned in C sub-collections, each of which is allocated to a "column" of R processors, which will hold the index of that collection in a term-based partitioning. The point for conjunctive queries is the following: for any concrete policy used to partition and group terms, the probability of co-residence of a pair of terms of a query increases as the number of processors decreases, so the communication cost tends to decrease with the number of rows of the matrix (the optimal being an arrangement of one single row and P columns, that is a normal document-partitioned system). At the same time, when the number of columns increases, so does the overhead typical of local indexes, so one can expect that there is an optimal configuration somewhere in between the extreme approaches. The proposal of this paper is, therefore, to analyze the performance of different configurations for a fixed number P of processors, ranging from P rows and one column (term partition) to one row and P columns.

2 A Two-Dimensional Partitioning Index

The processors form a two-dimensional array of R rows and C columns; in one of the dimensions (the rows) the index is seen as partitioned by terms, in the other dimension (columns) as partitioned by documents. The document collection is partitioned therefore in C sub-collections, each of which is allocated to a "column" of R processors, which will hold the index of that collection with a term-based partitioning. This two-dimensional scheme brings about different ways of evaluating a query. The one studied in this paper is to first distribute the query among the columns (the processors that contain query terms in each column) as with a local index. Then, at each column, the intersection must be resolved as in a global index by invoking the processors of the column that hold the required terms, and finally merging the results obtained at each column.

In the case $R = 1, C = P$ (document partitioned index), each processor holds the posting lists of the whole set of terms appearing in the documents assigned to it. Conversely, when the index is term partitioned ($R = P, C = 1$), the documents

are considered as an indivisible package. As soon as we leave these extreme cases to consider a 2D scheme with more than one row and more than one column, the need appears of dealing simultaneously with term and document partitions. The question that arises is: once that C and R are fixed, how the two kind of partitions can be optimized? This question regards not only which technique or criteria is used to optimize each of them but, and this is a novelty specific to the two-dimensional partitioning of the index, how the two partitions are combined. For example, the partition of the terms could be done independently of how the documents will be partitioned. Or the terms could be partitioned taking into account the information of the document partition. Also we could partition first the terms and then the documents, and many other possibilities. To make things more complex, in each of those schemes one can use different algorithms for term and document partition, yielding an enormous amount of possibilities. There is a wide literature regarding how these partitions can be optimized (see for example [5,7,12,13,15,16]). The different trade-offs must be evaluated upon a baseline cost model which we develop in the following. In Section 3 we describe the particular algorithms we used to partition both terms and documents.

Basic Cost. The processing of a query can be decomposed in a series of operations that are executed in different processors. These are the primitive operations such as broadcast or communication, list intersection, merging, ranking, etc. Each of these operations has a cost, and their sum conforms the computation and communication cost of a query. In addition, each processor incurs in a certain overhead due to hardware use, network access and system scheduling tasks among others. The weight of these overheads in the total cost turns out to be high, so it cannot be neglected. In a local index the number of participating processors per query is much greater than in the global scheme.

In the following we will assume that a certain number q of queries are initially presented at every processor and then new queries arrive as the system delivers answers for previous queries. So, at every moment there are $q * P$ live queries in the system. In that framework, providing that a good load balance is obtained, we can assume that the whole set of P processors can work in parallel, and there will not be idle times. To simplify, from now on we will consider only two-term queries of the form $t_1 \wedge t_2$. We will use the following notation:

- $t_i(x, y)$: Expected time employed by a processor to compute the intersection of two lists of lengths x and y respectively.
- $t_m(x)$: Expected time employed by a processor to merge a set of lists of total length x.
- $t_r(x)$: Expected time employed by a processor to rank a list of length x.
- $I(x, y)$: the expected length of the intersection of two lists of length x and y.
- γ: time employed to transmit a unit of information from one processor to another.

Let ℓ be the expected length of a posting list (considering all the files of the system). We will assume that to prepare a result list of K results using a local

index distributed among P processor, each processor will send to the originator of the query its best $2K/P$ postings and that the $2K$ results obtained that way are, with high probability, enough to answer the query. Was that not the case, another extra request would be generated for a subset of the processors, but we ignore that in this paper.

Local Index. The sequence of tasks performed *in parallel at each of the P processors* for a set of q queries, and their corresponding costs, can be described as follows:

Action	**Cost**
Broadcast the q queries of each processor to all other processors	$q(P-1)\gamma$
For each query two lists of expected length ℓ/P are intersected	$qPt_i(\frac{\ell}{P}, \frac{\ell}{P})$
For each of the $q * P$ queries, the resulting lists are ranked	$qPt_r(I(\frac{\ell}{P}, \frac{\ell}{P}))$
For each query, send $\frac{2k}{P}$ results to the originator of the query	$qP\frac{2k}{P}\gamma$
For each query originated at that proc., merge the P lists received	$qt_m(2K)$

Global Index. Let ℓ_{min} be the expected length of the shortest among the two posting lists of the terms of a query. Let $\alpha(X)$ be the probability of co-residence of the two terms of a query given that the terms are partitioned in X processors. With probability $(1-\alpha(P))$ the query should be distributed among two processors, so we need to broadcast the two terms to the two processors holding them, and the processor holding the shortest among the two lists send it to the other one. With probability $\alpha(P)$ the two terms are co-resident in one processor, so the query must just be sent to it. In both cases, the processing is completed by intersecting the two lists, ranking the result and sending the best K elements to the originator of the query. All this can be summarized in the following table. Recall that we are assuming that q queries are submitted to each processor, so the probabilities $\alpha(P)$ and $(1-\alpha(P))$ can be interpreted as fractions of the total number of queries.

Action	**Cost**
(Non co-residence) Send the terms to their two processors	$(1-\alpha(P))q2\gamma$
(Non co-residence) The shortest list is sent to the other processor	$(1-\alpha(P))q\gamma\ell_{min}$
(Co-residence) Send the two terms to one processor	$\alpha(P)q\gamma$
Intersect the two lists l	$qt_i(\ell,\ell)$
Rank the resulting list	$qt_r(I(\ell,\ell))$
The best K elements of the resulting list are sent to the originator	$qK\gamma$

2D Index. We will analyze this model assuming we have R rows and $C = P/R$ columns. The sequence of tasks to be developed at each processor (always assuming q queries per processor) starts with the broadcast of the q queries to each of the C columns (to a random processor at the column). The R processors of each column must then resolve a total of qP queries, so each one of them will hold expectedly $\frac{qP}{R} = qC$ queries. So this part will be executed in parallel by the C columns, and within each column by the R processors of the column, therefore we can think that the P processors are working in parallel. In each column the terms may be co-resident at the same processor (row) or not, with

probabilities respectively $\alpha(R)$ and $(1-\alpha(R))$, so different tasks will be executed for the corresponding fraction of the queries. After that, always at column level, but with the C columns working in parallel, intersection and ranking of the lists ($q * C$ queries at each processor). Finally, each column (actually, the processor in the column that has computed and ranked the intersection) must send its results to the originator of the query, that will merge the results.

Action	**Cost**
Broadcast the q queries to a random processor in each column	$qC\gamma$
(Non co-residence) Send the two terms and then send the shortest list from one processor to the other one	$(1-\alpha(R))*$ $(qC2\gamma + qC\gamma\ell_{min})$
(Co-residence) Send the two terms to their processors	$\alpha(R)qC\gamma$
$q * C$ intersections of lists of expected length ℓ/C	$qCt_i(\ell/C)$
$q * C$ rankings of the lists at each processor	$qCt_r(I(\ell/C, \ell/C))$
For each of the qC queries, send $\frac{2K}{C}$ results to its originator	$qC\frac{2K}{C}\gamma$
Merge the C lists of length $2K/C$ received in each processor	$qt_m(2K)$

Overhead. To compute the real cost associated with a query we have to add to the expressions developed in the previous section a fixed cost or overhead (that we will denote as β). This will be counted for every processor participating in a query. In a local index each query will have an overhead of $P * \beta$. In a global index the terms may be co-resident or not at each column, so the overhead may be seen as a random variable with expected value $(\alpha(P) + (1 - \alpha(P)) * 2) * \beta$. Finally, in the general 2D case with R rows, one or two processors participate at each column so the expected value of the overhead is $C*(\alpha(R)+(1-\alpha(R))*2)*\beta$.

3 Experimental Evaluation

For term partition we used a term-clustering heuristic oriented to reducing communication cost and at the same time maintaining the load balance of the system. This heuristic, based on the one used in [8], tries to assign to the same machine pairs of terms of high cost (a function of the relative frequency and length of the shortest posting lists of its terms). We will refer to this heuristic as TCH. The basic heuristic that we used for document clustering tries to group similar documents (cosine measure) and assign them to the same processor. It starts with a certain number of documents that are chosen initially as cluster centers. These cluster centers are selected so that they are sufficiently different from each other. Then we insert into each cluster the documents that are closer to each cluster center. Finally, the clusters are assigned to the different machines in a round-robin fashion [3]. We will refer to this heuristic as DCH. For document partitioning we also consider a simple Random partition (DRH). The first two-dimensional heuristics we considered were to partition terms and documents independently, using TCH for terms and either DRH or DCH for the documents. We will refer to these heuristics as 1.a and 1.b respectively.

Another family of heuristics consists in partitioning first the documents, using either DRH or DCH, and then the terms using TCH separately for each column,

taking into account the documents that were assigned to each column (heuristics 2.a and 2.b respectively). Also by first partitioning documents we could use the information of that partition to produce one single partition for the terms to be used across all the columns. These heuristics will be referred to as 3.a (with DRH) and 3.b (with DCH).

A different approach may be taken if we first partition the terms and then the documents. Given as input an initial partition of the terms, the heuristic 4 tries to distribute the documents among the columns so as to minimize the communication cost. We consider only pairs of non co-resident terms (as co-resident pairs will not require further communication). The intuition behind the heuristic is to try to minimize in two ways the lengths of the posting lists that must be transfered: (a) separating in different columns documents that *are not* part of the intersection of popular pairs, and (b) minimizing the length of the shortest posting list at each column by increasing the variance of the lengths of the lists. The (last) heuristic 5 constructs the partitions of terms and documents simultaneously, considering pairs of queries one by one, in decreasing order of cost. For each query, it decides whether to consider it to group together its terms, or to separate the documents of their posting lists.

The final expression for the cost *of a single query* will be obtained by considering the computation and communication costs plus the overhead incurred by every participating processor. For that, we need to adapt the values given in Section 2 to an individual query instead of a set of q queries, getting a per-query cost of:

$$C\gamma + (1-\alpha(R))(C2\gamma + C\ell_{min}\gamma + 2C\beta) + \alpha(R)(C\gamma + C\beta) + \quad (1)$$
$$Ct_i(\ell/C) + Ct_r(I(\ell/C, \ell/C)) + C\frac{2K}{C}\gamma + t_m(2K)$$

This formula is valid for the case in which the term partition is uniform across all the columns (i.e. the two terms of a query are assigned to the same row at each column), and therefore are co-resident or not uniformly in all the columns.

We did our experiments on two inputs: Collection 1 is a sample of the Chilean web with $\approx 160K$ documents, Collection 2 contains a subset of $\approx 2M$ documents of a 1.5TB sample of the UK's web and $\approx 250K$ queries taken from a one-year log of a major search engine's site. We simulated and measured the performance of every heuristic with different configurations on $P = 256$ processors. The number of rows ranged from 1 (local indexing) to 256(global-indexing) using successive powers of two.

For the simulation we defined particular costs for the different primitive functions, based on benchmarking runs we did on the same collections. The values are expressed relative to a base-line in terms of ranking time defined as $t_r(x) = x$. Intersection and merge operations require in average $t_i(x, y) = \min(x\log(y), x + y)/4$ and $t_m(x) = x/4$ respectively. The values for β and γ were chosen to achieve proper agreement with what we have observed using two actual implementations of document- and term-partitioned inverted files for disjunctive queries. We run experiments on the two indexes, in which the pure global index resulted on average 20% more efficient than the pure local one, so the values for β and γ were chosen so as to satisfy that relation.

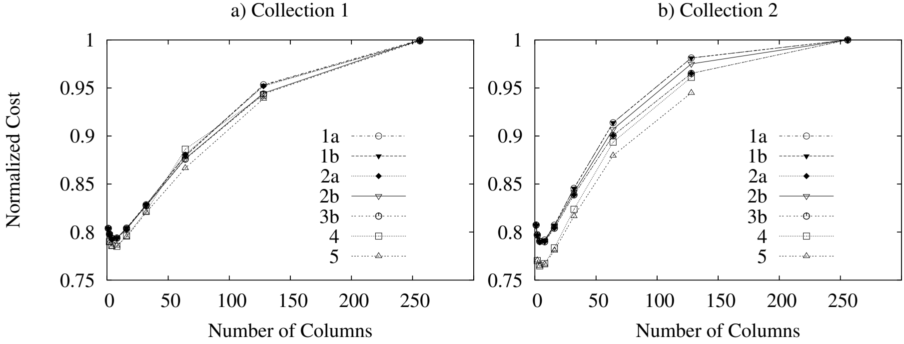


Fig. 1. Normalized costs as a function of number of columns, for different heuristics

The graphics in figure 1 summarizes the results of our simulation. It shows total costs (processing+communication+overhead) as a function of the number of columns, for the two collections. All the costs were normalized by dividing them by the maximum cost, that occurs for all the heuristics when the number of columns is 256 (i.e. when the 2D index becomes a simple local index). We observe that an important improvement in the cost is achieved by arranging the 256 processors in a two-dimensional array, of 8×32 or 4×64, for all heuristics. Therefore, the main claim that an improvement can be obtained with a 2D index against the classical local and global indexes is verified.

It may be observed that there is not a big difference in the performances of the heuristics, although heuristics 4 and 5 behave consistently better than the others in almost all configurations (the latter being a bit better in general). Note that heuristics 4 and 5 cannot be applied for simple local and global indexes. These seem to be the only heuristics that take advantage of the two-dimensional structure and the possibility of combining clustering techniques for terms and documents. The results shown in the figure were computed for particular values of the parameters β and γ. The difference between the best and worst configurations is of more than 20%.

4 Conclusions and Further Work

The preliminary results obtained in our simulations are a positive signal towards the continuation of our study in that direction. An immediate task we have to focus on is the realization of further and deeper experiments, with real executions in real environments, with larger document collections and query logs. Those experiments should include the usage of different total number P of processors.

An interesting subject of further research regards the possibility of dynamically reconfiguring the arrangement of the processors to adapt to different types of queries, and also the use of non rectangular arrangements (rows or columns of different length). Finally, we plan to analyze how do different ranking policies at row and column level may affect the performance of the system.

References

1. Badue, C., Baeza-Yates, R., Ribeiro, B., Ziviani, N.: Distributed query processing using partitioned inverted files. In: SPIRE (2001)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval
3. Costa, G.V., Marin, M., Reyes, N.: Parallel query processing on distributed clustering indexes. Journal of Discrete Algorithms (7) , 03–17 (2009)
4. Jeong, B.S., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. IEEE Trans. Parallel and Distributed Systems 16(2), 142–153 (1995)
5. Lucchese, C., Orlando, S., Perego, R., Silvestri, F.: Mining query logs to optimize index partitioning in parallel web search engines. In: INFOSCALE (2007)
6. MacFarlane, A.A., McCann, J.A., Robertson, S.E.: Parallel search using partitioned inverted files. In: SPIRE (2000)
7. Marin, M., Costa, G.V.: High-performance distributed inverted files. In: CIKM 2007 (2007)
8. Marin, M., Gomez-Pantoja, C., Gonzalez, S., Gil-Costa, V.: Scheduling Intersection Queries in Term Partitioned Inverted Files. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 434–443. Springer, Heidelberg (2008)
9. Moffat, A., Webber, W., Zobel, J., Baeza-Yates, R.: A pipelined architecture for distributed text query evaluation. Information Retrieval 10(3), 205–231 (2007)
10. Ribeiro-Neto, B.A., Barbosa, R.A.: Query performance for tightly coupled distributed digital libraries. In: ACM Conf. Digital Libraries, pp. 182–190 (1998)
11. Stanfill, C.: Partitioned posting files: a parallel inverted file structure for information retrieval. In: SIGIR (1990)
12. Suel, T., Mathur, C., Wu, J.W., Zhang, J., Delis, A., Kharrazi, M., Long, X., Shanmugasundaram, K.: ODISSEA: A peer-to-peer architecture for scalable web search and information retrieval. In: WWW 2003 (2003)
13. Tang, C., Dwarkadas, S.: Hybrid global-local indexing for efficient peer-to-peer information retrieval. In: NSDI (2004)
14. Tomasic, A., García-Molina, H.: Performance issues in distributed shared-nothing information-retrieval systems. Information Processing & Management 32(6), 647–665 (1996)
15. Xi, W., Sornil, O., Luo, M., Fox, E.A.: Hybrid partition inverted files: Experimental validation. In: Agosti, M., Thanos, C. (eds.) ECDL 2002, vol. 2458, p. 422. Springer, Heidelberg (2002)
16. Zhang, J., Suel, T.: Optimized inverted list assignment in distributed search engine architectures. In: IEEE IPDPS 2007(2007)
17. Zhong, M., Shen, K., Seiferas, J.I.: Correlation-aware object placement for multi-object operations. In: ICDCS 2008, pp. 512–521 (2008)
18. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys 38(2) (2006)

Indexing Variable Length Substrings for Exact and Approximate Matching

Gonzalo Navarro[1,⋆] and Leena Salmela[2]

[1] Department of Computer Science, University of Chile
gnavarro@dcc.uchile.cl
[2] Department of Computer Science and Engineering,
Helsinki University of Technology
lsalmela@cs.hut.fi

Abstract. We introduce two new index structures based on the q-gram index. The new structures index substrings of variable length instead of q-grams of fixed length. For both of the new indexes, we present a method based on the suffix tree to efficiently choose the indexed substrings so that each of them occurs almost equally frequently in the text. Our experiments show that the resulting indexes are up to 40 % faster than the q-gram index when they use the same space.

1 Introduction

We consider indexing a text for exact and approximate string matching. Given a text $T = t_1 t_2 \ldots t_n$, a pattern $P = p_1 p_2 \ldots p_m$, and an integer k, the approximate string matching problem is to find all substrings of the text such that the edit distance between the substrings and the pattern is at most k. The edit distance of two strings is the minimum number of character insertions, deletions, and substitutions needed to transform one string into the other. We treat exact string matching as a subcase of approximate string matching with $k = 0$.

Partitioning into exact search is a popular technique for approximate string matching both in the online case [1,2,11,13], where the text is not preprocessed, and in indexing approaches [3,4,7,10,12], where an index of the text is built. Suppose that the edit distance between two strings, S and R, is at most k. If we split S into $k+1$ pieces, then at least one piece must have an exact occurrence in R. In the online approach, we thus split the pattern into $k+1$ pieces, search for all the pieces in the text, and verify all the matches found for approximate occurrences of the whole pattern using a dynamic programming algorithm that runs in $\mathcal{O}(m^2)$ time per verification. In the indexing approach we have two options. If we index all text positions, we can proceed as in the online case: split the pattern into $k+1$ pieces, search for all the pieces in the index, and verify all the matches found. Another option is to index the text at fixed intervals. Now we search for all pattern substrings of corresponding length in the index and verify

⋆ Supported in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 214–221, 2009.
© Springer-Verlag Berlin Heidelberg 2009

1 2 3 4 5 6 7 8 9 0 1
a a a b a a b b a a \$

2-gram	positions	difference coded positions
aa	1, 2, 5, 9	1, 1, 3, 4
ab	3, 6	3, 3
$a\$$	10	10
ba	4, 8	4, 4
bb	7	7

Fig. 1. The 2-grams and the 2-gram index of the text $T =$ "aaabaabbaa\$"

the matches found to obtain the approximate occurrences of the whole pattern. The q-gram index of Navarro and Baeza-Yates [7] takes the former approach, while the q-samples index of Sutinen and Tarhio [12] utilizes the latter technique.

A problem of the q-gram index is that some q-grams may be much more frequent than others, which raises verification costs. A technique to choose the $k+1$ optimal pieces [7] was designed to alleviate this problem. In this work we develop two new indexes, the prefix free and the prefix coalesced index, based on the q-gram index [7]. The new indexes index substrings of variable length instead of q-grams of fixed length. The goal is to achieve roughly similar lengths in all position lists. In the prefix free index the set of indexed substrings forms a prefix free set, whereas in the prefix coalesced index this restriction is lifted. The experimental results show that the new indexes are up to 40 % faster than the q-gram index for the same space. Alternatively, the new indexes achieve as good search times as the q-gram index using less space. For example, when $m = 20$ and $k = 2$ the new indexes are as fast as the q-gram index using 30 % less space.

2 q-Gram Index

In this section we review previous work on the q-gram index [7], which indexes all q-grams of the text and uses partitioning into exact search to locate occurrences of a pattern. The value of q is fixed at construction time, and the q-grams that occur in the text form the vocabulary of the index. Together with each q-gram the index stores a list of positions where the q-gram occurs in the text. To save space the position lists are difference coded and compressed with variable length integer coding. The q-gram index can be built in $\mathcal{O}(n)$ time [7]. Figure 1 shows the 2-grams of the text "aaabaabbaa\$" and the corresponding 2-gram index.

To search for a pattern P with at most k differences, we first extract $k+1$ non-overlapping q-grams from the pattern. We then search for all these q-grams in the index and finally use dynamic programming to verify the positions returned by this search. For example, to search for the pattern $P =$ "abbab" with at most $k =$ 1 difference in the 2-gram index of Fig. 1, we first extract two non-overlapping 2-grams from the pattern: "ab" and "ba". Search on the index returns positions 3 and 6 for "ab" and positions 4 and 8 for "ba". Verifying around these positions we obtain the occurrences starting at positions 3 and 6.

In real texts some q-grams occur much more frequently than others, and if the pattern is longer than $(k+1)q$, we have several different ways of choosing the $k+1$

```
1.  for (i = 1; i ≤ m; i++)
2.      P[i,0] = R[i,m+1]
3.      C[i,0] = m+1
4.  for(r = 1; r ≤ k; r++)
5.      for(i = 1; i ≤ m−r; i++)
6.          P[i,r] = min(R[i,j] + P[j,r−1] | i < j ≤ m−r+1)
7.          C[i,r] = j that minimizes the above expression
```

Fig. 2. The dynamic programming algorithm for the optimal partitioning of the pattern

non-overlapping q-grams. Therefore we can speed up verification considerably by choosing the q-grams carefully. Furthermore, the pieces do not need to have the exact length q. If a piece is shorter than q, we find all q-grams starting with the piece in the index and verify all their positions. If the piece is longer than q, we search for the first q-gram of the piece in the index. By allowing pieces shorter than q, we can also search for patterns shorter than $(k+1)q$.

Navarro and Baeza-Yates [7] give the following method for finding the optimal partitioning of the pattern. It is relatively fast to compute the number of verifications a pattern piece will trigger. We use binary search to locate the q-gram(s) in the index and obtain a contiguous region of q-grams. If we store the accumulated list lengths, the number of verifications can easily be calculated by subtracting the accumulated list lengths at the endpoints of the region. By performing this search for all pattern pieces, we obtain a table R where $R[i,j]$ gives the number of verifications for the pattern piece $p_i \dots p_{j-1}$. Based on this table, we use dynamic programming to compute the table $P[i,k]$, which gives the total number of triggered verifications for the best partitioning for $p_i \dots p_m$ with k differences, and the table $C[i,k]$, which gives the position where the next piece starts in order to get $P[i,k]$. We then find the smallest entry $P[\ell_0,k]$ for $1 \le \ell_0 \le m-k$, which gives the final number of verifications. The pattern pieces that give this optimal number of verifications begin at $\ell_0, \ell_1 = C[\ell_0,k], \ell_2 = C[\ell_1,k-1] \dots \ell_k = C[\ell_{k-1},1]$. Figure 2 gives the pseudo code for the dynamic programming algorithm to find the optimal partitioning of the pattern. It runs in $\mathcal{O}(km^2)$ time, whereas R is easily built in $\mathcal{O}(qm \log n)$ time, which can be reduced to $\mathcal{O}(qm \log \sigma)$, where σ is the alphabet size, if the q-gram vocabulary is stored in trie form.

3 Prefix Free Index

Our new variants of the q-gram index index substrings of varying length instead of q-grams of fixed length. The indexed substrings form the vocabulary of the index. The aim is to choose the vocabulary so that each position of the text is indexed and the lengths of the position lists are as uniform as possible. In the first variant, the prefix free index, we further require that the vocabulary is a prefix free set, i.e. no indexed substring is a prefix of another indexed substring.

Let α be the threshold frequency and let the frequency of a string be the number of occurrences it has in the text T. Note that the frequency of the empty

substring	positions	difference coded positions
aaa	1	1
aab	$2,5$	$2,3$
$aa\$$	9	9
ab	$3,6$	$3,3$
$a\$$	10	10
b	$4,7,8$	$4,3,1$

(a) Prefix free index

substring	positions	difference coded positions
aa	$1,9$	$1,8$
aab	$2,5$	$2,3$
ab	$3,6$	$3,3$
$a\$$	10	10
ba	$4,8$	$4,4$
bb	7	7

(b) Prefix coalesced index

Fig. 3. A prefix free index and a prefix coalesced index for the text $T =$ "aaabaabbaa$"

string is n. The vocabulary now consists of all such substrings $S = s_1 \ldots s_i$ of the text that the frequency of S is at most α and the frequency of the prefix $s_1 \ldots s_{i-1}$ is greater than α. This choice ensures that the vocabulary is a prefix free set and no position list is longer than α. Again the position lists are difference coded and compressed with variable length integer coding. Figure 3(a) shows an example of a prefix free index with threshold frequency three.

To search for a pattern P with at most k differences, we first split the pattern into $k+1$ pieces and search for each piece in the index. If the indexed substrings starting with the piece are longer than the piece, we return all positions associated with any substring starting with the piece. If an indexed substring is a prefix of the piece, we return the positions associated with that indexed substring. The positions returned by this search are then verified with the $\mathcal{O}(m^2)$ dynamic programming algorithm to find the approximate occurrences of the pattern. As an example consider searching for the pattern $P =$ "abbab" in the prefix free index of Fig. 3(a) with at most $k = 1$ difference. We start by splitting the pattern into two pieces: "ab" and "bab". The search for "ab" in the index returns positions 3 and 6 and the search for "bab" returns positions 4, 7, and 8. We then verify around these positions and find the occurrences starting at positions 3 and 6.

Although the lengths of the position lists are more uniform than in the q-gram index, we still benefit from computing the optimal partitioning of the pattern. First of all, the lengths of the position lists still vary, and thus the number of verifications can be reduced by choosing pattern pieces with short position lists. Secondly, if the pattern is too short to be partitioned into long enough pieces such that we would get only one position list per pattern piece, it is not clear how to select the pieces without the optimal partitioning technique.

Finding the optimal partitioning of the pattern works similarly to the q-gram index. We first use binary search to locate the indexed substring(s) corresponding to each pattern piece $p_i \ldots p_{j-1}$ for $1 \leq i < j \leq m+1$. This search returns a contiguous region of indexed substrings. If we again store the accumulated position list lengths, we can determine the number of triggered verifications fast. This number is stored in the table $R[i,j]$. We then compute the tables $P[i,k]$ and $C[i,k]$ and obtain from these tables the optimal partitioning of the pattern. The overall time to find the optimal partitioning is $\mathcal{O}(m^2(\log n + k))$.

To choose the vocabulary of the index, we use a simplified version of the technique of Klein and Shapira [5] for constructing fixed length codes for compression of text. Their technique is based on the suffix tree of the text. A cut in a suffix tree is defined as an imaginary line that crosses exactly once each path in the suffix tree from the root to one of the leaves. The lower border of a cut is defined to be the nodes immediately below the imaginary line that forms the cut. Klein and Shapira show that a lower border of a cut forms a prefix free set and a prefix of each suffix of the text is included in the lower border. Thus the lower border of a cut can be used as a vocabulary in the prefix free index.

We choose the vocabulary as follows. First we build the suffix tree of the text and augment it with the frequencies of the nodes. The frequency of a node is the frequency of the corresponding substring of the text. We then traverse the suffix tree in depth first order. If the frequency of a node is at most the threshold frequency α, we add the corresponding string S to the vocabulary and we also add the corresponding leaves to the position list of the string S.

The suffix tree can be built in $\mathcal{O}(n)$ time. The traversal of the tree also takes $\mathcal{O}(n)$ time and we do $\mathcal{O}(1)$ operations in each node. After the traversal the position lists are sorted, which takes $\mathcal{O}(n \log \alpha)$ total time. Finally the position lists are difference coded and compressed taking $\mathcal{O}(n)$ total time. Thus the construction of the prefix free index takes $\mathcal{O}(n \log \alpha)$ time.

4 Prefix Coalesced Index

In the second new variant of the q-gram index, the prefix coalesced index, we require that the vocabulary includes some prefix of each suffix of the text, and if the vocabulary contains two strings S and R such that R is a proper prefix of S, then all positions of the text starting with S are assigned only to the position list of S. Again the position lists are difference coded and the differences are compressed with variable length integer coding.

To choose the vocabulary, we build the suffix tree of the text and augment it with the frequencies of the nodes. The suffix tree is traversed in depth first order so that the children of a node are traversed in descending order of frequency. When we encounter a node whose frequency is at most the threshold frequency α, we add the corresponding string to the vocabulary, subtract the original frequency of this node from the frequency of its parent node and reconsider adding the string corresponding to the parent node to the vocabulary. When a string is added to the vocabulary, we also add the leaves to its position list. Figure 3(b) shows an example of a prefix coalesced index with threshold frequency two.

The suffix tree can again be built in $\mathcal{O}(n)$ time. Because we need to sort the children of each node when traversing the suffix tree, the traversal now takes $\mathcal{O}(n \log \sigma)$ time, where σ is the size of the alphabet. After the traversal the handling of the position lists takes $\mathcal{O}(n \log \alpha)$ as in the prefix free index. Thus the construction of the prefix coalesced index takes $\mathcal{O}(n(\log \alpha + \log \sigma))$ time.

We refine the searching procedure as follows. We again start by splitting the pattern into $k + 1$ pieces and search the index for each piece. If the piece is a

prefix of several indexed substrings, we return all position lists associated with these indexed substrings. If an indexed substring is a *proper* prefix of the piece, we return only this position list. Otherwise searching on the index and optimal partitioning of the pattern work exactly the same way as in the prefix free index.

5 Experimental Results

To save space our implementations of the prefix free and the prefix coalesced indexes use a suffix array [6] instead of a suffix tree when constructing the index. The traversal of the suffix tree is simulated using binary search on the suffix array.

For compressing the position lists in all the indexes, we use bytewise compression of the differences. In this scheme, the highest bit is 0 in the last byte of the coding and 1 in other bytes. The integer is formed by concatenating the seven lowest bits of all the bytes in the coding.

The experiments were run on a 1.0 GHz AMD Athlon dual core processor with 2 GB of memory, running Linux 2.6.23. The code is in C and compiled with `gcc` using `-O2` optimization. We used the 200 MB English text from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`, and the patterns are random substrings of the text. For each pattern length, 1,000 patterns were generated.

Figure 4 shows the search times for the q-gram index and the prefix free and prefix coalesced indexes for various pattern lengths and values of k. The q-gram

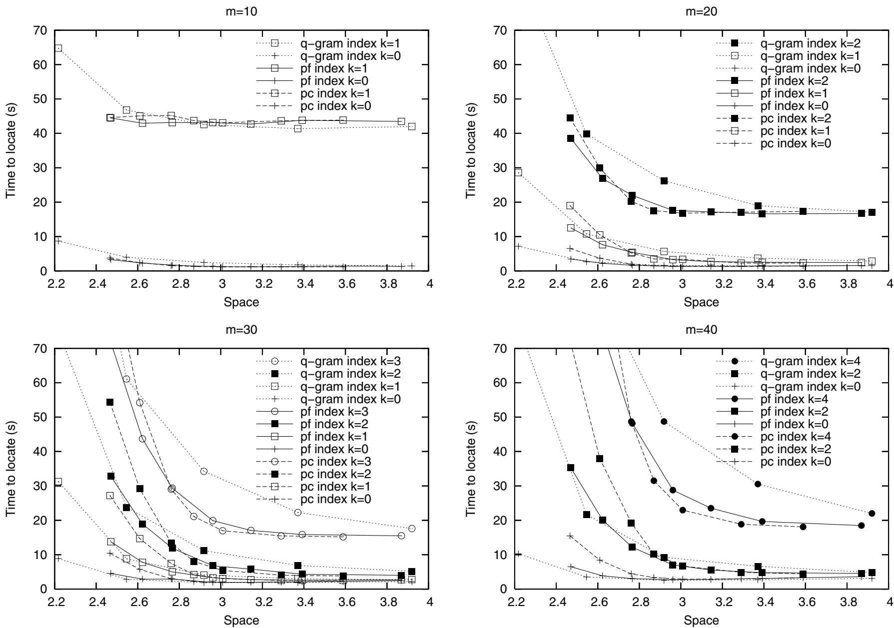


Fig. 4. Search times for the different indexes for various values of k and m. The space fraction includes that of the text, so it is of the form $1 + \frac{\text{index size}}{\text{text size}}$.

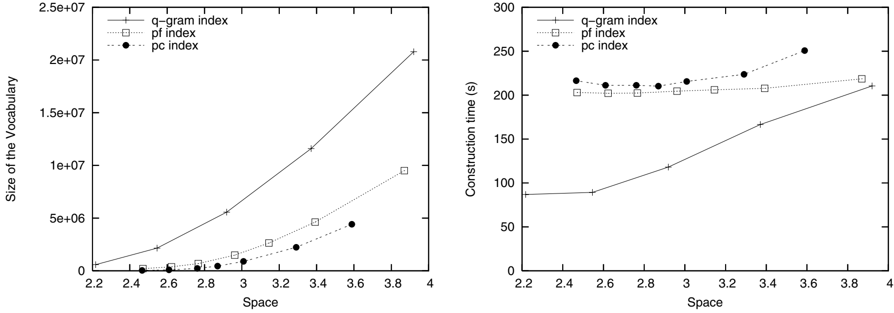


Fig. 5. The vocabulary size and the construction time of the q-gram index and the prefix free and prefix coalesced indexes as a function of space usage

index was tested with 5 values of q: 4, 5, 6, 7, and 8. The prefix free index was tested with 7 values for the threshold frequency α: 200, 500, 1,000, 2,000, 5,000, 10,000, and 20,000. The prefix coalesced index was also tested with 7 values for the threshold frequency α: 100, 200, 500, 1,000, 2,000, 5,000, and 10,000. We see that the new indexes generally achieve the same performance as the q-gram index using less space. The prefix coalesced index allows more flexibility in selecting the vocabulary, and so the position list lengths are more uniform than in the prefix free index. Thus the search times in the prefix coalesced index are slightly lower than in the prefix free index. However, when we reduce the available space the prefix free index becomes faster than the prefix coalesced index.

Figure 5 shows the vocabulary size for the different indexes. We see that the vocabulary of the q-gram index is much larger than the vocabulary of the prefix free and the prefix coalesced indexes. Some of the q-grams are very frequent in the text and their long position lists compress very efficiently, allowing the q-gram index to use a larger vocabulary. In the prefix free and prefix coalesced indexes the position lists have a more uniform length, and thus these lists do not compress as well, so the vocabulary is much smaller. We can also see that the vocabulary of the prefix free index is larger than the vocabulary of the prefix coalesced index, again reflecting the lengths of the position lists.

Figure 5 also shows the construction time for the different indexes. The construction time of the prefix free and prefix coalesced indexes increases only little when space usage is increased because the most time consuming phase of their construction is the construction of the suffix array, which takes the same time regardless of the space usage of the final index.

6 Conclusions and Further Work

We have presented two new indexes for exact and approximate string matching based on the q-gram index. They index variable length substrings of the text to achieve more uniform lengths of the position lists. The indexed substrings in the prefix free index form a prefix free set, whereas in the prefix coalesced index

this restriction is lifted. Our experiments show that the new indexes are up to 40 % faster than the q-gram index for the same space. This shows that lists of similar length are definitely beneficial for the search performance, although they are not as compressible and thus shorter substrings must be indexed.

Our techniques receive a parameter α, giving the maximum allowed list length, and produce the largest possible index that fulfills that condition. Instead, we could set the maximum number of substrings to index, and achieve the most uniform possible index of that vocabulary size. For this we would insert the suffix tree nodes into a priority queue that sorts by frequency, and extract the most frequent nodes (with small adaptations depending on whether the index is prefix free or prefix coalesced). The construction time becomes $\mathcal{O}(n \log n)$.

Future work involves extending more sophisticated techniques based on q-grams and q-samples, such as those requiring several nearby pattern pieces to be found before triggering a verification, and/or based on approximate matching of the pattern pieces in the index [8,9].

References

1. Baeza-Yates, R., Navarro, G.: Faster approximate string matching. Algorithmica 23(2), 127–158 (1999)
2. Baeza-Yates, R., Perleberg, C.: Fast and practical approximate string matching. Information Processing Letters 59(1), 21–27 (1996)
3. Holsti, N., Sutinen, E.: Approximate string matching using q-gram places. In: Proc. Finnish Symp. on Computer Science, pp. 23–32. University of Joensuu (1994)
4. Jokinen, P., Ukkonen, E.: Two algorithms for approximate string matching in static texts. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 240–248. Springer, Heidelberg (1991)
5. Klein, S.T., Shapira, D.: Improved variable-to-fixed length codes. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 39–50. Springer, Heidelberg (2008)
6. Manber, U., Myers, G.: Suffix arrays: A new method for online string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
7. Navarro, G., Baeza-Yates, R.: A practical q-gram index for text retrieval allowing errors. CLEI Electronic Journal 1(2) (1998)
8. Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Engineering Bulletin 24(4), 19–27 (2001)
9. Navarro, G., Sutinen, E., Tarhio, J.: Indexing text with approximate q-grams. Journal of Discrete Algorithms 3(2–4), 157–175 (2005)
10. Shi, F.: Fast approximate string matching with q-blocks sequences. In: Proc. WSP 1996, pp. 257–271. Carleton University Press (1996)
11. Sutinen, E., Tarhio, J.: On using q-gram locations in approximate string matching. In: Spirakis, P.G. (ed.) ESA 1995. LNCS, vol. 979, pp. 327–340. Springer, Heidelberg (1995)
12. Sutinen, E., Tarhio, J.: Filtration with q-samples in approximate string matching. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 50–63. Springer, Heidelberg (1996)
13. Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM 35(10), 83–91 (1992)

Expectation of Strings with Mismatches under Markov Chain Distribution

Cinzia Pizzi[1,⋆] and Mauro Bianco[2]

[1] Dipartimento di Ingegneria dell' Informazione, Università di Padova, Italy
[2] Department of Computer Science, Texas A&M University, USA
cinzia.pizzi@dei.unipd.it, bmm@cse.tamu.edu

Abstract. We study a problem related to the extraction of over-represented words from a given source text x, of length n. The words are allowed to occur with k mismatches, and x is produced by a source over an alphabet Σ according to a Markov chain of order p. We propose an online algorithm to compute the expected number of occurrences of a word y of length m in $O(mk|\Sigma|^{p+1})$. We also propose an offline algorithm to compute the probability of any word that occurs in the text in $O(k|\Sigma|^2)$ after $O(nk|\Sigma|^{p+1})$ pre-processing. This algorithm allows us to compute the expectation for all the words in a text of length n in $O(kn^2|\Sigma|^2+nk|\Sigma|^{p+1})$, rather than in $O(n^3|\Sigma|^{p+1})$ that can be obtained with other methods. Although this study was motivated by the motif discovery problem in bioinformatics, the results find their applications in any other domain involving combinatorics on words.

1 Introduction

The problem of extracting unusually frequent or rare patterns from observed sequences has been the subject of much study in Molecular Biology. The problem can be set up under different assumption about the model of the patterns to discover, called *motifs*, and the algorithmic approach (see [3,6,9] and references within). The inherent variability of biological sequences brings about the problem of how to account for mutations, insertions or deletions, and how one should define the motif search space [8]. Profile-based models are often used with maximum-likelihood approaches [2], while pattern-based models exploit significance measures such as *z-scores* or *p-values* [5,7] to discern between interesting and uninteresting patterns. We are specifically interested in this latter case when the variability is measured by Hamming distance. About the definition of the search space, a general framework would consider as a valid motif any string generated from the given alphabet. This approach is very rigorous, but it suffers from an intrinsic exponential complexity. An alternative framework, called *sequence based*, would consider as valid motifs only those strings that actually

⋆ Corresponding author. Work by this author was supported in part by the Research Grant "Avere Trent'anni".

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 222–233, 2009.
© Springer-Verlag Berlin Heidelberg 2009

occur at least once in the sequence to analyze. Indeed, it is reasonable to expect that at least one instance of the motif, or part of it that can be used as an anchor, occurs exactly in the sequence under analysis.

Here the problem under study is the computation of the expected frequency with mismatches of motifs that occur in the input sequence at least once. The background distribution is a Markov chain. This is a more realistic framework than independent and identically distributed (*i.i.d.*). In Sec.2 we give basic definitions and previous work. In Sec.3 we present Algorithm 1 for the online computation of the expected frequency of a single word of length m with k mismatches in $O(mk|\Sigma|^{p+1})$ for a Markov chain of order p. In Sec.4 we present Algorithm 2 to compute the expected frequency of any word in a text of length n in $O(k^2|\Sigma|)$ after $O(nk|\Sigma|^{p+1})$ preprocessing. This algorithm computes the expected frequency with k mismatches of all the words in the text in $O(nk|\Sigma|^{p+1} + n^2k^2|\Sigma|)$ versus $O(n^3|\Sigma|^{p+1})$ of existing methods or direct application of Algorithm 1.

2 Preliminaries

We are interested in computing the expectation of the substrings of the text. These could next be used to compute some kind of z-score based on the comparison of actual and expected frequency.

2.1 Basic Definitions

Let us consider a text string $X = X_1X_2 \dots X_n$, randomly generated by a source that emits symbols from an alphabet Σ according to a given probability distribution $\wp$. Given an arbitrary fixed pattern $y = y_1y_2 \dots y_m$, with $m \leq n$, and a fixed number of mismatches $k \leq m$, then the indicator variable Z_i takes value 1 if y occurs starting at position i in X, 0 otherwise. Then $Z = \sum_{i=1}^{m-n+1} Z_i$ is the random variable that counts the number of occurrences of y in X.

If the X_i are identically distributed, i.e. $P(X_i = y_j) = p_j$ for every i, where p_j is the probability of the symbol y_j according to $\wp$, and independent, then the expected probability for y to occur at position i is:

$$E(Z_i|y) = P(X_i = y_1, \dots, X_{i+m-1} = y_m) = \prod_{j=1}^{m} P(X_{i+j-1} = y_j) = \prod_{j=1}^{m} p_j = \hat{p}$$

Thus, the expected number of occurrences is $E(Z|y) = (n - m + 1)\hat{p}$.

If we assume that the X_i are dependent according to a Markov chain of order M, then we will have a similar formula, but with conditional probabilities. For example, for order $M = 1$ we have:

$$E(Z_i|y) = p_1 \prod_{j=2}^{m} (p_j|p_{j-1})$$

2.2 Previous Work on Expectation with Mismatches

The expected number of occurrences with k mismatches for a word y is given by the summation of the expectation of all the words that belong to the neighbor at distance k from y. This neighborhood has size $\binom{m}{k}|\Sigma - 1|^k$, and the computation of each probability takes m steps.

In [1] a dynamic programming approach is proposed to compute the expected probability with k mismatches, under i.i.d. hypothesis, for words that occur in a text of length n. The expectation of any word in the text is computed in $O(k^2)$ time after $O(kn)$ pre-processing. The expectation of all the words of fixed length m is computed in $O(kn)$, hence amortized $O(k)$ time per word.

In [4] a different approach is proposed that computes the expectation of a *generic* word $y \in \Sigma^*$, in time $O(|y|k)$ under i.i.d. hypothesis, and $O(|y|k|\Sigma|^{p+1})$ under Markov chain of order p.

3 Expectation with Mismatches: Algorithm 1

The naive approach consists in the enumeration of all the possible ways to distribute the k mismatches among the m position of the pattern, hence it has the same time complexity than for the i.i.d. naive approach: $O(m^{k+1}|\Sigma|^k)$, that is exponential in the number of mismatches.

3.1 Online Algorithm 1

Here we describe an algorithm for Markov chains of the first order that runs in $O(mk|\Sigma|^2)$. Our approach consists in the generation of all possible distribution of mismatches, rather than all possible patterns in the k-neighborhood of y. In practice we consider an alphabet $\Lambda = \{C, W\}$, where 'C' is *correct*, and 'W' is *wrong*, i.e. a mismatch. For example, the string "CCWCWC" means that, in a pattern of length six, two mismatches occur at positions three and five. A string $w \in \Lambda^m$ is called *(m,k)-mismatch pattern* or simply *mismatch pattern* when m and k are clear from the context.

Building the (m,k)-Tree. Let us consider a pattern $y = y_1 y_2 \ldots y_m$, and k mismatches. To visualize all possible mismatch patterns we build a tree T of height m, called (m, k)-Tree. Each node of the tree has at most two children. A node at height h (depth from the root $m - h + 1$) is associated with the symbol y_{h+1}. If the node is the left child of its parent then position h is *correct*, otherwise it is the site of a mismatch. For ease of visualization we labeled the arcs with the symbols taken from Λ, so that the mismatch patterns can be spelt out by the concatenation of the arc labels on the paths from any leaf up to the root node.

Whenever a mismatch occurs, any of the $|\Sigma| - 1$ symbols that are different from the correct one can replace it. Hence, the expectation of the pattern with mismatches depends on which symbol was chosen for the substitution. Since

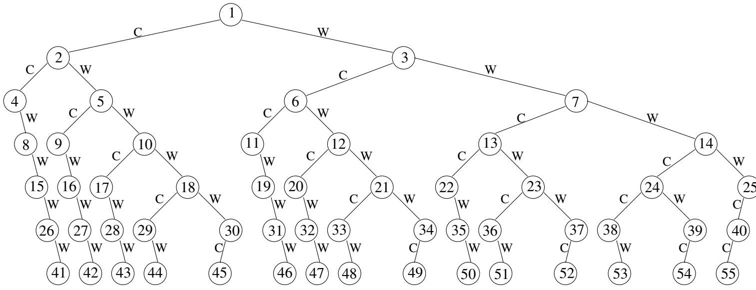


Fig. 1. An example of (m, k)-Tree for patterns of length $m = 6$ and $k = 4$ mismatches

we must consider all possible substitutions we associate each node to a vector $\mathbf{v}$=$[v_1 v_2 \dots v_{|\Sigma|}]$. The vector associated with a node u at height h will store the probability of $y_1 \dots y_h$ to occur with a number of mismatches given by the number of right children in the path from u to any of its leaves. In particular v_i will store the contribution to this probability given by the strings that have the ith symbol of the alphabet Σ in their last position. The overall probability is given by $|\mathbf{v}|$.

Example for aabcab
Let us consider the node 14 in Fig.1. This node is associated with the strings $\bar{a}ab\bar{c}$, $a\bar{a}b\bar{c}$, and $aa\bar{b}\bar{c}$. Assuming that a is the first symbol of Σ, b the second, and so on, we will have that v_1 will store the probability of having either $\bar{a}ab$, $a\bar{a}b$, and $aa\bar{b}$ in the first three positions of the pattern, and a in the fourth position. Similarly, v_2 will store the probability of having either $\bar{a}ab$, $a\bar{a}b$, and $aa\bar{b}$ in the first three positions of the pattern, and b in the fourth position, and so on.

3.2 The Algorithm

To obtain the overall probability of y with k mismatches under the first order Markov chain hypothesis we proceed bottom up, from the leaves to the root. At each step for a generic node u we sum the vectors of the left child $\mathbf{v_l}$ and of the right child $\mathbf{v_r}$, to obtain a vector $\mathbf{t} = \mathbf{v_l} + \mathbf{v_r}$. This vector is multiplied by a matrix that gives conditional probabilities with respect to the symbol assigned to the node u. Finally, we sum up the content of the vector at the root node to obtain the value of the expectation. We explain the algorithm with the help of the example in Fig.2, where a (3,2)-Tree is built for $y = abb$ and $\Sigma = \{a, b, c, d\}$.

Initial Set Up. Consider a leaf that is a left child of its parent. This means that the first letter of y, an a in our example, is correct, hence the vector v of the leaf must be initialized with the value $[P_a, 0, 0, 0]$. This is the case of node 9. On the other hand, a leaf that is a right child of its parent must be initialized with $[0, P_b, P_c, P_d]$. This is the case of both nodes 7 and 8.

Propagation. Consider now a generic node u. First of all the vectors of its children nodes must be summed to obtain the vector $\mathbf{t}$. This takes $O(|\Sigma|)$ time. Next we

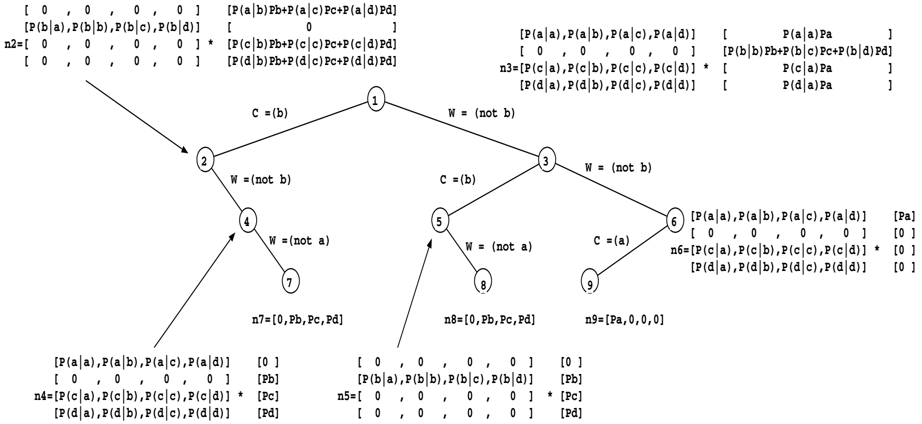


Fig. 2. An example of bottom-up propagation of the conditional probabilities from the leaves to the root node. The sum of the elements of the vector at the root node gives the total probability of having $y = abb$ with 2 mismatches.

must consider whether u is a left or a right child, and what is the symbol s associated with its height.

If u is a left child, then the symbol s is correct in this path. We compute the product between $\mathbf{t}$ and the vector $\mathbf{p} = (P(s|s_1), P(s|s_2), \ldots, P(s|s_{|\Sigma|}))$ of conditional probabilities of s. This takes time $O(|\Sigma|)$.

If u is a right child, then the symbol s is a mismatch in this path. In this case we have to perform the same computation as before for all the symbols $s' \in \Sigma$, with $s' \neq s$. This takes time $O(|\Sigma|^2)$.

We can generalize the procedure described above, without increasing the asymptotic time complexity, considering each step as the product of the vector $\mathbf{t}$ by a matrix $M_{|\Sigma|\times|\Sigma|}$. The matrix M is built differently depending whether the symbol s associated with the node is considered a mismatch or not.

If the symbol is correct (left children), all the rows corresponding to symbols $s' \neq s$ will be set at 0, while the row corresponding to s will hold the values $P(s|s_1), P(s|s_2), \ldots, P(s|s_{|\Sigma|})$. If the symbol is a mismatch, then its row is 0 and all the other rows will hold the conditional probabilities of the corresponding mismatch symbols. The final vector for a node is given by $M_s\mathbf{t}$ or $M_{\bar{s}}\mathbf{t}$.

3.3 Analysis of Complexity in the Tree

We want to compute the number of nodes in a (m, k)-Tree.

The Number of Arcs

The number of right arcs of a (m, k)-Tree is:

$$R(m, k) = 1 + R(m-1, k) + R(m-1, k-1) \tag{1}$$

The basis of the recurrence are: $R(m, m) = m$, since if we want to distribute m mismatches in a string of length m, all positions are mismatches and there is

a single path in the tree in which all nodes have right children; and $R(m, 0) = 0$, since if no mismatch occurs, we have only one path in the tree in which all nodes are left children.

To build a tree for a string of length m we can proceed recursively: we create a root node with two children. The left child tells us that the added position is correct, so we need to have a subtree for a string of length $m-1$ and still k mismatches. On the other hand, the right child tells us that in the new position we have a mismatch, hence the subtree on the right must be built on a string of length $m-1$ with $k-1$ mismatches. The number of arcs of the subtree rooted at the left child is $R(m-1, k)$, while the number of arcs of the subtree rooted at the left child is $R(m-1, k-1)$. To obtain the total number of right arcs for the tree we need to add the arc from the root to the right subtree, thus obtaining Equation (1).

Lemma 1. *The number of right arcs $R(m, k)$ is:*

$$R(m, k) = \binom{m+1}{k} - 1$$

Proof. The proof is by induction.

Basis.

$$k = 0 : R(m, 0) = \binom{m+1}{0} - 1 = 1 - 1 = 0$$
$$k = m : R(m, m) = \binom{m+1}{m} - 1 = m + 1 - 1 = m$$

Induction.

Let us assume as inductive hypothesis that $R(m, k) = \binom{m+1}{k} - 1$ and prove our formula for $m+1$.

$$\begin{aligned} R(m+1, k) &= 1 + R((m+1) - 1, k) + R((m+1) - 1, k-1) \\ &= 1 + \binom{m+1}{k} - 1 + \binom{m+1}{k-1} - 1 = \binom{(m+1)+1}{k} - 1 \end{aligned}$$ □

Following a similar discussion for the left arcs:

$$L(m, k) = 1 + L(m-1, k) + L(m-1, k-1)$$

with dual basis $L(m, m) = 0$, and $L(m, 0) = m$ and lemma. The proof follows the same steps as for $R(m, k)$.

Lemma 2. *The number of left arcs $L(m, k)$ is:*

$$L(m, k) = \binom{m+1}{k+1} - 1$$

Theorem 1. *The total number of nodes in the (m, k)-Tree is* $\binom{m+2}{k+1} - 1$.

Proof. Since the number of nodes is $N(m,k) = R(m,k) + L(m,k) + 1$ *(the root) we have from Lemma 1 and Lemma 2:*

$$N(m,k) = R(m,k) + L(m,k) + 1 = \binom{m+2}{k+1} - 1 \qquad \square$$

At each node we have $O(|\Sigma|^2)$ operations, hence the overall time complexity is $O(\binom{m+2}{k+1}|\Sigma|^2)$.

3.4 Reducing the Time Complexity

The (m,k)-Tree allows us to consider only once the parts of the paths that are shared among some subsets of all the possible distributions of mismatches. For example, in Fig.1, the distributions $CWWWCW$ (node 44) and $CWWWWC$ (node 45) share the common path $CWWW$ from root to node 18, that is computed only once. Although with respect to the naive approach we eliminated the exponential factor $|\Sigma|^k$ reducing it to $|\Sigma|^2$, we still have exponential dependency of the string length m with respect to the number of mismatches k.

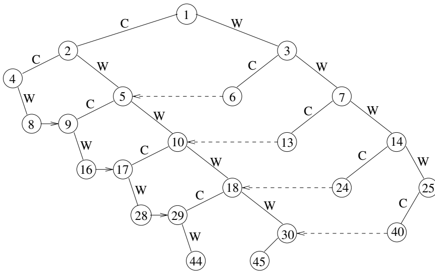


Fig. 3. Pruned Tree for "aabcda"

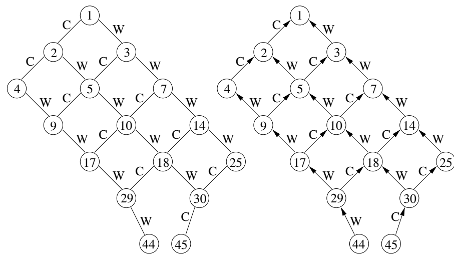


Fig. 4. Graph and DAG for "aabcda"

However, if we look carefully at Fig.1 we notice that many subtree structures in the tree are repeated. For example, the subtree rooted at node 5 holds the same mismatch distributions than the subtree rooted at node 6. The same happens for the subtrees rooted at node 10,12, and 13. If we redraw the tree in Fig.1 by taking into account these redundancies, we obtain a rather cleaner picture in which the dotted arcs represent the links to the equivalent nodes (see Fig.3). If we further collapse the equivalent nodes we obtain a graph of $(m-k+1) \times (k+1)$ nodes. Giving an orientation to the arcs in such a way that they follow the direction of the computation we obtain a directed acyclic graph (see Fig.4). It can be noticed that only 16 nodes are needed to build the DAG. Given a word y of length m, every path from the bottom corner (nodes 44 and 45) to the upper corner node (node 1) has length m, and represents a way to distribute the k mismatches in the pattern y. In fact, every horizontal move corresponds to a correct character, while every vertical move corresponds to a mismatch. To reach the node 1 from

either node 44 or 45, we need to traverse exactly k vertical edges, and exactly $m-k$ horizontal edges, hence we will describe a pattern of length m with exactly k mismatches.

To perform the computation described before we can map our graph in a table with $k+1$ rows and $m-k+1$ columns. The cell (i,j) holds the probability related to the prefix of y of length $i+j$, when i mismatches occur. Let $\mathbf{t_{i,j}}$ be the vector holding the partial probabilities, we have:

$$\mathbf{t_{i,j}} = M_{y_{i+j}} \mathbf{t_{i-1,j}} + M_{\bar{y}_{i+j}} \mathbf{t_{i,j-1}}$$

The table must be filled row by row. The number of cells is $(m-k+1) \times (k+1)$, and for each cell we spend $O(|\Sigma|^2)$ time. The total complexity is $O(mk|\Sigma|^2)$. This algorithm is polynomial in both the string length and the number of mismatches.

The procedure we just described can be generalized for a Markov chain of order p. In this case the matrix M has size $|\Sigma|^p \times |\Sigma|$, so the complexity is $O(mk|\Sigma|^{p+1})$.

4 Expectation with Mismatches: Algorithm 2

In this section we propose an algorithm that after $O(nk|\Sigma|^{p+1})$ preprocessing, computes the expectation with k mismatches of any word in the text in $O(k^2|\Sigma|)$ time. We explain the algorithm for the case $p = 1$ for ease of discussion.

4.1 Preprocessing

In this phase we build a table V where the entry corresponding to the pair (i,j), denoted by $V^{i,j} = (v_t^{ij})_{t=1\ldots|\Sigma|}$, is the vector of the expectations of the prefix $P_j = x_1 \ldots x_j$ with i mismatches. The single entry v_t^{ij} is the contribution to the expectation E_{ij} when considering strings $w_1 \ldots w_j$ in the neighborhood $N_i(P_j)$ at distance i from P_j such that $w_j = s_t \in \Sigma$: $E_{ij} = \sum_{t=1}^{|\Sigma|} v_t^{ij}$. The values of $V^{i,j}$ are computed by dynamic programming:

$$V^{i,j} = M_{x_j} V^{i,j-1} + M_{\bar{x}_j} V^{i-1,j-1}$$

Table V is built in $O(nk|\Sigma|^2)$ for Markov chains of order $p = 1$. The generalization takes $O(nk|\Sigma|^{p+1})$. We also build an array A in which $A[i]$ stores the probability of the prefix of length i according to the given symbols distribution. This array is built in $O(n)$: $A[1] = p_{x_1}$, $A[j] = p_{x_j|x_{j-1}} A[j-1]$ for $1 < j \leq n$.

4.2 Processing

Given table V and array A, a pair of indexes (s,e), and a number of mismatches k, we want to compute the expectation with k mismatches of $x_s \ldots x_e$. The contributions to the expectation of $x_1 \ldots x_e$ with i mismatches are stored in $V^{i,e}$. Starting from these values we need to get rid of the contribution to the

expectation of the prefix $x_1 \dots x_{s-1}$. We need to pay attention at position s, where a mismatch will affect both $p_{s+1|s}$ and $p_{s|s-1}$. In order to manipulate the contribution we take E_{ie} and build another vector $U^{ie} = (u_t^{ie})$ where each component u_t^{ie} stores the contribution to E_{ie} when the first symbol of the pattern we are interested in (i.e. position s in x) is the symbol s_t. The corresponding contribution to the expectation between s and e will be denoted by $C_{s_t}^{i,s,e}$. It is straightforward to observe that $C^{0,s,e} = A[e]/A[s]$ and $Exp(0,s,e) = C^{0,s,e} p_{x_s}$. We will then proceed increasing the number of mismatches, computing U_{ie} and $C^{i,s,e}$ for $i = 1 \dots k$. We first describe the procedure step-by-step for $i = 1$ and $i = 2$, and then we generalize the processing.

4.3 Computing $U^{1,e}$ in $O(|\Sigma|)$

To calculate the components of $U^{1,e}$ we have to deal with two cases:

a. A mismatch occurs at position s ($s_t \neq x_s$)

$$\begin{aligned} u_t^{1,e} &= P_{1\dots s-1} p_{s_t|x_{s-1}} p_{x_{s+1}|s_t} P_{s+2\dots e} \\ &= (p_{x_1} p_{x_2|x_1} \cdots p_{x_{s-1}|x_{s-2}}) p_{s_t|x_{s-1}} p_{x_{s+1}|s_t} (p_{x_{s+2}|x_{s+1}} \cdots p_{x_e|x_{e-1}}) \\ &= A[s-2] p_{s_t|x_{s-1}} p_{x_{s+1}|s_t} A[e]/A[s+1] \end{aligned}$$

b. A match occurs at position s ($s_t = x_s$)

$$u_t^{1,e} = \sum_{t=1}^{|\Sigma|} v_t^{1,e} - \sum_{t:s_t \neq x_s} u_t^{1,e} = E_{1e} - \sum_{t:s_t \neq x_s} u_t^{1,e}$$

The time complexity is $O(|\Sigma|)$.

How to Compute $C^{1,s,e}$. The probability of $x_1 \dots x_e$ with 1 mismatch is:

$$P(x_1 \dots x_e, 1) = P(x_1 \dots x_{s-1}, 1) P(x_s \dots x_e, 0) + P(x_1 \dots x_{s-1}, 0) P(x_s \dots x_e, 1)$$

We are interested in computing $P(x_s \dots x_e, 1)$:

$$P(x_s \dots x_e, 1) = [P(x_1 \dots x_e, 1) - P(x_1 \dots x_{s-1}, 1) P(x_s \dots x_e, 0)] / P(x_1 \dots x_{s-1}, 0)$$

The probability $P(x_1 \dots x_e, 1)$ can be taken from $U^{1,e} = (u_t^{1,e})_{t \in \Sigma}$.

The values of $P(x_1 \dots x_{s-1}, 1)$ can be taken from $V^{1,s-1} = (v_t^{1,s-1})_{t \in \Sigma}$. Each $v_t^{1,s-1}$ needs to be multiplied by $p_{x_s|s_t}$, that depends on t, and then by $p_{x_{s+1}|x_s} \cdots p_{x_e|x_{e-1}}$. All these contributions refer to the case in which x_s is not a site for a mismatch, hence they will be subtracted from $u_{x_s}^{1,e}$.

Moreover, for all the components $u_{s_t}^{1,e}$ we have to get rid of the contribution of the prefix of length $s-1$. Each component needs to be divided by $p_{s_t|x_{s-1}}$, which depends on t and then all of them are divided by $p_{x_1} \cdots p_{x_{s-1}|x_{s-2}}$.

In summary:

$$C^{1,s,e} = \begin{bmatrix} u_1^{1,e}/p_{s_1|x_{s-1}} \\ u_2^{1,e}/p_{s_2|x_{s-1}} \\ \cdots \\ (u_{x_s}^{1,e} - \sum_{t=1}^{|\Sigma|} (v_t^{1,s-1} p_{x_s|s_t})(p_{x_{s+1}|x_s} \cdots p_{x_e|x_{e-1}})/p_{x_s|x_{s-1}} \\ \cdots \\ u_{|\Sigma|}^{1,e}/p_{s_{|\Sigma|}|x_{s-1}} \end{bmatrix} \frac{1}{p_{x_1} \cdots p_{x_{s-1}|x_{s-2}}}$$

That can be re-written as:

$$C^{1,s,e} = \begin{bmatrix} u_1^{1,e}/p_{s_1|x_{s-1}} \\ u_2^{1,e}/p_{s_2|x_{s-1}} \\ \cdots \\ (u_{x_s}^{1,e} - v_{x_s}^{1,s} c^{0,s,e})/p_{x_s|x_{s-1}} \\ \cdots \\ u_{|\Sigma|}^{1,e}/p_{s_{|\Sigma|}|x_{s-1}} \end{bmatrix} \frac{1}{p_{x_1} \cdots p_{x_{s-1}|x_{s-2}}}$$

The computation of each component requires $O(1)$, hence $O(|\Sigma|)$ time overall. The final value of expectation is: $Exp^1(s,e) = \sum_{t=1}^{|\Sigma|} c_t^1 p_{s_t}$.

Computing $U^{2,e}$. We assume to have at hand $V^{1,e}$, $V^{2,e}$, $U^{1,e}$, $C^{1,s,e}$.

If $s_t \neq x_s$ there is 1 mismatch at position s, so the contributions to the expectation of $x_1 \dots x_e$ with 2 mismatches that involve s_t are:

$1 \dots s-1$	s	$s+1 \dots e$
1 mismatch	x	0 mismatch
0 mismatch	x	1 mismatch

$$\begin{aligned} u_t^{2,e} &= (\textstyle\sum_{i=1}^{|\Sigma|} v_i^{1,s-1} p_{s_t|s_i}) p_{x_{s+1}|s_t} (p_{x_{s+2}|x_{s+1}} \cdots p_{x_e|x_{e-1}}) \\ &\quad + (\textstyle\sum_{i=1}^{|\Sigma|} c_i^{1,s+1,e} p_{s_i|s_t})(p_{x_1} \cdots p_{x_{s-1}|x_{s-2}}) p_{s_t|x_{s-1}} \\ &= v_{s_t}^{1,s} c_{s_t}^{1,s,e} + c_{s_t}^{1,s,e} c_{s_t}^{1,1,s} \end{aligned}$$

The computation requires $O(1)$ for each $s_t \neq x_s$.

If $s_t = x_s$ the mismatches must occur at positions other than s.

$1 \dots s-1$	s	$s+1 \dots e$
2 mismatches	=	0 mismatch
1 mismatch	=	1 mismatch
0 mismatch	=	2 mismatches

As before we can compute this component directly:

$$u_t^{2,e} = \sum_{t \in \Sigma} v_i^{2,e} - \sum_{t:s_t \neq x_s} u_t^{2,e} = E_{2e} - \sum_{t:s_t \neq x_s} u_t^{2,e}$$

The overall time needed to compute $U^{2,e}$ is $O(|\Sigma|)$.

How to Compute $C^{2,s,e}$. The probability of $x_1 \dots x_e$ with 2 mismatch is:

$$\begin{aligned} P(x_1 \dots x_e, 2) = & P(x_1 \dots x_{s-1}, 0) P(x_s \dots x_e, 2) + \\ & + P(x_1 \dots x_{s-1}, 1) P(x_s \dots x_e, 1) + \\ & + P(x_1 \dots x_{s-1}, 2) P(x_s \dots x_e, 0) \end{aligned}$$

We are interested in computing $P(x_s \dots x_e, 2)$:

$$\begin{aligned} P(x_s \dots x_e, 2) = & [P(x_1 \dots x_e, 2) - P(x_1 \dots x_s - 1, 2) P(x_s \dots x_e, 0) + \\ & - P(x_1 \dots x_s - 1, 1) P(x_s \dots x_e, 1)] / P(x_1 \dots x_s - 1, 0) \end{aligned}$$

For ease of notation we assign letters to the factors:

$$P(x_s \dots x_e, 2) = [A - BC - DE]/F$$

The factor A is taken from $U^{2,e}$. The product BC assumes that x_s is correct, so it will contribute only to the value of $u^{2,e}_{x_s}$.

$$\sum_{i=1}^{|\Sigma|} v_i^{2,s-1} p_{x_s|x_{s_i}} (p_{x_{s+1}|x_s} \cdots p_{x_e|x_{e-1}}) = v^{2,s}_{x_s} C^{0,s,e}_{x_s}$$

In the product DE there might or might not be a mismatch at position s. Hence we need to consider separately the two cases.

If s is a site of a mismatch the other mismatch must occur in the prefix of length $s-1$. The contribution to subtract to each $s_t \neq x_s$ is:

$$(\sum_{i=1}^{|\Sigma|} v_i^{1,s-1} p_{s_t|s_i}) p_{x_{s+1}|s_t} (p_{x_{s+2}|x_{s+1}} \cdots p_{x_e|x_{e-1}}) = v^{2,s}_{s_t} C^{1,s,e}_{s_t}$$

If s is not the site for a mismatch, then one mismatch occurs in the prefix of length $s-1$ and the other must occur between $s+1$ and e. Such contribution is:

$$\sum_{i=1}^{|\Sigma|} v_i^{1,s-1} p_{x_s|s_i} \sum_{j=1}^{|\Sigma|} p_{x_j|x_s} C^{1,s+1,e}_{s_j} = v^{1,s}_{x_s} C^{1,s,e}_{x_s}$$

In summary:

$$C^{2,s,e} = \begin{bmatrix} (u_1^{2,e} - v^{2,s}_{s_1} C^{1,s,e}_{s_1}) / p_{s_1|x_{s-1}} \\ (u_2^{2,e} - v^{2,s}_{s_2} C^{1,s,e}_{s_2}) / p_{s_2|x_{s-1}} \\ \cdots \\ (u^{2,e}_{x_s} - v^{2,s}_{x_s} C^{0,s,e}_{x_s} - v^{1,s}_{x_s} C^{1,s,e}_{x_s}) / p_{x_s|x_{s-1}} \\ \cdots \\ (u^{2,e}_{|\Sigma|} - v^{2,s}_{s_{|\Sigma|}} C^{1,s,e}_{s_{|\Sigma|}}) / p_{s_{|\Sigma|}|x_{s-1}} \end{bmatrix} \frac{1}{p_{x_1} \cdots p_{x_{s-1}|x_{s-2}}}$$

$1 \ldots s-1$	s	$s+1 \ldots e$
1 mismatch	x	i-2 mismatches
2 mismatches	x	i-3 mismatches
...	x	...
i-1 mismatches	x	0 mismatches

$1 \ldots s-1$	s	$s+1 \ldots e$
1 mismatches	=	i-1 mismatches
2 mismatches	=	i-2 mismatches
...	=	...
i mismatches	=	0 mismatches

4.4 Generalization for i Mismatches

When i mismatches occur, if $s_t \neq x_s$ there is 1 mismatch at position s. The other $i-1$ mismatches can be distributed in any of the $i-1$ combinations (Table - left) that we need to subtract to the components of $U^{k,e}$. If $s_t = x_s$ the ways to distribute the mismatches are k (Table - right). The time complexity for the component $s_t = x_s$ when i mismatches occur needs i operations, while for the remaining components it needs $i-1$. In both cases it is $O(i)$, so the overall complexity is $O(i|\Sigma|)$. Indeed we need to compute the expectation for all $i \leq k$ if we want the expectation for a fixed number of mismatches k. So for the total complexity we need to sum up:

$$\sum_{i=1}^{k} i|\Sigma| = |\Sigma|\frac{k(k+1)}{2} = O(|\Sigma|k^2)$$

If we are interested in the computation of the expectation of all the words in a text this takes $O(kn^2|\Sigma|^2 + nk|\Sigma|^{p+1})$, rather than in $O(n^3|\Sigma|^{p+1})$ obtained by application of Algorithm 1 or of the algorithm in [4] to each word.

References

1. Apostolico, A., Pizzi, C.: Motif Discovery by Monotone Scores. Discrete Applied Mathematics 155(6-7), 695–706 (2007); special issue Computational Molecular Biology Series
2. Bailey, T.L., Williams, N., Misleh, C., Li, W.W.: MEME: Discovering and Analyzing DNA and Protein Sequence Motifs. NAR 34, W369–W373, (2006)
3. Brazma, A., Jonassen, I., Ukkonen, E., Vilo, J.: Predicting Gene Regulatory Elements in Silico on a Genomic Scale. Genome Research 11, 1202–1215 (1998)
4. Boeva, V., Clément, J., Régnier, M., Vandenbogaert, M.: Assessing the significance of sets of words. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 358–370. Springer, Heidelberg (2005)
5. Régnier, M., Vandenbogaert, M.: Comparison of Statistical Significance Criteria. J. Bioinformatics and Computational Biology 4(2), 537–552 (2006)
6. Sandve, K., Drablos, F.: A survey of motif discovery methods in an integrated framework. Biology Direct 1(11) (2006)
7. Sinha, S., Tompa, M.: YMF: a Program for Discovery of Novel Transcription Factor Binding Sites by Statistical Overrepresentation. NAR 31(13), 3586–3588 (2003)
8. Stormo, G.D.: DNA Binding Sites: Representation and Discovery. Bioinformatics 16(1), 16–23 (2000)
9. Tompa, M., et al.: Assessing Computational Tools for the Discovery of Transcription Factor Binding Sites. Nature Biotechnology 23(1), 137–144 (2005)

Consensus Optimizing Both Distance Sum and Radius

Amihood Amir[1], Gad M. Landau[2,⋆], Joong Chae Na[3,⋆⋆],
Heejin Park[4,⋆⋆⋆], Kunsoo Park[5], and Jeong Seop Sim[6]

[1] Bar-Ilan University, 52900 Ramat-Gan, Israel
[2] University of Haifa, Haifa 31905, Israel, and
Polytechnic Institute of NYU, NY 11201-3840, USA
[3] Sejong University, Seoul 143-747, South Korea
jcna@sejong.ac.kr
[4] Hanyang University, Seoul 133-791, South Korea
[5] Seoul National University, Seoul 151-742, South Korea
[6] Inha University, Incheon 402-751, South Korea

Abstract. The consensus string problem is finding a representative string (consensus) of a given set $\mathbb{S}$ of strings. In this paper we deal with the consensus string problems optimizing both distance sum and radius, where the distance sum is the sum of (Hamming) distances from the strings in $\mathbb{S}$ to the consensus and the radius is the longest (Hamming) distance from the strings in $\mathbb{S}$ to the consensus. Although there have been results considering either distance sum or radius, there have been no results considering both as far as we know.

We present two algorithms to solve the consensus string problems optimizing both distance sum and radius for three strings. The first algorithm finds the optimal consensus string that minimizes both distance sum and radius, and the second algorithm finds the bounded consensus string such that, given constants s and r, the distance sum is at most s and the radius is at most r. Both algorithms take linear time.

1 Introduction

The multiple string comparison problem is one of fundamental research topics in computational biology and combinatorial pattern matching [1,9,10]. Finding a representative string of a given set $\mathbb{S}$ of strings, called a *consensus string* (or *closest string* or *center string*), is a major problem in multiple string comparison, which is closely related to the motif recognition problem. Among the conditions

⋆ This work was partially supported by the Israel Science Foundation grant 35/05, the Israel-Korea Scientific Research Cooperation and Yahoo.

⋆⋆ Corresponding author.

⋆⋆⋆ This work was supported by the Korea Foundation for International Cooperation of Science & Technology(KICOS) through a grant provided by the Korean Ministry of Education, Science & Technology(MEST) in 2009 (No. K20717000007-09B0100-00710).

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 234–242, 2009.
© Springer-Verlag Berlin Heidelberg 2009

that a string should satisfy to be accepted as a consensus, the two most important conditions are

1. to minimize the sum of (Hamming) distances from the strings in $\mathbb{S}$ to the consensus, and
2. to minimize the longest distance (or radius) from the strings in $\mathbb{S}$ to the consensus.

In this paper we deal with two related but different problems about finding a consensus string. The first one is finding an *optimal consensus string* minimizing both distance sum and radius as follows.

Problem 1. Optimal consensus
Given a set $\mathbb{S} = \{S_1, \ldots, S_k\}$ of k strings of length n, find a string X (if any) that minimizes both $\sum_{1 \le i \le k} d(X, S_i)$ and $\max_{1 \le i \le k} d(X, S_i)$ where $d(A, B)$ is the Hamming distance between strings A and B.

If such an optimal consensus string exists, it can be accepted as a consensus string of $\mathbb{S}$. However, sometimes such a string does not exist and a string satisfying loose conditions may be sought for as follows.

Problem 2. Bounded consensus
Given a set $\mathbb{S} = \{S_1, \ldots, S_k\}$ of k strings of length n and two positive integers s and r, find a string X (if any) satisfying both $\sum_{1 \le i \le k} d(X, S_i) \le s$ and $\max_{1 \le i \le k} d(X, S_i) \le r$.

Although minimizing both distance sum and radius from a consensus is important, researchers have only focused on finding a consensus minimizing either the distance sum or the radius. Minimizing the distance sum is rather easy. We can find a string X that minimizes the distance sum by selecting the character occurring most often in each position of the strings in $\mathbb{S}$. However, minimizing the radius is a hard problem in general. For general k, the problem of finding a string X such that $\max_{1 \le i \le k} d(X, S_i) \le r$ is NP-hard even when characters in strings are drawn from the binary alphabet [4]. Thus, attention has been restricted to approximation solutions [2,5,6,11,12,13,14] and fixed-parameter solutions [7,8,14,15].

For fixed-parameter solutions, Stojanovic [15] proposed a linear-time algorithm for $r = 1$. Gramm et al. [7,8] proposed the first fixed-parameter algorithm running in $O(kn+kr^{r+1})$ time for finding a string X such that $\max_{1 \le i \le k} d(X, S_i) \le r$. Ma and Sun [14] presented another algorithm running in $O(kn+kr(16|\Sigma|)^r)$ time, where Σ denotes the alphabet. Furthermore, there have been some algorithms for a small constant k. Gramm et al. [7] proposed a direct combinatorial algorithm for finding a string X that minimizes the radius for three strings. Sze et al. [16] showed a condition for the existence of a string whose radius is less than or equal to r. Boucher et al. [3] proposed an algorithm for finding a string X such that $\max_{1 \le i \le 4} d(X, S_i) \le r$ for four binary strings. For brief surveys on approximation solutions, readers are referred to [3,14]. However, as far as we know, there have been no results on finding a consensus string minimizing both distance sum and radius.

In this paper we present the first algorithms to solve the consensus string problems minimizing both distance sum and radius for the set of three strings (i.e., when $k = 3$).

- We present an algorithm to solve the optimal consensus string problem (Problem 1). The algorithm finds a string X that minimizes both distance sum ($\sum_{1 \le i \le 3} d(X, S_i)$) and radius ($\max_{1 \le i \le 3} d(X, S_i)$) if such a string exists. Otherwise, the algorithm returns a string with the minimum distance sum among the strings whose radii are minimum. On top of the powerful functionalities of the algorithm, the algorithm is very efficient. It takes only $O(n)$ time to do all the computation above.
- We present an algorithm to solve the suboptimal consensus problem (Problem 2). The algorithm returns a string X (if any) satisfying both $\sum_{1 \le i \le 3} d(X, S_i) \le s$ and $\max_{1 \le i \le 3} d(X, S_i) \le r$ for given s and r. This algorithm runs in $O(n)$ time. In addition, the algorithm can be modified for faster execution if input strings are given in advance and r and s are given later. The modified algorithm computes the minimum of $\sum_{1 \le i \le 3} d(X, S_i) + \max_{1 \le i \le 3} d(X, S_i)$ for any string X. The minimum can be computed from the input strings even before r and s are given. Later, when r and s are given, the problem can be solved in $O(1)$ by using the minimum. This is very useful when r and s are given later or when several problems with different pairs of r and s are asked on the same input strings.

This paper is organized as follows. In Section 2, we give some definitions and notations. We present our algorithms for the consensus problems in Section 3. Finally we give concluding remarks in Section 4.

2 Preliminaries

For a string S, let $S[i]$ denote the ith character of S. For two strings X and Y, $d(X, Y)$ is defined as the Hamming distance between X and Y. Let $\mathbb{S} = \{S_1, \ldots, S_k\}$ be a set of k strings of equal length n. Given a string X, the *(consensus) radius* of X for $\mathbb{S}$, denoted by $R_{\mathbb{S}}(X)$, is defined as $\max_{1 \le p \le k} d(X, S_p)$ and the *(consensus) distance sum* of X for $\mathbb{S}$, denoted by $E_{\mathbb{S}}(X)$, is defined as $\sum_{1 \le p \le k} d(X, S_p)$. We omit the set notation $\mathbb{S}$ if not confusing. Then, Problem 1 is finding a string X that minimizes both $E(X)$ and $R(X)$, and Problem 2 is finding a string Y such that $E(Y) \le s$ and $R(Y) \le r$. We call a solution of Problem 1 an *optimal consensus string* and a solution of Problem 2 a *bounded consensus string*.

Consider the alignment of a string X and the strings in $\mathbb{S}$. Because the Hamming distance allows only substitutions, $X[i]$ is aligned with $S_p[i]$'s ($1 \le p \le k$). Thus, $\mathbb{S}$ can be regarded as a $k \times n$ character matrix, where the ith column consists of the ith characters of the k strings. For each column, we call the *majority* the character occurring most often and the *minority* the character occurring most seldom.

If we only consider the distance sum, that is, we want to find a string X with the minimum distance sum, X can be found easily by choosing the majority in

each column. However, the problem of finding a string Y such that $R(Y) \leq r$ is NP-hard even when restricted to a binary alphabet [4]. Thus, Problems 1 and 2 are also NP-hard in general.

3 Consensus String for Three Strings

In this section we consider the consensus string problems for $\mathbb{S} = \{S_1, S_2, S_3\}$. We first describe an algorithm for computing a string X with the minimum radius and show that X computed by the algorithm also minimizes the distance sum (Problem 1). Then, we show how to compute a bounded consensus string Y from X (Problem 2).

3.1 String with the Minimum Radius

Consider the alignment of the three strings S_1, S_2, and S_3. The column in every position i is divided into the following five types. See Figure 1.

- Type 0: $S_1[i] = S_2[i] = S_3[i]$ (all matches).
- Type 1: $S_1[i] \neq S_2[i] = S_3[i]$ ($S_1[i]$ is the minority).
- Type 2: $S_2[i] \neq S_1[i] = S_3[i]$ ($S_2[i]$ is the minority).
- Type 3: $S_3[i] \neq S_1[i] = S_2[i]$ ($S_3[i]$ is the minority).
- Type 4: $S_1[i] \neq S_2[i]$, $S_2[i] \neq S_3[i]$, and $S_3[i] \neq S_1[i]$ (all mismatches).

Let c_j ($0 \leq j \leq 4$) denote the number of columns for type j. Without loss of generality, we assume that $c_1 \geq c_2 \geq c_3$.

Let E_{min} be the smallest sum of Hamming distances of S_1, S_2, S_3 from any string X', i.e., $E_{min} = min_{X'} \sum_{1 \leq p \leq 3} d(X', S_p)$. Obviously, the minimum distance sum $E_{min} = c_1+c_2+c_3+2c_4$. Let R_{min} be the smallest max of Hamming distances of S_1, S_2, S_3 from any string X'. i.e., $R_{min} = min_{X'} \max_{1 \leq p \leq 3} d(X', S_p)$. The following lemma gives a lower bound for the minimum radius R_{min}.

Lemma 1. *$R_{min} \geq \max(L_1, L_2)$, where $L_1 = (c_1 + c_2 + c_4)/2$ and $L_2 = (c_1 + c_2 + c_3 + 2c_4)/3$.*

Proof. First, R_{min} is greater than or equal to half of the distance between two farthest strings (i.e., S_1 and S_2) by Hamming distance. That is, $R_{min} \geq (c_1 + c_2 + c_4)/2 = L_1$. Moreover, $R_{min} \geq E_{min}/3 = L_2$. Indeed if there exists a string X' such that $R(X') < E_{min}/3$, then $E(X') < E_{min}$, which is a contradiction. $\square$

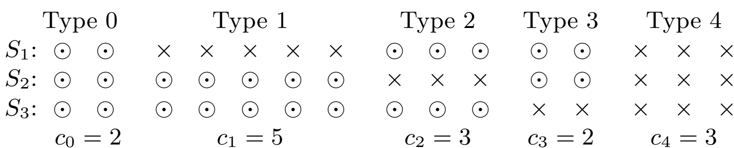


Fig. 1. Types of columns in the alignment of 3 strings, where $\odot$ and $\times$ represent match and mismatch characters at each position, respectively

Remark. Because R_{min} is an integer, L_1 is exactly $\lceil (c_1 + c_2 + c_4)/2 \rceil$ and L_2 is $\lceil (c_1 + c_2 + c_3 + 2c_4)/3 \rceil$. Throughout the paper, the ceiling function or the floor function should be applied to all fractional expressions including L_1 and L_2. For simplicity, however, we assume that values of all fractional expressions are integers.

By comparing the two lower bounds L_1 and L_2, we get the following.

Corollary 1. *$R_{min} \geq L_2$ if $c_1 + c_2 \leq 2c_3 + c_4$, and $R_{min} \geq L_1$ otherwise.*

Thus, if an algorithm computes a string whose radius is L_2 when $c_1+c_2 \leq 2c_3+c_4$ and L_1 when $c_1 + c_2 > 2c_3 + c_4$, the algorithm always computes a string with the minimum radius.

Now we describe how to compute a string X with the minimum radius and show that the radius of X is $\max(L_1, L_2)$. Basically, we select one of $S_1[i]$, $S_2[i]$, and $S_3[i]$ in each position i. We always select the majority in every column of type 0. In columns of other types, we select characters in the following way. We have two cases.

I. When $c_1 + c_2 \leq 2c_3 + c_4$, i.e., $L_1 \leq L_2$.

Let $c_{41} = (c_4 + 2c_1 - c_2 - c_3)/3$, $c_{42} = (c_4 + 2c_2 - c_1 - c_3)/3$, and $c_{43} = (c_4 + 2c_3 - c_1 - c_2)/3$. Obviously, $c_{41} + c_{42} + c_{43} = c_4$. Then, we compute a string X in the following way.

- In every column of Types 0-3, select the majority.
- In columns of type 4, select c_{41} characters of S_1, c_{42} characters of S_2, and c_{43} characters of S_3.

Now, we prove (1) that c_{41}, c_{42}, and c_{43} are nonnegative and (2) that the string X is a string with the minimum radius by showing that its radius is L_2.

- c_{41}, c_{42}, and c_{43} are nonnegative.
  - c_{43} is nonnegative by the condition $c_1 + c_2 \leq 2c_3 + c_4$.
  - c_{42} is nonnegative if inequality $c_1 + c_3 \leq 2c_2 + c_4$ is satisfied. Since we assume that $c_3 \leq c_2$, $c_1 + c_3 \leq c_1 + c_2$ and $2c_3 + c_4 \leq 2c_2 + c_4$, and thus $c_1 + c_3 \leq 2c_2 + c_4$ by the condition $c_1 + c_2 \leq 2c_3 + c_4$.
  - The proof that c_{41} is nonnegative is similar to the proof that c_{42} is nonnegative.
- The radius of X is L_2.

  The distances of strings S_1, S_2, and S_3 from X are as follows:
  - $d(S_1, X) = c_1 + c_{42} + c_{43} = c_1 + (c_4 + 2c_2 - c_1 - c_3)/3 + (c_4 + 2c_3 - c_1 - c_2)/3$ $= (c_1 + c_2 + c_3 + 2c_4)/3 = L_2$.
  - $d(S_2, X) = c_2 + c_{41} + c_{43} = c_2 + (c_4 + 2c_1 - c_2 - c_3)/3 + (c_4 + 2c_3 - c_1 - c_2)/3$ $= (c_1 + c_2 + c_3 + 2c_4)/3 = L_2$.
  - $d(S_2, X) = c_3 + c_{41} + c_{42} = c_3 + (c_4 + 2c_1 - c_2 - c_3)/3 + (c_4 + 2c_2 - c_1 - c_3)/3$ $= (c_1 + c_2 + c_3 + 2c_4)/3 = L_2$.

  Since $d(S_1, X) = d(S_2, X) = d(S_3, X) = L_2$, the radius (i.e. $\max_{1 \leq p \leq 3} d(X, S_p)$) is L_2.

II. When $c_1 + c_2 > 2c_3 + c_4$, i.e., $L_1 > L_2$.

We separate this case into two subcases $c_1 - c_2 < c_4$ and $c_1 - c_2 \geq c_4$.

(a) *When $c_1 - c_2 \leq c_4$.*

Let $c_{41} = (c_4 + c_1 - c_2)/2$, $c_{42} = (c_4 - c_1 + c_2)/2$, and $c_{43} = 0$. Obviously, $c_{41} + c_{42} + c_{43} = c_4$. Then, we compute a string X in the following way.

- In every column of Types 0-3, select the majority.
- In columns of type 4, select c_{41} characters of S_1, c_{42} characters of S_2, and c_{43} characters of S_3.

Now, we prove (1) that c_{41} and c_{42} are nonnegative and (2) that the string X is a string with the minimum radius by showing that its radius is L_1.

- c_{41}, c_{42}, and c_{43} are nonnegative.
  - c_{41} is nonnegative by the assumption $c_1 \geq c_2$.
  - c_{42} is nonnegative by the condition $c_1 - c_2 \leq c_4$.
- The radius of X is L_1.
  The distances of strings S_1, S_2, and S_3 from X are as follows:
  - $d(S_1, X) = c_1 + c_{42} + c_{43} = c_1 + (c_4 - c_1 + c_2)/2 = (c_4 + c_1 + c_2)/2 = L_1$.
  - $d(S_2, X) = c_2 + c_{41} + c_{43} = c_2 + (c_4 + c_1 - c_2)/2 = (c_4 + c_1 + c_2)/2 = L_1$.
  - $d(S_3, X) = c_3 + c_{41} + c_{42} = c_3 + c_4 < L_1$. (One can show $L_1 - (c_3 + c_4) > 0$ using the condition $c_1 + c_2 > 2c_3 + c_4$.)

  Thus the radius of X is L_1.

(b) *When $c_1 - c_2 > c_4$.*

Let $c_{11} = (c_1 + c_2 + c_4)/2$ (nonnegative trivially) and $c_{12} = (c_1 - c_2 - c_4)/2$ (nonnegative due to $c_1 - c_2 > c_4$). Then, we compute a string X in the following way.

- In every column of Types 0, 2, and 3, select the majority.
- In columns of type 1, select c_{11} majority characters and c_{12} minority characters (i.e. characters of S_1).
- In every column of type 4, select the character of S_1.

Now, we prove that the string X is a string with the minimum radius by showing that its radius is L_1.

- $d(S_1, X) = c_{11} = (c_1 + c_2 + c_4)/2 = L_1$.
- $d(S_2, X) = c_{12} + c_2 + c_4 = (c_1 - c_2 - c_4)/2 + c_2 + c_4 = (c_1 + c_2 + c_4)/2 = L_1$.
- $d(S_3, X) = c_{12} + c_3 + c_4 = (c_1 - c_2 - c_4)/2 + c_3 + c_4 = (c_1 - c_2 + 2c_3 + c_4)/2 \leq L_1$. (One can show $L_1 - (c_1 - c_2 + 2c_3 + c_4)/2 \geq 0$ using the assumption $c_2 \geq c_3$.)

Thus, the radius of X is L_1.

Conclusively, the algorithm computes a string with the minimum radius.

Lemma 2. *Given the string set $\mathbb{S}$, a string with the minimum radius for $\mathbb{S}$ can be found in $O(n)$ time.*

Proof. We have already shown that the radius of string X computed by the algorithm is minimum. Consider the time complexity. The types of columns and c_i $(0 \leq i \leq 4)$ can be determined by scanning three strings once. Furthermore, other computations can be done in constant time and character selections in every column can be done by scanning the strings once. Thus, the algorithm takes $O(n)$ time. □

3.2 Optimal Consensus String

Consider the relation between the radius and the distance sum. In cases I and II (a), X is a string with the minimum distance sum as well as with the minimum radius because we select the majority in every column. In case II (b), however, X is not a string with the minimum distance sum. We can decrease the distance sum by reducing the number of minority selections in columns of type 1. If so, however, the radius increases as much as the distance sum decreases. The following lemma shows the relation between the radius and the distance sum in case II (b).

Lemma 3. *In case of II (b), $R(Z) + E(Z) \geq R_{min} + E_{min} + M$ for any string Z, where $M = (c_1 - c_2 - c_4)/2$.*

Proof. Recall that $R_{min} = L_1 = (c_1 + c_2 + c_4)/2$ and $E_{min} = c_1 + c_2 + c_3 + 2c_4$. Let Z be a string such that $E(Z) = E_{min} + t$, where t is the number of minority selections in all columns when constructing Z. Then, we prove this lemma by showing that $R(Z) \geq R_{min} + M - t = c_1 - t$. Let m_j $(1 \leq j \leq 3)$ be the number of minority selections (i.e., characters of S_j) in columns of type j when constructing Z. Obviously, $m_1 + m_2 + m_3 = t$. Let m_{4j} $(1 \leq j \leq 3)$ be the number of characters of S_j selected in columns of type 4 when constructing Z. Then,

$$\begin{aligned} d(S_1, Z) &= c_1 - m_1 + m_2 + m_3 + m_{42} + m_{43} \\ &= c_1 - t + 2m_2 + 2m_3 + m_{42} + m_{43} \text{ (using } m_1 = t - m_2 - m_3) \\ &\geq c_1 - t. \end{aligned}$$

Thus, $R(Z) = \max(d(Z, S_1), d(Z, S_2), d(Z, S_3)) \geq c_1 - t$. □

Corollary 2. *The string X computed by the above algorithm is a string that minimizes $R(X) + E(X)$.*

Lemma 4. *The above algorithm computes an optimal consensus string if exists.*

Proof. We have already shown that the radius of string X is minimum. In cases I and II (a), X is also a string with the minimum distance sum. In case II (b), there is no optimal consensus string by Lemma 3 because $c_1 - c_2 - c_4 > 0$. □

Lemma 5. *There is no optimal consensus string if and only if both $c_1 + c_2 > 2c_3 + c_4$ and $c_1 - c_2 > c_4$ (case II (b)).*

Lemma 6. *If there is no optimal consensus string, the above algorithm computes a string X whose distance sum is smallest among all strings with the minimum radius.*

Proof. The radius of string X is minimum. By Corollary 2, the distance sum of X is smallest among strings whose radius is $R(X)$. □

3.3 Bounded Consensus String

We show how to compute a bounded consensus string Y from X (Problem 2). In cases I and II (a), a solution is easy. Because $R(X) = R_{min}$ and $E(X) = E_{min}$, X is a bounded consensus string if $R(X) \leq r$ and $E(X) \leq s$, and there is no bounded consensus string otherwise. Consider case II (b). If $R(X) > r$, $E(X) > s$, or $R(X) + E(X) > r + s$ (by Corollary 2), there is no bounded consensus string. Otherwise, we can find a bounded consensus string by decreasing the number of minority selections when constructing X.

Lemma 7. *A bounded consensus string can be found in $O(n)$ time if exists.*

Lemma 8. *Let $M = (c_1 - c_2 - c_4)/2$ if $c_1 + c_2 > 2c_3 + c_4$ and $c_1 - c_2 > c_4$ (case II (b)), and $M = 0$ otherwise. Then, there exists a bounded consensus string if and only if $R_{min} \leq r$, $E_{min} \leq s$, and $R_{min} + E_{min} + M \leq r + s$,*

By Lemmas 2, 4 and 8, we get the following theorem.

Theorem 1. *Problems 1 and 2 for three strings can be solved in $O(n)$ time.*

4 Concluding Remarks

We considered the consensus string problem optimizing both distance sum and radius, and proposed a linear-time algorithm for three strings. Moreover, we studied the conditions for which there exists an optimal consensus string or a bounded consensus string for three strings. It remains an open problem to find a consensus string for $k \geq 4$ strings. Another open problem is to find a consensus string when strings are compared by the edit distance. This problem doesn't look easy even for three strings.

References

1. Altschul, S., Lipman, D.: Trees, stars, and multiple sequence alignment. SIAM Journal on Applied Mathematics 49, 197–209 (1989)
2. Ben-Dor, A., Lancia, G., Perone, J., Ravi, R.: Banishing bias from consensus sequences. In: Hein, J., Apostolico, A. (eds.) CPM 1997. LNCS, vol. 1264, pp. 247–261. Springer, Heidelberg (1997)
3. 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)

4. Frances, M., Litman, A.: On covering problems of codes. Theory of Computing Systems 30(2), 113–119 (1997)
5. 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)
6. Gasieniec, L., Jansson, J., Lingas, A.: Approximation algorithms for Hamming clustering problems. Journal of Discrete Algorithms 2(2), 289–301 (2004)
7. 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)
8. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37(1), 25–42 (2003)
9. Gusfield, D.: Algorithms on Strings, Tree, and Sequences. Cambridge University Press, Cambridge (1997)
10. Karp, R.M.: Mapping the genome: some combinatorial problems arising in molecular biology. In: Proceedings of the 25th Annual ACM Symposium on Theory of Computing, pp. 278–285 (1993)
11. 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)
12. 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)
13. Li, M., Ma, B., Wang, L.: On the closest string and substring problems. Journal of the ACM 49(2), 157–171 (2002)
14. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. In: Proceedings of the 12th Annual International Conference on Research in Computational Molecular Biology, pp. 396–409 (2008)
15. Stojanovic, N., Berman, P., Gumucio, D., Hardison, R., Miller, W.: A linear-time algorithm for the 1-mismatch problem. In: Proceedings of the 5th International Workshop on Algorithms and Data Structures, pp. 126–135 (1997)
16. 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)

Faster Algorithms for Sampling and Counting Biological Sequences

Christina Boucher

David R. Cheriton School of Computer Science,
University of Waterloo
cabouche@cs.uwaterloo.ca

Abstract. A set of sequences S is *pairwise bounded* if the Hamming distance between any pair of sequences in S is at most $2d$. The CONSENSUS SEQUENCE problem aims to discern between pairwise bounded sets that have a consensus, and if so, finding one such sequence s^*, and those that do not. This problem is closely related to the motif-recognition problem, which abstractly models finding important subsequences in biological data. We give an efficient algorithm for sampling pairwise bounded sets, referred to as MarkovSampling, and show it generates pairwise bounded sets uniformly at random. We illustrate the applicability of MarkovSampling to efficiently solving motif-recognition instances. Computing the expected number of motif sets has been a long-standing open problem in motif-recognition [1,3]. We consider the related problem of counting the number of pairwise bounded sets, give new bounds on number of pairwise bounded sets, and present an algorithmic approach to counting the number of pairwise bounded sets.

1 Introduction

Given a number of DNA sequences, *motif-recognition* is the task of discovering similar subsequences without prior knowledge of the consensus or their placement within the sequence. The following combinatorial formulation of motif-recognition is due to Pevzner and Sze [16]: let $S = \{S_1, \ldots, S_n\}$ be a set of m-length sequences, and s^* be the *consensus sequence*, a fixed and unknown sequence of length l that is contained in each S_i as a subsequence but is corrupted with at most d substitutions. The aim is to determine s^* and the location of the motif instances in each sequence.

Motif-recognition is NP-complete and thus, unlikely to be solved in polynomial time, unless P = NP [9]. Li *et al.* proved the existence of a PTAS for motif-recognition, however, this result is only of theoretical interest due to the high degree in the polynomial complexity of the associated algorithm [13]. Nonetheless, there are numerous algorithms developed to solve specific instances of the problem, including PROJECTION [1], Winnower [16], pattern driven approaches [19], MITRA [8], PSM1 [17], PMSprune [3], the Voting algorithm [2] and several others.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 243–253, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Closely related to motif-recognition is the CONSENSUS SEQUENCE[1] problem that is defined as follows: given a parameter d and a set of sequences $S = \{s_1, \ldots, s_n\}$ each of length l, does there exist a (consensus) sequence s^* that has Hamming distance at most d from each sequence in S. We denote the Hamming distance between sequences s_i and s_j as $H(s_i, s_j)$. CONSENSUS SEQUENCE is NP-complete even when interest is restricted to the binary alphabet [10]. A set of sequences S is *pairwise bounded* if the distance between any pair sequences s_i and s_j in S is at most $2d$. Hence, the CONSENSUS SEQUENCE problem essentially reduces to separating pairwise bounded sets with a consensus, and if so, finding one such sequence s^*, from those that do not. A set of sequences S is a *motif set* if there exists a consensus sequence, s^*; S is a *decoy set* if it is pairwise bounded but does not have a consensus.

Improving upon existing algorithms for CONSENSUS SEQUENCE would assist in developing algorithms that solve the combinatorial model of motif-recognition. If the *weight* of a set of sequences S, defined as the sum of the Hamming distance of each pair of sequences in S (*i.e.* $\sum_{\forall \{s_i, s_j\} \in S} H(s_i, s_j)$), can heuristically determine whether S is a motif set then we could significantly improve upon many existing motif-recognition algorithms. Crucial for this heuristic to solve CONSENSUS SEQUENCE is knowledge of the probability distribution of the weight of a random motif set and that of a random decoy set; a separation between these distributions is necessary in order to use the weight as an indicator to the existence of a consensus sequence. There exists a trivial algorithm to generate valid motif sets – simply choose any l-length sequence as the consensus sequence and sample with replacement from the set of all sequences that are of distance at most d from that sequence. Unfortunately, there does exist a methodology to determine the probability distribution of the weight of a random decoy set, nor does there exist an efficient algorithm that solves the related problem of generating pairwise bounded sets uniformly at random (u.a.r.), or even near-uniformly at random. The existence of an algorithm to generate pairwise bounded sets u.a.r. could be used to determine the probability distribution of the weight of a random decoy set by sampling pairwise bounded sets u.a.r. and rejecting the motif sets.

The focus of this paper is on the investigation of sampling and counting pairwise bounded sets. Efficiently counting and sampling elements from a complicated distribution is a well-investigated area and has been instrumental in the study of several NP-complete problems, including the sampling and counting versions of the following problems: graph colouring [5,12,14], perfect matchings in a graph [18], Hamiltonian path [6], knapsack problems [4], and independent set [7].

Many motif-recognition programs (including PROJECTION [1], Winnower [16], PSM1 [17], PMSi, and PMSP [3]) enumerate through all, or almost all, pairwise bounded sets and therefore, the efficiency of these applications is closely tied to the number of pairwise bounded sets. An important open problem is to

[1] CONSENSUS SEQUENCE problem is also referred as the RADIUS DECISION problem [10] and the CLOSEST STRING problem [11].

determine a bound on the expected number of pairwise bounded sets. A related problem of determining the expected number of motif sets has been studied by Buhler and Tompa [1] and Davila *et al.* [3]. We consider the following counting problem:

#PAIRWISE BOUNDED
INSTANCE: Parameters n, l and d.
OUTPUT: The number of distinct pairwise bounded sets of sequences.

We denote $\mathcal{S}(n, l, d)$ as the number of distinct pairwise bounded sets with respect to the parameters n, l, and d. We give new deterministic bounds on $\mathcal{S}(n, l, d)$ and discuss an algorithmic method that approximately counts the number of pairwise bounded sets.

To our knowledge, there does not exist any known methods to generate pairwise bounded sets u.a.r. We present a first approach to sampling pairwise bounded sets that is based on random walks. Intuitively, the problem with using a random walk in order to sample pairwise bounded sets is that the probability distribution of the specific sequences is unknown. We give an improvement to the standard random walk technique that mitigates this effect and show this method efficiently produces a uniform sample. We show how this sampling algorithm can be used to algorithmically count the number of pairwise bounded sets and give new bounds on the number of pairwise bounded sets.

2 Sampling Pairwise Bounded Sets

2.1 An Inefficient Sampling Method

The most natural sampling method to generate observations from a distribution is that of *rejection sampling*, where an element is generated at random from a sample space containing the target sample, it is determined whether the set is contained in the target sample and if not, it is omitted from the sample. Rejection sampling is most efficient when the number of elements to be rejected is minimized. The following rejection sampling method samples pairwise bounded sets u.a.r.: select n random sequences from the set of all $|\Gamma|^l$ sequences, if the set is pairwise bounded then include it in the sample, and otherwise reject it. Although, rejection sampling generates pairwise bounded sets u.a.r.this method is intractable when n, l, and d become significantly large. Unfortunately, to determine the exact inefficiency of rejection sampling we are required to know something about the probability that a set of sequences is a pairwise bounded set, which is the very question we aim to answer. Without further knowledge about the probability of the occurrence of pairwise bounded sets, it is unlikely a tight bound on the efficiency of this sampling method can be determined.

2.2 A Sampling Method Using Random Walks

We develop a sampling method, referred to as *MarkovSampling*, that restricts the sampling to pairwise bounded sets. It will be useful to view a set of l-length sequences of size n as a matrix where each column is a set of n symbols;

hence, we denote a set of n sequences as $[c_1 c_2 \dots c_l]$, where each c_i is a n-length column vector. Rather than constructing a pairwise bounded set by selecting n sequences, as in rejection sampling, we generate the matrix column by column while ensuring the pairwise bounded condition holds.

Given columns $c_1, \dots, c_k$ where $k \leq l$, we let $X_{i,j}$ be equal to 1 if the distance between sequences s_i and s_j up to the first k columns is equal to $2d$; otherwise we let $X_{i,j}$ be equal to 0. We define a graph G such that exists a vertex v_i for each sequence s_i, and an edge between v_i and v_j if and only if $X_{i,j} = 1$. The $(k+1)th$ column, c_{k+1} must have the property that if $X_{i,j} = 1$ then s_i and s_j are equal to the same alphabet symbol in c_{k+1} (otherwise the distance between s_i and s_j will go above $2d$) and if $X_{i,j} = 0$ then it does not matter if s_i and s_j mismatch in c_{k+1}. Therefore, a column vector c_{k+1} is feasible to add if and only if s_i and s_j match whenever v_i and v_j are in the same component of G. It is trivial to generate a random feasible column c: just find the components of G and randomly assign any symbol of Γ to each component. All these computations can be done in time polynomial in n and d.

Unfortunately, this does not solve the sampling problem right away. We augment this approach slightly to obtain a more uniform sample. Given that k columns have been obtained, the probability s_i and s_j will match at the $k+1$ column is $1/|\Gamma|$ if $X_{i,j} = 0$. In rejection sampling, a l-length sequence s is generated at random and n sequences are selected at random from the set of all sequences that have Hamming distance at most $2d$ away from s; the probability that sequences s_i and s_j match, which we denote as $p_{i,j}$, is $\left(\frac{(l-d_i)(l-d_j)}{l^2}\right) + \left(\frac{d_i d_j}{l^2(|\Gamma|-1)}\right)$, where $H(s_i, s) = d_i$ and $H(s_j, s) = d_j$. We augment the sampling algorithm defined above as follows: initially, we select a random d_i for each s_i within the set $\{0, \dots, 2d\}$ then columns are generated as above but instead of any two sequences s_i and s_j mismatching with probability $1/|\Gamma|$ (when $X_{i,j} = 0$), they mismatch with probability $1 - p_{i,j}$. When $X_{i,j}$ becomes equal to 1 we replace d_i and d_j with $\lfloor (d_i + d_j)/2 \rfloor$. We define this procedure of generating a pairwise bounded set as *GeneratePairwiseBoundedSet*.

The main drawback of this approach is that we do not know the probability distribution of the columns. GeneratePairwiseBoundedSet will generate a random pairwise bounded set according to a distribution but the resulting distribution will not necessarily be u.a.r. Instead, we iterate GeneratePairwiseBoundedSet N times and construct a generation tree, denoted as T, containing all N pairwise bounded sets. Level i of T corresponds to column number i (*i.e.* level 0 corresponds to the root of T when there exists no columns yet generated, level 1 is the first column). Each time column i is generated in GeneratePairwiseBoundedSet it is added to level i of T. After N pairwise bounded samples are generated and added to T, a pairwise bounded set can then be generated at random by taking a random walk on this generation tree. This is a Markov chain where, at each step the number of neighbours can be calculated in an efficient way. We refer to this sampling method as *MarkovSampling*, where k pairwise bounded sets are generated in $O(n^2 lN + kl)$ time. Next, we consider the number of samples required to ensure T is well representative of the set of all pairwise

Table 1. Data illustrating the difference mean and standard deviation of the samples produced by MarkovSampling and rejection sampling

(n, l, d, N)	μ	σ	μ_{MS}	σ_{MS}	(n, l, d, N)	μ	σ	μ_{MS}	σ_{MS}
(10, 10, 3, 100)	142	23.8	142	56.6	(20, 10, 3, 100)	549	65.7	542	182
(10, 10, 3, 500)	143	22.6	143	42.2	(20, 10, 3, 500)	549	66.3	547	99
(10, 10, 3, 1000)	142	22.9	143	22.3	(20, 10, 3, 1000)	549	66.1	548	67.4
(10, 10, 3, 1500)	143	22.3	142	23.9	(20, 10, 3, 1500)	548	63	549	66.3
(10, 10, 3, 2000)	143	24	142	23.6	(20, 10, 3, 2000)	549	66.1	549	66.4
(10, 15, 4, 100)	194	30.4	192	50.7	(20, 15, 4, 100)	740	85.7	742	150.2
(10, 15, 4, 500)	194	31.7	194	43.1	(20, 15, 4, 500)	742	87.7	744	98.2
(10, 15, 4, 1000)	191	32.1	193	31	(20, 15, 4, 1000)	739	85.6	740	84.5
(10, 15, 4, 1500)	194	30.7	194	30	(20, 15, 4, 1500)	739	86	739	85.5
(10, 15, 4, 2000)	193	30.7	192	31.4	(20, 15, 4, 2000)	740	85.4	739	86.2
(10, 18, 6, 100)	322	41.6	324	81.2	(20, 18, 6, 100)	1305	125	1302	221
(10, 18, 6, 500)	321	41.7	321	75	(20, 18, 6, 500)	1295	125	1298	180.1
(10, 18, 6, 1000)	324	41.5	325	41	(20, 18, 6, 1000)	1298	126	1296	124.1
(10, 18, 6, 1500)	325	42.9	324	40.5	(20, 18, 6, 1500)	1296	126.8	1294	125.3
(10, 18, 6, 2000)	321	41.3	322	42.4	(20, 18, 6, 2000)	1300	125.2	1296	126.6

bounded sets. In this next Section we show experimentally that MarkovSampling achieves a uniform sample of pairwise bounded sets for relatively small value of N (*i.e.* $N = 1000$).

Algorithm 1. MarkovSampling

Input: parameters N, n, l, d, and k.
Output: a set of k pairwise bounded sets
1. Run GeneratePairwiseBoundedSet to generate a tree T of N pairwise bounded sets.
2. Randomly walk from the root of T down to a leaf of T, by selecting a neighbour at random at each step.
3. Repeat step 2 k times.

2.3 Experimental Evaluation

We compare the distribution of the weight of the sample sets produced by rejection sampling and MarkovSampling, and show the existence of a dichotomy in the distributions of the weight of a random motif set and that of a random decoys set. Let μ denote the mean weight for the sample of pairwise bounded sets produced by rejection sampling, and σ denote the associated standard deviation. We define μ_{MS} and σ_{MS} similarly for the sample of pairwise bounded sets produced by MarkovSampling. For several values of l, d and n we varied N and generated 1000 samples using both MarkovSampling (with the associated parameter N) and rejection sampling, and calculated μ and σ, and μ_{MS} and σ_{MS}.

Table 1 shows the change in μ_{MS} and σ_{MS} as N increases; rejection sampling is not dependent on N and therefore, we expect no change in μ and σ as N changes. Table 1 illustrates three sets of data: $l = 10$ and $d = 3$, when $l = 15$ and $d = 4$, and when $l = 18$ and $d = 6$. For each (l, d) pair, we consider when the number of sequences (*i.e.* parameter n) is equal to 10 and 20. The data show that when N is equal to 1000 the difference between the μ and μ_{MS}, and σ and σ_{MS} is minimal, for when $n = 10$ and $n = 20$. For values of N smaller than 1000, the standard deviation for MarkovSampling (*i.e.* σ_{MS}) was significantly larger than the standard deviation computed for rejection sampling (*i.e.* σ). For different values of l, d and n this trend is also witnessed. Hence, for the remainder of our experiments we set N equal to 1000.

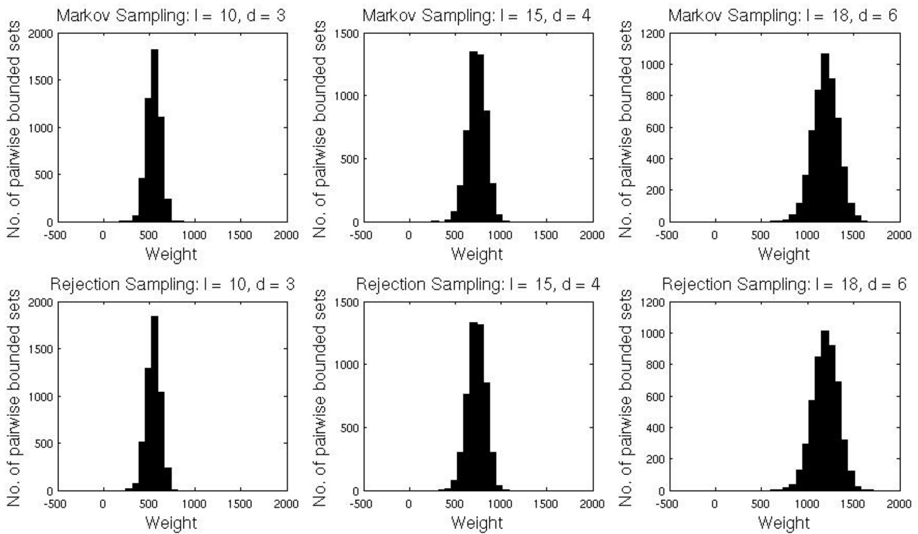


Fig. 1. An histogram of the weights of the 5000 samples generated by MarkovSampling and rejection sampling. The values of l and d were varied and the size of the set of sequences n was set to 20.

We consider the distribution of the weight of samples produced by MarkovSampling with $N = 1000$ and rejection sampling. For several values of l and d, and $n = 20$ we generated 5000 pairwise bounded sets using the two sampling methods and considered the distribution of the weights of the pairwise bounded sets; Figure illustrates these distributions. For all values of n, l, and d the difference between the distribution of the weights for the samples produced by rejection sampling and MarkovSampling were minimal; the probability distribution of the weight of a random pairwise bounded set produced by MarkovSampling was indistinguishable from that of rejection sampling. There was no noticeable difference in the mean weight computed for MarkovSampling and rejection sampling.

2.4 Application of MarkovSampling to Motif-Recognition

A key motivation for the development of algorithms to generate pairwise bounded sets u.a.r. is the development of efficient algorithms for motif-recognition. Majority of motif-recognition programs detect "candidate" motif sets then use a refinement stage to determine if any candidate set is a valid motif set. Almost all motif-recognition programs require solving the CONSENSUS SEQUENCE problem a number of times. In the previous section, we showed the existence of a separation between the probability distribution of the weight of a random motif set and that of a random decoy set. By exploiting this separation we can efficiently find motif sets – namely, the separation would allow decoy sets and motifs sets to be quickly discerned by using the weight as an indicator. The weight can trivially be calculated in $O(n^2 l)$ time.

For varied values of n, l and d, we used MarkovSampling to generate 1000 valid motif sets and 1000 decoy sets. More specifically, to generate a motif set (or decoy set) using MarkovSampling we generated a pairwise bounded set, determine whether it contains a consensus sequence, and include the set in the sample if a consensus sequence exists (or does not exist). We investigated when (l, d) is equal to $(15, 4)$ and $(18, 6)$, for each (l, d) we set n to be 20 and then 50. Figure 2 illustrates this data. Clearly, a dichotomy exists between the distribution of the weight of valid motif sets and decoy sets. As the value of n increases, the separation between the distributions becomes more prevalent the distribution becomes more centralized around the mean; an increase in the value of n leads to an increase in certainty that the weight can be used as an indicator. When n is even moderately large the weight can determine if the set is a motif set with high accuracy.

3 Determining the Number of Pairwise Bounded Sets

3.1 Analytical Bounds on the Number of Pairwise Bounded Sets

Buhler and Tompa [1] determined an estimation for the expected number of motifs that will occur in a given motif-recognition instance; given the parameters n, m, l, and d, the expected number of such sets was estimated to be $4^l(1 - (1 - p_{l,d})^{m-l+1})^m$, where $p_{l,d}$ is defined as $\sum_{i=0}^{d} \binom{l}{i} \left(\frac{3}{4}\right)^i \left(\frac{1}{4}\right)^{l-i}$. The above formula does not take into account that motif instances may overlap and therefore, is only estimation; this estimate is not accompanied by any approximation guarantee. Nonetheless, it is claimed to give an estimate of the computational difficulty of specific motif-recognition instances [1].

As previously mentioned, many motif-recognition programs must enumerate through all (or almost all) pairwise bounded sets and therefore, determining the expected number of pairwise bounded sets is an important aspect of determining computational hardness of specific motif-recognition instances. We consider the problem of determining an accurate bound on the number of pairwise bounded sets, with respect to n, l, and d. An accurate count of the number of pairwise bounded sets will, in turn, lead to determining the expected number of pairwise

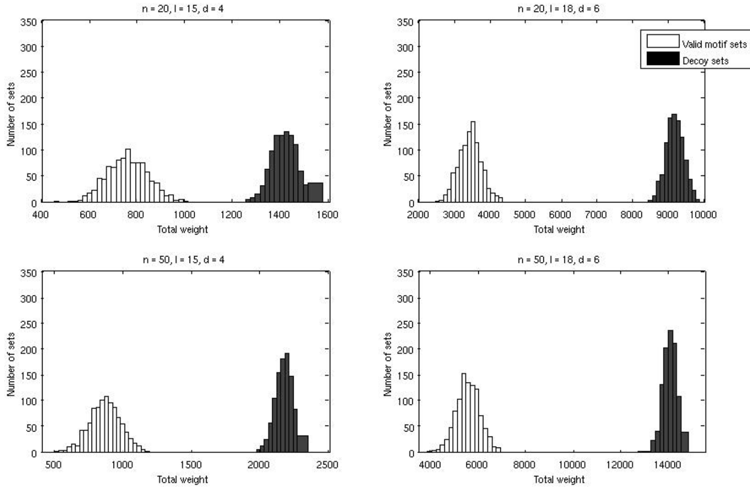


Fig. 2. An illustration of the distribution of the weights of the 1000 valid motif sets and 1000 decoy sets for varied values of n, l, and d. The histograms in black (white) represent the distribution of the weights of decoy sets (motif sets).

bounded sets in a given data set. Proposition 1 gives some initial bounds on the number of pairwise bounded sets. Studying the existence of tighter bounds on $\mathcal{S}(n, l, d)$ warrants further investigation. We note that *mod* denotes the modulo function

Proposition 1. *Let $\mathcal{S}(n, l, d)$ be the number of unique pairwise bounded sets for given values of n, l, d, and alphabet Γ. Then $\mathcal{S}(n, l, d)$ is at least*

$$|\Gamma|^l \left(\lfloor l/2d \rfloor \sum_{i=1}^{2d} \binom{2d}{i} (|\Gamma| - 1)^i + \sum_{i=1}^{\bar{d}} \binom{\bar{d}}{i} (|\Gamma| - 1)^i + 1 \right)$$

where $\bar{d} = l \bmod 2d$ and is at most $|\Gamma|^l \left(\sum_{i=0}^{2d} \binom{l}{i} (|\Gamma| - 1)^i \right)^{n-1}$.

Proof. The upper bound for $\mathcal{S}(n, l, d)$ is obtained by considering the number of possible sets obtained by choosing one sequence s at random (from the set of all l-length sequences) then randomly selecting $n - 1$ sequences from the set of all sequences that have distance at most $2d$ from s. Trivially, this encompasses the set of all pairwise bounded sets since any pairwise bounded set can be described in this manner.

The lower bound requires more careful construction to ensure that we count each unique set only once. There exists $\lfloor l/2d \rfloor$ non-overlapping partitions of the l positions into $2d$ size blocks. We choose a sequence s at random from all $|\Gamma|^l$ possible l-length sequences, and count the number sequences that have between 1 and $2d$ mismatches with s in one of the associated blocks. We restrict the

mismatches to be only within one of the $2d$-size partitions since we want to ensure we do not over-count. Therefore, since there are $\lfloor l/2d \rfloor$ possible partitions, there exists at least $\lfloor l/2d \rfloor |\Gamma|^l \sum_{i=1}^{2d} \binom{2d}{i}(|\Gamma|-1)^i$ pairwise bounded sets. In addition, there exists $|\Gamma|^l$ possible sets where all n sequences match and hence, to obtain a tighter lower bound we add $|\Gamma|^l$.

$$\mathcal{S}(n,l,d) \geq |\Gamma|^l \left(\lfloor l/2d \rfloor \sum_{i=1}^{2d} \binom{2d}{i} (|\Gamma|-1)^i + 1 \right)$$

We account for the $l \ mod \ 2d$ remaining positions. Similarly, there exists at least $|\Gamma|^l \sum_{i=1}^{\bar{d}} \binom{\bar{d}}{i}(|\Gamma|-1)^i$ pairwise bounded sets that contain at least one mismatch in the $l \ mod \ 2d$ remaining positions, where $\bar{d} = l \ mod \ 2d$. We obtain the required lower bound by adding this last term to our previous count. □

3.2 Algorithmically Counting Pairwise Bounded Sets

There exists little information as to the number of decoy sets or the number of pairwise bounded sets. This lack of information is partially due to the fact that there currently does not exist a combinatorial characterization of a pairwise bounded set beyond the rudimentary combinatorial definition. We consider algorithmic approaches to solving #Consensus Sequence. We develop an approximate counting algorithm, referred to as *ApproxCount*, using MarkovSampling. An advantage of this method of using sampling algorithms is that the running time and accuracy can be adjusted easily be adjusted.

Algorithm 2. ApproxCount

Input: T obtained from ConstructGenerationTree
Output: estimation of upper and lower bounds to $\mathcal{S}(n,l,d)$
Repeat N times:
  Random walk from the root of T down to a leaf
  Calculate $\prod_{i=1}^{n} \delta_i$, $\delta_1, \delta_2, \ldots \delta_n$ be the degrees of the vertices encountered
  Calculate α_l and α_h
  Store the values $\alpha_l \cdot \prod_{i=1}^{n} \delta_i$ and $\alpha_h \cdot \prod_{i=1}^{n} \delta_i$ in *lower* and *upper*, respectively.
Return the average of the values stored in *lower* and *upper*.

The sampling algorithm described in Section 2.2 will generate a tree structure T, which represents the sample set of sequences obtained in the sampling algorithm. The number of pairwise bounded sets can be estimated by computing several random walks from the root of T to a leaf and determining the degrees $\delta_1, \ldots, \delta_n$ of the vertices as encountered. Since all $|\Gamma|^n$ sets of n symbols are possible choices for the $2d$ columns without the pairwise bounded condition being violated, a tree representing all pairwise bounded sets is such that all vertices at levels 0 to $2d$ will have degree $|\Gamma|^n$. Hence, to estimate the number of pairwise bounded sets from T, we multiple $\prod_{i=1}^{n} \delta_i$ by appropriate values based on the degrees of the vertices witnessed; for each random walk from the root of T to a

leaf, let α_l be the minimum value in the set $\{|\Gamma|^n/\delta_1, |\Gamma|^n/\delta_2, \ldots, |\Gamma|^n/\delta_{2d}\}$ and similarly, α_h be the maximum value in the set $\{|\Gamma|^n/\delta_1, |\Gamma|^n/\delta_2, \ldots, |\Gamma|^n/\delta_{2d}\}$. We obtain an estimate of the lower and upper bounds for the number of pairwise bounded sets based on this random walk as: $\alpha_l \cdot \prod_{i=1}^{n} \delta_i$ and $\alpha_h \cdot \prod_{i=1}^{n} \delta_i$, respectively. In order to obtain tight bounds we repeat this random walk procedure N times, each time producing the values $\alpha_l \cdot \prod_{i=1}^{n} \delta_i$ and $\alpha_h \cdot \prod_{i=1}^{n} \delta_i$, and take an average of all the estimates for the upper and lower bounds.

4 Conclusions and Future Work

In this paper, we develop an efficient algorithm for generating pairwise bounded sets, and empirically show that the distribution of the weights of the sample sets produced by MarkovSampling is indistinguishable from that of rejection sampling. MarkovSampling illustrates the existence of a dichotomy between the distribution of the weight of a random motif set and that of a random decoy set– such a separation can be exploited to efficiently solve Consensus Sequence. We discuss counting the number of pairwise bounded sets, prove bounds on the number of pairwise bounded sets, and develop an algorithmic method to count the number of pairwise bounded sets. Our focus has been on the combinatorial model of motif-recognition and while this is indeed an important and well-investigated problem, further investigation is needed into the applicability of this model in discovering motifs in real biological data.

This is the first study on this topic that the authors are aware of and, as such, there exists many practical and theoretical problems left for open for investigation. Proof of the existence (or non-existence) of a *fully polynomial almost uniform sampler (FPAUS)* for sampling pairwise bounded sets u.a.r. warrants investigation. The development of a rapidly mixing *Markov chain Monte Carlo (MCMC)* algorithm could show the existence of an algorithm for u.a.r. generation of pairwise bounded sets and should be further explored. MCMC methods have been successful in producing a sampling method for some combinatorial sampling problems. Lastly, proving the existence of a tighter (upper or lower) bound on the number of pairwise bounded sets remains open.

Acknowledgements

The author is grateful to Nick Wormald for his discussions and insights concerning the results presented in this paper. Further, I would like to gratefully acknowledge research support of the National Sciences and Engineering Research Council of Canada.

References

1. Buhler, J., Tompa, M.: Finding motifs using random projections. J. Comp. Bio. 9(2), 225–242 (2002)
2. Chin, F.Y.L., Leung, C.M.: Voting algorithms for discovering long motifs. In: Proc. APBC 2005, pp. 261–271 (2005)

3. Davila, J., Balla, S., Rajasekaran, S.: Fast and practical algorithms for planted (l, d) motif search. IEEE/ACM Trans. Comput. Biol. Bioinf. 4(4), 544–552 (2007)
4. Dyer, M.: Approximate counting by dynamic programming. In: Proc. STOC 2003, pp. 693–699 (2003)
5. Dyer, M., Frieze, A.: Randomly colouring graphs with lower bounds on girth and maximum degree. In: Proc. FOCS 2001, pp. 579–587 (2001)
6. Dyer, M., Frieze, A., Jerrum, M.: Approximately counting Hamilton paths and cycles in dense graphs. SIAM J. Comput. 27(5), 1262–1272 (1998)
7. Dyer, M., Frieze, A., Jerrum, M.: On counting independent sets in sparse graphs. In: Proc. FOCS 1999, pp. 210–217 (1999)
8. Eskin, E., Pevzner, P.A.: Finding composite regulatory patterns in DNA sequences. Bioinformatics 18(1), 354–363 (2002)
9. Evans, P.A., Smith, A., Wareham, H.T.: On the complexity of finding common approximate substrings. Th. Comp. Sci. 306, 407–430 (2003)
10. Frances, M., Litman, A.: On covering problems of codes. Th. Comp. Sys. 30, 113–119 (1997)
11. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica, 37, 25–42
12. Hayes, T.P., Vigoda, E.: A non-Markovian coupling for randomly sampling colorings. In: Proc. FOCS 2003, pp. 618–627 (2003)
13. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. J. Comp. and Sys. Sci. 65(1), 73–96 (2002)
14. Molloy, M.: The glauber dynamics on colorings of a graph with high girth and maximum degree. In: Proc. STOC 2002, pp. 91–98 (2002)
15. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
16. Pevzner, P., Sze, S.: Combinatorial approaches to finding subtle signals in DNA sequences. In: Proc. ISMB 2000, pp. 344–354 (2000)
17. Rajasekaran, S., Balla, S., Huang, C.H.: Exact algorithms for the planted motif problem. J. Comp. Bio. 12(8), 1117–1128 (2005)
18. Sinclair, A., Jerrum, M.: Approximate counting, uniform generation and rapidly mixing. Inform. and Comput. 82, 93–133
19. Sze, S., Lu, S., Chen, J.: Integrating sample-driven and patter-driven approaches in motif finding. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 438–449. Springer, Heidelberg (2004)

Towards a Theory of Patches

Amihood Amir[1,2,⋆] and Haim Parienty[1]

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

Abstract. Many applications have a need for indexing unstructured data. It turns out that a similar ad-hoc method is being used in many of them - that of considering small particles of the data.

In this paper we formalize this concept as a tiling problem and consider the efficiency of dealing with this model. We present an efficient algorithm for the one dimension tiling problem, and prove the two dimension problem is hard. We then develop an approximation algorithm with an approximation ratio converging to 2. We show that the "one-and-a-half" dimensional version of the problem is also hard.

1 Motivation

The proliferation of digital data is staggering. Even with the speed of current computers, sequential search is impossible in many applications. If efficient indexing techniques are not available, the data is, for all intents and purposes, lost.

Dictionaries and concordances were in use by scholars for generations. By the late 1940's the field of *Information Retrieval* was created. For many years that information was mostly textual, and a large body of scientific work has been established in the field [21,8,17]. With the advent of Computational Biology, digital libraries, and the Web, indexing of non-textual data is becoming increasingly more crucial. Some examples are provided below.

Computational Biology

The three dimensional structure of the protein plays an important role in its functionality and as F. Cohen [2] writes "... similar sequences yield similar structures, but quite distinct sequences can produce remarkably similar structures" [10]. Searching the growing database of protein structures for structure similarity is, therefore, an important task. In essence, a good indexing method is necessary. Unfortunately, the rate of growth of the protein database exceeds the rate of development of indexing methods. The processes that most state-of-the-art methods use to-date extract various local features, such as curvature or torsion angles, and index by these features. This is both time-consuming and limited by the selected features. Current methods can not efficiently index protein structure for more than a few thousands proteins. Consequently, the methods aggregate

⋆ Partly supported by ISF grant 35/05.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 254–265, 2009.
© Springer-Verlag Berlin Heidelberg 2009

proteins with some rough similarity, find a group that is "roughly similar" to the given protein and then check within this group. The disadvantage of this method is that it blurs local features that may actually be important, and thus may point the search algorithm to the wrong group. Recently, a new approach (for example [13]) is taken, where the protein structure is represented as letter strings using structural alphabets.

Linguistics. *Lexical categorization* is an important task of Natural Language Processing. The idea is to correctly tag parts of speech (e.g. as verbs, nouns). Sets of constraints have been suggested as possible aids in the task of lexical categorization [12]. Parikh mappings have been used for identifying such sets [1]. (A *Parikh mapping* is a function counting for each letter of an alphabet the occurrences of this letter in a word w [19].) In seeking the Parikh mapping, one is interested in the *content* of a substring, but in a **scrambled** order. In [1] some interesting techniques were developed, and it was apparent that problems that have known efficient solutions in the traditional pattern matching context, are still open for exploration. In particular, can a text be efficiently indexed for Parikh mapping?

Computer Vision

Indexing images is one of the important challenges of web retrieval. Currently, image searches in all search engines are actually textual searches. The image captions are indexed and not the images themselves. Not only is it impossible to scan a picture and ask to find all "similar" pictures, but even such a mundane task as, given a picture, finding the image from which the given picture has been cropped, is not efficiently doable. State-of-the-Art methods of indexing images index features that group a set of images into a similar prototype, e.g. having the same color histogram. Nevertheless, as in the case of Biology, such methods are not capable of finding images that are similar to the input image by some other feature.

Audio Indexing

Indexing large audio archives has emerged recently [15] as an important research topic as large audio archives now exist. There are several possible goals to audio indexing – speaker indexing and speech indexing. Both are important for a variety of commercial and security applications. As in the applications above, the methods of use are preprocessing the corpus by selecting a set of features that roughly describe similar inputs, and then hierarchically seeking the closest match to an input segment. The method suffers from the same weakness as the computer vision and biological structure indexing.

The above applications all point to the need of breaking up data to small particles and using them as identifiers of the data.

2 Intuition of Main Idea

We will describe below in a general "hand-wavy" manner the intuition of an idea that has been applied in the various domains we had discussed. We will then give the combinatorial abstraction of the idea and outline our results.

The main idea we would like to explore is the following. Since we do not want to commit ourselves a-priori to any particular shape, model, or relation, we slice the object into a very large amount of small pieces. Intuitively, when we get the jumbled pieces of two objects, we consider only those pieces that occur in both piles. If there are many of these, we have a potential similarity. However, even a large number of pieces in the intersection may not mean that their sources are similar. As an extreme example consider two black and white matrices A and B. A is a checkerboard and B has a black top and a white bottom. If we slice the matrices into squares the size of a square in the checkerboard we will end up with exactly the same small pieces in both jumbled matrices, but they are clearly very different.

Thus, we go a step further. Every model has a set of rules whereby two pieces can be judged as adjacent. If we can piece together large sub-objects from the similar pieces of both objects, we expect the given objects to be, indeed, similar. For example, an image of a car in the desert and a car in the forest would have the car in common.

The above idea has been used successfully in computer vision. Ullman and his groups used such "image fragments" for object classification(e.g. [20,4]), since such fragmentation gives implicit spatial information. The computer vision world has indeed embraced the *patches* model. The idea has also been employed in the graphics community (see e.g. [22]).

The more primitive idea of comparing just the number of "small pieces" is quite old. Color histogram has been used for decades as a crude index for content-based image retrieval systems [23]. It has been used in Natural Language Processing as well (e.g. [12]).

The human genome projects [6,5] and other genomes discoveries are based on a similar idea. Namely, the input are short sequences taken from copies of the DNA of the same cell, and the goal is to assemble one copy of this DNA sequence.

In this paper, we seek to combinatorially define the idea of *patches*, and rigorously analyse its possibilities. This paper is devoted to the model definitions and initial technical results. When modeling real life applications, one needs to "clean up" and abstract many phenomena. For example, the shape, size, and dimension of the fragments, as well as the criteria for attaching adjacent pieces are application dependent.

For simplicity we assume an "exact match" in joining pieces. The major function necessary for a patch metric, is constructing the full object from its fragments. This task is interesting in its own right and has drawn attention in the pattern recognition community [25].

3 Combinatorial Definition

Our first task developing a combinatorial definition for the "real world" problem of pattern matching in a patch model. The definition below abstracts the problem to the combinatorial realm.

Definition 1. *Given matrix M,* $\begin{pmatrix} m_{0,0} \cdots\cdots\cdots m_{0,n} \\ \cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots \\ m_{n,0} \cdots\cdots\cdots m_{n,n} \end{pmatrix}$

A is a division of M to patches *if* $A = \{a_{0,0}, \cdots a_{0,n-1}, \cdots\cdots a_{n-1,0}, \cdots a_{n-1,n-1}\}$ *and* $\forall i,j\ a_{i,j} = \begin{bmatrix} m_{i,j}, m_{i,j+1} \\ m_{i+1,j}, m_{i+1,j+1} \end{bmatrix}$.

The problem we are concerned with is the inverse.

Definition 2. *The problem of* constructing an image from patches *is defined as follows:*

INPUT: $A = \{a_0, \cdots, a_{n^2-1}\}$ *be a set of* 2×2 *matrices over alphabet* Σ.

OUTPUT: Construct an $(n+1) \times (n+1)$ *matrix* $M = \begin{pmatrix} m_{0,0} \cdots\cdots\cdots m_{0,n} \\ \cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots \\ m_{n,0} \cdots\cdots\cdots m_{n,n} \end{pmatrix}$ *such that A is the division of M to patches, if such a matrix exists. Otherwise report that no matrix can be constructed from the input.*

Example. $A = \{ \begin{bmatrix} a,b \\ a,a \end{bmatrix}, \begin{bmatrix} a,a \\ b,a \end{bmatrix}, \begin{bmatrix} a,a \\ a,a \end{bmatrix}, \begin{bmatrix} b,b \\ a,b \end{bmatrix}, \begin{bmatrix} b,b \\ a,a \end{bmatrix}, \begin{bmatrix} a,b \\ a,a \end{bmatrix}, \begin{bmatrix} a,a \\ b,b \end{bmatrix}, \begin{bmatrix} b,a \\ b,a \end{bmatrix}, \begin{bmatrix} b,a \\ a,b \end{bmatrix} \}$

The patches in set A can be constructed into text M, $M = \begin{pmatrix} a & b & b & a \\ a & a & a & b \\ b & b & a & a \\ a & b & a & a \end{pmatrix}$.

It seems from the definition that we have reduced our problem to a *tiling* problem. Tiling problems are quite old in Combinatorics and Computer Science, starting from geometrical tiling problems (e.g. [16,3]), through Computability issues (e.g. [24]), and ending in the complexity of various tiling problems [9,18]. To our knowledge, the version we are proposing has only been used by Levin [14], and that to prove $\mathcal{NP}$-completeness.

Nevertheless, we believe the version defined in this paper models an extremely important indexing problem and thus its complexity issues ought to be more thoroughly studied. This paper takes some preliminary steps in this direction.

4 One Dimension Tiling

We start by simplifying the problem to one dimension.

Let Σ be an alphabet. Denote by $\alpha = < x, y >$ a pair over alphabet Σ^2, $(x, y \in \Sigma)$.

Let A be a set $A = \{\alpha_1, ..., \alpha_n\}$, $\alpha_j \in \Sigma^2$; $j = 1, ..., n$.

Definition 3. *A is called a* tiling *if its elements can be arranged in order* $\alpha_1, ...,$ α_n *such that* $\forall i, 1 \leq i \leq n-1, y_i = x_{i+1}$.

4.1 One Dimension Algorithm

We provide an algorithm for solving the one dimension tiling problem.

Definition 4. *Let* $\alpha, \beta \in A$. *There is a* connection between α and β *if there exists* $A_1 \subseteq A$, *such that* A_1 *is a tiling, and* $\alpha, \beta \in A_1$.

Let $a \in \Sigma$. *Denote by* $||a||_x$ ($||a||_y$) *the number of times a appears in the left (right) side of a pair in A.*

Let $\alpha \in A$. *Denote by* $x_\alpha \in \Sigma$ ($y_\alpha \in \Sigma$) *the symbol on the left (right) side of pair* α,

A tiling $\alpha_1, ..., \alpha_n$ *is* cyclic *if* $x_{\alpha_1} = y_{\alpha_n}$.

Note that a cyclic tiling can actually start at any pair in the tiling, since the last pair can be connected to the first pair.

Example. $< a, b >, < b, c >, < c, d >, < d, a >$ is can also be tiled as $< b, c >,$ $< c, d >, < d, a >, < a, b >$ and as $< c, d >, < d, a >, < a, b >, < b, c >$.

The following lemma gives necessary and sufficient conditions for tiling.

Lemma 1. *Let* Σ *be the alphabet symbols occurring in A. A is a tiling iff:*

(a.1) *At least* $|\Sigma| - 2$ *characters a, fulfill the condition* $||a||_x = ||a||_y$
(a.2) $\forall a \in \Sigma$, $||a||_y - 1 \leq ||a||_x \leq ||a||_y + 1$
(a.3) *If there is an* $a \in \Sigma$, *such that* $||a||_x = ||a||_y + 1$, *then there exists* $b \in \Sigma$ *such that* $||b||_x + 1 = ||b||_y$.
(b) $\forall \alpha, \beta \in A$ *there is a connection between* α *and* β.

Proof. $\Rightarrow$ simple.

$\Leftarrow$ Our problem is reducible to the directed version of Euler's *Königsberg Bridge problem.* We define the the problem below.

Definition 5. *The* Directed Königsberg Bridge problem *is defined as follows:*

INPUT: A directed multi-graph $G = (V, E)$.
OUTPUT: Find a path that traverses all edges of the graph, visiting every edge exactly once (it is permissible to visit a vertex multiple times).

Euler [7] showed a necessary condition for solving the undirected version of the problem, and Hierholzer [11] proved that the undirected version of the problem can be solved if and only if the graph is connected, and there are exactly two or zero nodes of odd degree.

The reduction is as follows: Construct a directed multi-graph $G = (V, E)$, whose vertices are the alphabet letters of Σ. For pair $\alpha =< x, y >$ there is a directed edge from vertex x to vertex y.

A proof that our conditions, indeed, cause the graph to be connected, and that considers the directed version will be provided in the final version.

4.2 Pattern Matching

The *tiled pattern matching problem* is defined as follows.

Definition 6. Tiled pattern matching of P in A is:

INPUT: $A = \{\alpha_1, ..., \alpha_n\}$, $P = \{\tau_1, ..., \tau_m\}$, $\alpha_i, \tau_j \in \Sigma^2$, $i = 1, ..., n$; $j = 1, ..., m$
OUTPUT: Decide if $\exists$ *tiling of* A *and tiling of* P *such that* P *tiling is a substring of* A *tiling.*

Since P must be a substring of A, then $P \subseteq A$. In the first stage check if P itself is a tiling.

If P is cyclic, we can check A with all possible beginnings of P. However, another strategy is better. It suffices to check if $A \setminus P$ is a tiling and if there is a common alphabet symbol to P and $A \setminus P$. If P is not cyclic, let τ_1, τ_k be the first and last pairs of P, respectively. Let $\alpha_1, ..., \alpha_j$ be a tiling of $A \setminus P$. There is a tiled matching of P in A if either $x_{\alpha_1} = y_{\tau_k}$, $y_{\alpha_j} = x_{\tau_1}$, or if there exist two pairs $\alpha_i, \alpha_{i+1} \in A \setminus P$, where $y_{\alpha_i} = x_{\tau_1}$, and $x_{\alpha_{i+1}} = y_{\tau_k}$.

5 Two Dimensional Tiling

In this section we show that two dimensional tiling is hard and find an approximation algorithm.

5.1 Two Dimensional Tiling is $\mathcal{NP}$-Hard

Theorem 1. *The problem of* constructing an image from patches *as defined in Definition 2, is hard.*

Proof. By reduction from Levin's tiling problem [14].

Definition 7. *A patch A may be* placed *to the right (left, top, bottom) of patch B if the pair of letters on the right (left, top, bottom) side of B are the same as the pair on the left (right, bottom, top) of patch A.*

A tiled square descriptor *f is a function that, given a square of patches placed next to each other, outputs its first row and the list of patches used.*

Levin's tiling problem *is inverting the tiled square descriptor, i.e.*

INPUT: A row R of n patches placed next to each other, and a multiset S of $(n-1)n$ *patches.*
DECIDE: Whether there exists a square of patches placed next to each other, whose first row is R and where the rest of the tiles used are exactly those in the multiset S. We call such a square a Levin Square.

Levin showed [14] that Levin's tiling problem is $\mathcal{NP}$-complete.

We polynomially reduce Levin's tiling problem to our two-dimensional image construction problem as follows.

Given the input for Levin's tiling problem, and let the first row R be the n patches $\{x_0, \cdots, x_n\}$.

We have two tasks ahead of us. The first is to show that the problem of placing patches *next to each other* in Levin's tiling problem is equivalent to that of constructing an image from *overlapping* patches. The second challenge is to force the patches in row R to be the first row in the image constructed from our patches.

Definition 8. *The* Square Tiling Problem *is defined as follows:*

INPUT: A multiset S of n^2 patches.
DECIDE: Whether there exists a square of patches placed next to each other, whose patches are exactly those in the multiset S.

The equivalence of Square Tiling and our Image Construction problem is established by the following lemma.

Lemma 2. *n^2 patches can be placed in an $n \times n$ Levin Square iff those n^2 patches can be used to construct an $(n+1) \times (n+1)$ image. (Note that the size of the Levin Square is given in patches and the size of the image is given in pixels.)*

Proof. An $n \times n$ Levin square constructs a $2n \times 2n$ matrix of symbols. Erasing all even columns and rows, excluding the last even column and row, produces an $(n+1) \times (n+1)$ image constructed from the given patches.

Conversely, given an $(n+1) \times (n+1)$ image, doubling all columns and rows excluding the first and last, produces a Levin Square constructed from the given patches. □

We now reduce Levin's tiling problem to the Square Tiling problem. Assume the first row is $\{x_0, \cdots, x_n\}$. Replace this set of patches by n new patches $\{x'_0, \cdots, x'_n\}$ where the symbols on the top of $x_0, \cdots, x_n$ are changed to new symbols which do not exist in the alphabet, but that cause $x'_0, \cdots, x'_n$ to be placed next to each other in that order. Specifically, let $\{a_1, ..., a_{n+1}\}$ be new symbols not in Σ, replace the top symbols of patch x_i by a_i, a_{i+1}.

Example: If the first row is:

$$\{\begin{bmatrix}1,2\\3,4\end{bmatrix}, \begin{bmatrix}2,5\\4,6\end{bmatrix}, \begin{bmatrix}5,3\\6,3\end{bmatrix}, \begin{bmatrix}3,7\\3,0\end{bmatrix}\}$$

then change it to:

$$\{\begin{bmatrix}a_1,a_2\\3,4\end{bmatrix}, \begin{bmatrix}a_2,a_3\\4,6\end{bmatrix}, \begin{bmatrix}a_3,a_4\\6,3\end{bmatrix}, \begin{bmatrix}a_4,a_5\\3,0\end{bmatrix}\}$$

where $a_1, a_2, a_3, a_4, a_5 \notin \Sigma$.

Now, every construction of a Levin Square is forced to put on top in the given order, the patches $x'_1, ..., x'_n$, and is thus a solution to Levin's tiling problem. No solution to our problem means no solution to Levin problem. □

5.2 Matching Equal Symbols

Lemma 2 allows us to consider the problem of matching patches when their borders match, without recourse to an overlap. We have seen that the image reconstruction problem is $\mathcal{NP}$-hard. It is therefore natural to ask whether relaxing the condition that allows placing two patches next to each other yields a more tractable problem.

Definition 9. *Let $\Sigma = \{1, 2, \cdots, |\Sigma|\}$. A patch A may be* closely placed *to the right (left, top, bottom) of patch B if the pair of symbols $< b_1, b_2 >$ on the right (left, top, bottom) side of B satisfy $|a_1 - b_1| \leq k$ and $|a_2 - b_2| \leq k$, where $< a_1.a_2 >$ is the pair on the left (right, bottom, top) of patch A, and k is a given fixed constant.*

The Near Square Tiling problem *is defined as follows:*

INPUT: A multiset S of n^2 patches.
DECIDE: Whether there exists a square of patches closely placed *next to each other, whose patches are exactly those in the multiset S.*

Theorem 2. *The Near Square Tiling Problem is $\mathcal{NP}$-hard.*

Proof. Given input n patches over alphabet Σ, multiply every symbol in the alphabet by k getting a new alphabet $\Sigma' = \{1, ...|\Sigma|k\}$. Now the only solution to the Near Square Tiling problem is the solution to the Square Tiling problem. □

5.3 An Approximation Algorithm

Having established the hardness of the image reconstruction problem, we now seek ways to approximate the image. We need to decide first what it is we are trying to approximate. Granted that we can not efficiently reconstruct the image, we can try to develop an efficient algorithm that reconstructs the largest square it can from the patches. Another possibility, and this is the one we took, is reconstructing an image from *all* the patches, but allowing *errors*, where two patches that are placed next to each other but whose symbols don't match, introduce an error. The square tiling of n patches actually has $2n-2\sqrt{n}$ matches. The algorithm below constructs a square with at most n matches, thus we have an algorithm that approximates the matches within a factor that converges to 2 as n grows to infinity.

We would like to construct $\sqrt{n}$ matching rows. If that were done, we would have $n - \sqrt{n}$ matches. Unfortunately, we will not be able to guarantee matching rows. Rather, there will be an additional $\sqrt{n}$ errors within the rows.

For simplicity of exposition, we consider a pair of symbols $< x, y >$, $x, y \in \Sigma$, as a single new symbol in a new alphabet $\Pi = \Sigma \times \Sigma$. Each column of two symbols in a two-dimensional patch becomes a new symbol, converting every two-dimensional patch into a one-dimensional patch.

Example. $\begin{bmatrix} a, b \\ c, d \end{bmatrix}$ will be written as $\begin{bmatrix} ac & bd \end{bmatrix}$.

Notation. Call symbols which occur the same number of times on the right and left side of a tile, *circular symbols*. Symbols for which $||a||_x \neq ||a||_y$, we call *uncircular*.

Algorithm's Idea

The algorithm has four stages:

1. **Finding Uncircular Symbols Phase:** Find the set U of uncircular symbols in Π. Because of Lemma 1 we know that for such symbols $||a||_x = ||a||_y + i_a$, where $\Sigma_{a \in U} i_a \leq \sqrt{n}$. Furthermore, we also know that $\sum i_a = s \leq \sqrt{n}$.
2. **Uncircular Rows Construction Phase:** For each symbol $a \in U$ for which $||a||_x = ||a||_y + i_a$, start constructing i_a row in a greedy fashion as in Lemma 1 until it is impossible to continue. We now have x rows, $x \leq \sqrt{n}$, whose first (and possibly last) element are not circular. If no patches remain we go to the Row Length Adjusting Phase. If there are remaining patches, they are all circular. We go to the Circular Rows Construction Phase.
3. **Circular Rows Construction Phase:** As in Lemma 1 construct, in a greedy fashion, rows starting in circular symbols. Because of the reasons proven in Lemma 1, all these rows are *circular*, i.e., begin and end in the same symbol. At this point, insert all circular rows, wherever possible, into other rows, until no more such insertions are possible. We prove below that the total number of remaining rows, both circular and uncircular, are no greater than $\sqrt{n}$. The problem is that some of them may have length greater than $\sqrt{n}$ and some may have a shorter length.
4. **Row Length Adjusting Phase:** For each row whose length exceeds $\sqrt{n}$, cut it to as many rows of length $\sqrt{n}$ as possible. All these rows are now complete. The remaining subrow of length less than $\sqrt{n}$, is added to one of the incomplete rows whose length is less than $\sqrt{n}$ (possibly introducing a mismatch).

   Iterate on this phase as long as there remain incomplete rows. Note that there are no more than $\sqrt{n}$ iterations, and each one introduces at most one mismatch.

   At the end of this phase we have exactly $\sqrt{n}$ rows of length exactly $\sqrt{n}$ each, and at most $\sqrt{n}$ mismatches within these rows. Placing these rows one on top of the other in any sequence gives a $\sqrt{n} \times \sqrt{n}$ matrix with at most $\sqrt{n}$ mismatches within the rows and $(\sqrt{n} - 1)\sqrt{n}$ mismatches between the rows for a total of $\sqrt{n}^2 = n$ mismatches.

Time. Phase 1 can be done in time $O(n \log n)$ by sorting. Phase 2 can be done in linear time. Phase 3's most complex part is a *union-find* which can be implemented in time $O(n\alpha(n))$, where $\alpha(x)$ is the inverse Ackerman function. Phase 4 can be implemented in linear time. Thus the total algorithm time is $O(n \log n)$.

Correctness

The correctness hinges on Lemma 1, and on the fact that the patches were produced from a $\sqrt{n} \times \sqrt{n}$ image. This guarantees that there is equal number of circular symbols on the left side and the right side of the patches, and thus an equal number of uncircular symbols on the left and right sides of the patches. Therefore, all rows constructed during Phase 2, start and end with uncircular symbols, and there are no more than $\sqrt{n}$ such rows.

Lemma 3 proves that the total number of rows after Phase 3 is no larger than $\sqrt{n}$. Now Phase 4 is clear.

Lemma 3. *Given k one dimensional rows, the greedy algorithm reconstructs at most k rows.*

Proof. Will be provided in the final version.

5.4 Largest Common Image

So far we have discussed the problem of constructing an image from patches. Patches can be used as a method for indexing images. In this context it is necessary to find the largest sub-image common to two given images.

Definition 10. *Let A and B be two sets of patches, each of size n. The* largest common image *of A and B is the largest set of patches in the intersection of A and B that can be placed in a square.*

Theorem 3. *Computing the largest common image is $\mathcal{NP}$-hard.*

Proof. By reduction from the Square Tiling problem. Let S be the set of n patches that are input to the Square Tiling problem, Take $A = B = S$. The Largest Common Image of A and B is A iff there is a Square Tiling of S. □

5.5 "One and a Half" Dimensions

We have seen that a one dimensional image, where each patch connects to two neighbors, one on its right and one on its left, can be efficiently reconstructed. We have seen that a two dimensional image, where a patch connects to four neighbors - top, bottom, left and right - is $\mathcal{NP}$-hard. What happens in "one and a half" dimensions, i.e., where each patch connects to *three* neighbors?

The case of a patch connecting to three neighbors is possible if the patches are equilateral triangles, rather than squares. Equilateral triangles can still tile the plane (see Figure 1) but there are only three neighbors to every patch. Does the reduction of a degree of freedom make the tiling efficiently computable, or is it still $\mathcal{NP}$-hard?

It turns out that there is still one other equilateral polygon that can tile the plane - the hexagon. Since the hexagon has 6 neighbors, we expect that reconstructing an image from hexagons is hard. We will, indeed, see that this is the case. However, the hexagon tiling helped us solve the triangular tiling case.

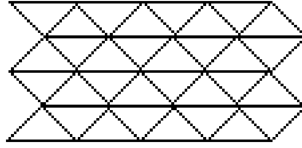


Fig. 1. triangles tiling the plane

Theorem 4. *The Hexagonal Tiling Problem and the Triangle Tiling Problem are $\mathcal{NP}$-hard.*

Proof. Will be provided in the final version.

6 Conclusion and Open Problems

We gave a combinatorial model for a common ad-hoc technique used by many application domains for indexing. This is a specific version of square tiling. We also took some tentative initial steps in studying this model.

We presented an efficient algorithm for the one dimension tiling problem, and proved the two dimension problem is hard. We then developed an approximation algorithm with an approximation ratio converging to 2. We have barely begun to scratch the surface of these problems, and much research with potential applications is still left to be done.

Some interesting examples of open problems that need to be tackled in order to understand the applicability of this model to indexing are approximating the largest contiguous sub-image of a given set of patches, and studying the tiling problem of non-square tiles for special forms.

References

1. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via parikh mapping. J. of Discrete Algorithms 1(5-6), 409–421 (2003)
2. Cohen, F.E.: Folding the sheets: Using computational methods to predict the structure of proteins. In: Lander, E.S., Waterman, M.S. (eds.) Calculating the Secrets of Life: Contributions of the Mathematical Sciences to Molecular Biology, pp. 236–271. National Academy Press (1995)
3. de Bruijn, N.G.: Algebraic theory of penrose's nonperiodic tiling of the plane. Indagationes mathematicae (1981)
4. Epshtein, B., Ullman, S.: Identifying semantically equivalent object fragments. In: Proc. IEEE Conference on Computer vision and Pattern Recognition (CVPR), vol. 1, pp. 2–9 (2005)
5. Collins, et al.: nternational human genome sequencing consortium. initial sequencing and analysis of the human genome. Nature 409(6822), 860–921 (2001)
6. Vender, J.C., et al.: The sequence of the human genome. Science 291(5507), 1304–1351 (2001)

7. Euler, L.: Solutio problematis ad geometriam situs pertinentis. Commentarii Academiae Scientiarum Petropolitanae 8, 128–140 (1741)
8. Frakes, W.B., Baeza-Yates, R.: Information Retrieval Data Structure and Algorithms. Prentice-Hall, Englewood Cliffs (1992)
9. 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)
10. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
11. Hierholzer, C.: Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechnung zu umfahren. Mathematische Annalen 6, 30–32 (1873)
12. Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A.: Constraint Grammar. A Language Independent System for Parsing Unrestricted Text. Walter de Gruyter, Berlin (1995)
13. Kolodny, R., Koehl, P., Guibas, L., Levitt, M.: Small libraries of protein fragments model native protein structures accurately. Journal of Molecular Biology 323(2), 297–307 (2002)
14. Levin, L.A.: Universal sorting problems. Problemy Peredachi Informatsii 9(3), 265–266 (1973) (in Russian)
15. Lu, G.: Indexing and retrieval of audio: A survey. Multimedia Tools and Applications 15(3), 269–290 (2001)
16. Makovicky, E.: 800-year-old pentagonal tiling from maragha, iran, and the new varieties of aperiodic tiling it inspired. In: Hargittai, I. (ed.) Fivefold Symmetry. World Scientific, Singapore (1992)
17. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008)
18. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
19. Parikh, R.J.: On context-free languages. Journal of the ACM 14(4), 570–581 (1966)
20. Vidal-Naquet, M., Ullman, S., Sali, E.: A fragment-based approach to object representation and classification. In: Arcelli, C., Cordella, L.P., Sanniti di Baja, G. (eds.) IWVF 2001. LNCS, vol. 2059, pp. 85–102. Springer, Heidelberg (2001)
21. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. Computer Series. McGraw-Hill, New York (1983)
22. Shilane, P., Funkhouser, T.A.: Distinctive regions of 3d surfaces. ACM Transactions on Graphics 26(2) (2007)
23. Stricker, M., Swain, M.: The capacity of color histogram indexing. In: Proc. IEEE Conference on Computer vision and Pattern Recognition (CVPR), pp. 704–708 (1994)
24. Wang, H.: Proving theorems by pattern recognition. Bell systems Technical journal, 1–42 (1961)
25. Zhu, L., Zhou, Z., Hu, D.: Globally consistent reconstruction of ripped-up documents. IEEE Transactions on Pattern Analysis and Machine Intelligence 30(1), 1–13 (2008)

The Frequent Items Problem, under Polynomial Decay, in the Streaming Model

Guy Feigenblat, Ofra Itzhaki, and Ely Porat

Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
{feigeng,yzhakio,porately}@cs.biu.ac.il

Abstract. We consider the problem of estimating the frequency count of data stream elements under polynomial decay functions. In these settings every element arrives in the stream is assigned with a time decreasing weight, using a non increasing polynomial function. Decay functions are used in applications where older data is less significant \ interesting \ reliable than recent data. We propose 3 poly-logarithmic algorithms for the problem. The first one, deterministic, uses $O(\frac{1}{\epsilon^2}\log N(\log\log N+\log U))$ bits. The second one, probabilistic, uses $O(\frac{1}{\epsilon^2}\log\frac{1}{\epsilon\delta}\log N)$ bits and the third one, deterministic in the stochastic model, uses $O(\frac{1}{\epsilon^2}\log N)$ bits. In addition we show that using additional additive error can improve, in some cases, the space bounds. This variant of the problem is important and has many applications. To our knowledge it was never studied before.

1 Introduction

Processing large data is a significant subject in nowadays research. In the internet epoch, we have mass amount of data to process, in the size of Terabytes or even Petabytes. Many applications, such as IP communication management, audio and video streaming, sensor reading and stock exchange tracking are characterized by data arriving in mass amounts, sequentially and rapidly. Most of the frameworks for these applications don't have enough workspace to process the data and store it entirely. Thus, most of the algorithms typically store a synopsis of the data using much less space than the amount of the arriving data. Using the synopsis, the algorithm should be able to answer queries regarding the data and clearly, there is a tradeoff between the size of the synopsis and the precision of the returned answers. As an outcome, there is a constant need to develop new algorithms, more accurate and efficient, for online management and monitoring of these applications. These settings introduced, a decade ago, a model of computation called data stream (Streaming).

In the streaming model each arriving data item has the same relative contribution, i.e. weights the same. However, in many applications older data is less significant \ interesting \ reliable than recent data. Therefore, we would like to weight newer data more heavily than older data. This can be done using decay functions, which are non increasing functions, that assign weight to each arriving item. Each weight is constantly updated as a function of the time elapsed since

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 266–276, 2009.
© Springer-Verlag Berlin Heidelberg 2009

the item was first observed in the stream. There are few types of commonly used decay functions, such as Sliding window, Polynomial and Exponential decay. When we consider Sliding window decay, each data item in the window is assigned with one unit weight, while all the others, outside the window, are discounted. Other types of decay functions assign each item with different weight, diminishes as a function of the time elapsed.

For many applications needs, both sliding window and exponential decay aren't sufficient. The advantage of polynomial decay is that the weights decrease in a smoother way. Namely, the ratio between two different elements is constant, as will be elaborated in the sequel.

In this work we propose 3 sketch based algorithms for estimating data stream elements decayed frequencies under polynomial decay. The first one, is deterministic, the second one is probabilistic and the third one, is deterministic in the stochastic model. Our work serves both theoretical and practical aspects. Specifically, we address a variant of the frequency count problem that to our knowledge was never studied before, in an efficient and simple way. Moreover, multiple sketches of the deterministic algorithm can be joined together easily, which is an advantage for the distributed streaming scenario. In addition, this variant can be suitable for applications where other types of decay are too rigid.

One application that can utilize this work is caching system for web servers. There are many replacements policies for caching, where the most commonly used are the LRU (Least Recently Used) policies. However, it isn't the case for caching web servers [1,2]. Results of researches show that using LFU (Least Frequently Used) replacement policies with aging mechanism (i.e. frequency count with decay) perform much better [1]. There are more frequency based caching algorithms like in [3].

1.1 Decay Functions

We represent the definitions given by Cohen and Strauss in [4]. Consider a stream where $f(t) \geq 0$ is the item value of the stream obtained at time t. For simplicity we assume our stream only receives values at discrete times, and therefore, t is integral. We define a decay function $g(x) \geq 0$ for $x \geq 0$ to be a non-increasing function. At time T the weight of an item that arrived at time $t \leq T$ is $g(T-t)$ and the decayed value of that item is $f(t)g(T-t)$. The decayed sum (DSP) under the decay function $g(x)$ is defined as: $V_g(T) = \sum_{t \leq T} f(t)g(T-t)$. When $f(t)$ receives only binary values, we refer to $V_g(T)$ as decay count (DCP), since it aggregates under decay the number of positive bits.

As mentioned above, there are 3 types of commonly used decay functions: Sliding Window, Exponential and Polynomial Decay. Formally, the Sliding Window decay for a window size W, assigns weights $\forall\ 0 \leq x \leq W\ \ g(x) = 1$ and $g(x) = 0$ otherwise. Exponential Decay (ExpD) for a given parameter $\lambda > 0$, assigns weights $g(x) = \exp(-\lambda x)$. Polynomial Decay (PolyD) for a given parameter $\alpha > 0$, assigns weights $g(x) = \frac{1}{x^\alpha}$.

1.2 Model and Problem Definition

Consider a data stream of N elements drawn from a universe U. Each is assigned at arrival with a time decreasing weight. Elements constantly arrive and due to the size of the stream, it is only allowed to perform one pass over the data. Furthermore, the storage available is poly-logarithmic in N and the data should be processed in minimum time.

Let $\widetilde{N}$ be the sum of weights assigned to the elements, such that $\widetilde{N} = \sum_{i=1}^{N} g(i)$, and $\epsilon \in (0,1)$ be the error factor. Our goal is to approximate, up to error of $\epsilon\widetilde{N}$, the decayed frequency of each element observed in the stream, i.e. we would like to approximate the decayed count (DCP) of each element with error less then $\epsilon\widetilde{N}$.

1.3 Related and Previous Work

As was mentioned above, to our knowledge, this variant of the problem was not studied before.

Cohen and Strauss in [4] introduced the time decay sum and time decay average under general decay function. In addition, they developed a data structure (sketch) - Weight Based Merging Histograms (WBMH) - that guarantees $(1 \pm \epsilon)$ multiplicative approximation for the sum of values of the elements observed in the stream, under polynomial decay. This structure uses at most $O(\frac{1}{\epsilon} \log N \log\log N)$ bits of space, where $\epsilon \in (0,1)$ and N denotes the length of the stream. They also showed a lower bound of $\Omega(\log N)$ bits for this problem. Kopelowitz and Porat in [5] proposed an improved algorithm - Altered Exponential Histograms (AEH) - matching the lower bound from [4]. Their algorithm combines WBMH [4] and Exponential Histograms (EH) [6]. They also proposed another model where additive error is allowed. For more details see the sequel.

Cormode et al [7] proposed a time decay sketch technique for summarizing decayed streaming data. The sketch can give estimates for various decayed aggregates. It is mainly targeted to the distributed streaming scenarios, such as sensor networks, since it is duplicate insensitive - meaning that re-insertion of the same data will not affect the estimates of the aggregates. Furthermore, multiple sketches computed over distributed data can be combined without losing accuracy.

In another paper, Cormode et al [8] proposed deterministic algorithm for approximating several decayed aggregates in out of order streams. Under these settings, elements do not necessarily arrive in the order of their appearance in the stream. The algorithm works with Sliding window, Polynomial and Exponential decay.

Not a lot of work was done on estimating decayed stream aggregates, however, many algorithms were proposed for approximating stream aggregates in general and frequency count in particular. For a more detailed description of the work that was done see [9].

The first deterministic algorithm, which provides additive approximation for the frequencies of data streams elements, is the Misra-Gries Algorithm [10]. This algorithm uses m counters, where the size of m depends on the approximation accuracy, and provides $O(1)$ amortized processing time per element. The same algorithm was rediscovered by Demaine et al [11] and Karp et al [12], who reduced the processing time to $O(1)$ in the worst case. In order to get approximation with maximum error ϵN where N denotes the length of the stream, we set $m = \lceil \frac{1}{\epsilon} \rceil$. Demaine et al [11] also developed algorithm for the stochastic model.

Manku and Motwaniet [13] proposed 2 algorithms for computing frequencies of elements. The first one, *StickySampling*, is probabilistic algorithm which identifies all items that their true frequency exceeds $(s - \epsilon)N$ with probability $1 - \delta$, where N denotes the length of the stream, $s \in (0, 1)$ is a user specified threshold and $\epsilon \in (0, 1)$ is the maximum error. The expected number of counters is at most $O(\frac{1}{\epsilon} \log(s^{-1}\delta^{-1}))$. The second algorithm, *LossyCounting* is a deterministic algorithm. It guarantees the same precision using at most $O(\frac{1}{\epsilon} \log(\epsilon N))$ counters, regardless of s.

Arasu and Manku [14] proposed 2 algorithms for calculating frequency count over Sliding Window decay. The first one, deterministic, uses $O(\frac{1}{\epsilon} \log^2(\frac{1}{\epsilon}))$ counters and the second one, probabilistic, provides approximation with probability at least $(1 - \delta)$ by using $O(\frac{1}{\epsilon} \log(\epsilon\delta)^{-1})$ counters.

Cormode and Muthukrishnan in [15] introduced the Count-Min sketch. They developed a poly-logarithmic data structure for summarizing data streams, utilizing pair-wise independent hash functions. Each arriving element is mapped from the universe U to entries in the sketch. At query time, the result of the query is the minimum value of the entries mapped to the element. The size of the sketch is $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ and the error guarantee per query is ϵL_1 with probability $1 - \delta$, where L_1 denotes the sum of frequencies of the elements observed. For further details see the sequel.

2 Deterministic Algorithm

2.1 Weight Based Merging Histograms

The Weight Based Merging Histograms were introduced by Cohen and Strauss in [4]. These histograms provide $\epsilon-$ multiplicative approximation for Poly Decay count using $O(\frac{1}{\epsilon} \log N \log\log N)$ bits of space (where N denotes the stream's length).

Weight Based Histograms (WBMH) as other types of histograms, such as Exponential histograms [6], aggregate values into buckets. The main difference is that in WBMH the boundaries of the buckets are dependent on the decay function, and not on a particular stream instance. This means that the buckets' timestamps don't need to be stored in the buckets explicitly.

WBMH utilize the fact that if a decay function has the property that $g(x)/g(x + \Delta)$ is non-increasing with x for any time frame Δ, then the ratio of two items remains fixed, or approaches one as time advances. This means that as time progresses, elements in larger vicinities have the same decayed weight

up to a multiplicative factor. WBMH utilize this property by grouping together in the same bucket, values with similar weights. Buckets boundaries are determined in the following way. Let $b_0 = 1$ and b_1 be the maximum value such that $(1+\epsilon)g(b_1 - 1) \geq g(b_0)$. In the same manner, let b_i be the maximum value such that $(1+\epsilon)g(b_i - 1) \geq g(b_{i-1})$. Furthermore, the number of elements in each bucket depends on the stream, but the boundaries don't. The range $[b_i, b_{i+1} - 1]$ is referred as a region.

In order to approximate the decay count (DCP) of the stream, WBMH work in the following way. When a new element arrives it is added to the "current" (first) bucket. At any time T where $T \equiv 0(mod\ b_1)$ the "current" bucket is sealed, and a new one is opened. Whenever there exist an i and two buckets, such that the two buckets are within a region $[b_i, b_{i+1} - 1]$, they are merged into one. The DCP is the sum of the buckets multiplied by their region boundary.

Notice that the decay weight of each element in the same bucket is within $1 \pm \epsilon$ factor. Thus, the decayed count can be computed by multiplying the number of non-zero values in the bucket, with the bucket's corresponding region boundary. Since keeping an exact counter for the number of non-zero values in the bucket is costly, an estimated counter in the size of $O(\log \log N)$ bits is used. The approximated decayed count of the stream can be computed by summing the approximated decayed count of every bucket. It was showed in [4] that under Poly Decay there are $O(\frac{1}{\epsilon} \log N)$ different b_is for every function. Since 2 buckets within the same region are merged, the total number of buckets is $O(\frac{1}{\epsilon} \log N)$.

2.2 Algorithm

In this section, we describe a deterministic process that approximates the decayed frequency counts of data stream elements, under Poly Decay, with maximum error of $\epsilon \widetilde{N}$ per element. The space used in this process is $O(\frac{1}{\epsilon^2} \log N(\log \log N + \log U))$ storage bits, where N denotes the length of the stream.

For this process we will exploit the property that polynomial decay divides the stream to $O(\frac{1}{\epsilon} \log N)$ boundaries and the way buckets are constructed in [4] as was explained above. Since elements in a bucket have almost the same weight, the approximated decayed frequency of a single element in a bucket, is the number of times it appears multiply by the bucket's corresponding region boundary. Thus, the approximated decayed frequency of an element over the entire stream, is the sum of the approximated decayed frequencies over all the buckets. Formally, denote F_e as the DCP of element e, $\widetilde{f_e^i}$ as the approximated number of appearances of element e in bucket i and w_i as the weight corresponding to the region of bucket i. The approximated decayed frequency of element e over the stream is $\sum_{i=1}^{O(\frac{1}{\epsilon} \log N)} \widetilde{f_e^i} w_i$. Notice that we suffer from two approximation factors.

Our algorithm works as follows. We approximate in each bucket, the frequencies of the elements observed by it. In order to do so we utilize the Misra-Garies algorithm [10](or others as described in [11,12]) as a black box in each bucket. Notice that since we require approximation of ϵ, there should be $O(\frac{1}{\epsilon})$ counters in each black box. The buckets are constructed the same way as in WBMH.

Whenever a new element e arrives, it is added to the current bucket by sending it to the bucket's corresponding black box. At any time T where $T \equiv 0(mod\ b1)$ the current bucket is sealed and a new one is opened.

Whenever there exist an i and two buckets that are within a region $[b_i, b_{i+1}-1]$ they are merged into one. Merge operation is preformed in the following manner. Suppose we need to merge buckets i and j. Denote by N_i the elements observed by bucket i and by N_j the elements observed by bucket j. In each bucket there are $O(\frac{1}{\epsilon})$ counters corresponding to the $O(\frac{1}{\epsilon})$ elements with the highest frequencies. In the new bucket, created during the merge, we keep the $O(\frac{1}{\epsilon})$ elements, with the highest frequencies from both buckets. We can merge the buckets in a straight forward way (without considering the decay), since both buckets are in the same region and thus the elements' weights are roughly the same. Notice that, the number of elements allegedly observed by the new bucket is $N_i + N_j$ and the maximum errors in the old buckets i and j are ϵN_i and ϵN_j respectively. We get that the maximum error in the new merged bucket is $\epsilon(N_i + N_j)$, therefore, the ϵ approximation guarantee is preserved.

Whenever we are asked to retrieve the elements decayed frequencies we scan the buckets and for each element we sum it's frequencies multiplied by the bucket corresponding boundary.

Theorem 1. *Let $g(.)$ be a polynomial decay function such that $g(x)/g(x+1)$ is non increasing with x. The data structure uses $O(\frac{1}{\epsilon^2} log^2 N)$ storage bits and provides approximation with maximum error of $\epsilon\widetilde{N}$, for the elements decayed frequencies.*

Proof. In each bucket we use an instance of the Misra-Garies algorithm as a "black box" with error parameter ϵ'. In addition, we pick our boundaries using the given decay function with error parameter ϵ'', formally $\forall i\ (1+\epsilon'')g(b_i - 1) \geq g(b_{i-1})$. Since in each boundary there can be at most 2 buckets, we get that the total number of buckets is $O(\frac{1}{\epsilon''} \log N)$. Adding the "black box" to each bucket yields a total of $O(\frac{1}{\epsilon'\epsilon''} log^2 N)$ storage bits. Notice that for each element monitored by a counter, we need to maintain it's ID using $O(\log U)$ bits, but usually $\log U \leq \log N$ and thus bounded by $O(\log N)$.

Recall that our algorithm suffers from two approximation factors: the first one is from the way we approximate the frequencies in each bucket and the second is from using boundaries instead of exact decay weights. Combing these approximations together we get $(1 \pm \epsilon'')(F_e \pm \epsilon'\widetilde{N}) = F_e \pm \epsilon'\widetilde{N} \pm \epsilon'' F_e \pm \epsilon''\epsilon'\widetilde{N}$. Notice that $\epsilon'' F_e \leq \epsilon''\widetilde{N}$. Choosing $\epsilon \geq \epsilon' + \epsilon'' + \epsilon'\epsilon''$ by setting $\epsilon' = \epsilon'' = \frac{\epsilon}{3}$ provides the desired approximation guarantee and yields total of $O(\frac{1}{\epsilon^2} \log^2 N)$ storage bits. □

Lemma 1. *Under polynomial decay, $O(1)$ processing operations are required in amortized per element observed in the stream.*

Proof. We use amortization argument for proving the lemma. Let $c = \lceil \frac{1}{\epsilon^2} \log N \rceil$. We hold a buffer of size c and build the sketch in steps. At the first step, when c becomes full we iterate over it and build the sketch structure. The

cost of this operation is c. The number of buckets created in the sketch is $O(\frac{1}{\epsilon}\log(\frac{1}{\epsilon^2}\log N)) = O(\frac{1}{\epsilon}\log\log N)$. We do the same at every round, but in addition we have to append and merge the old sketch at the end of the new sketch. Since the old sketch has $O(\frac{1}{\epsilon}\log\log N)$ buckets each of size $O(\frac{1}{\epsilon})$, the merge operation is bounded by $O(\frac{1}{\epsilon^2}\log\log N)$. Thus, the processing time is bounded by the size of the buffer, namely, $O(\frac{1}{\epsilon^2}\log N)$. Since we do this operation every c time units, we get that the amortized processing time is $O(1)$. □

We can further reduce the space by allowing additional multiplicative approximation. It can be done by using estimated counters, in each black box. It is sufficient to save the exponent, using $O(\log\log N)$ bits and in addition the most significant $O(\log\frac{1}{\epsilon}+\log\log N)$ bits, for each counter. This extra relaxation is possible since the number of merge operations, for each bucket, bounded by $O(\log N)$. This reduces the overall size of the structure to $O(\frac{1}{\epsilon^2}\log N(\log\log N + \log U))$. Due to lack of space, further details are omitted, see [4] for a similar counter method.

Using this structure, multiple histograms can be joined together quiet easily without losing approximation guarantees. Joining two histograms can be done by iterating over the regions and merging 2 buckets at a time (merging buckets as explained above). This property is an advantage when considering distributed streams.

3 Probabilistic Algorithm

In this section, we describe a probabilistic algorithm that approximates with high probability $(1-\delta)$, the decayed frequency count under Poly Decay, with maximum error $\epsilon\widetilde{N}$. The space used by this algorithm is $O(\frac{1}{\epsilon^2}\log\frac{1}{\epsilon\delta}\log N)$ bits, where N denotes the current length of the stream and $\epsilon,\delta \in (0,1)$. Our algorithm utilizes Altered Exponential Histograms [5] and Count-Min sketches [15].

3.1 Count-Min Sketch

As mentioned above, the Count-Min sketch is a poly-logarithmic space data structure for summarizing data streams. The idea of the sketch is to model the stream as a vector X of $|U|$ dimensions, where U denotes the universe. The current state of the vector (stream) at time t is $X(t) = [X_1(t), ..., X_i(t), ..., X_{|U|}(t)]$. Initially, X is the zero vector: $\forall i\ X_i(0) = 0$. When element $i \in U$ arrives in the stream at time t, it is modeled as if the vector's i'th entry is updated (incremented). The value of the i'th entry at time t, i.e. $X_i(t)$, is the frequency of element i. The user specifies 2 parameters (ϵ,δ), where ϵ denotes the error parameter and δ is the probability of failure. Since it isn't possible to maintain the entire vector, a vector sketch is constructed. The sketch is represented as a two-dimensional array of size wd, denotes by $count[\ ,\ ]$, where $w = \lceil\frac{e}{\epsilon}\rceil$ and $d = \lceil\ln\frac{1}{\epsilon\delta}\rceil$. The accuracy estimates for individual query in the sketch depends on the L_1 norm of vector X at any time t.

One of the queries to the sketch is the point query, denotes by $Q(i)$. It returns the approximated frequency of element $i \in U$ at any time t. Formally, for any

time t, $X_i(t)$ is the frequency of element i and let $\widetilde{X_i}(t)$ be the approximated frequency of element i. The query retrieves frequency estimation such that $X_i \leq \widetilde{X_i}$ and with probability at least $1-\delta$, $\widetilde{X_i} \leq X_i + \epsilon||X(t)||_1$.

3.2 Algorithm

Our idea adapts the Count-Min sketch structure and combines Alter Exponential Histograms (AEH) as a black box in each sketch entry. The function of the AEH is to calculate each entry value under the Poly decay function. The sketch uses wd two-dimensional array, initially set to zero. In addition, d hash functions : $h_1...h_d$: $1...|U| \rightarrow 1...w$, are chosen uniformly at random from a pairwise-independent family.

When an element i arrives in the stream at time t, the data structure is updated by adding "1" bit, to the AEH corresponding to each entry mapped to the element. Formally, $\forall\ 1 \leq j \leq d$, $count[j, h_j(i)].\ AEH.\ increment(1)$.

In order to retrieve decayed frequency estimation of element i, the entry with the minimum value of all the entries mapped to the element is returned. Formally, $Q(i) = min_j\ count[j, h_j(i)].\ AEH.\ value()$.

Notice that under these settings X_i denotes the decayed frequency of element i and $\widetilde{X_i}$ denotes the approximated decayed frequency of element i. In addition, recall that we are calculating the frequency by estimating the DCP for each observed element. For any polynomial decay function $g(.)$, we set the accuracy estimates to be dependent on $\widetilde{N}$, since this is the sum of the decayed weights of the elements. Thus, the maximum frequency error expected per element, is set to be $\epsilon\widetilde{N}$ with high probability.

Theorem 1. *Let $g(.)$ be a polynomial decay function such that $g(x)/g(x+1)$ is non increasing with x. The data structure uses $O(\frac{1}{\epsilon^2}\log\frac{1}{\epsilon\delta}\log N)$ storage bits and provides approximation for single element decayed frequency (Point Query), such that $X_i \leq \widetilde{X_i}$; and with probability at least $(1-\epsilon\delta)$, $\widetilde{X_i} \leq X_i + \epsilon\widetilde{N}$. The processing time per element is $O(\log\frac{1}{\epsilon\delta})$.*

Proof. We use a Count-Min sketch with parameters ϵ', δ. We put in each entry of the sketch an instance of AEH with error parameter ϵ''. ϵ', ϵ'' will be determined later. Under these settings, the total space consumption is $O(\frac{1}{\epsilon'\epsilon''}\log\frac{1}{\epsilon'\delta}\log N)$ bits.

First we consider the error in each Point Query, overlooking the error factor from the AEH. The query proof is actually an adaptation of the proof from [15] and therefore we only sketch it. By pairwise independence of the hash functions we get that the probability of collision, for each entry in a row, is less then $1/range(h_j) = \frac{\epsilon'}{e}$. In [15] the authors showed that the expected error in each sketch entry is less then $\frac{\epsilon'}{e}L_1$, which equal $\frac{\epsilon'}{e}\widetilde{N}$ in these settings. In addition, by pairwise independence of h_j and linearity of expectations, it was shown that $Pr[\widetilde{X_i} \geq X_i + \epsilon' L_1] \leq \epsilon'\delta$.

Combing the AEH, we suffer from 2 errors. The first one, multiplicative error of $(1 \pm \epsilon'')$ from the AEH and the second one, additive error of $\epsilon'\widetilde{N}$ from the

Count-Min structure. Setting $\epsilon' = \epsilon'' = \frac{\epsilon}{3}$, provides the desired approximation guarantee and a total space consumption of $O(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon\delta} \log N)$ bits.

As for the processing time, whenever an element arrives we update $O(\log \frac{1}{\epsilon'\delta})$ rows in the two-dimensional array. In each update we increment the histogram which cost $O(1)$ in amortized per update [5]. □

Using the above theorem we can now present an algorithm for decayed frequency count, which follows an idea from [16,15]. For each element, we use the Count-Min data structure to estimate its count, and keep a heap of the top $\lceil \frac{1}{\epsilon} \rceil$ elements seen so far.

Given a data stream, for each element i observed at time t we do the following:

1. Update the entries mapped to i in the Count-Min sketch
2. Retrieve the decayed frequency of element i by running $Q(i)$ query
3. If i is in the heap, increment its count (by adding "1" to it's histogram)
4. Else, if $Q(i)$ is greater then the smallest value in the heap then,
   (a) Generate AEH instance equal the the histogram corresponding to $Q(i)$
   (b) Pop the heap (remove the smallest value)
   (c) Add the newly created AEH instance to the heap

At query time the heap is scanned and all elements in the heap with estimated count above $\epsilon\widetilde{N}$ are output. The probability that an element will not be properly estimated during a point query is less than $\epsilon\delta$. Since an improper estimation of an element is only when the approximation is above the $\epsilon\widetilde{N}$ threshold, an improper estimation in the decayed frequency count algorithm can be caused only by one of the $\frac{1}{\epsilon}$ elements in the heap. Applying union bound shows that the total probability of error in the algorithm is bounded by δ.

In the heap we keep $\lceil \frac{1}{\epsilon} \rceil$ elements. Each element is associated with AEH of size $O(\frac{1}{\epsilon} \log N)$ bits and $O(\log U)$ bits for it's ID. Since usually $\log U \leq \log N$, the total space of the heap is $O(\frac{1}{\epsilon^2} \log N)$ bits. We conclude that the total space consumption of the algorithm is $O(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon\delta} \log N)$ bits.

4 Deterministic Algorithm in the Stochastic Model

In this section we present an algorithm for decayed frequency count in the stochastic model. In this model, an arbitrary probability distribution specifies the relative frequencies of the elements and the order the elements occur in the stream is uniformly random.

Algorithm for frequency count in the stochastic model was proposed by Demaine et al in [11]. The algorithm divides the stream into rounds. At the beginning of each round, the first distinct m elements are sampled, which is equivalent to sampling m elements uniformly at random. At the end of the first round the top $\frac{m}{2}$ elements (with highest frequencies) are saved and the others are discounted. At the next rounds, only $\frac{m}{2}$ counters are used for sampling. At the end of each round, total of $\frac{m}{2}$ elements with the highest frequencies, from the current and the previous rounds are saved. Applying Chernoff bounds show that

the counts obtained during a round are close to the actual frequencies of the elements. For further details and completeness see [11].

We adapt this idea but instead of using m binary counters we use histograms, namely the AEH histograms. As was mentioned above, each histogram is of size $O(\frac{1}{\epsilon}\log N)$ bits. We get that the total size of the structure is $O(m\frac{1}{\epsilon}\log N)$. It is sufficient to set $m = \lceil\frac{1}{\epsilon}\rceil$ for getting good approximation with high probability.

5 Decay with Additional Additive Error

The AEH approximates the DCP by $(1 \pm \epsilon)$ multiplicative approximation. We now consider another model in which in addition to the multiplicative error, an additive error of $\epsilon_2 \in (0, 1)$ is allowed (in the AEH). This kind of relaxation, was discussed in [5]. It was shown that when considering binary streams with polynomial decay, there is a need to differentiate between 2 cases. In the first case where $\alpha > 1$, it is sufficient to maintain only the last $\frac{1}{\epsilon_2}$ elements observed by the stream. It can be done by saving them in AEH and therefore reduce the space to $O(\log(\frac{1}{\epsilon_2}))$ per histogram. In the second case where $0 \leq \alpha \leq 1$, a lower bound was proved showing that it isn't possible to do better.

Since we can calculate the decayed frequencies using the AEH, allowing this extra relaxation can reduce the space consumption. Namely, using this method in the probabilistic algorithm (see section 3), reduces the space consumption to $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon_2}\log\frac{1}{\epsilon\delta}\log N)$ bits. Similarly can be done to the algorithm in the stochastic model.

References

1. Arlitt, M.F., Williamson, C.L.: Trace-driven simulation of document caching strategies for internet web servers. Simulation Journal 68, 23–33 (1996)
2. Cao, P., Irani, S.: Cost-aware www proxy caching algorithms. In: Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, pp. 193–206 (1997)
3. Friedrich, R., Arlitt, M., Arlitt, M., Cherkasova, L., Cherkasova, L., Dilley, J., Friedrich, R., Jin, T.: Evaluating content management techniques for web proxy caches. In: Proceedings of the 2nd Workshop on Internet Server Performance (WISP 1999), Atlanta GA (1999)
4. Cohen, E., Strauss, M.J.: Maintaining time-decaying stream aggregates. J. Algorithms 59(1), 19–36 (2006)
5. Kopelowitz, T., Porat, E.: Improved algorithms for polynomial-time decay and time-decay with additive error. Theor. Comp. Sys. 42(3), 349–365 (2008)
6. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows (extended abstract). In: SODA 2002: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, pp. 635–644 (2002)
7. Cormode, G., Tirthapura, S., Xu, B.: Time-decaying sketches for sensor data aggregation. In: PODC 2007: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pp. 215–224. ACM, New York (2007)

8. Cormode, G., Korn, F., Tirthapura, S.: Time-decaying aggregates in out-of-order streams. In: PODS 2008: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 89–98. ACM, New York (2008)
9. Muthukrishnan, S.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science 1(2) (2005)
10. Misra, J., Gries, D.: Finding repeated elements. Technical report (1982)
11. 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)
12. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. 28(1), 51–55 (2003)
13. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Bressan, S., Chaudhri, A.B., Li Lee, M., Yu, J.X., Lacroix, Z. (eds.) CAiSE 2002 and VLDB 2002. LNCS, vol. 2590, pp. 346–357. Springer, Heidelberg (2003)
14. Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: PODS 2004: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 286–296. ACM, New York (2004)
15. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms 55(1), 58–75 (2005)
16. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 693–703. Springer, Heidelberg (2002)

Improved Approximation Results on the Shortest Common Supersequence Problem

Zvi Gotthilf and Moshe Lewenstein

Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
{gotthiz,moshe}@cs.biu.ac.il

Abstract. The problem of finding the Shortest Common Supersequence (*SCS*) of an arbitrary number of input strings is a well-studied problem. Given a set L of k strings, s_1, s_2, ..., s_k, over an alphabet Σ, we say that their *SCS* is the shortest string that contains each of the input strings as a subsequence. The problem is known to be *NP-hard* [8] even over binary alphabet [12]. In this paper we focus on approximating two *NP-hard* variants of the *SCS* problem. For the first variant, where all input strings are of length 2, we present a $2 - \frac{2}{1+\log n \log\log n}$ approximation algorithm, where $|\Sigma| = n$. This result immediately improves the $2 - \frac{4}{n+1}$ approximation algorithm presented in [17]. Moreover, we present a $\frac{7}{6}$ ($\approx 1.166\bar{6}$) approximation algorithm for the restricted variant (but still *NP-hard*) where all input strings are of length 2 and every character in Σ has at most 3 occurrences in L.

1 Introduction

The problem of finding the *Shortest Common Supersequence* (SCS) of a finite set of strings, L, is a well-studied problem [7,8,10,14,15,16]. An *SCS* of a set L is a shortest string that is a supersequence of every string in L.

The *SCS* problem is known to be NP-hard [8] even in the case of a binary alphabet [12]. However, if the number of input strings is fixed, their *SCS* can be found in polynomial running time [15,16].

The *SCS* problem has several applications in various areas, such as data compression, text editing and computational biology.

Many results regarding the hardness and the approximabilty of the *SCS* problem were presented during the last two decades, see [1,6,7,17]. Also, several heuristics and computational experiments can be found [2,3,4,11].

Throughout this paper we use the following notations. Given a set L of k strings, s_1, s_2, ..., s_k, over an alphabet Σ of size n, we denote by *SCSl*, the problem of finding *Shortest Common Supersequence*, where all input strings are of length l. Moreover, we denote by *SCSl(r)*, the problem of finding *SCS*, where all input strings are of length l and every character in Σ has at most r occurrences in L.

Timkovsky [16] showed that both *SCS2(3)* and *SCS3(2)* are NP-hard problems. On the other hand, *SCS2(2)* can be solved in polynomial running time.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 277–284, 2009.
© Springer-Verlag Berlin Heidelberg 2009

In this paper we focus on approximating the *SCS2* and the *SCS2(3)* problems. We define the approximation version of the *SCS* problem as follows. Let OPT_{scs} be the optimal solution for the *SCS* problem and let APP_{scs} be the result of the approximation algorithm *APP* (such that APP_{scs} is a common supersequence of s_1, s_2, ..., s_k). The approximation ratio of the *APP* algorithm will be the smallest ratio between $|APP_{scs}|$ and $|OPT_{scs}|$ over every possible input set L.

For the *SCS2* problem we present a $2 - \frac{2}{1+\log n \log\log n}$ approximation algorithm. This result immediately improves the $2 - \frac{4}{n+1}$ approximation algorithm presented in [17]. Moreover, we present a $\frac{7}{6}$ ($\approx 1.166\bar{6}$) approximation algorithm for the *SCS2(3)* problem.

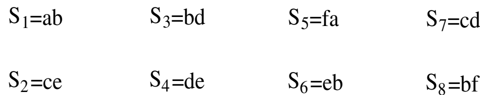
2 Approximating *SCS2*

In this section we present a simple approximation algorithm for the *SCS2* problem. Our algorithm utilizes the close relation between the *SCS2* problem and the classical *Minimum Feedback Vertex Set* problem. Our algorithm consists of the following three stages:

(i) Given a set L, construct a corresponding directed graph $G(V, E)$.
(ii) Find an approximated *Minimum Feedback Vertex Set* on $G(V, E)$.
(iii) Construct a common supersequence of L, based on the *Feedback Vertex Set* from the second stage.

In the first stage of the algorithm, we transfer the input strings of the *SCS2* instance, L, into a corresponding directed graph $G(V, E)$. Every character $c_i \in \Sigma$ represents a vertex v_i in $G(V, E)$. Then, we set a directed edge from v_i to v_j iff $c_i c_j \in L$. See Figure 1 as an example of the above construction.

The following corollary, regarding the above construction, is crucial to the analysis of our algorithm.



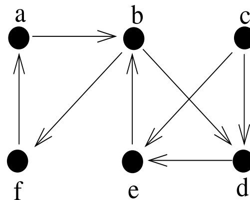


Fig. 1. From an *SCS2* instance into a directed graph

Corollary 1. *Let L be an SCS2 instance and let $G(V, E)$ be its corresponding directed graph. Denote by V_{MFVS} the Minimum Feedback Vertex Set of $G(V, E)$. The following must hold:*

(i) The SCS of the set L is of length $n + |V_{MFVS}|$.
(ii) The SCS of a set L that corresponds to an acyclic graph can be found in polynomial running time, using a topological sorting [13].

In the second stage of the algorithm, we use the classical $\log n \log\log n$ approximation algorithm [5] for the *Minimum Feedback Vertex Set* problem in order to find a feedback vertex set in $G(V, E)$. Using this set of vertices we divide Σ into two distinct groups, which will be used in the last stage of the algorithm, where we construct a common supersequence of L.

Let $V' \subseteq V$ be the feedback vertex set that we found over $G(V, E)$ and let $F \subseteq \Sigma$ be a the set characters that corresponds to V', according to the construction of $G(V, E)$. We denote with P an arbitrary permutation of F.

Given a vertex v, we denote with $E(v)$ the set of all edges that v is one of their endpoints. Similarly, we denote with $E(V')$, the set of all edges that $v_i \in V'$ is one of their endpoints. Note that $G(V\backslash V', E\backslash E(V'))$ is an acyclic directed graph, and thus, according to Corollary 1, its *SCS* can be found. Let us denote this common supersequence by P'.

We then output $P \cdot P' \cdot P$ as a common supersequence of L, where '$\cdot$' stands for the concatenation of two strings.

2.1 Approximation Ratio Analysis

Let V_{MFVS} be the *Minimum Feedback Vertex Set* of $G(V, E)$. Since we use a $\log n \log\log n$-approximation algorithm for the *Minimum Feedback Vertex Set* problem, we can conclude that $|V'| \leq |V_{MFVS}| \log n \log\log n$.

Now, according to Corollary 1, $|OPT_{scs}| = n + |V_{MFVS}|$. Moreover, we can bound the length of $P \cdot P' \cdot P$ by $\min(n + |V_{MFVS}| \log n \log\log n, 2n)$. Thus, we can conclude that our algorithm yields an approximation ratio of $\frac{2n}{n+\frac{n}{\log n \log\log n}}$ $= \frac{2\log n \log\log n}{1+\log n \log\log n} = 2 - \frac{2}{1+\log n \log\log n}$.

2.2 Time and Correctness Analysis

First we prove that $P \cdot P' \cdot P$ is a common supersequence of L. To do so, we divide the strings of L into four groups:

(1) Input strings such that both characters $\in F$
(2) Input strings such that only their first character $\in F$
(3) Input strings such that only their second character $\in F$
(4) Input strings such that both characters $\notin F$

Now, we show that every string (of length 2) that belongs to one of those groups must be a subsequence of $P \cdot P' \cdot P$. Clearly, every string in (1) must be a subsequence of $P \cdot P$. Moreover every string in (2) or (3) must be a subsequence

of $P \cdot P'$ or $P' \cdot P$, respectively. In addition, according to Corollary 1 and the construction of $G(V, E)$, every string that belongs to the fourth group must be a subsequence of P'.

Clearly the running time of the algorithm is polynomial in L. Note that we construct $G(V, E)$ in linear time and we use a polynomial time approximation algorithm for the *Minimum Feedback Vertex Set* problem. Moreover, as already mentioned, the topological sorting of $G(V \backslash V', E \backslash E(V'))$ is also done in polynomial running time.

3 Approximating *SCS2(3)*

In this section we present a $\frac{7}{6}$-approximation algorithm for the *SCS2(3)* problem. Similarly to our algorithm for the *SCS2* problem, we utilize the close relation between the *SCS2(3)* and the *Minimum Feedback Vertex Set* problems. Our algorithm consists of the same three stages as in the previous algorithm, but with a major change regarding the *Feedback Vertex Set* that we find during the second stage.

Given a directed graph $G(V, E)$ and a vertex $v_i \in V$, we denote by $d_{in}(v_i)$, $d_{out}(v_i)$ and $d(v_i)$ its in-degree, out-degree and total-degree, respectively.

As mentioned, the first stage of the algorithm where we construct $G(V, E)$ based on L is identical to the first stage of the above-described algorithm. Afterwards, we use the following algorithm, Algorithm 1, in order to find a *Feedback Vertex Set*, V', over $G(V, E)$.

Algorithm 1: Approximated Minimum Feedback Vertex Set(G(V,E))

```
V' = Ø;
Perform a cleanup process on G;
while G contains a anchored cycle C_j and an anchor v_i do
    Remove v_i and E(v_i) from G;
    V' = V' ∪ v_i;
    Perform a cleanup process on G;
while ∃ a vertex v_i that satisfies: d_in(v_i) = 2 do
    Along the path starting from v_i, find the first vertex v_j such that d(v_j) = 3;
    Remove v_i, v_j, E(v_i) and E(v_j) from G;
    V' = V' ∪ v_j;
    Perform a cleanup process on G;
    while G contains an anchored cycle C_j and an anchor v_i do
        Remove v_i and E(v_i) from G;
        V' = V' ∪ v_i;
        Perform a cleanup process on G;
output V';
```

We use the following definitions in order to simplify the description and the analysis of our algorithm.

Definition 1. *Given a directed graph $G(V, E)$, a* cleanup process *is the removal of all edges and vertices of G that are not part of any cycle.*

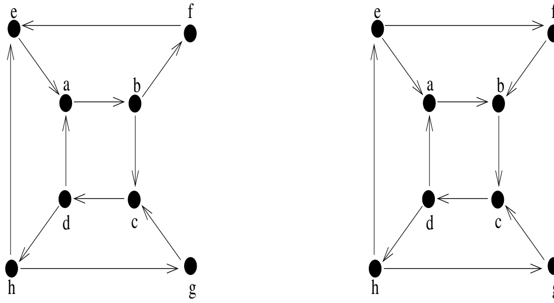


Fig. 2. Cycle 'abcd', *non-anchored* versus *anchored*

Definition 2. *Given a directed graph $G(V, E)$, a cycle C_j in G and a vertex $v_i \in C_j$. We say that C_j is an* anchored cycle *and that v_i is its* anchor *iff after performing a* cleanup process *on $G(V \setminus v_i, E \setminus E(v_i))$, the graph does not contain any of the C_j vertices.*

We use the following corollary, regarding the *anchored cycles*, in order to show that Algorithm 1 must stop.

Corollary 2. *Given a directed graph $G(V, E)$, where each vertex has a bounded degree of three, and cycle C_j in G. If C_j contains at most one vertex with in-degree $=$ 2 or at most one vertex with out-degree $= 2$ then C_j is an* anchored cycle*.*

Notice that an *anchored cycle* might have more than one *anchor*. In Figure 2 we can see that the cycle 'abcd' is not an *anchored cycle* in the left graph, while it is an *anchored cycle* in the right graph (where vertex d acts as its *anchor*).

Algorithm 1 is initialized by setting V' to $\emptyset$ and by performing a *cleanup process*. Afterwards, as long as G contains an *anchored cycle*, we attach its *anchor* (or one of them) to V' and we remove it from G. We then perform a *cleanup process* on G.

Note that, at the end of the first part of the algorithm, the graph does not contain any *anchored cycles*. Therefore, in the second part we handle the non-*anchored cycles* as follows.

As long as G contains a vertex v_i such that $d_{in}(v_i) = 2$ (and clearly $d(v_i) = 3$) we perform the following process. We search for the first vertex, v_j, along the path (starting from v_i) that satisfies: $d(v_j) = 3$. We then attach v_j to V', we remove v_i and v_j from G and perform a *cleanup process* on G. In case that G contains *anchored cycles*, we handle them (one after the other) according to the first stage of the algorithm. If no *anchored cycle* exists, we search for another vertex with in-degree of two, in order to start the above process again.

The algorithm stops iff G is empty. Note that, at the end of the second part of Algorithm 1, G does not contain:

(i) A vertex v_i such that $d_{in}(v_i) = 2$.
(ii) An *anchored cycle*.
(iii) Vertices or edges that are not part of any cycle.

According to (i), (ii) and Corollary 2, we can conclude that G does not contain cycles at all. Thus, we can conclude from (iii) that G must be empty at the end of the second stage.

We now turn to the *SCS2(3)* problem. Similarly to the previous section, let V' be a *Feedback Vertex Set*, let P be an arbitrary permutation of F (the set of characters that corresponds to V') and let P' be the *SCS* of the acyclic directed graph $G(V \backslash V', E \backslash E(V'))$.

We then output $P \cdot P' \cdot P$ as a common supersequence of L.

3.1 Approximation Ratio Analysis

In this subsection we prove that the above approximation algorithm for the *SCS2(3)* problem yields an approximation ratio of $\frac{7}{6}$. To do so, we show that $|V'| \leq |V_{MFVS}| + \frac{n}{6}$ (where V_{MFVS} is the *Minimum Feedback Vertex Set* of $G(V, E)$).

We classify the vertices of V' into two distinct groups:

(i) *ANCHORS* - vertices attached to V' as *anchors* of *anchored cycles*.
(ii) *NON-ANCHORS* - the rest of V' vertices.

The following lemma provides an upper bound on $|ANCHORS|$.

Lemma 1. $|ANCHORS| \leq |V_{MFVS}|$

Proof. Let x be the maximum number of vertex disjoint cycles in $G(V, E)$. Notice that after attaching an anchor into V', we perform a *cleanup process*. Thus, we remove all the vertices and all the edges of an *anchored cycle* from G. Therefore, $|ANCHORS| \leq x \leq |V_{MFVS}|$. □

The following Lemma provides an upper bound on $|NON\text{-}ANCHORS|$.

Lemma 2. $|NON\text{-}ANCHORS| \leq \frac{n}{6}$

Proof. Let $v_j \in NON\text{-}ANCHORS$ and let y be the number of degree 3 vertices in G (clearly $y \leq n$). We now show that after attaching v_j into V' we reduce y by at least 6.

Recall that we only attach a *non-anchor* vertex into V' during the second part of Algorithm 1. In that case, we select an arbitrary vertex v_i such that $d_{in}(v_i) = 2$ and then we search for the first vertex, v_j, along the path (starting from v_i) that satisfies $d(v_j) = 3$. We then attach v_j to V', we remove v_i and v_j from G and we perform a *cleanup process* on G.

Since we remove v_i and v_j from G, y is already reduced by 2. We now focus on the *cleanup process*. Different from the previous cleanup processes, here we detail how the process works in order to achieve tighter bounds.

Let $e_1 \in E(v_i)$ be an edge that is not part of the path from v_i to v_j. We start the *cleanup process* by deleting e_1 from G. We then continue travelling (backwards) along the path that ends with e_1. We remove all the vertices and all the edges along this path until we face a vertex v_k that satisfies: $d(v_k) = 3$ and $d_{out}(v_k) = 2$. In this case we only remove a single outgoing edge of v_k. If we did not face such a vertex then we already removed an *anchored cycle* from

G, which contradicts the assumption that G does not contain such cycles. Thus, we can conclude that we must face such a vertex v_k which is not of degree 3 anymore. Therefore, we can reduce y by one.

We do the same for additional three edges: $e_2 \in E(v_i)$ and $e_3, e_4 \in E(v_j)$. Note that in case of forward travelling (instead of backwards) we continue until we face a vertex v_k that satisfies: $d(v_k) = 3$ and $d_{in}(v_k) = 2$. Therefore, during the *cleanup process* we reduce y by at least 4.

Altogether, for every vertex $v_j \in$ *NON-ANCHORS* we must reduce y by at least 6 and thus, $|NON\text{-}ANCHORS| \leq \frac{n}{6}$. □

Now, according to Corollary 1, $|OPT_{scs}| = n + |V_{MFVS}|$. Moreover, according to Lemma 1 and Lemma 2, we can bound $|P \cdot P' \cdot P|$ by $\min(\frac{7n}{6} + |V_{MFVS}|, 2n)$. Thus, we can conclude that our algorithm for the *SCS2(3)* problem yields an approximation ratio of $\frac{7}{6}$.

3.2 Time and Correctness Analysis

As long as the output of Algorithm 1 is a *Feedback Vertex Set*, the correctness analysis from the previous section must also holds for this case.

Since in every step of the algorithm we only remove edges and vertices that are not part of any cycle in $G(V \setminus V', E \setminus E(V'))$, we can conclude that the output must be a *Feedback Vertex Set*.

Note that the running time of Algorithm 1 is polynomial in L and thus, our running time analysis from the previous section must also holds for this case.

4 Conclusion and Open Questions

In this paper, we presented a $2 - \frac{2}{1+\log n \log\log n}$ approximation algorithm for the *SCS2* problem, which immediately improves the $2 - \frac{4}{n+1}$ approximation algorithm presented in [17]. We also presented a $\frac{7}{6}$-approximation algorithm for the *SCS2(3)* problem.

Our algorithms are based on the close relation between the *SCS* and the *Minimum Feedback Vertex Set* problems. To tighten this close relation, it is easy to see that using our construction of $G(V, E)$, any γ-approximation algorithm for the *Minimum Feedback Vertex Set* problem can lead to a $2 - \frac{2}{\gamma+1}$ approximation algorithm for the *SCS2* problem.

A natural open question is whether there are better approximation algorithms for the the *SCS2* and the *SCS2(3)* problems, which improves the above approximation factors ?

References

1. Bafna, V., Lawler, E.L., Pevzner, P.A.: Approximation Algorithms for Multiple Sequence Alignment. Theoretical Computer Science 182(1-2), 233–244 (1997)
2. Barone, P., Bonizzoni, P., Vedova, G.D., Mauri, G.: An approximation algorithm for the shortest common supersequence problem: an experimental analysis. In: ACM Symposium on Applied Computing, pp. 56–60 (2001)

3. Cotta, C.: A Comparison of Evolutionary Approaches to the Shortest Common Supersequence Problem. In: Cabestany, J., Prieto, A.G., Sandoval, F. (eds.) IWANN 2005. LNCS, vol. 3512, pp. 50–58. Springer, Heidelberg (2005)
4. Cotta, C.: Memetic Algorithms with Partial Lamarckism for the Shortest Common Supersequence Problem. In: Mira, J., Álvarez, J.R. (eds.) IWINAC 2005. LNCS, vol. 3562, pp. 84–91. Springer, Heidelberg (2005)
5. Even, G., Naor, J., Schieber, B., Sudan, M.: Approximating Minimum Feedback Sets and Multicuts in Directed Graphs. Algorithmica 20(2), 151–174 (1998)
6. Fraser, C.B., Irving, R.W.: Approximation Algorithms for the Shortest Common Supersequence. Nordic Journal of Computing 2(3), 303–325 (1995)
7. Jiang, T., Li, M.: On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. SIAM Journal on Computing 24(5), 1122–1139 (1995)
8. Maier, D.: The Complexity of Some Problems on Subsequences and Supersequences. Journal of the ACM 25(2), 322–336 (1978)
9. Middendorf, M.: The Shortest Common Nonsubsequence Problem is NP-Complete. Theoretical Computer Science 108(2), 365–369 (1993)
10. Pevzner, P.A.: Multiple Alignment, Communication Cost, and Graph Matching. SIAM Journal on Applied Mathematics 52(6), 1763–1779 (1992)
11. Pietrzak, K.: On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. Journal of Computer and System Sciences 67(4), 757–771 (2003)
12. Räihä, K.J., Ukkonen, E.: The Shortest Common Supersequence Problem over Binary Alphabet is NP-Complete. Theoretical Computer Science 16, 187–198 (1981)
13. Reingold, E.M., Nievergelt, J., Deo, N.: Combinatorial Algorithms. Prentice-Hall Inc., Englewood Cliffs (1977)
14. Rubinov, A.R., Timkovsky, V.G.: String Noninclusion Optimization Problems. SIAM Journal on Discrete Mathematics 11(3), 456–467 (1998)
15. Sankoff, D.: Minimal Mutation Trees of Sequences. SIAM Journal on Applied Mathematics 28, 35–42 (1975)
16. Timkovsky, V.G.: Complexity of common subsequence and supersequence problems and related problems. Kibernetika 5, 1–13 (1989); English Translation in Cybernetics 25, 565–580 (1990)
17. Timkovsky, V.G.: Some Approximations for Shortest Common Nonsubsequences and Supersequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 258–269. Springer, Heidelberg (2008)

Set Intersection and Sequence Matching

Ariel Shiftan and Ely Porat

Department of Computer Science, Bar-Ilan University, Ramal Gab 52900, Israel
{shiftaa,porately}@cs.biu.ac.il

Abstract. In the classical pattern matching problem, one is given a text and a pattern, both of which are sequences of letters, and is required to find all occurrences of the pattern in the text. We study two modifications of the classical problem, where each letter in the text and pattern is a set (*Set Intersection Matching* problem) or a sequence (*Sequence Matching* problem). Two "letters" are considered to be match if the intersection of the two corresponding sets is not empty, or if the two sequences have a common element in the same index. We show the first known non-trivial and efficient algorithms for these problems, for the case the maximum set/sequence size is small. The first, randomized, that takes $\Theta\left(2^d n \ln n \log m\right)$ time, where d is the maximum set/sequence size, and can also fit, with slight modifications, for the case one is also interested in up to k mismatches. The second is deterministic and takes $\Theta\left(4^d n \log m\right)$. The third algorithm, also deterministic, is able to count the number of matches at each index of the text in total running time $\Theta\left(\sum_{i=1}^{d} \binom{|\Sigma|}{i} n \log m\right)$.

1 Introduction and Related Work

The classic pattern matching problem is defined as follows: We are given a text $T = t_1, t_2, t_3, ..., t_n$ of size n and a pattern $P = p_1, p_2, p_3, ..., p_m$ of size m, which are both sequences of letters belonging to a pre-defined set — the alphabet $\Sigma = u_1, u_2, u_3, ..., u_l$. We are required to find all occurrences of the pattern in the text. Linear time solutions were given in [7,8].

Two forms of approximation for the problem which are commonly researched involves *don't cares* and *mismatch count*. The first is about the presence of a wild-card letter, which matches any other letter, was first solved in $\Theta(n \log m \log |\Sigma|)$ time [4], and later in $\Theta(n \log m)$ time [11,10]. The second is match/mismatch count, e.g. get the number of matching letters when comparing the pattern to the text at location i, for all locations. This mismatch count (at each location compared) is actually the Hamming distance between the text and the pattern, and was given an $\Theta(n|\Sigma| \log m)$ time solution in [4].

The subset matching problem, first introduced by Cole and Hariharan [1], extends that classic string matching problem and defines both the pattern and text to be sequences of sets of characters. Formally, each text location t_i and each pattern location p_i is a set of characters, not a single character, taken from a certain alphabet. Pattern P is said to match text T at location i, if $p_j \subseteq t_{i+j}$

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 285–294, 2009.
© Springer-Verlag Berlin Heidelberg 2009

for all j, $0 \leq j < m$. Cole and Hariharan proposed a near-linear time randomized algorithm [1], and improved it [2] to a deterministic one. Amihood, Porat and Lewenstein proposed [3] approximate version with don't cares.

Generalized strings over alphabet Σ are sequences of sets, where each set is a subset of sigma, which are possibilities for letters in that position. For example, the generalized string [{a,b},{b,d}] matches the strings ab, ad, bb, bd. This type of strings is used in [6], and while specifying the pattern to look for in regular expressions.

As the subset matching problem, the two problems studied in this paper define the pattern and text to be sequences of sets (or sequences) too, but a match between two 'letters' is when the intersection of the two sets is not empty, in the Set Intersection Matching problem, or when the two sequences have a common element in the same place, in the Sequence Matching problem (formal definitions are given in the next section). The Set Intersection Matching problem is actually searching for a generalized string inside another one.

Sample applications for the algorithms in this paper are when there is possible errors in both the pattern and the text. For example, consider the case when both the pattern and the text were acquired using an OCR algorithm that reports few options for each letter. One can create a set of all possible letters for each such letter, and then run Set Intersection Matching algorithm in order to get the occurrences of such faulty pattern in the text.

1.1 Overview

Section 2 defines the two problems formally. Then, sections 3,4,5 presents the three different algorithms for the problem. Finally, section 6 concludes and proposes future work.

2 Preliminary

2.1 Problem Definition

In the Set Intersection Matching problem, each letter is actually a set, and two of those 'letters' match if the intersection between them is not empty.

more formally:

Definition 1. The Set Intersection Matching Problem
Input:

- *Alphabet* $\Sigma = u_1, u_2, u_3, ..., u_l$
- *Text* $T = t_1, t_2, t_3, ..., t_n$*, where each* t_i *is a set of letters from the alphabet* Σ*, and* $|t_i|$ *is the size of the set* t_i*.*
- *Pattern* $P = p_1, p_2, p_3, ..., p_m$*, where each* p_i *is a set of letters from the alphabet* Σ*, and* $|p_i|$ *is the size of the set* p_i*.*
- d *— maximum set size. Formally:* $d = max\left(\max_{i\in[1,m]}|p_i|, \max_{j\in[1,n]}|t_j|\right)$

Output: All locations $1 \leq i \leq n-m+1$*, such that for all* $0 \leq j \leq m-1$: $t_{i+j} \cap p_j \neq \emptyset$

Example 1. Consider the text T = [{a,b}, {b,c,d}, {a,d}, {a,c,d}, {b}] and the pattern P = [{a,b}, {a,c}, {c,d}]. The matches for all possible locations in the text are described in Table 1.

Table 1. Set Intersection example

Index (i)	Intersection (j)			Match
	1	2	3	
1	{a,b}	{c}	{d}	√
2	{b}	{a}	{c,d}	√
3	{a}	{a,c}	∅	×

Note that the above definition ignores the order of the elements inside each set. In contrast, the Sequence Matching problem does consider the order. In this problem, both the pattern and the text are composed of sequences of letters from the alphabet, and a match between two sequences occurs when there is at least one index where both sequences have the same letter. Formally:

Definition 2. The Sequence Matching Problem
Input:

– *Alphabet* $\Sigma = u_1, u_2, u_3, ..., u_l$
– *Text* $T = t_1, t_2, t_3, ..., t_n$*, where each* t_i *is a sequence of letters from the alphabet* Σ*, and* $|t_i|$ *is the size of the sequence* t_i*.*
– *Pattern* $P = p_1, p_2, p_3, ..., p_m$ *, where each* p_i *is a sequence of letters from the alphabet* Σ*, and* $|p_i|$ *is the size of the sequence* p_i*.*
– d *— maximum sequence size. Formally:*
$d = max\left(\max_{i\in[1,m]}|p_i|, \max_{j\in[1,n]}|t_j|\right)$

Output: All locations $1 \leq i \leq n - m + 1$*, such that for all* $0 \leq j \leq m - 1$*, there exists* $1 \leq k \leq min\left(|t_{i+j}|, |p_j|\right)\ :\ t_{i+j}[k] = p_j[k]$

Example 2. Consider the text T = [[a,b], [b,c,d], [a,d], [a,c,d], [b]] and the pattern P = [[a,b], [a,c], [c,d]]. The matches for all possible locations in the text are described in Table 2.

Table 2. Sequence Matching example

Index (i)	Matching? (j)			Match
	1	2	3	
1	[1,0]	[0,1,0]	[0,1]	√
2	[0,0,0]	[1,0]	[0,0,0]	×
3	[1,0]	[1,1]	[0,0]	×

The two problems are equivalent, up to a small difference in efficiency. Thus, we will allow ourselves to solve in the sequel only one of the following.

To reduce Set Intersection to Sequence Matching one can replace each of the elements in each sequence with a couple represents the element and its index. For example: $[\{a, b\}, \{b, c, d\}] \Rightarrow [\{(a, 1), (b, 2)\}, \{(b, 1), (c, 2), (d, 2)\}]$. By doing so, the intersection can contain only elements that are in the same index, which is actually the definition of the Sequence Matching problem. The price we pay for it is the enlargement of the alphabet size — Σ, where the new size is bounded by $\Sigma \times d$.

To reduce Sequence Matching to Set Intersection one should first repeat the text set's letters d times each. Then, he should pad the patters set's letters and repeat the new set (including the padding) d times. This causes the intersection to appear in the same index of both the pattern and the text. For example: $T = [[a, b], [b, c, d]], P = [[a, b], [a, c]] \Rightarrow T = [[a, a, a, b, b, b], [b, b, b, c, c, c, d, d, d]], P = [[a, b, X, a, b, X, a, b, X], [a, c, X, a, c, X, a, c, X]]$ for d=3 and X a padding letter. In this case the price is the squaring of d.

2.2 Hardness of the Problem

As one can easily see, the problem can be trivially solved in time $\Theta\left(nmd^2\right)$.

The main hardness in the above problems lies in the lack of the following transitive property, which is the base for fast pattern matching algorithms. The transitive relation states that if $a = b$ and $b = c$ then $a = c$, and these algorithms use this property in order to avoid comparing elements which their matching could be concluded. As this property does not hold in those problems, the existing algorithms cannot be used.

3 Randomized Algorithm

In this section we show a randomized algorithm for the Set Intersection Matching problem. We start by giving a 'bad' randomized algorithm - an algorithm that is wrong with very high probability, and then we reduce the failure probability by running the algorithm several times.

The algorithm works as follows: First, we choose a random hash function $h : \Sigma \to \{0, 1\}$. Then, we build a new text T_h using linear phase which scans the original text T, and writes at position i of T_h one of $0, 1$ or ϕ (where ϕ stand for don't care) depending on the value of T at position $i - t_i$. We write 0 if for all $x \in t_i$ $h(x) = 0$, 1 if for all $x \in t_i$ $h(x) = 1$, and ϕ if $\exists x, y \in t_i$ $h(x) = 0$ $h(y) = 1$. We build P_h from the original pattern P using the same way.

The idea of the algorithm is based on the following two trivial lemmata:

Lemma 1. *If $t_{i+j} \cap p_j \neq \emptyset$ then for any hash function, the resulting letters will not match.*

Lemma 2. *If $t_{i+j} \cap p_j = \emptyset$ the probability that these two places will not match is $2^{-(|t_{i+j}|+|p_j|)}$.*

Our 'bad' random algorithm will run regular pattern matching algorithm on binary alphabet to match between T_h and P_h. If we want only to test for match or not, we could use two convolutions (which are $O(n \log m)$). For the case we also interested in up to k errors (up to k sets that does not intersect), we can run the algorithm from [5] which will return the places of the mismatch in time $O(nk \log^2 m)$. Note that the algorithm, in either case, fails to return the true answer with high probability.

Theorem 1. *If we run the 'bad' random algorithm* $\frac{3}{2}4^d \ln n$ *times the failure probability of the algorithm will be less than* $\frac{1}{n}$.

Proof. We deal with the following cases separately: Testing for an intersection match or not, and counting the number of places that does not intersect up to k.

In the first case, look on a specific shift in which we check whether the pattern matches the text. If the pattern matches at this position, our algorithm will always return that it matches. If the pattern does not match on that location, then with probability higher than 2^{-2d+1} each iteration reports that it does not match. Therefore, with probability less then $(1-2^{-2d+1})^{\frac{3}{2}4^d \ln n} = \frac{1}{n^3}$ we will obtain a false match. As we do the same for less than n shifts, the overall probability that there exist a shift which our algorithm fail on it is bounded by $\frac{1}{n^2}$.

If we want to count the non intersect groups up to k, assume that $t_{i+j} \cap p_j = \emptyset$. The probability to miss it at each iteration is less than $1 - 2^{-2d+1}$. Therefore the probability that we will miss it in all of the iterations is bounded by $(1-2^{-2d+1})^{\frac{3}{2}4^d \ln n} = \frac{1}{n^3}$. We have at most nm pairs for which $t_{i+j} \cap p_j = \emptyset$, hence the probability to miss one of them is bounded by $\frac{m}{n^2} < \frac{1}{n}$.

4 Deterministic Algorithm

4.1 Sequence Matching Algorithm

In this section we first show simple algorithm for the Sequence Matching problem in the case $d = 2$, and then generalize it for $d > 2$.

4.2 Simple Case Study ($d = 2$)

Assume the length of all sequences is lesser than or equal 2. It is easy to see that one can pad all the sequences which are smaller than 2 with unused letters (one for text sequences, and another one for pattern sequences), in order to get all sequences to be of the same length, so we can assume that all sequences are of length 2.

Theorem 2. *The sequence matching problem, in the case where for all* $1 \le i \le m$ $|p_i| \le 2$ *and for all* $1 \le i \le n$ $|t_i| \le 2$*, can be solved in* $\Theta(n \log m)$ *time.*

Proof. Denote the sequences of the text $T = t_1, t_2, t_3, ..., t_n$ by $t_i = [x_i, y_i]$, and the sequences of the pattern $P = p_1, p_2, p_3, ..., p_m$ by $p_i = [a_i, b_i]$, and mind the following formula for a pre-defined i:

$$\sum_{j=0}^{m-1} [(x_{i+j} - a_j)(y_{i+j} - b_j)] \tag{1}$$

Each of the summand in (1) equals 0 if and only if $x_{i+j} = a_j$ or $y_{i+j} = b_j$, which means the two sequences match. Thus equals 0 if the text matches the pattern at index i.

Expanding each summand yields:

$$\sum_{j=0}^{m-1} (x_{i+j}y_{i+j} - a_j y_{i+j} - b_j x_{i+j} + a_j b_j) \tag{2}$$

In order to solve the problem one should get the value of the formula for all $1 \leq i \leq n - m + 1$, and this can be done by calculating each of the four parts independently, and sum it up afterwards. The first and last parts can be easily calculated for all i's in $\Theta(n + m)$ time, by one pass over the text and pattern and handling a window. The two other parts can be easily calculated for all i's as well, in $\Theta(n \log m)$ time using convolutions [4].

Notice that the sum for few i's can be 0 by chance. In order to avoid that one should calculate the sum of squares of the formula. For details see the next subsection.

4.3 The General Solution

We now generalize the previous algorithm for cases where $d > 2$. As in the former case, it is easy to see that one can pad all the sequences which are smaller than d with unused letters, in order to get all sequences to be of the same length, so we can assume that all sequences are of size d. The following theorem with its proof shows that there is an algorithm for solving the problem in $\Theta\left(4^d n \log m\right)$:

Theorem 3. *The sequence matching problem, in the case where for all* $1 \leq i \leq m\, |p_i| \leq d$ *and for all* $1 \leq i \leq n\, |t_i| \leq d$*, can be solved in* $\Theta\left(4^d n \log m\right)$ *time.*

Proof. Denote the sequences of the text and pattern by $t_i = [x_{i_1}, x_{i_2} \ldots x_{i_d}]$ and $p_i = [a_{i_1}, a_{i_2} \ldots a_{i_d}]$, respectively, and consider the following formula for a pre-defined i:

$$\sum_{j=0}^{m-1} \left[\left(x_{i+j_1} - a_{j_1}\right)\left(x_{i+j_2} - a_{j_2}\right) \ldots \left(x_{i+j_d} - a_{i_d}\right)\right] \tag{3}$$

As before, each summand in (3) equals 0 if and only if $x_{i+j_1} = a_{j_1}$ or $x_{i+j_2} = a_{j_2}$ or $\ldots$ or $x_{i+j_d} = a_{j_d}$, which means the two sequences match. Hence the

sum of the formula, which is the result of summation of m contiguous such parts, equals 0 if there is a match at index i.

Expanding each summand yields:

$$\sum_{j=0}^{m-1} \left(x_{i+j_1} x_{i+j_2} \dots x_{i+j_d} - x_{i+j_1} x_{i+j_1} \dots x_{i+j_d-1} a_{j_d} \;\dots\; + (-1)^d a_{j_1} a_{j_2} \dots a_{j_d} \right) \tag{4}$$

In order to solve the problem one should get the value of the formula for all $1 \leq i \leq n-m+1$, and this can be done by calculating each term independently, and sum it up later. Notice that the formula consists of 2^d terms, where each of them are multiplication of text elements of a specific sequence, and pattern elements of another sequence. For each part, the multiplication between elements from the text can be calculated for all i's in linear time, and the same for multiplication between elements from the pattern, using a window as in the previous subsection. Then, using convolutions we can get the term's value for all $1 \leq i \leq n-m$, in total time $\Theta\left(2^d n \log m\right)$.

As before, in order to avoid getting 0 by chance, one should calculate the sum of squares of the formula. All parts of formula (3) will be squared, and the result is $2^{2d} = 4^d$ parts in the extracted formula, with similar structure.

5 Deterministic Algorithm with Mismatch Count

In this section we present another deterministic algorithm for the Set Intersection Matching problem, which is also able to count the number of mismatches between the pattern and any location in the text.

Theorem 4. *The Set Intersection Matching problem, can be solved in time* $\Theta\left(\sum_{i=1}^{d} \binom{|\Sigma|}{i} n \log m\right)$ *(which is bounded by* $O\left(|\Sigma|^d n \log m\right)$*).*

Proof. We define (S,i)-*match-count* for a set S of letters from Σ, and index i in the pattern, to be the number of times S is in the intersection, when checking for intersection match at index i of the text. Formally,

Definition 3. (S,i)-match-count: *for* $S \subseteq \Sigma$ *and* $0 < i \leq n-m+1$*,* (S,i)-*match-count* $= |\{j \mid 0 \leq j < m$ *and* $S \subseteq t_{i+j} \cap p_{j+1}\}|$

We denote (S,i)-*match-count* by $\Phi(S,i)$. The value of $\Phi(S,i)$ for a given S subset of Σ can be calculated in $\Theta(n \log n)$ for all i's in the following way: First, we replace each set in the text and the pattern with 1 if S is a subset of it, and 0 otherwise. Then, we use the convolution method to get the desired result.

In addition, we need the following result from the binomial expansion: by setting $x = -1$ in the binomial expansion $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$ we get

$$
\begin{aligned}
0 &= \sum_{k=0}^{n} \binom{n}{k} (-1)^k \\
s \Rightarrow -\binom{n}{0} (-1)^0 &= \sum_{k=1}^{n} \binom{n}{k} (-1)^k \\
\Rightarrow -1 &= \sum_{k=1}^{n} \binom{n}{k} (-1)^k \\
\Rightarrow 1 &= \sum_{k=1}^{n} \binom{n}{k} (-1)^{k+1} \qquad (5)
\end{aligned}
$$

Now, consider the case where one is trying to match the pattern P to the text T at location i. We now show that

Lemma 3. *The number of set intersection matches at location i is equal to*

$$\sum_{S \subseteq \Sigma, |S| \leq d} (-1)^{|S|+1} \Phi(S, i) \qquad (6)$$

The value of formula (6) is summation of all (S, i)-match-count for all subsets S of odd size of Σ, minus all (S, i)-match-count for all subsets S of even size. Thus, for each of the locations we intersect and test for match (e.g. t_i against p_1, t_{i+1} against p_2, etc.), the formula counts the number of odd subsets of the intersection minus the number of even subsets of it, and sum it all up afterwards. Assuming the intersection size in such location is k, we get $\sum_{i=1}^{k} \binom{k}{i} (-1)^{i+1}$, which equals 1 using (5). Notice that for locations where the intersection is empty the formula counts nothing, hence we get that formula (6) equals exactly the number of set intersection matches at location i, as claimed by the lemma.

The proof of the theorem is now trivial, as we already see that $\Phi(S, i)$ can be calculated in time $\Theta(n \log m)$, and the number of elements summed up in the lemma is exactly $\sum_{i=1}^{d} \binom{|\Sigma|}{i}$ (Note that $d \leq |\Sigma|$, as all sets are subsets of Σ).

Example 3. Consider the text $T = [\{a,b\}, \{b,c,d\}, \{a,d\}, \{a,c,d\}, \{b\}]$ and the pattern $P = [\{a,b\}, \{a,c\}, \{c,d\}]$. (S, i)-match-count for all possible indexes i, and for all relevant subsets $S \subseteq \Sigma$ are described in Table 3. One can see that the result is exactly the mismatch count of the Set Intersection Matching problem, for each matching location i.

Example 4. Consider the case one is trying to calculate the match count between the above pattern to the above text at location 3 of the text. Analysis of correctness of the algorithm for that case is described in Table 4. The last column, which sums up the rows, is exactly the same as the last column of the previous example. By summing up columns (odd size - even size) we get 1 for each index

Table 3. Set Intersection match count example

S	$(S, 1)$-match-count	$(S, 2)$-match-count	$(S, 3)$-match-count
{a}	1	1	2
{b}	1	1	0
{c}	1	1	1
{d}	1	1	0
{a,b}	1	0	0
{a,c}	0	0	1
{a,d}	0	0	0
{b,c}	0	0	0
{b,d}	0	0	0
{c,d}	0	1	0
sum(odd)-sum(even)	3	3	2

Table 4. Set Intersection match count analysis

S	Intersection (j)			$(S, 3)$-match-count
	$\{a, d\} \cap \{a, b\} = \{a\}$	$\{a, c, d\} \cap \{a, c\} = \{a, c\}$	$\{b\} \cap \{c, d\} = \emptyset$	
{a}	1	1	0	2
{b}	0	0	0	0
{c}	0	1	0	1
{d}	0	0	0	0
{a,b}	0	0	0	0
{a,c}	0	1	0	1
{a,d}	0	0	0	0
{b,c}	0	0	0	0
{b,d}	0	0	0	0
{c,d}	0	0	0	0
sum(odd)-sum(even)	1	$2 - 1 = 1$	0	$3 - 1 = 2$

where the intersection is not empty, and 0 otherwise. Hence it is clear why we get the match count, which is 2, when summing up the last column (odd size - even size).

6 Conclusions and Future Work

We studied the Set Intersection Matching and Sequence Matching problems, and presented first known non-trivial and efficient (for the case the maximum set/sequence size is small) algorithms for solving those problem: randomized one for the Set Intersection problem, deterministic for the Sequence Matching problem, and another deterministic with match count for the Set Intersection problem. The time complexity of all of the algorithms is exponential in the size

of the largest sequence/set. We also presented reductions between the problems, hence all of the above algorithms can solve both problems (with the cost of the reduction).

Future work is required in order to develop efficient algorithms for the case the sequences/sets and the alphabet are big, or in order to prove hardness and that such algorithm does not exist.

References

1. Cole, R., Hariharan, R.: Tree Pattern Matching to Subset matching in Linear Time. SIAM J. Comput. 2(4), 1056–1066
2. Cole, R., Harihan, R., Indyk, P.: Tree pattern matching and subset matching in deterministic $o(n \log^3 n)$time. In: Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 245–254 (1999)
3. Amir, A., Porat, E., Lewenstein, M.: Approximate subset matching with Don't Cares. In: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, Washington, D.C., United States, January 07-09, 2001, pp. 305–306 (2001)
4. Fischer, M.J., Paterson, M.S.: String-matching and other products. Technical Report. UMI Order Number: TM-41., Massachusetts Institute of Technology (1974)
5. Clifford, R., Efremenko, K., Porat, E., Rothschild, A.: From coding theory to efficient pattern matching. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 778–784 (2009)
6. Marschall, T., Rahmann, S.: Probabilistic Arithmetic Automata and Their Application to Pattern Matching Statistics. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 95–106. Springer, Heidelberg (2008)
7. Boyer, R.S., Moore, J.S.: A fast string matching algorithm. Communications of the ACM 20, 762–772 (1977)
8. Knuth, D.E., Morris, J.H., Pratt, V.B.: Fast pattern matching in strings. SIAM Journal of Computing 6, 323–350 (1977)
9. Abrahamson, K.: Generalized string matching. SIAM J. 16(6), 1039–1051 (1987)
10. Clifford, P., Clifford, R.: Simple deterministic wildcard matching. Inf. Process. Lett. 101(2), 53–54 (2007)
11. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the Thiry-Fourth Annual ACM Symposium on theory of Computing (STOC 2002), Montreal, Quebec, Canada, May 19 - 21, 2002, pp. 592–601. ACM, New York (2002)

Generalised Matching

Raphael Clifford[1], Aram W. Harrow[2],
Alexandru Popa[1], and Benjamin Sach[1]

[1] Department of Computer Science, University of Bristol, UK
{clifford,popa,sach}@cs.bris.ac.uk
[2] Department of Mathematics, University of Bristol, UK
a.harrow@bris.ac.uk

Abstract. Given a pattern p over an alphabet Σ_p and a text t over an alphabet Σ_t, we consider the problem of determining a mapping f from Σ_p to Σ_t^+ such that $t = f(p_1)f(p_2)\ldots f(p_m)$. This class of problems, which was first introduced by Amir and Nor in 2004, is defined by different constraints on the mapping f. We give NP-Completeness results for a wide range of conditions. These include when f is either many-to-one or one-to-one, when Σ_t is binary and when the range of f is limited to strings of constant length. We then introduce a related problem we term *pattern matching with string classes* which we show to be solvable efficiently. Finally, we discuss an optimisation variant of generalised matching and give a polynomial-time $\min(1, \sqrt{k/\mathrm{OPT}})$-approximation algorithm for fixed k.

1 Introduction

We consider a class of pattern matching problems where individual characters in the pattern are permitted to match entire substrings of the text. When the substrings are restricted to have length exactly one, then the problems of finding efficient search algorithms are exactly those typically found in the rich and successful literature of combinatorial pattern matching. Our interest here lies when the substrings may have length greater than one. In many cases, it has not been known up to this point even whether polynomial-time pattern matching algorithms exist.

The problems we analyse take the following general form. The input is a pattern $p = p_1p_2\ldots p_m$ and a text $t = t_1t_2\ldots t_n$. We wish to find a mapping f from Σ_p to Σ_t^+ (substrings of t of length at least one) so that $t = f(p_1)f(p_2)\ldots f(p_m) = f(p)$. For example, if $p = aba$ and $t = xyyx$ then if f maps $a \rightarrow x$, $b \rightarrow yy$, we say that there is a match. However if $t = xyyz$, then no such mapping can be found. The first results in this model were given by Amir and Nor [2,3] who considered a problem called generalised function matching (GFM) with wildcards. Here single character wildcards may occur in the input and f is allowed to be any function. They show that if the pattern alphabet has constant size, then a polynomial algorithm can be found but that the problem is NP-Complete otherwise. The problem of generalised parameterised matching

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 295–301, 2009.
© Springer-Verlag Berlin Heidelberg 2009

(GPM) is also defined in an analogous way to GFM except that f is now required to be an injection. That is if $p = aba$ and $t = xxx$ then there is a GFM match but not a GPM match as a and b cannot both map to x in a generalised parameterised match. As the names suggest, these problems arise from a natural extension of parameterised matching, introduced by Baker [4] in 1993, and function matching, which was considered by Amir et al. [1].

Our contributions are twofold. Primarily, we answer a number of open problems posed by Amir and Nor in their initial work. Namely, we prove that both GFM and GPM are NP-Complete with or without wildcards (Sections 2 and 3) and give a $\min(1, \sqrt{k/\mathrm{OPT}})$-approximation for a Hamming distance based variant of GFM for fixed k (Section 5). We also extend the work to include a new variant of generalised matching based on pattern matching with string classes (Section 4). Due to space constraints, some of the proofs are omitted.

2 GFM Is NP-Complete

In this Section we show that the problem of generalised function matching is NP-Complete (with or without wildcards) via a reduction from 3-SAT. Specifically, given an instance of 3-SAT with N variables and M clauses we show how to construct an instance of GFM (with length polynomial in M) which has a solution if and only if the 3-SAT instance is satisfiable. In the process we show that the reduction is valid even for quite severely restricted versions of the problem.

The construction starts by prepending two \$ symbols to the beginning of both the pattern and the text. The text alphabet Σ_t has only two symbols, \$ and 0 with \$ serving as a delimiter in both the pattern and text. The pattern alphabet Σ_p will include the delimiter, a pair a_i and A_i for each variable and a distinct symbol c_i for each clause. The A_i's represent the negation of the variables a_i. The constructed pattern and text will contain an equal number of \$ characters which forces \$ to map to \$ under any valid function, since any other mapping of \$ must contain \$\$ as a prefix, breaking the equality assumption.

For each variable a_i, we add to the text the string \$000\$ and to the pattern $\$a_iA_i\$$. In this way a variable can be mapped to 00 or 0. To fix notation we say that 0 represents True and 00 represents False. For each clause, we add to the text the string \$000000\$ (6 zeros) and to the pattern the string $\$xyzc_i\$$ where x, y, z are the variables from the clause (or their negations, as appropriate) and c_i is a different symbol for each clause.

Theorem 1. *The generalised function matching problem is NP-Complete.*

Proof. GFM is in NP as any candidate mapping function f can be checked in polynomial time. To show NP-Hardness, take a 3-SAT instance ϕ and use the above construction to produce a pattern and text, called p and t respectively. We prove that there is a GFM solution for p and t if and only if ϕ is satisfiable.

If there is a GFM solution then we know that the \$ symbol in the pattern is matched to the same symbol in the text. The variable gadgets also ensure that variables have consistent assignments. The clause gadgets ensure that at least

one of the three literals in each clause is assigned to the value 0, representing True. Therefore, it suffices to read the mapping found to give an assignment of truth values to variables which satisfies ϕ.

If ϕ is satisfiable, then there must be a GFM solution for p and t. This follows as we are guaranteed that not all the symbols from a clause can be mapped to 00 and therefore c_i will be able to be matched to a nonempty substring in each clause gadget.

Finally, we observe that, with a small modification, the proof still holds under two severe restrictions:

Corollary 1. *Generalised function matching remains NP-Complete when Σ_t contains only two distinct symbols and we restrict f so that $|f(x)| \leq 2$ for all $x \in \Sigma_p$.*

3 GPM Is NP-Complete

Generalised parameterised matching adds an extra constraint to the conditions set by GFM. The mapping f must now be injective.

It is instructive first to see why we cannot simply translate the reduction for GFM. The problem is that all "variable characters" are mapped to 0 or 00 in the GFM reduction so a GPM solution could never occur with more than two variables. Therefore we need to design new gadgets to overcome this difficulty.

We present a reduction from 1-in-3 SAT, an NP-Complete variant of 3-SAT [6]. 1-in-3 SAT is defined as 3-SAT but with the additional constraint that in a satisfying assignment, each clause must contain exactly one True literal, instead of at least one for 3-SAT. Given an instance of 1-in-3 SAT, ϕ, with N variables named $a_1, a_2 \ldots a_N$ and M clauses, we construct an instance of GPM which matches if and only if ϕ is satisfiable. The instance has the form, $t = \$\$V_tC_t$ and $p = \$\$V_pC_p$. Here V_t and V_p encode the variables in ϕ, and C_t and C_p encode the clauses. As in Section 2, the \$\$ symbols ensure that \$ maps to \$.

We will often say that $f(\{a, b\}) = \{x, y\}$, by which we mean that $(f(a) = x$ and $f(b) = y)$ or $(f(a) = y$ and $f(b) = x)$ and also define 0^k to be a string of 0s of length k. For example, $0^3 = 000$.

Each variable results in a pair of strings P_i, T_i. We then concatenate these to form variable gadgets $V_p = P_1P_2 \ldots P_N$ and $V_t = T_1T_2 \ldots T_N$. The components P_i, T_i are defined as $P_i = \$a_iA_i\$$ and $T_i = \$0^{4i-1}\$$.

As with the variables, each clause results in a pair of strings, P'_i and T'_i. These are concatenated as gadgets $C_p = P'_1P'_2 \ldots P'_M$ and $C_t = T'_1T'_2 \ldots T'_M$ respectively. Suppose the ith clause is $(v_j \vee v_k \vee v_l)$, with $v_x \in \{a_x, A_x\}$ for $x \in \{j, k, l\}$. We then define $P'_i = \$v_jv_kv_l\$$ and $T'_i = \$0^{2(j+k+l)-1}\$$.

Lemma 1. *The mapping f gives a GPM match for pattern $\$\$V_p\$C_p$ and text $\$\$V_t\$C_t$ iff for all i, $f(\{a_i, A_i\}) = \{0^{2i-1}, 0^{2i}\}$ and $f(C_p) = C_t$.*

Lemma 1 allows us to consider only a restricted set of functions in the proof of our reduction below. The proof of the Lemma is by induction on the set of

mapped strings in a matching function. We will let 0^x represent True iff x is odd. This can be seen as a generalisation of the GFM reduction where 0 represented True and 00 represented False.

Theorem 2. *Generalised parameterised matching is NP-Complete.*

Proof. Given an instance of 1-in-3 SAT ϕ we construct pattern $p = \$\$V_p\$C_p$ and text $t = \$\$V_t\$C_t$ as described above. We show that p GPM matches t iff there exists a 1-in-3 satisfying assignment for ϕ.

We define a function, f_σ for each assignment of truth values, σ. First set $f_\sigma(\$) = \$$. For each i, if variable a_i is True then let $f_\sigma(a_i) = 0^{2i-1}$ and $f_\sigma(A_i) = 0^{2i}$. Otherwise let $f_\sigma(a_i) = 0^{2i}$ and $f_\sigma(A_i) = 0^{2i-1}$. It follows from Lemma 1 that any f which creates a GPM match is equal to f_σ for some σ. Thus we do not need to consider other functions.

Assume that σ satisfies ϕ. Consider the i-th clause, $(v_j \vee v_k \vee v_l)$ and, wlog, assume v_j is True and v_k, v_l are False. Therefore, $f(P'_i)$ equals,

$$f(\$)f_\sigma(v_j)f_\sigma(v_k)f_\sigma(v_l)f(\$) = \$0^{2i-1}0^{2j}0^{2l}\$ = \$0^{2(i+j+l)-1}\$ = T'_i$$

as required. As i was arbitrary, this holds for all clauses so C_P GPM matches C_T under f_σ. Thus, by Lemma 1, p GPM matches t under f_σ.

Conversely, assume that p GPM matches t under some function f_σ. Again, consider the i-th clause, $(v_j \vee v_k \vee v_l)$. As $f_\sigma(\$) = \$$, P'_i GPM matches T'_i. Assume for a contradiction that all three literals are False,

$$f_\sigma(v_j) = 0^{2j}, f_\sigma(v_j) = 0^{2k} \text{ and } f_\sigma(v_j) = 0^{2l}$$

$$f_\sigma(P'_i) = \$f_\sigma(v_j)f_\sigma(v_k)f_\sigma(v_l)\$ = \$0^{2(j+k+l)}\$ \neq T'_i\ .$$

Similarly, if one or zero literals are False then $f_\sigma(T_i) = 0^{2(j+k+l)-2}$ or $f_\sigma(T_i) = 0^{2(j+k+l)-3}$ respectively. Therefore, exactly one literal is True and the clause is 1-in-3 satisfied. Again, as i was arbitrary, σ satisfies ϕ.

We might now ask if the GPM problem can be restricted in the same way as GFM in Corollary 1:

Corollary 2. *GPM remains NP-Complete when Σ_t contains only two distinct symbols or when we restrict f so that $|f(x)| \leq 2$ for all $x \in \Sigma_p$ (but not both).*

4 Generalised Pattern Matching with String Classes

We now discuss a variant of generalised function matching which does permit a polynomial-time solution. The input is specified by a pattern, p, a text, t, and a set of string classes defined by a function C from characters in the pattern to sets of strings over the text alphabet. Pattern p matches text t if and only if t can be written as $s_1 s_2 \ldots s_m$ where $s_i \in C(p_i)$. As we will see, an important feature of the problem definition is that different occurrences of the same character in the pattern can match to different substrings if the substrings are in the same class. The problem can be seen as a generalisation of the problem known as *pattern matching with character classes* [5].

Example 1. If the text is $t = banana$, the pattern is $p = ABA$ and the classes are $C(A) = \{ban, ba, n, an, na\}$ and $C(B) = \{anan, na, b, a\}$, then we say that p matches t. A possible matching is a mapping $A_1 \to ba$, $B \to na$ and $A_2 \to na$ where A_i denotes the ith A in the pattern.

We present an $O(nmk \log n)$ solution to this problem, where k is the length of the longest string from any class. The solution is based on dynamic programming. Let $d(i, j)$ be 1 if the first j characters of p match the first i characters of t and 0 otherwise. If $d(n, m) = 1$ then p matches t. The recurrence formula is:

$$d(i, j) = 1 \text{ iff } \exists 0 \leq \ell < i \text{ s.t. } d(\ell, j - 1) = 1 \text{ and } t[\ell + 1..i] \in C(p_j),$$

where $t[\ell + 1..i] = t_{\ell+1}t_{\ell+2} \dots t_i$ and $C(p_j)$ is the character class of p_j. We define $d(0, 0)$ to be 1. We then calculate all $d(i, j)$ values using generalised suffix trees for the classes which have been precomputed in a preprocessing stage.

Theorem 3. *Assume k is the length of the longest substring in any of the classes and $|C|$ is the total length of all the substrings classes. The pattern matching problem with string classes can be solved in $O(nmk \log n)$ time with $O(|C| \log n)$ preprocessing of the pattern.*

5 An Approximation Algorithm for GFM

In this Section we will introduce an optimisation version of GFM for which we are able to provide a polynomial-time approximation algorithm. Our motivation is to give a measure of how close two strings are to having a GFM match.

Hamming similarity. We define the Hamming similarity between two strings of the same length to be the number of positions in which the two strings are equal. For input text t and pattern p, we are interested in the maximum Hamming similarity between p and any string p' of the same length which has a GFM match with t. As the original GFM problem is NP-Complete, this optimisation problem is NP-Hard.

When changing a symbol at a position in the pattern, we can choose a character that does not occur elsewhere. The new unique character will therefore be permitted to match any non-empty substring of the text. We write such unique characters using the wildcard symbol '$*$' in order to emphasise their role in any matching.

Example 2. If $p = aba$ and $t = xyyz$, then we can keep at most two positions in the pattern unchanged in order to have a GFM match. Hence the Hamming similarity is 2. One option is modify the pattern so that $p = ab*$.

We present a polynomial-time approximation algorithm that achieves a $\min(1, \sqrt{k/\mathrm{OPT}})$ approximation ratio for the Hamming similarity problem, for fixed k. The algorithm is as follows.

1. Select all $S \subset \Sigma_p$ with $|S| = k$. There are $\binom{|\Sigma_p|}{k} \leq m^k$ such S.
   (a) For each $i = 1, \ldots, m$, set p_i^S to p_i if $p_i \in S$ or to the wildcard symbol, otherwise.
   (b) Let H_S denote the Hamming similarity between p^S and t.
2. Let $M = \max_S H_S$.
3. Output $\max(M, |\Sigma_p|)$.

We claim that that $\text{OPT} \geq \max(M, |\Sigma_p|) \geq \min(\text{OPT}, \sqrt{k \cdot \text{OPT}}) = \text{OPT} \cdot \min(1, \sqrt{k/\text{OPT}})$. In other words,

Lemma 1. *The algorithm given above achieves an approximation ratio of* $\min(1, \sqrt{k/\text{OPT}})$ *for the Hamming GFM similarity problem.*

Proof. We know that $\text{OPT} \leq M|\Sigma_p|/k$ as M is the maximum for any set of k characters. Therefore $k\text{OPT} \leq M|\Sigma_p|$. It follows that either M or $|\Sigma_p|$ is $\geq \sqrt{k\text{OPT}}$. Therefore, the approximation ratio follows immediately as $\sqrt{k\text{OPT}}/\text{OPT} = \sqrt{k/\text{OPT}}$.

We now describe how to perform the first step of the algorithm in polynomial time. We choose all subsets of k characters from Σ_p and all subsets of k substrings of t and solve the problem for each independently. We fix for the moment the set of characters to be $a_1, \ldots a_k$ and the set of substrings that these are mapped to be $s_1, \ldots, s_k$.

The solution is based on dynamic programming. We define the function $f(i, j)$ to be the best solution to the problem for $t_1 t_2 \ldots t_j$ and $p_1 p_2 \ldots p_i$. We define $f(0, i) = 0 \; \forall i$ and $f(i, j) = 0 \; \forall i > j$. The Hamming GFM similarity of the entire string is $f(m, n)$. We now show how we compute $f(i, j)$ when $i \leq j$.

1. If $p_i \notin \{a_1, \ldots, a_k\}$,then
   $f(i, j) = \max\{f(i-1, j-1), f(i-1, j-2), \ldots, f(i-1, i-1)\}$.
2. If $\exists z$ s.t. $p_i = a_z$, then
   $f(i, j) = \max_k\{f(i-1, j-k) + I(t_{j-k+1} \ldots t_j = s_z)\}$,
   where I is the indicator function.

Theorem 4. *For any k there is an algorithm that runs in time $n^{O(k)}$ and achieves an approximation ratio of* $\min(1, \sqrt{k/\text{OPT}})$ *for the Hamming GFM similarity problem.*

We must first show that the dynamic programming procedure computes the right function and then that it runs in polynomial time. We can see immediately that $f(0, i) = 0 \; \forall i$ because in this case the pattern is empty. Also, $f(i, j) = 0 \; \forall i > j$ because every character of the pattern must map at least one character from the text, even if it is replaced by a wildcard. The computation of $f(i, j)$ has two cases.

- $p_i \notin \{a_1, \ldots, a_k\}$. In this case we cannot increase the number of characters in our set that can be mapped. However we know that p_i will be set to a wildcard and therefore we find the maximum of the previous results for different length substrings that the wildcard maps to.

– $\exists z$ s.t $p_i = a_z$. We can either map p_i to s_z and increase the number of mapped characters by one, which can only happen if $t_{j-|s_z|+1} \dots t_j = s_z$ or we do the same as in the previous case.

The running time of the approximation algorithm is polynomial in n and m for constant k, since there are $\binom{|\Sigma_p|}{k} \leq \binom{m}{k}$ ways of selecting k characters of Σ_p and at most $\binom{n^2}{k}k!$ ways of choosing k substrings of t.

Acknowledgements

We thank Igor Nor and Amihood Amir for their very helpful contributions to our understanding of the general problem area and Tom Hinton for advice regarding approximation algorithms. The first and third authors also thank the EPSRC for their support.

References

1. Amir, A., Aumann, A., Cole, R., Lewenstein, M., Porat, E.: Function matching: Algorithms, applications, and a lower bound. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 929–942. Springer, Heidelberg (2003)
2. Amir, A., Nor, I.: Generalized function matching. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 41–52. Springer, Heidelberg (2004)
3. Amir, A., Nor, I.: Generalized function matching. Journal of Discrete Algorithms 5(3), 514–523 (2007)
4. Baker, B.S.: A theory of parameterized pattern matching: algorithms and applications. In: Proceedings of the 25*th* Annual ACM Symposium on the Theory of Computing (STOC), pp. 71–80 (1993)
5. Linhart, C., Shamir, R.: Faster pattern matching with character classes using prime number encoding. Journal of Computer and System Sciences 75(3), 155–162 (2009)
6. Schaefer, T.J.: The complexity of satisfiability problems. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC), pp. 216–226. ACM Press, New York (1978)

Practical Algorithms for the Longest Common Extension Problem

Lucian Ilie[⋆,⋆⋆] and Liviu Tinta

Department of Computer Science, University of Western Ontario
N6A 5B7, London, Ontario, Canada
ilie@csd.uwo.ca

Abstract. The *Longest Common Extension* problem considers a string s and computes, for each of a number of pairs (i, j), the longest substring of s that starts at both i and j. It appears as a subproblem in many fundamental string problems and can be solved by linear-time preprocessing of the string that allows (worst-case) constant-time computation for each pair. The two known approaches use powerful algorithms: either constant-time computation of the Lowest Common Ancestor in trees or constant-time computation of Range Minimum Queries (RMQ) in arrays. We show here that, from practical point of view, such complicated approaches are not needed. We give two very simple algorithms for this problem that require no preprocessing. The first needs only the string and is significantly faster than all previous algorithms on the average. The second combines the first with a direct RMQ computation on the Longest Common Prefix array. It takes advantage of the superior speed of the cache memory and is the fastest on virtually all inputs.

1 Introduction

The *longest common extension (LCE)* problem takes as input a string s and many pairs (i, j) and computes, for each pair (i, j), the longest substring of s that occurs both starting at position i and at j in s. That is, the longest common prefix of the suffixes of s that start at positions i and j, respectively. Sometimes the problem receives two strings as input, s and t, and is required to compute, for each pair (i, j), the longest common prefix of the ith suffix of s and jth suffix of t. This reduces to the previous problem by considering the string $s\$t$, where \$ is a letter that does not appear in s and t.

The LCE problem appears as a subproblem in many fundamental string problems, such as k-mismatch problem and k-difference global alignment [14, 18], computation of (exact or approximate) tandem repeats [7, 13, 15], or computing palindromes and matching with wild cards [6]. Very efficient algorithms are obtained and it is not clear how to solve those problems without employing LCE solutions.

The LCE problem can be optimally solved by linear-time preprocessing of the string s so that the answer for each pair (i, j) can be computed in constant

⋆ Research supported in part by NSERC.
⋆⋆ Corresponding author.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 302–309, 2009.
© Springer-Verlag Berlin Heidelberg 2009

time. Two powerful algorithms are employed to achieve this bound. The first is the constant-time computation of the Lowest Common Ancestor in trees (with linear-time preprocessing) [2, 3, 8, 19]. When applied to the suffix tree [6] of the string s, it easily yields the solution for the LCE problem. The second uses constant-time computation of Range Minimum Queries (RMQ) in arrays (with linear-time preprocessing) [2, 3, 5, 17]. Applied to the longest common prefix array (LCP) of s (that is part of the suffix array data structure of s, see Section 2), this gives again a solution of the LCE problem. The RMQ-based solution is more efficient in practice [5].

In this paper we look at the LCE problem from a practical point of view. Our aim is to provide simple and efficient algorithms. As it is often the case, the best worst-case algorithms need not be the fastest in practice. Indeed, already [5] considered a simplified algorithm that resolves each (i, j) pair in $\mathcal{O}(\log n)$ time (with linear-time preprocessing); this algorithm performs the best in practice.

Our starting point is the observation that, on the average, the LCE values are very small. We give the precise limit of this average, for a given alphabet size, when the string length goes to infinity. An important consequence is that the algorithm that directly compares the suffixes starting at positions i and j is optimal on the average and significantly faster in practice, on the average, than all previous ones. It needs only the string s; no preprocessing.

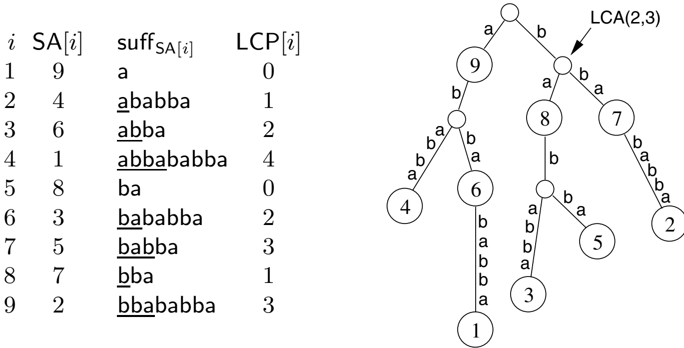
This new algorithm is the fastest for all pairs (i, j) except when the positions corresponding to i and j in the suffix array are very close (as that implies a very long common extension). For those cases we employ an algorithm that uses direct computation of RMQ. The combination of the two, added by the superior speed of the cache memory, produces an algorithm that, while still very simple (no preprocessing required; uses only the existing LCP array), is the fastest on virtually all inputs.

All proofs have been omitted due to lack of space.

2 Basic Definitions

Let A be an alphabet with $\text{card}(A) = \ell \geq 2$. Let $s \in A^*$ be a string of length $|s| = n$. For any $1 \leq i \leq n$, the ith letter of s is $s[i]$ and $s[i\,..\,j] = s[i]s[i+1]\cdots s[j]$. In this notation $s = s[1\,..\,n]$. Let also suf_i denote the suffix $s[i\,..\,n]$ of s. For $1 \leq i \neq j \leq n$, the length of the longest common prefix of the strings suf_i and suf_j is called the *longest common extension* of the two suffixes, denoted by $\mathsf{LCE}_s[i, j]$. When s is understood, it will be omitted.

Assuming a total order on the alphabet A, the *suffix array* of s, [16], denoted SA, gives the suffixes of s sorted ascendingly in lexicographical order, that is, $\mathsf{suf}_{\mathsf{SA}[1]} < \mathsf{suf}_{\mathsf{SA}[2]} < \cdots < \mathsf{suf}_{\mathsf{SA}[n]}$. The suffix array of the string abbababba is shown in the second column of Fig. 1. The suffix array is often used in combination with another array, the longest common prefix (LCP) array that gives the length of the longest common prefix between consecutive suffixes of SA, that is, $\mathsf{LCP}[i] = \mathsf{LCE}[\mathsf{SA}[i-1], \mathsf{SA}[i]]$; see the fourth column of Fig. 1 for an example. By definition, $\mathsf{LCP}[1] = 0$.



i	$\mathsf{SA}[i]$	$\mathsf{suff}_{\mathsf{SA}[i]}$	$\mathsf{LCP}[i]$
1	9	a	0
2	4	ababba	1
3	6	abba	2
4	1	abbababba	4
5	8	ba	0
6	3	bababba	2
7	5	babba	3
8	7	bba	1
9	2	bbababba	3

Fig. 1. The SA and LCP arrays (left) and the suffix tree (right) for the string abbababba. We have $\mathsf{LCE}(2,3) = \mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[3]+1, \mathsf{SA}^{-1}[2]) = \mathsf{RMQ}_{\mathsf{LCP}}(7,9) = 1$; this is also the depth of the node $\mathsf{LCA}(3,2)$ in the suffix tree.

The suffix array of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time by any of the algorithms in [9, 11, 12]. The longest common prefix array can be computed also in $\mathcal{O}(n)$ time by the algorithm of [10].

The *LCE problem* is: given a string s and a set of pairs (i,j), compute $\mathsf{LCE}(i,j)$ for each pair. It can be solved by preprocessing the string s in linear time so that each $\mathsf{LCE}(i,j)$ is computed in constant time. The first solution uses constant-time computation of the Lowest Common Ancestor [2, 3, 8, 19] applied to the suffix tree; see an example in Figure 1. The second, more efficient, uses constant-time computation of Range Minimum Queries (RMQ) in arrays [2, 3, 5, 17] applied to the LCP array. In general, we have $\mathsf{LCE}(i,j) = \mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i]+1, \mathsf{SA}^{-1}[j])$. Note the need for the inverse suffix array SA^{-1}; an example is shown in Figure 1.

We shall denote the LCE algorithm of [5] based on constant-time RMQ computation by $\mathrm{RMQ}_{\mathrm{const}}$. The practically most efficient algorithm of [5] computes each $\mathsf{LCE}(i,j)$ in (suboptimal) $\mathcal{O}(\log n)$ time; it will be denoted by $\mathrm{RMQ}_{\log}$.

3 An Average-Case Optimal Algorithm for LCE

The starting point of our approach is the observation that most LCE values are very small. The main result of this section estimates the average value of the LCE over all strings of a given length n, that is,

$$\mathsf{Avg_LCE}(n,\ell) = \frac{1}{\ell^n} \sum_{s \in A^n} \Big(\frac{1}{\binom{n}{2}} \sum_{1 \le i < j \le n} \mathsf{LCE}_s(i,j) \Big)$$

Theorem 1. *(i) For any $\ell \ge 2$, $\lim_{n\to\infty} \mathsf{Avg_LCE}(n,\ell) = \frac{1}{\ell - 1}$.*

(ii) For any $n \ge 2$ and $\ell \ge 2$, $\mathsf{Avg_LCE}(n,\ell) < \frac{1}{\ell - 1}$.

The result in Theorem 1 has an important implication for our purpose, that is, no sophisticated algorithms are necessary for computing LCEs. Direct comparison of the two suffixes requires, on average, only one comparison and therefore our DirectComp algorithm (see Figure 2) is optimal on the average.

DirectComp(s, i, j)

1. $t \leftarrow 0$
2. **while** $\big((j + t \leq n)$ **and** $(s[i + t] = s[j + t])\big)$ **do**
3. $\quad t \leftarrow t + 1$
4. **return** t

Fig. 2. Computing LCE by direct comparison

Table 1. Files from Canterbury (five largest ones), Manzini, and Pizza&Chili corpora and some randomly generated with various sizes and number of letters. The first six columns contain, in order: file source, file name, size (in megabytes), alphabet size, average LCE, maximum LCE. The last three contain the average running times for solving the LCE problem using $\text{RMQ}_{\text{const}}$, RMQ_{log}, and DirectComp, resp., given in microseconds per input pair.

	File	size	alph.	Avg_LCE	max_LCE	$\text{RMQ}_{\text{const}}$	RMQ_{log}	DirectComp
Canterbury	book1	0.7	82	0.0736	104	1.34	1.11	0.07
	kennedy.xls	1	256	0.3946	18	1.37	1.17	0.11
	E.coli	4.4	4	0.3371	2815	1.43	1.12	0.21
	bible.txt	3.9	63	0.0915	551	1.28	1.00	0.21
	world192.txt	2.3	93	0.0693	543	1.41	1.21	0.20
Manzini	chr22.dna[1]	33	4	0.3419	1777	1.46	1.17	0.20
	etext99	100	146	0.0732	286352	1.53	1.20	0.21
	howto	38	197	0.0909	70720	1.51	1.20	0.21
	jdk13c	66	113	0.0444	37334	1.44	1.16	0.22
	rctail96	109	93	0.0692	26597	1.50	1.21	0.22
	rfc	111	120	0.2140	3445	1.50	1.21	0.21
	sprot34.dat	105	66	0.0860	7373	1.49	1.20	0.22
	w3c2	99	256	0.0341	990053	1.50	1.22	0.21
Pizza&Chili	sources	201	230	0.0497	307871	—	—	0.20
	pitches	53	133	0.0420	25178	1.63	1.28	0.20
	proteins	1129	27	0.0625	647051	—	—	0.20
	DNA	385	16	0.3500	1378596	—	—	0.21
	English	2108	239	0.0753	4735603	—	—	0.22
	XML	282	96	0.0538	1084	—	—	0.20
random	rand_100_2	100	2	1.0000	52	1.51	1.23	0.29
	rand_100_4	100	4	0.3333	26	1.52	1.22	0.27
	rand_100_20	100	20	0.0526	11	1.48	1.23	0.28
	rand_1000_2	1000	2	1.0000	55	—	—	0.31
	rand_1000_4	1000	4	0.3333	29	—	—	0.30
	rand_1000_20	1000	20	0.0526	13	—	—	0.30

We tested the DirectComp, $\text{RMQ}_{\text{const}}$, and $\text{RMQ}_{\log}$ algorithm for the text files in the Canterbury[1], Manzini[2], and Pizza&Chili[3] corpora, as well as for some random files we generated. The results are shown in the last three columns of Table 1. All tests were done on a Sun Fire V440 Server, using one UltraSPARC IIIi processor at 1593MHz, 1MB L2 Cache, 4GB RAM, running SunOS 5.10. The programs were compiled using `gcc 3.4.3` with options `-O3 -fomit-frame-pointer`. One million random (i, j) pairs were generated and all three algorithms were run on those. Each experiment was repeated three times and the average times are shown. The preprocessing times for $\text{RMQ}_{\text{const}}$ and $\text{RMQ}_{\log}$ were not counted.

In general, our algorithm is roughly five times faster than $\text{RMQ}_{\log}$, the previous fastest algorithm. (Our comparison between $\text{RMQ}_{\text{const}}$ and $\text{RMQ}_{\log}$ gives results similar to [5].) Due to the additional space needed (for a file of size n, more than $24n$ bytes are needed), the RMQ-based algorithms could not handle files large than 160 MB (see also Table 2).

Table 2. Preprocessing and memory requirements for a file of size n; we assume an integer is represented on 4 bytes

Algorithm	$\text{RMQ}_{\text{const}}$	$\text{RMQ}_{\log}$	DirectMin	DirectComp
Preprocessing	RMQ data structures, SA^{-1}, LCP		SA^{-1}, LCP	—
Memory (bytes)	$24n+$		$8n$	n

4 The Worst Case

As seen in the previous section, our DirectComp algorithm performs significantly better than the best ones to date on the average. However, when counting the expected number of operations performed by each algorithm, the difference should be even bigger. That is due to the lower speed of RAM compared to cache memory. Most of the time is spent on accessing the large arrays. We turn this property into our advantage by trying to do better not only on the average but also in the worst case.

In this section we give first a number of results that help us get an idea of how large the maximum LCE is expected to be as well as an estimate on how many "large" LCE values are expected. Denoting $\mathsf{max_LCE}(s) = \max_{i,j} \mathsf{lcp}_s(i, j)$, we have the following theorem:

Theorem 2. *For any $n \geq 2$ and $\ell \geq 2$, we have*

(i) For any $s \in A^n$, $\mathsf{max_LCE}(s) > \log_\ell(n) - 2$.
(ii) There exists an $s \in A^n$ such that $\mathsf{max_LCE}(s) < \log_\ell(n)$.

[1] `http://corpus.canterbury.ac.nz/`
[2] `http://web.unipmn.it/ manzini/lightweight/corpus/`
[3] `http://pizzachili.dcc.uchile.cl/`

(iii) The average maximum LCE*,* $\mathsf{Avg_max_LCE}(n, \ell)$*, satisfies*

$$\log_\ell(n) - 2 \le \mathsf{Avg_max_LCE}(n, \ell) \le 2\log_\ell(n) \ .$$

(iv) The average number of pairs (i, j) *with* $\mathsf{LCE}(i, j) \ge \log_\ell(n)$ *is less than* $n/2$*.*

(v) The average number of pairs (i, j) *with* $\mathsf{LCE}(i, j) \ge 2\log_\ell(n)$ *is less than* $1/2$*.*

The implications of these results are that most LCE values are expected to be small and therefore our DirectComp algorithm performs better for most pairs. For the remaining few, we look for a different solution in the sequel. The maximum LCE can be much larger than expected (see the sixth column of Table 1) but our solution avoids the large LCE values.

The RMQ-based algorithms are better for a very small fraction of the input (i, j) pairs, namely those for which the difference between $\mathsf{SA}^{-1}[i]$ and $\mathsf{SA}^{-1}[j]$ is very small, as that usually implies large $\mathsf{LCE}[i, j]$ value. But, for such cases, there is another, very simple, algorithm, already considered by [5], that performs the best. It requires no preprocessing. Instead, it computes directly the minimum of the values $\mathsf{LCP}[\mathsf{SA}^{-1}[i] + 1 \mathinner{..} \mathsf{SA}^{-1}[j]]$. This algorithm, called DirectMin, is described below.

DirectMin(LCP, i, j)

1. $\mathit{low} \leftarrow \min(\mathsf{SA}^{-1}[i], \mathsf{SA}^{-1}[j])$
2. $\mathit{high} \leftarrow \max(\mathsf{SA}^{-1}[i], \mathsf{SA}^{-1}[j])$
3. $t \leftarrow \mathsf{LCP}[\mathit{low} + 1]$
4. **for** k **from** $\mathit{low} + 2$ **to** high **do**
5.   **if** $\mathsf{LCP}[k] < t$ **then** $t \leftarrow \mathsf{LCP}[k]$
6. **return** t

Fig. 3. Direct computation of the range minimum

Table 2 contains a summary of the memory and preprocessing requirements for each of the four algorithms: $\mathrm{RMQ}_{\mathrm{const}}$, $\mathrm{RMQ}_{\log}$, DirectMin, and DirectComp. The first two require the SA^{-1} array to compute the corresponding positions in the LCP array and the data structures necessary for the constant-(logarithmic-, resp.) time computation of the RMQ values. DirectMin requires SA^{-1} and LCP for the same reason but no additional space. DirectComps needs only the text.

We tested the performance of all four algorithms discussed for the files in Table 1. We run them on pairs at a given distance, $\texttt{step} = |\,\mathsf{SA}^{-1}[j] - \mathsf{SA}^{-1}[i]|$, in the suffix array, represented on the abscissa in logarithmic scale; the ordinate gives the time in microseconds. All pairs at a given distance have been considered for each computation. The results are given in Figure 4.

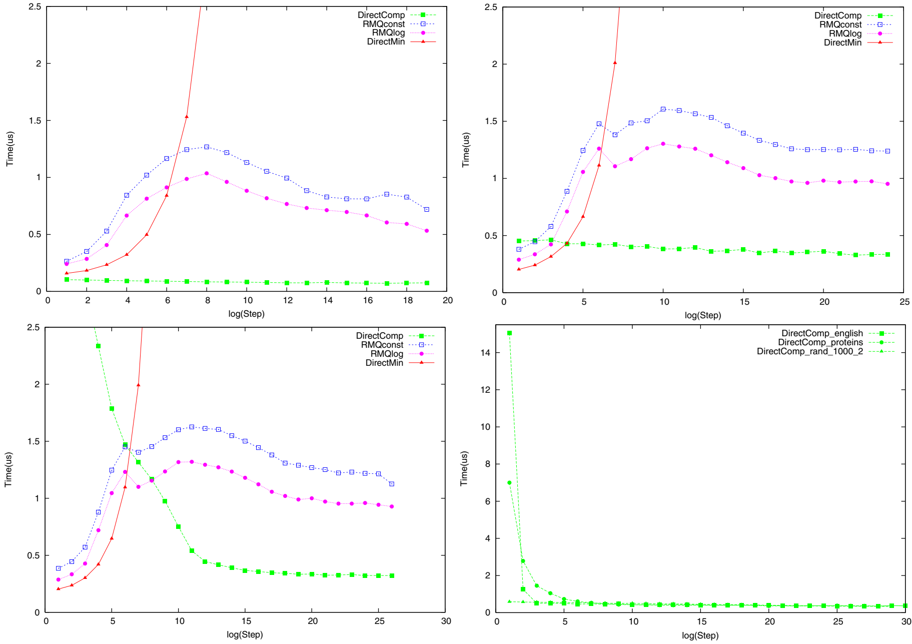


Fig. 4. In row-wise order, the first three plots give the behavior of the four algorithms we discuss, DirectMin, DirectComp, $\text{RMQ}_{\text{const}}$, and RMQ_{log}, on the files `book1`, `chr22.dna`, and `jdk13c`, resp. For files of size less than 1MB, DirectComp is the best on all inputs. The behavior of the four algorithms for the files from Canterbury corpus as well as for the smaller random files is in-between `book1` and `chr22.dna`; for the files in Manzini corpus and `pitches` from Pizza&Chile the behavior is in-between `chr22.dna` and `jdk13c`. The file `jdk13c` is the only one where the combination DirectComp-DirectMin is slightly slower than RMQ_{log} on a very small interval. The last plot gives the behavior of DirectComp on `rand_1000_2`, `English` and `proteins`; it is the only algorithm that can handle those. The performance is impressive; only at distance 2 some of the times are higher; such a case would be very easily handled by DirectMin given enough space for the SA^{-1} and LCP arrays.

5 Conclusions

We gave very simple algorithms for the LCE problem that are the best in practice with respect to both time and space. When the pairs are randomly distributed, DirectComp should be used. If the performance on every single input matters, then the combination DirectComp-DirectMin should be used. Only DirectComp can handle very large files and the performance on those is very good.

Further applications of our approach are to be investigated. For instance, our algorithms might be used to reduce the space required by the pattern search algorithm based on the Enhanced Suffix Array [1]. This is theoretically the most efficient algorithm and has good performance in practice.

Acknowledgements. Lucian Ilie would like to thank Maxime Crochemore for useful discussions. The statistics of the large files were computed on SHARCNET (`www.sharcnet.ca`).

References

1. Abouelhoda, M.I., Kurtz, S., Ohlenbusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms 2, 53–86 (2004)
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. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. 22, 221–242 (1993)
4. de Bruijn, N.G.: A combinatorial problem. Nederl. Akad. Wetensch. Proc. 49, 758–764 (1946)
5. Fischer, J., Heun, V.: Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
6. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
7. Gusfield, D., Stoye, J.: Linear time algorithm for finding and representing all tandem repeats in a string. J. Comput. Syst. Sci. 69, 525–546 (2004)
8. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13, 338–355 (1984)
9. 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)
10. Kasai, T., Lee, G., 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)
11. 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)
12. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. Discrete Algorithms 3(2-4), 143–156 (2005)
13. Landau, G., Schmidt, J.P., Sokol, D.: An algorithm for approximate tandem repeats. J. Comput. Biol. 8, 1–18 (2001)
14. Landau, G., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: Proc. of STOC, pp. 220–230. ACM Press, New York (1986)
15. Main, M., Lorentz, R.J.: An O(n log n) algorithm for finding all repetitions in a string. J. Algorithms 5, 422–432 (1984)
16. Manber, U., Myers, G.: Suffix arrays: a new method for on-line search. SIAM J. Comput. 22(5), 935–948 (1993)
17. de Castro Miranda, R., Ayala-Rincón, M.: A Modification of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays. In: Setubal, J.C., Verjovski-Almeida, S. (eds.) BSB 2005. LNCS (LNBI), vol. 3594, pp. 210–213. Springer, Heidelberg (2005)
18. Myers, G.: An O(nd) difference algorithm and its variations. Algorithmica 1, 251–266 (1986)
19. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17, 1253–1262 (1988)

A Last-Resort Semantic Cache for Web Queries

Flavio Ferrarotti[1,2], Mauricio Marin[1], and Marcelo Mendoza[1]

[1] Yahoo! Research Latin America, Santiago of Chile
[2] University of Santiago of Chile

Abstract. We propose a method to evaluate queries using a last-resort semantic cache in a distributed Web search engine. The cache stores a group of frequent queries and for each of these queries it keeps minimal data, that is, the list of machines that produced their answers. The method for evaluating the queries uses the inverse frequency of the terms in the queries stored in the cache (`Idf`) to determine when the results recovered from the cache are a good approximation to the exact answer set. Experiments show that the method is effective and efficient.

1 Introduction

Current Web search engines are built upon the concept of *search nodes* in which each node holds a fraction of the document collection and contains an inverted file that allows fast determination of the documents that best match a given query. First a broker machine receives the query and broadcasts it to all search nodes and then it receives their local top-k documents to produce the top-k documents that globally best match the query. This scheme is convenient for maintenance and indexing purposes. However, every single query hits the whole set of search nodes which can degrade query throughput and limit scalability.

A number of caching strategies have been developed to reduce the average number of nodes involved in the solution of queries. On the broker side we can have a query answer cache or result cache that prevents sending the most frequent queries to the search nodes. For each query, this cache stores the complete answer presented to the user. On the nodes side, we can have a cache of inverted lists (which reduces secondary memory activity) and/or a cache of pre-computed results such as document scores or intersection of inverted lists (which reduces the running time cost of queries).

Another (complementary) way of reducing the number of nodes hit by a given query is to send the query to a selected group of nodes. To achieve this it is necessary to perform document clustering and evenly distribute those clusters of documents onto the search nodes. For the sake of practicality, the clusters can be built from a subset of the whole document collection whilst the remaining documents are distributed at random. The subset can be taken from a large query log and queries themselves can be used to correlate documents during the clustering process. In addition, a certain representation of the nodes contents is required to determine the target group of nodes where to send each query. However, now those nodes can only deliver an approximation to the exact answer

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 310–321, 2009.
© Springer-Verlag Berlin Heidelberg 2009

of queries which can be acceptable when the search engine is operating at a very high traffic of queries. In this paper we put ourselves in such a scenario and the state of the art work in this case is the one presented in [14,15], which is based on document clustering and representation of node contents.

Differently to [15] we work on the idea of keeping at the broker machine a compact cache that we call *last-resort cache* (LCache). Our method has the advantage of both being able to reduce the number of nodes and yet providing exact query results, and delivering good approximated results in cases of high traffic of queries. Both strategies in combination are expected to improve overall query throughput at the expense of a modest increase in memory requirements on the broker machine. Nevertheless our method can be accommodated in the extra space demanded by [15] with the advantage that the computations involved in the determination of candidate nodes are less demanding.

The LCache is meant to contain the most minimal data about the answer of each cached query, namely the list of search nodes IDs from which each document in the global top-k results come from, and it can be administered with any existing caching policy [16]. Notice that if we distribute related document clusters into the same nodes and the remaining ones (sorted by centers similarity distance) in contiguous-ID nodes, then the lists of nodes stored in the LCache are highly compressible. Moreover the length of the list of node IDs is expected to be smaller than k. Notice that the LCache can also work in a setting in which documents are distributed at random onto the search nodes. Here small memory space is still feasible since for large scale systems the number of nodes is expected to be larger than the number of top results for queries. In other words, an entry in the LCache requires much less space than a corresponding entry in the result cache.

The LCache works in tandem with the result cache kept at the broker side and it has the following two uses. The first one is as a scheme able to further improve query throughput whilst it still allows the broker to provide *exact* answers to queries. This can be made in at least two alternative ways. Firstly, as argued in [16], storing a query in the result cache should consider the cost of computing its answer upon the search nodes since the main objective of the result cache is improving query throughput. Queries requiring a comparatively less amount of computing time can be prevented from being stored in the result cache. We suggest using the LCache for storing those queries which can lead to a potential gain in throughput as they can now be sent directly to the nodes capable of providing the global top-k results.

Secondly, independent of the kind of queries stored in the LCache, their computing time can be further reduced at the cost of more memory per LCache entry by storing k pairs (nodeID, docID) corresponding to the global top results of the respective query. The idea is to use each LCache entry to go directly to the node and retrieve the specific document in order to get the snippet and other data required to construct the answer presented to the user. This only requires secondary memory operations and no documents ranking is necessary. The pairs (nodeID, docID) are also highly compressible since we can re-numerate the

document IDs at each node following the order given by the document clusters stored in them. In any case, the source node IDs are necessary to support the semantic part of the LCache discussed next and developed in the remaining sections of this paper (the study of the trade-offs for the alternative uses of the LCache is out of scope in this paper and will be developed in a future work.) Notice that the list of node IDs corresponding to the query answers stored in the result cache should be also kept in the LCache to improve its semantic properties.

The second use of the LCache is as a *semantic cache* that can reduce the number of search nodes involved in the determination of approximated answers to queries not found in either cache at the broker side. This is useful when the search engine is under high query traffic and less hardware resources are required to be assigned to the processing of individual queries so as to avoid throughput degradation. In this case we send the query to the nodes referenced by LCache entries that share query terms. In this paper we propose a method to achieve this at a reasonable recall. The reader is referred to [15] for an efficient method to deal with high query traffic situations.

A semantic cache allows the search engine to respond to new queries using the answers from previous queries stored in the cache. To do this, lists of answers from queries similar to the new query are utilized, elaborating a list of similar answers (search nodes in our case). Generally, the similarity between the queries is measured based on the quantity of terms that both queries have in common. Despite the fact that the list of answers from a semantic cache is not exact, in many cases it is precise enough to be presented to the user. Our method uses the inverse frequency of the terms stored in the cache to estimate whether the approximated answer covers a significant quantity of relevant documents. If not, the query is broadcast to all search nodes for full evaluation.

The rest of the paper is organized as follows. In Section 2 we review related work. In Section 3 we introduce the strategy of evaluation of queries using the last-resort semantic cache. In Section 4 we show experimental results. Finally, we conclude in Section 5.

2 Related Work

In the method proposed in [14,15], a large query log is used to form P clusters of documents (one per search node) and Q clusters of queries by using the co-clustering algorithm proposed in [5]. This defines a matrix where each entry $V_{c,d}$ contains a measure of how pertinent the query cluster c is to the document cluster d. In addition, for each query cluster c a text file containing all the terms found in the queries of cluster c is maintained. Upon reception of a query q the broker computes how pertinent the query q is to the Q query clusters by using the BM25 cosine similarity method. These values $V_{q,c}$ are used in combination with the matrix entries $V_{c,d}$ to compute a ranking of document clusters so that the first one is the most likely to contain an important fraction of the exact answer to the query q. The second one further improves the approximation and so on.

The basic assumption is that under low query traffic the exact answer to a query q is obtained by sending q to all nodes. However, when the query traffic is high enough, this may not be feasible since it can overload the system, so their strategy consists of using the above values $V_{q,c}$ and $V_{c,d}$ to determine which of the P nodes can provide the best approximated answer to the query q and send q to this node only. The results from this node are then cached at the broker side and passed back to the user. Afterwards when query traffic is restored to normal, the approximated answers kept in the cache can be further improved until they are the exact ones by sending the associated query to the remaining nodes one by one, in descending order of the ranking of document clusters. During that time interval the broker responds users with the current approximated answers stored in the cache.

Regarding caching strategies, one of the first ideas studied in literature was having a static cache of results which stores queries identified as frequent from an analysis of a query log file. Markatos *et al.* [13] showed that the performance of the static caches is generally poor mainly due to the fact that the majority of the queries put into a search engine are not the frequent ones and therefore, the static cache reaches a low number of hits. In this sense, dynamic caching techniques based on replacement policies like LRU or LFU achieved a better performance. Along the same lines, Lempel and Moran [11] calculate a score for the queries that allows for an estimation of how probable it is that a query will be made in the future, a technique called Probability Driven Caching (PDC). Lately, Fagni *et al.* [6] have proposed a structure for caching where they maintain a static collection and a dynamic one, achieving good results, called Static-Dynamic Caching (SDC). In SDC, the static part stores frequent queries and the dynamic part handles replacement techniques like LRU or LFU. With this, SDC achieved a hit ratio higher than 30% in experiments conducted on a log from Altavista. Long and Suel [12] have shown that upon storing pairs of frequent terms determined by the co-occurrence in the query logs, it is possible to better increase the hit ratio. For those, the authors proposed putting the pairs of frequent terms at an intermediate level of caching in between the broker cache and the end-server caches. Baeza-Yates *et al.* [2] have shown that caching posting lists is also possible, obtaining higher hit rates than when just doing results and/or terms. Recently, Gan and Suel [16] have studied weighted result caching in which the cost of processing queries is considered at the time of deciding the admission of given query to the result cache. They propose eviction policies which are consequent with that additional feature.

Godfrey and Gryz [8] provide a framework for identifying the distinct cases that are presented to evaluate queries in semantic caches. Fundamentally, they identify cases that they call semantic overlap, in those which it is possible for the probe query to obtain a list of answers from the cache with a good recall. The semantic caching techniques have been tested with partial success in relational databases [9], due to some cases of semantic overlap where it is possible to find expressions in SQL for probe queries and remainder queries. However, cases also exist where the problem is algorithmically intractable.

In the context of web search engines, the queries correspond fundamentally to a conjunction of terms, for which the problem is algorithmically tractable. This reduces the problem to *set containment* of the number of objects in the cache that could satisfy some condition of semantic overlap being verifiable in linear time. In this context, Chidlovskii *et al.* [4] have proposed evaluation methods for web queries using semantic caches. The proposed methods store clusters of co-occurring answers identified as regions, each of these associated to a signature. New frequent queries are also associated to signatures, where regions with similar signatures are able to develop their answer. The author's measured recall and value of false positives (regions identified as similar but that do not provide results relevant to the queries) denoted as the FP-rate, with differing rates of success depending on the type of case analyzed. In some sense this is the closest work in spirit to our paper. Our method differs on the use of a compact last-resort cache which stores search nodes instead of actual results. To deal with this kind of cache, we have to adapt the known evaluation methods for semantic cache to this new setting. Among others things, given a query q to be evaluated by the semantic cache, we use the `Idf` of the terms in the cache to decide which queries are relevant to q. Up to our knowledge, this is the first time that the concept of `Idf` is used in semantic caching.

Amiri *et al.* [1] deal with the problem of the scalability of the semantic caching methods for the query containment case. These are queries whose results are stored in regions of the cache that also recommend non-relevant results, raising the FP-rate. The authors are able to reduce the FP-rate for query containment introducing a data structure called Merged Aggregated Predicates (MAP) that stores conjunctions of frequent terms which are each directed towards the regions of the semantic cache that contain them. In this way, a new query is searched first in the MAP and later directed to the cache. The main disadvantage of this method is that it uses additional space to store the MAP structure and in the overhead introduced to maintain it updated. Recently, Falchi *et al.* [7] has proposed the use of caching techniques based on the similarity of queries from the perspective of metric spaces. The authors propose the evaluation of a similarity function between queries to determine the regions of the cache relevant to an evaluated query. Conducting experiments on a Content Based Image Retrieval system (CBIR), they were able to attain a hit ratio of 20% reducing the average cost of processing queries by 20%.

3 The Semantic Caching Strategy

The aim of this paper is, given a query which is not present in the LCache but shares some of its terms with terms of queries in the cache, to identify a small subset of machines that produce most of the top-k results for the query. To do this, we need to distribute the document collection over an array of search nodes by means of a document clustering algorithm as we will see in the experiments section.

A last-resort cache must satisfy two fundamental constraints: the evaluation of the queries in the cache must be light in terms of computation and in terms of

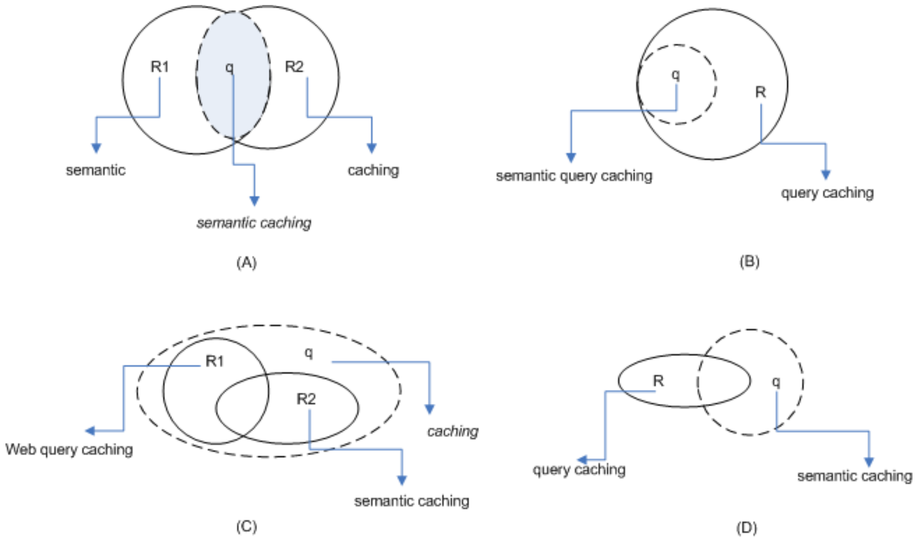


Fig. 1. Semantic overlap cases. (A) Exact query containment, (B), Approximated query containment, (C) Region containment, and (D) N-terms difference.

the space needed to store each cache entry. This means, among other things, that we must only use information available in the cache or that we can eventually add to the cache without considerably increasing its size.

Following Godfrey y Gryz [8], given a conjunctive query q to be evaluated in the cache, we will call a situation that produces a non-empty intersection between one of the cache machine lists and the machine list that allows an exact answer for q to be calculated, a *semantic overlap*.

There are four fundamental types of semantic overlaps [3]. A) *Exact query containment*: The answer to q lies in the intersection of the answers to two or more queries in the cache. B) *Approximated query containment*: There is a query in the cache whose answer strictly includes the answer to q. C) *Region containment*: The answer to q is a superset of the union of the answer to one or more queries in the cache. D) *N-terms difference*: There is a query in the cache whose answer has a nonempty intersection with the answer to q. Figure 1 describes these four cases of semantic overlap.

Note that, for case (A) it could seem at first sight that we could calculate the exact list of machines needed to retrieve the top-k results for q by simply taking the intersection of the list of machines of the relevant queries from the cache. However, since we only have the machines needed to retrieve the top-k results for those queries, it may actually happen that the set of machines in the intersection of their corresponding lists, is not the exact set of machines needed to retrieve all the top-k results for q. A similar observation also holds for case (B). For instance, the list of machines which has the top k-results for a query in *query caching* in Figure 1 may not include all machines needed to retrieve the top-k results for a query in *semantic query caching*.

On the other hand, the fact that with our method of semantic caching we have to find only the machines that potentially have the top-k results for q, instead of the actual top-k documents, considerably increases our chances of success. Indeed, as shown in our experiments we are able to recover most of the top-k results even for cases (C) and (D) in which the set of actual answers for the relevant queries in the cache does not cover the result set for q.

The cases of semantic overlap can be detected by using computationally cheap syntactic comparisons of query terms. Since we are dealing with conjunctive queries, we will consider a query as a set of terms. Let q be the conjunctive query that we want to evaluate, let C be the set of queries in the cache, and let $C|_q = \{q_i \in C | q_i \cap q \neq \emptyset\}$, i.e., $C|_q$ is the set of queries in the cache that overlap with q. We determine which cases of semantic-overlap apply to q as follows:

- Case (A): There is an $S \subseteq C|_q$ such that $|S| \geq 2$ and $q = \bigcup_{q_i \in S} q_i$.
- Case (B): There is a $q_i \in C|_q$ such that $q_i \subset q$.
- Case (C): There is an $S \subseteq C|_q$ such that $q \subset \bigcup_{q_i \in S} q_i$.
- Case (D): There is a $q_i \in C|_q$ such that $q \cap q_i$, $q_i \setminus q$ and $q \setminus q_i$ are not empty.

Note that a given query q to be evaluated in the cache may be a match for more than one of the four semantic-overlap cases. In fact, it could be a match with all four cases. In such situations, we process the query following the strategy associated with the case which has the highest precedence. In our approach the precedence order of cases is A (highest), B, C and D (lowest).

In more detail, given a query q which is not found in the LCache, our semantic caching algorithm proceeds as follows. First, it determines if there is a case of semantic-overlap that applies to q. If not, q is sent to all machines. Otherwise, the algorithm chooses the highest precedence case that applies to q. For each case the algorithm applies a different course of action.

- Case (A) –Exact query containment case– Only the machines which are in the intersection of the sets of machines corresponding to the relevant queries form the cache (i.e., corresponding to the queries in S), are visited.
- Case (B) –Approximated query containment case– The algorithm visits the machines listed for the queries in $\{q_i \in C|_q \mid q_i \subset q$ and for every $q_j \in C|_q$ for which $q_j \subset q$, it holds that $|q_i| \geq |q_j|\}$.

For the final two cases we apply a novel idea in this context which consists of using the inverse frequency (`Idf`) of the terms associated with the queries stored in the LCache. This is made to decide if it is useful to send the query just to the subset of machines obtained with the following two cases (we calculate the `Idf` of the terms using Salton's standard formula).

- Case (C) –Region containment case– Visit the machines listed for the queries in $\{q_i \in S \mid$ for every $t_j \in q_i \setminus q$ it holds that $\texttt{Idf}(t_j) < \texttt{threshold}\}$. If this set is empty, then q is sent to all machines.
- Case (D) –N-terms difference case– Visit the machines listed for the queries in $\{q_i \in C|_q \mid q_i \cap q, q_i \setminus q,$ and $q \setminus q_i$ are not empty, and for every $t_j \in q_i \setminus q$ it holds that $\texttt{Idf}(t_j) < \texttt{threshold}\}$. If this set is empty, then q is sent to all machines.

The idea in these two cases is to discriminate which of those queries in the LCache that have semantic overlap with q, are relevant to approximate the list of machines which store the top-k documents for q. Given a query q' from the LCache that shares some terms with q, if the `Idf` of the terms in $q' \setminus q$ is lower than an experimental threshold, that indicates that those terms are very general and thus the ability to discriminate the relevant documents for q is determined by the terms that are shared with q. In such a case we consider that the machines listed for those candidate queries are relevant for the query q.

An important aspect in this scheme is to determine the `Idf` threshold under which a term is considered with little ability to discriminate relevant documents. We determine this threshold in the next section. Note that to store this extra information, we only need to add one bit for each term in the cache. The bit is set to 1 if the `Idf` of the term is below the experimental threshold, otherwise it is set to 0.

4 Experimental Results

In the experiments for this work we used a 10 million documents sample of the Web UK. We used a very large query log containing queries submitted into this domain. From this log we took 100,000 unique and randomly selected queries, and for each one of these queries we calculated the top-100 documents by executing the BM25 ranking method upon the set which results from intersecting the inverted lists associated with each query term. We used this set of queries and results to group the documents into 32 clusters. Each of these clusters was then assigned to a different search nodes. The clustering was done using the standard k-means method as realized in CLUTO [10]. The input for the clustering algorithm was a set of query-vectors. The *query-vector* of a document d is an m-dimensional vector $(a_1, \ldots, a_m)$, where m is the number of available queries and, for $1 \leq j \leq m$, $a_j = 1$ iff the document d is among the top-k candidate documents for the j-th query in the cache. The documents in the collection which

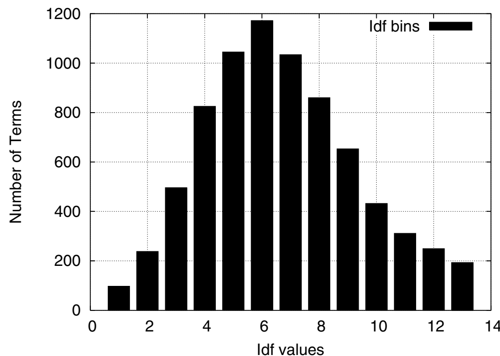


Fig. 2. Distribution of the `Idf`s of the terms in the cache

did not appear among the top-100 documents of any of the 100,000 queries in the log, were distributed evenly among the machines using a random strategy.

From the large query log we determined the frequency of occurrence of the 100,000 unique queries. We initialized the LCache with the top-10,000 most frequent queries. Each entry in the cache stores the set of machines that have the top-20 results for the corresponding query. That is, we set the *exact answer* to a query q to be the top-20 results for q. We found that $66,810$ queries out of the $90,000$ remaining queries not included in the cache, matched at least one of the four cases of semantic-overlap defined in the Section 3. This was the set of queries we evaluated in our experiments. In a real setting the remaining $90,000 - 66,810$ queries (25.76%) would be sent to all search nodes.

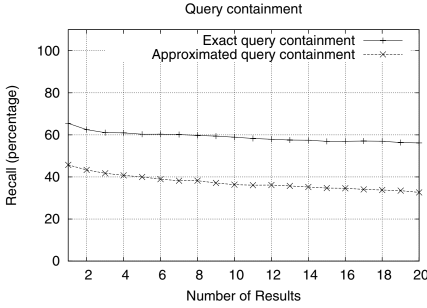
The $10,000$ queries stored in the LCache contain $7,900$ different terms. To evaluate our method we calculated the `Idf` of each of these terms upon the text collection. In Figure 2, we show the distribution of these `Idf` values. Given this distribution, we decided to study the behavior of our method using `Idf` thresholds of $1, 3, 5, 7, 9$, and 11.

The LCache matched 41.09%, 45.40%, 48.88%, 50.57%, 51.35% and 51.54% of the query testing collection, for `Idf` threshold values of $1, 3, 5, 7, 9$ and 11, respectively. A summary of the distribution of the queries in each case is detailed in Table 1.

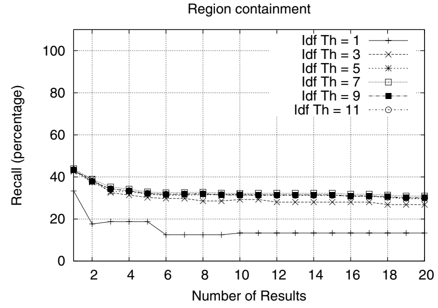
The remaining queries (note that 74.24% of the testing queries share at least one term with the LCache) are evaluated by our method sending them to all the machines. This is due to the effect of the `Idf` threshold filter (see the last row of Table 1).

Table 1. Percentage of queries in each case

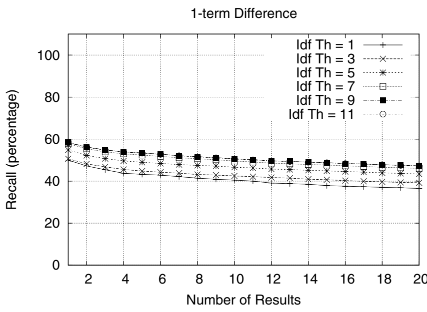
<table>
<tr><td>Case</td><td colspan="6">Percentage</td></tr>
<tr><td>Exact Qu. Cont.</td><td colspan="6">1.74</td></tr>
<tr><td>Empty Intersection</td><td colspan="6">0.17</td></tr>
<tr><td>Total A</td><td colspan="6">1.91</td></tr>
<tr><td>App. Qu. Cont. (1-term diff.)</td><td colspan="6">24.23</td></tr>
<tr><td>App. Qu. Cont. (2-terms diff.)</td><td colspan="6">9.04</td></tr>
<tr><td>App. Qu. Cont. (3-terms diff.)</td><td colspan="6">2.68</td></tr>
<tr><td>App. Qu. Cont. (N-terms diff.)</td><td colspan="6">1.18</td></tr>
<tr><td>Total B</td><td colspan="6">37.14</td></tr>
<tr><td><code>Idf</code> Threshold</td><td>1</td><td>3</td><td>5</td><td>7</td><td>9</td><td>11</td></tr>
<tr><td>Total C</td><td>0.02</td><td>0.09</td><td>0.19</td><td>0.26</td><td>0.27</td><td>0.27</td></tr>
<tr><td>1-term Diff.</td><td>1.17</td><td>3.68</td><td>5.82</td><td>6.87</td><td>7.43</td><td>7.56</td></tr>
<tr><td>2-terms Diff.</td><td>0.64</td><td>1.91</td><td>2.83</td><td>3.28</td><td>3.45</td><td>3.49</td></tr>
<tr><td>3-terms Diff.</td><td>0.17</td><td>0.54</td><td>0.78</td><td>0.87</td><td>0.90</td><td>0.92</td></tr>
<tr><td>N-terms Diff.</td><td>0.04</td><td>0.13</td><td>0.21</td><td>0.24</td><td>0.25</td><td>0.25</td></tr>
<tr><td>Total D</td><td>2.02</td><td>6.26</td><td>9.64</td><td>11.26</td><td>12.03</td><td>12.22</td></tr>
<tr><td>Total LCache</td><td>41.09</td><td>45.40</td><td>48.88</td><td>50.57</td><td>51.35</td><td>51.54</td></tr>
<tr><td>Queries filtered by the method</td><td>33.15</td><td>28.84</td><td>25.36</td><td>23.67</td><td>22.89</td><td>22.70</td></tr>
</table>



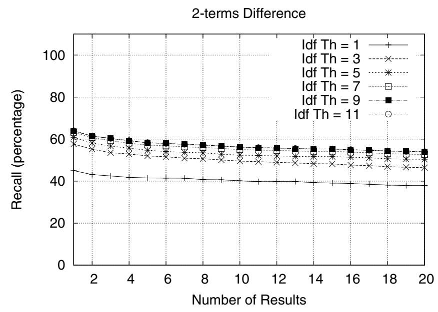
(a)



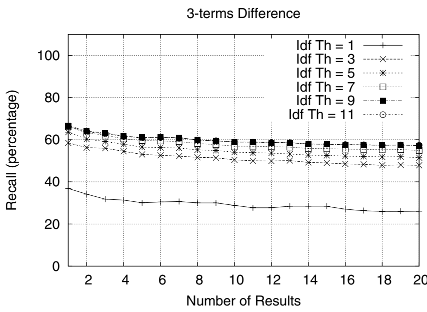
(b)



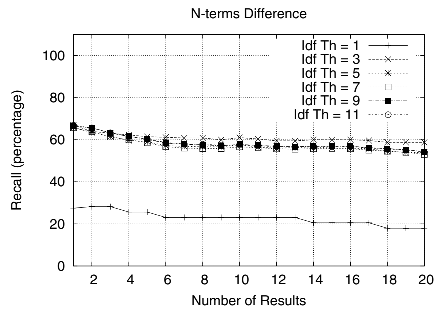
(c)



(d)



(e)



(f)

Fig. 3. Recall curves: (a) Exact/Approx. containment cases, (b) Region containment case, (c)-(e) N-terms difference: (c) $N = 1$, (d) $N = 2$, (e) $N = 3$, and (f) $N \geq 4$

The recall curves for the approximated query containment (case B) and exact query containment (case A) are shown in Figure 3 (a). Note that for the exact query containment case (case A), the average number of top-20 answers for the queries that are found in the search nodes recommended by our method, is 60%,

Table 2. Percentage of visited search nodes for cases (C) and (D)

Idf Threshold	Case C Reg. Cont.	Case D 1term	2terms	3terms	Nterms
1	10.24	14.68	11.78	08.60	7.81
3	12.25	16.74	20.27	22.62	23.96
5	12.07	18.88	22.17	24.03	24.79
7	12.55	20.13	23.26	25.72	25.65
9	12.72	20.27	23.37	25.82	25.46
11	12.72	20.41	23.43	25.76	25.16
Oracle	28.91	29.53	28.28	28.54	26.97

approximately. For the case B, the recall is lower. The recall curves for region containment are shown in Figure 3 (b). As expected, higher Idf thresholds give higher recall values. In Figure 3 (c)-(e), we show the recall curves for the N-terms difference case for N $= 1, 2, 3$, and for N ≥ 4. The recall curves show that our semantic caching method is quite effective for this case, recovering in average 60% of the exact answers for threshold values over 3. Higher recall values are achieved for an Idf threshold equals to 9.

For the case A we need to visit only the 24.90% of the search nodes. In average we visit the 11% to solve the case (B). More precisely, the method visit the 11.96%, 11.85%, 12.02% and 12.26% of the search nodes when the query has 1, 2, 3 or more than 3 terms of difference with the query storaged in the LCache.

In Table 2 we show the percentage of search nodes visited for the cases (C) and (D). These results show that when the Idf threshold increases its value, the average number of nodes to be visited also increases. The last row shows the percentage of nodes determined by an optimal method, namely an oracle at the broker side capable of telling us the exact search nodes required for producing the top-20 results for each query.

Our results are very promising. For example, in case (A) we visit only the 24.9% of the search nodes achieving recall values closed to 60%. In case (D), for an Idf threshold equals to 9 we also reach a recall value closed to 60% visiting only the 20% of the search nodes (for $N = 1$). In case (B) the method visits 11% of the machines reaching a recall close to 40%. A similar situation is observed in the case (C).

5 Conclusions

We have proposed a method for reducing the number of search nodes involved in the solution of queries submitted to a Web search engine. The method is based on the use of a compact cache that we have called LCache, which stores the search node IDs that produced the top-k results for previous queries. This cache can be used to produce the top-k results for new queries that are found in the LCache and were not located in the standard result cache kept at the broker machine. This certainly improves query throughput and we have left the evaluation of the feasible improvement for future work.

Instead, the focus of this paper was on how to use the data stored in the LCache under a situation of high query traffic. In this case it is desirable to reduce the visited search nodes even for queries not found in any of the two caches maintained in the broker. To cope with this situation we have proposed a method to use the LCache as a semantic cache which instructs the broker to send queries to the most promising search nodes. The experimental results suggest that the method can reduce the number of visited search nodes per query sharing cache terms, visiting only a small fraction of the search nodes in a range that goes from 11% to 25% achieving recall values between 40% and 60%.

A key issue to be further investigated is how to use the query answer data stored in the result cache to achieve node reduction at a reasonably good recall.

References

1. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: Scalable template-based query containment checking for web semantic caches. In: ICDE (2003)
2. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: Design trade-offs for search engine caching. ACM TWEB 2(4) (2008)
3. Chidlovskii, B., Roncancio, C., Schneider, M.: Semantic Cache Mechanism for Heterogeneous Web Querying. Computer Networks 31(11-16), 1347–1360 (1999)
4. Chidlovskii, B., Borghoff, U.: Semantic Caching of Web Queries. VLDB Journal 9(1), 2–17 (2000)
5. Dhillon, I., Mallela, S., Modha, D.S.: Information-theoretic co-clustering. In: KDD (2003)
6. Fagni, T., Perego, R., Silvestri, F., Orlando, S.: Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. ACM TOIS 24(1), 51–78 (2006)
7. Falchi, F., Lucchese, C., Orlando, S., Perego, R., Rabitti, F.: A Metric Cache for Similarity Search. In: LSDS-IR (2008)
8. Godfrey, P., Gryz, J.: Answering Queries by Semantic Caches. In: Bench-Capon, T.J.M., Soda, G., Tjoa, A.M. (eds.) DEXA 1999. LNCS, vol. 1677, pp. 485–498. Springer, Heidelberg (1999)
9. Jónsson, B., Arinbjarnar, M., THórsson, B., Franklin, M., Srivastava, D.: Performance and overhead of semantic cache management. In: TOIT, vol. 6(3) (2006)
10. Karypis, G.: CLUTO software for clustering high-dimensional datasets, V. 2.1.1, http://glaros.dtc.umn.edu/gkhome/
11. Lempel, R., Moran, S.: Predictive caching and prefetching of query results in search engines. In: WWW (2003)
12. Long, X., Suel, T.: Three-level caching for efficient query processing in large Web search engines. In: WWW (2005)
13. Markatos, E.: On caching search engine query results. Computer Communications 24(7), 137–143 (2000)
14. Puppin, D., Silvestri, F., Perego, R., Baeza-Yates, R.: Load-balancing and caching for collection selection architectures. In: INFOSCALE (2007)
15. Puppin, D., Silvestri, F., Perego, R., Baeza-Yates, R.: Tuning the Capacity of Search Engines: Load-driven Routing and Incremental Caching to Reduce and Balance the Load. To appear in TOIS (2009)
16. Gan, Q., Suel, T.: Improved Techniques for Result Caching in Web Search Engines. In: WWW (2009)

A Task-Based Evaluation of an Aggregated Search Interface

Shanu Sushmita, Hideo Joho, and Mounia Lalmas

Department of Computing Science, University of Glasgow

Abstract. This paper presents a user study that evaluated the effectiveness of an aggregated search interface in the context of non-navigational search tasks. An experimental system was developed to present search results aggregated from multiple information sources, and compared to a conventional tabbed interface. Sixteen participants were recruited to evaluate the performance of the two interfaces. Our results suggest that the aggregated search interface is a promising way of supporting non-navigational search tasks. The quantity and diversity of the retrieved items which participants accessed to complete a task, increased in the aggregated interface. Participants also found the aggregated presentation easier to access to retrieved items and to find relevant information, compared to the conventional interface.

1 Introduction

A recent study reported that 80% of queries submitted to search engines are non-navigational [14]; people are often seeking general information on a broad topic such as "global warming" or "nutrition". Information needs behind such non-navigational queries are often satisfied by relevant information collected from multiple documents in different genres. Due to the increased quantity and diversity of multimedia contents available on the web, images, audio, movies are also becoming relevant to many queries. A conventional way of gathering relevant information from several *information sources* (e.g., web, image, news, wiki) is to browse the search results of individual sources separately available in search engines.

However, a new paradigm of search result presentation has been emerging; aggregated search interfaces. An aggregated search interface is designed to integrate retrieval results from different information sources into a single result page. In this paradigm, users do not have to visit separate pages to browse the search results to access a range of retrieved items. There appears to be two types of integration; blended and non-blended. A blended integration tends to present a single ranked list based on multiple sources, while a non-blended integration tends to present multiple sources in a separate panel in the same page.

Although a log analysis suggested a potential need of aggregated search interfaces [19], there are many unexplored research questions in this paradigm. One such question is the effectiveness of aggregated search interfaces in supporting non-navigation search tasks. In this paper, we present a task-based user

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 322–333, 2009.
© Springer-Verlag Berlin Heidelberg 2009

study which compares the performance of an aggregated search interface to a conventional interface.

The outline of the paper is as follows. Section 2 discusses background and related work. Our experimental design is described in Section 3. Section 4 presents the results of our study, and their analysis. Finally, Section 5 discusses our findings and future work.

2 Background and Related Work

The search interfaces such as Grouper [5] and Flamenco [3] are some of the conventional ways of organizing retrieved documents or an entire document collection. The former is based on the clustering approach whereas the later follows the faceted browsing approach. Clustering aims to group similar documents together so that users can see multiple aspects from a set of retrieved documents. Whereas, the faceted browsing approach enables users to navigate along the structure of the collection, for example, according to the age, style or school, and creator for an art gallery collection [3]. Users can submit a query but also can browse other items via related facets. Although these approaches are useful for getting multiple aspects of a given query, they are typically single source applications.

Federated search, distributed information retrieval, and metasearch engines are the techniques that aim at providing results from various sources. With the former two, a broker receives the query from the user and selects a relevant sub-set of collections for that query. The top ranked results returned from the selected collections are merged into a single list. Current collection selection methods compare the query with the summary of each collection (term statistics [11] or sample documents [17,16]) and rank collections accordingly.

A metasearch engine sends a user query to several other search engines and/or databases and aggregates the results into a single list or displays them according to their source. Metasearch engines enable users to enter search criteria once and access several search engines simultaneously. They operate on the premise that the web is too large for any one search engine to index it all and that more comprehensive search results can be obtained by combining results from several search engines. This also may save the user from having to use multiple search engines separately.

An aggregated search can be seen as an extension of metasearch as it also provides information from different sources. However, the distinction of information sources is more apparent in aggregated search interfaces since the individual information sources retrieve items from very different collections. Yahoo! alpha[1] and Naver[2] are an example of such aggregation approach. These two systems use the non-blended integration where individual sources are presented in a dedicated panel within a single result page, while other search engines adapt a blended integration.

[1] http://au.alpha.yahoo.com/

[2] http://www.naver.com/

In order to support users with a broad query or ambiguous information need, providing diverse information to users has become necessary. More attention is now being paid towards providing diverse results to the users (see e.g. [1,2]). For example, a study to measure the diversity within image search results can be seen in [4].

Aggregated search also attempts to achieve diversity by presenting results from different information sources (image, video, web, news, etc) on one result page. Here, the aim is to provide diversity across information sources. However, evaluation outcomes regarding the effectiveness and usefulness of aggregated search have been limited in the literature, which we intent to remedy with our work. In this paper, we describe a task-based evaluation of an aggregated search interface.

3 Experimental Design

A within subject experiment design was used in our study, where two search interfaces (controlled and experimental) were tested by sixteen participants, performing two search tasks with each interface.

In the following subsections we define the research hypotheses of this study and discuss the experiment designed to investigate the hypotheses.

3.1 Research Hypotheses

The overall hypothesis of our study is that *an aggregated presentation can facilitate non-navigational search tasks by offering diversified search results*. More specifically, we formulate the following sub-hypotheses to investigate:

H1 An aggregated presentation can increase the quantity and diversity of documents viewed by users to complete a task.
H2 An aggregated presentation can increase the quantity and diversity of relevant information collected by users to complete a task.
H3 An aggregated presentation can improve users' perceptions on the search system.

While an increased number of clicks can be seen as a sign of confusion in navigational queries, informational search tasks often require to view a range of documents to complete the task. Therefore, an effective interface should be able to facilitate the browsing of retrieved documents (**H1**). This should also affect the relevant information collected to complete a task (**H2**). Finally, participants were expected to have a positive perception on the system that enabled them to perform a task successfully (**H3**).

3.2 Search Interfaces

Two search interfaces, called DIGEST system, were devised to address our research hypotheses. Both interfaces used the same back-end search engine (Yahoo!

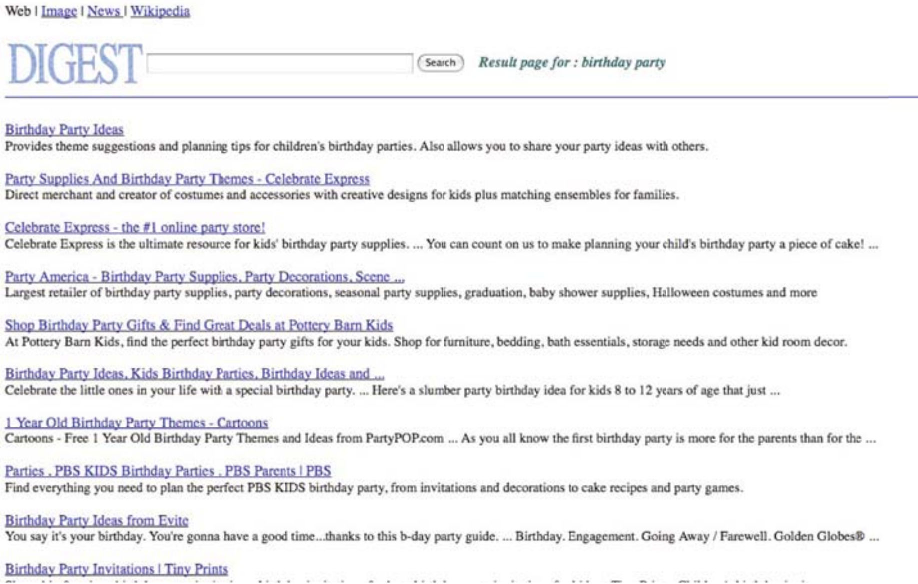


Fig. 1. Controlled System (Tabbed)

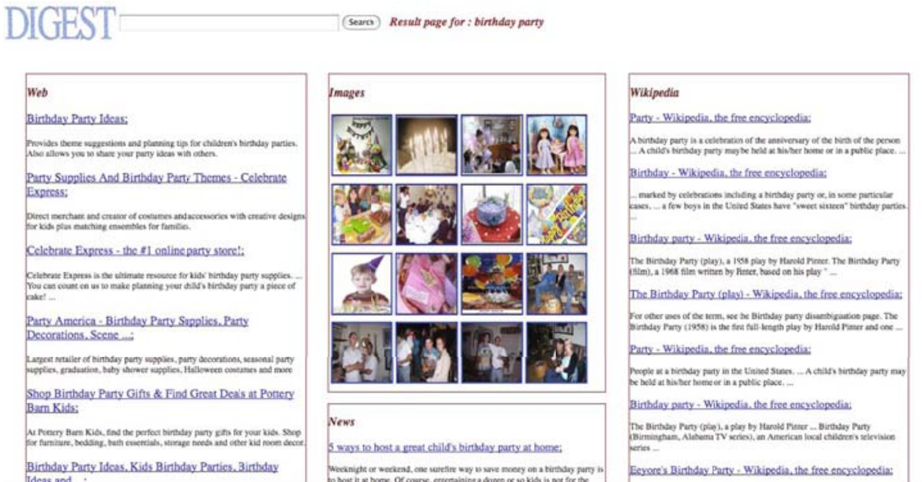


Fig. 2. Experimental System (Aggregated)

search API). For a given query, the API was set to retrieve the top 30 items from four information sources, in this paper, Web, Image, News, and Wiki. The difference between the two interfaces was the presentation of retrieved items.

Figure 1 shows the controlled system where the results from the four sources were presented in a separate tab. The default source was set to the Web tab, and users can click other tabs at the top of the interface to view the results from

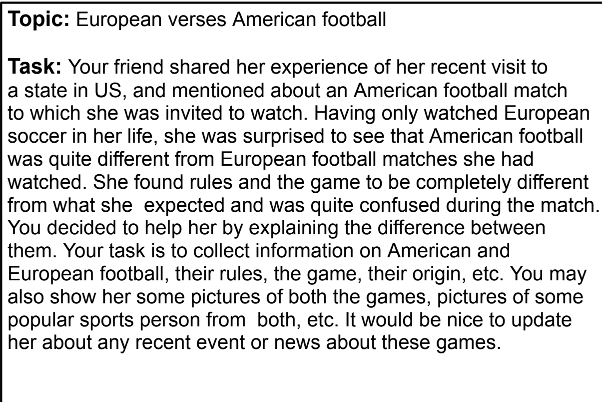
other sources. This represented a conventional vertical presentation of search results available in major search engines. The controlled system presented the first 10 results for every selected information source with an option of "more results" at the bottom to view the remaining 20 results (in chunks of 10).

Figure 2 presents the experimental system where the results from the four sources were integrated into a single page. This represented an aggregated presentation of search results. The first 10 web results, 12 image results, 10 wiki results and 5 news results, were shown, in each corresponding panel. Every information source on the experimental system also had an option of "more results" (similar to the controlled system) in order to view the remaining results. The layout of the four sources was arbitrarily designed and fixed throughout the experiment. A formal study to determine an optimal layout is left for future work.

3.3 Task

Participants of our user study were asked to perform non-navigational search tasks using the interfaces described above. Each search task was based on the simulated work task situation framework proposed by [10]. The framework was designed to encourage participants to engage with an artificial task by giving a situational background scenario of the task. Figure 3 shows an example of the search scenario. As can be seen, our search tasks required to browse several documents and collect relevant information from multiple sources. Participants were asked to copy and paste relevant texts, URLs, and images to a word processing software during the task. We used the software as an electronic notebook. Examples of notebooks made by participants are shown in Figure 4.

We prepared 6 search scenarios so that participants could choose the scenarios based on their interest. This design aimed to facilitate participants engagement with the artificial search tasks. Participants were given 15 minutes to complete



Topic: European verses American football

Task: Your friend shared her experience of her recent visit to a state in US, and mentioned about an American football match to which she was invited to watch. Having only watched European soccer in her life, she was surprised to see that American football was quite different from European football matches she had watched. She found rules and the game to be completely different from what she expected and was quite confused during the match. You decided to help her by explaining the difference between them. Your task is to collect information on American and European football, their rules, the game, their origin, etc. You may also show her some pictures of both the games, pictures of some popular sports person from both, etc. It would be nice to update her about any recent event or news about these games.

Fig. 3. An example of a simulated task

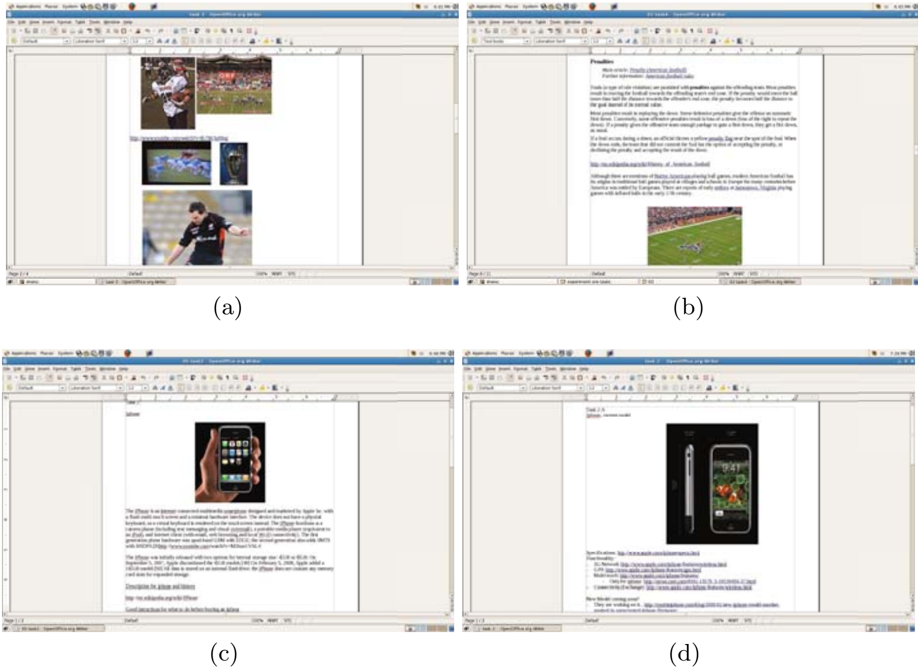


Fig. 4. Sample information collected during search tasks by participants

each task. Each participant performed four search tasks, two with the experimental system and two with the controlled system. The order of the systems was rotated to reduce learning effects.

3.4 Participants

The experiment was carried out with 11 males and 5 females from our university. Out of 16 participants, 7 were undergraduate students, 2 postgraduate students, 3 PhD students, and 4 were research staff members. The participants were from various educational fields, namely, computing, business management, arts and commerce. The participants were recruited through our call for participation email distributed to several lists. An entry questionnaire established that 82% of participants stated that they had accessed more than one information source to complete a search task. Therefore, our participants were not totally unfamiliar with search tasks that require multiple sources. However, none had used our interfaces or tasks before.

3.5 Procedure

For each participant, the experiment was performed in the following manner. When they arrived at the experiment site, they were welcomed and explained

the overall aim of the experiment. When they agreed to participate, a consent form was signed. Then, they were asked to fill in an entry questionnaire to capture their profile and search background. Next, they had a training session with both interfaces using a sample search task. The training session typically lasted for five minutes.

Then, they were asked to perform the first search task by selecting the most interesting scenario from the six scenarios. During the task, the system automatically logged participants' interaction with the interface. When the first task was completed, they were asked to fill in a post-task questionnaire to capture their subjective assessments on the system and task. Then, participants were informed of the change of the interface, and the second scenario was selected. This was repeated four times. After the completion of the four tasks, they were asked to fill in an exit questionnaire to capture their perceptions of systems and tasks as a whole. Participants were rewarded fifteen pounds for their participation after the experiment.

4 Results

This section presents the results of our experiment based on the research hypotheses stated in Section 3.1. We had a total of 32 search sessions per system in the analysis. To measure the statistical significance of the results, we applied both t-test (parametric) and Wilcoxon signed rank test (non-parametric) to the difference between the controlled and experimental systems. All tests were paired and two-sided, and critical value was set to 0.05, unless otherwise stated.

4.1 Quantity and Diversity of Documents Viewed

The first hypothesis **H1** looked at the effect of an aggregated presentation on the quantity and diversity of documents participants viewed to complete a task. To examine this hypothesis, we first analysed participants' click-through data on different information sources. The results are shown in Table 1.

The bottom row of the table shows the average number of retrieved items viewed to complete a task. As can be seen, participants viewed a significantly larger number of items in the experimental system when compared to the controlled system. The breakdown of the information sources suggests that the

Table 1. Frequency of participants' clicks per information sources (N=32)

Source	Controlled system		Experimental system		T-Test	Wilcoxon-Test
	Mean	SD	Mean	SD	p-value	p-value
Web	7.7	6.7	8.6	5.9	.3696	.1847
News	1.7	1.7	1.0	1.2	.0663	.1039
Wiki	1.0	1.5	2.7	2.2	**.0002**	**.0005**
Image	2.4	3.2	6.8	7.7	**.0013**	**.0004**
All	12.8	8.8	19.1	12.1	**.0002**	**.0024**

Table 2. Combination of information sources, where W=web, I= image, N=news and Wi= wiki

Diversity	Sources	Controlled system	Experimental system
1	W	4	0
2	W+I	2	3
2	W+N	1	2
2	W+Wi	1	2
2	I+Wi	0	1
3	W+I+N	12	1
3	W+N+Wi	3	1
3	W+I+Wi	1	9
4	W+I+N+Wi	8	13
	Total	32	32

difference was due to the significantly different frequency in the Wiki and Image sources. These results provide a support for that the aggregated presentation increased the quantity of retrieved items viewed.

We also looked at the combination of information sources accessed by participants to complete a task. The results are shown in Table 2. As can be seen, in five more sessions, participants accessed all four information sources in the experimental system when compared to the controlled system. Also, more sessions were completed by a single source (Web) in the controlled system. This suggests that the aggregated presentation encouraged participants to view more diversified sources from search results. We also noticed that the frequent source was different in the two systems. When we looked at the diversity score 3, the sources of Web, Image, and News was the most popular combination in the controlled system while the Web, Image, and Wiki were the most common combination in the experimental system. We will discuss this aspect later.

Overall, our results provided some evidence to support **H1**.

4.2 Quantity and Diversity of Relevant Information Collected

The second hypothesis examined whether or not an aggregated presentation increased the quantity and diversity of relevant information collected by participants to complete a task. To answer this hypothesis, we performed a similar analysis to the previous section but on the number of texts, images, and URLs collected in the notebook. The number of texts was counted based on the number of paragraphs. The results of the analysis are shown in Table 3.

Again, the bottom row of the table shows the average number of collected items to complete a task. As can be seen, participants collected five more items in the experimental system when compared to the controlled system. The difference was found to be significant by the Wilcoxon test. The breakdown of collected items shows that participants tended to collect more items in all three types (Texts, Images, and URLs) when they used the experimental system. However, no difference was found to be significant.

Table 3. Information collection using Controlled & Experimental systems

	Controlled system		Experimental system		T-Test	Wilcoxon-Test
	Mean	SD	Mean	SD	p-value	p-value
Text	7.8	13.2	10.8	21.4	.3657	.3211
Images	3.3	2.8	4.6	3.4	.1140	.0815
URLs	6.1	5.3	7.4	7.1	.1956	.3250
All	17.3	12.7	22.7	18.6	.1173	**.0409**

Table 4. Information collected using Controlled & Experimental systems for text, image and ulr combinations. Here, I=image, T= text and U = url.

Diversity	Information Type	Controlled system	Experimental system
1	I	0	2
1	U	2	0
2	I+T	9	6
2	I+U	11	1
2	T+U	0	12
3	I+T+U	10	11
	Total	32	32

Table 4 shows the combination of the collected items. As can be seen, the number of sessions where all three types were collected (diversity score 3) was similar across the systems. The frequency in the other two diversity scores (diversity score 1 and 2) was also found to be comparable. However, there was some noticeable difference in the combinations. More specifically, the combination of Image and Text (I+U) and combination of Text and URLs (T+U) had a very different frequency across the systems. The cause of this difference is not entirely clear to us. We are currently examining the log files to get further insight into this phenomenon.

To summarise, our results provided partial evidence to support the quantity aspect of **H2**, but no obvious evidence was found to support the diversity aspect of the hypothesis.

4.3 User Perceptions

The last hypothesis looked at the effect of the aggregated presentation on participants' perceptions of the systems. To answer this hypothesis, we analysed participants' subjective assessments on the systems, which were captured by a 5-point Likert scale in the exit questionnaire. More specifically, we asked their agreement on the two following statements for each of the two systems.

Q1 The system was useful to complete my search tasks (1 = Strongly agree; 5 = Strongly disagree).

Q2 It was easy to find relevant information with the system (1 = Strongly agree; 5 = Strongly disagree).

Table 5. Users' perceptions on the systems (N=16)

	Controlled system		Experimental system		T-Test	Wicoxon-Test
	Mean	SD	Mean	SD	p-value	p-value
Q1	2.4	1.1	1.9	1.1	.1311	.1771
Q2	2.4	1.0	1.8	0.9	**.0430**	**.0466**

Since our hypotheses expected the experimental system to have a better assessment than the controlled system, the statistical tests were applied with paired but one-tailed where an alternative was set to be greater. Note that a lower value represented a higher degree of agreement in our analysis. The results are shown in Table 5. As can be seen, participants tended to find the experimental system easier to find relevant information to complete a task. Although participants tended to give a better score on the experimental system regarding the usefulness, the difference was not found to be significant.

We also asked participants which system was easier to access search results in the exit questionnaire. 75% of participants selected the experimental system for the question. Overall, these results provide partial evidence to support **H3**.

5 Discussion and Future Work

Aggregation is an emerging paradigm of the search result presentation. There are many unexplored questions in this area. In this paper, we performed a task-based user study to compare the effectiveness of an aggregated presentation to a conventional presentation. In particular, we investigated the effect of the aggregated presentation on the quantity and diversity of information objects accessed by users in non-navigational search tasks. This section first discusses the limitation of our study, followed by the implications of our results on the design of aggregated search interfaces.

There are some limitations in our study. First, we used only one back-end search engine to test the effectiveness of the interfaces. Although this made the comparison fair, the implication of our results is limited to this particular engine. Second, we tested the systems with a small number of topics compared to a system-centred evaluation. Other types of tasks such as a decision-making task will also give us a better understanding of the effect of aggregated presentation. Third, the collected items were based on perceived relevance and the quality of collected items was not assessed. Finally, the layout of aggregation was fixed in our experiment. This seems to have an implication on participants' information seeking behaviour, which will be discussed next.

Beaulieu [9] observed the trade-off between the complexity of search interfaces and cognitive load of the users. This applies to the design of aggregated search interfaces, too. Our experimental system used a more complex presentation than the controlled system to integrate multiple information sources in a single page. Therefore, the aggregated interface could increase the cognitive load of the end-users. However, our experimental results suggested that participants

were capable of interacting with an aggregated presentation, and tended to find the experimental system easier to find relevant information when compared to the controlled system. This might be due to the fact that the controlled system still required extra effort to select information sources to access a range of retrieved items.

Another implication was that the layout of aggregation was likely to affect people's selection of information sources. In Section 4.1, we found that the combination of the Web, Image, and News was the most common selection in the controlled system while the Web, Image, and Wiki were the popular selection in the experimental system. They were exactly the same order of the sources in the interfaces. The tab on the top of the controlled interface listed the sources in the order of Web, Image, News, and Wiki. The top three panels of the aggregated interface were the Web, Image, and Wiki. This suggests that people's browsing of information sources can be sequential, and their attention moves horizontally rather than scrolling down the result page vertically. This also implies that an aggregated search interface might be able to offer an effective support by optimising the order of information sources for different tasks or queries.

The last point leads us to formulate our future work which will be addressing research questions such as "Is there an optimal combination and order of information sources?", "How can we model the optimal combination and order of information sources for a given query or task?", "Is the effect of layout strong enough to affect task performance?"

In conclusion, our study provided empirical evidence to support that an aggregated presentation of information sources can increase the quantity and diversify of the retrieved items accessed to complete non-navigational search tasks. Participants tended to find the aggregated presentation easier to access retrieved items and to find relevant information. Although these positive effects were not strong enough to increase the number of relevant information collected, we speculate that an intelligent way of organising information sources is a key to achieve such a goal.

Acknowledgements. This work was carried out in the context of research partly funded by a Yahoo! Research Alliance Gift and the MIAUCE project (Ref: IST-033715). Mounia Lalmas is currently funded by Microsoft Research/Royal Academy of Engineering.

References

1. Radlinski, F., Dumais, S.: Improving personalized web search using result diversification. In: ACM SIGIR conference on Research and development in information retrieval, pp. 691–692 (2006)
2. Coyle, M., Smyth, B.: On the importance of being diverse: analysing similarity and diversity in web search. In: Source Intelligent Information Processing II, pp. 341–350 (2004)
3. Yee, K.P., Swearingen, K., Li, K., Hearst, M.: Faceted metadata for image search and browsing. In: SIGCHI conference on Human factors in computing systems, pp. 401–408 (2003)

4. Sanderson, M., Tang, J., Arni, T., Clough, P.: What Else Is There? Search Diversity Examined. In: Boughanem, M., et al. (eds.) ECIR 2009. LNCS, vol. 5478, pp. 562–569. Springer, Heidelberg (2009)
5. Zamir, O., Etzioni, O.: Grouper: a dynamic clustering interface to Web search results. Computer Networks: The International Journal of Computer and Telecommunications Networking (1999)
6. Heimonen, T., Aula, A., Hutchinson, H., Granka, L.: Comparing the User Experience of Search User Interface Designs. In: CHI 2008 Workshop on User Experience Evaluation Methods in Product Development (2008)
7. Wilson, M.L., Schraefel, M.C., White, R.W.: Evaluating Advanced Search Interfaces using Established Information-Seeking Models. Journal of the American Society for Information Science and Technology (2009)
8. Hoeber, O., Yang, X.D.: User-Oriented Evaluation Methods for Interactive Web Search Interfaces. In: IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops, pp. 239–243 (2007)
9. Beaulieu, M.: Experiments with interfaces to support query expansion. Journal of Documentation 53(1), 8–19 (1997)
10. Borlund, P.: Experimental components for the evaluations of interactive information retrieval systems. Journal of Documentation 56(1), 71–90 (2000)
11. Callan, J.: Distributed Information Retrieval. In: Advances in Information Retrieval, vol. 5, pp. 127–150. Kluwer Academic Publishers, Dordrecht (2000)
12. Diaz, F.: Integration of news content into web results. In: ACM International Conference on Web Search and Data Mining, pp. 182–191 (2009)
13. Dumais, S., Cutrell, E., Chen, H.: Optimizing search by showing results in context. In: SIGCHI Conference on Human Factors in Computing Systems, pp. 277–284 (2001)
14. 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)
15. Kural, Y., Robertson, S., Jones, S.: Deciphering cluster representations. Inf. Process. Manage. 37(4), 593–601 (2001)
16. Shokouhi, M., Baillie, M., Azzopardi, L.: Updating collection representations for federated search. In: ACM SIGIR conference on Research and development in information retrieval, pp. 511–518 (2007)
17. Si, L., Callan, J.: Relevant document distribution estimation method for resource selection. In: ACM SIGIR conference on Research and development in informaion retrieval, pp. 298–305 (2003)
18. Si, L., Lu, J., Callan, J.: Distributed information retrieval with skewed database size distributions. In: dg.o 2003: Annual national conference on Digital government research, pp. 1–6. Digital Government Society of North America (2003)
19. Sushmita, S., Joho, H., Lalmas, M., Jose, J.M.: Understanding domain "relevance" in web search. In: WWW 2009 Workshop on Web Search Result Summarization and Presentation (2009)

Efficient Language-Independent Retrieval of Printed Documents without OCR

Walid Magdy[1,*], Kareem Darwish[2], and Motaz El-Saban[2]

[1] School of Computing, Dublin City University, Dublin 9, Ireland
wmagdy@computing.dcu.ie
[2] Cairo Microsoft Innovation Center, Microsoft, Smart Village, B115, Abou Rawash, Egypt
{kareemd,motazel}@microsoft.com

Abstract. Recent book digitization initiatives have facilitated the access and search of millions of books. Although OCR remains essential for retrieving printed documents, OCR engines remain limited in the languages they handle and are generally expensive to build. This paper proposes a language independent approach that enables search through printed documents in a way that combines image-based matching with conventional IR techniques without using OCR. While image-based matching can be effective in finding similar words, complementing it with efficient retrieval techniques allows for sub-word matching, term weighting, and document ranking. The basic idea is that similar connected elements in printed documents are clustered and represented with ID's, which are then used to generate equivalent textual representations. The resultant representations are indexed using an IR engine and searched using the equivalent ID's of the connected elements in queries. Though, the main benefit of the proposed approach lies in languages for which no OCR exists, the technique was tested on English and Arabic to ascertain the relative effectiveness of the approach. The approach achieves more than 61% relative effectiveness compared to using OCR for both languages. While the reported numbers are lower than that of OCR-based approaches, the proposed method is fully automated, does not require any supervised training, and allows documents to be searchable within a few hours.

Keywords: Printed Documents Retrieval, Image Based Retrieval, OCR.

1 Introduction

Recent initiatives have focused on digitizing large repositories of legacy books [2][32]. Such initiatives have been successful in digitizing millions of books in a variety of languages, and search has been one of the leading services for these digitized books. Optical Character Recognition (OCR) is the most common approach for generating searchable versions of the books; however, this process typically introduces errors in the textual representations of books depending on several factors, such as quality of paper, printing, font, OCR training, and scanning. Further, morphological and

* Walid performed the work while being at the Cairo Microsoft Innovation Center.

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 334–343, 2009.
© Springer-Verlag Berlin Heidelberg 2009

orthographic features of some languages, such as connected characters, diacritics, dots, and word compounds complicate the OCR process considerably. These factors complicate the development of OCR engines and make the engines generally language dependent.

This paper introduces a technique for searching printed documents based on matching between text queries and scanned documents in an intermediate representation, avoiding matching in the text domain, availing the need for OCR, and avoiding matching in the image domain with the inherent computational inefficiencies and blunted ability to properly weigh documents. Briefly, connected components in scanned documents, which may be single characters or multiple connected characters, are extracted and clustered based on their shapes; each cluster is assigned an ID; and intermediate representations of documents, based on the ID's corresponding to connected components, are generated and indexed. When a user issues a text query, the text is rendered into an image; the connected components in the rendered image are matched with the connected components from the documents; an intermediate representation of the query, based on the ID's corresponding to connected components, is generated; and the intermediate representation of the query is used to query indexed documents. This approach has the advantage of being computationally tractable, effective for documents for which no OCR exists, generally language independent, applicable for sub-word matching, and with proper query formulation can use existing weighting formulas. The technique is tested on document images in Arabic and English, which have mature OCR's and whose OCR's provide solid baselines to compare against, to show that the proposed approach works on languages with dramatically different orthographies. The paper is organized as follows: Section 2 surveys related work; Section 3 describes the proposed approach; Section 4 describes experimental setup; Section 5 discusses the results; and Section 6 concludes and suggests future work.

2 Related Work

Retrieval of OCR degraded text documents has been reported on for many languages, including English [7][10][30], Chinese [33], and Arabic [4]. Generally, retrieval effectiveness is adversely affected by increased degradation and decreased redundancy of search terms in documents [9].

To overcome the effects of OCR errors, several techniques were proposed including character n-gram indexing [4][7], OCR error correction [14][15][20], query garbling [3], fusing multiple OCR outputs [16], or avoiding the OCR stage altogether.

Word spotting is a method for avoiding OCR in searching printed documents. It involves transforming query text into an image and matching the image against the document images. This approach was used to search the George Washington collection [24][26], a set of handwritten manuscripts, and to search document images in several languages [18]. This approach is suitable for documents where automatic character recognition generally yields poor results or when no OCR exists. One of the major disadvantages of word spotting is the computational demand, especially when performed at query time [18]. Hence, some work has focused on optimizing the spotting process by clustering similar words into defined classes, assigning an ASCII

equivalent to each cluster manually, and then searching the ASCII equivalent using the text of the query [17] utilizing user feedback [12] or a training set [25]. Kumar et al. [13] improved efficiency using locality sensitive hashing to efficiently find nearest neighbors. Another attempt involved using preset "keywords" rendered into images and matched against words document images [28]. Although more efficient, the list of "keywords" inherently limits the number of words that can found. Another drawback stems from the limited ability to compute word statistics such as term and document frequencies, which are important for ranking. Although some attempts were made to compute such statistics [28], many of the word spotting papers report precision and recall for finding single word tokens in document images on rather small datasets and/or a small number of keywords [12][13][28]. Lastly, word spotting handles whole words and not characters, even though retrieval of degraded documents using character n-grams is known to be better than retrieval using words.

A hybrid approach tries to integrate information from word spotting and automatically recognized text to achieve better search results [29]. Although these techniques proved their effectiveness in retrieving information from the image domain, they suffer from: scalability issues, compared to OCR based techniques, the limited ability to use text statistics to aid ranking, and the inability to perform sub-word matching. These issues are addressed in this work. Also, images of words need to be segmented prior to matching, which is not a requirement for this work. Further, this paper compares retrieval effectiveness of the technique against retrieval effectiveness on ground truth (or near ground truth) data as a reference. Such comparison is absent from most word spotting papers. Finally, this paper investigates the issue of having multiple fonts, which is not addressed in most previous work.

As for Arabic, upon which the proposed technique is tested, several challenges exist including different letter shapes according to position and optional use of word elongations and ligatures are prevalent. Most letters contain dots that distinguish them from other letters and optional diacritics may exist. The attachment of pronouns and coordinating conjunctions to words leads to an estimated 60 billion possible word surface forms [1]. There are several commercial Arabic OCR systems [8][11].

3 Approach

This paper introduces a technique for searching printed documents based on matching between text queries and scanned documents in an intermediate representation. As in Figure 1, a set of printed document images are segmented into a set of elements, where an element is a connected shape representing a character or a group of connected characters, as in Arabic. These elements (shapes) are clustered according to measurable features to group similar elements together. Each cluster is assigned a unique cluster ID, along with a feature vector representing the mean of the vectors of features in a cluster. Then each element in every document image is replaced with the corresponding cluster ID in the same reading order of the text in the document image to generate an alternative representation of the document. Clustering and document translation are insensitive to language features. Note that depending on the language of a document, reading order could be left to right, right to left, or top down. In this paper, the reading order of the language is manually provided, namely right-to-left for

Arabic and left-to-right in English. The resulting documents in the alternative representation are indexed.

Segmentation of a document image into elements is carried out using connected component analysis [5]. Elements smaller than a certain size, namely dots, Hamza, and diacritics in Arabic, are neglected, because they are not independent characters and they are often confused with speckle and dust. Although removing diacritics and Hamza helps retrieval, removing dots adversely affects retrieval and requires further investigation to avoid their removal. Each segmented element is represented by the pixel pattern in a bounding box around the element. A scale and translation invariant pattern matching method is used to compare elements to one another with summation of absolute difference as the similarity measure. To improve clustering and matching efficiency, the ratio of width to height is first compared, allowing up to 20% difference in ratio. Scale invariance is achieved through normalizing the size of each element to a height of 54 pixels in order to overcome the variations in width to height ratio and font size. This is especially important for headers, typically in larger sizes. Translational invariance is introduced in element matching by allowing a shift of up to two pixels. Clustering is done using a fast sequential clustering algorithm that reportedly produces acceptable clustering called BSAS [31]. BSAS uses two parameters: a threshold θ that decides when to form a new cluster, and the maximum number of clusters. A value of 0.13 is used for θ (based on side experiments) with no limit on the number of clusters to make the clustering adaptive to any collection or language at hand. To avoid the input order dependence on the final clusters found, the BSAS algorithm is run twice on the dataset. The representative prototype for each cluster is chosen to be the mean (centroid) for computational efficiency. Once clustering is done, a document image is converted to the new representation using the cluster ID's. No attempt is done to segment words, as word segmentation is language dependent and is often error-prone. A phrase operator is utilized at query time to compensate for lack of segmentation. One of the issues that arise is that an element may be assigned to the wrong cluster. To overcome this problem, each element is used to search through the clusters and the closest clusters are generated, potentially replacing an element by more than 1 cluster ID. Ideally, the search engine would be modified to allow for multiple representations (or equivalents) of a token to be indexed as one token as in [22]. However, most retrieval engines do not support such indexing and the situation is remedied in part via the use of proximity operators as described later. In case multiple fonts are used in a book, the clustering technique would normally generate separate clusters for each font automatically. At query time, a query would be rendered in more than one font to match the fonts in the document images. Currently, the closest fonts are determined manually. The rendered query image is then segmented into elements and each element is matched to the clusters in the document images. To increase robustness to errors, for each element, a set of candidate clusters ID's is generated, and the ID's are considered as "synonyms". A query consisting of an ordered set of IDs is formulated given the candidate cluster IDs for each element. For example, given the query "this", after rendering the query and segmenting it into elements (where the elements are just the characters in this example), the elements could match the candidate cluster IDs {12, 23}, {10, 32, 291}, {102, 34, 44}, and {61} for "t", "h", "i", and "s" respectively. Consequently, the formulated query would

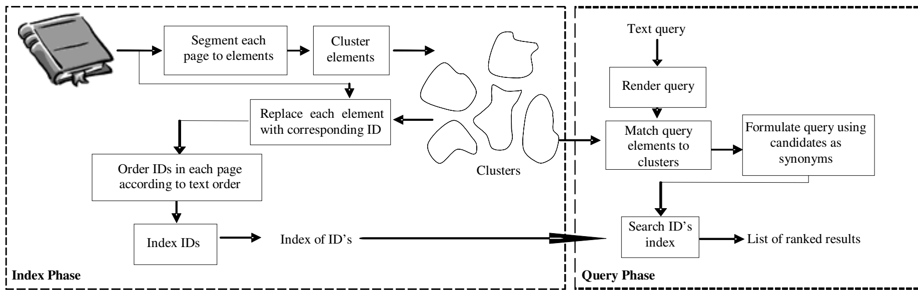


Fig. 1. Overall overview of the system

be "#phrase(#synonym(12, 23) #synonym (10, 32, 291) #synonym (102, 34, 44) 61)". Although using multiple cluster ID's for each element increases ambiguity, retrieval engines are generally tolerant to ambiguity [27]. The phrase operator would insure proper ordering elements in a document without performing word segmentation.

4 Experimental Setup

Two data sets are used to test the proposed technique. The first is the ZAD collection which consists of 2,730 documents, which are obtained from a classical religious Arabic book [4]. The ZAD collection has the advantage of having an error-free (clean) text copy and a document image version that is scanned at 300x300 dpi and an OCR'ed version that was OCR'ed using Sakhr's Automatic Reader, with a word error rate of approximately 39%. The collection has 25 topics and relevance judgments, which were built by exhaustively searching the collection. The number of relevant documents per topic ranges from 3 to 72, averaging 20. The average query length is 5.4 words [4]. Despite the relatively small size of the ZAD collection, the clean version of the ZAD collection has the advantage of behaving in a manner similar to the largest existing standard test collection, namely the TREC 2002 Cross Language Retrieval Track collection [4][21].

The second collection is an English collection composed of 701 images from the second volume of the Engineering Encyclopedia that was published in 1942 and is available from the Internet Archive[1]. Since no query relevance judgments (qrels) are available for this book, the authors extracted a set of 200 entry titles that were subjectively judged as non-repeating in the encyclopedia and used them as queries. The average number of words per query is 2.1 words. The queries are intended to emulate a known-item retrieval task. The book was scanned at 300x300 dpi and OCR is provided by the Internet Archive. Although exact character and word error rates are not available for the OCR of the book, the OCR is subjectively estimated to be nearly error free. There are two fonts being used in the printed book, namely one that is similar to Times New Roman, which constitutes the body of the entries, and another that is similar to Arial Round, which constitutes entry headings. The collection is henceforth referred to as the ENG collection. The authors opted to develop their own

[1] The book is available from the Internet Archive (www.archive.org) as images and OCR.

test collection, because although the document images for the documents in TREC confusion track exist, the start and end of actual OCR'ed documents for the track do not match the boundaries of the available document images [10]. Therefore, the confusion track qrels cannot be used directly. For efficiency, since the number of English characters is limited, each character was rendered individually in lower and upper-cases in the two fonts and each variation was assigned cluster ID's. The varying ID's were used as synonyms at query time.

Indexing and retrieval was done using the Indri retrieval toolkit, which combines inference network models with language modeling [19]. Indri query language includes a synonym operator, #syn(…) [23], and a proximity operator that behaves similar to a phrase operator, #N(…), to restrict the sequence order of elements in one word in the query, where N is the number of inserted elements within a sequence. At query time, for the Arabic text, a query word is just the surface form of the query, while for English a word is the sequence of character 3 to 5-grams [7][2]. In case of replacing each element with the best matching ID in the indexing phase, N is set to 1 (no insertions are allowed). When replacing elements with more than one ID in the indexing phase, N is set to 2 * (No. of Equivalents – 1) + 1.

Mean average precision (MAP) and Mean Reciprocal Rank (MRR) are used as the figures of merit for testing retrieval effectiveness for the ZAD collection and the ENG respectively. When needed, a paired two tailed t-test with p-value of 0.05 is used to indicate statistical significance. The t-test is sufficiently reliable though the normality condition might not be met [27].

5 Results and Discussion

For the ZAD collection, the clustering of elements in the document images yields 12,058 clusters, which is very close to the number of unique elements in the clean text, namely 11,152 unique elements. During the indexing step, each element in the document images is assigned M ID's corresponding to the nearest clusters and M is varied between 1 and 3. Similarly, when query elements are transformed into ID's, each element is replaced by N ID's of closest element clusters, where N ranges between 1 and 5, and the resulting N ID's for a single element are placed inside the Indri synonym operator. Although increasing the values of M and N would increase ambiguity, the increase would help alleviate mismatch errors and moderate use of synonyms is tolerable [3][22][23]. Table 1 reports the mean average precision (MAP) over 25 queries for varying values of M and N.

Table 1 indicates that the optimal values of M and N are 3, leading to a MAP of 0.23. It is worth noting that some of the 25 queries are relatively long with up to 10 words. Nonetheless, the retrieval process usually takes few milliseconds without careful optimization. There are 4 baselines experimental setups, namely retrieval against the clean version and OCR'ed versions of the ZAD collection with and without dots that distinguish letters from each other. To remove dots, letters that have the same shape, but different dots, are conflated. The runs where dots are removed are

[2] Side experiments have shown that using a combination of 3, 4, 5-grams produced the best results.

Table 1. MAP for the ZAD collection with varying values of M and N

	N = 1	*N = 2*	*N = 3*	*N = 4*	*N = 5*
M = 1	0.15	0.16	0.19	0.19	0.20
M = 2	0.12	0.16	0.20	0.19	0.19
M = 3	0.12	0.16	0.23	0.21	0.21

Table 2. MAP of the proposed technique compared to the baselines

Best from Table 1	Clean	Clean (no dots)	OCR	OCR (no dots)
0.23	0.45	0.33	0.38	0.25

Table 3. MRR for the ENG collection with varying values of M and N using Arial Round, Times New Roman, and combined runs

		N = 2	N = 3	N = 4	N=5	N=6
Arial Round	M = 1	0.49	0.53	0.50	0.51	0.49
	M = 2	0.43	0.44	0.37	0.35	0.32
	M = 3	0.42	0.39	0.32	0.27	0.25
Times New Roman	M = 1	0.24	0.32	0.36	0.39	0.37
	M = 2	0.32	0.32	0.34	0.33	0.28
	M = 3	0.32	0.27	0.27	0.28	0.22
Combined runs	M = 1	0.50	0.54	0.55	0.58	0.59
	M = 2	0.48	0.48	0.46	0.46	0.42
	M = 3	0.51	0.44	0.41	0.39	0.35

designed to simulate the situation where dots could not be used to disambiguate letters or when a language uses few or no dots. Table 2 compares the proposed technique with all 4 baselines.

The results in Table 2 show that the proposed technique achieves 61% relative effectiveness compared to OCR, primarily because dots are ignored. However, when dots are eliminated from the OCR text, the proposed technique achieves 92% relative effectiveness (the difference is statistically significant). Notably, most languages do not use dots to disambiguate letters as in Arabic or other languages such as Farsi, Urdu, and Kurdish. Also, despite the connected characters in Arabic leading to a large number of different elements, the proposed approach achieves acceptable retrieval effectiveness.

For the ENG collection, the clustering of elements in the document images yields 309 clusters, which is more than the actual number of unique elements in the text (roughly 200 – including lower and uppercase letters, different fonts, punctuation, etc.). Since the number of elements in English is limited to just a few hundred, only the first 50 document images are used to generate the clusters. During indexing, each element in the document images is assigned M ID's corresponding to the nearest clusters and M is varied between 1 and 3. Similarly, when query elements are transformed into ID's, each element is replaced by N ID's of closest element clusters, where N ranges between 2 and 4, and the resulting N ID's for a single element are placed inside the Indri synonym operator. Each element in the query is generated in uppercase and lowercase forms. Also, since the text of the book is written using two

different fonts, the queries are constructed using Arial Round once and using Times New Roman a second time. Table 3 reports MRR values over the 200 queries for varying values of M and N using Arial Round, Times New Roman, and combined runs, for which the score of each document in the ranked list(s) is the average score from both ranked lists or is the score given by either (if it existed in only one).

The results show that indexing more than one variation is not as good as using the synonym operator at query time. This could be due to having too few natural clusters of elements in English, compared to Arabic, or to the fact that elements are constituted of one character in English, compared to multiple characters in Arabic. Both factors can yield to greater ambiguity in English that may adversely affect the use of the proximity operator at query time. Ideally, it would be better to have IR engine support for indexing multiple variants as 1 token. Also, results in Tables 3 show the sensitivity of the technique to the choice of font with which the query is rendered, where a good choice would lead to better matching with existing clusters and better retrieval and vice versa. Although retrieval might have been better if the entire book had 1 font, using multiple fonts and matching them in document images can be useful. The proposed technique allows for many processing steps that are not easily possible using image-based matching including: character n-gram matching, case folding, multiple font handling, and term weighting.

The proposed approach achieves 63% relative effectiveness compared to a OCR-based system, which yielded an MRR of 0.93, over the same set of 200 queries. The relative effectiveness for English is consistent with the relative effectiveness achieved for Arabic, suggesting that the technique is fairly language independent.

6 Conclusion and Future Work

This paper introduces a language independent approach for performing IR on printed documents. The approach combines image matching with text retrieval techniques to perform efficient retrieval of document images without OCR. Experimental results show that the proposed approach can achieve more than 61% relative effectiveness compared to using OCR-based techniques. The proposed technique is intended to address retrieval for languages for which there is no good OCR and without the extensive effort required to build an OCR system. The technique was applied to Arabic and English to compare the technique to OCR based retrieval. The proposed technique achieves a comparable level of efficiency in search, compared to OCR based retrieval, while being language independent and while allowing for multiple language scripts, which would cluster independently, provided that multi-lingual queries are rendered properly. Unlike many of the previous word spotting techniques, the proposed approach does not require full word matching, makes use of the retrieval engine facilities such as ranking, synonym operators, and proximity operators, does not require word segmentation, does not require training data, and is capable of handling a limited number of fonts simultaneously. For future work, some issues linger related to image processing and IR. For image processing, different matching techniques can be tested that can be more effective and allow font-independent matching. Also, other clustering techniques can be tested for more accurate clusters generations. As for IR, a search engine can be built that allows the indexing of synonyms with different weights for each, which would reflect on the image matching candidates.

References

1. Ahmed, M.: A Large-Scale Computational Processor of Arabic Morphology and Applications. MSc. Thesis, Faculty of Engineering, Cairo University, Cairo, Egypt (2000)
2. Barret, W., Hutchison, L., Quass, D., Nielson, H., Kennard, D.: Digital Mountain: From Granite Archive to Global Access. In: Intl. Workshop on Doc. Image Analysis for Libraries, pp. 104–121 (2004)
3. Darwish, K., Oard, D.: Probabilistic Structured Query Methods. In: SIGIR, pp. 338–344 (2003)
4. Darwish, K., Oard, D.: Term Selection for Searching Printed Arabic. In: SIGIR, pp. 261–268 (2002)
5. Gonzalez, R., Woods, R.: Digital Image Processing, 3rd edn. (2008)
6. Ester, M., Kriegel, H., Sander, J., Xu, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: KDD (1996)
7. Harding, S., Croft, W., Weir, C.: Probabilistic Retrieval of OCR-degraded Text Using N-Grams. In: European Conference on Digital Libraries, pp. 345–359 (1997)
8. Hassibi, K.: Machine Printed Arabic OCR. In: AIPR Workshop: Interdisciplinary Computer Vision, SPIE Proceedings, vol. 2103, pp. 126–134 (1994)
9. Hawking, D.: Document Retrieval in OCR-Scanned Text. In: 6th Parallel Comp. Workshop, P2-F (1996)
10. Kantor, P., Voorhees, E.: Report on the TREC-5 Confusion Track. TREC-5, p. 65 (1996)
11. Kanungo, T., Marton, G., Bulbul, O.: OmniPage vs. Sakhr: Paired Model Evaluation of Two Arabic OCR Products. In: SPIE Conf. on Doc. Recognition and Retrieval (VI), vol. 3651, pp. 109–120 (1999)
12. Konidaris, T., Gatos, B., Ntzios, K., Pratikakis, I., Theodoridis, S., Perantonis, S.J.: Keyword-guided word spotting in historical printed documents using synthetic data and user feedback. In: IJDAR (2007)
13. Kumar, A., Jawahar, C., Manmatha, R.: Efficient Search in Doc. Image Collections. In: ACCV (2007)
14. Lu, Z., Bazzi, I., Kornai, A., Makhoul, J., Natarajan, P., Schwartz, R.: A Robust, Language-Independent OCR System. In: AIPR Workshop: Advances in Computer Assisted Recognition, SPIE, vol. 3584 (1999)
15. Magdy, W., Darwish, K.: Arabic OCR Error Correction Using Character Segment Correction, Language Modeling, and Shallow Morphology. In: EMNLP, pp. 408–414 (2006)
16. Magdy, W., Darwish, K., Rashwan, M.: Fusion of Multiple Corrupted Transmissions and its Effect on Information Retrieval. In: Seventh Conference on Language Engineering, ESOLEC, pp. 351–358 (2007)
17. Manmatha, R., Croft, W.B.: Word Spotting: Indexing Handwritten Archives (1997)
18. Marinai, S., Marino, S., Soda, G.: Font Adaptive Word Indexing of Modern Printed Documents. Transactions Pattern Analysis and Machine Intelligence (2006)
19. Metzler, D., Croft, W.B.: Combining the Language Model and Inference Network Approaches to Retrieval. Info. Processing and Management 40(5), 735–750 (2004)
20. Mittendorf, E., Schäuble, P.: IR can Cope with Many Errors. IR 3(3), 189–216 (2000)
21. Oard, D., Gey, F.: The TREC 2002 Arabic/English CLIR Track. In: TREC 2002 (2002)
22. Oard, D.W., Ertunc, F.: Translation-Based Indexing for Cross-Language Retrieval. In: Crestani, F., Girolami, M., van Rijsbergen, C.J.K. (eds.) ECIR 2002. LNCS, vol. 2291, pp. 324–333. Springer, Heidelberg (2002)
23. Pirkola, A.: Effects of Query Structure and Dict. Setups in Dict.-Based Cross-Lang. IR. SIGIR (1998)

24. Rath, T., Manmatha, R.: Word Image Matching Using Dynamic Time Warping. In: CVPR (2), vol. 521 (2003)
25. Rath, T., Manmatha, R., Lavrenko, V.: Search Engine for Historical Manuscript Images. In: SIGIR (2004)
26. Rath, T., Manmatha, R.: Word spotting for historical documents. In: IJDAR 2007 (2007)
27. Sanderson, M.: Word Sense Disambiguation and IR. PhD thesis, University of Glasgow (1997)
28. Sankar, P., Jawahar, C.: Prob. Reverse Annotation for Large Scale Image Retrieval. In: CVPR (2007)
29. Srihari, S.N., Ball, G.R., Srinivasan, H.: Versatile Search of Scanned Arabic Handwriting. In: Doermann, D., Jaeger, S. (eds.) SACH 2006. LNCS, vol. 4768, pp. 57–69. Springer, Heidelberg (2008)
30. Taghva, K., Borsack, J., Condit, A.: Effects of OCR errors on Ranking and Feedback using the Vector Space Model. Info. Processing and Management 32(3), 317–327 (1996)
31. Theodoridis, S., Koutroumbas, K.: Pattern Recognition, 3rd edn. Academic Press, London (2006)
32. Thoma, G., Ford, G.: Automated Data Entry System: Performance Issues. In: SPIE Conference on Document Recognition and Retrieval IX, pp. 181–190 (2002)
33. Tseng, Y., Oard, D.: Document Image Retrieval Techniques for Chinese. In: SDIUT, pp. 151–158 (2001)

Sketching Algorithms for Approximating Rank Correlations in Collaborative Filtering Systems

Yoram Bachrach[1], Ralf Herbrich[1], and Ely Porat[2]

[1] Microsoft Research Ltd., Cambridge, UK
[2] Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel

Abstract. Collaborative filtering (CF) shares information between users to provide each with recommendations. Previous work suggests using sketching techniques to handle massive data sets in CF systems, but only allows testing whether users have a high proportion of items they have both ranked. We show how to determine the *correlation* between the rankings of two users, using concise "sketches" of the rankings. The sketches allow approximating Kendall's Tau, a known rank correlation, with high accuracy ϵ and high confidence $1-\delta$. The required sketch size is logarithmic in the confidence and polynomial in the accuracy.

1 Introduction

Recommender provide a user with recommendations regarding information items she is likely to find interesting. These systems compare user profiles to reference characteristics. Sometimes these characteristics are obtained from the content of the item (in the *content based* approach), and sometimes from information regarding the tastes of other users, in the *collaborative filtering (CF)* approach.

We consider a CF domain, where each user *ranks* the items she examined. Consider Alice, who asks the CF system to give a prediction for a certain item. The CF system must search for users who have ranked many of the items Alice has ranked. Then, the system should consider their rankings, and decide whether these users' tastes are similar to Alice's. A naive way to do this is to store the complete item lists and rankings for each user. However, this requires storing an enormous amount of data.The work [2] proposed a sketching technique for computing the *proportional intersection* (PI) of the ranked item lists. Rather than storing the full information they suggested *very concise* descriptions of ranked item lists, called *sketches*. Give a target accuracy $\epsilon > 0$ and a target confidence δ, their method returned an approximation $\hat{x}$ to the actual PI x, such that with probability of at least $1-\delta$ the approximation is accurate enough, so $|\hat{x}-x| \leq \epsilon$. The major shortcoming of [2] is that *it did not allow computing a correlation grade between the rankings*. Even if there are many items ranked by both users, it is hard to construct a recommendation based solely on this information, as they may have given very different ratings these items. This work extends [2] and provides methods for computing the *correlation* between the rankings using *sketching techniques*. We construct an *extremely concise* representation of the

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 344–352, 2009.
© Springer-Verlag Berlin Heidelberg 2009

user's item rankings, called a *rank correlation sketch*. Our sketches are designed to approximate Kendall's Tau [8], a well known rank correlation grade, while maintaining an only a small fraction of the information.

Consider Alice and Bob, who have each examined and ranked a set of n items, giving the item liked most has a rank of 1, the second best has a rank of 2, and so on until the worst item with the rank n. A known statistic to measure the correspondence between two rankings is Kendall's Tau [8]. Given the rankings of Alice and Bob, and given two items, A and B, we call the items a *concordant pair* if Alice and Bob agree on their order (i.e. if both Alice and Bob prefer A over B or if both prefer B to A). When Alice and Bob disagree on these items they are called a *discordant pair*. Given two rankings, we denote by n_c the number of concordant pairs, and by n_d the number of discordant pairs. Every pair is either concordant or discordant, so $n_c = n - n_d$.

Definition 1. *Kendall's Tau of* r_a *and* r_b *is:* $\tau_{r_a,r_b} = \frac{n_c - n_d}{\frac{1}{2}n(n-1)}$

The total number of pairs is $\frac{1}{2}n(n-1)$, so $P(C) = \frac{n_c}{\frac{1}{2}n(n-1)}$ is the probability of a uniformly randomly chosen pair to be a concordant, and $P(D) = \frac{n_d}{\frac{1}{2}n(n-1)}$ is the probability for a discordant pair. Thus Kendall's Tau can be expressed as $\tau_{r_a,r_b} = p(C) - P(D) = P(C) - (1 - P(C)) = 2P(C) - 1$.

Consider the users of the CF system, $a_1, \ldots, a_m$. Each a_i has a ranking r_i of the items she had experience with. Under our model, we only maintain a sketch S_i of each ranking.

A sketching framework allows approximating Kendall's Tau τ_{r_i,r_j} between any two users, with a target confidence and accuracy.

Definition 2. *A rank correlation sketching framework with confidence* δ *and accuracy* ϵ *maintains only* $S_1, S_2, \ldots, S_m$*, and for any two users,* a_i *and* a_j*, allows computing* τ_{r_i,r_j} *with accuracy of at least* ϵ *and with confidence of at least* $1 - \delta$*. That is, the framework returns an approximation* $\hat{\tau_{i,j}}$ *for* τ_{r_i,r_j} *such that with probability of at least* $1 - \delta$ *we have* $|\tau_{r_i,r_j} - \hat{\tau_{i,j}}| \leq \epsilon$*.*

2 Sketches for Approximating Rank Correlation

Our sketching framework extends [2], so we first review that technique. Consider Alice and Bob, with the set C_1 of items that Alice has rated, and the set C_2 of items that Bob has rated, from the universe U of items, where $|C_1| = |C_2|$. Consider a sketch S_i that is the identity of a *single* item chosen uniformly at random from C_i. The probability of choosing the same item in S_1 and S_2 depends on $\frac{|C_1 \cap C_2|}{|C_1|}$, and is small. One insight comes from deciding to let the sketch S_i be the *minimal* item from C_i. If the minimal item in $C_1 \cup C_2$ is in $C_1 \cap C_2 = T$, we are guaranteed to find the item in $S_1 \cap S_2$. However, always using the *minimal* item always generates the same S_1, S_2. The methods in [2] overcome this by using min-wise independent hashes. Let H be a family of functions such that each $h \in H$ is a function $h : X \to Y$, where Y is completely ordered. We say

H is min-wise independent if, when randomly choosing $h \in H$, for any subset $C \subseteq X$, any $x \in C$ has an equal probability of being the minimal under h.

Definition 3. *H is min-wise independent, if for all $C \subseteq X$, for any $x \in C$, $Pr_{h \in H}[h(x) = min_{a \in C} h(a)] = \frac{1}{|C|}$.*

The work [7] constructs such families. The work in [2] uses them to build sketches for approximating the PI. In that work they use integers to define the identity of items in U (where $|U| = u$), so any subset of items $C \subseteq U$, is represented as a list of $|C|$ integers in $[u]$ ($[u]$ denoting $\{1, 2, \ldots, u\}$). They use a family H of min-wise independent functions from $[u]$ to $[n^2]$. Thus, although the domain is the huge universe of $[u]$ items, the hashed values are in the smaller range of $[n^2]$ items[1]. The methods of [2] consider users a_1, a_2, each with a list C_i of examined items, such that $|C_1| = |C_2|$. The sketches they propose approximate the PI between the two users, $p_{1,2} = \frac{|C_1 \cap C_2|}{|C_1|} = \frac{|C_1 \cap C_2|}{|C_2|}$. These sketches are based on randomly choosing hashes from H. Given $h \in H$, we can apply h on all the integers in C_1 and examine the minimal integer we get, $m_1^h = min_{x \in C_1} h(x)$. We can do the same to C_2 and examine $m_2^h = min_{x \in C_2} h(x)$. The following Lemma is proved in [2].

Lemma 1. *$Pr_{h \in H}[m_1^h = m_2^h] = \frac{p_{1,2}}{2 - p_{1,2}}$.*

We refer to the the sketches used by [2] as *item sketches*. They are defined as follows. Let $v_k = \langle h_1, h_2, \ldots, h_k \rangle$ be a tuple of k randomly chosen functions from the min-wise independent family H, and let C_i be the set of items that user a_i has examined. Denote the minimal item in C_i under h_j as $m_i^{h_j} = min_{x \in C_i} h_j(x)$.

Definition 4 (Item Sketches). *The H_k sketch of C_i, $S(C_i)$, is the list of minimal items in C_i under the k randomly chosen functions from h: $S^k(C_i) = (m_i^{h_1}, m_i^{h_2}, \ldots, m_i^{h_k})$.*

There are several key observations regarding item sketches. First, since H is min-wise independent, each sketch $S(C_i)$ on its own is a list of k random items from C_i (after applying a hash function on each item). Second, due to Lemma 1, randomly choosing a function $h \in H$ and testing whether $m_1^h = m_2^h$ is a Bernoulli trial, with success probability of $\alpha = \frac{p_{a,b}}{2 - p_{a,b}}$. We denote by X_i the random variable of the Bernoulli trial using hash h_i, so $X_i = 1$ if $m_a^{h_i} = m_b^{h_i}$, and $X_i = 0$ otherwise. Given an item sketch of k hashes, we get k such Bernoulli trials, $X_1, \ldots, X_k$, and can estimate $\alpha = \frac{p_{a,b}}{2 - p_{a,b}}$ as $\frac{\sum_{i=1}^{k} X_i}{k}$. Since $\alpha = \frac{p_{a,b}}{2 - p_{a,b}}$, we have $p_{a,b} = \frac{2\alpha}{1+\alpha}$, so given an estimate $\hat{\alpha}$ for α, we can estimate $p_{a,b}$ as $\hat{p_{a,b}} = \frac{2\hat{\alpha}}{1+\hat{\alpha}}$. The work [2] shows that to approximate the PI $p_{a,b}$ within accuracy ϵ and confidence $1 - \delta$, it is enough to use $k = \frac{\ln \frac{2}{\delta}}{2 \frac{\epsilon^2}{9}}$ hashes. The methods in [2] do not show how to compute the correlation between the rankings.

[1] The methods of [2] build on the results of [7], which show that using a range of n^2 integers mitigates the effect of collisions in the hashed values. Thus, the probability of two different items in $[u]$ to be mapped to the same value after applying the hash (a collision) is very small.

2.1 Rank Correlation Sketches

We now describe our method for constructing a rank correlation sketching framework. We first return to Alice and Bob. Now suppose we have a set of items I that *both* Alice and Bob have ranked, from the universe U of items. Denote by n the size of I, so $|I| = n$. We are interested in approximating τ_{r_a,r_b}, Kendall's Tau rank correlation between Alice's ranking of the items in I and Bob's ranking. We denote by n_c the number of concordant pairs, and by n_d the number of discordant pairs, so $\tau_{r_a,r_b} = \frac{n_c - n_d}{\frac{1}{2}n(n-1)}$. As noted in Section 1, the probability $P(C) = \frac{n_c}{\frac{1}{2}n(n-1)}$ is closely relate to Kendall's Tau, and $\tau_{r_a,r_b} = 2P(C) - 1$.

Lemma 2 (Approximating $P(C)$ and Kendall's Tau). *Approximating $P(C)$ with accuracy $\frac{\epsilon}{2}$ gives an approximation to Kendall's Tau with accuracy ϵ.*

Proof. We use the approximation $P(\hat{C})$ for $P(C)$ to approximate Kendall's Tau. Our approximation for τ_{r_a,r_b} is $\tau_{\hat{r_a},r_b} = 2P(\hat{C}) - 1$. If our error in our estimation of $P(C)$ is at most $\frac{\epsilon}{2}$, we have $|P(C) - P(\hat{C})| \leq \frac{\epsilon}{2}$, so $|\tau_{r_a,r_b} - \tau_{\hat{r_a},r_b}| = |2P(C) - 1 - (2P(\hat{C}) - 1)| = |2(P(C) - P(\hat{C}))| \leq 2 \cdot \frac{\epsilon}{2} = \epsilon$. Thus, to approximate Kendall's Tau with accuracy ϵ it is enough to approximate $P(C)$ with accuracy $\frac{\epsilon}{2}$.

We now consider a pair of items chosen uniformly at random from I, $x, y \in I$. Given Alice's and Bob's rankings, we can test whether this is a concordant pair. This is a Bernoulli trial, with a success probability of $P(C)$. We define the random variable of this Bernoulli trial as: $X_1 = \begin{cases} 1 & \text{if } x, y \text{ is a concordant pair} \\ 0 & \text{if } x, y \text{ is a discordant pair} \end{cases}$

Given k_p such pairs, we have a sequence of k_p such Bernoulli trials, $X_1, \ldots, X_{k_p}$. Let X be the number of successes in this series of Bernoulli trials, $X = \sum_{j=1}^{k_p} X_j$. We have chosen the pairs uniformly at random, so the X_is are identical but independent. Thus X has the Binomial distribution $X \sim B(k, \alpha)$, and the *maximum likelihood estimator* for $P(C)$ is $P(\hat{C}) = \frac{X}{k_p}$. We now derive the required number of random item pairs required to approximate $P(C)$ with accuracy ϵ_c and confidence δ_c. To achieve the desired accuracy and confidence, the number of sampled pairs, k_p, must be large enough. We find the appropriate k_p by using Hoeffding's inequality [6].

Theorem 1 (Hoeffding's inequality). *Let $X_1, \ldots, X_n$ be independent random variables, where all X_i are bounded so that $X_i \in [a_i, b_i]$, and let $X = \sum_{i=1}^{n} X_i$. Then the following inequality holds:* $\Pr(|X - \mathrm{E}[X]| \geq n\epsilon) \leq 2\exp\left(-\frac{2\,n^2\,\epsilon^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right)$.

Let $X_1, \ldots, X_{k_p}$ be the series k_p of Bernoulli trials, as defined above. Again, let $X = \sum_{j=1}^{k_p} X_j$, and take $P(\hat{C}) = \frac{X}{k_p}$ as an estimator for $P(C)$. All X_i are either 0 or 1 (so they are bounded between these values), and $\mathrm{E}[X] = k_p \cdot P(C)$. Thus, the following holds: $\Pr(|X - k_p P(C)| \geq k\epsilon_c) \leq 2e^{-2\,k_p\,\epsilon_c^2}$. Therefore the following

also holds: $\Pr(|P(\hat{C}) - P(C)| \geq \epsilon_c) \leq 2e^{-2\,k_p\,\epsilon_c^2}$. We now extract the number of pairs required so that this probability is below some required confidence level δ_c.

Theorem 2 (Pair Samples for Approximating $P(C)$). *A confidence interval for $P(C)$ is $[P(\hat{C}) - \epsilon_c, P(\hat{C}) + \epsilon_c]$. This interval holds the correct $P(C)$ with probability of at least $1 - \delta_c$. The required number of pair samples to perform this is $k_c = \frac{\ln \frac{2}{\delta_c}}{2\,\epsilon_c^2}$.*

Proof. We use Hoeffding's inequality to bound the error below our target confidence level δ_c, and get: $Pr(|P(\hat{C}) - P(C)| \geq \epsilon_c) \leq 2\,e^{-2\,k_c\,\epsilon_c^2} \leq \delta_c$. We extract ϵ_c and k_c: $-2\,k_c\,\epsilon_c^2 \leq \ln \frac{\delta_c}{2}$. Equivalently: $\epsilon_c^2 \geq -\frac{\ln \frac{\delta_c}{2}}{2\,k}$. Finally we get the following: $\epsilon_c \geq \sqrt{\frac{1}{2k_c} \ln \frac{2}{\delta_c}}$ and $k_c \geq \frac{\ln \frac{2}{\delta_C}}{2\,\epsilon_c^2}$.

The required number of pairs in Theorem 2 considered approximating $P(C)$ and not Kendall's Tau. However, due to Lemma 2 we get the following corollary.

Corollary 1 (Pair Samples for Approximating Kendall's Tau). *The following is an approximation for Kendall's Tau: $\hat{\tau_{r_a,r_b}} = 2P(\hat{C}) - 1$. In order for it to have accuracy ϵ_t and confidence δ_t the required number of random pairs samples is $k_t = \frac{2\ln \frac{2}{\delta_t}}{\epsilon_t^2}$.*

2.2 From Item Sketches to Rank Correlation Sketches

We now augment the sketches of [2] to approximate Kendall's Tau. The PI sketches of [2] approximate the PI. When the CF system attempts to provide Alice (with items C_a) with a recommendation, it filters out users who do not have a high enough PI with her, so only users with a PI exceeding a threshold, p^*, remain. Consider a candidate, Bob (with item set C_b), where the PI of Alice and Bob is $p_{a,b}$. By definition of the PI, $p_{a,b} = \frac{|C_a \cap C_b|}{|C_a|} = \frac{|C_a \cap C_b|}{|C_b|}$[2], and since Bob has passed the filtering stage we have $p_{a,b} \geq p^*$. The item sketch from Definition 4 randomly chooses k hashes from H, and lists the minimal items under these k hashes. By definition of H as a min-wise independent family, for any user's set of items C, any item has an equal probability of being minimal under the hash, so $Pr_{h \in H}[h(x) = min_{a \in C} h(a)] = \frac{1}{|C|}$. Let $h \in H$ be a randomly chosen hash function from H. We denote the minimal item in C_i under h as $m_i^h = min_{x \in C_i} h(x)$. We show that if Alice and Bob have a PI of at least p^*, the probability of having the same value at each sketch location is at least $\frac{p^*}{2-p^*}$.

Lemma 3 (Probability Of The Same Item Appearing In Two Sketches). *Let Alice and Bob be two users with a PI of at least p^*, Alice with item set C_a and Bob with item set C_b. Then $Pr_{h \in H}[m_a^h = m_b^h] \geq \frac{p^*}{2-p^*}$ (i.e. the probability of Alice and Bob having* same *minimal item under h is at least $\frac{p^*}{2-p^*}$).*

[2] Note that we are still assuming the same size of item set per user.

Proof. We denote the PI of Alice and Bob as $p_{a,b} \geq p^*$. Due to Lemma 1 we have $Pr_{h \in H}[m_a^h = m_b^h] = \frac{p_{a,b}}{2-p_{a,b}}$, and since $f(x) = \frac{x}{2-x}$ is monotonically increasing in the domain $[0,1]$ we have $Pr_{h \in H}[m_a^h = m_b^h] \geq \frac{p^*}{2-p^*}$.

Consider Alice and Bob, with a PI of at least p^*. Lemma 3 states that any location i has a probability of at least $p_s = Pr_{h_i \in H}[m_a^{h_i} = m_b^{h_i}] \geq \frac{p^*}{2-p^*}$ to contain the same value in Alice's sketch and in Bob's sketch. Since the range of the hashes in H is $[n^2]$ (where n is the number of items examined by each user), having the same minimal item under h, $m_a^{h_i} = m_b^{h_i}$, indicates with high probability that this is the *same* item, so $|\{x \in C_a | h_i(x) = m_a^{h_i}\}| = |\{y \in C_b | h_i(y) = m_b^{h_i}\}| = 1$, and both C_a and C_b contain only one item x (so $x \in C_a$ and $x \in C_b$) such that $h_i(x) = m_a^{h_i} = m_b^{h_i}$.

Let h_i be the hash for the i'th location in the item sketch. The augmenting part of the sketch includes the rank of the item that is minimal under h. We denote the ranking of user a over the items in C_a as r_a, so r_a maps items from C_a to their rank in $[n]$ (where $|C_a| = n$). Thus $r_a : C_a \rightarrow [n]$ is reversible. We randomly choose a hash for each sketch location. Given the hash h_i for location i, we consider the items who are minimal under h_i, i.e. $M = \{x \in C_a | h_i(x) = m_a^{h_i}\}$. If $|M| = 1$ we denote $M = \{m\}$, and denote $g_a^i = r_a(m)$. If $|M| \geq 1$, which occurs with a very low probability, we denote m' to be the minimal item in M (under the original ordering, not under h_i), and denote $g_a^i = r_a(m')$. The sketch for user a in the i'th location contains the minimal item in C_a under h_i, and its ranking in user a's eyes.

Definition 5 (Rank Correlation Sketches). *The H_k rank correlation sketch of C_a, $S^k(C_a)$, contains the both the item sketch and the rank sketch. As before, the item sketch is just the list of minimal items in C_a under the k randomly chosen hashes, so $S^k_{items}(C_a) = (m_a^{h_1}, m_a^{h_2}, \ldots, m_a^{h_k})$, and the rank sketch contains the ranks of these items, so $S^k_{ranks}(C_a) = (g_a^1, \ldots g_a^n)$. The rank correlation sketch is simply the concatenation of these two sketches.*

Consider two locations i and j where the item sketch for both Alice and Bob is the same, i.e where $m_a^{h_i} = m_b^{h_i}$ and $m_a^{h_j} = m_b^{h_j}$. Each such location is called a *sketch collision*. Given two such collisions, with high probability the ranking sketch at these two locations refers to the same items, i.e. there are two items x, y such that $x, y \in C_a$ and $x, y \in C_b$, and such that $g_a^i = r_a(x), g_b^i = r_b(x), g_a^j = r_a(y), g_b^j = r_b(y)$. Since any item has an equal probability to be minimal under a random hash $h \in H$ (as H is min-wise independent), the sketches in these locations provide us with Alice's and Bob's rankings for a pair of items chosen uniformly at random from $C_a \cap C_b$. Corollary 1 gives the required number of pairs to approximate Kendall's Tau, but each pair requires two independent collisions[3]. Thus, approximating Kendall's Tau is reduced to finding a sketch that would have the required number of collisions with high probability.

[3] Notice that given m sketch collisions we can generate $\frac{m(m-1)}{2}$ pairs, but these pairs would not be independent.

2.3 Collisions and Sketch Size

Consider Alice, who seeks a recommendation from the CF system. The CF system has a PI threshold p^*, and selects only candidates who have a higher PI with her. As shown in [2], in order to compute the PI with accuracy ϵ_i and confidence δ_p, it is enough to use a sketch based on $k_p = \frac{\ln \frac{2}{\delta_p}}{2\frac{\epsilon_p^2}{9}}$ hashes. After filtering out candidates with too low a PI, the CF system remains with candidates, and computes Kendall's Tau for each of them, with accuracy ϵ_t and confidence $1-\delta_t$.

Lemma 3 shows that the probability of a collision in each location is at least $p = \frac{p^*}{2-p^*}$. Thus each location is a Bernoulli trial, with success probability of at least p (success being a collision). Theorem 1 shows that approximating Kendall's Tau with accuracy ϵ_t and confidence δ_t requires $2k_t = \frac{4\ln\frac{2}{\delta_t}}{\epsilon_t^2}$ sketch collisions. We determine the size of the sketch needed to have such a required number of collisions with probability of at least $1-\delta_c$. Given a sketch based on m hashes, the number of collisions X has the Binomial distribution with parameters m, p. We require $k = 2k_t$ collisions, and thus are interested in the cumulative distribution function $F(k, m, p) = P(X \leq k) = \sum i = 0^k \binom{m}{i} p^i (1-p)^{n-i}$. We find a sketch size m that is high enough that $F(k, m, p)$ is below our confidence level δ_c, using the following result from [3]:

Theorem 3 (Binomial Distribution Tail Bound)

$$F(k, m, p) \leq \exp\left(-2\frac{(mp-k)^2}{n}\right)$$

Theorem 4 (Rank Correlation Sketch Size). *A rank correlation sketching framework for users with PI of at least p^*, where $p = \frac{p^*}{2-p^*}$, requires sketch size of $m \geq \frac{k}{p} + \frac{\ln\frac{1}{\delta_c}}{4p^2}(1+3\sqrt{k})$, where $k = 2k_t = \frac{4\ln\frac{2}{\delta_t}}{\epsilon_t^2}$.*

Proof. We require a sketch size m such that $F(k, m, p) \leq \exp\left(-2\frac{(mp-k)^2}{n}\right) \leq \delta_c$. Thus we require $\frac{-2(mp-k)^2}{n} \leq \ln \delta_c$, or that $\frac{(mp-k)^2}{m} \geq \frac{\ln\frac{1}{\delta_c}}{2}$. We denote $d = \frac{\ln\frac{1}{\delta_c}}{2}$. The requirement is thus that $p^2m^2 + (-2pk-d)m + k^2 \geq 0$. Solving the quadratic equation (and taking the bigger solution) we get $m \geq \frac{2pk+d+\sqrt{4pkd+d^2}}{2p^2}$ or that $m \geq \frac{k}{p} + \frac{d}{2p^2} + \frac{\sqrt{4pkd+d^2}}{2p^2}$. An even strong requirement is that $m \geq \frac{k}{p} + \frac{d}{2p^2}(1 + \sqrt{4k+1})$, or even that $m \geq \frac{k}{p} + \frac{d}{2p^2}(1+\sqrt{9k}) = \frac{k}{p} + \frac{d}{2p^2}(1+3\sqrt{k})$. We finally get that the requirement is $m \geq \frac{k}{p} + \frac{\ln\frac{1}{\delta_c}}{4p^2}(1+3\sqrt{k})$.

Thus the sketch size is polynomial in the accuracy, and logarithmic in the confidence[4].

[4] There are two confidence levels, δ_t the maximal probability of mis-approximating Kendall's Tau, and δ_c, the maximal probability of not having enough sketch collisions. From the union bound, the probability of having a bad approximation is at most $\delta_c + \delta_t$, and the sketch size is logarithmic in both.

3 Related Work

There are many examples of CF systems, such as GroupLens [9] and Ringo [11]. [9] uses the Pearson correlation, while [11] uses other measures. This paper tackles the problem of handling the massive data sets in CF systems. We suggested sketching to approximate rank correlations. One example of a sketching technique is [4]. We extend [2] to compute rank correlations, using a min-wise independent family of hashes. Such families were treated in [7]. Our methods fits in the Locally Sensitive Hashing (LSH) [5] framework, but is specialized for CF systems. Similar approach are Random Projections [1] and Semantic/Spectral Hashing [10,12].

4 Conclusion

A challenge in CF systems is handling huge amounts of information. We have suggested a sketching approach to *approximate* the rank correlation with a given accuracy and confidence. The sketch size is logarithmic in the confidence, and polynomial in the accuracy. There are many directions for future research. Our methods only allow computing Kendall's Tau and not other rank correlations, such as Spearman's Rho. Also, we assume a complete ranking over items, and do not allow for ties. Another shortcoming of our analysis here is that it is only theoretical. It would be desirable to test these methods on real data sets.

References

1. Achlioptas, D.: Database-friendly random projections: Johnson-Lindenstrauss with binary coins. JCSS 66 (2003)
2. Bachrach, Y., Porat, E., Rosenschein, J.S.: Sketching techniques for collaborative filtering. In: IJCAI 2009, Pasadena, California (July 2009) (to appear)
3. Chung, K.L.: Elementary Probability Theory with Stochastic Processes. Springer, Heidelberg (1974)
4. Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M.: An approximate L1-difference algorithm for massive data streams. SIAM J. Comput. 32(1), 131–151 (2002)
5. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB: International Conference on Very Large Data Bases, Morgan Kaufmann Publishers, San Francisco (1999)
6. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301), 13–30 (1963)
7. Indyk, P.: A small approximately min-wise independent family of hash functions. Journal of Algorithms 38(1), 84–90 (2001)
8. Kendall, M.G.: A new measure of rank correlation. Biometrika 30, 81–93 (1938)

9. Resnick, P., Iacovou, N., Suchak, M., Bergstorm, P., Riedl, J.: Grouplens: An open architecture for collaborative filtering of netnews. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 175–186. ACM Press, New York (1994)
10. Salakhutdinov, R., Hinton, G.: Semantic hashing. In: International Journal of Approximate Reasoning (December 2008)
11. Shardan, U., Maes, P.: Social information filtering: Algorithms for automating "word of mouth". In: ACM CHI 1995, vol. 1, pp. 210–217 (1995)
12. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: Advances in Neural Processing Systems (2008)

Author Index

Adiego, Joaquín 114
Adsett, Connie R. 174
Akutsu, Tatsuya 7
Allan, James 143
Amir, Amihood 234, 254

Bachrach, Yoram 344
Baeza-Yates, Ricardo 206
Balasubramanian, Niranjan 143
Bast, Hannah 194
Bianco, Mauro 222
Boucher, Christina 243
Brisaboa, Nieves R. 18, 114, 122

Celikik, Marjan 194
Clifford, Raphael 295
Craveiro, Olga 156

Darwish, Kareem 334

El-Saban, Motaz 334

Feigenblat, Guy 266
Ferrarotti, Flavio 310
Feuerstein, Esteban 206
Fukagawa, Daiji 7

Gagie, Travis 1
Geraldo, André Pinto 165
Gil-Costa, Veronica 206
Gog, Simon 51
Gonçalves, Marcos A. 165
Gotthilf, Zvi 277

Harrow, Aram W. 295
Herbrich, Ralf 344
Hon, Wing-Kai 75, 182

Ilie, Lucian 302
Inagaki, Kazumasa 102
Itzhaki, Ofra 266

Joho, Hideo 322

Ladra, Susana 18, 122
Lalmas, Mounia 322
Lam, Tak-Wah 39
Landau, Gad M. 234
Lee, Taehyung 31
Lewenstein, Moshe 277

Macedo, Joaquim 156
Madeira, Henrique 156
Magdy, Walid 334
Marchand, Yannick 174
Marin, Mauricio 206, 310
Martínez-Prieto, Miguel A. 114
Mendoza, Marcelo 131, 310
Mizrahi, Michel 206
Moreira, Viviane P. 165

Na, Joong Chae 31, 234
Navarro, Gonzalo 18, 122, 214

Ohlebusch, Enno 51
Okanohara, Daisuke 90

Parienty, Haim 254
Park, Heejin 234
Park, Kunsoo 31, 234
Pizzi, Cinzia 222
Popa, Alexandru 295
Porat, Ely 266, 285, 344
Puglisi, Simon J. 1

Sach, Benjamin 295
Sadakane, Kunihiko 90
Salmela, Leena 214
Sánchez-Martínez, Felipe 114
Shah, Rahul 75, 182
Shiftan, Ariel 285
Sim, Jeong Seop 234
Sirén, Jouni 63
Sushmita, Shanu 322

Takasu, Atsuhiro 7
Tam, Alan 39
Thankachan, Sharma V. 75

Tinta, Liviu 302
Tomizawa, Yoshihiro 102
Turpin, Andrew 1

Vitter, Jeffrey Scott 75

Wu, Edward 39
Wu, Shih-Bin 182

Yiu, Siu-Ming 39
Yokoo, Hidetoshi 102

Zamora, Juan 131