Edgar Chavez
Stefano Lonardi (Eds.)

LNCS 6393

String Processing and Information Retrieval

17th International Symposium, SPIRE 2010
Los Cabos, Mexico, October 2010
Proceedings

Springer

Lecture Notes in Computer Science 6393

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

Edgar Chavez Stefano Lonardi (Eds.)

String Processing and Information Retrieval

17th International Symposium, SPIRE 2010
Los Cabos, Mexico, October 11-13, 2010
Proceedings

Springer

Volume Editors

Edgar Chavez
Universidad Michoacana
School of Physics and Mathematics
Edificio "B", Ciudad Universitaria, Morelia, Mich., 5800, Mexico
E-mail: elchavez@umich.mx

Stefano Lonardi
University of California
Dept. of Computer Science and Engineering
Engineering Building II, Riverside, CA 92521, USA
E-mail: stelo@cs.ucr.edu

Library of Congress Control Number: 2010935846

CR Subject Classification (1998): H.3, J.3, H.2.8, I.5, I.2.7, H.5.1

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

ISSN 0302-9743
ISBN-10 3-642-16320-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16320-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the papers presented at the 17th International Symposium on String Processing and Information Retrieval (SPIRE 2010), held October 11–13, 2010 in Los Cabos, Mexico.

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 talk 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; querying and interface design), 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 109 submissions written by authors from 30 different countries. Each submission was reviewed by at least three reviewers, with a maximum of five reviews for particularly challenging papers. The Program Committee accepted 39 papers (corresponding to $\approx$35% acceptance rate): 26 long papers and 13 short papers. In addition to these presentations, SPIRE 2010 also featured invited talks by Gonzalo Navarro (Universidad de Chile) and Mark Najork (Microsoft Research, USA).

We are especially thankful to the members of the Program Committee, who provided us with thorough and timely reviews. Every PC member completed all their reviews on a very tight schedule. We wish to thank the SPIRE Steering Committee, via their coordinator Ricardo Baeza-Yates and the editorial office staff at Springer. The student volunteer staff led by Francisco Claude was especially helpful with attending local matters for the conference. The students of the Information Retrieval course of CICESE helped in gathering all the kits for attendees. Yahoo! Research had generously provided sponsorship for partial support for accommodation costs for student volunteers. We finally thank Universidad Michoacana and University of California for donating the time of the organizers.

September 2010

Edgar Chavez
Stefano Lonardi

SPIRE 2010 Organization

Organizing Institution

SPIRE 2010 was organized by the Department of Computer Science of the CICESE Research Center in Ensenada, México and the Universidad Michoacana in Morelia, México.

Program Committee Chairs

Edgar Chávez	Universidad Michoacana/CICESE, Mexico
Stefano Lonardi	University of California Riverside, USA

Publicity and Local Organization

Jeremy Barbay	University of Chile, Chile
Francisco Claude	University of Waterloo, Canada
Marco Patella	University of Bologna, Italy

Program Committee

Amihood Amir	Bar-Ilan University, Israel and Johns Hopkins University, USA
Alberto Apostolico	Georgia Tech, USA and University of Padua, Italy
Mikhail Atallah	Purdue University, USA
Ricardo Baeza-Yates	Yahoo! Research, Spain and University of Chile, Chile
Alvaro Barreiro	University of A Coruña, Spain
Paolo Boldi	University of Milan, Italy
Carlos Castillo	Yahoo! Research, Spain
Edgar Chavez	Universidad Michoacana, Mexico
Fabio Crestani	University of Lugano, Switzerland
Maxime Crochemore	Université Paris-Est, France
Bruce Croft	University of Massachusetts, USA
Andrea Esuli	CNR, Italy
Martin Farach-Colton	Rutgers University, USA
Dan Gusfield	UC Davis, USA
Gregory Kucherov	CNRS, France
Stefano Lonardi	UC Riverside, USA
Alex Lopez-Ortiz	University of Waterloo, Canada
Giovanni Manzini	University of East Piedmont, Italy

Veli Mäkinen	University of Helsinki, Finland
Alistair Moffat	Melbourne University, Australia
Costas Iliopoulos	King's College London, UK
Moshe Lewenstein	Bar-Ilan University, Israel
Massimo Melucci	University of Padua, Italy
Ian Munro	University of Waterloo, Canada
Thierry Lecroq	University of Rouen, France
Gonzalo Navarro	University of Chile, Chile
Vibhu Mittal	Google, USA
Tao Jiang	UC Riverside, USA
Horacio Rodriguez	Politècnica de Catalunya, Spain
Kunihiko Sadakane	Kyushu University, Japan
Marie-France Sagot	INRIA, France
Cenk Sahinalp	Simon Fraser University, Canada
Fabrizio Silvestri	CNR, Italy
Steven Skiena	Stony Brook University, USA
Jens Stoye	Bielefeld University, Germany
Gabriel Valiente	UPC, Spain
Nivio Ziviani	Federal University of Minas Gerais, Brazil
Michal Ziv-Ukelson	Tel Aviv University, Israel

Additional Reviewers

Saïd Abdeddaïm	David Fernandes	Roman Kolpakov
Margareta Ackerman	Bruno Fonseca	Tsvi Kopelowitz
Eneko Agirre	Frantisek Franek	Sebastian Kreft
Jussara Almeida	Robert Fraser	Fernando Krell
Diego Arroyuelo	Kimmo Fredriksson	Juha Kärkkäinen
Abdullah N Arslan	Yalena Frid	Mounia Lalmas
Stefano Baccianella	Travis Gagie	Fumei Lam
Golnaz Badkobeh	Shima Gerani	Shir Landau
Roi Blanco	Wolfgang Gerlach	Arnaud Lefebvre
Jochen Blom	Marcos Gonçalves	Yury Lifshits
Fabiano Botelho	Rob Gysel	David Loker
Mark Carman	Faraz Hach	Violetta Lonati
Supaporn Chairungsee	Iman Hajirasouliha	David Losada
Kaichun Chang	Meng He	Parvaz Mahdabi
Francisco Claude	Robert Homann	Diego Marcheggiani
J.C. Clemente Litran	Peter Husemann	Ilya Markov
Colin Cooper	Heikki Hyyrö	Guilherme Menezes
Marco Cristo	Giacomo Inches	Patrick Nicholson
Rodrigo Cánovas	Katharina Jahn	Javier Parapar
Phuong Dao	Hossein Jowhari	Solon Pissis
Simone Faro	Shahin Kamali	Cinzia Pizzi
Arash Farzan	Mostafa Keikha	Elise Prieur

Simon Puglisi
Jakub Radoszewski
Tomasz Radzik
Mathieu Raffinot
Paolo Ribeca
Jairo Rocha
Daniil Ryabko
Mert Saglam
Alejandro Salinger
Mikaël Salson
Dennis Shasha
Tetsuo Shibuya
Jouni Sirén
Tatiana Starikovskaya
Kristian Stevens
German Tischler
Dekel Tsur
Adriano Veloso
Rossano Venturini
Sebastiano Vigna
Niko Välimäki
Anthony Wirth
Shou-Jun Xu
Deniz Yorukoglu

Steering Committee

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

Venues

SPIRE 2010 was the 17th edition of the Symposium on String Processing and Information Retrieval. The first four events concentrated mainly on string processing and were held in South America under the title South American Workshop on String Processing (WSP) in 1993, 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	2003	Manaus, Brazil
1995	Valparaíso, Chile	2004	Padua, Italy
1996	Recife, Brazil	2005	Buenos Aires, Argentina
1997	Valparaíso, Chile	2006	Glasgow, UK
1998	Santa Cruz de la Sierra, Bolivia	2007	Santiago, Chile
1999	Cancún, Mexico	2008	Melbourne, Australia
2000	A Coruña, Spain	2009	Saariselkä, Finland
2001	Laguna de San Rafael, Chile	2010	Los Cabos, Mexico
2002	Lisbon, Portugal		

Sponsoring Institutions

Yahoo! Research
Universidad Michoacana, México
CONACyT, México

Table of Contents

Crowdsourcing and Recommendation

Indexes and Compressed Indexes

Theory

String Algorithms I

Compression

Querying and Search User Experience

String Algorithms II

Document Analysis and Comparison

Compressed Indexes

String Matching

Erratum

Querying the Web Graph
(Invited Talk)

Marc Najork

Microsoft Research, 1065 La Avenida, Mountain View, CA, USA
najork@microsoft.com
http://research.microsoft.com/~najork/

Abstract. This paper focuses on using hyperlinks in the ranking of web search results. We give a brief overview of the vast body of work in the area; we provide a quantitative comparison of the different features; we sketch how link-based ranking features can be implemented in large-scale search engines; and we identify promising avenues for future research.

Keywords: Web graph, link analysis, ranking, PageRank, HITS, SALSA.

1 The Ranking Problem

One of the fundamental problems of information retrieval is the ranking problem: given a corpus of documents and a query reflecting a user's information need, and having drawn from the corpus all the "result" documents that satisfy the query, order the results by decreasing relevance with respect to the user's information need. The aim of ranking algorithms is to maximize the utility of the result list to the user; or (more subjectively) to maximize the user's satisfaction.

Ranking algorithms deployed in commercial search engines typically draw on a multitude of individual features, where each feature manifests itself as a numerical score, and combine these features in some way, e.g. by weighted linear combination. Features can be classified across many possible dimensions. Figure 1 shows one possible taxonomy incorporating two dimensions: when a feature is computed, and what it is based on. Query-dependent or "dynamic"

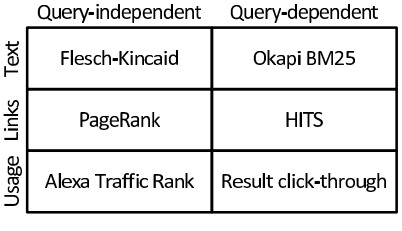


Fig. 1. A two-dimensional classification of ranking features

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 1–12, 2010.

features take the query into account and thus can only be computed at query time, whereas query-independent or "static" features do not rely on the query and thus can be computed ahead of time. Loosely speaking, dynamic features estimate the relevance of a document with respect to a query, while static features estimate its general quality.

Much of the classic research in information retrieval focused on small- to medium-sized curated corpora of high-quality documents, and assumed that queries are issued by trained library scientists. Consequently, early ranking algorithms leveraged textual features, based on the words or "terms" contained in documents and queries. Examples of query-independent textual features are readability measures such as Flesch-Kincaid [15]; examples of query-dependent textual features include BM25 [30]. As information retrieval broadened to web search, some of these assumptions became less tenable (the web is not curated; pages are more variable in length and quality; and users of web search engines are less experienced and tend to frame fairly short queries), but the fact that documents are interconnected by hyperlinks made new features available. Examples of query-independent link-based features include PageRank [28]; examples of query-dependent link-based features include HITS [16] and its many descendants. Finally, the explosive growth in the popularity of the web itself and of commercial web search engines generates an enormous amount of data on user activity both on search engines and the web at large, which can be harnessed into usage-based features. An example of a query-independent usage-based feature is Alexa Traffic Rank [2]; an example of a query-dependent usage-based feature is result click-through rate [31].

In the remainder of this paper, we will focus on link-based features. However, it is worth pointing out that all features – whether text-, link- or usage-based – are generated by human activity: by authors creating documents and endorsing other authors' documents through hyperlinks, and by users surfing the web, framing queries, and clicking on results. So, the ranking problem is about identifying features generated by observable human activities that are well-correlated with human satisfaction with the ordering (or more generally the selection and presentation) of the results of a query. In short, at its very core ranking is neither a mathematical nor an algorithmic problem, but a social one, concerned with the behaviors and interactions of people. Given that search is ultimately a social activity and that so far we are not not able to model human behavior well enough to accurately predict the effectiveness of any new ranking algorithm, the importance of experimental studies cannot be overemphasized.

2 Using Hyperlinks to Rank Web Search Results

Marchiori was the first to propose leveraging hyperlinks for ranking web search results. Specifically, he suggested computing a conventional test-based score for the result pages of a query, and then to propagate these scores (appropriately attenuated) to neighboring pages in the web graph [20]. This work inspired other researchers to consider hyperlinks as ranking features in their own right.

Hyperlinks can be viewed as endorsements made by web page authors of other web pages, and they can be classified along several dimensions. First, we can distinguish between "egotistic" and "altruistic" hyperlinks[1] – links that endorse pages in which the author has a vested interest vs. those that endorse other pages. It is very hard to identify altruistic links with high confidence, but there are some easy-to-determine indicators of egotistic links: links that point to pages on the same web server, on a web server in the same domain, on a server with the same IP address or in the same IP subnet, or registered to the same entity. Second, we can distinguish between "topical" and "templatic" links[2] – links that point to pages that are on the same topic as the linking page vs. those that point to topically dissimilar pages and often are part of a design template applied to the entire web site (for example, links to the site's privacy policy). In this world view, altruistic topical links provide the most meaningful endorsements.

2.1 PageRank

Page (together with Brin, Motwani and Winograd) proposed a purely link-based ranking function that went beyond the straightforward counting of endorsing hyperlinks in two respects: First, he suggested that the reputation of a page should be dependent on the reputation of its endorsing pages, and second, he suggested that the endorsement ability of a page should be divided among the pages it endorses. Furthermore, to prevent degenerate behavior in the presence of cycles and sink pages, he suggested affording each page a guaranteed minimal score. The resulting ranking function is the now-famous PageRank [28]. In order to define it formally, we will need some notation.

Web pages and hyperlinks induce a graph with vertex (page) set V and edge (link) set $E \subseteq V \times V$. We write $I(v, E)$ to denote the set of pages $\{u : (u, v) \in E\}$ that link to v, and $O(u, E)$ to denote the set of pages $\{v : (u, v) \in E\}$ that u links to. The teleportation vector $t : V \to [0, 1]$ controls the guaranteed minimum score of each page, the link weight matrix $W : V \times V \to [0, 1]$ is typically informed by the graph's adjacency matrix and controls the endorsement power of each hyperlink, and the damping factor λ balances the influence of either component. Using this notation, the PageRank $p(v)$ of a page v is defined in its most general form as the fixed point of the following recurrence relation:

$$p(v) = \lambda t(v) + (1 - \lambda) \sum_{u \in V} p(u) W(u, v)$$

or, using linear algebra notation, as:

$$p = \lambda t + (1 - \lambda) pW$$

Yet more abstractly, p is the principal eigenvector of the transition matrix T (i.e. the fixed point of the recurrence relation $p = pT$) defined as $T(u, v) =$

[1] Egotistic links are called "nepotistic" by Davison [9] and "intrinsic" by Kleinberg [16].

[2] Qi et al. refer to topical links as "qualified links" [29].

$\lambda t(v) + (1 - \lambda)W(u, v)$, assuming that p adds up to 1. In the classical definition of PageRank, $t(v) = \frac{1}{|V|}$ for all $v \in V$ (i.e. each page has the same guaranteed minimum score), and $W(u, v) = \frac{1}{|O(u,E)|}$ for all $v \in O(u, E)$ and 0 otherwise (i.e. page u uniformly endorses all pages it links to).

p, t and the rows of W are typically constrained to add up to 1, such that they can be viewed as probability distributions. That raises the question of how to deal with a "terminal" or "dangling" web page u that does not contain any hyperlinks, i.e. $\sum_{v \in V} W(u, v) = 0$. Page et al. proposed pruning such terminal vertices from the web graph (which may require multiple iterations, since pruning such terminal vertices and their incoming links may leave other vertices without outgoing links), computing PageRank on the core graph, and finally adding the pruned vertices and edges back in and propagating scores to them [28]. As it turns out, we can ignore the problem. To understand why, let us assume that we add a "phantom vertex" ϕ to the graph, plus a reflexive edge (ϕ, ϕ) and a phantom edge from each terminal to ϕ. The modified graph is free of terminal vertices. We set $t(\phi) = 0$, $W(u, \phi) = 1$ iff $O(u, E) = \emptyset$ and 0 otherwise, $W(\phi, \phi) = 1$, and $W(\phi, v) = 0$ for all $v \in V$. If we compute p on this graph using power iteration, $\|p\|_1$ (the ℓ_1-norm of p) will not change over iterations. Significantly, the $p(v)$ score of any non-phantom vertex is the same as it would be had we computed p using power iteration on the original graph while resigning ourselves that zero-sum rows in W will cause $\|p\|_1$ to shrink. The score mass that simply disappears when computing PageRank on the original graph can be found in the phantom node of the augmented graph. In other words, adding the phantom node and phantom edges to the graph is a useful intellectual device, but not actually needed for computing PageRank.

One very popular interpretation of the PageRank formula is the "random surfer model". In this model, a "surfer" traverses the web graph, visiting a new vertex at each transition. The surfer either "steps" with probability $1 - \lambda$ or "jumps" with probability λ. The destination vertex of a step is conditioned on the departure vertex, while that of a jump is not. Assuming that W is based on the web graph's adjacency matrix, taking a step means following a link. The surfer transitions from vertex u to vertex v with probability $T(u, v) = \lambda t(v) + (1 - \lambda)W(u, v)$. In this interpretation, $p(v)$ is the probability that the surfer is at vertex v at any given point in time. The random surfer model is very intuitive, and it relates PageRank to Markov random walks. Unfortunately, it also has led to a fair amount of confusion in the community: the random surfer model suggests that PageRank is modeling the web surfing behavior of users, i.e. of consumers of web content; the endorsement model described above suggests that PageRank is modeling the cross-reference behavior of authors, i.e. of producers of web content. We subscribe to the latter interpretation. Incidentally, commercial search engines have fairly direct ways of observing user behavior (e.g. through browser toolbars) and thus little need to model it.

Since PageRank is query-independent, it can be computed off-line. The PageRank vector p is typically computed using power iteration. Major commercial search engines maintain web graphs with tens of billions of vertices, and thus

compute p in a distributed setting. There are two standard approaches of doing this. The first approach uses data-parallel frameworks such as MapReduce [10], Hadoop [13] or DryadLINQ [32]. In this setting, the link weight matrix W, the teleportation vector t, and the old and new score vectors p reside on disk and are processed in a streaming fashion. Multiplying W and p corresponds to a join operation in relational algebra followed by a group-by, and the data streams to be joined must be sorted on the join key. W can be sorted ahead of time, while p has to be re-sorted every iteration. The second, ad-hoc approach partitions p, t and the rows of W such that all entries corresponding to pages on the same web server fall into the same partition. The new score vector is kept in memory (partitioned over many machines). W, t and the old score vector, which all reside on disk, are read in a streaming fashion, and fractions of the old scores are used to increment entries of the new score vector. These updates have a random-access pattern, but most will affect the local partition of the score array due to link locality – typically, most hyperlinks in a web page refer to other pages on the same web server. After every iteration, the new score vector is written to disk and becomes the old vector in the next iteration. The advantage of the ad-hoc approach over the MapReduce-approach is that it does not require any sorting; the two disadvantages are that it requires more engineering (e.g. to provide for fault-tolerance) and that computing scores for a larger graph requires more memory (typically by provisioning additional machines).

PageRank scores are computed off-line, and used at query-time to score results. As stated above, major search engines maintain corpora of tens of billions of documents. At the same time, they typically aim to answer queries within a fraction of a second [19]. In order to answer queries over such a large corpus and with such tight latency constraints, engines maintain all or at least the "hot" part of the index in main memory, partitioned across many machines, with each partition replicated multiple times across machines to achieve fault tolerance and increase throughput. The web corpus is commonly partitioned by document, i.e. all the terms of a given document are stored in the same sub-index. When a query arrives at the search engine, it is distributed to a set of index-serving machines that together hold the full index. Each machine finds the set of results in its sub-index that satisfy the query, scores these results using locally available features, and sends the top-k results to a result aggregation machine, which integrates them into the overall result set, possibly performing a more in-depth scoring. Since PageRank is a query-independent document score, the PageRank vector can be partitioned in the same way as the web corpus, and index-serving nodes can determine the PageRank scores of results using local table lookups. In other words, the query-time portion of PageRank is both extremely efficient and extremely scalable.

PageRank's elegance and intuitiveness, combined with the fact that it was credited for much of Google's extraordinary success, led to a plethora of research. Broadly speaking, this research falls into four classes: PageRank's mathematical properties (e.g. [17]); variations of the PageRank formula (e.g. [3]); computing PageRank efficiently (e.g. [14,21]); and adversarial attacks against PageRank.

Table 1. Effectiveness of various ranking features, in isolation

Feature	Class	NDCG@10	Source
PageRank	link/q-ind	0.092	[23]
HITS	link/q-dep	0.104	[23]
inter-domain indegree	link/q-ind	0.106	[23]
SS-SALSA-3	link/q-dep	0.153	[25]
SALSA	link/q-dep	0.158	[24]
SALSA-SETR	link/q-dep	0.196	[26]
BM25F	text+link/q-dep	0.221	[23]

Alas, there is a paucity in experimental validations of its effectiveness as a ranking function. Part of this is due to the fact that it is hard to assemble a large web corpus, and expensive to compile human judgments required in Cranfield-style evaluations of ranking effectiveness. Early studies tried to overcome these obstacles by leveraging existing search engines to obtain linkage information [4]; more recent work by our group performed substantial web crawls and used a test set compiled by a commercial search engine [23]. To our surprise, we found that PageRank in its classic form (with uniform teleportation and link bias, and $\lambda = 0.15$) is even less effective than simply counting altruistic (inter-domain) hyperlinks. Table 1 summarizes the results of these studies, which were all based on the same data sets.

At first glance, it is surprising that PageRank does not outperform inter-domain link-counting – after all, link-counting ignores the reputation of the linking page, and considers only the immediate neighborhood of each web page. We believe that there are two reasons why classic PageRank fares so poorly: First, it treats all links the same; it ignores whether they are egotistic or altruistic, topical or templatic. Inter-domain link-counting on the other hand will discard links that are obviously egotistic. Second, being credited as an important factor in Google's ranking algorithm, PageRank is under attack by legions of search engine optimizers. In its classic formulation, each page receives a guaranteed minimum score, so the obvious attack is to publish millions of pages that all endorse a single landing page, which will receive the (slightly dampened) sum of the scores of the endorsing pages. The key enabler for spammers is that web pages can be automatically generated on the fly, so publishing even a very large collection of pages is virtually free. This technique is known as link spam or link bombs, and there are several studies on the most effective shape of such link bombs [1,12].

Given that the key enabler of link spam is the low cost of publishing, the appropriate countermeasure is to correlate the teleportation vector with a feature that has actual economic cost. Examples of features that have non-zero cost are domain names (since it requires payment to a registrar), IP addresses (IPv4 addresses in particular are becoming quite scarce), and of course actual traffic on a page [22]. For example, using visits(u) to denote the number of visits to page u in a given amount of time (optionally weighted by the dwell-time), we can define the teleportation vector to discount unpopular pages:

$$t(u) = \frac{\text{visits}(u)}{\sum_{v \in V} \text{visits}(v)}$$

As a second example, using domain(u) to denote the domain of web page u, we can define the teleportation vector to discount domains with many web pages:

$$t(u) = \frac{1}{|\bigcup_{v \in V} \text{domain}(v)| \; |\{v \in V : \text{domain}(u) = \text{domain}(v)\}|}$$

Using the random-surfer analogy, in the probability-λ event of a jump, the surfer does not choose a page uniformly at random, but instead first chooses a domain and then a page within that domain, thereby giving equal minimum-endorsement ability to each (nonzero-cost) domain instead of each (zero-cost) page. As a third example, using addresses(u) to denote the IP address(es) of the web server(s) serving page u, we can define the teleportation vector to discount servers with many web pages while giving credit to multi-hosted pages:

$$t(u) = \frac{1}{|\bigcup_{v \in V} \text{addresses}(v)|} \sum_{a \in \text{addresses}(u)} \frac{1}{|\{v \in V : a \in \text{addresses}(v)\}|}$$

Again using the random-surfer analogy, in the event of a jump the surfer first chooses a web server IP address and then chooses a page served by that machine.

In addition to adjusting the teleportation vector to dilute the ability of link farms to endorse target pages, we could adjust the link weight matrix to prefer altruistic or topical links. As we said above, any altruistic link classifier suffers from a non-negligible false-positive rate: it is impossible to rule out the possibility that two web publishers are colluding. On the other hand, the topicality of a link from u to v can be captured by straightforward means, e.g. by quantifying the textual similarity between u and v, using, say, the cosine similarity between their tf.IDF-weighted term vectors [29]. Given a similarity measure $\sigma : V \times V \to \mathbb{R}$ where higher values indicate higher similarity, we can define W as follows:

$$W(u,v) = \frac{\sigma(u,v)}{\sum_{w \in O(u,E)} \sigma(u,w)} \quad \text{if } (u,v) \in E;\ 0 \text{ otherwise}$$

There are many other possible techniques for distinguishing topical from templatic links, for example segmenting the page into "blocks" and discounting links contained in navigational or advertising blocks [6].

Hopefully the above examples will convince you that there is room for improving PageRank, by finding ways to make it a more effective ranking function and more resilient to link-spam. Any such research should be data-driven: given the availability of large web corpora (e.g. the billion-page ClueWeb09 crawl [8]) and associated test sets (e.g. the TREC 2009 web track judgments [7]), it is possible to evaluate ranking functions in a repeatable fashion. As a corollary, studies of PageRank's mathematical properties and efforts to speed up its computation have more impact if they don't assume t and W to be uniform in any way.

2.2 HITS and Its Variants

At about the same time as Page et al. proposed PageRank, Jon Kleinberg suggested a different link-based ranking algorithm called "Hyperlink-Induced Topic Search" [16]. HITS takes the result set of a query as its input, expands the result set to include immediately neighboring pages in the web graph, projects this expanded vertex set onto the full web graph to obtain a neighborhood graph, and computes scores for each vertex in that neighborhood graph. Since it takes a query's result set as input, HITS is query-dependent, and the latency introduced by computing it at query-time is a major concern to commercial search engines, given the high correlation between response time and audience engagement [19].

Fundamentally, HITS consists of two distinct steps: First, given a result set, compute a neighborhood graph; and second, compute scores for each vertex in the neighborhood graph. Kleinberg suggested that the neighborhood graph should be computed by extending the result set to include all vertices that are within distance 1 in the link graph, ignoring "intrinsic" (egotistic by our terminology) links. In order to keep high-in-degree result vertices from inflating the neighborhood vertex set too much, he suggested including just (say) 50 randomly chosen endorsing vertices of each such high-indegree result. The neighborhood graph consists of the neighborhood vertices and those edges of the full web graph that connect neighborhood vertices and are not intrinsic. To formalize this, we write $\mathcal{R}_n(A)$ to denote a uniform random sample of n elements from set A ($\mathcal{R}_n(A) = A$ iff $|A| \leq n$), and we assume that all intrinsic edges have been removed from the web graph (V, E). Using this notation and given a result set $R \subseteq V$ to query q, Kleinberg's neighborhood graph consists of a neighborhood vertex set V_R and a neighborhood edge set E_R:

$$V_R = \bigcup_{u \in R} \{u\} \cup \mathcal{R}_{50}(I(u, E)) \cup O(u, E)$$

$$E_R = \{(u, v) \in E : u \in V_R \wedge v \in V_R\}$$

Kleinberg furthermore suggested computing two scores for each $v \in V_R$: an authority score $a(v)$ and a hub score $h(v)$, the former indicating whether v is a good authority (i.e. how relevant v is with respect to q), and the latter indicating whether v is a good hub (i.e. if v links to pages that are good authorities). Kleinberg defined a and h in a mutually recursive fashion, as $a(v) = \sum_{u \in I(v, E_R)} h(u)$ and $h(u) = \sum_{v \in O(u, E_R)} a(v)$, and suggested computing the fixed point of this recurrence by performing power iteration and normalizing a and h to unit length after each step. Using linear algebra notation leads to a more concise definition. If we define the neighborhood graph's adjacency matrix as $A(u, v) = 1$ iff $(u, v) \in E_R$ and 0 otherwise, then the authority score vector is the principal eigenvector of the matrix $A^T A$, and the hub score vector is the principal eigenvector of the matrix AA^T. Based on experimental studies [23], authority scores are useful ranking features, while hub scores carry virtually no signal.

Lempel and Moran suggested a variant of HITS called the Stochastic Approach to Link Sensitivity Analysis, or SALSA for short [18]. SALSA combines

ideas from HITS and PageRank: The SALSA authority score vector is the stationary probability distribution of a random walk over the neighborhood graph, where each transition consists of choosing an incoming link and traversing it backwards, and then choosing an outgoing link and traversing it forwards. Using linear algebra notation, the authority score vector is the principal eigenvector of the matrix $I^T O$, where $I(u,v) = \frac{1}{|I(v,E_R)|}$ iff $(u,v) \in E_R$ and 0 otherwise, and $O(u,v) = \frac{1}{|O(u,E_R)|}$ iff $(u,v) \in E_R$ and 0 otherwise. Despite their similarity, SALSA scores are significantly better ranking features than HITS scores [24].

In the course of performing the aforementioned experimental studies [23,24] into the effectiveness of HITS and SALSA as ranking features, we found that the choice of neighborhood graph has a significant impact on effectiveness. We discovered four modifications to the neighborhood selection algorithm that each increase effectiveness: first, using consistent instead of random sampling; second, sampling both the endorsed as well as the endorsing vertices of each result; third, omitting edges that don't touch results; and fourth, sampling the eligible edges [26]. Using $\mathcal{C}_n(A)$ to denote an unbiased consistent sample [5] of n elements from set A, we select the neighborhood graph of result set R as follows:

$$V_R = \bigcup_{u \in R} \{u\} \cup \mathcal{C}_a(I(u,E)) \cup \mathcal{C}_b(O(u,E))$$

$$E_R = \{(u,v) \in E : (v \in R \wedge u \in V_R \cap \mathcal{C}_c(I(v,E))) \vee (u \in R \wedge v \in V_R \cap \mathcal{C}_d(O(u,E)))\}$$

The free variables a, b, c, and d in the above formulas determine how many adjacent vertices and edges are sampled for each result vertex. In our experiments, effectiveness was maximal for a, b in the mid-single digits and c, d around 1000. The *SALSA-SETR* row of Table 1 provides effectiveness numbers.

Experimenting with HITS-like ranking algorithms requires fast access to vertices and edges in the web graph. While it would be possible to use a standard relational database to store the graph, extracting the neighborhood graph of a given result set exhibits a very random access pattern, such that disk latency would become a serious bottleneck. For this reason, we developed the Scalable Hyperlink Store [27], a bespoke system that maintains a web graph in main memory, distributed over multiple machines and employing compression techniques that leverage structural properties of web graphs, e.g. link locality. This infrastructure enables us to get one-minute turnarounds when scoring 100 TREC queries; however, the time required for scoring an individual query is pushing the boundary of what is acceptable for a commercial search engine, where the aim is to keep overall query latencies within fractions of a second.

In order to overcome this problem, we experimented with techniques for performing as much of the SALSA computation off-line as possible. We explored two approaches. The first and more radical approach is to assume that each page in the web graph is a singleton result set (to some unspecified query), and to off-line perform a separate SALSA computation for each page. More specifically, for each page u, we extract the neighborhood graph $(V_{\{u\}}, E_{\{u\}})$ around u, compute SALSA authority scores for all vertices in $V_{\{u\}}$, and store a mapping from u to the k highest-scoring $v \in V_{\{u\}}$ together with their scores. At query time,

having determined the result set R for the given query, we retrieve the entries for each $v \in R$ from the mapping (yielding a multi-set S of $k|R|$ vertex-score pairs), and for each $v \in R$ we find all occurrences of v in S and sum up their scores to produce an overall score for v. The parameter k controls a trade-off between ranking effectiveness and required space. The *SS-SALSA-3* row in Table 1 shows the effectiveness for $k = 10$ (i.e. using 120 bytes per page in the web corpus). The effectiveness is below that of Lempel and Moran's "classic" SALSA, but the query-time portion of the computation is much cheaper, simply requiring a table lookup for each result. The table can be distributed across multiple machines in the same way the index is in a modern search engine; the score multi-set of each result can be looked up by the same index-serving machine that held the result, but the final scoring has to be performed by the result aggregator.

The second, less radical approach is to move the neighborhood graph extraction off-line while keeping the score computation on-line [26]. The basic idea is to compute and store a short summary for each node in the web graph, consisting of two small samples of endorsing and endorsed vertices, and two Bloom filters containing larger samples of endorsing and endorsed vertices. At query time, we look up the summary of each vertex in the result set, we use these summaries to construct an approximation of the neighborhood graph, and we compute SALSA authority scores on this approximate graph. The size of a summary is governed by how many neighbors are sampled and how many hash functions are used by the Bloom filter, and (as one might expect) there is a trade-off between size and ranking effectiveness. Choosing parameters that lead to 500 byte summaries makes this algorithm as effective as the (purely online) SETR variant of SALSA described above. The summary table can be distributed across the index-serving machines and summaries can be passed along with results to the result aggregator; however, the construction of the approximate neighborhood graph and the subsequent scoring requires all summaries, and therefore has to be performed on the machine responsible for query distribution and result aggregation.

Much of the design space in HITS-like ranking algorithms remains to be explored. For example, HITS considers only the distance-one neighborhood of the result set. In the Companion algorithm [11], Dean and Henzinger suggested including more-distant neighbors; how does this impact ranking effectiveness? As another example, HITS and SALSA both discard egotistic hyperlinks, but make no attempts to incorporate link topicality. Could effectiveness be improved by weighing each edge according to the textual similarity of the linked pages, in a fashion similar to what we suggested above for PageRank? And as a final example, how affected are HITS and SALSA by link spam, and could we improve their resiliency by using ideas similar to what was described above, e.g. by sampling endorsing and endorsed vertices in a traffic-biased fashion?

3 Conclusion

This paper gives a high-level overview of the two main approaches to leveraging hyperlinks as ranking features: query-independent features as exemplified

by the PageRank algorithm, and query-dependent features as exemplified by the HITS algorithm. The main research challenge in this space is to identify features that (in combination with many other features) improve ranking effectiveness, but that at the same are very efficient to determine at query time. For PageRank-style features, query-time efficiency is not a significant issue, but ranking effectiveness is, due to the increase in link spam as well as the decreasing fraction of on-topic links. Effectiveness can be improved by biasing PageRank towards on-topic links and against "spammy" web pages. On the other hand, various descendants of HITS produce highly effective ranking features, but have a high query-time cost. Two approaches to containing this expense are to implement highly-optimized infrastructure for executing HITS-like computations, or to devise ways to to move as much of the computation as possible off-line.

The ranking problem is about identifying and leveraging observable human activities (production by authors and consumption by readers) that correlate well with human satisfaction with the performance of an IR system. In order to truly solve this problem, we need a model of these authors and readers. But at present, we do not possess any model with predictive abilities – i.e. a model that would allow us to *a priori* predict the performance of a new ranking feature and that would stand up to subsequent experimental validation. In the absence of such a model, experimental validation is of paramount importance!

References

1. Adali, S., Liu, T., Maddon-Ismail, M.: Optimal link bombs are uncoordinated. In: 1st Intl. Workshop on Adversarial Information Retrieval on the Web, pp. 58–69 (2005)
2. Alexa Traffic Rank, http://www.alexa.com/help/traffic-learn-more
3. Baeza-Yates, R., Boldi, P., Castillo, C.: Generalizing PageRank: damping functions for link-based ranking algorithms. In: 29th Annual ACM Conference on Research and Development in Information Retrieval, pp. 308–315 (2006)
4. Borodin, A., Roberts, G.O., Rosenthal, J.S., Tsaparas, P.: Link analysis ranking: algorithms, theory, and experiments. ACM Trans. Internet Technology 5, 231–297 (2005)
5. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. In: 30th Annual ACM Symposium on Theory of Computing, pp. 327–336 (1998)
6. Cai, D., He, X., Wen, J.-R., Ma, W.-Y.: Block-level link analysis. In: 27th Annual ACM Conference on Research and Development in Information Retrieval, pp. 440–447 (2004)
7. Clarke, C.L.A., Craswell, N., Soboroff, I.: Overview of the TREC 2009 Web track. In: 18th Text REtrieval Conference (2009)
8. The ClueWeb09 dataset, http://boston.lti.cs.cmu.edu/Data/clueweb09/
9. Davison, B.D.: Recognizing nepotistic links on the Web. In: AAAI Workshop on Artificial Intelligence for Web Search, pp. 23–28 (2000)
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: 6th Symposium on Operating Systems Design & Implementation, pp. 137–149 (2004)

11. Dean, J., Henzinger, M.: Finding related pages in the World Wide Web. In: 8th Intl. World Wide Web Conference, pp. 389–401 (1999)
12. Gyöngyi, Z., Garcia-Molina, H.: Link spam alliances. In: 31st Intl. Conference on Very Large Data Bases, pp. 517–528 (2005)
13. Hadoop, http://hadoop.apache.org/
14. Kamvar, S.D., Haveliwala, T.H., Manning, C.D., Golub, G.H.: Extrapolation methods for accelerating PageRank computations. In: 12th Intl. World Wide Web Conference, pp. 261–270 (2003)
15. Kincaid, J.P., Fishburn, R.P., Rogers, R.L., Chissom, B.S.: Derivation of new readability formulas for Navy enlisted personnel. Research Branch Report 8-75, U.S. Naval Air Station, Memphis (1975)
16. Kleinberg, J.: Authoritative sources in a hyperlinked environment. J. ACM 46, 604–632 (1999)
17. Langville, A.N., Meyer, C.D.: Deeper inside PageRank. Internet Mathematics 1, 335–380 (2004)
18. Lempel, R., Moran, S.: SALSA: The stochastic approach for link-structure analysis. ACM Transactions on Information Systems 19, 131–160 (2001)
19. Linden, G.: Marissa Mayer at Web 2.0, http://glinden.blogspot.com/2006/11/marissa-mayer-atweb-20.html
20. Marchiori, M.: The quest for correct information on the Web: hyper search engines. In: 6th Intl. World Wide Web Conference, pp. 265–274 (1997)
21. McSherry, F.: A uniform approach to accelerated PageRank computation. In: 14th Intl. World Wide Web Conference, pp. 575–582 (2005)
22. Najork, M.: Systems and methods for ranking documents based upon structurally interrelated information. US Patent 7,739,281 (filed 2003, issued 2010)
23. Najork, M., Zaragoza, H., Taylor, M.: HITS on the Web: how does it compare? In: 30th Annual ACM Conference on Research and Development in Information Retrieval, pp. 471–478 (2007)
24. Najork, M.: Comparing the effectiveness of HITS and SALSA. In: 16th ACM Conference on Information and Knowledge Management, pp. 157–164 (2007)
25. Najork, M., Craswell, N.: Efficient and effective link analysis with precomputed SALSA maps. In: 17th ACM Conference on Information and Knowledge Management, pp. 53–61 (2008)
26. Najork, M., Gollapudi, S., Panigrahy, R.: Less is more: sampling the neighborhood graph makes SALSA better and faster. In: 2nd ACM Intl. Conference on Web Search and Data Mining, pp. 242–251 (2009)
27. Najork, M.: The Scalable Hyperlink Store. In: 20th ACM Conference on Hypertext and Hypermedia, pp. 89–98 (2009)
28. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the Web. Technical Report, Stanford InfoLab (1999)
29. Qi, X., Nie, L., Davison, B.: Measuring similarity to detect qualified links. In: 3rd Intl. Workshop on Adversarial Information Retrieval on the Web, pp. 49–56 (2007)
30. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M., Gatford, M.: Okapi at TREC-3. In: 3rd Text REtrieval Conference (1994)
31. Xue, G.-R., Zeng, H.-J., Chen, Z., Yu, Y., Ma, W.-Y., Xi, W., Fan, W.: Optimizing web search using web click-through data. In: 13th ACM Intl. Conference on Information and Knowledge Management, pp. 118–126 (2004)
32. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: 8th Symposium on Operating Systems Design & Implementation, pp. 1–14 (2008)

Incremental Algorithms for Effective and Efficient Query Recommendation

Daniele Broccolo[1], Ophir Frieder[2], Franco Maria Nardini[1], Raffaele Perego[1], and Fabrizio Silvestri[1]

[1] ISTI-CNR, Pisa, Italy
[2] Department of Computer Science, Georgetown University, Washington DC, USA

Abstract. Query recommender systems give users hints on possible *interesting queries* relative to their information needs. Most query recommenders are based on static knowledge models built on the basis of past user behaviors recorded in query logs. These models should be periodically updated, or rebuilt from scratch, to keep up with the possible variations in the interests of users. We study query recommender algorithms that generate suggestions on the basis of models that are updated continuously, each time a new query is submitted. We extend two state-of-the-art query recommendation algorithms and evaluate the effects of continuous model updates on their effectiveness and efficiency. Tests conducted on an actual query log show that contrasting model aging by continuously updating the recommendation model is a viable and effective solution.

1 Introduction

A key challenge for web search engines is improving user satisfaction. Therefore, search engine companies exert significant effort to develop means that correctly "guess" what is the real hidden intent behind a submitted query.

In the latest years, web search engines have started to provide users with query recommendations to help them refine queries and to quickly satisfy their needs. Query suggestions are generated according to a model built on the basis of the knowledge extracted from query logs. The model usually contains information on relationships between queries that are used to generate suggestions. Since the model is built on a previously collected snapshot of a query stream, its effectiveness decreases due to interest shifts [9]. To reduce the effect of aging, query recommendation models must be periodically re-built or updated.

We propose two novel incremental algorithms, based on previously proposed, state-of-the-art query recommendation solutions, that update their model continuously on the basis of each new query processed. Designing an effective method to update a recommendation model poses interesting challenges due to: i) *Limited memory availability* – queries are potentially infinite, and we should keep in memory only those queries "really" useful for recommendation purposes,

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 13–24, 2010.

ii) *Low response time* – recommendations and updates must be performed efficiently without degrading user experience.

Some of the approaches considered in related works are not suitable for continuous updates because modifying a portion of the model requires, in general, the modification of the whole structure. Therefore, the update operation would be too expensive to be of practical relevance. Other solutions exploit models which can be built incrementally. The two algorithms we propose use two different approaches to generate recommendations. The first uses association rules for generating recommendations, and it is based on the *static* query suggestion algorithm proposed in [11], while the second uses click-through data, and its *static* version is described in [2].

We named the new class of query recommender algorithms proposed here "*incrementally updating*" query recommender systems to point out that this kind of systems update the model on which recommendations are drawn without the need for rebuilding it from scratch. We conducted multiple tests on a large real-world query log to evaluate the effects of continuous model updates on the effectiveness and the efficiency of the query recommendation process. Result assessment used an evaluation methodology that measures the effectiveness of query recommendation algorithms by means of different metrics. Experiments show the superiority of incrementally updating algorithms with respect to their static counterparts. Moreover, the tests conducted demonstrated that our solution to update the model each time a new query is processed has a limited impact on system response time.

The main contributions presented in this work are: i) a novel class of query recommendation algorithms whose models are continuously updated as user queries are processed, ii) two new metrics to evaluate the quality of the recommendations computed, iii) an analysis of the effect of time on the quality and coverage of the suggestions provided by the algorithms presented and by their static counterparts.

2 Related Work

The wisdom of the crowds, i.e., the behavior of many individuals is smarter than the behavior of few intelligent people, is the key to query recommenders and to many other web 2.0 applications. We now present a review of state-of-the-art techniques for query recommendation.

Document-based. Baeza-Yates *et al.* in [1] propose to compute groups of related queries by running a clustering algorithm over the queries and their associated information recorded in the logs. Semantically similar queries may even not share query-terms if they share relevant terms in the documents clicked by users. Query suggestions are ranked according to two principles: i) the similarity of the queries to the input query, and ii) the support, which measures how much the answers of the query have attracted the attention of users. The solution is evaluated by using a query log containing 6,042 unique queries from the TodoCL search engine.

Click-through-based. Beeferman and Berger in [4] apply a hierarchical agglomerative clustering technique to click-through data to find clusters of similar queries and similar URLs in a Lycos log. A bipartite graph is created from queries and related URLs which is iteratively clustered by choosing at each iteration the two pairs of most similar queries and URLs. The experimental evaluation shows that the proposed solution is able to enhance the quality of the Lycos's query recommender which was used as baseline.

Cao *et al.* propose a query suggestion approach based on contexts [8]. A query context consists of recent queries issued by a user. The query suggestion process is structured according to two steps. An offline phase summarizes user queries into concepts (i.e., a small set of similar queries) by clustering a click-through bipartite graph, and an on-line step finds the context of the submitted query, and its related concepts suggesting associated queries to the user. This solution was experimented with a large query log containing 1,812 millions of queries.

Session-based. Boldi *et al.* introduce the concept of *Query Flow Graph* [5] (QFG). Authors define a QFG as a directed graph in which nodes are queries, and edges are weighted by the probability $w(q_i, q_j)$ of being traversed. Authors highlight the utility of the model in two concrete applications, namely, *finding logical sessions* and *query recommendation*. Boldi *et al.* refine the previous study in [6], [7] proposing a query suggestion scheme based on a random walk with restart model. The query recommendation process is based on reformulations of search missions. Baraglia *et al.* showed that the QFG model ages [3] and propose strategies for updating it efficiently.

Fonseca *et al.* use an association rule mining algorithm to devise query patterns frequently co-occurring in user sessions, and a query relations graph including all the extracted patterns is built [10]. A click-through bipartite graph is then used to identify the concepts (synonym, specialization, generalization, etc.) used to expand the original query.

3 Incremental Algorithms for Query Recommendation

Our hypothesis is that continuously updating the query recommendation model is feasible and useful. As validation, we consider two well-known query recommendation algorithms and modify them to continuously update the model on which recommendations are computed. It is worth mentioning that not all query recommendation algorithms can be redesigned to update their model on-line. For example, some of the approaches presented in Section 2 are based on indexing terms of documents selected by users, clustering click-through data, or extracting knowledge from users' sessions. Such operations are very expensive to perform on-line and their high computational costs would compromise the efficiency of the recommender system.

The two algorithms considered use different approaches for generating recommendations. The first uses association rules [11] (henceforth *AssociationRules*), while the second exploits click-through data [2] (henceforth *CoverGraph*). Hereinafter, we will refer to the original formulations of the two algorithms as "*static*",

as opposed to their relative incremental versions which will be called "*incremental*".

3.1 Static solutions

Static solutions work by preprocessing historical data (represented by past users' activities on query logs), building an on-line recommendation module that is used to provide suggestions to users.

AssociationRules. Fonseca *et al.* uses association rules as a basis for generating recommendations [11]. The algorithm is based on two main phases. The first uses query log analysis for session extraction, and the second basically extracts association rules and identifies highly related queries. Each session is identified by all queries sent by an user in a specific time interval ($t = 10$ minutes). Let $I = I_1, \ldots, I_m$ be the set of queries and T the set of user sessions t. A session $t \in T$ is represented as a vector where $t_k = 1$ if session t contains query $k \in [1, \ldots, m]$, 0 otherwise.

Let X be a subset of I. A session t satisfies X, if for all items I_k in X, $t_k = 1$.

Association rules are implications of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds with i) a *confidence* factor of c if $c\%$ of the transactions in T that contains X also contains Y, and ii) a *support* s if $s\%$ of the sessions in T contains $X \cup Y$. The problem of mining associations is to generate all the rules having a support greater than a specified minimum threshold (*minsup*). The rationale is that distinct queries are considered related if they occurs in many user sessions.

Suggestions for a query q are simply computed by accessing the list of rules of the form $q \Rightarrow q'$ and by suggesting the q''s corresponding to rules with the highest support values.

CoverGraph. Baeza-Yates et al. use click-through data as a way to provide recommendations [2]. The method is based on the concept of *cover graph*. A *cover graph* is a bipartite graph of queries and URLs, where a query and a URL are connected if the URL was returned as a result for the query and a user clicked on it.

To catch the relations between queries, a graph is built out of a vectorial representation for queries. In such a vector-space, queries are points in a high-dimensional space where each dimension corresponds to a unique URL u that was, at some point, clicked by some user. Each component of the vector is weighted according to the number of times the corresponding URL has been clicked when returned for that query. For instance, suppose we have five different URLs, namely, $u_1, u_2, \ldots, u_5$, suppose also that for query q users have clicked three times URL u_2 and four times URL u_4, the corresponding vector is $(0, 3, 0, 4, 0)$. Queries are then arranged as a graph with two queries being connected by an edge if and only if the two queries share a non-zero entry, that is, if for two different queries the same URL received at least one click. Furthermore, edges are weighted according to the cosine similarity of the queries they connect. More formally, the weight of an edge $e = (q, q')$ is computed according

to Equation 1. In the formula, D is the number of dimensions, i.e., the number of distinct clicked URLs, of the space.

$$W(q, q') = \frac{q \cdot q'}{|q| \cdot |q'|} = \frac{\sum_{i \leq D} q_i \cdot q'_i}{\sqrt{\sum_{i \leq D} q_i^2} \sqrt{\sum_{i \leq D} {q'_i}^2}} \quad (1)$$

Suggestions for a query q are obtained by accessing the corresponding node in the cover graph and extracting the queries at the end of the top scoring edges.

3.2 Incremental algorithms

The interests of search-engine users change over time, and new topics may become popular. Consequently, the knowledge extracted from query logs can suffer from an aging effect, and the models used for recommendations rapidly become unable to generate useful and interesting suggestions [3]. Furthermore, the presence of "*bursty*" [12] topics could require frequent model updates independent of the model used.

The algorithms proposed in Section 3.1 use a statically built model to compute recommendations. *Incremental* algorithms are radically different from static methods for the way they build and use recommendation models. While static algorithms need an off-line preprocessing phase to build the model from scratch every time an update of the knowledge base is needed, incremental algorithms consist of a single online module integrating the two functionalities: i) *updating* the model, and ii) providing suggestions for each query.

Starting from the two algorithms presented above, we design two new query recommender methods continuously updating their models as queries are issued. Algorithms 1 and 2 formalize the structure of the two proposed incremental algorithms that are detailed in the following. The two incremental algorithms differ from their static counterparts by the way in which they manage and use data to build the model. Both algorithms exploit *LRU* caches and *Hash* tables to store and retrieve efficiently queries and links during the model update phase.

Our two incremental algorithms are inspired by the *Data Stream Model* [13] in which a stream of queries are processed by a database system. Queries consist modifications of values associated with a set of data. When the dataset fits completely in memory, satisfying queries is straightforward. Turns out that the entire set of data cannot be contained in memory. Therefore, an algorithm in the data stream model must decide, at each time step, which subset of the set of data is worthwhile to maintain in memory. The goal is to attain an approximation of the results we would have had in the case of the non-streaming model. We make a first step towards a data stream model algorithmic framework aimed at building query recommendations. We are aware that there is significant room for improvement, especially in the formalization of the problem in the streaming model. Nonetheless, we show empirically that an incremental formulation of two popular query recommender maintains the high accuracy of suggestions.

IAssociationRules. Algorithm 1 specifies the operations performed by *IAssociationRules*, the incremental version of AssociationRules.

Algorithm 1. IAssociationRules

```
1: loop
2:    (u, q) ← GetNextQuery() {Get the query q and the user u who submitted it.}
3:    ComputeSuggestions (q, σ) {Compute suggestions for query q over σ.}
4:    if ∃LastQuery (u) then
5:       q' ← LastQuery (u)
6:       LastQuery (u) ← q {Update the last query submitted by u.}
7:       if ∃σ_{q',q} then
8:          ++σ_{q',q} {Increment Support for q' ⇒ q.}
9:       else
10:         LRUInsert (σ, (q', q)) {Insert an entry for (q', q) in σ. If σ is full, remove
            an entry according to an LRU policy.}
11:      end if
12:   else
13:      LRUInsert (u, q, LastQuery) {Insert an entry for (u, q) in LastQuery. If
         LastQuery is full, remove an entry according to an LRU policy.}
14:   end if
15: end loop
```

The data structures storing the model are updated at each iteration. We use the LastQuery auxiliary data structure to record the last query submitted by u. Since the model and the size of LastQuery could grow indefinitely, whenever they are full, the LRUInsert function is performed to keep in both structures only the most recently used entries.

Claim. Keeping up-to-date the AssociationRule-based model is $O(1)$.

The proof of the claim is straightforward. The loop at line 3 of Algorithm 1 is made up of constant-cost operations (whenever we use hash structures for both LastQuery and σ). LRUInsert has been introduced to maintain the most recently submitted queries in the model.

ICoverGraph. The incremental version of CoverGraph adopts a solution similar to that used by IAssociationRules. It uses a combination of LRU structures and associative arrays to incrementally update the (LRU managed) structure σ. Algorithm 2 shows the description of the algorithm. The hash table queryHasAClickOn is used to retrieve the list of queries having c among their clicked URLs. This data structure is stored in a fixed amount of memory, and whenever its size exceeds the allocated capacity, an entry is removed on the basis of a LRU policy (this justifies the conditional statement at line 6).

Claim. Keeping up-to-dated a CoverGraph-based model is $O(1)$.

Actually, the cost depends on the degree of each query/node in the cover graph. As shown in [2], i) the degree of nodes in the cover graph follows a power-law

Algorithm 2. ICoverGraph

```
1: Input: A threshold τ.
2: loop
3:    (u, q) ← GetNextQuery() {Get the query q and the user u who submitted it.}
4:    ComputeSuggestions (q, σ) {Compute suggestions for query q over σ.}
5:    c = GetClicks (u, q)
6:    if ∃queryHasAClickOn (c) then
7:       queryHasAClickOn (c) ← q
8:    else
9:       LRUInsert (queryHasAClickOn, c)
10:   end if
11:   for all q′ ≠ q ∈ queryHasAClickOn (c) s.t. W ((q, q′)) > τ do
12:      if w > τ then
13:         if ∃σ_{q′,q} then
14:            σ_{q,q′} = w
15:         else
16:            LRUInsert (σ, (q′, q) , w)
17:         end if
18:      end if
19:   end for
20: end loop
```

distribution, and ii) the maximal number of URLs between two queries/nodes is constant, on average. The number of iterations needed in the loop at line 11 can be thus considered constant.

From the above methods, it is clear that to effectively produce recommendations, a continuous updating algorithm should have the following characteristics:

- The algorithm must cope with an undefined number of queries. LRU caches can be used to allow the algorithm to effectively keep in memory only the most relevant items for which it is important to produce recommendations.
- The lookup structures used to generate suggestions and maintain the models must be efficient, possibly constant in time. Random-walks on graph-based structures, or distance functions based on comparing portions of texts, etc., are not suitable for our purpose.
- A modification of an item in the model must not involve a modification of the entire model. Otherwise, update operations take too much time and jeopardize the efficiency of the method.

4 Quality Metrics

Assessing the effectiveness of recommender systems is a though problem that can be addressed either through *user-studies* or via automatic evaluation mechanisms.

We opted for an automatic evaluation methodology conducted by means of two novel metrics based on the analysis of users' traces contained in query logs. Both metrics measure the overlap between queries actually submitted by the

users and recorded in the tails of users' sessions and suggestions generated starting from the first queries in the same sessions. The more users actually submitted queries suggested, the more the recommender system is considered effective.

To focus the evaluation on the most recently submitted queries in the user session, we introduce the *QueryOverlap* and the *LinkOverlap* metrics defined as follows. Let $\mathbb{S}$ be the set of all users' sessions in the query log, and let $S = \{q_1, \ldots, q_n\}$ be a user session of length n. We define $S_1 = \{q_1, \ldots, q_{\lfloor \frac{n}{2} \rfloor}\}$ to be the set of queries in the first half of the session, and let $R_j = \{r_1, \ldots, r_m\}$ be the set of top-m query recommendations returned for the query $q_j \in S_1$[1]. For each q_j, we now define $S_2 = \{q_{j+1}, \ldots, q_n\}$ to be the $n - j$ most recently submitted queries in the session, and

$$QueryOverlap = \frac{1}{K} \sum_{\substack{r_i \in R_j \\ s_k \in S_2}} [r_i = s_k] f(k) \tag{2}$$

$$LinkOverlap = \frac{1}{K} \sum_{\substack{r_i \in \text{findClk}(R_j) \\ s_k \in \text{clk}(S_2)}} [r_i = s_k] f(k) \tag{3}$$

where $[expr]$ is a boolean function whose result is 1 if $expr$ is *true* or 0 otherwise, $clk(S_2)$ is a function returning the set of clicked URLs by the user for the queries in S_2, $findClk(R_j)$ is a function returning the set of clicked URLs by other users for the queries in R_j, and $f(k)$ is a weighting function allowing us to differentiate the importance of each recommendation depending on the position it occupies in the second part of the session. The value of K is defined as $\sum_{k=1}^{m} f(k)$, where $m = |S_2|$ for the *QueryOverlap*, and $m = |clk(S_2)|$ for the *LinkOverlap* metric. K normalizes the values in the range $[0, 1]$. Finally, the *Coverage* of a recommendation model is defined as the fraction of queries for which a recommendation can be computed.

5 Experiments

5.1 Experimental Setup

We conducted our experiments on a collection consisting of the first $3,200,000$ queries from the AOL query log [14]. The AOL data-set contains about 20 million queries issued by about $650,000$ different users, submitted to the AOL search engine over a period of three months from 1st March, 2006 to 31st May, 2006.

5.2 Results

First, we analyze the effect of time on the static models (Section 3.1) showing that this type of models age as time passes.

[1] In our experiments we use $m = 5$.

The plots reported in Figures 1, 2, and 3, show the effectiveness of query suggestions on a per time window basis for both the static and incremental algorithms. We use a "timeline" composed of 10 days of the query log. The "timeline" is divided into ten intervals, each corresponding to one day of queries stored in the query log (about 400,000 queries). The queries in the first time interval were used to train the models used by the algorithms. While static models are trained only on the first interval, the incremental counterparts update their model on the basis of the queries submitted during the entire timeline considered. Effectiveness of recommendations generated by the different algorithms during the remaining nine days considered is measured by means of the *LinkOverlap*, *QueryOverlap*, and *Coverage* metrics.

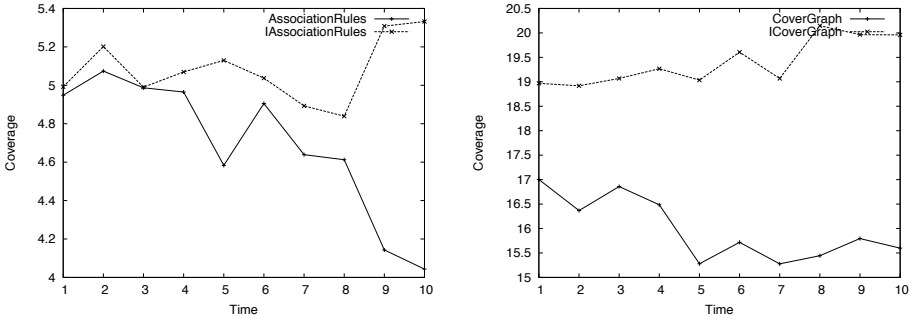


Fig. 1. Coverage for AssociationRules, IAssociationRules, CoverGraph, and ICoverGraph as a function of the time

Our first finding is illustrated in Figure 1, where coverage, i.e., the percentage of queries for which the algorithms are able to generate recommendations, is plotted as a function of time. In both plots, the coverage measured for the static versions of the recommendation algorithms decreases as time passes. In particular, at the end of the observed period, AssociationRules and CoverGraph lose 20%, and 9% of their initial coverage, respectively. Even if CoverGraph appears to be more robust than AssociationRules, both algorithms suffer an aging effect on their models. On the other hand, the coverage measured for the two incremental algorithms is always greater than the one measured for their respective static versions. In particular, at the end of the observed period, the IAssociationRules algorithm covers 23.5% more queries with respect to its static version, while ICoverGraph covers 22% more queries with respect to CoverGraph. This is due to the inclusion in the model of new and "fresh" data.

Figures 2 and 3 illustrate the effectiveness of recommendations produced by the static and incremental versions of the AssociationRules and CoverGraph algorithms as a function of the time. Both QueryOverlap and LinkOverlap in Figures 2 and 3 are measured in the above described setting.

From the plots we can see that the two static algorithms behave in a slightly different way. CoverGraph seems to suffer more than AssociationRules for the aging of its recommendation model.

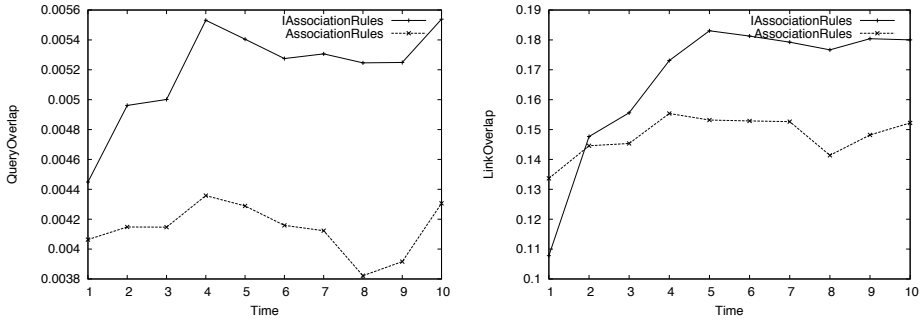


Fig. 2. QueryOverlap, and LinkOverlap for AssociationRules, and IAssociationRules as a function of the time

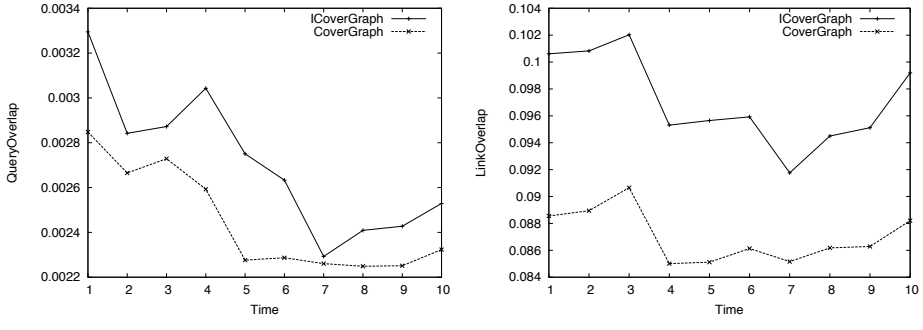


Fig. 3. QueryOverlap, and LinkOverlap for CoverGraph, and ICoverGraph as a function of the time

By considering both QueryOverlap and LinkOverlap metrics, AssociationRules is able to return better quality recommendations than CoverGraph, but, as Figure 1 shows, the coverage of queries for which suggestions can be generated is lower. In particular, CoverGraph is able to give suggestions to a number of queries which is three times larger than the one measured with AssociationRules.

We argue that incremental algorithms for query recommendation can provide better recommendations because they do not suffer for model aging, and can rapidly cover also *bursty* topics. From the figures, it is evident that the effectiveness of recommendations provided by both static and incremental models eventually stabilize. Indeed, the proposed incremental algorithms IAssociationRules and ICoverGraph produce better recommendations. With the exception of the initialization phase (see Figure 2, LinkOverlap) in which the model warms up, the percentage of effective suggestions generated by the two incremental algorithms during the entire period observed is larger than those provided by their static counterparts.

5.3 Efficiency Evaluation

The feasibility of an incremental update of the recommendation model is an important point of our work. The update operations must run in parallel with the query processor; thus, those operations must not constitute a bottleneck for the entire system. As analyzed in Section 3.2, we propose a method for keeping up-to-date two algorithms in constant time. The payoff in terms of processing time is, thus, constant. Furthermore, in the incremental algorithms we use efficient data structures, and an optimized implementation of the model update algorithm. We measure the performances of the two incremental algorithms in terms of mean response time. For each new query, our algorithms are able to update the model, and to produce suggestions in, on-the-order-of, a few tenth of a second. Such response times guarantee the feasibility of the approach on a real-world search engine where the query recommender and the query processor run in parallel.

6 Conclusions

We studied the effects of incremental model updates on the effectiveness of two query suggestion algorithms. As the interests of search-engine users change over time and new topics become popular, the knowledge extracted from historical usage data can suffer an aging effect. Consequently, the models used for recommendations may rapidly become unable to generate high-quality and interesting suggestions.

We introduced a new class of query recommender algorithms that update "*incrementally*" the model on which recommendations are drawn. Starting from two state-of-the-art algorithms, we designed two new query recommender systems that continuously update their models as queries are issued. The two incremental algorithms differ from their static counterparts by the way in which they manage and use data to build the model. In addition, we proposed an automatic evaluation mechanism based on two new metrics to assess the effectiveness of query recommendation algorithms.

The experimental evaluation conducted by using a large real-world query log shows that the incremental update strategy for the recommendation model yields better results for both coverage (more than 20% queries covered by both IAssociationRules, and ICoverGraph) and effectiveness due to the "fresh" data that are added to the recommendation models. Furthermore, this improved effectiveness is accomplished without compromising the efficiency of the query suggestion process.

References

1. Baeza-Yates, R., Hurtado, C., Mendoza, M.: Query recommendation using query logs in search engines. In: Lindner, W., Mesiti, M., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) EDBT 2004. LNCS, vol. 3268, pp. 588–596. Springer, Heidelberg (2004)
2. Baeza-Yates, R., Tiberi, A.: Extracting semantic relations from query logs. In: KDD 2007, pp. 76–85. ACM, New York (2007)

3. Baraglia, R., Castillo, C., Donato, D., Nardini, F.M., Perego, R., Silvestri, F.: The effects of time on query flow graph-based models for query suggestion. In: Proc. RIAO 2010 (2010)
4. Beeferman, D., Berger, A.: Agglomerative clustering of a search engine query log. In: Proc. KDD 2000, pp. 407–416. ACM, New York (2000)
5. Boldi, P., Bonchi, F., Castillo, C., Donato, D., Gionis, A., Vigna, S.: The query-flow graph: model and applications. In: Proc. CIKM 2008, pp. 609–618 (2008)
6. Boldi, P., Bonchi, F., Castillo, C., Donato, D., Vigna, S.: Query suggestions using query-flow graphs. In: Proc. WSCD 2009, pp. 56–63. ACM, New York (2009)
7. Boldi, P., Bonchi, F., Castillo, C., Vigna, S.: From 'dango' to 'japanese cakes': Query reformulation models and patterns. In: Proc. WI 2009. IEEE CS, Los Alamitos (2009)
8. Cao, H., Jiang, D., Pei, J., He, Q., Liao, Z., Chen, E., Li, H.: Context-aware query suggestion by mining click-through and session data. In: Proc. KDD 2008, pp. 875–883 (2008)
9. Cayci, A., Sumengen, S., Turkay, C., Balcisoy, S., Saygin, Y.: Temporal dynamics of user interests in web search queries. ICAINAW 0, 762–767 (2009)
10. Fonseca, B.M., Golgher, P., Pôssas, B., Ribeiro-Neto, B., Ziviani, N.: Concept-based interactive query expansion. In: Proc. CIKM 2005. ACM, New York (2005)
11. Fonseca, B.M., Golgher, P.B., de Moura, E.S., Ziviani, N.: Using association rules to discover search engines related queries. In: LA-WEB 2003. IEEE CS, Los Alamitos (2003)
12. Kleinberg, J.: Bursty and hierarchical structure in streams. In: Proc. KDD 2002. ACM, New York (2002)
13. Muthukrishnan, S.: Data streams: algorithms and applications. Found. Trends Theor. Comput. Sci. 1(2), 117–236 (2005)
14. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proc. INFOSCALE 2006. ACM, New York (2006)

Fingerprinting Ratings for Collaborative Filtering — Theoretical and Empirical Analysis

Yoram Bachrach and Ralf Herbrich

Microsoft Research

Abstract. We consider fingerprinting methods for collaborative filtering (CF) systems. In general, CF systems show their real strength when supplied with enormous data sets. Earlier work already suggests sketching techniques to handle massive amounts of information, but most prior analysis has so far been limited to non-ranking application scenarios and has focused mainly on a theoretical analysis. We demonstrate how to use fingerprinting methods to compute a *family* of rank correlation coefficients. Our methods allow identifying users who have similar rankings over a certain set of items, a problem that lies at the heart of CF applications. We show that our method allows approximating rank correlations with high accuracy and confidence. We examine the suggested methods empirically through a recommender system for the Netflix dataset, showing that the required fingerprint sizes are even smaller than the theoretical analysis suggests. We also explore the of use standard hash functions rather than min-wise independent hashes and the relation between the quality of the final recommendations and the fingerprint size.

1 Introduction

Recommender systems supply users with items they are likely to find interesting. Some methods use the content of the information item (in the *content based approach*). We focus on the alternative *collaborative filtering approach* (CF systems), where the system predicts whether an item is likely to interest the target user, based on the ranking of that item by other users. One obstacle in constructing real-world CF systems is the need to handle huge volumes of information.

Previous work [7] suggested a technique for computing the similarity between users, based on *sketching* — rather than storing the full lists of items for each user, it stores a concise *fingerprint* of the lists of examined items, called a *sketch*. These fingerprints are extremely short, much shorter than compression techniques allow, but only allow specific computations on the data. The sketches of [7] allow approximating the *proportional intersection similarity* (PI) of any two users. This method has been extended in [6], where similar fingerprints were used to compute the correlation between two user's rankings of items.

Both [7,6] have significant shortcomings. First, they focused on a *very specific* rank correlation coefficient — Kendall's Tau [19]. Other correlations, such as Spearman's rank correlation [23], are more appropriate for some settings [15]. For

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 25–36, 2010.

example, Spearman's Rho has the meaningful interpretation as a Pearson correlation coefficient, and known statistical tests can use it in significance testing. Second, their sketches use *min-wise independent families of hashes* (MWIFs). MWIFs are hard to construct and slow to use. Third, they only analyze sketches *theoretically*, lacking empirical evidence regarding the quality of the sketches in terms of the quality of the *final recommendations* based on these approximations. Our contributions are:

1. We suggest a similar fingerprint which allows computing a *family* of rank correlation coefficients, including the prominent *Spearman rank correlation*.
2. We discuss *empirical analysis* of such techniques, based on *Collabriprint*, our CF infrastructure which uses fingerprinting techniques, that wastested on the Netflix [8] dataset. Our empirical analysis shows that in practice:
   - It suffices to use smaller sketches than the theoretical results require.
   - It is possible to use *standard* hash functions, such as MD5 [21], instead of the more complex MWIFs, and still obtain high accuracy and confidence.
   - The final recommendation's quality depends on the fingerprints' size. Even small fingerprints result in high quality recommendations.

2 Preliminaries

We first briefly explain the problem of fingerprinting in CF systems. Consider Alice and Bob, who have *both* examined a set of n items. In some CF domains, the mere fact that a user has examined an item implicitly tells the CF system that the user liked the item. In other domains, explicit information is available as users rate examined items on a certain scale. CF systems first seek users who share similar rating patterns with the target user, and use their ratings to generate a prediction for how the target user would rate items she has not examined. A method for approximating the *Proportional Intersection (PI)* was suggested in [7]. Given two users, Alice and Bob who examined the *same* number of items, their PI is defined as follows. Denote by C_i the set of items Alice examined, and by C_j the set of items Bob examined. Both users examined the same number of items, so $|C_i| = |C_j|$. The PI is $\frac{|C_i \cap C_j|}{|C_i|} = \frac{|C_i \cap C_j|}{|C_j|}$. The Jackard measure is a similar measure when $|C_i| \neq |C_j|$, and is defined as $J_{i,j} = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$.

The PI and Jackard measures only consider *which* items were examined. *Rank correlations*, such as Spearman's Rho and Kendall's Tau, measure the similarity between two rankings (orderings) of the same items. Spearman's Rho is simply a special case of the Pearson product-moment coefficient, in which the data sets are converted to rankings before calculating the coefficient. Let $x_i = r_a(i)$ and $y_i = r_b(i)$ be the rankings of item i, given by Alice and Bob, and let $d_i = x_i - y_i$. Spearman's Rho ρ_{r_a,r_b} can be computed using the following direct formula: $\rho_{r_a,r_b} = 1 - \frac{6 \sum_{i=0}^{n} d_i^2}{n(n^2-1)}$. Both Kendall's Tau and Spearman's Rho range from -1 (strong negative correlation) to 1 (strong positive correlation).

Computing user similarity metrics allow constructing CF recommender systems, by predicting the rating a target user would give any unexamined item,

based on the ratings given by other users, weighted according to similarity to the target user. User similarity can be computed using the full information, consisting of the lists of examined items and their ratings for each user. However, such data sets can be extremely large, so it is desirable to compute similarities while minimizing the size of the data. Fingerprinting provides a good tradeoff between the required storage and the quality of the predictions.

The method of [7] approximates the PI $p_{i,}$ for any two users i, j, by maintaining short fingerprints of the lists of examined items, called *sketches*. This method assumes i and j have equal size lists of items C_1, C_2 (so $|C_i| = |C_j| = n$), and the size of each sketch depends on the target confidence δ and accuracy ϵ. The method returns an approximation $\hat{p}_{i,j}$ to $p_{i,j}$ such that, with probability of at least $1 - \delta$, $|p_{i,j} - \hat{p}_{i,j}| \leq \epsilon$. Building on this work, a method for computing the *Kendall Tau correlation* was proposed in [6]. This improves recommendations, since even users who examined similar items may rate them differently.

Similarly to the above techniques, we also use a Min-Wise Independent Family of hashes (MWIF). Let H be a family of functions over the source X and target Y, so each $h \in H$ is a function $h : X \rightarrow Y$, where Y is completely ordered. We say that H is MWIF if when randomly choosing a function $h \in H$, for any subset $C \subseteq X$, any $x \in C$ has an equal probability to be minimal under h.[1]. Formally, we say that H is MWIF, if for all $C \subseteq X$, for any $x \in C$, $Pr_{h \in H}[h(x) = min_{a \in C} h(a)] = \frac{1}{|C|}$. MWIF computations are slow, making them ill-suited for many practical applications. For full discussion of MWIFs and their construction see [10,17].

3 Rank Correlation Fingerprints

Let i, j be two users, and C_i, C_j the set of items each has examined. We now present our fingerprinting method, based on randomly choosing hashes h from a MWIF H. Similarly to [7], we consider the identities of items in the set C_i of items examined by each user as integers, apply h to all these integers and examine the minimal value obtained. Given a randomly chosen $h \in H$ we denote minimal value obtained after applying h to all elements in C_i as $m_i^h = min_{x \in C_i} h(x)$. Performing the same on C_j we denote $m_j^h = min_{x \in C_j} h(x)$. We now examine the probability that $m_i^h = m_j^h$. Theorem 1 in [7] has shown that when $|C_i| = |C_j|$ so the PI is $p_{i,j} = \frac{C_i \cap C_j}{|C_i|} = \frac{C_i \cap C_j}{|C_j|}$, we have $Pr_{h \in H}[m_i^h = m_j^h] = \frac{p_{i,j}}{2 - p_{i,j}}$. We provide a similar proof for the Jackard measure, $J_{i,j} = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$.

Theorem 1. $Pr_{h \in H}[m_i^h = m_j^h] = J_{i,j}$.

Proof. Denote $x = J_{1,2}$. The set $C_i \cup C_j$ contains three types of items: items that appear *only* in C_i, items that appear *only* in C_j, and items that appear in $C_i \cap C_j$. When an item in $C_i \cap C_j$ is minimal under h, i.e., for some $a \in C_i \cap C_j$

[1] It does not matter which distribution is used to choose h from H, as long as this distribution makes H a MWIF.

we have $h(a) = min_{x \in C_1 \cup C_2} h(x)$, we get that $min_{x \in C_i} h(x) = min_{x \in C_j} h(x)$. On the other hand, if for some $a \in C_i \cup C_j$ such that $a \notin C_i \cap C_j$ we have $h(a) = min_{x \in C_1 \cup C_2} h(x)$, the probability that $min_{x \in C_i} h(x) = min_{x \in C_j} h(x)$ is negligible[2]. Since H is MWIF, any element in $C = C_i \cup C_j$ is equally likely to be minimal under h. However, only elements in $I = C_i \cap C_j$ would result in $m_i^h = m_j^h$. Thus $Pr_{h \in H}[m_i^h = m_j^h] = \frac{1}{|C_i \cup C_j|} \cdot |C_i \cap C_j| = \frac{|C_i \cap C_j|}{|C_i \cup C_j|} = J_{i,j}$.

The fingerprints used in [7] are called *item sketches*, and are created using k hash functions. Let $v_k = \langle h_1, h_2, \ldots, h_k \rangle$ be a tuple of k randomly chosen functions from the MWIF H, and let C_i be the set of items that user i has examined. Denote the minimal item in C_i under h_s as $m_i^{h_s} = min_{x \in C_i} h_s(x)$.

Definition 1 (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: $S^k(C_i) = (m_i^{h_1}, m_i^{h_2}, \ldots, m_i^{h_k})$.*

We call a hash h_s where $m_i^{h_s} = m_j^{h_s}$ a collision hash, and say location s is a *sketch collision* for i, j. The work [7] shows that in order to approximate the PI $p_{a,b}$ with accuracy ϵ and confidence δ, it is enough to use $k = \frac{\ln \frac{2}{\delta}}{2 \frac{\epsilon^2}{9}}$ hashes. However, they do not compute how well the users' tastes correlate, a problem later addressed in [6] where the Kendall Tau correlation is approximated. We focus on a different family of correlations, based on Spearman's Rho. CF systems seek users similar to a target user, filtering out users with a low Jackard similarity to that user. Similarly to [6] we assume the CF system filters out any user with a Jackard score (or PI score) lower than some value p^*, and augment the item sketches to compute rank correlations. The system then recommends items based on scores that weight rankings given by users according to their similarity with the target user. A strong user similarity metric is rank correlation.

Our fingerprints are the *item sketches* of Definition 1, augmented with the *rating* of the minimal item under the hash. Consider Alice and Bob, with Jackard similarity of at least p^*. The item sketches in Definition 1 use k random hashes, and the fingerprint is the list of the minimal items under each hash. Due to Theorem 1, given users i with items C_i and j with items C_j, the probability of a collision for i, j on any location s (i.e. $P(m_i^{h_s} = m_j^{h_s})$) depends on $J_{i,j}$. Due to Theorem 1, if $J_{i,j} \geq p^*$, any location has a probability of at least p^* of being a collision. A collision in location s is $h_s(q)$, where q is an identity of an item chosen uniformly at random from $C_i \cap C_j$ (an item both i and j examined). Our fingerprints include the rating of the item q[3].

Similarly to item sketches (Definition 1), each location is built using a randomly chosen hash. Let h_i be the hash for the i'th location. The augmentation

[2] Such an event requires that two *different* items, $x_i \in C_i$ and $x_j \in C_j$ to be mapped to the same value $h^* = h(x_i) = h(x_j)$, and that this value would also be the minimal value obtained when applying h to both all items in C_i and in C_j. As discussed in [17], the probability for this is negligible when h's range is large enough.

[3] The sketches of [6] are similar, although we employ a very different algorithm to compute Spearman's Rho (whereas they compute Kendall's Tau).

for location i contains the *rating* of the item that is minimal under h_i. When constructing the sketch for user a, we consider the user's item set C_a and the ratings of the items in C_a. The rating of user a for items in C_a is denoted as r_a. Thus, r_a maps items in C_a to their rating. Given h_i, consider the set of items that are minimal under h_i[4], i.e. $M = \{x \in C_a | h_i(x) = m_a^{h_i}\}$. If only one item is minimal under the hash, so $|M| = 1$, we denote $M = \{m\}$, and denote the rating of that item as $g_a^i = r_a(m)$. Only with a very low probability do we have $|M| > 1$. If $|M| > 1$, denote m' to be the minimal item in M, under some predetermined 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 rating in a's eyes. We denote the sketch for user a with items C_a (where the sketch is based on $H_k = \langle h_1, \ldots, h_k \rangle$, the k randomly chosen hashes from the MWIF), as $S^k(C_a)$.

Definition 2 (Rank Correlation (RC) Sketches). *The H_k RC sketch of C_a, $S^k(C_a)$, contains the both the item sketch and the rank sketch. The item sketch is the list of minimal items in C_a under the k randomly chosen hash functions from, 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 the concatenation of these two sketches.*

The fingerprint size required for approximating Kendall's Tau using RC sketches was analyzed in [6]. We provide a similar analysis for a Spearman's Rho. Observe that an RC sketch collision for two users[5] provides the ratings of each of the two users of a *randomly chosen* item from $C_a \cap C_b$. Thus, a collision provides $r_a(x), r_b(x)$ for a randomly chosen items $x \in C_a \cap C_b$. We now determine *how many* collisions are required to approximate Spearman's Rho with a target accuracy and confidence. We wish to return an approximation $\hat{\rho_{a,b}}$ to Spearman's Rho $\rho_{a,b}$ such that with probability of at least $1 - \delta$ we have $|\rho_{r_i,r_j} - \hat{\rho_{i,j}}| \leq \epsilon$. We use the following theorem from [6] regarding the required number of hashes to provide at least k collisions.

Theorem 2. *Let k be a certain required sketch collisions, and let p be a bound from below on the Jackard similarity of any two users. The required fingerprint size to achieve the required number k of sketch collisions with probability $1 - \delta_c$ is $m \geq \frac{k}{p} + \frac{\ln \frac{1}{\delta_c}}{4p^2}(1 + 3\sqrt{k})$.*

The sketch collision probability depends on the Jackard similarity (as shown in Theorem 1). Theorem 2 shows that given a minimal Jackard similarity, a long enough fingerprint would provide the required number of collisions with

[4] If each item is hashed to a different value then there is only one item whose value under the hash is minimal. However, several items may be mapped to the same value, so there may be several items minimal under the hash.

[5] Recall that a sketch collision is a sketch location i with hash function h_i where the minimal items of the two users (a with items C_a and b with items C_b) under h_i are the same, so $m_a^{h_i} = m_b^{h_i}$.

high probability. The required fingerprint length is logarithmic in the required confidence δ_c and polynomial in the required number of collisions. We now show that a family of rank correlations, including Spearman's Rho, can be computed using the RC sketches of Definition 2, generalizing the results of [6].

We now show how the RC sketches from Definition 2 allow computing a family of rank correlations. Members of this family could be expressed as a certain bounded function of the rank differences, summed across all items. We begin by an analysis of Spearman's Rho, and then generalize to this family of rank correlations. The definition of Spearman's Rho had a direct formula for it: $\rho_{r_a,r_b} = 1 - \frac{6\sum_{i=0}^{n} d_i^2}{n(n^2-1)}$. We first note that d_i is simply the difference between the rating of a certain item in the first user's eyes and in the second user's eyes. Consider an item x chosen uniformly at random from the set of possible items. We can examine $r_a(x)$ and $r_b(x)$ and define the following random variable.

Definition 3. *The Spearman's Rho random variable X_i, for item x is* $X_i = 1 - \frac{6(r_a(x)-r_b(x))^2}{n^2-1}$

The random variable X_i has an expectation of: $\mathrm{E}[X_i] = \mathrm{E}[1 - \frac{6(r_a(x)-r_b(x))^2}{n^2-1}] = 1 - \frac{6}{n^2-1}\mathrm{E}[(r_a(x) - r_b(x))^2] = 1 - \frac{6}{n^2-1} \cdot \frac{1}{n}(\sum_i (r_a(i) - r_b(i))^2 = 1 - \frac{6\sum_{i=0}^{n} d_i^2}{n(n^2-1)} = \rho_{r_a,r_b}$.

We now denote $\rho = \rho_{r_a,r_b}$ for short. Given k such random variables, $X_1, \ldots, X_k$, we can use $\frac{1}{k}\sum_{i=1}^{k} X_i$ as an estimate for ρ. We now derive the required number of such random items to approximate ρ with accuracy ϵ and confidence δ. To achieve the desired accuracy and confidence, the number of sampled items, k, must be large enough. We find the appropriate k by using Hoeffding's inequality [16] (see similar analysis for very different uses in [11,4,3]).

Theorem 3 (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 $X_1, \ldots, X_k$ be the series k of random variables, as defined above. Let $X = \sum_{j=1}^{k} X_j$, and take $\hat{\rho} = \frac{X}{k}$ as an estimator for ρ.

Theorem 4. *A confidence interval for ρ is $[\hat{\rho} - \epsilon, \hat{\rho} + \epsilon]$. This interval holds the correct ρ with probability of at least $1 - \delta$. The required number of pair samples to perform this is $k \geq \frac{18 \ln \frac{2}{\delta}}{\epsilon^2}$*

Proof. We use Hoeffding's inequality to bound the error below the target confidence level δ. We note that $0 \leq \frac{d_i^2}{n^2-1} \leq 1$. Due to the definition of X_i (see Definition 3), all X_i are bounded between -5 and 1, and $\mathrm{E}[X] = k \cdot \rho$. Thus, from Hoeffding's inequality, the following holds: $\Pr(|X - k\rho| \geq k\epsilon) \leq 2e^{-\frac{1}{18}k\,\epsilon^2}$. Therefore the following also holds: $\Pr(|\hat{\rho} - \rho| \geq \epsilon) \leq 2e^{-\frac{1}{18}k\,\epsilon^2}$. We get that $-\frac{1}{18}k\,\epsilon^2 \leq \ln\frac{2}{\delta}$. Finally we obtain: $\epsilon \geq \sqrt{\frac{18\ln\frac{2}{\delta}}{k}}$ and $k \geq \frac{18\ln\frac{2}{\delta}}{\epsilon^2}$.

Using fingerprints with a the length determined by Theorem 2, we have a high probability of getting a large enough number of sketch collisions. Each such sketch collision gives the rating $r_a(x), r_b(x)$ of a certain randomly chosen item x, that both users (a and b) ranked. Thus, with high probability, we obtain a series of random variables as required by Theorem 4. To compute an estimate for Spearman's Rho, we take the rankings $r_a(x), r_b(x)$ of each item x that occurs on a sketch collision, and use them to compute $X_i = 1 - \frac{6(r_a(x)-r_b(x))^2}{n^2-1}$, the random variables defined above. Given c sketch collisions, as above, we use $\frac{1}{c}\sum_{i=1}^{c} X_i$ as an estimate for ρ. The analysis so far was specific for Spearman's Rho. However, we now show the same type of an analysis can be used for many similar rank correlation functions.

Theorem 5. *Let a be a constant and the function f be bounded between certain constant values b_l and b_h. The previous fingerprinting approach can be used to compute any rank correlation of the form: $\alpha = a + \frac{1}{n}\sum_i f(r_a(i), r_b(i))$.*

Proof. Let a be a constant and f a function bounded between b_l and b_h, and consider a rank correlation of the form defined above. We can define a set of random variables X_i as in Definition 3. The expectancy of the X_i's would be α. Also, since f is bounded, we can apply Hoeffding in the same way. Note the bound distance $|b_h - b_l|$ only changes the resulting constant in the expression derived for the fingerprint size, so the approach works well for any bounds. Performing the analysis similarly to Theorem 4 gives the fingerprint size for any member of this family of functions, and the same RC sketches can be used.

4 Empirical Analysis

We tested the CF fingerprinting approach by analyzing approximations of similarity metrics in the Netflix [8] movie ratings dataset. As discussed in the introduction, there are several disadvantages to the approaches of [7,6]: the use of MWIFs, the high theoretical bound on the fingerprint length, and the lack of empirical evaluation regarding the quality of the similarity approximation and *final recommendations*. We discuss how to overcome these drawbacks, and support this with empirical evidence. We show how to replace MWIFs with MD5 [21], widely used hash function. We show that the accuracy of the procedure in practice is much higher than the theoretical bounds, and empirically investigate the relation between overall recommendation accuracy and the fingerprint length.

The Netflix dataset is a movie ratings dataset, released in October 2006 by Netflix (www.netflix.com) [8]. It contains a 100 million anonymous movie ratings, given by half a million users on a collection of 17,000 movies. Fingerprinting allows approximating user similarity with high accuracy. Our framework, called *Collabriprint* was built using C# and F#. We used it on the Netflix dataset, running several tests. We computed both movie to movie similarity through the PI/Jackard similarity of the sets of users who watched the movies, and rank correlation similarity through Kendall's Tau and Spearman's Rho correlation

between users's rating of movies. We have examined the approximation error in similarity and the change in recommendation quality for different fingerprint lengths, as measured by the number k of hashes used. Our implementation has used various MD5 hash functions rather than MWIFs.

Although MWIF hashes are required for the theoretical results, constructing and using such a family is computationally expensive, and there are no widely used implementations of them. As an alternative, we have chosen to use the MD5 hash [21], a widely used hash. Since we require many such functions, we used HMAC (keyed Hash Message Authentication Code) versions of MD5, HMAC-MD5. HMACs are computed using a hash function in combination with a key, where different keys result in different hash functions, all of which appear to have a random behavior. We chose MD5 for several reasons: it is a cryptographic hash functions with semi-random behavior; It has an HMAC version; It is commonly used in many applications, and there are widely available libraries implementing it; It works quite quickly in terms of computation time.

Our first tests were conducted on randomly chosen movies pairs. For each pair we computed the Jackard similarity using the full data set, and through fingerprints. Denoting the correct PI as p and the PI estimate as $\hat{p}$, the inaccuracy for the movie pair is $e = |p - \hat{p}|$. Given an accuracy level ϵ we say the experiment had a big error if $e \geq \epsilon$, and say it was accurate if $e < \epsilon$. Let s be a sequence of m experiments. Given ϵ, denote by b_ϵ the number of experiments with a big error, and $g_\epsilon = m - b_\epsilon$ the number of accurate ones. We denote the fraction of bad experiments as $f_b(\epsilon) = \frac{b_\epsilon}{m}$. Let δ be a confidence level. The *empirical accuracy* for a target confidence δ, is the maximal ϵ for which $f_b(\epsilon)$, the fraction of bad experiments, is at most δ. For our analysis we used a confidence level $1-\delta = 0.9$. For each fingerprint size s, we chose 2000 random movie pairs. For each such pair we performed 10 experiments, each using a different fingerprint of s random hash functions. Thus, for each fingerprint size we had $20,000$ experiments. We measured the empirical accuracy for that sequence. The theoretical fingerprint size for target accuracy $\epsilon = 0.1$ and target confidence $1-\delta = 0.9$ (from the bounds in [7]), is $s = 1350$. The required size for $\epsilon = 0.15$ and $1 - \delta = 0.9$ is $s = 600$. We tested the empirical accuracy $\epsilon_e(s)$ for fingerprint sizes of $15, 20, 25, \ldots, 100$ and of $150, 200, \ldots, 650$ (all of which are much shorter than the required size for accuracy $\epsilon = 0.1$ and $1 - \delta = 0.9$). Figure 1 shows the empirical accuracy (measured in the experiment sequence) and the theoretical accuracy (obtained from the theoretical formulas), for a confidence level of $1 - \delta = 0.9$. Lower accuracy numbers are better, as the accuracy is the maximal allowed error. Figure 1 shows that on the Netflix dataset, the actual accuracy is much better than the theoretical bounds predict.

We also attempted to find the required fingerprint size to achieve a certain target accuracy ϵ (with a target confidence of $1 - \delta = 0.9$). To get the empirical required fingerprint size s_e for target accuracy ϵ, we found the minimal fingerprint size s such that the empirical accuracy ϵ_e for that size is better than the required accuracy ϵ (i.e. $\epsilon_e(s) < \epsilon$). The following figure presents both the theoretical and empirical required sizes for different values of target accuracy.

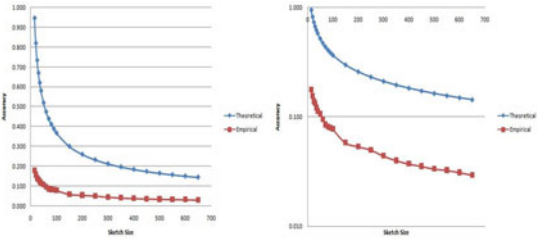


Fig. 1. Theoretical accuracy and empirical accuracy, for confidence 0.9 (right - logarithmic scale)

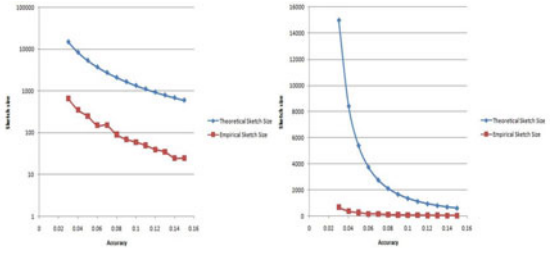


Fig. 2. Required sketch sizes for target accuracy, for confidence 0.9 (left - logarithmic scale)

As Figure 2 shows, the required fingerprint size in empirical tests is much smaller than the theoretical bounds. The figure shows the empirical fingerprint size is roughly proportional to the theoretical bounds. The empirical size is about only 5% of the theoretical required size. The above results indicate that in practice it is not necessary to use large sizes to achieve very good accuracy. Given a dataset sample, we suggest finding the right size to use empirically.

We analyzed the quality of the recommendations based on fingerprints of different length. We implemented a simple recommendation algorithm, based on [9], where the score for item i for target user u (using the user set U of recommender) is $\hat{u} + k \cdot \sum_{s \in U} sim(u,s) \cdot (s[i] - \hat{s})$ where $sim(u,s)$ is the similarity between u, s, such as Jackard, Spearman Rho or Kendall's Tau, $s[i]$ is the ranking user s gives item i, and $\hat{u}$ is the average rating of user u. The value k is used as a normalizing factor, typically $\frac{1}{\sum_{s \in U} sim(u,s)}$. Our recommender set U was the 1000 most Jackard similar users, and we used Kendall Tau for sim. Both measures can be computed using the full data, or by fingerprinting. Obviously, the fingerprint scores differ from the full data scores.

Consider the scores computed for each movie in the full data set, which we call *true scores*. When ordering movies according to the true scores, the first items are the best recommendations. We call an item in the top 5% of the list *relevant items*. Now consider scores computed using the fingerprints only, which

we call *fingerprint scores*. Sorting the list by fingerprint scores, and taking the top items, we obtain the recommendations made using the fingerprints. The quality of the fingerprint method is determined by its *precision*, the proportion of relevant items out of all the fingerprint recommendations. The following figure presents the relation between the fingerprint size (number of hashes used), and the quality of the recommendations.

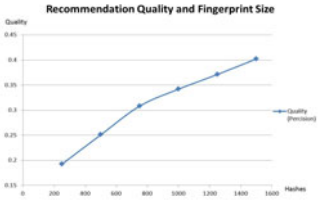


Fig. 3. Recommendation quality

Another important parameter is the *recall* of the method, the proportion of the relevant items that are covered. The recall is the proportion of relevant items (top 5% items under the true scores) covered by the top 5% of the items under the fingerprint scores. The recall values we measured range from 26% for 250 hashes to 33% for 1500 hashes. These results indicate that the quality of the recommendations is strongly related to the length of the fingerprint used. As seen in Figure 3, although longer fingerprints increase the quality, the quality improvement rate drops as more hashes are used. In some domains fingerprinting may allow the data to fit in RAM, rather than secondary storage (disks), and we suggest choosing the highest fingerprint length that allows the data to fit in memory, to maximize recommendation quality.

5 Related Work

We analyzed the famous Netflix dataset [8], a relatively recent CF domain. Early recommender systems include GroupLens [20] and Ringo [22]. Today's CF systems, as used by Amazon.com, MovieFinder.com and Launch.com face massive datasets. CF algorithms correlate human ratings to predict future preferences. There are many such correlations, such as the Pearson correlation used in [20] or the cosine similarity used in [9]. We focused on fingerprinting in massive CF systems. Similar works use fingerprints to approximate relations between strings. The work [13] presents a sketch for the L_1-difference, and [12] examines Hamming norm. This work extends [7,6]. Both are purely theoretical, while this work includes theoretical analysis of different rank correlations and empirical analysis. While we use MD5 hashes for the empirical analysis, our theoretical results are based on MWIF hashes. MWIFs were treated in [10,17]. We hope such techniques can be used to build fingerprints for various uses, such as collaborative filtering [7], trust and reputation aggregation [18,5] and general preference aggregation and voting procedures [14].

Other techniques also concisely represent data relations. Our methods use the Locally Sensitive Hashing (LSH) [2] framework, but our analysis is based on assumptions that are specific to CF. Similar approaches are Random Projections [1] and Spectral Hashing [24]. Our methods are simple and efficient, and the empirical analysis shows they perform well on real CF datasets.

6 Conclusion

We suggest fingerprinting methods for CF systems, extending previous works to allow computing a family of rank correlations, including Spearman's Rho. We also provide empirical analysis of the suggested methods. Our results are based on *Collabriprint*, a complete fingerprinting based recommender system. Our results show that it is possible to use simple hash functions (rather than MWIFs) and short fingerprints to obtain high quality recommendations.

Several questions remain open for future research. First, we used a simple CF approach, and it would be interesting to see how the fingerprint size affects more sophisticated approaches. Also, it might be possible to create more sophisticated fingerprints to improve recommendation quality while still keeping the fingerprints small. Also, an even shorter fingerprint may be possible in certain restricted domains. Finally, other fingerprinting applications would be welcome.

References

1. Achlioptas, D.: Database-friendly random projections: Johnson-Lindenstrauss with binary coins. JCSS 66 (2003)
2. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM 51(1), 117–122 (2008)
3. Bachrach, Y., Betzler, N., Faliszewski, P.: Probabilistic possible winner determination. AAAI 38 (2010)
4. Bachrach, Y., Markakis, E., Resnick, E., Procaccia, A.D., Rosenschein, J.S., Saberi, A.: Approximating power indices: theoretical and empirical analysis. Autonomous Agents and Multi-Agent Systems 20(2), 105–122 (2010)
5. Bachrach, Y., Parnes, A., Procaccia, A.D., Rosenschein, J.S.: Gossip-based aggregation of trust in decentralized reputation systems. Autonomous Agents and Multi-Agent Systems 19(2), 153–172 (2009)
6. Bachrach, Y., Herbrich, R., Porat, E.: Sketching algorithms for approximating rank correlations in collaborative filtering systems. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 344–352. Springer, Heidelberg (2009)
7. Bachrach, Y., Porat, E., Rosenschein, J.S.: Sketching techniques for collaborative filtering. In: IJCAI 2009, Pasadena, California (July 2009)
8. Bell, R.M., Koren, Y.: Lessons from the netflix prize challenge. SIGKDD Explor. Newsl. 9(2), 75–79 (2007)
9. Breese, J.S., Heckerman, D., Kadie, C.: Empirical analysis of predictive algorithms for collaborative filtering. In: Proceedings of UAI 1998, pp. 43–52. Morgan Kaufmann, San Francisco (1998)
10. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. JCSS 60(3), 630–659 (2000)

11. Clifford, R., Efremenko, K., Porat, E., Rothschild, A.: K-mismatch with don't cares. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 151–162. Springer, Heidelberg (2007)
12. Cormode, G., Datar, M., Indyk, P., Muthukrishnan, S.: Comparing data streams using Hamming norms. IEEE Trans. Knowl. Data Eng. 15(3), 529–540 (2003)
13. 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)
14. Hemaspaandra, E., Spakowski, H., Vogel, J.: The complexity of Kemeny elections. Theoretical Computer Science 349(3), 382–391 (2005)
15. Higgins, J.J.: An introduction to modern nonparametric statistics. Thomson Learning (2004)
16. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301), 13–30 (1963)
17. Indyk, P.: A small approximately min-wise independent family of hash functions. Journal of Algorithms 38(1), 84–90 (2001)
18. Jusang, A., Ismail, R., Boyd, C.: A survey of trust and reputation systems for online service provision. Decision Support Systems 43(2), 618–644 (2007)
19. Kendall, M.G.: A new measure of rank correlation. Biometrika 30, 81–93 (1938)
20. Resnick, P., Iacovou, N., Suchak, M., Bergstorm, P., Riedl, J.: Grouplens: An open architecture for collaborative filtering of netnews. In: Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work, Chapel Hill, North Carolina, pp. 175–186. ACM, New York (1994)
21. Rivest, R.L.: The md5 message-digest algorithm (rfc 1321)
22. Shardan, U., Maes, P.: Social information filtering: Algorithms for automating "word of mouth". In: ACM CHI 1995, vol. 1, pp. 210–217 (1995)
23. Spearman, C.: The proof and measurement of association between two things 1904. The American Journal of Psychology 100(3-4), 441–471 (1987)
24. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: Advances in Neural Processing Systems (2008)

On Tag Spell Checking

Franco Maria Nardini[2], Fabrizio Silvestri[2],
Hossein Vahabi[1,2], Pedram Vahabi[4], and Ophir Frieder[3]

[1] IMT, Lucca, Italy
[2] ISTI-CNR, Pisa, Italy
[3] Department of Computer Science
Georgetown University, Washington DC, USA
[4] University of Modena and Reggio Emilia, Modena, Italy

Abstract. Exploiting the cumulative behavior of users is a common technique used to improve many popular online services. We build a tag spell checker using a graph-based model. In particular, we present a novel technique based on the graph of tags associated with objects made available by online sites such as Flickr and YouTube. We show the effectiveness of our approach on the basis of an experimentation done on real-world data. We show a precision of up to 93% with a recall (i.e., the number of errors detected) of up to 100%.

1 Introduction

Differing from query spell checking, the goal of tag spelling correction is to enable the tagged object to be *actually* retrieved. Correcting "hip hop" as "hip-hop", when the latter is more frequent than the former, is a good way to allow people to find the resource when querying for the concept "hip-hop"[1]. By tagging a resource, a user wants that resource to be easily found. When querying, a user formulates a sentence-like text to retrieve the desired concept and to satisfy her/his information need. On the other hand, with tags, users leave "*breadcrumbs*" for others to detect. Like "*breadcrumbs*", tags do not have any particular inter-relationship apart from the fact that they were left by the same user.

We exploit the collective knowledge [1,2] of users to build a spell checking system on tags. The main challenge is to enable tag spell checkers to manage *sets of terms* (with their relative co-occurrence patterns) instead of strings of terms, namely, queries.

Much previous work is devoted to query spell checking. Differing from queries, namely short strings made up of two or three terms, tags are sets of about ten terms per resource. We exploit this relatively high number of tags per resource to provide correct spelling for tags. Indeed, our method exploits correlation between tags associated with the same resource. We are able to detect and correct

[1] The hidden assumption we do is that people formulate queries for resources, following the same mental process as people tagging resources.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 37–42, 2010.

common variations of tags by proposing the "right", i.e., the most commonly used, versions.

We evaluate our method through a user study on a set of tagged resource coming from YouTube. Note that our experiments are fully reproducible. Instead of using proprietary data sources, like search engines' query logs, we leverage publicly available resources.

Research on spell checking has focused either on non-word errors or on real-word errors [3]. Non-word errors such as *ohuse* for *house* can easily be detected by validating each word against a lexicon, while real-world errors, e.g., *out* in *I am going our tonight*, are difficult to detect. Cucerzan *et al.* [4] investigate the use of implicit and explicit information about language contained in query logs for spelling correction of search queries. Zhang *et al.* [5] propose an approach to spelling correction based on Aspell. Shaback *et al.* [6] propose a multi-level feature-based framework for spelling correction via machine learning. Merhav *et al.* [7] use a probabilistic approach to enrich descriptors with corrected terms in a P2P application. Ahmad *et al.* [8] apply the noisy channel model to search query spelling corrections.

Unlike previous approaches, we cannot rely on information about sequences of terms, or n-grams. We must, thus, find another way to contextualize tags within other tags that are used in association with the same resource.

2 Model Description

In tagging objects, users associate a set of words, i.e., tags, with a resource, e.g., a video, a photo, or a document, having in mind a precise semantic concept. Actually, tagging is the way users allow their resources to be found. Based on this hypothesis, our spell checker and corrector presents two important features: i) it is able to identify a misspelled tag, ii) it proposes a ranked list of "*right*", i.e., most likely to come to users' minds, tags associated with the misspelled tag.

We use a weighed co-occurrence graph model to capture relationships among tags. Such relationships are exploited to detect a misspelled tag and to identify a list of possibly correct tags.

Let R be a set of resources. Let Σ be a finite alphabet. Let $T \subseteq \Sigma^*$ be a set of tags associated with each resource. Let $\gamma : R \rightarrow T$ be a function from resources to set of tags mapping a resource with its associated set of tags. Furthermore, let $T^* = \cup \{\gamma(r), \forall r \in R\}$ be the union of all tags for all resources in R.

Let $G = (V, E)$ be an undirected graph. V is the set of nodes where each node represents a tag $t \in T^*$, and E is the set of edges defined as $E = V \times V$. Given two nodes, u, v, they share an edge if they are associated at least once with the same resource. More formally, $E = \{(u, v)|u, v \in V,$ and $\exists r \in R|u, v \in \gamma(r)\}$. Both edges and nodes in the graph are weighted. Let $u, v \in V$ be two tags. Let $w_e : E \rightarrow \mathbb{R}$ be a weighting function for edges measuring the co-occurrence of the two tags, namely, the number of times the two tags appear together for a resource. For a given node $v \in V$, $w_v : V \rightarrow \mathbb{R}$ associates a tag with its weight.

Given two nodes $u, v \in V$, let $P_{u,v}$ be the set of all paths of any length between u and v. Let $\sigma : V \times V \rightarrow \mathbb{R}$, where $\sigma(u, v) = min_{pathlength} P_{u,v}$ be the function providing the length of the shortest path between the two nodes u, v.

Given a tag $t \in V$, and a threshold value for shortest path l, we define "neighbor nodes" the set of nodes $N_t^l = \{t_1 \in V | \sigma(t_1, t) \leq l\}$. "Neighbor nodes" are then filtered by using the tag frequency. For each node in N_t^l, we select from the set nodes having a frequency greater than the frequency of the tag t.

Let $NG_t^l = \{t_1 \in N_t^l | w_v(t_1) > w_v(t)\}$ be the set of neighbors of t at maximum distance l having a frequency greater than the tag t. Given two nodes u, v, let $d(u, v)$ be a function returning the edit distance of the two tags u, v. By applying d to a tag t and tags in its neighborhood, NG_t^l, we define the "candidate neighbor nodes" as follows.

Definition 1. *Given a tag $t \in V$, and a threshold value for the edit distance δ, we define $F_t = \{t_1 \in NG_t^l | d(t_1, t) \leq \delta\}$ as the "*candidate neighbor node*" set.*

We use the *candidate neighbor node* set to check if the tag t is misspelled and, if needed, to find better tags to be used instead of t. We assume that, if F_t is empty then t is a right tag, and it does not need any correction. This approach allows us to have high effectiveness due to relationships between neighbor nodes. The method is also very efficient as we explore only a part of the graph in to find candidate neighbor nodes.

Our approach to the spelling correction problem using tags is defined as:

> TagSpellingCorrection: Given $s \in \Sigma^*$, find $s' \in T^*$ such that $d(s, s') \leq \delta$ and $P(s'|s) = max_{t \in T^* : d(s,t) \leq \delta} R(t|s)$, where, $d(s, t)$ is a distance function between the two tags, δ is a threshold value, $P(s'|s)$ is the probability of having s' as a correction of s, and $R(t|s) = 1$ if $t \in F_s$, or 0, otherwise.

Algorithm (1) solves the TagSpellingCorrection problem providing a list of possible right tags for a given tag t. It filters the co-occurrence graph by sorting "neighbor nodes" by their importance, and by considering only the top-r most frequent ones.

Given a node $t \in V$, and a $r \in \mathbb{N}$, we define a function $T_p : V \times \mathbb{N} \rightarrow O$ taking the top-r most important nodes of a node t, where $O = \{v_i \in N_t^1, i = 1, ..., r | w_e(v_j) \geq w_e(v_{j+1})$, and $w_e(v_1) = max_{t \in N_t^l}(w_e(t))$ with $j \in 1, ..., r - 1\}$.

3 Experiments

To evaluate our spelling correction approach we built a dataset of tags from YouTube. In particular, we crawled 568,458 distinct YouTube videos obtaining a total of 5,817,896 tags.

Precision and recall are evaluated by means of a *user-study* to get the percentage of good corrections. We asked assessors to evaluate four complete vocabularies produced by four different runs of the Algorithm (1). The four runs differed from the set of parameters used to produce the vocabulary.

Algorithm 1. FindCorrectTag

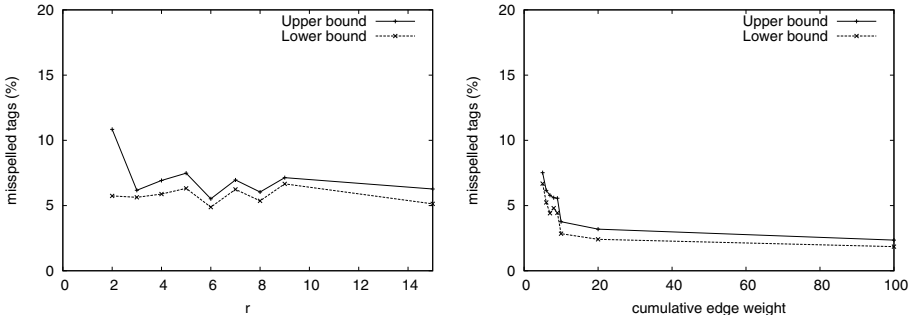
Input: $G = (V, E)$ co-occurrence graph, a tag $t \in V$, a threshold level for shortest path $l > 0$, and a threshold value for edit distance $\delta > 0$, the number of top nodes to consider $r \in \mathbb{N}$, node and edge threshold weights f, k.
Output: a list F_t of correct tags for t.
$F_t = \{\}$, $Temp_s = \{\}$, $s = t$, $V_{f,k} = \{\}$, $E_{f,k} = \{\}$
for all $t \in V$ **do**
  if $(w_v(t) > k)$ **then**
    $V_{f,k} = V_{f,k} \cup \{t\}$
  end if
end for
for all $e \in E$ **do**
  if $(w_e(e) > f)$ **then**
    $E_{f,k} = E_{f,k} \cup \{e\}$
  end if
end for
$LevelwiseBreadthFirst(G_{f,k}, t, l, 1)$;
for all $t_1 \in V_{f,k}$ in $Temp_s$ **do**
  if $(w_v(t_1) > w_v(t)) \wedge (d(t_1, t) > \delta)$ **then**
    $F_t = F_t \cup \{t_1\}$
  end if
end for

To compute recall we performed an estimation of the number of total wrong tags in the collection. We avoided performing a user-study over all the collection of tags by taking samples of dimension $n = 101$ and by computing their average values. The sample was chosen with the Mersenne-Twister algorithm. We computed the *confidence interval* [9] with $\alpha = 0.05$, and the *tstudent* values equal to 1.984.

Figure 1(a) shows the percentage of detected misspellings by varying the top-r most weighted edges of each node. Generally, it decreases as the threshold on the tag frequency increases. Figure 1(b) shows the percentage of estimated misspelled tags by varying the cumulative edge weight. It decreases as the threshold on edges

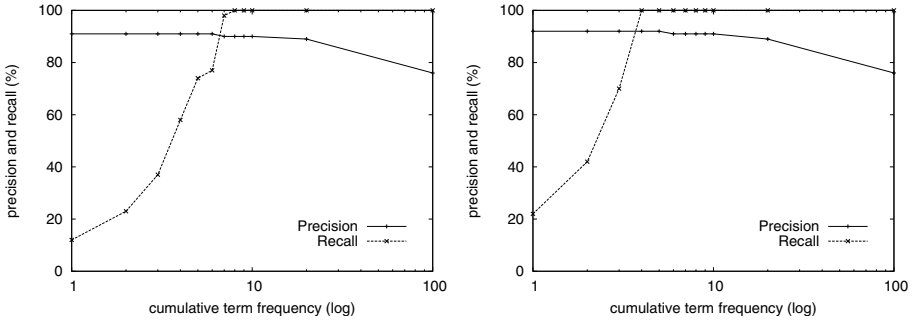
Algorithm 2. $LevelwiseBreadthFirst(G, t, l, state)$

Input: $G_{f,k}$ a filtered co-occurrence graph, a tag $t \in V$, a threshold level for shortest path l, the current *state*
Output: a set of nodes.
if $state < l$ **then**
  $Temp_s = Temp_s \cup \{Tp(s, r)$ as set$\}$. {get the top-r neighbors nodes of s at distance 1.}
  for all $t_1 \in V$ in $Temp_s$ **do**
    $LevelwiseBreadthFirst(G, t_1, l, state + 1)$
  end for
end if



(a) (b)

Fig. 1. (a) Upper and lower bound of the average tag frequency by varying top-r most important neighbors, (b) misspelled tags (%) by varying cumulative edge frequency



(a) (b)

Fig. 2. Precision and recall (%) by varying the tag frequency (log). (a) $r = 10$, $l = 1$, $\delta = 1$, $k > 0$, (b) $r = 10$, $l = 2$, $\delta = 1$, $k > 0$.

weights increases. This means that by removing edges with an associated weight less than k, the percentage of misspelled tags (not isolated nodes) in the whole set decreases.

Figure 2(a) shows values for precision and recall by varying the threshold on tag frequency. Such a threshold works as a filter on low frequency tags. This evaluation uses only the first level of neighbors of tags ($l = 1$). Precision is high (up to 90%), and recall improves significantly (up to 100%) putting a threshold on the tag frequency to values close to 10. On the other hand, by fixing such threshold to values greater than 20, our method starts losing precision. Figure 2(b) shows values for precision and recall by varying the threshold on tag frequency when the evaluation uses two levels of neighbors of a tag ($l = 2$). Precision is still high (up to 90%), while recall improves significantly (up to 100%) by putting a threshold on tag frequency to values close to 4.

4 Conclusions

We presented a tag spelling correction method exploiting a graph model representing co-occurrences between tags. Tags from YouTube's resources are collected and represented on a graph. Such a co-occurrence graph is then used in combination with an edit distance and term frequency to obtain a list of right candidates for a given possibly misspelled term. Experiments show that this collaborative spell checker yields a precision up to 93%, with a recall of 100% (in many cases). We plan to extend this work by considering not only co-occurrence of tags within the same objects, but also to consider the effect of neighborhoods at various distance levels.

References

1. Silvestri, F.: Mining query logs: Turning search usage data into knowledge. Found. Trends Inf. Retr. 4, 1–174 (2010)
2. Surowiecki, J.: The Wisdom of Crowds. Anchor (2005)
3. Kukich, K.: Techniques for automatically correcting words in text. ACM Comput. Surv. 24, 377–439 (1992)
4. Cucerzan, S., Brill, E.: Spelling correction as an iterative process that exploits the collective knowledge of web users. In: Proc. EMNLP (2004)
5. Whitelaw, C., Hutchinson, B., Chung, G.Y., Ellis, G.: Using the web for language independent spellchecking and autocorrection. In: Proc. EMNLP 2009. ACL (2009)
6. Schaback, J.: Multi-level feature extraction for spelling correction (2007)
7. Merhav, Y., Frieder, O.: On multiword entity ranking in peer-to-peer search. In: Proc. SIGIR 2008. ACM, New York (2008)
8. Ahmad, F., Kondrak, G.: Learning a spelling error model from search query logs. In: Proc. HLT 2005. ACL (2005)
9. Freund, J.: Mathematical Statistics. Prentice-Hall, Englewood Cliffs (1962)

Compressed Self-indices Supporting Conjunctive Queries on Document Collections

Diego Arroyuelo[1], Senén González[2], and Mauricio Oyarzún[3]

[1] Yahoo! Research Latin America,
Blanco Encalada 2120, Santiago, Chile
darroyue@yahoo-inc.com
[2] Department of Computer Science, Universidad de Chile
sgonzale@dcc.uchile.cl
[3] Universidad de Santiago de Chile
mauricio.silvaoy@usach.cl

Abstract. We prove that a document collection, represented as a unique sequence T of n terms over a vocabulary Σ, can be represented in $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, such that a conjunctive query $t_1 \wedge \cdots \wedge t_k$ can be answered in $O(k\delta \log\log |\Sigma|)$ adaptive time, where δ is the instance difficulty of the query, as defined by Barbay and Kenyon in their SODA'02 paper, and $H_0(T)$ is the empirical entropy of order 0 of T. As a comparison, using an inverted index plus the adaptive intersection algorithm by Barbay and Kenyon takes $O(k\delta \log \frac{n_M}{\delta})$, where n_M is the length of the shortest and longest occurrence lists, respectively, among those of the query terms. Thus, we can replace an inverted index by a more space-efficient in-memory encoding, outperforming the query performance of inverted indices when the ratio $\frac{n_M}{\delta}$ is $\omega(\log |\Sigma|)$.

1 Introduction, Results and Previous Work

Text retrieval systems allow the access to big text collections in order to retrieve information that satisfies the information needs of users, so they are fundamental for information retrieval (IR) [2]. Let $\mathcal{D} = \{D_1, \ldots, D_N\}$ be a collection of N documents, where each document D_i is modeled as a sequence of *terms* (or *words*) from a vocabulary Σ of size $|\Sigma|$. We assume that every original term in Σ has been assigned a unique term identifier. Thus, from now by "term" we will mean "term identifier". *Conjunctive queries* $t_1 \wedge \cdots \wedge t_k$, asking to report the documents that contain all the $t_1, \ldots, t_k \in \Sigma$, are one of the most common kinds of queries issued to text retrieval systems. We assume a model where all the documents that satisfy the query are retrieved (as opposed to a model where only the most relevant documents need to be found).

Text retrieval systems are usually based on *inverted indices* [2,37], which consists of a *vocabulary table* (containing the $|\Sigma|$ distinct *terms* from the collection) and an *occurrence list* for every term $c \in \Sigma$, storing the identifiers of the documents that contain the term c. Conjunctive queries are supported by intersecting the occurrence lists of the query terms [2,1,16,34]. However, the occurrence lists

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 43–54, 2010.

must be precomputed and stored, which requires considerable amounts of space. So these must be compressed, and usually stored on secondary storage [2], in both cases yielding a slowdown in query processing time.

Given the popularity of inverted indices, it is not a surprise that most of the work on reducing the space requirements in IR has focussed on compressing the occurrence lists of inverted indices. Among the vast literature on the topic, we can cite some recent relevant work [36,34]. An alternative that has emerged in recent years, and that seeks to avoid the use of secondary storage when dealing with large volumes of data, is that of compressed/succinct data structures [26,30]. Because of the continuously growing data repositories available from different sources, reducing the space used by an algorithm is fundamental for efficiency matters. Hence, in recent years there have been several studies on succinct and compressed data structures, achieving small and functional data structures. For instance, there are representations for general trees of n nodes using just $2n+o(n)$ bits, while supporting a complete set of operations on it in $O(1)$ time [8,19,33]. This is $\Theta(\log n)$ times smaller than the traditional representation. Thus, succinct data structures are not only space-efficient, but also versatile and functional.

There are also compressed data structures for full-text search [30], where all the occurrences of a pattern in a text must be reported. These are called compressed self-indices, which replace a text with a representation that uses space proportional to that of the compressed text, allows indexed searches over the text, and extracting any text snippet. These indices provide interesting trade-offs in practice [18], allowing one to replace the traditional suffix trees and suffix arrays in many scenarios. This is an active and mature area of research. Compressed data structures supporting operations rank and select [11,20,4,3] are also fundamental. Given a sequence of symbols S, $\mathsf{rank}_c(S,i)$ counts the number of c's in $S[1..i]$, and $\mathsf{select}_c(S,j)$ yields the position of the j-th occurrence of c in S. Operation $\mathsf{access}(S,i)$, which yields $S[i]$, is also relevant to retrieve the indexed sequence. These operations are usually the core for more complex succinct data structures [30], and shall be central to our work. Succinct data structures have also been successfully used to represent graphs [17,13], functions and permutations [6,7], and to support document retrieval [32,35,25], among others.

However, the impact of succinct/compressed data structures has not been as strong as expected in the area of document reporting in text retrieval systems. As we said, most of the achievements to reduce the space requirements in IR are related to compressing the inverted indices. There are several solutions for the problem of document reporting [29,32,35,21,25]. However, none of these support IR queries, like conjunctive queries, which must be solved by first generating the occurrence lists of the search patterns, to then apply standard intersection algorithms on these lists. This is, however, non-efficient, since an inverted index has precomputed these lists, and only the price for the intersection must be paid for. We aim at performing conjunctive queries directly over the index, without generating the occurrence lists beforehand. The idea of emulating list intersection algorithms in this way was hinted at [7], though without details and not regarding document collections, but just single texts. Their representation is based on a

data structure for permutations, which represents, somehow, the *wavelet tree* [24] of the text. Thus, they cannot profit from the use of more efficient representations for rank/select (e.g., [3]).

Instead of storing the occurrence lists, the work [10] proposes to use a compressed index to generate them on the fly, thus saving considerable space (though increasing the query time). However, they index just single texts, and *all* the query occurrences need to be found. This can be adapted to represent document collections, though the total query time becomes proportional to the number of query occurrences, rather than the number of documents containing it.

Thus, there is little (or none) support in the compressed data structure literature for operations that are fundamental in IR. An important result in this track would be, for instance, being able to replace an inverted index by a compressed/succinct encoding, while still supporting efficient conjunctive queries.

Let $T[1..n]$ denote the sequence obtained from the concatenation of the documents in the collection. Our main contribution is an study on the support of conjunctive queries in self-indexed text retrieval systems. We show that any rank/select data structure (supporting operation access in time $O(a)$, rank in $O(r)$ time, and select in $O(s)$ time) is powerful enough to support the following IR operations: (i) Extracting any document snippet of length ℓ in $O(a \cdot \ell)$ time; (ii) obtaining the occurrence list (of length *occ*) of a given query term in time $O((r+s)occ)$; and (iii) conjunctive queries $t_1 \wedge \cdots \wedge t_k$, for $k \geq 2$, in $O(k\delta(r+s))$ adaptive time, where δ is the instance difficulty of the query, as defined by [5].

In particular, we use the rank/select data structure from [3], and achieve $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, such that snippet extraction can be carried out in $O(\ell)$ time, and conjunctive queries can be answered in $O(k\delta \log\log |\Sigma|)$ adaptive time, where $H_0(T)$ is the empirical entropy of order 0 of T [28]. It is important to note that $\delta \leq n_m$, where n_m is the length of the shortest occurrence list among those of the query terms. As a comparison, the time achieved by an inverted index plus the adaptive intersection algorithm by Barbay and Kenyon [5] is $O(k\delta \log \frac{n_M}{\delta})$, where n_M is the length of the longest occurrence list among those of the query terms. Thus, we can replace an inverted index by a more space-efficient in-memory encoding, which outperforms inverted indices when the ratio n_M/δ is $\omega(\log |\Sigma|)$.

2 Preliminary Concepts and Notation

Succinct Data Structures for rank *and* select. Given a sequence $S[1..n]$ over an alphabet $\Sigma = \{1, \ldots, |\Sigma|\}$, operation $\mathsf{rank}_c(S, i)$, for $c \in \Sigma$, counts the number of c's occurring in $S[1..i]$. Operation $\mathsf{select}_c(S, j)$ is defined as the position of the j-th occurrence of c in S (we assume that select yields $n + 1$ iff the number of c's in S is less than j). The representation by Barbay et al. [3] uses $nH_0(S) + o(n)(H_0(T) + 1)$ bits, where $H_0(T) \leq \log |\Sigma|$ denotes the empirical entropy of order 0 of T [28]. Operations rank_c and select_c on S, for any $c \in \Sigma$, are supported in $O(\log\log |\Sigma|)$ time, as well as access in $O(1)$ time [3] . Let r and s denote the time complexities for the rank and select operations, respectively.

Our Representation for the Document Collection. Let $\mathcal{D} = \{D_1, \ldots, D_N\}$ be a *document collection* of size N, where each document is represented as a sequence $D_i[1..l_i]$ of l_i *terms* from a vocabulary Σ of size $|\Sigma|$. Assuming that '\$'$\notin \Sigma$ is a special separator symbol, we build the sequence:

$$T[1..n] = \$D_1\$D_2\$ \cdots \$D_N\$$$

of length $n = 1 + \sum_{i=1}^{N} (l_i + 1)$ and size $n \log |\Sigma|$ bits. The order of the concatenation is arbitrary. A convenient order for document ranking purposes can be the one given by a global ranking function, such as Hits [27] or Pagerank [9].

Each document D_i is assigned a unique *document identifier* i. If we represent T with a rank/select data structure, then given any position $1 \leq j \leq n$, operation $\mathsf{get\_docid}(j) \equiv \mathsf{rank}_{\$}(T, j)$ yields the document identifier of the document j belongs to. Given a document identifier $1 \leq i \leq N$, one can obtain the starting position within T for document D_i as $\mathsf{get\_doc}(i) \equiv \mathsf{select}_{\$}(T, i) + 1$.

3 Succinct Encodings for Document Reporting

The work [10] showed that an instance of rank/select data structure (in particular, a *wavelet tree* [24]) is competitive with an inverted index for reporting *all* the occurrences of a query term t. This is relevant for text searching, where all the occurrences need to be found. In our case, however, we should search for every occurrence of t, and for each determine the document that contains it, which is reported (without repetitions). However, this is wasteful when there are many occurrences of t, but just a few documents actually contain it.

To work in time proportional to the number of documents containing the query term t, we locate the first occurrence of t within T by using $j = \mathsf{select}_t(T, 1)$. We compute next the document identifier $d = \mathsf{get\_docid}(j)$ of the document containing the term, and report it. Then, with $j = \mathsf{select}_{\$}(T, d + 1)$ we jump up to the end of the current document, and $f = \mathsf{rank}_t(T, j)$ counts the number of occurrences of t up to position j. Then, we jump to the next document containing an occurrence of t by means of $\mathsf{select}_t(T, f + 1)$, and repeat the procedure. The running time of this algorithm is $O((r+s)occ)$. By using the data structure from [3] we obtain $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, while the occ documents containing a query term can be computed in $O(occ \log \log |\Sigma|)$ time.

4 Efficient Support for Conjunctive Queries

Traditionally, conjunctive queries are supported by intersecting the occurrence lists of the individual query terms [2,1,16,34]. If the document collection has been encoded as in Section 3, a simple solution for a query $t_1 \wedge t_2$ could be to generate the occurrence lists $\mathtt{Occ}_1[1..n_1]$ and $\mathtt{Occ}_2[1..n_2]$ for t_1 and t_2, respectively, and then intersect them by using any intersection algorithm [1,16,34]. The lower bound for the intersection problem in the comparison model is $\Omega(n_1 \log \frac{n_2}{n_1})$ time, assuming that $n_1 \leq n_2$, which is achieved by the work of Baeza-Yates [1].

Thus, the time for conjunctive queries would be $O((r+s)(n_1+n_2)+n_1 \log \frac{n_2}{n_1})$ in the worst case. For $k > 1$, this time is $O((r+s)\sum_{i=1}^{k} n_i + kn_m \log \frac{n_M}{n_m})$, where n_i denotes the length of the occurrence list of t_i, and n_m and n_M denote the lengths of the shortest and longest occurrence lists, respectively. By using an adaptive intersection algorithm [15,5], the time would be $O((r+s)\sum_{i=1}^{k} n_i + k\delta \log \frac{n_M}{\delta})$, where $\delta \leq n_m$ is the difficulty of the instance as defined by [5].

Thus, generating the occurrence lists before the intersection phase is wasteful, since not every occurrence of t_i's is useful for the intersection. Our aim is to avoid this cost, so we can conceptually think of our indices as storing the lists.

4.1 A Simple Worst-Case Algorithm for Conjunctive Queries

Given a query $t_1 \wedge t_2$, a first approach to reduce the query cost is to obtain the occurrence list $\mathtt{Occ}_1[1..n_1]$ for t_1 as in Section 3. Then, for every document $\mathtt{Occ}_1[i]$ containing t_1, we check whether t_2 occurs within it, which is true iff:

$$\mathsf{rank}_{t_2}(T, \mathsf{get\_doc}(\mathtt{Occ}_1[i]+1)-1) - \mathsf{rank}_{t_2}(T, \mathsf{get\_doc}(\mathtt{Occ}_1[i])) > 0. \quad (1)$$

We work in time $O((r+s)n_1)$, thus saving the time to generate the list of t_2. To minimize the time, we should generate first the shortest occurrence list. However, we must store the length of the occurrence list of every term, which uses $|\Sigma| \log N$ extra bits of space, and is usually negligible in practice.

For queries of the form $t_1 \wedge \cdots \wedge t_k$, we first sort the query terms by their number of occurrences, then generate the occurrence list for the less frequent term, and then use it to drive the candidate checking, considering one term at a time in the order given by the sorting. Thus, the total time is $O(k \log k + k(r+s)n_m)$. By using the data structure from [3], we obtain:

Theorem 1. *Given a document collection T represented as a sequence of n terms over a vocabulary Σ, it can be replaced by a representation that uses $nH_0(T) + o(n)(H_0(T)+1)$ bits of space, supports extracting any document snippet of length ℓ in $O(\ell)$ time, and finding all the documents that answer a query $t_1 \wedge \cdots \wedge t_k$ in $O(k \log k + kn_m \log\log |\Sigma|)$ worst-case time, where n_m denotes the length of the shortest occurrence list among those of the query terms.*

4.2 An Adaptive Algorithm for Conjunctive Queries

Instead of performing a pair-wise intersection (as with the previous algorithm), this time we search for all query terms at once, in some sense carrying out the "intersection" as we search for them.

For a query $t_1 \wedge t_2$, we search for the first occurrence of each of the query terms, $s_1 = \mathsf{select}_{t_1}(T, 1)$ and $s_2 = \mathsf{select}_{t_2}(T, 1)$. Assume, without loss of generality, that $s_1 < s_2$ and that these correspond to document identifiers $d_i < d_{i+m}$, respectively, for $m \geq 1$. Then, notice that it is not necessary to search for the occurrences of t_1 in documents $d_{i+1}, \ldots, d_{i+m-1}$, since there is no occurrence of t_2 within them. Thus, we move s_1 up to the starting position of document

d_{i+m}, and search for the next occurrence of t_1 from there (by using select). If this lies within document d_m, then we report it as an occurrence. If the document identifier obtained is greater than d_m, then we move s_2 to the beginning of the document containing t_1, and repeat the procedure, moving forward through the text, until either s_1 or s_2 reach the value $n+1$ (recall that select yields $n+1$ when there are not enough occurrences).

Let n_1 and n_2 be the number of documents containing t_1 and t_2, respectively, and let $n_m = \min\{n_1, n_2\}$. It is not hard to see that the shortest occurrence list will be exhausted after carrying out at most $n_m + 1$ steps. This indicates that this algorithm works in $O((r+s)n_m)$ time in the worst case. An instance that yields such a behavior is as follows:

$$T = \$\cdots|\cdots t_1 \cdots|\cdots t_2 \cdots|\cdots t_1 \cdots|\cdots t_2 \cdots|\cdots t_1 \cdots|\cdots t_2 \cdots|\cdots t_1 \cdots\$$$

where the original separator '$' has been replaced by '|' for clarity (except for the first and last '$'). In this example, we work in time proportional to n_2, since it is smaller. Moreover, this method can profit from the distribution of the query terms across T. For example, if we have an instance like this:

$$T = \$\cdots|\cdots t_1 \cdots|\cdots|\cdots t_1 \cdots|\cdots|\cdots t_1 \cdots|\cdots\cdots|\cdots t_2 \cdots|\cdots|\cdots t_2 \cdots\$$$

and assuming that these are the only occurrences of t_1 and t_2 in T, it only takes $O(1)$ steps to determine that the result of the conjunctive query is empty. Notice the analogy with an integer intersection algorithm: this corresponds to the case where all values stored in the occurrence list of t_1 are smaller than those in the occurrence list of t_2. So our algorithm adapts nicely to this kind of instance.

In general, this algorithm is adaptive to this interleaving of the query terms within the document collection. For example, suppose that t_1 and t_2 occur within T in exactly δ of these groups. Then, notice that we need $O(\delta)$ steps to certify the result of the intersection. Though seen from a different perspective, notice that this is the same instance difficulty measure as the one introduced in [5], and also that $\delta \leq n_m$. Thus, the adaptive complexity of our algorithm is $O(\delta(r+s))$, which is $O((r+s)n_m)$ in the worst case.

This procedure can be generalized to any $k > 1$. Now, every query term t_i has an index s_i, which indicates the last occurrence of t_i that has been regarded. First, we look for $s_1 = \mathsf{select}_{t_1}(T, 1)$, and determine the document identifier $j_1 = \mathsf{get\_docid}(s_1)$ that contains it. Then, we look for the next occurrence of t_2 starting from document j_1. That is, let $j' = \mathsf{get\_doc}(j_1)$. Hence, we set $s_2 = \mathsf{select}_{t_2}(T, \mathsf{rank}_{t_2}(T, j') + 1)$. Then, let $j_2 = \mathsf{get\_docid}(s_2)$. So, we search for the next occurrence of t_3 starting from document j_2, and so on, regarding the query terms in a round-robin fashion. The process finishes when some s_i reaches $n+1$, since the corresponding list has been exhausted.

The adaptive analysis is similar to that for binary conjunctive queries. Hence, general conjunctive queries take $O(k\delta(r+s))$ time, where $\delta \leq n_m$ is our difficulty measure, as defined above. Thus, by using the data structure from [3], we obtain:

Theorem 2. *Given a document collection T represented as a sequence of n terms from an alphabet of size $|\Sigma|$, it can be replaced by a representation that uses $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, supports extracting any document snippet of length ℓ in $O(\ell)$ time, and finding all the documents that answer a query instance $t_1 \wedge \cdots \wedge t_k$ of difficulty $\delta \leq n_m$ in time $O(k\delta \log\log |\Sigma|)$, where n_m is the length of the shortest occurrence list among those of the query terms.*

5 Experimental Results

For our experimental results, we used a sample of 277,371 documents (taken at random) from the UK Web, collected by Yahoo! in 2006. This requires about 1.1 gigabytes, with a vocabulary of about 1.6 million terms. We used a query log of about 36 million queries submitted to `www.uk.yahoo.com` during three months of year 2006. Our computer is an Intel(R) Core(TM)2 Duo CPU at 2.80GHz, with 5 GB of RAM, and running version `2.6.31-20-server` of Linux kernel.

We decided to use a Huffman-shaped wavelet tree as a particular rank/select data structure, since these have proven to be a competitive choice in practice [12]. This uses $n(H_0(T) + 1) + o(n(H_0(T) + 1))$ bits. We based on the wavelet tree implementation from the `libcds` library, available in *Google code* [1]. Since our algorithms need an intensive use of select, we use the `darray` data structure from [31] [2] to represent the internal wavelet tree nodes. We modified the original `darray` implementation to reduce the overall space wasted, obtaining a wavelet tree that uses about 521 MB. The space achieved by representing the nodes with the very space-efficient data structure from [22] is about 500 MB. However, by using `darray` the times become around 4–5 times faster.

The first step of the algorithm from Section 4.1 generates the occurrence list $\texttt{Occ}_1[1..n_1]$ for term t_1. Then, for every remaining query term we check with Eq. (1) which of the documents in $\texttt{Occ}_1$ contain it. However, this involves the use of select. Let $\texttt{DocBegin}[1..N]$ be a table storing the document beginnings in $N \log n$ extra bits. An equivalent test, though faster in practice, is:

$$\mathsf{rank}_{t_2}(T, \texttt{DocBegin}[\texttt{Occ}[i] + 1] - 1) - \mathsf{rank}_{t_2}(T, \texttt{DocBegin}[\texttt{Occ}[i]]) > 0, \qquad (2)$$

Hence, the selects in this algorithm come from generating the occurrence list $\texttt{Occ}_1$, which amount to n_1 selects. The remainder works with rank.

In our experiments, we call `SLF` (from shortest-list first) the algorithm from Section 4.1, and `Adaptive` the one from Section 4.2. We searched for 1 million random queries from our query log, most of them of length from 1 to 6. We show in Table 1 the number of queries answered per second, for a random (top part) and a non-random (bottom part) ordering of the documents. In both cases, we show experimental results for queries that return a non-empty answer, as well as for queries that not necessarily have a non-empty answer. The non-random

[1] We thank Francisco Claude for providing the source code for the `libcds` library.
[2] We thank Kunihiko Sadakane for providing the source code for the `darray` data structure.

ordering aims at showing the adaptability of `Adaptive`. We use the following simple heuristic: we first take the vocabulary of the different terms from our query log, and construct an inverted index of the document collection for these query terms. We then take the occurrence list for the most frequent term (i.e., the longest list), and concatenate the documents that appear in the list. We proceed in the same way with the remaining occurrence lists, in the order given by their lengths (we avoid duplicate documents when performing the concatenation).

We also implemented an inverted index by representing the occurrence lists by their differences, and for every list we use the exact number of bits to represent the highest difference in the list. This allows us a much better decompression performance, while achieving an acceptable compression ratio: 189 MB, which is slightly more space than that used, for instance, by a Golomb-compressed inverted index. However, as we represent the lists by differences, we cannot apply binary/doubling search on them, so we cannot use efficient intersection algorithms like the ones at [1,5]. Hence, we use a two-level approach similar to the one at [14]. We sample one out of B values in the list, and represent them in absolute way. Thus, binary search can be used on them, which is followed by a sequential search on the corresponding block of the list. We chose a typical value $B = 8$, so we get an index that requires about 270 MB (recall that this does not include the text). It is important to note that we only store document identifiers in the occurrence lists. No extra information about term frequencies, positional information, or any other information for ranking, is stored in the lists.

As it can be seen, and unlike `Adaptive`, the inverted index and `SLF` are insensitive to the document ordering, as expected. It is important to note how the intensive use of `rank` (rather than `select`) in `SLF` yields in all cases a (sometimes slightly) better performance than `Adaptive`. We think that by using a better heuristic to concatenate the documents we can get better results.

As a comparison, rather than using `SLF` or `Adaptive` to compute the answer, we used the algorithm from Section 3 to generate the occurrence lists of each query term, yet without using any intersection algorithm afterwards to compute the final answer. We were able, in this way, to answer less than 10 queries per second. This indicates that, independently of the intersection algorithm used on the lists, this approach will be outperformed by ours. Thus, saving the time to generate the lists is very important.

When comparing with the inverted index, we conclude that our algorithms can be up to 5–7 times slower, while using about 1.92 times the space of the inverted index. However, the latter does not include the text, so snippet extraction is not possible. To achieve this, we must add the text to the index, for instance represented in compressed for with a wavelet tree (this would support extracting arbitrary text snippets). This would add about 500–520 MB to the inverted index, for a total space usage of about 1.5 times the space of our algorithms. Also, our algorithms store more information than the text itself. For instance, we can use `rank` to compute the frequency of a term within a given document. We can also know the positions of a term within a document, allowing this kind of information to be used in more involved ranking functions.

Table 1. Number of queries answered per second, for a random (top) and non-random (bottom) ordering of the document collection

	`SLF` (521 MB)	`Adaptive` (521 MB)	`Hybrid` (521 MB)	Inverted Index (270 MB, **no text**)
Every query has an answer	74	58	68	381
Not every query has an answer	102	82	95	583
Every query has an answer	77	74	82	381
Not every query has an answer	106	102	115	583

In the above experiments, algorithm `SLF` outperforms `Adaptive` in many cases. However, for queries with $k > 1$, we made the experiment of searching just for the two terms having the smallest occurrence lists. The result is that `Adaptive` is faster than `SLF` for doing this, as it can be seen in Fig. 1 (left), assuming that the document ordering explained above has been used. This is because the performance of `Adaptive` depends on the interleave factor δ. When we search for long queries, this factor can be high (close to n_{min}), because of the many terms that compose the query. However, if we search just for the two terms having the smallest lists, we minimize the probability of interleaving. Hence, `Adaptive` outperforms `SLF`. This fact improves with the query length, as it can be seen in Fig. 1 (left), which is explained by the result in Fig. 1 (right), which compares the length of the two shortest list for different query lengths. When searching for three terms, instead of two, we concluded that `SLF` outperforms `Adaptive`.

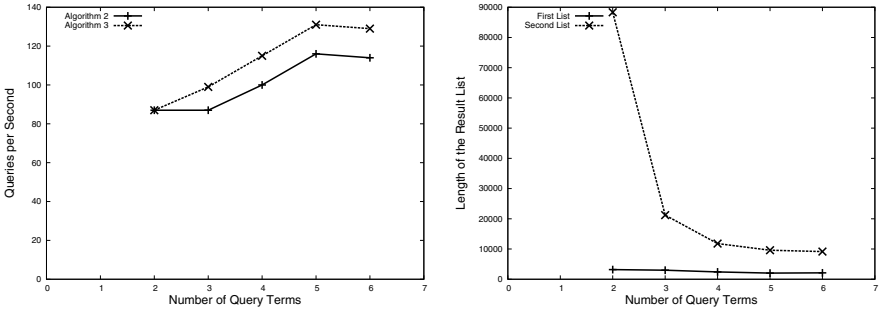


Fig. 1. Experimental time for searching the two terms with the shortest lists, for queries of different length (left); comparison of the length of the two shortest lists (right)

We use the above fact to define a hybrid scheme (called `Hybrid`) that first searches for the two terms with shortest lists using algorithm `Adaptive`, and then use this partial result to search with `SLF`. The result is shown in Table 1. As it can be seen, we obtain an improvement of about 10% in the query performance.

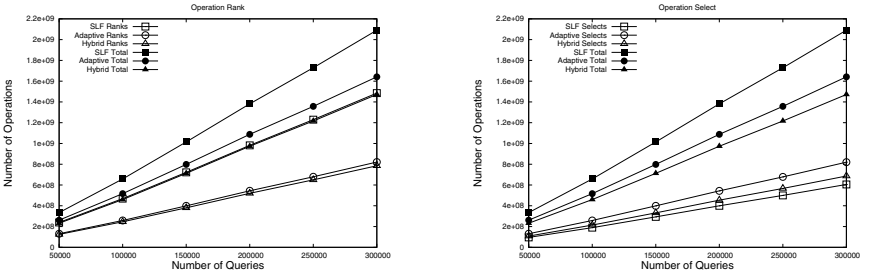


Fig. 2. Number of rank (left) and select (right) operations performed by our algorithms

Finally, in Fig. 2 we show the performance of our algorithms from another point of view: the number of rank/select operations needed to compute an answer. As it can be seen, the total number of operations (i.e., number of ranks plus number of selects) performed by algorithm `SLF` is about 1.27 times the number of operations of `Adaptive`, and 1.43 times the number of operations of `Hybrid`. However, according to our experiments, `SLF` outperforms the others when comparing the total time. This can be explained by the fact that most of the operations of `SLF` are just ranks (71%), which are supported in a much faster way in the wavelet tree implementation that we use [12]. Thus, our experimental results from Table 1 are clearly dependent on the rank/select representation used. A representation with a more efficient support for select would produce much better results for the `Adaptive` scheme. For instance, the data structure from [3] is able to support select in $O(1)$ time, so this could be relevant for our purposes. This representation could be also useful to compete against the inverted indices.

6 Conclusions

We proved that any rank/select data structures is powerful enough so as to represent a document collection using $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space and support IR operations on it. Thus, a conjunctive query $t_1 \wedge \cdots \wedge t_k$ can be answered in $O(k\delta \log\log |\Sigma|)$ adaptive time, where δ is the instance difficulty of the query [5], and $H_0(T)$ is the empirical entropy of order 0 of the collection. This outperforms the intersection algorithm [5] when the ratio $\frac{n_M}{\delta}$ is $\omega(\log |\Sigma|)$.

To conclude, our algorithms are powerful and simple to implement. The former is, on the one hand, because in theory we can perform as efficiently as (and in many cases even better than) the most efficient algorithms over inverted indices. The latter is, on the other hand, because we do not introduce any complicated data structure to support the operations, but rather any rank/select data structure can be used [12]. Our experimental results show that our result are promising. However, further work need to be done in order to obtain a query performance similar to that of inverted indices. We hope to achieve this by using a faster implementation for select, as for instance the one provided by [3].

We did not regard dynamic document collections in this paper (that is, document collections where documents are added and deleted, as well as modified). However, our algorithms can be also supported in such a case by using appropriate dynamic data structures for rank and select [23].

Acknowledgments. We thank Mauricio Marin and Gonzalo Navarro for many fruitful discussions, suggestions, and proofreading an earlier version of this paper.

References

1. Baeza-Yates, R.: A fast set intersection algorithm for sorted sequences. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 400–408. Springer, Heidelberg (2004)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. ACM Press / Addison-Wesley (1999)
3. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select with applications. CoRR, abs/0911.4981 (2009)
4. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: Proc. of SODA, pp. 680–689 (2007)
5. Barbay, J., Kenyon, C.: Adaptive intersection and t-threshold problems. In: SODA, pp. 390–399 (2002)
6. Barbay, J., Munro, J.I.: Succinct encoding of permutations: Applications to text indexing. In: Kao, M.-Y. (ed.) Encyclopedia of Algorithms. Springer, Heidelberg (2008)
7. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. STACS, pp. 111–122 (2009)
8. Benoit, D., Demaine, E., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
9. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks 30(1-7), 107–117 (1998)
10. Brisaboa, N., Fariña, A., Ladra, S., Navarro, G.: Reorganizing compressed text. In: Proc. SIGIR, pp. 139–146 (2008)
11. Clark, D., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. SODA, pp. 383–391 (1996)
12. 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)
13. Claude, F., Navarro, G.: Extended compact web graph representations. In: Elomaa, T. (ed.) Ukkonen Festschrift 2010. LNCS, vol. 6060, pp. 77–91. Springer, Heidelberg (2010)
14. Culpepper, J.S., 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)
15. Demaine, E., López-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: SODA, pp. 743–752 (2000)
16. Demaine, E., López-Ortiz, A., Munro, J.I.: Experiments on adaptive set intersections for text retrieval systems. In: Buchsbaum, A.L., Snoeyink, J. (eds.) ALENEX 2001. LNCS, vol. 2153, pp. 91–104. Springer, Heidelberg (2001)

17. Farzan, A., Munro, J.I.: Succinct representations of arbitrary graphs. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 393–404. Springer, Heidelberg (2008)
18. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. ACM Journal of Experimental Algorithmics 13 (2008)
19. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM 57(1) (2009)
20. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM TALG 3(2), article 20 (2007)
21. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
22. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. of WEA, pp. 27–38 (2005)
23. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science 410, 4414–4422 (2008)
24. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850 (2003)
25. Hon, W.-K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: FOCS, pp. 713–722 (2009)
26. Jacobson, G.: Succinct static data structures. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1988)
27. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM 46(5), 604–632 (1999)
28. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
29. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: SODA, pp. 657–666 (2002)
30. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
31. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. ALENEX, pp. 60–70 (2007)
32. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discrete Algorithms 5(1), 12–22 (2007)
33. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA, pp. 134–149 (2010)
34. Sanders, P., Transier, F.: Intersection in integer inverted indices. In: ALENEX (2007)
35. 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)
36. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. WWW, pp. 401–410 (2009)
37. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38(2) (2006)

String Retrieval for Multi-pattern Queries*

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

[1] Department of CS, National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] Department of CS, Louisiana State University, USA
{rahul,thanks}@csc.lsu.edu
[3] Department of EECS, The University of Kansas, USA
jsv@ku.edu

Abstract. Given a collection $\mathcal{D}$ of string documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n, which may be preprocessed, a fundamental task is to retrieve the most relevant documents for a given query. The query consists of a set of m patterns $\{P_1, P_2, ..., P_m\}$. To measure the relevance of a document with respect to the query patterns, we may define a score, such as the number of occurrences of these patterns in the document, or the proximity of the given patterns within the document. To control the size of the output, we may also specify a threshold (or a parameter K), so that our task is to report all the documents which match the query with score more than threshold (or respectively, the K documents with the highest scores).

When the documents are strings (without word boundaries), the traditional inverted-index-based solutions may not be applicable. The single pattern retrieval case has been well-solved by [14,9]. When it comes to two or more patterns, the only non-trivial solution for proximity search and common document listing was given by [14], which took $\tilde{O}(n^{3/2})$ space. In this paper, we give the first linear space (and partly succinct) data structures, which can answer multi-pattern queries in $O(\sum |P_i|) + \tilde{O}(t^{1/m} n^{1-1/m})$ time, where t is the number of output occurrences. In the particular case of two patterns, we achieve the bound of $O(|P_1| + |P_2| + \sqrt{nt} \log^2 n)$. We also show space-time trade-offs for our data structures. Our approach is based on a novel data structure called the *weight-balanced wavelet tree*, which may be of independent interest.

1 Introduction

Given a collection $\mathcal{D}$ of string documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n, the *document retrieval problem* is to pre-process this collection, so that when a set of patterns comes as a query, we can efficiently output those documents relevant to this query in some ranked order. This is one of the most fundamental problems in Information Retrieval, and finds applications in web search engines, SQL databases,

* This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082 and US NSF Grants CCF-1017623 and CCF-0621457.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 55–66, 2010.

and genome alignment tools. The ranking of documents is done using relevance scores, which might depend on the frequencies of the patterns in the document, the proximity of occurrences of the patterns as they appear in the document, or simply the static (query-independent) PageRank [2] of the document.

Traditionally, the input documents are split into words and then an inverted index is built over such data. However, in the case of genome data or some Asian texts, there may be no natural word demarcation, so that the inverted index may require too much space, or may only provide limited searching capabilities. Alternatively, full-text indexes like suffix trees and suffix arrays have been successfully used when the query consists of only one pattern [12,14,17,18,9].

For two-pattern queries (listing all documents with both the patterns), the only known solution was given by [14], which requires $\tilde{O}(n^{3/2})$ space and answers a query in $O(|P_1| + |P_2| + \sqrt{n} + t)$ time, where P_1 and P_2 are the query patterns, and t is the size of the output[1]. Recently, Cohen and Porat [3] proposed an elegant framework for the set-intersection problem, through which the index space is reduced greatly to $O(n \log n)$, while admitting slightly worse query time bounds which, instead of having $\sqrt{n} + t$ term, aims to achieve a $\sqrt{nt}$ term. They achieve $O(|P_1| + |P_2| + \sqrt{nt} \log^{2.5} n)$ query bounds for common document listing problem. In this paper, we build a framework which can handle string retrieval under various relevance notions. Our solution is based on the wavelet-tree-based document retrieval scheme given by Välimaki and Mäkinen [18] augmented with other primitives. We introduce a new version of wavelet tree called *weight-balanced wavelet tree* and deploy a multi-way search paradigm on it [3]. Consequently, we show that the index space can further be reduced to linear, while the query can be answered *more quickly* in $O(|P_1| + |P_2| + \sqrt{nt} \log^{1.5} n)$ time. In addition, our framework can easily be extended to handle queries with more than two patterns.

To avoid retrieving many unwanted results, we also focus on retrieving the most *relevant* documents, where either the K highest-scoring documents, or those with scores above some threshold, are reported. This is in contrast to most of the earlier work, which focussed on retrieving all the possible documents.

One of the pressing issues for suffix-tree-based solutions is their space usage. For instance, a simple implementation of Muthukrishnan's index [14] for frequency based retrieval for single pattern takes about 250 times the original text [20] in practice. Such indexes quickly become impractical for massive data sets. With the advent of the field of succinct data structures, compressed text indexes have been developed which can now compete in terms of space with the inverted indexes (which are typically only 5% overhead over the actual text size). In this paper, we also show how to achieve space-time tradeoffs so that our indexes can be made partly succinct.

1.1 Comparison with Previous Work

Document Retrieval Problems on Single Pattern (P):

[1] The notation $\tilde{O}$ ignores poly-logarithmic factors. Precisely, $\tilde{O}(f(n)) \equiv O(f(n) \log^{O(1)} n)$.

- **Document Listing:** List all the documents which contain P.
- **K-mine:** With an extra online parameter K, return all documents which contain at least K occurrences of P.
- **K-repeats:** With an extra online parameter K, list all documents which contain a pair of occurrences of P that are at most distance K apart.

Muthukrishnan [14] gave a linear space structure for the document listing problem which can answer queries in optimal $O(|P| + t)$ time, which improves the previous $O(|P| \log |\mathcal{D}| + t)$ time index by Matias et. al. [12], where t denotes the size of the output. He also gave indexes that answer K-mine and K-repeats in optimal $O(|P| + t)$ time. The space requirements for both cases is $O(n \log n)$.

Sadakane [17] showed how to solve document listing problems using succinct data structures. Similar work done by Välimaki and Mäkinen [18] shows how to achieve partly succinct space by maintaining a wavelet tree of a document array. Here, document array is an array $D_A[1..n]$ such that $D_A[i]$ stores the id of the document where the ith smallest suffix belongs to. For K-mine and K-repeats problems, Hon et. al. [9] gave a unified framework which solves these problems in optimal $O(|P| + t)$ time, while using only linear space. For the top-K version of the above problems, they gave a linear space index with $O(|P| + K \log K)$ query time. Further they showed how to solve these problems in succinct space.

Document Retrieval Problems on Two Patterns (P_1 and P_2):

- **Document Listing:** We need to list all documents where both P_1 and P_2 occur at least once. Muthukrishnan [14] gave an $\tilde{O}(n^{3/2})$-space index that answers a query in $O(|P_1| + |P_2| + \sqrt{n} + t)$ time. Recently, Cohen and Porat [3] proposed an index taking $O(n \log n)$ words of space, based on their solution to the fast set intersection (FSI) problem. They showed that the document listing problem can be reduced to indexing a $\log n$-partition of the suffix ranges and then solving $\log^2 n$ online set intersection queries. Their solution takes $O(n \log n)$ space and $O(|P_1| + |P_2| + \sqrt{nt} \log^{2.5} n)$ query time, where as our new index takes $O(n)$ space and $O(|P_1| + |P_2| + t \log n + \sqrt{nt} \log^{1.5} n)$ time.
- **K-mine:** We need to list all documents that contain at least K total occurrences of P_1 and P_2. Unfortunately, the above suffix-range partition technique for the document listing problem [3] does not work. To the best of our knowledge, no non-trivial solution (except by computing the number of occurrences of P_1 and P_2 explicitly for each document that contains both P_1 and P_2) is known. In the paper we propose an linear space index with $O(|P_1| + |P_2| + t \log n + \sqrt{nt} \log^2 n)$ query time for this problem.
- **K-repeats:** We need to list all documents that contain a pair of occurrences of P_1 and P_2 that are at most K distance apart. Precisely, let $mindist_i(P_1, P_2)$ denote the minimum distance between an occurrence of P_1 and an occurrence of P_2 in document d_i. Our target is to list all documents d_i with $mindist_i(P_1, P_2) \leq K$. Muthukrishnan [14] showed how to reduce this problem to the *close common colors problem*. Consequently, he gave an index with $O(n^{3/2} \log n)$-space, which can answer the query in $O(|P_1| + |P_2| +$

$\sqrt{n}\log n + t)$ time. In the paper we propose an linear space index with $O(|P_1| + |P_2| + K\log^2 n + \sqrt{nK}\log^2 n)$ query time for this problem.

2 Preliminaries

We introduce some data structures which form the building blocks of our indexes. Throughout the paper, we use $\mathcal{D}$ to denote a given collection of documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n. The patterns which come as online query are denoted by P_1 and P_2. We assume that the characters in documents as well as the patterns are taken from an alphabet set Σ of size σ.

2.1 Suffix Trees and Suffix Arrays

Given a text $T[1...n]$, a substring $T[i...n]$ with $1 \leq i \leq n$ is called a suffix of T. The lexicographic arrangement of all n suffixes of T in a compact trie is called the *suffix tree* of T [19], where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a character string and for any node u, path(u) is the string formed by concatenating the edge labels from root to u. For any leaf v, path(v) is exactly the suffix corresponding to v. For a given pattern P, a node u is defined as the *locus node* of P if it is the closest node to the root such that P is a prefix of path(u); such a node can be determined in $O(|P|)$ time. Similarly *generalized suffix tree* (GST) is a compact trie which stores all suffixes of all strings in a given collection $\mathcal{D}$ of strings. For the purpose of our index, we define an extra array D_A called *document array*, such that $D_A[i] = j$ if and only if the $i$the lexicographically smallest suffix is from document d_j.

Suffix array $SA[1...n]$ of a text T is an array such that $SA[i]$ stores the starting position of the ith lexicographically smallest suffix of T [11]. In SA the starting positions of all suffixes with a common prefix are always stored in contiguous range. The suffix range of a pattern P is defined as the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, P is a common prefix of the suffix which starts at $SA[j]$.

2.2 Wavelet Tree

Let $A[1...n]$ be an array of length n, where each element $A[i]$ is a symbol drawn from a set Σ of size σ. The *wavelet tree* (WT) [6] for A is an ordered balanced binary tree on Σ, where each leaf is labeled with a symbol in Σ, and the leaves are sorted alphabetically from left to right. Each internal node W_k represents an alphabet set Σ_k, and is associated with a bit-vector B_k. In particular, the alphabet set of the root is Σ, and the alphabet set of a leaf is the singleton set containing its corresponding symbol. Each node partitions its alphabet set among the two children (almost) equally, such that all symbols represented by the left child are lexicographically (or numerically) smaller than those represented by the right child. For the node W_k, let A_k be a subsequence of A by retaining only those symbols that are in Σ_k. Then B_k is a bit-vector of length $|A_k|$, such

that $B_k[i] = 0$ if and only if $A_k[i]$ is a symbol represented by the left child of W_k. Indeed, the subtree from W_k itself forms a wavelet tree of A_k. Note that B_k is augmented with Raman et al.'s scheme [15] to support constant-time bit-rank and bit-select operations and we do not store A_k's explicitly. The total space requirement of wavelet tree is $n \log \sigma(1 + o(1))$ bits and it can support orthogonal range reporting in $O((|output| + 1) \log \sigma)$ time [10,21].

2.3 Weight-Balanced Wavelet Tree

We propose a modified version of the wavelet tree called *weight-balanced wavelet tree* (WBT), where we maintain the number of 0's and 1's in any bit-vector B_k almost equal. As a result, the two children of a node may not represent equal (or almost equal) distinct number of symbols, but they should represent almost equal total number of symbols. For this, we first consider the symbols in Σ by their number of occurrences (in ascending order) in A, instead of lexicographic order. Consider a node W_k which represents the alphabet $\{\sigma_1, \sigma_2, ...\}$, such that $f_1 \leq f_2 \leq \cdots$, where f_i is the number of occurrences of σ_i in A. Then $\sum f_i = |B_k| = n_k$. The partition of the alphabet for the child nodes is performed as follows. Initialize $n_\ell = n_r = 0$. Pick up the symbol σ with the most occurrences f. If $n_\ell \leq n_r$, we put σ to the left child, and increment n_ℓ by f. Otherwise, if $n_\ell > n_r$, we put σ to the right child, and increment n_r by f. The process continues until all the symbols are distributed. Once B_k of a node is computed, we perform the procedure recursively in the child nodes until the node represents a single symbol. This way of partitioning ensures the following property(proof omitted).

Lemma 1. *Let W_k be a node in WBT at depth δ_k, and B_k denote its associated bit-vector. Let $n_k = |B_k|$. Then we have $n_k \leq 4n/2^{\delta_k}$.* □

An interesting property is that a WBT is inherently compressible(with out compressing the individual bit vectors).

Lemma 2. *The space of a weight-balanced wavelet tree of an array A of size n is $n(H_0(A) + 2)$ bits, where $H_0(A)$ is the 0th-order empirical entropy of A.*

Proof: Let the depth of a leaf corresponding to the symbol σ_i be δ_i. Then σ_i contributes f_i bits in each bit-vector corresponding to the nodes from root to this leaf (excluding the leaf). Hence the contribution of σ_i towards the total space is $f_i \cdot \delta_i$. By Lemma 1, $\delta_i \leq \log(4n/f_i)$. Therefore, the total size of a weight-balanced wavelet tree is at most $\sum f_i \cdot (\log(n/f_i) + 2) = n(H_0(A) + 2)$ bits. □

3 Any-One Index

In this section we describe an index which we call an *Any-One index*. This is a building block for the indexes described in later sections. The input query consists of two patterns P_1 and P_2, and a parameter K. Any-One index for K-mine problem answers whether or not there exists at least one document d_i with

score $= f_{i1} + f_{i2} \geq K$, where f_{ij} denotes the number of occurrences of P_j in d_i. An Any-One index for K-repeats problem answers whether there exists at least one document d_i with $mindist_i(P_1, P_2) \leq K$.

3.1 Index Construction

Let GST be the generalized suffix tree for the collection $\mathcal{D}$, and GSA be the corresponding generalized suffix array. We maintain a document array D_A where $D_A[i] = j$ if the ith lexicographically smallest suffix belongs to d_j. Let WT be the wavelet tree of D_A. We define a parameter $g = \sqrt{n/\beta}$ which is called *group size*. For K-repeats, we also maintain the wavelet trees (WT_i's) over suffix arrays (SA_i's) of individual documents d_i's.

First, starting from left in GST, we group every g contiguous leaves together to form a group. Thus the first group consists of $\ell_1, ..., \ell_g$, the next group consists of $\ell_{g+1}, ..., \ell_{2g}$, and so on, where ℓ_j denotes the jth leaf. The total number of groups is $O(n/g)$. Now for each group we mark the least common ancestor (LCA) of the first and the last leaves. Furthermore, we continue the marking as follows: if two nodes are marked, we mark their LCA also. It can be easily shown that the total number of marked nodes is $O(n/g)$. Now suppose for a marked node u, its subtree contains the leaves $\ell_x, \ell_{x+1}, \ldots, \ell_y$, then we refer the range $[x, y]$ as the *suffix range* corresponding to u.

Lemma 3. *The suffix range $[L, R]$ of any pattern P can be split into a suffix range $[L', R']$ corresponding to some marked node u^*, and two other small ranges $[L, L' - 1]$ and $[R' + 1, R]$ with $L' - L < g$ and $R - R' < g$.* □

Essentially, the suffix range $[L, R]$ of P corresponds to $R - L + 1$ leaves in the GST. This set of leaves can be partitioned into three groups: one that is under the subtree of u^* which contains $R' - L' + 1$ leaves, and the remaining two with those on the left of $\ell_{L'}$ and those on the right of $\ell_{R'}$. We shall refer to the latter two groups of leaves as *fringe leaves*, each group contains fewer than g leaves.

Score Matrix: We make use of score matrices to facilitate the K-mine and K-repeats queries. The score matrix M for each query is a two-dimensional $O(n/g) \times O(n/g)$ matrix with $O(n^2/g^2)$-word space. For K-mine, the matrix M stores the highest possible score value for any document (along with document id) for each pair of marked nodes. Precisely, we set $M(u^*, v^*) = \max_i\{f_{iu^*} + f_{iv^*}\}$, where f_{iu^*} and f_{iv^*} are the number of leaves in the subtree of u^* and v^* whose suffixes are from document d_i. For K-repeats, we store score as the minimum value of *mindist* between P_{u^*} and P_{v^*}, where $P_{u^*} = \mathrm{path}(u^*)$ and $P_{v^*} = \mathrm{path}(v^*)$. Precisely, we set $M(u^*, v^*) = \min_i\{mindist_i(P_{u^*}, P_{v^*})\}$.

Total Space: This index consists of the GST for $\mathcal{D}$ ($O(n)$ words), the wavelet tree of D_A ($O(n)$ words), individual wavelet trees (for K-repeats) WT_i's ($O(\sum |d_i|) = O(n)$ words), and the score matrix ($O(n^2/g^2)$ words). If we choose $g = \sqrt{n/\beta}$, the score matrix takes $O(n\beta)$ words, and hence the total space is $O(n\beta)$.

3.2 Query Answering

The query input consists of two patterns P_1 and P_2, and a parameter K. We search for these patterns in GST, find their locus nodes u_1 and u_2, and obtain their suffix ranges $[L_1, R_1]$ and $[L_2, R_2]$, respectively. From these suffix ranges, we find the suffix ranges $[L'_1, R'_1]$ and $[L'_2, R'_2]$ (as described in Lemma 3), and the corresponding marked LCA nodes u^* and v^*. Now we check for the score matrix value $M(u^*, v^*)$. For K-mine, if $M(u^*, v^*) \geq K$, it will immediately imply that there exists some document d_i with $f_{i1} + f_{i2} \geq K$. However, if $M(u^*, v^*) \leq K$, we cannot conclude at this point that there are no documents satisfying the threshold. This is because there can be some documents in which most occurrences of P_1 and P_2 correspond to the fringe leaves. Hence, we check the documents corresponding to fringe leaves separately using the *Check-Fringe* operation described below. Similarly, for K-repeats, if $M(u^*, v^*) \leq K$, it will immediately imply that there exists some document d_i with $mindist_i(P_1, P_2) \leq K$. Otherwise, we cannot conclude anything yet, and we shall check the fringe leaves. The Check-Fringe operations for K-mine and K-repeats are as follows.

Check-Fringe for K-mine: Firstly we identify the document d_c corresponding to each fringe leaf ℓ_a by traversing the wavelet tree WT of D_A. Let B_1 represent the bit-vector in the root of WT. If $B_1[a] = 0$, we move to the $rank_0(B_1, a)$th position in the left child of B_1, else we move to the $rank_1(B_1, a)$th position in the right child of B_1. Here $rank_0(B, a)$ and $rank_1(B, a)$ represent the number of $0's$ and $1's$ in $B[1..a]$, respectively. This procedure is continued recursively until we reach a leaf node in WT, and the document d_c is identified. Then we can count the number of suffixes in $[L_1, R_1]$ and $[L_2, R_2]$ that belong to d_c. This can be done by traversing the WT of D_A using a similar fashion. Consequently, we obtain the total occurrences of P_1 and P_2 in the document d_c, and check if it is above the threshold K.

Check-Fringe for K-repeats: Similar to the above described method, for each fringe leaf ℓ_f, we first identify the corresponding document d_c. Next, for those suffixes that correspond to d_c within the suffix range $[L_1, R_1]$, we identify the contiguous suffix range $[l_1, r_1]$ in the individual SA of d_c. Such a *translation* operation can be done by traversing the WT of D_A (i.e. by translating $[L_1, R_1]$ to the leaf node corresponding to document d_c). Similarly, we obtain the translated suffix range $[l_2, r_2]$ for $[L_2, R_2]$.

Let $\ell_{f'}$ be the position of ℓ_f after translation and SA_c denotes the suffix array of d_c. Then we need to check the following: (Case 1) If $\ell_{f'} \in [l_1, r_1]$, we check if there exists any $\ell_{f''} \in [l_2, r_2]$ such that $|SA_c[\ell_{f'}] - SA_c[\ell_{f''}]| \leq K$. (Case 2) Otherwise, if $\ell_{f'} \in [l_2, r_2]$, we check if there exists any $\ell_{f''} \in [l_1, r_1]$ such that $|SA_c[\ell_{f'}] - SA_c[\ell_{f''}]| \leq K$. In order to perform these checks efficiently, we maintain the wavelet tree WT_c of SA_c and perform the following orthogonal range searching queries, respectively [10]:

Case 1: Use position range $[l_2, r_2]$, value bound $[SA_c[\ell_{f'}] - K, SA_c[\ell_{f'}] + K]$;
Case 2: Use position range $[l_1, r_1]$, value bound $[SA_c[\ell_{f'}] - K, SA_c[\ell_{f'}] + K]$.

If the output (number of occurrences) of this search is at least one, then d_c is a valid output for the K-repeats query[2].

Analysis: In both K-mine and K-repeats, the Check-Fringe operation involves a wavelet tree traversal per fringe leaf. Since the total number of fringe leaves is less than $4g$, the total time for the Check-Fringe operations is $O(g \log n) = O(\sqrt{n/\beta} \log n)$ [10].

Lemma 4. *We can maintain an $O(n\beta)$-space Any-One index, such that given a query consisting two patterns P_1 and P_2, and a parameter K, we can answer if the number of outputs for the K-mine or the K-repeats queries is at least one, using $O(|P_1| + |P_2| + \sqrt{n/\beta} \log n)$ time.* □

Using an Any-One index, top-1 queries can be answered with the same time bounds. In the next section, we describe our main indexes for answering the original K-mine and K-repeats queries.

4 Efficient Indexes for K-Mine and K-Repeats Problems

We maintain the collection of documents $\mathcal{D}$ in the form of a balanced binary tree, such that the root node represents all documents and further these documents are divided evenly among the child nodes. This procedure is continued recursively until we reach a node (leaf) which represents a single document. We maintain an Any-One index for each $\mathcal{D}_k$, where $\mathcal{D}_k$ is the set of documents represented by an internal node W_k in the binary tree. Query answering is performed as follows: first we check if there exists at least one document in the whole set $\mathcal{D}$, which satisfies our threshold condition. This is performed using the Any-One index at the root of the binary tree. If the answer is YES, we do a multi-way search in both child nodes, which are two mutually exclusive and exhaustive subsets of $\mathcal{D}$. This procedure is continued recursively until we reach a leaf node in the binary tree. At any node, if the Any-One index returns NO, we do not need to continue the recursive step further in its sub-tree.

To maintain the Any-One index for $\mathcal{D}_k$, we will need to maintain the wavelet tree WT_k of the subsequence of the document array whose documents belong to $\mathcal{D}_k$. Instead of storing these wavelet trees separately for each node, we can just store the wavelet tree WT of the original D_A. It is because WT_k is exactly the same as the sub-tree of WT rooted at W_k. Next, let GST_k denote the Generalised suffix tree for $\mathcal{D}_k$, which is a sub-graph of the original GST for $\mathcal{D}$. Therefore, instead of storing all GST_k's, we store only the original GST and the parenthesis encoding [16,1,13] (along with the marked nodes information) of individual GST_k's.

To speed up the query, we hope that the searching of P_1 and P_2 in GST_k can be avoided. That is, when we query the Any-One index for $\mathcal{D}_k$, we hope that the desired marked nodes and the ranges, can be obtained without searching

[2] Here, we shall use the range successor query [21] in the case of K-repeats.

P_1 and P_2 again. This can be performed by translating the suffix ranges using the wavelet-tree (to the node W_k in WT). Let $[L_{1k}, R_{1k}]$ and $[L_{2k}, R_{2k}]$ be the translated suffix ranges in GST_k, then the corresponding marked nodes can be obtained in $O(1)$ (as described in Lemma 3) using parenthesis encoding and marked nodes information.

However, there is a challenge involved. We will need to maintain a score matrix separately for each Any-One index. The total space thus exceeds $O(n\beta)$, so that we will need to use a larger group size (roughly $\sqrt{\log n}$ times) to reduce the total space back to $O(n\beta)$. Consequently, the time spent at any node W_k for Check-Fringe operation could be $O(\sqrt{n_k/\beta}\log^{1.5} n)$, and in the worst case, this could lead to a total of $O(\sqrt{n_k/\beta}\log^{2.5} n)$ time. Note that the $\log n$ blow-up in the worst case comes from the fact that n_k may remain nearly the same if we go down one level in the wavelet tree. In order to achieve faster searching, we use the weight-balanced wavelet tree (WBT) instead, so that we can ensure that n_k is getting reduced exponentially as we traverse down in the tree. Further, we choose a separate blocking factor $g_k = 2^{-\delta_k/2}\sqrt{(n \log n/\beta)}$ for each bit-vector B_k at the node W_k with depth δ_k. In summary, our index consists of the following components.

1. GST: $O(n)$ space;
2. WBT: $O(\sum |d_i| \log(n/|d_i|) + n) + o(n \log |\mathcal{D}|) = O(n \log |\mathcal{D}|)$ bits of space;
3. Parenthesis encodings of GST_k's: It consists of the encodings for GST_k ($4n_k$ bits: suffix tree of a text of length n_k contains at the max $2n_k - 1$ nodes) and for the marked nodes E_k ($2n_k$ bits). Hence the total space is $O(\sum_k n_k) =$ $O(\text{size of } WBT) = O(n \log |\mathcal{D}|)$ bits;
4. Score matrices: The space for score matrix M_k associated with the Any-One index at W_k is $O(n_k^2/g_k^2)$. By Lemma 1, $n_k \leq 4n/2^{\delta_k}$. Now, since g_k is set to $2^{-\delta_k/2}\sqrt{(n \log n/\beta)}$, M_k takes $O(2^{-\delta_k} n\beta/\log n)$ space. The number of nodes at a depth i is almost 2^i. So the total space taken by all score matrices can be bounded as $O(\sum_k (n_k/g_k)^2) = \sum_{i=0}^{\log n} 2^i (2^{-i} n\beta/\log n) = O(n\beta)$.

Lemma 5. *The total space for the above index is $O(n\beta)$ words.* □

4.1 Query Answering

The query consists of input patterns P_1 and P_2 and a parameter K. First we search for these patterns in GST and find their corresponding suffix ranges. Once we get these ranges, we do not need to use the GST any more. Starting from the root node in the weight-balanced wavelet tree, we use the parenthesis encodings to find the marked ancestors u^* and v^*, and check the score matrix entry. If the score matrix entry satisfies the threshold condition (i.e., $M(u^*, v^*) = \max_i\{f_{iu^*} + f_{iv^*}\} \geq K$ for K-mine, and $M(u^*, v^*) = \min_i\{mindist_i(P_1^*, P_2^*)\} \leq K$ for K-repeats), we translate both these ranges into the child nodes of the WBT and continue the procedure recursively until we reach a node which represents a single document. This document is listed as a valid output. During this WBT traversal, whenever the threshold condition is not satisfied at some node, we do

Check-Fringe operations and we will not traverse further down in its sub-tree. In summary, we have the following theorem.

Theorem 1. *We can design an $O(n\beta)$ space index for answering K-mine or K-repeats queries (on two patterns) in $O(|P_1| + |P_2| + t \log n + \sqrt{(nt/\beta)} \log^2 n)$ time, where t is the number of outputs and β is a tunable parameter.*

Proof sketch: The query time mainly consists of the time for tree traversal, the time for the Check-Fringe operations, and the $O(|P_1| + |P_2|)$ time for the initial search in GST. For any leaf node which produces a valid output, we may have checked its sibling node (which can be a leaf), so that the number of leaf nodes visited can be at the most $2t$. Since the height of the tree is $O(\log n)$, the total number of nodes visited (tree traversal time) is $O(t \log n)$. During the tree traversal, if we perform *Check-Fringe* operation at a node, we do not traverse further down from that node. Hence the number of such nodes where we perform *Check-Fringe* operations can be bounded by $O(t \log n)$. Here if we ignore all the nodes in WBT which were not visited, it can be viewed as a binary tree Δ such that all the nodes in WBT where we performed *Check-Fringe* operations become a leaf node in Δ. The number of *Check-Fringe* operations at a leaf ℓ (in Δ at a depth $depth(\ell)$) is $O(2^{-depth(\ell)/2}\sqrt{n \log n/\beta})$ (which is the blocking factor of the GST_k corresponding to that node). Hence the total number for *Check-Fringe* operations can be bounded as $O(\sum_\ell 2^{-depth(\ell)/2}\sqrt{n \log n/\beta})$. We have $\sum_\ell 2^{-depth(\ell)/2} \leq \sqrt{(\sum_\ell 1^2)(\sum_\ell 2^{-depth(\ell)})}$ using Cauchy-Schwarz's inequality[3] and for any binary tree $\sum_\ell 2^{-depth(\ell)} \leq 1$ according to Kraft's inequality. In our case $\sum_\ell 1^2 = O(t \log n)$ and time per *Check-Fringe* operation is $O(\log n)$. Therefore the total time for *Check-Fringe* operations can be bounded by $O(\sqrt{(nt/\beta)} \log^2 n)$. □

In the document listing problem the score is just a binary (YES or NO) and we do not store the document id, hence we choose a smaller blocking factor ($g_k = 2^{-\delta_k/2}\sqrt{n/\beta}$) which improves the query time to $O(|P_1| + |P_2| + t \log n + \sqrt{(nt/\beta)} \log^{3/2} n)$.

5 Top-K Queries

For the sake of similicity, assume all top-K answers are coming from leaf nodes(ignore fringe leaves for now) in WBT. Those K leaves can be found out in $O(K \log^2 n)$ time as follows. First we find the leaf (say ℓ^1)corresponding to the top-1 score, by greedly following the branch with highest score matrix entry. Note that we travel $O(\log n)$ nodes to reach this leaf node. In order to find the leaf(ℓ^2) corresponding to top-2, we *insert* all the nodes (along with the score), which are the siblings of nodes in the path(ℓ^1) into a *max-heap*. Then we do an *extract-max* operation to find the top-2 score and we traverse further down from

[3] $\sum_{i=1}^n x_i y_i \leq \sqrt{\sum_{i=1}^n x_i^2}\sqrt{\sum_{i=1}^n y_i^2}$.

that sibling node greedily to reach ℓ^2. We do this procedure K times (which includes $O(K \log n)$ *insert* and K *extract-max* operations). Hence the total time is $O(K \log^2 n)$.

Now we need to check the fringe leaves which we have ignored before. Using similar analysis as in section 5, this can be performed in $O(\sqrt{nK/\beta} \log^2 n)$ time. For retrieving the top-K answers, we sort all the answers from fringe leaves and the K answers from leaf nodes together and retrieve the Top-K highest scored (unique) documents.

Theorem 2. *We can design an $O(n\beta)$ space index for answering top-K version of the document retrieval queries (on two patterns) in $O(|P_1| + |P_2| + K \log^2 n + \sqrt{(nK/\beta)} \log^2 n)$ time, where β is a tunable parameter.* □

6 Generalization to Multi-pattern Queries

Finally, we show how to extend our indexes to handle multi-pattern queries where query consists of m patterns $P_1, P_2, ..., P_m$. Similar to our two-pattern index, we maintain a generalized suffix tree of all documents and a wavelet tree over the document array. Since the score of a particular document depends upon m patterns, we use an m-dimensional score matrix. In order to maintain space bounds we choose the blocking factor $g_k = \tilde{O}(2^{-\delta_k(1-1/m)} n^{1-1/m} \beta^{-1/m})$. Therefore the number of marked nodes corresponding to GST_k is $O(n_k/g_k) = \tilde{O}(2^{-\delta_k} n\beta)^{1/m}$ and the size of the score matrix $M_k = \tilde{O}((n/g_k)^m) = \tilde{O}(2^{-\delta_k} n\beta)$. By carrying out similar analysis as before (except we use Hölder's inequality[4] instead of Cauchy-Schwarz's inequality), we obtain the following Theorem.

Theorem 3. *For multi-pattern queries, document listing, K-mine, and K-repeats problems can be answered in $O(\sum_{i=1}^{m} |P_i|) + \tilde{O}(t + (t/\beta)^{1/m} n^{1-1/m})$ and Top-K queries in $O(\sum_{i=1}^{m} |P_i|) + \tilde{O}(K + (K/\beta)^{1/m} n^{1-1/m})$ time by maintaining an $O(n\beta)$-space index.* □

Note that the space requirement of our index can be reduced by using a Compressed Suffix Array [7,5] instead of GST and by tuning the score matrix size to $o(n)$. It still remains an open question if a fully succinct space bound (i.e., without $O(n \log |\mathcal{D}|)$ bits term of WBT size) can be achieved for these problems.

References

1. 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)
2. Brin, S., Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks 30(1-7), 107–117 (1998)
3. Cohen, H., Porat, E.: Fast Set Intersection and Two Patterns Matching. In: LATIN (2010)

[4] Hölder's inequality: $\sum_{i=1}^{n} x_i y_i \leq (\sum_{i=1}^{n} x_i^p)^{1/p} (\sum_{i=1}^{n} y_i^q)^{1/q}$, where $1/p + 1/q = 1$.

4. Ferragina, P., Giancarlo, R., Manzini, G.: The Myriad Virtues of Wavelet Trees. Inf. and Comp. 207(8), 849–866 (2009)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms (TALG) 3(2) (2007)
6. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: SODA, pp. 841–850 (2003)
7. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SICOMP 35(2), 378–407 (2005)
8. Hon, W.K., Shah, R., Vitter, J.S.: Ordered Pattern Matching: Towards Full-Text Retrieval. Tech Report TR-06-008, Dept. of CS, Purdue University (2006)
9. Hon, W.K., Shah, R., Vitter, J.S.: Space-Efficient Framework for Top-k String Retrival Problems. In: FOCS, pp. 713–722 (2009)
10. Mäkinen, V., Navarro, G.: Rank and Selected Revisited and Extended. TCS 387(3), 332–347 (2007)
11. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SICOMP 22(5), 935–948 (1993)
12. 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)
13. Munro, J.I., Raman, V.: Succinct Representation of Balanced Parentheses and Static Trees. SICOMP 31(3), 762–776 (2001)
14. Muthukrishnan, S.: Efficient Algorithms for Document Retrieval Problems. In: SODA, pp. 657–666 (2002)
15. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. TALG 3(4) (2007)
16. Sadakane, K.: Compressed Suffix Trees with Full Functionality. TCS, 589–607 (2007)
17. Sadakane, K.: Succinct Data Structures for Flexible Text Retrieval Systems. JDA 5(1), 12–22 (2007)
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)
19. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. Switching and Automata Theory, pp. 1–11 (1973)
20. Wu, S.B., Hon, W.K., Shah, R.: Efficient Index for Retrieving Top-k Most Frequent Documents. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 182–193. Springer, Heidelberg (2009)
21. Yu, C.C., Hon, W.K., Wang, B.F.: Efficient Data Structures for the Orthogonal Range Successor Problem. In: Ngo, H.Q. (ed.) COCOON 2009. LNCS, vol. 5609, pp. 96–105. Springer, Heidelberg (2009)

Colored Range Queries and Document Retrieval

Travis Gagie[1,⋆], Gonzalo Navarro[1,⋆], and Simon J. Puglisi[2]

[1] Dept. of Computer Science, Univt. of Chile
{tgagie,gnavarro}@dcc.uchile.cl
[2] School of Computer Science and Information Technology
Royal Melbourne Institute of Technology
simon.puglisi@rmit.edu.au

Abstract. Colored range queries are a well-studied topic in computational geometry and database research that, in the past decade, have found exciting applications in information retrieval. In this paper we give improved time and space bounds for three important one-dimensional colored range queries — colored range listing, colored range top-k queries and colored range counting — and, thus, new bounds for various document retrieval problems on general collections of sequences. Specifically, we first describe a framework including almost all recent results on colored range listing and document listing, which suggests new combinations of data structures for these problems. For example, we give the fastest compressed data structures for colored range listing and document listing, and an efficient data structure for document listing whose size is bounded in terms of the high-order entropies of the library of documents. We then show how (approximate) colored top-k queries can be reduced to (approximate) range-mode queries on subsequences, yielding the first efficient data structure for this problem. Finally, we show how a modified wavelet tree can support colored range counting in logarithmic time and space that is succinct whenever the number of colors is superpolylogarithmic in the length of the sequence.

1 Introduction

A *range query* on a sequence $S[1, n]$ of elements in $[1, \sigma]$ takes as arguments two indices i and j and returns information about $S[i, j]$. This information could be, for example, the minimum or maximum value in $S[i, j]$ [12], the element with a specified rank in sorted order [15] (e.g., the median [7]), the mode [17], a complete list of the distinct elements [31], the frequencies of the elements [35], a list of the k most frequent elements for a given k [20], or the number of distinct elements [6]. In this paper, motivated by problems in document retrieval, we consider the latter three kinds of problems, which are often referred to as "colored" range queries: colored range listing (with or without color frequencies), colored range top-k queries, and colored range counting. These have been

⋆ Partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 67–81, 2010.

associated, respectively, to very relevant document retrieval queries on general texts [31,35,37,20,15,12,9]: listing the documents where a pattern appears (possibly computing term frequencies), finding the most relevant documents to a query (under a $tf \times idf$ scheme, for example), and computing document frequencies. Such techniques have been shown to be competitive [9], even beating classical inverted indexes on natural-language texts.

In Section 2 we describe a framework that includes almost all recent results on colored range listing and the related problem of document listing. This framework suggests new combinations of data structures that yield interesting new bounds, including the fastest compressed data structures for colored range listing and an efficient data structure for document listing whose space occupancy is bounded in terms of the higher-order entropies of the library of documents. In Section 3 we describe what seems to be the first data structure to support efficient, general approximate colored range top-k queries. By "approximate" we mean that we are given an $\epsilon > 0$ with S and we guarantee that no element we do not list occurs more than $1 + \epsilon$ times more often in the range than any element we list. Finally, in Section 4 we describe a new solution to the colored range counting problem, reducing the space bound from $\mathcal{O}(n \log n)$ bits to $n \log \sigma + \mathcal{O}(n \log \log n)$ bits without changing the $\mathcal{O}(\log n)$ time bound. The improvements for colored range queries we present in Sections 3 and 4 are not competitive with the state of the art when mapped to the more specific problem of document retrieval. However, as we discuss in Section 5, data structures for general colored range queries can be applied to information retrieval scenarios that specialized document-retrieval data structures cannot.

2 Listing

Related work. The problem of colored range listing (CRL) is to preprocess a given sequence $S[1, n]$ over $[1, \sigma]$ such that later, given a range $S[i..j]$, we can quickly list all the distinct elements ("colors") in that range. Almost all recent data structures for CRL (and the related problem of document listing) are based on a key idea by Muthukrishnan [31] (see [23] for older work). He defined $C[1, n]$ to be the array in which $C[j]$ is the largest value $i < j$ such that $S[i] = S[j]$, or 0 if there is no such i, so that $S[\ell]$ is the first occurrence of a color in $S[i..j]$ if and only if $i \leq \ell \leq j$ and $C[\ell] < i$. He showed how, if we store C in an $\mathcal{O}(n \log n)$-bit data structure due to Gabow, Bentley and Tarjan [14] that supports $\mathcal{O}(1)$-time range-minimum queries (RMQs), we can quickly find all the values in $C[i..j]$ less than i and, thus, list all the colors in $S[i..j]$. To do this, we find the minimum value $C[\ell]$ in $C[i..j]$; if it is less than i, then we output $S[\ell]$ and recurse on $S[i..\ell - 1]$ and $S[\ell + 1..j]$. Altogether, Muthukrishnan's CRL data structure uses $\mathcal{O}(n \log n)$ bits and $\mathcal{O}(1)$ time per color reported.

Muthukrishnan gave his solution to the CRL problem as part of a solution to the problem of document listing (DL), in which we are given a library of documents and asked to preprocess them such that later, given a pattern, we can quickly list all the distinct documents containing that pattern (see [29] for older work). Let $T[1, n]$ be the concatenation of the D documents. Muthukrishnan

defined the array $E[1,n]$ such that $E[i]$ is the document containing the starting position of the lexicographically ith suffix in T. If we store a suffix tree [38,1] for T then, given a pattern, we can quickly find the lexicographic ranks i and j of the first and last suffixes starting with the pattern. This is equivalent to finding the range $A[i..j]$ in the suffix array [27] A for T that lists the starting positions of all the suffixes of T that start with the pattern. Once we know i and j, we can implement a DL query as a CRL query on $E[i..j]$. Altogether, Muthukrishnan's DL data structure uses $\mathcal{O}(n \log n)$ bits and $\mathcal{O}(m + \mathsf{ndoc})$ time to list the ndoc documents containing a pattern of length m.

Sadakane [35] gave a slower but smaller version of Muthukrishnan's DL data structure, in which he replaced Gabow, Bentley and Tarjan's data structure by a $4n + o(n)$ bit index that, given a range $C[i..j]$, in $\mathcal{O}(1)$ time and without consulting C returns the position of the minimum value in that range (but not the value itself). He also replaced the suffix tree by a compressed suffix array (CSA) for T and showed how the CSA and a bit vector $V[1,n]$ can simulate access to E: 1s in V mark the positions in T where the documents start; then, for $1 \leq \ell \leq n$, $E[\ell] = \mathsf{rank}_1(V, \mathsf{CSA}[\ell])$, where $\mathsf{rank}_1(V,r)$ is the number of 1s in $V[1..r]$. It takes $D \log(n/D) + \mathcal{O}(D) + o(n)$ bits to store V such that a rank query takes $\mathcal{O}(1)$ time [33]. Sadakane did not store C at all so, when listing the distinct documents containing a pattern, he used a D-bit string to mark which documents he had already listed. He used a recursion similar to Muthukrishnan's, stopping whenever it finds a document already reported.

Altogether, Sadakane's DL data structure uses $|\mathsf{CSA}| + 4n + D \log(n/D) + \mathcal{O}(D) + o(n)$ bits and $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc} \cdot \mathsf{lookup}(n))$ time, where $\mathsf{search}(m)$ is the time to find the range $\mathsf{CSA}[i..j]$ containing the starting positions of suffixes beginning with the pattern and $\mathsf{lookup}(n)$ is the time to compute $\mathsf{CSA}[\ell]$ for any ℓ. (There are a number of CSA implementations, allowing various space/time tradeoffs [32].) He used $|\mathsf{CSA}| + 4n + o(n)$ additional bits for data structures to compute the pattern's frequency in each document, increasing the time bound to $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc}(\mathsf{lookup}(n) + \log\log \mathsf{ndoc}))$ (assuming $\mathsf{lookup}(n)$ is also the time to find $\mathsf{CSA}^{-1}[\ell]$, where CSA^{-1} is the inverse permutation).

Välimäki and Mäkinen [37] gave an alternative slower-but-smaller version of Muthukrishnan's CRL data structure, in which they used a $2n + o(n)$ bit, $\mathcal{O}(1)$ time RMQ succinct index due to Fischer and Heun [13] that requires access to C. Välimäki and Mäkinen showed how access to C can be implemented by rank and select queries on S; specifically, for $1 \leq \ell \leq n$, $C[\ell] = \mathsf{select}_{S[\ell]}(S, \mathsf{rank}_{S[\ell]}(S, \ell) - 1)$, where $\mathsf{select}_a(S,r)$ is the position of the rth occurrence of a in S. Välimäki and Mäkinen stored S in a multiary wavelet tree [10], which takes $nH_0(S) + o(n) \log \sigma$ bits and $\mathcal{O}(1 + \log\sigma / \log\log n)$ time; when σ is polylogarithmic in n, it takes $nH_0(S) + o(n)$ bits and $\mathcal{O}(1)$ time. The 0-th order empirical entropy $H_0(S) = \sum_a \frac{\mathsf{occ}(a,S)}{n} \log \frac{n}{\mathsf{occ}(a,S)}$, where $\mathsf{occ}(a,S)$ is the number of times element a occurs in S, is the Shannon entropy of the distribution of elements in S.

Altogether, their CRL data structure takes $nH_0(S) + 2n + o(n) \log \sigma$ bits and $\mathcal{O}(1 + \log\sigma / \log\log n)$ time per reported color. Combining this data structure with a CSA yields a DL data structure that takes $|\mathsf{CSA}| + n \log D + 2n + o(n) \log D$

bits and $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc}(1 + \log D/\log\log n))$ time. They also showed how to compute the pattern's frequency in a document d using two rank queries on E, $\mathsf{rank}_d(E, j) - \mathsf{rank}_d(E, i-1)$. Since multiary wavelet trees support rank queries in the same time as accesses, it follows that reporting the pattern's frequency in all the documents does not affect their time and space bounds. Finally, they noted that, using one select query per occurrence, they can list the positions of the pattern's occurrences in a specified document.

Gagie, Puglisi and Turpin [15] showed that a binary wavelet tree [18] can be used to compute range quantile queries on S in $\mathcal{O}(\log\sigma)$ time, and that these queries can be used to enumerate the distinct elements in $S[i..j]$, eliminating the need for RMQs. A binary wavelet tree for S takes $nH_0(S) + o(n)\log\sigma$ bits and supports access, rank and select in $\mathcal{O}(\log\sigma)$ time; therefore, by itself it is a CRL data structure that takes $\mathcal{O}(\log\sigma)$ time per reported element. Combining a wavelet tree for E with a CSA for T, we obtain a DL data structure that takes $|\mathsf{CSA}| + n\log D + o(n)\log D$ bits and $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc}\log D)$ time.

Hon, Shah and Vitter [20] described a solution to DL similar to Sadakane's but removing the $\Theta(n)$-bit space term. They pack $\log^\epsilon n$ consecutive cells of C into a block and build the RMQ data structure on the block minima (so it takes $\mathcal{O}(n/\log^\epsilon n)$ bits of space), and tries to report (avoiding repetitions) all the documents in the block that holds the minimum. Their whole data structure takes $|\mathsf{CSA}| + D\log(n/D) + \mathcal{O}(D) + o(n)$ bits and answers queries in time $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc}\log^\epsilon n \cdot \mathsf{lookup}(n))$, for any constant $\epsilon > 0$.

They can also return the number of times the pattern occurs in any document by using, like Sadakane, one CSA_d local to each document d. These add up to other $|\mathsf{CSA}|$ extra bits. To find out how many times document $d = E[\ell]$, $i \le \ell \le j$, appears in $E[i..j]$, it maps ℓ to position $p = \mathsf{CSA}[\ell] - \mathsf{select}_1(V, d) + 1$ within document d, and then to $\ell' = \mathsf{CSA}_d^{-1}[p]$. This is the first lexicographic occurrence of the pattern in CSA_d. The last occurrence is found by an exponential search and then binary search on $\mathsf{CSA}_d[\ell'..]$, for the largest c such that $\mathsf{CSA}^{-1}[\mathsf{CSA}_d[\ell' + c] + \mathsf{select}_1(V, d) - 1)] \le j$. Then the answer, $c + 1$, is obtained in time $\mathcal{O}(\mathsf{lookup}(n)\log c) = \mathcal{O}(\mathsf{lookup}(n)\log n)$.

New tradeoffs. All the previous solutions have essentially the same ingredients: for CRL, access to S, distinct color enumeration on S (implemented via RMQs on C or range quantile queries on S) and, to count the number of times each color occurs, rank on S; for DL, a suffix tree or CSA for T, access to E, distinct document enumeration on E and, to report the pattern's frequency in each document, rank on E. Solutions for CRL can be used for DL with the addition of a CSA for T, setting $S = E$ and $\sigma = D$. Recall that Sadakane's [35] and Hon, Shah and Vitter's [20] solutions for DL implement access to E using a CSA and bit vector V on T, so they cannot be used for general CRL.

Our main contribution in this section is the observation that, using new data structures for access, color enumeration and rank, we can obtain interesting new bounds for both CRL and DL. This is formalized in the next theorem.

Theorem 1. *Suppose we are given a sequence $S[1, n]$ over $[1, \sigma]$ and we store any data structure supporting access on S in time t_{acc} and any structure supporting*

distinct enumeration in a range of S in time t_{enum} per element (and any structure supporting rank on S in time t_{rank} if computing frequencies is desired). Then later, given i and j, we can list the distinct elements in $S[i..j]$ in time $\mathcal{O}(t_{\mathsf{acc}} + t_{\mathsf{enum}})$ per reported element, plus $\mathcal{O}(t_{\mathsf{rank}})$ to list its frequency in $S[i..j]$.

Corollary 1. *Given a concatenation $T[1, n]$ of D documents, we can store either*

- *the CSA for T and data structures supporting access, enumeration and rank on the corresponding array $E[1, n]$ in times t_{acc}, t_{enum} and t_{rank}, or*
- *the CSA for T, a bit vector occupying $D \log(n/D) + \mathcal{O}(D) + o(n)$ bits, and data structures supporting enumeration and rank on E as above,*

such that, given a pattern of length m, we can list the distinct documents containing that pattern in time $\mathcal{O}(\mathsf{search}(m))$ plus $\mathcal{O}(t_{\mathsf{acc}} + t_{\mathsf{enum}} + t_{\mathsf{rank}})$ per reported document, where $t_{\mathsf{acc}} = \mathsf{lookup}(n)$ in the second case and t_{rank} is required only in order to list the frequencies of the documents.

A selection of these data structures is shown in Table 1. If we choose a set of rows covering support for access and enumeration (and rank) then we can answer CRL queries (and return the frequency of each color). The space bound is the sum of the space bounds and the time bound per reported color is $\mathcal{O}(t_{\mathsf{acc}} + t_{\mathsf{enum}} + t_{\mathsf{rank}})$, the latter term for computing frequencies. For example,

2+9: is Välimäki and Mäkinen's scheme [37].
1: is the scheme by Gagie, Puglisi, and Turpin [15].
3+9+10: combining Ferragina and Venturini's [11] data structure with Fischer's [12] succinct index for RMQ and Grossi, Orlandi and Raman's [19] succinct index for rank gives a solution for CRL that takes $nH_k(S) + 2n + o(n) \log \sigma + n\, o(\log \sigma)$ bits and $\mathcal{O}(1)$ time per reported color, matching the time of Muthukrishnan's $\mathcal{O}(n \log n)$-bit space solution [31]. The k-th order empirical entropy $H_k(S)$ measures the compressibility of S when we use contexts of length k; see [28] for details. The frequency of any color can be obtained in time $\mathcal{O}(\log \log \sigma)$.
6+9: is similar to the above but the $n\, o(\log \sigma)$ space term is avoided, as the structure by Barbay, Gagie, Navarro and Nekrich [4] computes rank as well. This becomes the *least-space* reported solution to CRL, listing in $\mathcal{O}(1)$ time.
(4 or 5)+9: combining Barbay et al.'s [4] access and rank data structure with Fischer's [12] succinct index for RMQ gives a solution for CRL that takes $nH_0(S) + 2n + o(n)(H_0(S) + 1)$ bits and $\mathcal{O}(\log \log \sigma)$ bits per reported color and its frequency (variant *4*), which is the *fastest* compressed solution when we want all the frequencies; or $\mathcal{O}(1)$ per reported color and $\mathcal{O}(\log \log \sigma \log \log \log \sigma)$ per reported frequency (variant *5*), which trades frequency reporting time for constant-time listing.
[35]*+9:* replacing Sadakane's [35] RMQ data structure with the one by Fischer [12] improves Sadakane's space bound by $2n$ bits.
[20]*+10:* replacing Hon, Shah and Vitter's [20] CSA_d structures by that of Grossi et al. [19] speeds up counting document frequencies (here $t_{\mathsf{acc}} = \mathsf{lookup}(n)$).

Table 1. Space and time bounds for some data structures supporting operations on $S[1,n]$ over $[1,\sigma]$. The $\mathcal{O}(\sigma \log n)$ extra bits of wavelet trees [18,10] can be avoided [26] so we have not included it. The space bound in rows *3* and *6* holds for $k = o(\log_\sigma n)$. In rows *7* and *8*, g is the size (in bits) of a given context-free grammar generating S and only S and α is the inverse Ackermann function. The succinct index for RMQ in row *9* does not need access to the underlying data (i.e., C), but the succinct index for rank in row *10* does (i.e., S), hence the time of the latter depends on t_{acc}. Due to space constraints, here we write $\log^{[2]}$ and $\log^{[3]}$ for log log and log log log.

row	source	space (in bits)	t_{acc}	t_{enum}	t_{rank}
1	[18,15]	$nH_0(S) + o(n)\log\sigma$	$\mathcal{O}(\log\sigma)$	$\mathcal{O}(\log\sigma)$	$\mathcal{O}(\log\sigma)$
2	[10, Cor. 3.3]	$nH_0(S) + o(n)\log\sigma$	$\mathcal{O}\left(1+\frac{\log\sigma}{\log^{[2]} n}\right)$		$\mathcal{O}\left(1+\frac{\log\sigma}{\log^{[2]} n}\right)$
3	[11]	$nH_k(S) + o(n)\log\sigma$	$\mathcal{O}(1)$		
4	[4, Thm. 1]	$nH_0(S) + o(n)(H_0(S)+1)$	$\mathcal{O}\left(\log^{[2]}\sigma\right)$		$\mathcal{O}\left(\log^{[2]}\sigma\right)$
5	[4, Thm. 1]	$nH_0(S) + o(n)(H_0(S)+1)$	$\mathcal{O}(1)$		$\mathcal{O}\left(\log^{[2]}\sigma \log^{[3]}\sigma\right)$
6	[4, Thm. 2]	$nH_k(S) + o(n)\log\sigma$	$\mathcal{O}(1)$		$\mathcal{O}\left((\log^{[2]}\sigma)^2 \log^{[3]}\sigma\right)$
7	[5, Thm. 1]	$\mathcal{O}(g\,\alpha(g))$	$\mathcal{O}(\log n)$		
8	[5, Thm. 1]	$\mathcal{O}(g)$	$\mathcal{O}\left(\log n \log^{[2]} n\right)$		
9	[12, Thm. 1]	$2n + o(n)$		$\mathcal{O}(1)$	
10	[19, Thm. 5(a)]	$n\,o(\log\sigma)$			$\mathcal{O}\left(t_{\mathsf{acc}} \log^{[2]}\sigma\right)$

The $|\mathsf{CSA}|$ space is exchanged by $n\,o(\log d)$ bits, which can be less or more. We can then also discard the D-bit string marking documents used by both solutions [35,20] and replace it with rank queries on E.

(7 or 8)+9+10: combines Bille, Landau and Weimann's [5] grammar-based data structure for access, Fischer's [12] succinct index for RMQ, and Grossi et al.'s [19] succinct index for rank. González and Navarro [16] showed how to build a grammar generating an array that, together with some other small data structures, gives access to the suffix array (SA) A. Building Bille, Landau and Weimann's data structure for this grammar, we obtain a $\mathcal{O}(\log n)$-time data structure for DL whose size is bounded in terms of the high-order entropies of the library of documents. This is described next.

Theorem 2. *Given a concatenation $T[1,n]$ of D documents, we can store T in*

$$|\mathsf{CSA}| + 2n + o(n) + n\,o(\log D) + \mathcal{O}\left(\left(n\min(H_k(T),1) + D\right)\log\left(\frac{1}{\min(H_k(T),1) + D/n}\right)\alpha(n)\log n\right)$$

bits, for any $k \le \alpha\log_\tau n$, constant $0 < \alpha < 1$ and τ the size of the alphabet of T. Then given a pattern of length m, we can list the distinct documents contain-

ing that pattern in time $\mathcal{O}(\mathsf{search}(m))$ *plus* $\mathcal{O}(\log n)$ *to list each document, plus* $\mathcal{O}(\log n \log\log D)$ *to give its frequency.*

Proof. González and Navarro's algorithm takes advantage of the so-called *runs* of the SA, that is, areas $A[i..i+\ell]$ such that there is some other area $A[j..j+\ell]$ where $A[j+k] = A[i+k]+1$ for all $0 \le k \le \ell$. Let R be the number of runs with which the SA can be covered; it is known that $R \le \min(n, nH_k(T)+\sigma^k)$ for any k [25]. González and Navarro represent the SA differentially so that these areas become true repetitions, and use a grammar-based compression algorithm that represents A using at most $R\log(n/R)$ rules. We note that, in E, those SA runs become identical areas $E[i..i+\ell] = E[j..j+\ell]$ except for at most D cells where the document number can change when we advance one text position. It follows that, by applying the same compression algorithm [16] to E we obtain at most $(R+D)\log(n/(R+D))$ rules and hence the space given in the theorem. □

As a final note applying only to document collections, Sadakane's CSA [34] essentially represents a function Ψ such that $A[\Psi(i)] = A[i]+1$, which is stored in compressed form and any value computed in constant time. Thus one advances virtually in the text by successively applying Ψ. Now assume we sample E with a step r such that, for any i, $E[\Psi^j(i)]$ is sampled for some $0 \le j < r$. Then one computes any $E[i]$ value in time $\mathcal{O}(r)$ by following Ψ until hitting a sampled entry, whose value will be the same as $E[i]$ if we also sample every document end in the text collection. The space is $\mathcal{O}((n/r)\log r) + (n/r)\log D$ for a bitmap marking the sampled cells and an array with the sampled values, respectively. For example, using $r = \log D$ yields access to E (though not rank nor select on it) in the same time of a binary wavelet tree, within bit space $n+o(n)$. Depending on the relation between n and D, this can be an interesting alternative to using lookup and marking the document beginnings [35].

3 Top-k Queries

Improving the current-best solution for documents. Recently, Hon, Shah and Wu [21] described a data structure that stores a library T of D documents of total length n in $\mathcal{O}\big(n\log^2 n\big)$ bits such that later, given a pattern of length m and an integer $k \ge 1$, we can find the k documents that contain that pattern most frequently, in $\mathcal{O}(m + \log n \log\log n + k)$ time. We call this the document top-k problem (DTK). Hon, Shah and Vitter [20] gave solutions for DTK that store T in $\mathcal{O}(n\log n)$ bits and answer queries in $\mathcal{O}(m + k\log k)$ time, or in $2|\mathsf{CSA}| + o(n) + D\log(n/D) + \mathcal{O}(D)$ bits and $\mathcal{O}\big(\mathsf{search}(m) + k\log^{3+\epsilon} n \cdot \mathsf{lookup}(n)\big)$ time.

The last solution consists of a tree τ_k built for each k power of 2. For τ_k they divide E into blocks of size $z = k\log^{2+\epsilon} n$, and τ_k consists of the suffix tree nodes that are lowest common ancestors (lca) of end points of blocks, and transitively all the lcas of pairs of those nodes. At each node, τ_k stores the k most frequent documents within the whole blocks it contains, and their frequencies. Thus each τ_k requires $\mathcal{O}((n/z)k\log n) = \mathcal{O}\big(n/\log^{1+\epsilon} n\big)$ bits, and all the trees together add up to $\mathcal{O}(n/\log^{\epsilon} n)$ bits. At query time, to find the top-k documents in $E[i..j]$,

they increase k to the next power of 2 and find the highest node of τ_k whose range $[i'..j']$ is contained in $[i..j]$. They show that $i' - i \leq z$ and $j - j' \leq z$ by the *lca* properties. Then the query is answered by considering the k candidates given by τ_k and the $\mathcal{O}(z)$ further candidates found at positions of $E[i..i'-1]$ and $E[j'+1..j]$, for each of which they compute the frequency. The total time, considering priority queue operations, is $\mathcal{O}(\mathsf{search}(m) + z(t_{\mathsf{rank}} + \log k) + k \log k) = \mathcal{O}\big(\mathsf{search}(m) + k \log^{3+\epsilon} n \cdot \mathsf{lookup}(n)\big)$. This time bound can be improved to $\mathcal{O}\big(\mathsf{search}(m) + k \log D \log(D/k) \log^{1+\epsilon} n \cdot \mathsf{lookup}(n)\big)$ by noticing that (a) one needs only $\mathcal{O}(\log D)$ powers of 2 for k since $k \leq D$; (b) one can store the top-k elements in the τ_k trees and not their frequency. The k frequencies can be computed at query time without changing the time complexity since $k = o(z)$. Thus the k documents out of D can be stored in increasing order and as gamma-encoded differences, taking $\mathcal{O}(k \log(D/k))$ bits. Therefore we can use smaller blocks of size $z = k \log D \log(D/k) \log^{\epsilon} n$, which are processed faster, and still have $O(n/\log^{\epsilon} n) = o(n)$ space for the structure.

In addition, as shown in Section 2, by replacing the $|\mathsf{CSA}|$ bits of their solution for computing frequencies, by Grossi et al.'s [19] succinct index for rank, we achieve a new space bound of $|\mathsf{CSA}| + o(n) + D \log(n/D) + \mathcal{O}(D) + n\, o(\log D)$ bits, which can be better or worse than before, but the time is reduced to $\mathcal{O}(\mathsf{search}(m) + k \log D \log(D/k) \log^{\epsilon} n \cdot \mathsf{lookup}(n))$, for any ϵ (log-logarithmic terms disappear by adjusting ϵ).

An approximate solution to the general problem. We now give a solution to the approximate colored range top-k problem (CRTK), which asks us to preprocess a given sequence S such that later, given a range $S[i..j]$ and an integer $k \geq 1$, we can return an approximate list of the k elements ("colors") that occur most frequently in that range. We do not know of any previous efficient solutions to this problem, although finding the k most frequent or important items in various data sets and models is a well studied problem and there has been work on interesting special cases (see, e.g., [22,24]).

Greve, Jørgensen, Larsen and Truelsen [17] recently gave a data structure that, for any $\epsilon > 0$, stores S in $\mathcal{O}((n/\epsilon) \log n)$ bits such that we can find an element such that no element is more than $1 + \epsilon$ times more frequent in $S[i, j]$, in $\mathcal{O}(\log(1/\epsilon))$ time. Thus, their data structure solves the approximate CRTK problem for $k = 1$, which is called the approximate range-mode problem. We can assume their data structure also returns the frequency of the approximate mode in $S[i..j]$, since adding a rank data structure for S allows us to compute this and does not change their space bound. We show how to use their data structure as a building block to store S in $\mathcal{O}((n/\epsilon)(H_0(S) + 1) \log n)$ bits such that, given an integer k, we can approximately list the k most common elements and their frequencies in $\mathcal{O}(k \log \sigma \log(1/\epsilon))$ time.

We first build a binary wavelet tree for S [18]. This is a balanced tree where each node represents a range of $[1, \sigma]$: the root represents the full range, the leaves the individual symbols, and the children of a node represent the left and right halves of the node's interval. For each node v, let S_v be the subsequence of S consisting of characters labelling the leaves in v's subtree. The original wavelet tree does not

store S_v, but just a bitmap B_v of length $|S_v|$ telling whether each $S_v[i]$ went to the left or right child. Rank and select over those bitmaps allow accessing any $S[i]$, as well as computing $\mathsf{rank}_a(S,i)$ and $\mathsf{select}_a(S,i)$, in time $\mathcal{O}(\log\sigma)$, and the overall space is $n\log\sigma(1+o(1))$. It can also track any range $S[i..j]$ down to any node [26].

Here we do store each subsequence S_v in an instance of Greve et al.'s approximate range-mode data structure. For now, assume $[i,j]=[1,n]$ and that Greve et al.'s data structure returns the exact mode, rather than an approximation. Notice that, if $a_1,\ldots,a_{k'}$ are the k' most frequent elements and v is an ancestor of the leaf labelled $a_{k'}$ but not of those labelled $a_1,\ldots,a_{k'-1}$, then $a_{k'}$ is the mode in S_v. Let V be the set of ancestors of $a_1,\ldots,a_{k'-1}$ and let V' be the set of nodes whose siblings are in V but who are not in V themselves; V' contains the root of the tree if V is empty. We can find $a_{k'}$ by finding the mode of S_v for each $v\in V'$, finding their frequencies in S, and taking the most frequent.

We keep the modes for each $v\in V'$ in a priority queue, ordered by their frequencies and with the corresponding nodes of the wavelet tree as auxiliary data. Notice $a_{k'}$ is the head of the queue, so we can find and output it in $\mathcal{O}(1)$ time; let v be the corresponding node, i.e., the node in V' such that the mode of S_v is $a_{k'}$. To update the queue, we delete $a_{k'}$, perform range-mode queries on the siblings of nodes on the path from v to the leaf labelled $a_{k'}$, and add the modes to the queue. There are always $\mathcal{O}(k\log\sigma)$ nodes in the queue (the tree is of height $\mathcal{O}(\log\sigma)$) so, if we use a priority queue allowing $\mathcal{O}(\log(k\log\sigma))=\mathcal{O}(\log\sigma)$ time deletion and $\mathcal{O}(1)$ time insertion [8], then we can find the k most frequent elements in S in $\mathcal{O}(k\log\sigma\log(1/\epsilon))$ time. We can deal with general i and j by using the wavelet tree to compute the appropriate range in each subsequence [26]. As for the approximation, it is clear that, whenever we output an element, none of the elements not output yet can be more than $1+\epsilon$ times more frequent.

If we use a Huffman-shaped wavelet tree, then calculation shows that our space usage is $\mathcal{O}((n/\epsilon)(H_0(S)+1)\log n)$ bits. However, since a Huffman tree can be very deep (height $n-1$ for a very skewed distribution), this would compromise our time bound. Therefore, we use a an $\mathcal{O}(\log\sigma)$-restricted Huffman tree [30], which yields both the space and time bounds we want.

Theorem 3. *Given a sequence $S[1,n]$ over an alphabet of size σ and a constant $\epsilon>0$, we can store S in $\mathcal{O}((n/\epsilon)(H_0(S)+1)\log n)$ bits such that, given i, j and k, we can list k distinct elements such that no element is more than $1+\epsilon$ times more frequent in $S[i..j]$ than any of the ones we list, in $\mathcal{O}(k\log\sigma\log(1/\epsilon))$ time.*

This $(1+\epsilon)$-approximation makes sense in information retrieval scenarios, where top-k queries are understood to be just approximations to the ideal answer.

Corollary 2. *Given a set of D documents of total length n and a constant $\epsilon>0$, we can store them in $\mathcal{O}((n/\epsilon)\log n\log D)$ bits such that, given a pattern of length m and k, we can list k distinct documents such that no document contains that pattern more than $1+\epsilon$ times as often as any of the ones we list, in a total of $\mathcal{O}(m+k\log D\log(1/\epsilon))$ time.*

Although Corollary 2 is weaker than Hon, Shah and Vitter's uncompressed result, our approach applies to the general colored range query problem, and may

be faster than what the upper bound suggests. For example, if the documents are webpages sorted lexicographically by URL, then it is more likely that interesting patterns will occur often in clusters of documents than widely spread out [36,39]. In this case, leaves in a balanced wavelet tree for E that are labelled with the k distinct documents that contain the pattern most often, are likely to share many ancestors; if so, our data structure can speed up to $\mathcal{O}(m + k \log k \log(1/\epsilon))$.

The K-mining problem. Muthukrishnan [31] defined (document) K-mining (DKM) as the problem of finding all the documents in the library that contain a given pattern at least K times. He gave an $\mathcal{O}\big(n \log^2 n\big)$-bit data structure that, given K and a pattern of length m, answers queries in $\mathcal{O}(m)$ time plus $\mathcal{O}(1)$ time per reported document. Hon, Shah and Wu [21] noted that we can use binary search with a DTK data structure to solve DKM, with an $\mathcal{O}(\log n)$ slowdown for the queries. They then showed how we can use an $\mathcal{O}\big(n \log^2 n\big)$-bit data structure to find the largest k such that k documents contain the pattern K times, in $\mathcal{O}(\mathsf{search}(m) + \log n \log\log n)$ time. Hon, Shah and Vitter [20] gave an $\mathcal{O}(n \log n)$-bit data structure that answers K-mine queries in time $\mathcal{O}(m)$ plus $\mathcal{O}(1)$ per reported document. They also showed how to improve the space bound to $2|\mathsf{CSA}| + o(n) + D \log(n/D)$ bits at the cost of increasing the time $\mathcal{O}\big(\mathsf{search}(m) + k \log^{3+\epsilon} n \cdot \mathsf{lookup}(n)\big)$, which we can improve similarly as before. Neither of these solutions applies to general colored range queries, however.

Since our CRTK data structure outputs elements in (approximately) non-increasing order by frequency in the range, it also solves (approximately) the natural generalization of DKM: i.e., the colored range K-mine (CRKM) problem, which asks us to report all the elements that occur at least K times in $S[i..j]$. If we query our data structure until the next element it would report occurs fewer than $(1 + \epsilon)K$ times, then we use $\mathcal{O}(\log \sigma \log(1/\epsilon))$ time per reported element, but we may miss some elements that occur between K and $(1 + \epsilon)K$ times. Alternatively, if we query our data structure until the next element it would report occurs fewer than $K/(1 + \epsilon)$ times, then we find all the elements that occur at least K times, but we can bound our time only in terms of the number of elements that occur at least $K/(1 + \epsilon)$ times.

4 Counting

Given a wavelet tree for the array C we described in Section 2, and positions i and j, it is not difficult to count the number of values less than i in $C[i..j]$ [26], which is the number of distinct elements in $S[i..j]$ [31]. The wavelet tree for C takes $\mathcal{O}(n \log n)$ bits and does this counting in time proportional to its height, $\mathcal{O}(\log n)$. This already matches the best known solution, due to Bozanis, Kitsios, Makris and Tsakalidis [6]. In the rest of this section we show how to reduce the space bound to $n \log \sigma + \mathcal{O}(n \log\log n)$ bits.

Theorem 4. *We can represent a sequence $S[1, n]$ over alphabet $[1, \sigma]$ in $n \log \sigma + \mathcal{O}(n \log\log n)$ bits of space so as to count the number of distinct elements in any interval $S[i..j]$ in time $\mathcal{O}(\log n)$.*

Proof. Our structure represents $C[1,n]$ using a wavelet tree. We have already explained how to attain the given time bound. The remaining problem is that the wavelet tree for C requires $n \log n(1 + o(1))$ bits. We reduce the space to $n \log \sigma + \mathcal{O}(n \log \log n)$ as follows. Note that each symbol $c \in [1,\sigma]$ that appears at positions $c_1 < c_2 < \ldots < c_{n_c}$, $S[c_1] = S[c_2] = \ldots = S[c_{n_c}] = c$, induces a *chain* in C of the form $C[c_1] = 0, C[c_2] = c_1, C[c_3] = c_2, \ldots, C[c_{n_c}] = c_{n_c-1}$. Now consider the middle point $n/2$ of C. For any c, let us call m_c the last value such that $c_{m_c} < n/2$. Then for any c and any $k \leq m_c$ it holds $C[c_k] < n/2$, and for any $k > m_c$ it holds $C[c_k] \geq n/2$. Thus $C[c_{m_c+1}] = c_{m_c} \geq n/2$ and $C[c_{m_c}] < n/2$, and $i = m_c$ is the only value satisfying this for c. Thus all the sequence values are $C[i] < n/2$ for $i < n/2$. For $i \geq n/2$ there are at most σ positions $i = m_c \geq n/2$ such that $C[m_c] < n/2$, and all the rest are $C[i] \geq n/2$. Thus there are at most σ positions in C where $C[i] < n/2$ and $C[i+1] \geq n/2$, and at most σ positions where $C[i] \geq n/2$ and $C[i+1] < n/2$. Since the root bitmap B_v satisfies $B_v[i] = 0$ iff $C[i] < n/2$, there are at most σ transitions from 0 to 1 in B_v, and at most σ transitions from 1 to 0. Both children of v may contain the σ subsequences and thus each may contain up to σ transitions again. Thus, there are at most $2^d\sigma$ 0/1 and 1/0 transitions among all the bitmaps at depth d of the wavelet tree.

For $d \geq \log(n/\sigma)$ this upper bound is useless, so we may assume that bitmaps at depths $\log(n/\sigma)$ to $\log n - 1$ are incompressible. These add up to $n(\log n - \log(n/\sigma)) = n \log \sigma$ bits, plus $o(n \log \sigma)$ to provide *rank* and *select* capabilities to those bitmaps. For smaller d, we introduce a compression scheme. Consider the concatenation B_d of all the bitmaps at depth d. Then B_d contains at most $2^d\sigma$ runs of 0s and $2^d\sigma$ runs of 1s. We represent B_d using two sparse bitmaps. A bitmap $R_d[1,n]$ will mark with a 1 the beginning of each run of 0s or 1s. Let $o_1, o_2, \ldots$ the lengths of the runs of 1s. A second bitmap $O_d[1, rank_1(B_d, n)]$ will have a 1 at positions 1, $1+o_1$, $1+o_1+o_2$, ... Then $\mathsf{rank}_1(B_d, i)$ can be computed as follows. First we compute $x = \mathsf{rank}_1(R_d, i)$. Because $C[i] < i$, the first run of B_d is a 0-run, thus if x is odd then i is within a 0-run and otherwise within a 1-run. If x is odd, then we must count the 1s in the first $(x-1)/2$ 1-runs of B_d, that is, $\mathsf{rank}_1(B_d, i) = \mathsf{select}_1(O_d, (x+1)/2) - 1$. If, instead, x is even, then we must count the 1s in the first $x/2 - 1$ 1-runs and add the 1s in the current run. This is $\mathsf{rank}_1(B, d_i) = \mathsf{select}_1(O_d, x/2) + i - select_1(R_d, x)$.

We represent R_d with Raman et al.'s technique [33]. If R_d has m 1s, then the representation takes $m \log \frac{n}{m} + \mathcal{O}(m) + o(n)$ bits. At level d we have $m \leq 2^d\sigma$, thus R_d requires at most $2^d\sigma \log \frac{n}{2^d\sigma} + \mathcal{O}(2^d\sigma) + o(n)$ bits (that $o(n)$ is $\mathcal{O}(n \log \log n / \log n)$). Added over all the compressible levels we have $\sum_{d=0}^{\log(n/\sigma)-1} 2^d\sigma \log \frac{n}{2^d\sigma} + \mathcal{O}(2^d\sigma) + o(n) = \mathcal{O}(n) + o(n \log(n/\sigma))$.

Analogously, the O_d bitmap takes $\mathcal{O}(n) + o(n \log(n/\sigma))$ bits. Added to the incompressible levels, we have $n \log \sigma + o(n \log n)$ bits of space, or more precisely, $n \log \sigma + \mathcal{O}(n \log \log n)$. The preprocessing time is the same as for a classical wavelet tree over alphabet $[0, n-1]$. □

On the other hand, the array C can also provide access to S as follows. Sample the tth occurrence of each color c, say at $S[i] = c$, in a bitmap $B[1,n]$, that is $B[i] = 1$, and store the samples at $W[rank_1(B,i)] = c$. Then, we can find out

any $S[j]$ without storing S by repeatedly asking whether $B[i] = 1$, $B[C[i]] = 1$, $B[C[C[i]]] = 1$, and so on until finding a sampled value, in time $\mathcal{O}(t \log n)$. The extra space is $n + o(n) + \mathcal{O}((n/t) \log \sigma)$, so we can set $t = \mathcal{O}(\log^{\epsilon} n)$ for any constant $\epsilon > 0$ to make it $n + o(n) \log \sigma$. Therefore, our representation *replaces* S, as it can compute any $S[i]$ in time $\mathcal{O}\big(\log^{1+\epsilon} n\big)$. Its space occupancy, $n \log \sigma + o(n) \log \sigma + \mathcal{O}(n \log \log n)$, makes the representation *succinct* (i.e., $|S|(1 + o(1))$ bits) whenever σ is more than polylogarithmic in n.

Theorem 4, applied over sequence $S = E$, lets us compute document frequencies for arbitrary patterns. Find the suffix array interval $\mathsf{CSA}[i..j]$ corresponding to the pattern, and then count the different values in $E[i..j]$. For this particular case, however, there is a better solution [35] using $2n + o(n)$ bits and constant time, yet it does not generalize to colored range counting. On the other hand, since our representation provides access to E in time $t_{\mathsf{acc}} = \mathcal{O}\big(\log^{1+\epsilon} n\big)$, it can be regarded within the framework of Section 2.

5 Further Applications to Information Retrieval

We have presented new and efficient solutions for three natural colored range queries: colored range listing, colored range top-k queries, and colored range counting. Our solutions for colored range listing lead to the fastest compressed data structures for that problem and for document listing; our (approximate) solution for colored range top-k queries is, as far as we know, the first efficient data structure for that problem; and our solution for colored range counting reduces the space bound from $\mathcal{O}(n \log n)$ bits to $n \log \sigma + \mathcal{O}(n \log \log n)$ bits while maintaining $\mathcal{O}(\log n)$ query time. Although our solutions for colored range top-k queries and colored range counting do not give improved bounds for the corresponding document retrieval problems, our more general data structures may find applications to other information retrieval scenarios beyond ranges induced by searching for exact patterns in suffix trees or arrays.

A simple example of natural queries not fitting in the restricted model are lexicographic range queries. Imagine we look for patterns lexicographically in the range ["1969", "2010"] in documents; the result is a suffix array range that does not correspond to any suffix tree node. In this case, existing techniques for document retrieval based on suffix tree properties (such as for computing top-k queries [20] and for computing document frequencies [35]) will not work. The general techniques we have introduced in this article do.

Yet another scenario that is not captured by the suffix tree model is inverted indices for natural language text (as opposed to the general texts addressed in this paper) [3]. Consider that we store the list of documents where each vocabulary word appears, consecutively according to the order of the words in the vocabulary. If queries are simple words, then all the document retrieval problems we have considered are easily solved by storing the documents of each list ordered by decreasing term frequency. Yet, imagine we wish to provide *also* the same functionality on stemmed searching, upon user request at query time. One solution is to group together the vocabulary words sharing the same stem

so that, while individual word queries can be handled as usual, stemmed queries are handled by considering the concatenation of the lists of the words sharing the same stem. Then we can regard the concatenation of all inverted lists as the array E and use the general techniques developed in this paper to answer various document queries on stems: Document listing and counting algorithms apply verbatim, while those involving frequencies pose further challenges as each entry in the inverted lists is weighted by the term frequency of the word in the document. Other query operations, from case folding to thesauri expansion, can also be reduced to a proper grouping of lists.

Finally, there are information retrieval scenarios completely different from the text search framework. For example, colored range queries seem a natural tool for query mining [2], where logs of queries posed to search engines are recorded over periods of time, and then analyzed to discover trends in user behavior. By considering that each different query is a color, we can find the most popular queries or the number of distinct queries within any given time period, among many other potential queries of interest, which could in turn become new challenging colored range queries.

References

1. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. NATO ISI Series, pp. 85–96. Springer, Heidelberg (1985)
2. Baeza-Yates, R.: Applications of web query mining. In: Losada, D.E., Fernández-Luna, J.M. (eds.) ECIR 2005. LNCS, vol. 3408, pp. 7–22. Springer, Heidelberg (2005)
3. Baeza-Yates, R., Ribeiro, B.: Modern Information Retrieval. AW (1999)
4. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select with applications. Technical Report 0911.4981, arXiv (2010)
5. Bille, P., Landau, G.M., Weimann, O.: Random access to grammar compressed strings. Technical Report 1001.1565, arXiv (2010)
6. Bozanis, P., Kitsios, N., Makris, C., Tsakalidis, A.K.: New upper bounds for generalized intersection searching problems. In: Fülöp, Z., Gecseg, F. (eds.) ICALP 1995. LNCS, vol. 944, pp. 464–474. Springer, Heidelberg (1995)
7. Brodal, G.S., Gfeller, B., Jørgensen, A.G., Sanders, P.: Towards optimal range medians. Theoretical Computer Science (to appear)
8. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
9. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: Proc. ESA (2010) (to appear)
10. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms (TALG), 3(2), article 20 (2007)
11. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theoretical Computer Science 371(1), 115–121 (2007)
12. Fischer, J.: Optimal succinctness for range minimum queries. In: Proc. LATIN, pp. 158–169 (2010)

13. 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)
14. Gabow, H.N., Bentely, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. STOC, pp. 135–143 (1984)
15. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
16. González, R., Navarro, G.: Compressed text indexes with fast locate. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 216–227. Springer, Heidelberg (2007)
17. Greve, M., Jørgensen, A.G., Larsen, K.D., Truelsen, J.: Cell probe lower bounds and approximations for range mode. In: Gavoille, C. (ed.) ICALP 2010, Part I. LNCS, vol. 6198, pp. 605–616. Springer, Heidelberg (2010)
18. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 636–645 (2003)
19. Grossi, R., Orlandi, A., Raman, R.: Optimal trade-offs for succinct string indexes. In: Proc. ICALP, pp. 678–689 (2010)
20. Hon, W.-K., Shah, R., Vitter, J.: Space-efficient framework for top-k string retrieval problems. In: Proc. FOCS, pp. 713–722 (2009)
21. Hon, W.-K., Shah, R., Wu, S.-B.: Efficient index for retrieving top-k most frequent documents. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 182–193. Springer, Heidelberg (2009)
22. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-K query processing techniques in relational database systems. ACM Computing Surveys 40(4) (2008)
23. Janardan, R., Lopez, M.A.: Generalized intersection searching problems. International Journal of Computational Geometry and Applications 3(1), 39–69 (1993)
24. Karpinski, M., Nekrich, Y.: Top-K color queries for document retrieval. Technical Report 1007.1361, arXiv (2010)
25. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12(1), 40–66 (2005)
26. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387(3), 332–347 (2007)
27. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
28. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48(3), 407–430 (2001)
29. 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)
30. Milidiú, R.L., Laber, E.S.: Bounding the inefficiency of length-restricted prefix codes. Algorithmica 31(4), 513–529 (2001)
31. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. SODA, pp. 657–666 (2002)
32. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys, 39(1), article 2 (2007)
33. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. SODA, pp. 233–242 (2002)
34. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003)

35. Sadakane, K.: Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5(1), 12–22 (2007)
36. Silvestri, F.: Sorting out the document identifier assignment problem. In: Amati, G., Carpineto, C., Romano, G. (eds.) ECiR 2007. LNCS, vol. 4425, pp. 101–112. Springer, Heidelberg (2007)
37. 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)
38. Weiner, P.: Linear pattern matching algorithm. In: Proc. IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)
39. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. WWW, pp. 401–410 (2009)

Range Queries over Untangled Chains*

Francisco Claude, J. Ian Munro, and Patrick K. Nicholson

David R. Cheriton School of Computer Science, University of Waterloo, Canada
{fclaude,imunro,p3nichol}@cs.uwaterloo.ca

Abstract. We present a practical implementation of the first adaptive data structure for orthogonal range queries in 2D [Arroyuelo et al., ISAAC 2009]. The structure is static, requires only linear space for its representation, and can even be made implicit. The running time for a query is $O(\lg k \lg n + \min(k, m) \lg n + m)$, where k is the number of non-crossing monotonic chains in which we can partition the set of points, and m is the size of the output. The space consumption of our implementation is $2n + o(n)$ words. The experimental results show that this structure is competitive with the state of the art. We also present an alternative construction algorithm for our structure, which in practice outperforms the original proposal by orders of magnitude.

1 Introduction

Imagine that you are driving and would like to find a nearby gas station using the Global Positioning System (GPS) in your car. You might instruct the GPS to draw markers on the map to indicate gas stations. Internally, the GPS would compute which gas stations are located within the rectangle shown on the screen. Scenarios like the one just described are special because the set of elements being queried do not change very often: you might update the database of your GPS once every few months. In this paper, we will discuss a recent data structure that is well suited for these kinds of scenarios.

Consider a set of two-dimensional points, $\mathcal{P}$, where $n = |\mathcal{P}|$. An *Orthogonal Range Query*, $\mathcal{R}(x_1, y_1, x_2, y_2)$, where $x_1 \leq x_2$ and $y_1 \leq y_2$, corresponds to a maximal subset $\mathcal{A} \subseteq \mathcal{P}$ such that $\forall (x, y) \in \mathcal{A}$, $x_1 \leq x \leq x_2$, $y_1 \leq y \leq y_2$. Many solutions have been proposed for answering orthogonal range queries, such as:

- R-trees [11]: a tree decomposition of $\mathcal{P}$, similar to B-trees, where a set of nodes represents a set of (possibly overlapping) rectangles. The structure takes linear space and its worst case searching time is $O(n)$. However, in practice they tend to perform better.
- Kd-trees [13]: recursively divide the d-dimensional space with hyperplanes. In our case, $d = 2$, the kd-tree uses lines for dividing the set of points. As with R-trees, kd-trees require linear space, but the query time improves to $O(\sqrt{n} + m)$, which is optimal for a structure that requires linear space [12].

* This work was supported in part by NSERC Canada, the Canada Research Chairs Programme, the Go-Bell and David R. Cheriton Scholarships Program, and an Ontario Graduate Scholarship.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 82–93, 2010.

– Range trees [6]: a generalization of a binary search tree for multiple dimensions. The query time of this structure is $O(\lg n + m)$ time (when combined with fractional cascading [7]), but as a trade-off it requires $O(n \lg n)$ space.

In recent work [3], a static structure for orthogonal range queries was presented. This structure achieves the same complexity as kd-trees in the worst case, but has the advantage that it is adaptive: for *easy* data sets it can improve upon the worst-case lower bound. We will call this structure *u-chains* as a shortened form of *untangled-chains*.

The structure of the paper is organized as follows. First we present a more detailed description of the u-chains structure. Then we describe the implementation decisions taken during the experimental setup. Finally we present the experimental results and conclude about the work, pointing to interesting open problems.

2 The U-Chains Structure

The u-chains structure is constructed by partitioning the set of points into monotonic ascending and descending chains, where each point belongs to one and only one chain.

The problem of splitting a two-dimensional point set into two classes (descending and ascending) so that we get the minimum number of monotonic chains is NP-hard [8]. The maximum number of chains we can obtain is bounded by $O(\sqrt{n})$. Fomin et al. presented an algorithm that achieves a constant factor approximation in $O(n^3)$ time, as well as a greedy algorithm which achieves a logarithmic factor approximation in $O(n\sqrt{n}\log n)$ time [10]. Yang et al. [15] improved the Yehuda-Fogel method [4], giving an algorithm that does not guarantee an approximation factor, but generates at most $\lfloor\sqrt{2n + 1/4} - 1/2\rfloor$ chains in $O(n\sqrt{n})$ time.

If the direction is fixed, Supowit proposed an algorithm that runs in optimal $O(n \lg n)$ time and obtains the optimal number of chains [14]. (See Algorithm 1.)

Algorithm 1 – Supowit($\mathbf{p_1} \ldots \mathbf{p_n}$)

```
1: S ← ∅
2: for i = 1...n, where x(p_i) < x(p_j) ∀i < j ≤ n do
3:    let S' = {A ∈ S, miny(A) ≥ y(p_i)}
4:    if S' ≠ ∅ then
5:       let A_0 = argmin_A{miny(A), A ∈ S'}
6:       append p_i to A_0
7:    else
8:       add p_i as a chain to S
9: return S
```

For the u-chains structure, the chains are required to be untangled, meaning that no two chains cross or *tangle*. By running an untangling algorithm on the

output of Supowit's algorithm, it is possible to obtain the minimum number of untangled chains [3]. The main property that allows the untangling algorithm to work is that all tangles generated by Supowit's algorithm are so called *v-tangles*, and when processed in the right order, they generate the corresponding untangled set.

Next we give some basic definitions to introduce the notation used for describing the algorithm for obtaining the untangled set of chains.

Definition 1 (H). *[3] Given an edge (p_i, p_j), define $H^+(p_i, p_j)$ to be the open half-plane bounded by the line through p_i and p_j and containing the point $(x(p_i)+1, y(p_i)+1)$, and $H^-(p_i, p_j)$ symmetrically.*

Using this definition we can define a v-tangle:

Definition 2 (v-tangle). *[3] Suppose we have two chains C_1 and C_2 with edges $(p_1, p_2), \ldots, (p_{\ell-1}, p_\ell) \in C_1$ and $(q_1, q_2) \in C_2$ such that $p_1 \in H^-(q_1, q_2)$, $p_\ell \in H^-(q_1, q_2)$, and $p_i \in H^+(q_1, q_2)$ for all $1 < i < \ell$. Also, (q_1, q_2) is called the upper part of the v-tangle, and $(p_1, p_2), \ldots, (p_{\ell-1}, p_\ell)$ the lower part. We define the operation of untangling a v-tangle as removing $p_2 \ldots p_{\ell-1}$ from C_1 and inserting it to C_2, between q_1 and q_2.*

Definition 3 (rv-tangle). *[2] Suppose we have two chains C_1 and C_2 with edges $(q_1, q_2) \in C_1$ and $(p_1, p_2), \ldots, (p_{\ell-1}, p_\ell) \in C_2$ such that $p_1 \in H^+(q_1, q_2)$, $p_\ell \in H^+(q_1, q_2)$, and $p_i \in H^-(q_1, q_2)$ for all $1 < i < \ell$. We call such a tangle a* reverse v-tangle*, or* rv-tangle*.*

The untangling procedure is based on a simple building block shown in Algorithm 2 [2]. The main idea is to run one Untangling-Pass and then extract the lower chain. We then recurse on the residual point set, extracting the lower chain until no points remain. This version differs from the preliminary version [3], which contained a difficulty revealed by this experimental work and corrected in a later version [2]. This procedure takes $O(kn \lg n + k^2 n)$ time, where k is the minimum number of chains.

Algorithm 2 – Untangling-Pass(P)

```
1: Run Supowit(P) to get chains C_1, ..., C_s where C_s is the uppermost chain
2: for i = s down to 1 do
3:     for j = i − 1 down to 1 do
4:         Find and untangle all v-tangles between C_i and C_j
5: Return C_1, ..., C_s
```

When the chains are untangled and sorted, searching becomes easier in the following sense:

– Searching in a chain is equivalent to doing a binary search, because we can exploit the fact that the chain is monotonic.

– When a chain C does not intersect the query, we can discard all the chains to one side of C, given that we add dummy points at the beginning and end of each chain to enforce an ordering. The dummy points add $O(\sqrt{n})$ words of extra space[1].

The search algorithm described requires $O(\lg n \lg k + \min(k, m) \lg n + m)$ time, which can be improved to $O(\lg n + k + m)$ time using fractional cascading. In the worst case, when $k = \Theta(\sqrt{n})$, we achieve the optimal time/space trade-off: linear space and $O(\sqrt{n} + m)$ query time [12]. For easier instances (i.e., when k is small) the structure takes advantage of this fact and performs better.

3 Implementation Details

In the following subsections we describe the practical decisions taken during the implementation phase. We go through the partitioning and untangling preprocessing steps.

3.1 Partitioning the Points

For our experiments, we implemented two algorithms for partitioning the set of points into monotonic descending and ascending chains. The first was the algorithm by Yang et al. [15]. The basic idea of the algorithm is to start with all of the points in descending chains called *layers*, and to repeatedly identify and extract ascending chains. It is important to note that the number of chains generated by the algorithm can be different if we start with ascending layers and extract descending chains. Although the algorithm can be implemented to run in $O(n\sqrt{n})$ time using optimizations described in [4], we instead opted to implement the simplified version of the algorithm which runs in $O(n\sqrt{n}\log n)$ time. Even with the extra $\log n$ factor, the processing time was not prohibitive for data sets of over a million points.

The second algorithm was the greedy method of Fomin et al. [10]. Since they describe the greedy algorithm in terms of a graph co-coloring problem, we will provide a brief description. We start with two empty sets, one for ascending chains, and one for descending chains. We take the set of points and compute the maximal length ascending chain, as well as the maximal length descending chain. The longer of the two chains is then added to the appropriate set, and we repeat this process on the residual point set until no points remain. Each pass of our implementation of the greedy algorithm takes $O(n \log n)$ time [4], and it generates a number of chains that is within a logarithmic factor of optimal [10]. Therefore the overall running time of our implementation is $O(n\sqrt{n}\log^2(n))$, but can be improved to $O(n\sqrt{n}\log n)$ using the techniques from [4].

Once we have determined which points belong to which set, we discard the extra information about the chains provided by the partitioning algorithms. In

[1] We need two dummy points for each chain, but we are also spending this amount of space for the pointers to the chains.

Algorithm 3 – Untangling(P)

```
1: Run Supowit(P) to get chains C_1, ..., C_k where C_k is the uppermost chain
2: done ← false
3: A_l ← 1
4: A_u ← k
5: while A_l < A_u AND ¬done do
6:    for i = A_u down to A_l do
7:       for j = i − 1 down to A_l do
8:          Find and untangle all v-tangles between C_i and C_j
9:    Apply transformation of lemma 2 to C_{A_l} ... C_{A_u}
10:   for i = A_l to A_u do
11:      for j = i + 1 to A_u do
12:         Find and untangle all v-tangles between C_i and C_j
13:   Apply transformation of lemma 2 to C_{A_l} ... C_{A_u}
14:   A_l ← A_l + 1
15:   A_u ← A_u − 1
16:   if C_1, ..., C_k are untangled then
17:      done ← true
18: Return C_1, ..., C_k
```

order for our subsequent untangling algorithm to work in a provably correct manner we need to run Supowit's algorithm on both the ascending and descending sets of points.

3.2 Untangling Chains

In this section we propose an alternative method for untangling the chains. As we will see in the experimental results, this alternative approach allows for a much faster construction in practice. However, we first describe how to improve the running time of the previous algorithm [2].

Lemma 1. *The algorithm by Arroyuelo et al. [2] for finding the minimum number of monotonic untangled chains can be adapted to run in $O(n \lg n + k^2 n)$ time.*

Proof. The improvement comes from running Supowit's algorithm k times in $O(n \lg n + kn \lg k)$ time. The main observation is that we only need to sort the points once, after which we can use markers to keep track of which points are still participating (i.e., do not belong to chains that are removed). We can then generate the set of the participating points, sorted, in $O(n)$ time. This happens only k times, so we spend $O(nk + n \lg n)$ time for the sorting phase in the k passes. The construction of the chains itself is easier, since we never have more than k chains in A, each point added requires at most $O(\lg k)$ time, and again we do this for every point at most k times, obtaining the remaining $O(nk \lg k)$.

Adding the cost of the k runs of Supowit's algorithm to the k runs of the Untangling-Pass, we obtain $O(n \lg n + k^2 n)$ time in the worst case. □

The idea of the alternative algorithm is inspired by the previous proposal [2], but is less dependent on the properties of Supowit's algorithm. The following

simple lemma is key for our practical variation. It allows us to transform a set of chains having only rv-tangles into a set of chains that has only v-tangles.

Lemma 2. *Given a set of ordered chains over $\mathcal{P}$, where all tangles among two chains are rv-tangles, we can map this problem to one with ordered chains and only v-tangles.*

Proof. Let $m_x = \max\{x \mid \exists y,\ (x, y) \in \mathcal{P}\}$ and $m_y = \max\{y \mid \exists x,\ (x, y) \in \mathcal{P}\}$. We compute the new set $\bar{\mathcal{P}} = \{(m_x - x, m_y - y) \mid (x, y) \in \mathcal{P}\}$, maintaining the chains. The order of the chains is reversed. □

This observation allows to do one untangling pass, transform the points, and then repeat the procedure over the remaining set of chains, without discarding information about the chains. This idea is formalized in Algorithm 3.

We currently do not have a proof that Algorithm 3 will yield untangled chains after it finishes. However, we have found no case in practice where Algorithm 3 fails to untangle the chains. In fact, for all of the data sets we have tried in practice, the $O(n)$ time check on line 16 of the algorithm succeeds after a small constant number of passes. We conjecture that in the worst case k passes are required. We end this section by noting that we can run Algorithm 3 for k passes, and, in the advent of a failure, detect that the chains have not been untangled. After such a failure, we can then run the previous algorithm [2]. Since both algorithms have the same running time by Lemma 1, running Algorithm 3 does not affect the overall running time asymptotically. However, as we will see in the next section, there is a clear separation in practice between the two algorithms.

4 Experimental Results

The machine used for the experiments has an AMD Athlon(tm) 64 X2 Dual Core Processor 5600+, core speed 2900MHz, L1 Cache size 256KB, and L2 Cache size 1024KB. It has 4GB of main memory of speed 800MHz. The operating system installed is GNU/Linux – Ubuntu 9.10, with kernel 2.6.31-17-generic running in 64bits mode and the compiler installed in the system is GNU/g++ version 4.4.

Our implementation for preprocessing the data set was compiled with flags `-O2 -Wall`, the code for searching was compiled with `-O2 -frounding-math -Wall`.

We used data sets provided by the Georgia Tech TSP Web page[2]. We included the following sets: Italy, China, LRB744710 and World.

4.1 Partitioning

We first tried the different partitioning methods: `aa` for all chains ascending, `ad` for all descending, `gr` for the greedy approach [10], `pa` for Yang et al.'s method [15] starting with all points in ascending layers, and finally `pd` for Yang et al.'s method starting with all points in descending layers.

[2] http://www.tsp.gatech.edu

Table 1. Results of different partition methods

Data Set	Nr. Points	Partitioning Method	Nr. Chains	Avg. Nr. of Pts/Chain	Std. Dev. of Pts/Chain
Italy	16862	aa	285	59.16	31.78
		ad	207	81.46	47.38
		pa	173	97.47	62.69
		pd	160	105.39	57.17
		gr	141	119.59	56.87
China	71009	aa	520	136.56	51.36
		ad	697	101.88	53.87
		pa	367	193.49	68.57
		pd	373	190.37	103.96
		gr	324	219.16	57.65
LRB	744710	ad	1203	619.04	227.76
		aa	1129	659.62	271.43
		pd	1074	693.40	262.35
		pa	1006	740.27	262.79
		gr	1065	699.26	228.38
World	1904711	aa	4167	457.09	266.50
		ad	3046	625.32	314.56
		pa	1865	1021.29	574.67
		pd	1843	1033.48	558.83
		gr	1608	1184.52	424.15

Table 1 shows the number of chains obtained with each method for the data sets considered in this work. We also include the average number of points per chain and the standard deviation for the expected number of points per chain.

The results show that considering the points in a fixed direction provides a competitive result with respect to the other partitioning methods. Yet, in most cases (except `LRB`), the `gr` approach obtains the minimum number of chains.

4.2 Construction

We implemented the two untangling algorithms. Table 2 shows the results for the small data set and gives a representative idea on the running time of this new version. For the `World` data set, it takes about one day to untangle the chains using the original method, while the new algorithm takes about twenty minutes.

Table 2. Results of the two untangling methods

Data Set	Partitioning Method	Time (sec) Original [2]	Time (sec) This paper
Italy	aa	293.75	1.53
	ad	254.34	1.10
	pa	125.34	0.48
	pd	220.72	0.68
	gr	116.78	0.41

4.3 Query Generation

For measuring the time, we generated 6 different types of queries. For each data set we find the bounding box $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. We refer to the regular grid partition of the bounding box as `tile`. For the other queries we generate a random point[3] $[x_i, y_i]$ for each query, and the rectangle is determined as follows:

[3] By random we mean uniformly at random unless stated otherwise.

- `rand`: we generate the second point within $[x_i, x_{max}] \times [y_i, y_{max}]$.
- `tiny`: we generate the second point at random within $[x_i, x_i + (x_{max} - x_i)/K] \times [y_i, y_i + (y_{max} - y_i)/K]$, where $K = 100$.
- `med`: same as `tiny` but using $K = 10$.
- `tall`: we generate the second point at random within $[x_i, x_i + (x_{max} - x_i)/25] \times [y_i, y_{max}]$.
- `wide`: we generate the second point at random within $[x_i, x_{max}] \times [y_i, y_i + (y_{max} - y_i)/25]$.

Figure 1 shows roughly how the six types of queries look.

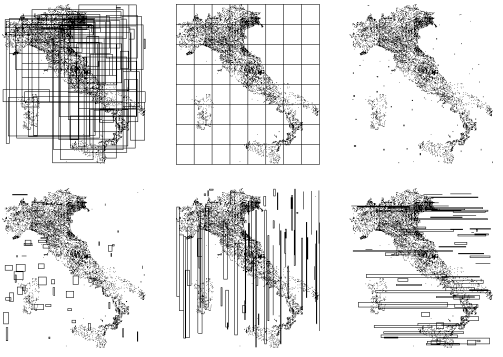


Fig. 1. Example of queries generated for `Italy`. From left to right, starting from the top row: `rand`, `tile`, `tiny`, `med`, `tall` and `wide`.

4.4 Range Searching with Different Partitioning Methods

We next measured the searching time considering the two variants: `binary` for binary searching every chain and `adaptive` for the version that binary searches a candidate chain to start, and uses the fact that the chains are untangled. Figures 2 and 3 show the results for the sets `LRB` and `World`. As we can see in Figure 2, for the `binary` variant, the running time depends on the number of chains obtained by the partitioning method, as expected. For the case of `adaptive`, the running time of `aa` and `ad` is sometimes better, since they do a binary search over all chains at once. In theory, `binary` should be better if $ad > c$, where a and d are the number of ascending and descending chains generated by the partitioning method, respectively, and c is the number of chains generated if we only consider one direction. However, in this argument, we are ignoring the number of points in each chain, which affects the final result.

In general `gr` behaves reasonably well compared to the other partitioning methods for `binary` and `adaptive`. For this reason, we use this partitioning method for the remaining experiments.

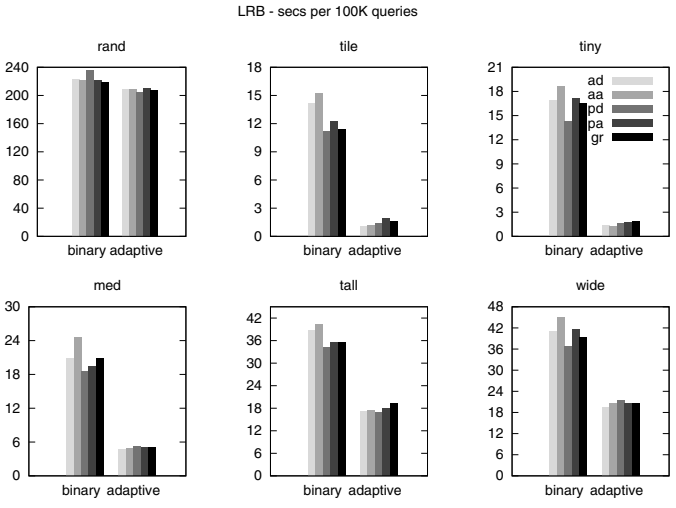


Fig. 2. Detailed time for U-Chains on `LRB`

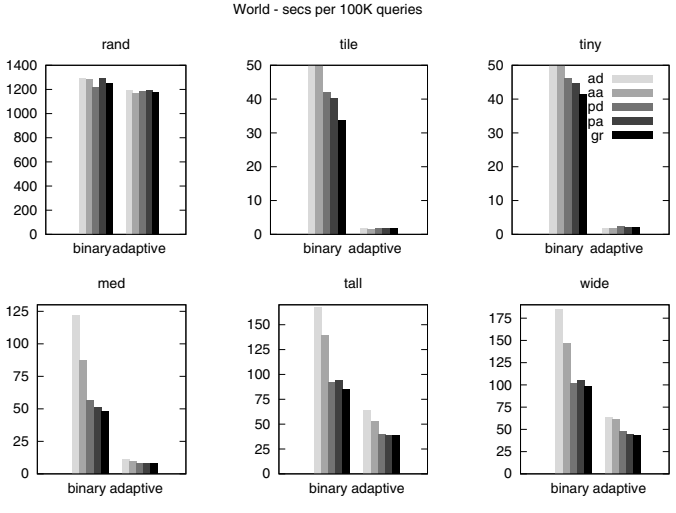


Fig. 3. Detailed time for U-Chains on `World`

4.5 Comparison to Previous Work

We tested our implementations against two structures implemented in the well known CGAL library [1] (version 3.4, default Ubuntu 9.10, multiverse packaging): Kd-tree and Range-tree. The code was compiled with flags `-O2 -frounding-math -Wall`.

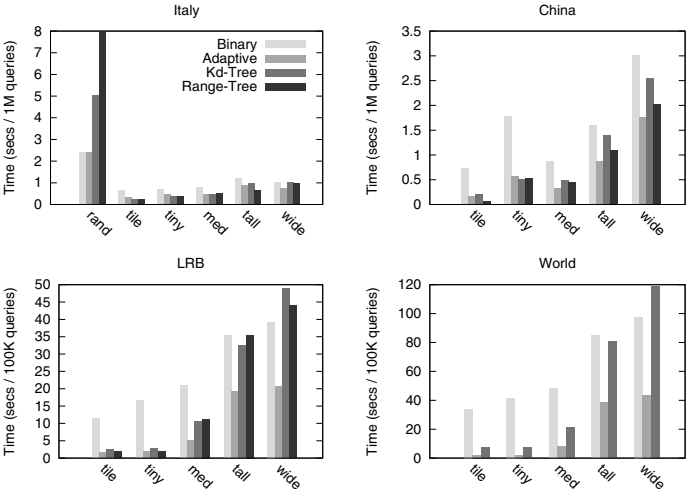


Fig. 4. Time for range queries in the data sets

Table 3. Results for `LRB`

Query Type	Partitioning Method	Time (sec) Binary	Time (sec) Adaptive	Time (sec) Kd-Tree	Time (sec) Range-tree
rand	aa	223.62	**208.90**	757.38	1144.92
	ad	221.03	**208.24**		
	gr	235.65	**205.06**		
	pa	222.15	**210.22**		
	pd	219.46	**207.13**		
tile	aa	14.21	**1.12**	2.60	1.87
	ad	15.26	1.17		
	gr	11.16	1.44		
	pa	12.21	1.90		
	pd	11.39	1.58		
tiny	aa	16.85	**1.36**	2.72	1.88
	ad	18.69	**1.32**		
	gr	14.27	1.62		
	pa	17.10	1.71		
	pd	16.48	1.92		
med	aa	20.82	**4.70**	10.43	11.23
	ad	24.46	**4.82**		
	gr	18.57	5.25		
	pa	19.42	5.11		
	pd	20.82	5.07		
tall	aa	38.71	**17.17**	32.35	35.49
	ad	40.32	**17.46**		
	gr	34.20	**16.87**		
	pa	35.60	17.96		
	pd	35.42	19.19		
wide	aa	41.00	**19.39**	48.99	43.85
	ad	44.86	20.42		
	gr	36.72	21.53		
	pa	41.65	20.53		
	pd	39.21	20.57		

Figure 4 shows the results obtained for the four data sets. We do not include Range-trees for `World` since the process required more than $7GB$ of RAM. For the larger data sets, `adaptive` dominates for all types of queries. In the two smaller data sets `adaptive` is competitive with the two other approaches considered, yet it does not present a clear time advantage with respect to Range-trees. The main advantage is the space consumption, when range-trees take more than $7GB$ of

RAM for `World`, adaptive requires only about $30MB$. The u-chains structure offers this interesting trade-off, but does not allow updates.

We include the table of values for `LRB`, we highlight every value that is within 5% of the minimum obtained.

5 Conclusions

In this paper we presented a practical implementation of the u-chains data structure. The experimental results show that this structure is efficient and suitable for two-dimensional orthogonal range queries over static data sets. In particular, for our two largest data sets, the adaptive search method on the u-chains structure significantly outperformed both range-trees and kd-trees. Although both of these structures are dynamic, we note that there are likely further optimizations that can be applied to our search methods to improve performance. A secondary advantage of the u-chains structure is that it only occupies $2n+O(\sqrt{n})$ words. Unlike range-trees and kd-trees, which require many pointers per node, each chain in the u-chains structure only stores its ordered set of points and its length. This small footprint certainly does not hurt performance, as it is likely to have better cache locality than the other data structures.

We included a new construction method (untangling algorithm) that can handle much larger data sets than the original proposal. The main advantage of the new algorithm is that it often terminates in much less than k passes, and, in the cases studied here, k is usually close to $\sqrt{n}$.

Our results leave many interesting open problems related to the u-chains:

- It would be an interesting challenge to make this structure dynamic, allowing for insertions and deletions.
- The performance of the structure could be improved by implementing fractional cascading, or any method that re-uses computational effort when jumping from chain to chain. It would be interesting to see how much fractional cascading would improve the search time.
- We could implement different methods to search the chains, like interpolation search or galloping search [5]. The impact of those variations will depend on the data sets considered, and a study of their impact would be of interest.
- Finally, we could look at ways to improve cache locality when searching the chains. For instance, considering the van Emde Boas layout for representing each chain [9].

References

1. CGAL, Computational Geometry Algorithms Library, `http://www.cgal.org`
2. Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Munro, J.I., Nicholson, P.K., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search, `http://www.recoded.cl/docs/untangling.pdf`

3. Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Munro, J.I., Nicholson, P.K., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search. In: 20th International Symposium on Algorithms and Computation (ISAAC), pp. 203–212 (2009)
4. Bar Yehuda, R., Fogel, S.: Partitioning a sequence into few monotone subsequences. Acta Informatica 35(5), 421–440 (1998)
5. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. J. Exp. Algorithmics 14 ,3.7–3.24 (2009)
6. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM 18(9), 509–517 (1975)
7. Chazelle, B., Guibas, L.J.: Fractional Cascading: I. A Data Structuring Technique. Algorithmica 1(2), 133–162 (1986)
8. Di Stefano, G., Krause, S., Lübbecke, M.E., Zimmermann, U.T.: On minimum k-modal partitions of permutations. Journal of Discrete Algorithms 6(3), 381–392 (2008)
9. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: 16th Annual Symposium on Foundations of Computer Science (FOCS), pp. 75–84 (1975)
10. Fomin, F.V., Kratsch, D., Novelli, J.C.: Approximating minimum cocolorings. Information Processing Letters 84(5), 285–290 (2002)
11. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: 1984 ACM SIGMOD international conference on Management of data (SIGMOD), pp. 47–57. ACM, New York (1984)
12. Kanth, K.V.R., Singh, A.K.: Optimal dynamic range searching in non-replicating index structures. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 257–276. Springer, Heidelberg (1998)
13. Lueker, G.S.: A data structure for orthogonal range queries. In: 19th Annual Symposium on Foundations of Computer Science (SFCS), pp. 28–34. IEEE Computer Society, Washington (1978)
14. Supowit, K.J.: Decomposing a set of points into chains, with applications to permutation and circle graphs. Information Processing Letters 21(5), 249–252 (1985)
15. Yang, B., Chen, J., Lu, E., Zheng, S.: Design and Performance Evaluation of Sequence Partition Algorithms. Journal of Computer Science and Technology 23(5), 711–718 (2008)

Multiplication Algorithms for Monge Matrices

Luís M.S. Russo[1,2]

[1] CITI / Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Quinta da Torre, 2829-516 Caparica, Portugal
lsr@di.fct.unl.pt
[2] INESC-ID, Knowledge Discovery and Bioinformatics Group, R. Alves Redol, 9, 1000-029 Lisbon, Portugal

Abstract. In this paper we study algorithms for the max-plus product of Monge matrices. These algorithms use the underlying regularities of the matrices to be faster than the general multiplication algorithm, hence saving time. A non-naive solution is to iterate the SMAWK algorithm. For specific classes there are more efficient algorithms. We present a new multiplication algorithm (MMT), that is efficient for general Monge matrices and also for specific classes. The theoretical and empirical analysis shows that MMT operates in near optimal space and time. Hence we give further insight into an open problem proposed by Landau. The resulting algorithms are relevant for bio-informatics, namely because Monge matrices occur in string alignment problems.

1 Introduction and Related Work

In this paper we study algorithms to multiply Monge matrices, more precisely the max-plus product of anti-Monge matrices, although we still refer to them as Monge. These matrices have a long history of algorithmic applications [1]. In Particular they occur in string processing problems, as DIST tables [2,3] or as Highest-Scoring Matrices (HSMs) [4]. Their applications to string processing problems, include: Cyclic LCS, Longest Repeated subsequence, Fully-Incremental LCS [5,6,4], etc.

Alves et al. [7] proposed an online algorithm to compute an implicit representation of HSMs in $O(nm)$ time and $O(m+n)$ working space, where m and n are the sizes of the strings being processed. Subsequently Tiskin observed that the core of these matrices has $O(n)$ size [4]. The core provides an alternative way to represent these matrices.

Given this representation the natural question is: can we multiply HSMs in linear time? proposed as an open problem by Landau[1] [8]. Tiskin made significant contributions by proposing $O(n^{1.5})$ and $O(n \log n)$ time algorithms [4,9,10]. The latter algorithm becomes $O(n \log^2 n / \log\log n)$ when considering the $O(\log n / \log\log n)$ access time, to the underlying representation structure [11].

This paper generalizes Landau's problem to core-sparse Monge matrices, i.e. $o(n^2)$ core size, and presents a core sensitive algorithm, MMT. For some of these problems Tiskin's algorithm can be applied, although affected by a variable factor ν. In the conditions of Landau's problem MMT runs in $O(n \log^3 n)$ time, including access costs. A

[1] The problem was formulated for DIST tables but it is essentially equivalent.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 94–105, 2010.

comprehensive comparison is given in section 4. The experimental results show, Section 4.1, that MMT runs in around $O(n \log^2 n)$ time, i.e., faster than the theoretical bound. Moreover MMT is a general multiplication algorithm that can be applied to general Monge matrices, namely related to alignment problems. Hence we obtain the first non-trivial solution for Fully-Incremental Alignment.

The structure of the paper is the following: Section 2 defines basic concepts; Section 3 describes the MMT algorithm; Section 4 gives a theoretical analysis of the several algorithms and experimental results of MMT; Section 5 concludes the paper.

2 Basic Concepts

This section presents basic concepts related to Monge matrices. In this paper $\log n$ is $\log_2 n$. Matrix row and column indexes start at 0. For matrix $\mathbf{A}$ consider the expression:

$$\Delta\mathbf{A}(i,i';j,j') = \mathbf{A}(i',j') - \mathbf{A}(i',j) - \mathbf{A}(i,j') + \mathbf{A}(i,j) \tag{1}$$

A matrix $\mathbf{A}$, of size $r \times c$, is **Monge**[2] iff $\Delta\mathbf{A}(i,i';j,j') \geq 0$, for any indexes i, i', j and j' such that $0 \leq i \leq i' < r$ and $0 \leq j \leq j' < c$. Figures 1 and 3 show examples of Monge matrices. Fig. 1 shows the non-zero $\Delta\mathbf{A}(i,i+1;j,j+1)$ values inside rectangles, and likewise for matrices $\mathbf{B}$ and $\mathbf{C}$. For example $\Delta\mathbf{A}(3,5;0,2) = (-3) - (-10) - (-20) + (-3) = 24$. The leftmost argument maximum of a given row is denoted as lax, *i.e.*, the smallest column index were the maximum of a row occurs. In Fig. 1 the leftmost maximums per row are in bold. For example $\operatorname{lax}\mathbf{A}[4,\_] = 1$, the respective maximum is $\mathbf{A}(4, \operatorname{lax}\mathbf{A}[4,\_]) = 6$. A matrix is **monotone** when $i \leq i'$ implies $\operatorname{lax}\mathbf{A}[i,\_] \leq \operatorname{lax}\mathbf{A}[i',\_]$. In other words the lax values increase as the row index increases. The Monge property implies that the matrices are also monotone, this can be observed in Fig. 1 by noticing that the bold values move to the right when descending by the rows.

The notion of core follows from an alternative characterization of Monge matrices. Let D denote a matrix, of size $(r-1) \times (c-1)$, with non-negative values, referred to as **density matrix**. The rows and columns of these matrices are indexed over half integers, *i.e.*, the first row and the first column are indexed by 0.5 instead of 0. Fig. 1 also contains examples of such matrices. The matrices consist of the values enclosed in rectangles, the omitted rectangles represent a 0. The respective **distribution** matrix d, of size $r \times c$, is defined, over the integers, as:

$$d(i,j) = \sum_{0<i'<i;\ 0<j'<j} D(i',j') \qquad \text{for all } 0 \leq i < r \text{ and } 0 \leq j < c \tag{2}$$

The next Lemma shows an alternative characterization of Monge matrices.

Lemma 1 ([1]). *A matrix $\mathbf{A}$, of size $r \times c$, is Monge iff there is an $r \times c$ distribution matrix d and two vectors $u \in \mathbb{R}^r$ and $t \in \mathbb{R}^c$ such that*

$$\mathbf{A}(i,j) = d(i,j) + u(i) + t(j) \qquad \textit{for all } 0 \leq i < r \textit{ and } 0 \leq j < c \tag{3}$$

[2] The usual Monge definition is $\Delta\mathbf{A}(i,i';j,j') \leq 0$, in which case $-\mathbf{A}$ verifies our condition.

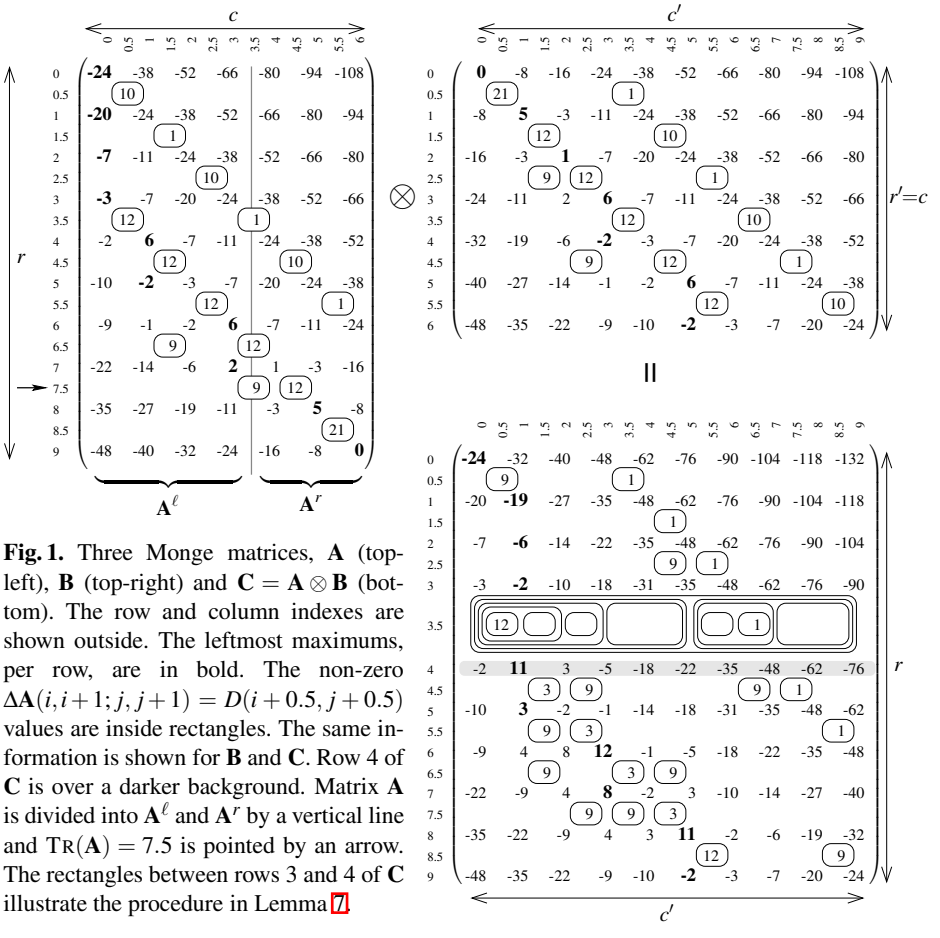


Fig. 1. Three Monge matrices, **A** (top-left), **B** (top-right) and $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ (bottom). The row and column indexes are shown outside. The leftmost maximums, per row, are in bold. The non-zero $\Delta\mathbf{A}(i, i+1; j, j+1) = D(i+0.5, j+0.5)$ values are inside rectangles. The same information is shown for **B** and **C**. Row 4 of **C** is over a darker background. Matrix **A** is divided into $\mathbf{A}^\ell$ and $\mathbf{A}^r$ by a vertical line and $\text{TR}(\mathbf{A}) = 7.5$ is pointed by an arrow. The rectangles between rows 3 and 4 of **C** illustrate the procedure in Lemma 7.

The proof of the lemma follows by defining $D(i+0.5, j+0.5) = \Delta\mathbf{A}(i, i+1; j, j+1)$ and observing that the $\Delta\mathbf{A}$ values are additive. The non-zero entries of D form the core of **A**, were $\delta(\mathbf{A})$ denotes its size. A matrix is considered sparse when $\delta(\mathbf{A}) = o(rc)$. Notice that $\Delta\mathbf{A}(i, i'; j, j') = \Delta d(i, i'; j, j')$, therefore computing the expression consists in summing the core entries inside $[i, i'] \times [j, j']$, for example $\Delta\mathbf{A}(3, 5; 0, 2) = 12 + 12$.

Definition 1. *For arbitrary matrices* **A**, **B**, *of sizes* $r \times c$ *and* $(c = r') \times c'$, *the max-plus product matrix,* $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$, *is* $\mathbf{C}(i, j) = \max_{0 \le k < c}\{\mathbf{A}(i, k) + \mathbf{B}(k, j)\}$.

Fig. 1 shows a sample max-plus matrix product. To perform the calculation we organize it into a sequence of *intermediate computation matrices*, **M**s, of size $c' \times r'$, such that $\mathbf{M}_i(j, k) = \mathbf{A}(i, k) + \mathbf{B}(k, j)$, for $0 \le i < r$, $0 \le j < c'$, $0 \le k < r' = c$. Fig. 3 shows $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$ and $\mathbf{M}_5$, ignore the tree-like structure on top. Each cell in the table contains an entry of the four **M**s. The bottom shows the calculation of the **M** values of cell $(4, 2)$. These matrices are used to compute the values of **C**, since $\mathbf{C}(i, j) = \mathbf{M}_i(j, \text{lax}\,\mathbf{M}_i[j, \_])$. For example row 4 of **C** can be obtained from $\mathbf{M}_4$. Observe that the bottom-left values of

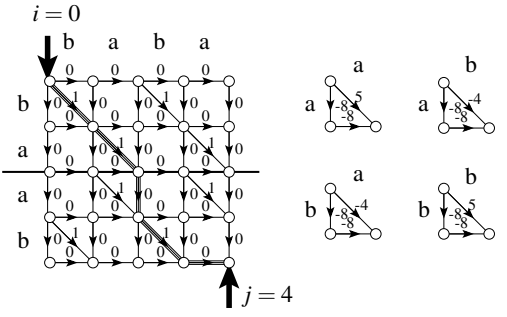


Fig. 2. (Left) Alignment DAGs of $S = ba$ and $T = baba$ and of $S' = ab$ with T. The two DAGs are united, horizontally, into the DAG of $S.S'$ and T. The horizontal line indicates the union. A highest-scoring path is represented by outlined arrows. This path corresponds $\mathrm{LCS}(S.S', T) = bab$, of size 3. (Right) Sample weights for Weighted Longest Common Subsequences.

each cell that are over a darker background correspond to row 4 of $\mathbf{C}$. The $\mathbf{M}_i$ matrices are Monge, because $\mathbf{B}$ is Monge, therefore there is a non-naive way to compute $\mathbf{C}$.

Lemma 2. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be Monge matrices, of sizes* $r \times c$ *and* $(c = r') \times c'$*, then the values of* $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ *can be obtained in* $O(rr' \max\{1, \log(c'/r')\})$ *time.*

Proof. Since the $\mathbf{M}_i$ matrices are Monge the SMAWK [12] algorithm obtains all the row maximums, of each $\mathbf{M}_i$, in $O(r' \max\{1, \log(c'/r')\})$ time. □

This paper does not explain the SMAWK algorithm, the interested reader should consult Aggarwall et al. [12]. A simple recursive algorithm finds the maximum of the middle row and divides $\mathbf{M}$ into two smaller sub-problems. For $\mathbf{M}$ of size $c' \times r'$ this process takes only $O(r' \log c')$ time. The MMT algorithm uses regularities of the $\mathbf{M}_i$ matrices and a variation of this algorithm.

2.1 Highest Score Matrices

String alignment problems are a source of Monge matrices. We denote by S, S' and T **strings** of size m, m' and n respectively; by Σ the **alphabet** of size σ; by $S[i]$ the symbol at position i, assuming that positions start at 0; by $S.S'$ **concatenation**; by $S = S[..i-1].S[i..j].S[j+1..]$ respectively a **prefix**, a **substring** and a **suffix**; note that $S[i..j]$, with $j < i$, denotes the empty string; A **subsequence** of S is obtained by deleting zero or more letters; a **Longest Common Subsequence** $\mathrm{LCS}(S,T)$ is a largest subsequence that can be obtained from both strings S and T. Consider the following example $S = ba$, $S' = ab$ and $T = baba$, where $m = m' = 2$ and $n = 4$. In this example $\mathrm{LCS}(S,T) = ba$, $\mathrm{LCS}(S',T) = ab$ and $\mathrm{LCS}(S.S',T) = bab$.

$\mathrm{LCS}(S,T)$ can be computed as a highest-scoring path in a DAG. The DAG is a grid of horizontal and vertical edges, with score 0, it contains diagonal edges, with score 1, for every pair of matching characters between the strings. See Fig. 2 for an example of such a DAG, notice that there is a diagonal edge on the top-left corner because S and T both start by b. A path corresponding to an LCS between $S.S'$ and T is highlighted. Depending on the starting and ending nodes of the path we can determine several LCS values, between S and a substring of T. The highest-score matrix (HSM) $\mathbf{H}_{S,T}$ stores these LCS values[3], *i.e.*, $\mathbf{H}_{S,T}(i,j) = \mathrm{LCS}(S, T[i..j-1])$. This matrix was denominated

[3] General $\mathbf{H}_{S,T}(i,j)$ values correspond to highest scoring paths on an infinite EADAG [9].

as $\textsc{Dist}(S,T)$ by Apostolico et al. [2], for edit distance problems. In this paper we follow the theory by Tiskin [9,10] but the results are essentially equivalent.

In this context the max-plus product, Def. 1, represents the fact that a highest-score path of $\mathbf{H}_{S,S',T}$ can be decomposed into a path of $\mathbf{H}_{S,T}$ followed by a path of $\mathbf{H}_{S',T}$. HSMs are Monge [2], core-sparse and unit, *i.e.*, $\delta(\mathbf{H}_{S,T}) = \min\{m,n\} = o(n^2)$ and the non-empty core entries are always 1. Therefore they can be multiplied efficiently.

Theorem 1 ([9]). *Given unit-Monge matrices* $\mathbf{A}$, $\mathbf{B}$, *both of size* $n \times n$, *the core entries of* $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ *can be obtained in* $O(n \log n)$ *time and* $O(n)$ *space.*

This result has a significant impact on string problems [13], namely Cyclic LCS, Longest Repeated Subsequence, Fully-Incremental LCS [5,6], etc. The latter problem consists in maintaining a data structure that returns the size of $\mathrm{LCS}(S,T)$ and supports updates to $\mathrm{LCS}(c.S,T)$, $\mathrm{LCS}(S.c,T)$, $\mathrm{LCS}(S,c.T)$ and $\mathrm{LCS}(S,T.c)$, where c is a new character. Notice that the first update is against the usual dynamic programing direction and therefore a naive approach requires $O(mn)$ time. Using Theorem 1 with $\mathbf{A} = \mathbf{H}_{c,T}$ the update to $\mathrm{LCS}(c.S,T)$ runs in $O(n \log n)$ time, which is competitive against state of the art solutions [6] of $O(n)$ time.

The concept of LCS can be extended to generic alignments, where the weight of the edges is an integer that depends on the characters involved. The resulting HSMs are generic Monge matrices, not necessarily unit. A simple case, Weighted Longest Common Subsequences (WLCS), occurs when the weight of the edges depends only on the type of edge, see the right part of Fig. 2.

3 Core Sensitive Multiplication

This section describes the Multiple Maxima Trees (MMT) algorithm for the max-plus product of Monge matrices. The algorithm receives the cores of $\mathbf{A}$ and $\mathbf{B}$ and outputs the core of $\mathbf{C}$. We describe the algorithm in two phases. First we propose a structure that *represents* the individual values of $\mathbf{C}$, *i.e.*, that we can use to access an individual value of $\mathbf{C}$; Second we explain how to use those values to determine the core of $\mathbf{C}$.

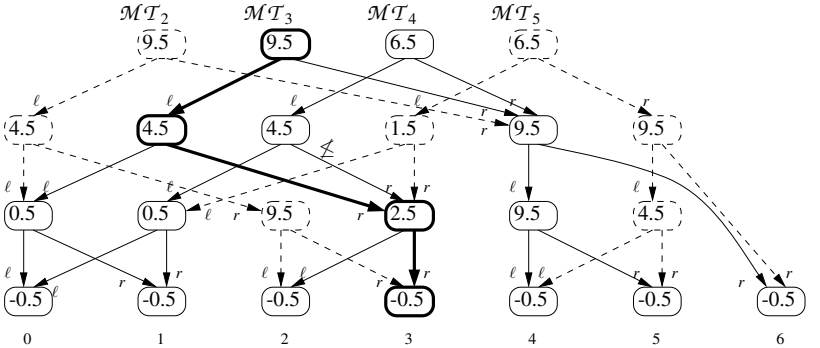
3.1 Representing $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$

We divide matrix $\mathbf{A}$, in half, into $\mathbf{A}^\ell$ and $\mathbf{A}^r$. The left sub-matrix $\mathbf{A}^\ell$ contains columns 0 to $\lceil c/2 \rceil - 1$. The right sub-matrix $\mathbf{A}^r$ contains columns $\lceil c/2 \rceil$ to $c-1$. For example in Fig. 1 $\mathbf{A}^\ell$ contains columns $0,1,2,3$ and $\mathbf{A}^r$ contains columns $4,5,6$. Notice that there will be an index for which the leftmost maximum changes from $\mathbf{A}^\ell$ to $\mathbf{A}^r$.

Definition 2. *The* ***transition point*** $\textsc{Tr}(\mathbf{A})$, *of a monotone matrix* $\mathbf{A}$, *is a half integer such that* $\operatorname{lax}\mathbf{A}[i,\_]$ *is* $\operatorname{lax}\mathbf{A}^\ell[i,\_]$ *for* $i < \textsc{Tr}(\mathbf{A})$ *and* $\lceil c/2 \rceil + \operatorname{lax}\mathbf{A}^r[i,\_]$ *for* $i > \textsc{Tr}(\mathbf{A})$.

Fig. 1 indicates that $\textsc{Tr}(\mathbf{A}) = 7.5$ with an arrow. Fig. 3 shows that $\textsc{Tr}(\mathbf{M}_4) = 6.5$.

Definition 3. *The* ***maxima tree*** $\mathcal{MT}_{\mathbf{A}}$, *of a Monge matrix* $\mathbf{A}$, *is a balanced binary tree. If* $\mathbf{A}$ *is empty* $\mathcal{MT}_{\mathbf{A}}$ *is also empty, otherwise the sub-trees that start at the children of the* ROOT *are* $\mathcal{MT}_{\mathbf{A}^\ell}$ *and* $\mathcal{MT}_{\mathbf{A}^r}$. *The* ROOT *stores* $\textsc{Tr}(\mathbf{A})$, *the remaining nodes store the corresponding transition points.*



	0	1	2	3	4	5	6
0	**-7, -3,** **-2, -10,**	-19, -15, -2, -10,	-40, -36, -23, -19,	-62, -48, -35, -31,	-84, -70, -56, -52,	-106, -92, -78, -64,	-128, -114, -100, -86,
1	-15, -11, -10, -18,	**-6, -2,** **11, 3,**	-27, -23, -10, -6,	-49, -35, -22, -18,	-71, -57, -43, -39,	-93, -79, -65, -51,	-115, -101, -87, -73,
2	-23, -19, -18, -26,	**-14, -10,** **3,** -5,	-23, -19, -6, **-2,**	-36, -22, -9, -5,	-58, -44, -30, -26,	-80, -66, -52, -38,	-102, -88, -74, -60,
3	-31, -27, -26, -34,	**-22, -18,** **-5,** -13,	-31, -27, -14, -10,	-32, -18, -5, **-1,**	-54, -40, -26, -22,	-67, -53, -39, -25,	-89, -75, -61, -47,
4	-45, -41, -40, -48,	**-35, -31,** **-18,** -26,	-44, -40, -27, -23,	-45, -31, -18, **-14,**	-55, -41, -27, -23,	-68, -54, -40, -26,	-90, -76, -62, -48,
5	-59, -55, -54, -62,	-49, -45, -32, -40,	**-48,** -44, -31, -27,	-49, **-35,** **-22, -18,**	-59, -45, -31, -27,	-60, -46, -32, -18,	-82, -68, -54, -40,
6	-73, -69, -68, -76,	-63, -59, -46, -54,	**-62,** -58, -45, -41,	-62, **-48,** **-35, -31,**	-72, -58, -44, -40,	-73, -59, -45, -31,	-83, -69, -55, -41,
7	-87, -83, -82, -90,	-77, -73, -60, -68,	**-76,** -72, -59, -55,	-76, **-62,** -49, -45,	-76, -62, **-48,** -44,	-77, -63, -49, **-35,**	-87, -73, -59, -45,
8	-101, -97, -96, -104,	-91, -87, -74, -82,	**-90,** -86, -73, -69,	-90, **-76,** -63, -59,	-90, -76, **-62,** -58,	-90, -76, -62, **-48,**	-100, -86, -72, -58,
9	-115, -111, -110, -118,	-105, -101, -88, -96,	**-104,** -100, -87, -83,	-104, **-90,** -77, -73,	-104, -90, **-76,** -72,	-104, -90, -76, **-62,**	-104, -90, -76, -62,

$$\mathbf{M}_2(4,2)=\mathbf{A}(2,2)+\mathbf{B}(2,4)=-24-20=-44; \quad \mathbf{M}_3(4,2)=\mathbf{A}(3,2)+\mathbf{B}(2,4)=-20-20=-40;$$
$$\mathbf{M}_4(4,2)=\mathbf{A}(4,2)+\mathbf{B}(2,4)=\ -7-20=-27; \quad \mathbf{M}_5(4,2)=\mathbf{A}(5,2)+\mathbf{B}(2,4)=\ -3-20=-23;$$

Fig. 3. (Bottom) Calculation of the $\mathbf{M}$ values of cell $(4,2)$. (Middle) Matrices $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$, $\mathbf{M}_5$ and the respective maxima trees. Each cell in the table shows a value from the four $\mathbf{M}$ matrices, $\mathbf{M}_2$ (top-left), $\mathbf{M}_3$ (top-right), $\mathbf{M}_4$ (bottom-left) and $\mathbf{M}_5$ (bottom-right). The bold values highlight the leftmost maximum, per row. The maximums of $\mathbf{M}_4$ are over a darker background, these values correspond to row 4 of $\mathbf{C}$ in Fig. 1. (Top) the maxima trees of these matrices. The nodes and branches that are exclusive to trees $\mathcal{MT}_2$ or $\mathcal{MT}_5$ are dashed. Each node contains the transition point of the respective sub-matrix. The representation is compact, similar sub-trees are not repeated. The edges of the trees are labeled ℓ and r depending on whether the corresponding sub-tree is left or right. The computation of $\operatorname{lax}\mathbf{M}_3[5,\_]=3$ is indicated by thicker lines.

Fig. 3 shows the maxima trees for $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$ and $\mathbf{M}_5$. At this point the reader should focus on an individual tree, $\mathcal{MT}_3$ for example[4]. It is a **fallacy** to assume that $\text{TR}(\mathbf{A}^\ell) \leq \text{TR}(\mathbf{A}) \leq \text{TR}(\mathbf{A}^r)$, Fig. 3 shows a counter example, indicated by $\not\leq$. The maxima tree can be used to compute lax values.

Lemma 3. *Let* $\mathbf{A}$ *be a Monge matrix, of size* $r \times c$*, its maxima tree* $\mathcal{MT}_{\mathbf{A}}$ *can be stored in* $O(c)$ *space and* $\operatorname{lax}\mathbf{A}[i,\_]$ *can be computed in* $O(\log c)$ *time.*

Proof. We compute $\operatorname{lax}\mathbf{A}[i,\_]$ recursively as $\operatorname{lax}\mathbf{A}^\ell[i,\_]$ if $i < \text{TR}(\mathbf{A})$ and as $\lceil c/2 \rceil + \operatorname{lax}\mathbf{A}^r[i,\_]$ otherwise, *i.e.*, move to the left or to the right child. The execution time is bounded by the height of the tree, that is $O(\log c)$. Since we store only one value per node, the space occupied by the tree depends on the number of nodes, *i.e.*, $O(c)$. □

Suppose we want to compute $\operatorname{lax}\mathbf{M}_3[5,\_]$. We start at the ROOT and since $5 < 9.5$ we move to the left, now since $5 > 4.5$ we move to the right, since $5 > 2.5$ we move to the right and reach the leaf of column $3 = \operatorname{lax}\mathbf{M}_3[5,\_]$. Notice that we can also compute the lax values of a sub-matrix corresponding to an internal node of $\mathcal{MT}_{\mathbf{A}}$.

Lemma 4. *The maxima tree of a Monge matrix* $\mathbf{A}$ *takes* $O(c \log r)$ *time to build.*

Proof. The tree can be built bottom-up. Computing $\text{TR}(\mathbf{A})$ for the ROOT can be done, in $O(\log r)$ steps, with a binary search. If $\mathbf{A}^\ell(i, \operatorname{lax}\mathbf{A}^\ell[i,\_]) \geq \mathbf{A}^r(i, \operatorname{lax}\mathbf{A}^r[i,\_])$ then $\text{TR}(\mathbf{A}) > i$, otherwise $\mathbf{A}^\ell(i, \operatorname{lax}\mathbf{A}^\ell[i,\_]) < \mathbf{A}^r(i, \operatorname{lax}\mathbf{A}^r[i,\_])$ and $\text{TR}(\mathbf{A}) < i$. For the remaining internal nodes the process is similar. This yields an overall $O(c(\log c)\log r)$ time. An amortized analysis shows that we are not paying $O(\log c)$ to obtain the lax values, as in Lemma 3. Most nodes are close to the leaves. Half of the nodes pay 1 operation for lax. One quarter of the nodes pay 2 operations, and so on. The overall time is $O(c \log r)$. □

Building all the $\mathcal{MT}_{\mathbf{M}_i}$ trees takes $O(rr' \log c')$ time and $O(rr')$ space. The resulting structure provides $O(\log r')$ access time to $\mathbf{C}(i,j) = \mathbf{M}_i(j, \operatorname{lax}\mathbf{M}_i[j,\_]))$. This is inefficient. The $\mathbf{M}_i$ matrices have regularities that considerably reduce their requirements.

Lemma 5. *Given Monge matrices* $\mathbf{A}$ *and* $\mathbf{B}$*, when* $\Delta\mathbf{A}(i,i';k,k') = 0$ *we have that* $\mathbf{M}_i(j,k) \leq \mathbf{M}_i(j,k')$ *iff* $\mathbf{M}_{i'}(j,k) \leq \mathbf{M}_{i'}(j,k')$.

Proof. Notice the prime in $\mathbf{M}_{i'}$. The following diagram proves the Lemma:

$$\mathbf{M}_i(j,k) - \mathbf{M}_i(j,k') = \mathbf{A}(i,k) - \mathbf{A}(i,k') + \mathbf{B}(k,j) - \mathbf{B}(k',j) \tag{4}$$

$$\| \tag{5}$$

$$\mathbf{M}_{i'}(j,k) - \mathbf{M}_{i'}(j,k') = \mathbf{A}(i',k) - \mathbf{A}(i',k') + \mathbf{B}(k,j) - \mathbf{B}(k',j) \tag{6}$$

Eqs 4 and 6 follow from the Def. of $\mathbf{M}_i$. Equation 5 follows from the hypothesis. □

Notice that the $\Delta\mathbf{A}(i,i';k,k') = 0$ hypothesis implies that $\Delta\mathbf{A}(i_1,i_2;k_1,k_2) = 0$ for any $i \leq i_1 \leq i_2 \leq i'$ and $k \leq k_1 \leq k_2 \leq k'$. This Lemma exposes the redundant information among the $\mathcal{MT}_{\mathbf{M}_i}$ trees. Namely if $\Delta\mathbf{A}(i,i';k,k') = 0$ and ϑ, ϑ' are nodes of $\mathcal{MT}_{\mathbf{M}_i}$ and $\mathcal{MT}_{\mathbf{M}_{i'}}$ whose corresponding rows are the $[k,k']$ interval then the sub-trees of ϑ

[4] Note that we simplified the notation.

and ϑ' are identical. Fig. 3 shows an example of this observation, matrices $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$ share the right sub-tree, since $\Delta\mathbf{A}(2,4;4,6) = 0$. Moreover $\mathbf{M}_5$ does not share the right sub-tree with $\mathbf{M}_4$, since $\Delta\mathbf{A}(4,5;4,6) = 10 \neq 0$.

Lemma 6. *Given Monge matrices* $\mathbf{A}$ *and* $\mathbf{B}$*, of sizes* $r \times c$ *and* $(c = r') \times c'$*, there is a representation of* $\mathbf{C}$ *with* $O(\log r')$ *access time, that needs* $O(r' + \delta(\mathbf{A}) \log r')$ *space and* $O(r + (r' + \delta(\mathbf{A})(\log r')^2) \log c')$ *time to be built.*

Proof. Use Lemma 4 to build the first tree, $\mathcal{MT}_1$, in $O(r' \log c')$ time. In general use Lemma 5 to build $\mathcal{MT}_{\mathbf{M}_{i+1}}$ from $\mathcal{MT}_{\mathbf{M}_i}$. If $\Delta\mathbf{A}(i, i+1; 0, c-1) = 0$ then $\mathcal{MT}_{\mathbf{M}_i}$ and $\mathcal{MT}_{\mathbf{M}_{i'}}$ are identical and the computation finishes. Otherwise we build a new ROOT for $\mathcal{MT}_{\mathbf{M}_{i'}}$ and proceed recursively to determine whether the left and right children of $\mathcal{MT}_{\mathbf{M}_{i'}}$ are new or the same as in $\mathcal{MT}_{\mathbf{M}_i}$. Each core value of $\mathbf{A}$ originates at most $\log r'$ new nodes. For each new node we recompute, bottom-up, the respective transition points, in $O((\log r') \log c')$ time each. The $O(r)$ term comes from the i cycle. □

3.2 Obtaining the Core

Using the representation of $\mathbf{C}$ we can obtain its core. This section explains how.

Lemma 7. *Given a Monge matrix* $\mathbf{C}$*, of size* $r \times c'$*, its core entries can be determined by inspecting* $O(r + \delta(\mathbf{C}) \log c')$ *entries of* $\mathbf{C}$*.*

Proof. For every i we compute $\Delta\mathbf{C}(i, i+1; 0, c'-1)$. If the result is 0 we conclude there is no core entry in $[i, i+1] \times [0, c'-1]$ and abandon the search. Otherwise we recursively consider $\Delta\mathbf{C}(i, i+1; 0, \lceil c'/2 \rceil)$ and $\Delta\mathbf{C}(i, i+1; \lceil c'/2 \rceil, c'-1)$. This procedure needs $O(\log c')$ for each core entry of $\mathbf{C}$ and $O(r)$ to consider every row of $\mathbf{C}$. □

Fig. 1 shows an illustration of the procedure described in the Lemma, between rows 3 and 4 of $\mathbf{C}$. We can now combine Lemmas 6 and 7 to obtain our main result.

Theorem 2. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be Monge matrices, of sizes* $r \times c$ *and* $(c = r') \times c'$*, the core of* $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ *can be computed in* $O(r \log r' + (r' + (\delta(\mathbf{C}) + \delta(\mathbf{A}) \log r') \log r') \log c')$ *time and* $O(\delta(\mathbf{C}) + r')$ *working space.*

Proof. Lemmas 6 and 7 yield an $O(\delta(\mathbf{C}) + r' + \delta(\mathbf{A}) \log r')$ working space solution. It is not necessary to store all the maxima trees $\mathcal{MT}_{\mathbf{M}_i}$. An iteration of the procedure in Lemma 7 inspects only $\mathbf{M}_i$ and $\mathbf{M}_{i+1}$. Hence it is enough to store only two trees in each iteration, which requires at most $O(r')$ space, see Lemma 3 and proof of Lemma 6. □

Using the equation $\mathbf{C} = (\mathbf{B}^T \otimes \mathbf{A}^T)^T$ the previous result gives an algorithm that runs in $O(c' \log c + (c + (\delta(\mathbf{C}) + \delta(\mathbf{B}) \log c) \log c) \log r)$ time and $O(\delta(\mathbf{C}) + c)$ space.

4 Analysis

This section presents a theoretical and empirical analysis of the several multiplication algorithms. Up to now the time to access an entry of $\mathbf{A}$ or $\mathbf{B}$ has been omitted. Since the working space can be $o(rc)$, it is not reasonable to assume $O(1)$ access time. For unit-Monge matrices this problem was solved [9] using a dominance counting structure [11],

Table 1. Comparison between max-plus multiplication algorithms, accounting for access time to $\mathbf{A}$ and $\mathbf{B}$, $O(1)$ for $\varepsilon = 2$, $O(\log n/\log\log n)$ for unit-Monge and $O(\log n)$ otherwise. For $\varepsilon \geq 1$ and $\varepsilon \neq 2$ the performance of MMT is unaltered even if access time is $O(1)$. The $\langle s(n), t(n)\rangle$ notation means $s(n)$ space and $t(n)$ time requirements.

$\delta = \Theta(n^\varepsilon)$	MMT, Theorem 2	SMAWK , Lemma 2	Theorem 1
$\varepsilon = 2$	$\langle O(n^2), O(n^2 \log n)\rangle$	$\langle O(n^2), O(n^2)\rangle$	—
$2 < \varepsilon \leq 1$	$\langle O(n^\varepsilon \log n), O(n^\varepsilon \log^3 n)\rangle$	$\langle O(n^\varepsilon \log n), O(n^2 \log n)\rangle$	—
unit-Monge, $\varepsilon = 1$	$\langle O(n), O(n \log^3 n)\rangle$	$\langle O(n), O(\frac{n^2 \log n}{\log\log n})\rangle$	$\langle O(n), O(\frac{n \log^2 n}{\log\log n})\rangle$
$1 < \varepsilon \leq 0$	$\langle O(n), O(n \log^2 n)\rangle$	$\langle O(n), O(n^2 \log n)\rangle$	—

which works in $O((\log \delta(\mathbf{A}))/\log\log \delta(\mathbf{A}))$ time and $O(\delta(\mathbf{A}))$ space. A similar result can be obtained with a wavelet tree [14]. Moreover by adding accumulated values by level it is possible to obtain the values of general Monge matrices, in $O(\log \delta(\mathbf{A}))$ time and $O(\delta(\mathbf{A}) \log \delta(\mathbf{A}))$ space.

To simplify the analysis let us assume that $O(r) = O(c) = O(r') = O(c') = O(n)$ and that $O(\delta(\mathbf{A})) = O(\delta(\mathbf{B})) = O(\delta(\mathbf{C})) = O(\delta)$. The algorithm of Theorem 1 needs to access entries at each step. Hence the access value becomes a factor of the overall time. The MMT algorithm always computes a lax value before accessing an entry, *i.e.*, the accesses of the algorithm are always of the form $\mathbf{C}(i,j) = \mathbf{M}_i(j, \operatorname{lax} \mathbf{M}_i[j,\_])$. Except in the amortized analysis of Lemma 4 the lax operation is $\Omega(\log n)$, see Lemma 3. Therefore the access time does not become a time factor in MMT. This claim is confirmed experimentally, in Section 4.1.

Table 1 shows a comparison of the different algorithms according to core size, $\delta = \Theta(n^\varepsilon)$. The analysis considers the best space and time requirements that includes the access time to $\mathbf{A}$ and $\mathbf{B}$. For $\varepsilon = 2$ the access time is $O(1)$, in this case Lemmas 6 and 7 have amortized performance, hence the $o(n^2 \log^3 n)$ time. Lemma 6 becomes an iterated Lemma 4. Lemma 7 loses a $O(\log n)$ factor. For $\varepsilon < 1$ the access time is $O(\log n)$, the dominating time is that of building $\mathcal{MT}_1$, Lemma 4. Because of the amortized analysis we need to count the access time, hence the $\Omega(n \log n)$ time.

Table 1 shows that the MMT algorithm is not always the most efficient, in particular SMAWK is faster by an $O(\log n)$ factor for $\delta = \Theta(n^2)$. This factor quickly disappears for $\varepsilon < 2$, not only because SMAWK is not sensitive to the core size, but also because access times are no longer $O(1)$. For the remaining core-sizes MMT is faster than SMAWK . For the particular case of unit-Monge matrices, with $\delta = \Theta(n)$, the algorithm of Theorem 1 is faster than MMT by a $O(\log n \log\log n)$ factor. However the experimental results show that in practice MMT is faster than what is predicted by theory.

Using MMT we obtain the first non-trivial solution for the Fully-Incremental Alignment problem, which consists in updating a generic alignment score from $\textsc{Align}(S,T)$ to $\textsc{Align}(c.S,T)$, $\textsc{Align}(S,c.T)$, $\textsc{Align}(S.c,T)$ or $\textsc{Align}(S,T.c)$, where $\textsc{Align}$ is the respective score. The resulting procedure takes $O((n+\delta)\log^3(n+\delta))$ time and $O(n+\delta\log\delta)$ space, where δ is the core size of the intervening matrices.

4.1 Experimental Results

We implemented MMT and tested it on an Intel Core2 Duo @1.33GHz, with 1.9 GiB of RAM running Xubuntu 9.10, with Linux Kernel 2.6.31. The code was compiled

with `gcc 4.4.1 -O9`. The prototype consists only of the multiplication and not of the representation of **A** and **B**, which are stored in memory and hence have $O(1)$ access time. To test the performance we multiplied HSM matrices, obtained from the LCS, as defined in Section 1. We also tested HSMs that resulted from generic alignments with the PAM weight matrix [15] and HSMs from Weighted Longest Common Subsequences (WLCS), using the weights in Fig. 2. The algorithm of Theorem 1 can be extended to support WLCSs, but the space and time are affected by a factor ν, in this scenario $\nu = 14$. On the other hand for PAM this technique does not apply, *i.e.*, the algorithm from Theorem 1 cannot be adapted for this problem. If it were possible it would have $\nu = 26$. The underlying strings S, S', T were random Bernoulli proteins, *i.e.*, $\sigma = 23$, that differed in a letter with 10% probability. The sizes were $m = m' = 4i$ and $n = 8i$ for i between 1 and 128. The results are shown in Fig. 4. To make the plots readable we show only 20% of the points. The x axis is indexed by a variable $N = \max\{n + \delta\}$. For precision we repeated each query during 10 seconds.

The plots on the left show the running time, in seconds, of the MMT algorithm and of an $O(\nu n \log(\nu n))$ time algorithm, denominated as Simulated. The ν factor is calculated from Tiskin's reduction procedure [10]. The simulated prototype is a divide and conquer algorithm that allocates an array of size $O(\nu n)$ and increases every cell from 0 to $\log(\nu n)$. At each step all the cells in the array are increased by 1, the array is then divided in half and each part is processed recursively. The recursion processes both sub-arrays, but in random order.

We compute a minimum squares (MSQ) estimates for c and d in $O(cN \log^d N)$. Since the behavior is asymptotic the first points may distort the values. To diminish this effect we computed several MSQ estimates, successively discarding the first points. We pessimistically chose the largest estimate.

The results show that for LCS and WLCS the δ values are similar to n, but not for PAM, notice the dispersion of the simulated points. Using $O(1)$ access time MMT is slower than the simulated algorithm for LCS and WLCS but, generally, faster for PAM. Contrary to what is predicted in Table 1, d is much smaller than 3, for a decent fitting of the model, *i.e.*, $R^2 > 0.95$, the time bound is always lower than $O(Nlog^2 N)$.

The graphics on the right show the number of operations for different sub-routines. The respective MSQs appear in the same line in the label. We estimate the ratio between node accesses and leaf accesses (Nodes/Leafs), which is always very close to $O(\log N)$. This value is important because it is against this time that the accesses to **A** and **B** add. If this ratio was 1 we would need to add a $O(\log n)$ factor to the final complexity. Since the ratio is close to $O(\log N)$ the accesses add only a constant term. The BinSteps line counts the average number of steps that is necessary in a binary search that computes a transition point. This value is $o(\log^1 N)$ because the MMT prototype uses inverse binary search. Instead of dividing an interval, it starts from a point and moves in powers of 2. Hence it usually runs on a small interval. TotalOps/N measures the total number of operations that the algorithm used, divided by N. The d estimate of this value is usually larger than the d value obtained in the time graphs, on the left. This may be related to cache effects. For LCS and WLCS the bound was at most $O(\log^{2.1} N)$, for PAM the value was higher, but the model fitting was extremely low.

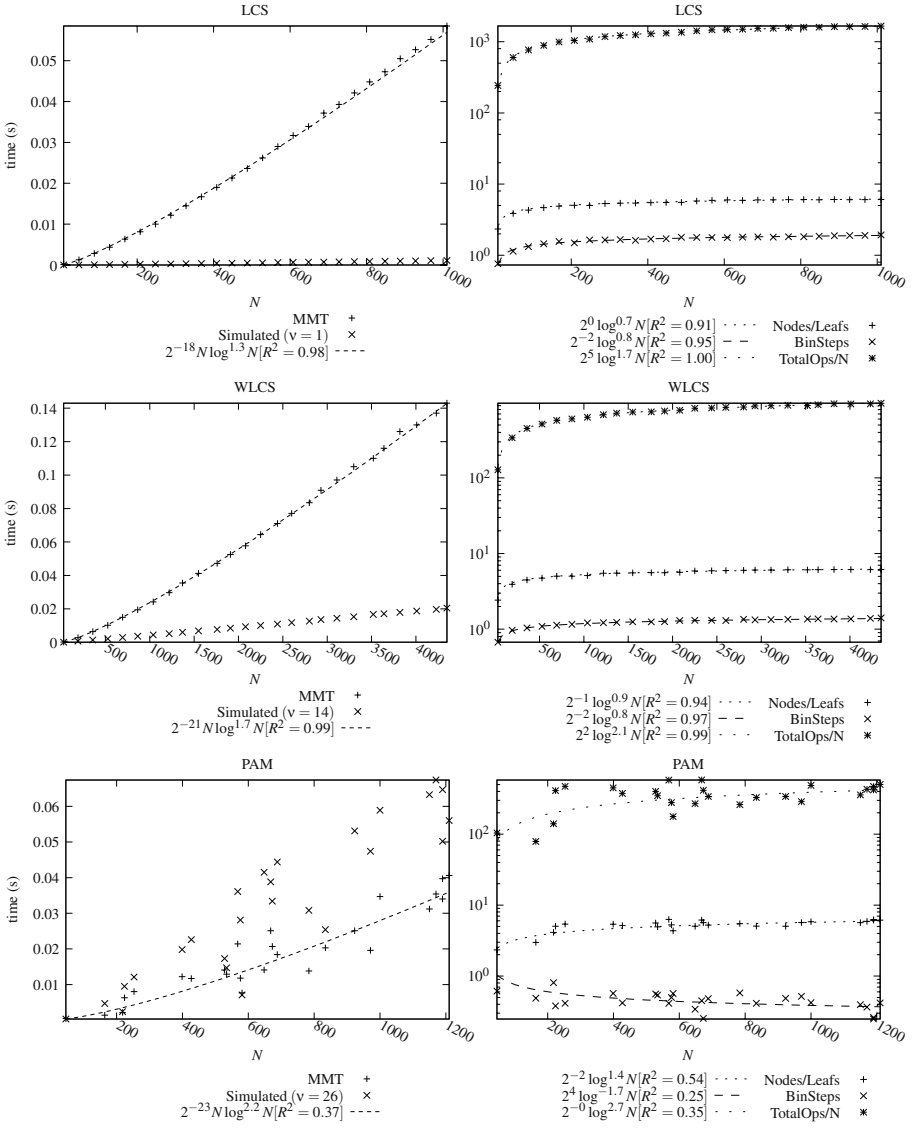


Fig. 4. Experimental testing of the MMT algorithm. The x-axis of the represents variable $N = \max\{n + \delta\}$. The y-axis of the graphs on the left measures the time in seconds. The y-axis of the graphs on the right measures the number of operations.

5 Conclusions

In this paper we studied algorithms for the max-plus product of Monge matrices. We analyzed the existing algorithms considering the core size, Table 1. The analysis showed that the existing algorithms are either sub-optimal or apply only to specific classes of

matrices. Alternatively we proposed a core sensitive algorithm, MMT. This algorithm is faster than the iterated *SMAWK* algorithm, except when the core is $O(n^2)$. For unit-Monge matrices the algorithm of Theorem 1 is theoretically faster than MMT, by an $O(\log n \log\log n)$ factor, which does not really hold in practice, *i.e.*, in practice MMT is faster than the theoretical bound. The algorithm of Theorem 1 can also be applied to matrices that result from Weighted LCSs, by using a ν time and space factor.

The MMT algorithm provided the first, as far as we know, non-trivial algorithm for the Fully-Incremental Alignment problem, a string processing problem that is relevant for bio-informatics. We expect MMT to have broad applications, since a myriad [9,16] of string processing and optimization problems [1] use Monge matrices.

Acknowledgments. We thank anonymous reviewers for several insightful remarks.

References

1. Burkard, R., Klinz, B., Rudolf, R.: Perspectives of Monge properties in optimization. Discrete Applied Mathematics 70(2), 95–161 (1996)
2. Apostolico, A., Atallah, M.J., Larmore, L.L., McFaddin, S.: Efficient parallel algorithms for string editing and related problems. SIAM J. Comput. 19(5), 968–988 (1990)
3. Schmidt, J.: All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. SIAM Journal on Computing 27, 972 (1998)
4. Tiskin, A.: Semi-local longest common subsequences in subquadratic time. J. Discrete Algorithms 6(4), 570–581 (2008)
5. Landau, G., Myers, E., Schmidt, J.: Incremental string comparison. SIAM Journal on Computing 27, 557–582 (1998)
6. Ishida, Y., Inenaga, S., Shinohara, A., Takeda, M.: Fully incremental LCS computation. In: Liśkiewicz, M., Reischuk, R. (eds.) FCT 2005. LNCS, vol. 3623, pp. 563–574. Springer, Heidelberg (2005)
7. Alves, C.E.R., Cáceres, E.N., Song, S.W.: An all-substrings common subsequence algorithm. Discrete Applied Mathematics 156(7), 1025–1035 (2008)
8. Landau, G.M.: Can dist tables be merged in linear time – An open problem. In: Proceedings of the Prague Stringology Conference 2006 (2006)
9. Tiskin, A.: Fast distance multiplication of unit-monge matrices. In: ACM-SIAM SODA, pp. 1287–1296 (2010)
10. Tiskin, A.: Semi-local string comparison: algorithmic techniques and applications. ArXiv e-prints (July 2007), arXiv0707.3619T
11. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
12. Aggarwal, A., Klawe, M., Moran, S., Shor, P., Wilber, R.: Geometric applications of a matrix-searching algorithm. Algorithmica 2(1), 195–208 (1987)
13. Iliopoulos, C.S., Mouchard, L., Rahman, M.S.: A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. Mathematics in Computer Science 1(4), 557–569 (2008)
14. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, 841–850 (2003)
15. Dayhoff, M.O., Schwartz, R.M., Orcutt, B.C.: A model of evolutionary change in proteins. Atlas of protein sequence and structure 5(suppl. 3), 345–351 (1978)
16. Crochemore, M., Landau, G., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM Journal on Computing 32(6), 1654–1673 (2003)

Why Large Closest String Instances Are Easy to Solve in Practice

Christina Boucher[1] and Kathleen Wilkie[2]

[1] David R. Cheriton School of Computer Science,
University of Waterloo
cabouche@cs.uwaterloo.ca
[2] Department of Applied Mathematics,
University of Waterloo
kpwilkie@math.uwaterloo.ca

Abstract. We initiate the study of the smoothed complexity of the Closest String problem by proposing a semi-random model of Hamming distance. We restrict interest to the optimization version of the Closest String problem and give a randomized algorithm, we refer to as *CSP-Greedy*, that computes the closest string on smoothed instances up to a constant factor approximation in time $O(\ell^3)$, where ℓ is the string length. Using smoothed analysis, we prove *CSP-Greedy* achieves a $\left((1+\frac{\epsilon e}{2^n})\right)^{\ell}$-approximation guarantee, where $\epsilon > 0$ is any small value and n is the number of strings. These approximation and runtime guarantees demonstrate that Closest String instances with a relatively large number of input strings are efficiently solved in practice. We also give experimental results demonstrating that *CSP-greedy* runs extremely efficiently on instances with a large number of strings. This counter-intuitive fact that "large" Closest String instances are easier and more efficient to solve gives new insight into this well-investigated problem.

1 Introduction

The Closest String is one of the central theoretical problems in bioinformatics and as such, has been studied extensively in bioinformatics and computational biology [7,8,14,15,17,20,21]. It has a wide variety of applications, including universal PCR primer design [10,15,18,26], genetic probe design [15], antisense drug design [9,15], finding transcription factor binding sites in genomic data [21], determining an unbiased consensus of a protein family [4], and motif-recognition [15,24,25]. The Closest String problem is NP-complete, unless P = NP [13], and therefore is unlikely to be solvable in polynomial time.

We initiate the study of the smoothed complexity of the optimization version of the Closest String problem, which can be defined as follows: given a set of ℓ-length strings $S = \{s_1, \ldots, s_n\}$[1] from the alphabet Σ, determine a string s of length ℓ such that $d(s, s_i) \leq d$ for all $s_i \in S$ and d is minimized. We refer

[1] Technically, this is a mulitset since we allow any string to occur multiple times

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 106–117, 2010.

to s as the *closest string* for the set S. Here $d(\cdot, \cdot)$ is the Hamming distance. The concept of smoothed analysis was introduced as an intermediate measure between average case analysis and worst case analysis; whereas average analysis studies the average behaviour of an algorithm over all instances of a problem, smoothed analysis studies the algorithm's average behaviour on each "local region" of the instance space [28]. If the algorithm has good average performance on each local region, then for any reasonable probabilistic distribution on the whole instance space, the algorithm should perform well. If the smoothed complexity is low, worst case instances are not robust under small changes. Most small changes to the instance destroy the property of being worst-case; a small random perturbation to the instance destroys the property of being worst-case.

Several other papers discuss the smoothed complexity of continuous problems [5,12] and of discrete problems [3,19]. The smoothed complexity of other string and sequence problems has been considered by Andoni and Krauthgamer [2], Manthey and Reischuk [22], and Ma [19]. Andoni and Krauthgamer [2] study the smoothed complexity of sequence alignment by the use of a novel model of edit distance; their results demonstrate the efficiency of several tools used for sequence alignment, most notably PatternHunter [21]. Manthey and Reischuk gave several results considering the smoothed analysis of binary search trees [22]. Ma demonstrated that a simple greedy algorithm runs efficiently in practice for Shortest Common Superstring [19], a problem that has application to string compression and DNA sequence assembly.

We give a smoothed analysis of an efficient probabilistic algorithm for Closest String instances where the alphabet is binary. To the best of our knowledge this is the first analysis of a natural random model of the Closest String problem. The main contributions of this paper are as follows:

- We describe our algorithm, *CSP-Greedy*, prove it achieves a 2-approximation guarantee, and give a bound on the probability that *CSP-Greedy* returns an optimal solution to the Closest String instance.
- Next, we empirically study *CSP-Greedy* and demonstrate that Closest String instances with a large number of strings (*i.e.* $n \geq 15$) are solved with extreme efficiency by this algorithm.
- Lastly, we define a natural model of perturbation for Closest String instances and use smoothed analysis to show that *CSP-Greedy* achieves an $1 + f(\epsilon, n, \ell)$-approximation guarantee in time $O(\ell^3)$, where $\epsilon > 0$ is any small value and $f(\epsilon, n, \ell)$ approaches zero in time that is exponential in n. Hence, this result gives analytical explanation for our empirical results.

1.1 Preliminaries

Let s be a string over the alphabet Σ. We restrict interest to Closest String instances where the alphabet is binary and hence, unless otherwise stated we assume Σ is the binary alphabet. Denote the length of s by $|s|$, and the jth letter of s by $s(j)$. Hence, $s = s(1)s(2)\ldots s(|s|)$. Lastly, we denote a function g to be an asymptotic estimation of f as $f \asymp g$.

Given a set of strings $S = \{s_1, \ldots, s_n\}$, each string of length ℓ, then a string s is a *closest string* for S if and only if there is no string s' such that $\max_{i=1,\ldots,n} d(s', s_i) < \max_{i=1,\ldots,n} d(s, s_i)$. Let s be a closest string for S then the *optimal closest distance* d is equal to $\max_{i=1,\ldots,n} d(s, s_i)$. We refer to a *majority vote* for S as the ℓ-length string containing the letter that occurs most often at each position; this string is not necessarily unique.

1.2 Previous Results

Lanctot *et al.* [15] gave a polynomial-time algorithm that achieves a $\frac{4}{3} + o(1)$ approximation guarantee. Li *et al.* [17] proved the existence of a polynomial time approximation scheme (PTAS) for this problem, though the high degree in the polynomial complexity of the PTAS algorithm renders this result only of theoretical interest. In 2005, Brejová *et al* [7] proved the existence of sharper upper and lower bounds for the PTAS on a slight variant of the Closest String problem (which they refer to as the Consensus Pattern problem) and in 2006, Brejová *et al* [8] improved upon the analysis of the PTAS for various random binary motif models. Andoni *et al.* [1] gave a novel PTAS that has improved time complexity, and most recently, Ma and Sun [20] presented a PTAS with time complexity $O(n^{\Theta(n^{-2})})$, which is currently the best known running time.

Another approach to investigate the tractability of this NP-complete problem is to consider the parameterized complexity of the Closest String problem. A problem φ is said to be *fixed-parameter tractable* (FPT) with respect to parameter k if there exists an algorithm that solves φ in $f(k) \cdot n^{O(1)}$ time, where f is a function of k that is independent of n [11]. Gramm *et al.* [14] demonstrated that the Closest String problem is FPT when the number of strings, denoted as $|S|$, remains fixed. This FPT result is based on an integer linear programming formulation with a constant number of variables (assuming n is fixed), and the application of the result of Lenstra [16], which states that ILP is polynomial-time solvable when the number of variables remains fixed. Unfortunately, such an integer programming formulation is only of theoretical interest since the corresponding algorithms lead to very long running times even when the number of strings is small. Other parameterizations of the Closest String problem also exist; for example, when d is fixed, the problem can be solved in $O(nl + nd(d+1)^d)$ time [14]. Ma and Sun gave an $O(n|\Sigma|^{O(d)})$ algorithm, which is a polynomial-time algorithm when $d = O(\log n)$ and Σ has constant size [20].

2 A Randomized Algorithm for Closest String

In [27], Schöning considers the following simple probabilistic algorithm for solving the NP-complete problem of k-SAT: randomly choose a starting assignment and subsequently augment this initial assignment until a satisfying one is obtained. Papadimitriou introduced this random paradigm in the context of 2-SAT and obtained an expected quadratic time bound [23]. This type of algorithms are referred to as Monte Carlo algorithms with one-sided error; a useful property of

such algorithm is that the error probability can be made arbitrarily small with repeated independent random repetitions of the search process. We analyze a similar probabilistic approach for the Closest String problem. Boucher and Brown [6] introduced a similar algorithm to Algorithm 2 for the decision version of the Closest String problem and conjectured about the probability that the algorithm successfully determines a solution to a Closest String instance; the results in this section resolve this conjecture.

Algorithm 2 begins with a string $s_{maj,0}$ randomly selected from all majority strings and iteratively *augments* the string so that it is closer to one of the strings in S a maximum of t times. Let t be the number of times a random majority string is chosen and augmented ℓ times. Later in the section we define t in terms of the parameters ℓ, n, and d. In order to determine a closest string corresponding to the optimal closest distance, this search process is repeated with the degeneracy parameter ranging from 0 to ℓ. We let Δd be the current degeracy in the search process.

Let $s_{maj,i}$ be $s_{maj,0}$ after it has been updated i times. At iteration $i+1$, we obtain the string $s_{maj,i+1}$ by augmenting $s_{maj,i}$ so that it has smaller Hamming distance to at least one string s_k in S where $d(s_k, s_{maj,i}) > \Delta d$. This process is repeated ℓ times at which point the process is restarted if there is at least one string s_k in S where $d(s_k, s_{maj,\ell}) > \Delta d$.

Algorithm 1. Procedure *augment*

Input: A set S of n ℓ-length strings, parameter d.
Output: A ℓ-length string s or "not found"
Let $\mathcal{S}$ be the set of all ℓ-length majority strings
Select $s_{maj,0}$ randomly from $\mathcal{S}$.
For $i = 0, \ldots, \ell$:
  If $d(s_{maj,i}, s_j) \leq d$ for all $s_j \in S$ then return s and terminate.
  Else $P = \{j : s_j \in S \text{ and } d(s_j, s_{maj,i}) > d\}$
  Choose any $p \in P$ and $1 \leq k \leq \ell$ such that $s_p(k) \neq s_{maj,i}(k)$
  Set $s_{maj,i+1}$ to be equal to s_p at position k and equal to $s_{maj,i}$ at all other positions
Return "not found"

Algorithm 2. *CSP-Greedy*

Input: A set S of n ℓ-length binary strings.
Output: A ℓ-length string s
For Δd each from $0 \rightarrow \ell$
  Repeat *augment* t times with parameter Δd

2.1 Approximation Guarantee for *CSP-Greedy*

In this subsection we give worst-case bounds on the approximation guarantee of *CSP-Greedy*. Also, we present examples of inputs for which the algorithm performs poorly and hence, give lower bounds on the approximation guarantee.

Theorem 1. *The approximation ratio of the CSP-Greedy algorithm is at most 2 for any alphabet size* $|\Sigma|$.

Proof. Since *CSP-Greedy* begins with a randomly selected majority string, it is sufficient to show that for any set of strings S the maximum distance from any majority string to any string in S is twice the optimal closest distance. Let S be a set of strings with optimal closest distance equal to d_{opt}. Without loss of generality assume 0^ℓ is a closest string for S and hence the maximum number of non-zero positions in each string $s_i \in S$ is at most d_{opt}. By the pigeonhole principle, the maximum number of positions containing greater than $n/2$ non-zero positions in a given column is at most $2d_{opt}$. Therefore, any majority string can have at most $2d_{opt}$ non-zero positions and is at distance at most $2d_{opt}$ from each string in S. □

The following example demonstrates that the 2-approximation guarantee is tight $S = \{10000001111, 01000001111, 11111111111, 11111111111\}$, where majority string is $c_1^* = 11111111111$ and optimal closest distance is $\ell/4$. On the first iteration the majority string could be altered to $c_2^* = 01111111111$. On the second iteration we could choose to alter this sequence back to c_1^*. It is possible to alternate between c_1^* and c_2^* and end up returning c_1^*, which is equal to twice the optimal closest distance.

2.2 Probabilistic Analysis of *CSP-Greedy*

The process of augmenting a randomly selected majority string ℓ times or until a closest string is found can be viewed as a Markov chain. This abstraction will be useful in achieving an upper bound on the probability that *CSP-Greedy* returns an optimal solution.

Let a set S be *uniquely satisfiable* if there exists exactly one string s^* where $d(s_i, s^*) \leq d^*$ for all $s_i \in S$. If S is an instance that does not have a string s such that $d(s, s_i) \leq d^*$ for all $s_i \in S$, then the *augment* procedure will return "not found". So we assume otherwise, that the set S is uniquely satisfiable and denote the probability of obtaining s^* when $\Delta d = d^*$ as p_s. If $s_{maj,i}$ is not equal to s^* then there is at least one letter of $s_{maj,i}$ that can be changed so that $d(s_{maj,i}, s^*)$ decreases by one; the probability of this occurring is at least $1/\ell$. Denote $X_i \in \{0, 1, \ldots, \ell\}$ $(i = 0, 1, \ldots)$ as the random variable that is equal to the Hamming distance between $s_{maj,i}$ and s^*, where i is the number of iterations of the *augment* procedure. Each time a position is selected and the value of that position is augmented, either the Hamming distance is increased or decreased by one.

The process $X_0, X_1, X_2, \ldots$ is a Markov chain with a barrier at state ℓ and contains varying time and state dependent transfer probabilities. This process is overly complicated and we instead choose to analyze the following process: $Y_0, Y_1, Y_2, \ldots$, where Y_i is the random variable which is equal to the state number after i steps and there exists infinitely many states. Initially, this Markov chain is started like the stochastic process above (*i.e.* $Y_0 = X_0$). As long as the inner loop is iterating, we let $Y_{i+1} = X_i - 1$ if the process decreases the Hamming distance

between $s_{maj,i}$ and s^* by one; and $Y_{i+1} = X_i + 1$ otherwise. After the loop exits, we continue with the same transfer probabilities. By induction on i, it is clear that for each i, $X_i \leq Y_i$, and it follows that p_s is at least $\Pr[\exists t \leq c\ell : Y_t = 0]$.

We made the assumption that the set was uniquely satisfiable, however, this assumption is not needed – the random walk may find another closest string while not in the terminating state but this possibility only increases the probability the algorithm terminates.

The following asymptotic estimation is used in the proof of the next theorem. See the appendix for the justification of this fact.

Fact 1. *For $i > 0$ and $j > 0$ the following asymptotic estimation exists:*

$$\sum_{i=0}^{j} \binom{2i+j}{i} \left(\frac{\ell-1}{\ell}\right)^i \left(\frac{1}{\ell}\right)^{i+j} \asymp \left(\frac{1}{\ell-1}\right)^j. \tag{1}$$

Theorem 2. *Let S be a set of strings and d^* be the optimal closest distance. Then the probability of procedure 'augment' obtaining a closest string for S is at least $e \cdot 2^{-\ell}$ for sufficiently large ℓ.*

Proof. Suppose s^* is a closest string for S and let k be the Hamming distance between $s_{maj,0}$ and s^*. Denote $\Pr[Y_{i+1} = j - 1 | Y_i = j]$ by q_i. Given that the Markov chain starts in some state j, it can reach a halting state in at least j steps by making transitions through the states $j-1$, $j-2$, …, $1, 0$. The probability of this happening is at least $\sum_{i=0}^{j} q_i$. For $i = 0, 1, 2, \ldots$ the halting state can be reached after $2i + j$ steps, where i steps are "bad" and $j + i$ steps are "good". The probability of this happening is:

$$\Pr[Y_{2i+j} = 0, \text{ and } Y_k > 0 \,\forall\, k < 2i + j \,|\, Y_0 = j],$$

which is at least $q_i^{i+k}(1 - q_i)^i$ times the number of ways of arranging i bad steps and $i + j$ good steps such that the sequence starts in state j, ends in state 0, and does not reach 0 before the last step. Using the Ballot theorem we know there are $\binom{2i+j}{i}\frac{i}{2i+j}$ possible arrangements of these i and $i + j$ steps. Therefore, the above probability is at least:

$$\binom{2i+j}{i}\frac{i}{2i+j}(1 - q_i)^i q_i^{i+j}.$$

This expression is not defined in the case $i = j = 0$. In this case, the probability is equal to 1. Thus, we have:

$$\begin{aligned}
\Pr[Y_{2i+j} = 0, \text{ and } Y_k > 0 \,\forall\, k < 2i + j | Y_0 = j] = \\
&\geq \sum_{j=0}^{\ell} 2^{-\ell} \binom{\ell}{j} \sum_{2i+j \leq c\ell} \binom{2i+j}{i} \frac{i}{2i+j} \left(\frac{\ell-1}{\ell}\right)^i \left(\frac{1}{\ell}\right)^{i+j} \\
&\geq 2^{-\ell} \sum_{j=0}^{\ell} \binom{\ell}{j} \sum_{i=0}^{j} \binom{2i+j}{i} \left(\frac{\ell-1}{\ell}\right)^i \left(\frac{1}{\ell}\right)^{i+j}
\end{aligned}$$

Using Fact 1, we have:

$$\begin{aligned}\Pr[Y_{2i+j} = 0, \text{ and } Y_k > 0 \,\forall\, k < 2i + j | Y_0 = j] &\asymp 2^{-\ell} \sum_{j=0}^{\ell} \binom{\ell}{j} \left(\frac{1}{\ell - 1}\right)^j \\ &= \left(\frac{\ell}{2(\ell - 1)}\right)^{\ell} \text{ [by the Binomial theorem]} \\ &\asymp e \cdot 2^{-\ell} \left(\frac{\ell}{\ell - 1}\right) \text{ [by Taylor's theorem]}\end{aligned}$$

For sufficiently large ℓ we have the probability is at least $e \cdot 2^{-\ell}$. □

The following corollary, an immediate consequence of the previous lemma, bounds the probability of error.

Corollary 1. *If $t = \ell$ and ℓ is sufficiently large, CSP-Greedy has time complexity $O(\ell^3)$ and one-sided error probability of no more than $\left(1 - e \cdot 2^{-\ell}\right)^{\ell}$.*

3 Experimental Results

We empirically investigate how the number of augmentations changes as the number of strings (parameter n) increases. For these experiments, we fix ℓ to be 15, d to be 4, and vary the value of n from 12 to 36. For each value of n, we generate 100 motif sets at random by selecting a string s at random from the set of all 2^{ℓ} possible strings and then n motif strings at random from the set of all strings of distance at most d from s. For each motif set we run Procedure *augment* twice–once with $s_{maj,0}$ initialized to be equal to a majority string, and a second time with $s_{maj,0}$ initialized to be equal to a random string–and in both cases, we count the number of alterations required to obtain a closest string. We determine the mean number of alterations for 100 motif sets. Figure 1 illustrates this data, which shows that the number of augmentations required to obtain a closest string is significantly larger if $s_{maj,0}$ is initialized to be a random string. Further, as the value of n increases the disparity between the number of augmentations of a majority string and the number of augmentations of a random string increases substantially.

Table 1 illustrates the change in the (Hamming) distance as the values n, ℓ, and d increase. We consider three different values of ℓ and d, varied the value of n from 5 to 35, generate 100 random motif instances, calculate the Hamming distance between $s_{maj,0}$ and s^* (a closest string found by Algorithm 2) and determine the mean Hamming distance of the 100 motifs. This data demonstrates a drastic decrease in the distance between the majority string and the closest string as n increases. Note that when the value of n is significantly large the distance between the majority string and the closest string is equal to zero – implying the majority string is a closest string.

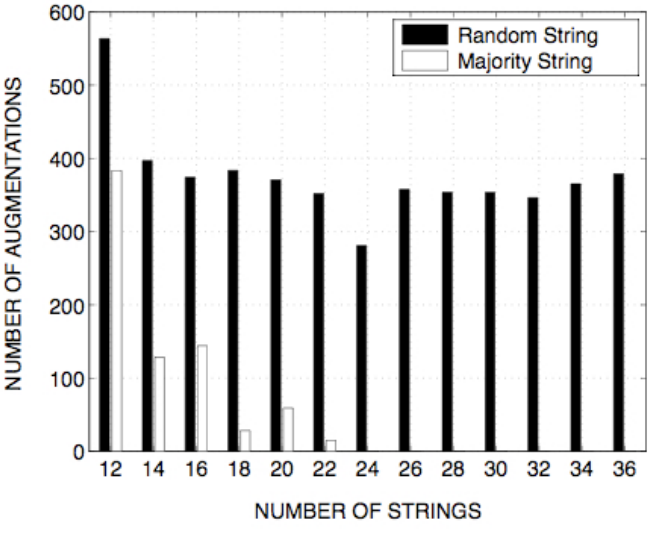


Fig. 1. Illustration of the effect of the initialization of the starting string on efficiency of Procedure *augment* as the value of n increases. A comparison between the mean number of augmentations required to obtain a closest string if starting from a majority string (white) or if starting from a random string (black). One hundred random motif sets are generated for each value of n and the mean of the augmentations is determined. ℓ is fixed to 15 and d is fixed to 4.

Table 1. An Illustration of the variation in the Hamming distance between a majority and closest string as ℓ, d, and n change. An illustration of the Hamming distance between a randomly chosen majority string and a closest string with respect to n. We considered the following (ℓ, d) pairs: $(15, 4)$, $(18, 6)$, and $(25, 5)$ and varied n to be every even value from 5 to 35. For each (ℓ, d) pair and value of n we generated 100 random motif sets, determined the Hamming distance between a randomly selected majority string and the closest string found by the algorithm, and calculated the mean Hamming distance.

n	(ℓ, d)		
	$(15, 4)$	$(18, 6)$	$(25, 5)$
5	3.2	2.9	3
8	2.7	2.3	2.1
10	0.8	1.0	1.8
12	0	0.7	1.2
15	0	0	0.5
25	0	0	0
30	0	0	0
35	0	0	0

In summary, our experimental results illustrate the following trends: as the value of n increases the (Hamming) distance between a randomly selected majority string and a closest string decreases and the number of augmentations required to obtain a closest string from a majority string decreases. Furthermore, regardless of the value of ℓ and d, for significantly large values of n the majority string is also likely to be a closest string. Similar results were also reported by Boucher and Brown [6].

4 Smoothed Analysis of *CSP-Greedy*

Using smoothed analysis, we demonstrate that the approximation ratio of the greedy algorithm on small perturbations of the worst-case instances is equal to $\left(1 + \frac{\epsilon(e-1)}{2^n}\right)^{\ell}$. Hence, our analysis shows that the approximation is equal to $1 + o(1)$ for significantly large n. This result explains why *CSP-Greedy* performs well in practice on large instances (*i.e.* instances containing a significantly large number of strings).

A *perturbed instance* S is defined to be $S' = \{s'_1, s'_2, \ldots, s'_n\}$, where each s'_i has length ℓ and each s'_i is obtained by mutating uniformly at random each letter of s_i with a small probability $p > 0$. In more general terms, an adversary chooses n length-ℓ sequences from the binary alphabet at random, and every symbol is perturbed with a small probability p. Next, let S be a closest string instance, S' be the corresponding perturbed instance, and q be $\Pr[s_i(j) = 1]$. We have

$$\begin{aligned} Pr[s'_i(j) = 0] &= \Pr[s_i(j) = 1] \Pr[s_i(j) \text{ was permuted}] + \\ &\quad \Pr[s_i(j) = 0] \Pr[s_i(j) \text{ was not permuted}] \\ &= qp + (1 - q)(1 - p). \end{aligned}$$

Assuming, $\epsilon > 0$ is a small number, and the perturbation probability $\frac{\epsilon \log(\ell n)}{\ell n} \leq p \leq \frac{1}{2}$, we obtain the following:

$$\Pr[s'_i(j) = 0] = 1 - q - p + 2qp \tag{2}$$

$$\geq 1 - q - p + q \quad \text{since } p \leq 1/2 \tag{3}$$

$$\geq \frac{\epsilon \log(\ell n)}{\ell n} \tag{4}$$

The next theorem, our main result, demonstrates that for significantly large ℓ, as n (and to a lesser extend ℓ) increases the approximation ratio approaches 1. This result explains our experimental results analytically.

Theorem 3. (*CSP-Greedy* under limited randomness) *For any given small $\epsilon > 0$, for perturbation probability $\frac{\epsilon \log(\ell n)}{\ell n} \leq p \leq \frac{1}{2}$ and significantly large ℓ, the expected ratio of 'CSP-Greedy' on the perturbed instances is $\left(1 + \frac{\epsilon e}{2^n}\right)^{\ell}$.*

Proof. Given a Closest String instance S, we define the instance as *good* if each majority string of S is also a closest string for S; otherwise, we define

the instance as *bad*. Since each iteration of the *augment* procedure begins by selecting a random majority string and determining whether it has distance at most d from each string in S, it follows that *CSP-Greedy* will return a closest string when S is a good instance. Let p_{bad} be the probability that the perturbed instance is bad.

In order to bound p_{bad} we first calculate the probability that a majority string does not match the closest string at a particular position. There exists at least one string s such that $d(s, s'_i) \leq d$ for all $s'_i \in S'$. Without loss of generality assume that 0^ℓ is a closest string. We calculate the probability that 0^ℓ is a majority string. Let $X_{i,j}$ be a binary random variable representing the symbol of s_i at the j-th position. For a given j, let the number of ones be $X_j = \sum_i X_{i,j}$.

$$\begin{aligned}\Pr[X_j > n/2] &= \sum_{i=n/2}^{n} \binom{n}{i} (\Pr[s'_i(j) = 1])^i (1 - \Pr[s'_i(j) = 1])^{n-i} \\ &\geq 1 - (1 - \Pr[s'_i(j) = 1])^n\end{aligned}$$

Let $\alpha = 1 - (1 - \Pr[s'_i(j) = 1])^n$. We give a lower bound for the probability that the instance is good.

$$\begin{aligned}1 - p_{bad} &= 1 - \Pr[S' \text{contains at least one bad column}] \\ &= 1 - \sum_{i=1}^{\ell} \binom{\ell}{i} \Pr[X_j > n/2]^i \\ &\geq 1 - \sum_{i=1}^{\ell} \binom{\ell}{i} \alpha^i \\ &\geq 1 - \alpha^\ell \text{ [by Binomial Theorem]}\end{aligned}$$

Therefore, it follows that p_{bad} is at most $(1 - (1 - \Pr[s'_i(j) = 1])^n)^\ell$ and hence, we get:

$$p_{bad} \leq (1 - (1 - \Pr[s'_i(j) = 1])^n)^\ell \leq \left(1 - \frac{\epsilon}{2^n}\right)^\ell$$

In Theorem 1 the greedy algorithm was proved to have a worst-case approximation ratio of 2. Therefore, by Theorem 1 and Corollary 1 we obtain the following:

$$\begin{aligned}E[ratio] &\leq 2 \left(1 - \frac{\epsilon}{2^n}\right)^\ell \left(1 - \frac{e}{2^\ell}\right)^\ell \text{ [by Corollary 1]} \\ &\leq 2 \left(1 + \frac{\epsilon e}{2^{n+\ell-1}} - \frac{e}{2^\ell} - \frac{\epsilon}{2^n}\right)^\ell\end{aligned}$$

For significantly large values of n and ℓ, we have:

$$\left(1 + \frac{\epsilon e}{2^{n+\ell-1}} - \frac{e}{2^\ell} - \frac{\epsilon}{2^n}\right)^\ell \leq \frac{1}{2^{1/\ell}} \left(1 + \frac{\epsilon e}{2^n}\right)^\ell,$$

and therefore,

$$E[ratio] \leq \left(1 + \frac{\epsilon e}{2^n}\right)^\ell. \qquad \square$$

We note that our perturbation is very small compared to the size of the instance. With perturbation probability $p = \frac{\epsilon \log(\ell n)}{\ell n}$, each set of n strings of length ℓ is expected to change by $\log(\ell n)$ letters.

Acknowledgements

The authors would like to thank Professor Bin Ma for his discussions and insights concerning the results presented in this paper and Professor Ming Li for suggesting this area of study.

References

1. Andoni, A., Indyk, P., Patrascu, M.: On the optimality of the dimensionality reduction method. In: Proc. of 47th FOCS, pp. 449–456 (2006)
2. Andoni, A., Krauthgamer, R.: The smoothed complexity of edit distance. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 357–369. Springer, Heidelberg (2008)
3. Banderier, C., Beier, R., Mehlhorn, K.: Smoothed analysis of three combinatorial problems. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 198–207. Springer, Heidelberg (2008)
4. Ben-Dor, A., Lancia, G., Perone, J., Ravi, R.: Banishing bias from consensus strings. In: Hein, J., Apostolico, A. (eds.) CPM 1997. LNCS, vol. 1264, pp. 247–261. Springer, Heidelberg (1997)
5. Blum, A., Dunagan, J.D.: Smoothed analysis of the perceptron algorithm for linear programming. In: Proc. of 13th SODA, pp. 905–914 (2002)
6. Boucher, C., Brown, D.G.: Detecting motifs in a large data set: applying probabilistic insights to motif finding. In: Rajasekaran, S. (ed.) BICoB 2009. LNCS, vol. 5462, pp. 139–150. Springer, Heidelberg (2009)
7. Brejová, B., Brown, D.G., Harrower, I., López-Ortiz, A., Vinař, T.: Sharper upper and lower bounds for an approximation scheme for CONSENSUS-PATTERN. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 1–10. Springer, Heidelberg (2005)
8. Brejová, B., Brown, D.G., Harrower, I., Vinař, T.: New bounds for motif finding in strong instances. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 94–105. Springer, Heidelberg (2006)
9. Deng, X., Li, G., Li, Z., Ma, B., Wang, L.: Genetic design of drugs without side-effects. SIAM J. Comput. 32(4), 1073–1090 (2003)
10. Dopazo, J., Rodríguez, A., Sáiz, J.C., Sobrino, F.: Design of primers for PCR amplification of highly variable genomes. CABIOS (9), 123–125 (1993)
11. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
12. Dunagan, J.D., Spielman, D.A., Teng, S.-H.: Smoothed analysis of the renegar's condition number for linear programming. In: Proc. of SIOPT (2002)
13. Frances, M., Litman, A.: On covering problems of codes. Th. Comp. Sys. 30(2), 113–119 (1997)
14. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37, 25–42 (2003)

15. Lanctot, J.K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. Inf. Comput. 185(1) (2003)
16. Lenstra, W.H.: Integer programming with a fixed number of variables. Math. of OR 8, 538–548 (1983)
17. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. J. Comput. Syst. Sci. 65(1), 73–96 (2002)
18. Lucas, K., Busch, M., Össinger, S., Thompson, J.A.: An improved microcomputer program for finding gene- and gene family-specific oligonucleotides suitable as primers for polymerase chain reactions or as probes. CABIOS 7, 525–529 (1991)
19. Ma, B.: Why greedy works for shortest common superstring problem. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 244–254. Springer, Heidelberg (2008)
20. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 396–409. Springer, Heidelberg (2008)
21. Ma, B., Tromp, J., Li, M.: PatternHunter: faster and more sensitive homology search. Bioinformatics 18(3), 440–445 (2002)
22. Manthey, B., Reischuk, R.: Smoothed analysis of binary search trees. Th. Comp. Sci. 378(3), 292–315 (2007)
23. Papadimitriou, C.H.: On selecting a satisfying truth assignment. In: Proc. of 32nd FOCS, pp. 163–169 (1991)
24. Pavesi, G., Mauri, G., Pesole, G.: An algorithm for finding signals of unknown length in DNA sequences. Bioinformatics 17, S207–S214 (2001)
25. Pevzner, P., Sze, S.: Combinatorial approaches to finding subtle signals in DNA strings. In: Proc. of 8th ISMB , pp. 269–278 (2000)
26. Proutski, V., Holme, E.C.: Primer master: A new program for the design and analyiss of PCR primers. CABIOS 12, 253–255 (1996)
27. Schöning, U.: A probabilistic algorithm for k-sat and constraint satisfaction problems. In: Proc. of 40th FOCS, pp. 410–414 (1999)
28. Spielman, D.A., Teng, S.-H.: Smoothed analysis of algorithms: why the simplex algorithm ususally takes polynomial time. In: Proc. of 33rd STOC, pp. 296–305 (2001)

A PTAS for the Square Tiling Problem

Amihood Amir[1,⋆], Alberto Apostolico[2,⋆⋆], Gad M. Landau[3,⋆⋆⋆], and Oren Sar Shalom[4]

[1] Bar-Ilan University, Israel and Johns Hopkins University, USA
amir@cs.biu.ac.il
[2] Georgia Tech, USA and University of Padova, Italy
axa@cc.gatech.edu
[3] University of Haifa, Israel, and NYU-Poly, USA
landau@cs.haifa.ac.il
[4] Bar-Ilan University, Israel
oren.sarshalom@gmail.com

Abstract. The Square Tiling Problem was recently introduced as equivalent to the problem of reconstructing an image from patches and a possible general-purpose indexing tool. Unfortunately, the Square Tiling Problem was shown to be $\mathcal{NP}$-hard. A 1/2-approximation is known.

We show that if the tile alphabet is fixed and finite, there is a Polynomial Time Approximation Scheme (PTAS) for the Square Tiling Problem with approximation ratio of $(1 - \frac{\epsilon}{2 \log n})$ for any given $\epsilon \leq 1$.

1 Motivation

Recently [2] Amir and Parienty introduced the *Patches Model* as a possible abstraction of ad-hoc techniques that have been used to index various domains. The idea is to slice the images that we want to index into many small overlapping *patches*, or *tiles*. Patterns whose patches match a large number of patches in an indexed object, are likely to appear in the image. Similar methods have been used in Computational Biology (e.g. [15,7,3,12,6,5]), Linguistics (e.g. [8,1]), Image Processing (e.g. [14,11,4]), or Audio Indexing (e.g. [9]).

The first task tackled in [2] was reconstructing an image from its patches – the *Square Tiling Problem*. Unfortunately, it was proven that this problem is $\mathcal{NP}$-hard. An image constructed from $n \times n$ tiles has $2n^2 - 2n$ "*seams*" between tiles. If it is correctly constructed then two adjacent tiles *match* at the seams, i.e., have equal alphabet symbols on their adjacent edges. We then say that the seam is *correct*, otherwise there is an *error* at the seam. We count a single error

⋆ Partly supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217.

⋆⋆ Partly supported by BSF grant 2008217.

⋆⋆⋆ Partly supported by the National Science Foundation Award 0904246, Israel Science Foundation grants 35/05 and 347/09, the Israel-Korea Scientific Research Cooperation, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 118–126, 2010.

at the seam whether one or two symbols are not equal at the adjacent tiles. An *approximation* in that context is an $n \times n$ square where "many" seams match. In [2] a polynomial-time algorithm that constructs a square with at least $\frac{1}{2}$ of the seams being correct, was shown. If the idea of patch indexing is to have any chance of applicability, much better approximations are necessary.

In this paper we present a Polynomial Time Approximation Scheme (PTAS) for the square tiling problem with a fixed finite alphabet. We show that for such finite alphabets, for any given ϵ and n^2 tiles, there is an algorithm polynomial in n and ϵ that constructs a square with at least $(1 - \frac{\epsilon}{2 \log n})n^2$ correct seams.

The paper is constructed as follows. We begin with background and definitions in Section 2. In Section 3 we show how to reconstruct a rectangle of size $n \times \log n$ tiles in polynomial time (or, for a given c an $n \times c$ rectangle). We then show in Section 4 how multidimensional knapsack techniques can be used to approximate a square tiling.

2 Background and Definitions

The definition below was used in [2] to combinatorially describe the patches concept.

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 converse.

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.

In [2] it was proven that the problem of constructing an image from overlapping patches is equivalent to the square tiling problem, where we are given 2×2 patches over alphabet Σ and we are only allowed to place patches next to each other if their symbols match. Formally:

Definition 3. *A patch A may be* correctly 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. The common edge of two patches is called a* seam. *The seam is* correct *if the adjacent tiles are correctly placed. Otherwise, it is an* error.

Definition 4. *The* Square Tiling Problem *is defined as follows:*

INPUT: A multiset S of n^2 tiles. Each tile is a 2×2 matrix over alphabet Σ
DECIDE: Whether there exists a square of tiles correctly placed next to each other, whose tiles are exactly those in the multiset.

In [2] it was proven that the Square Tiling Problem is $\mathcal{NP}$-hard.

We seek a polynomial time algorithm that approximates the square. The approximation means reconstructing an image from all of the tiles, with the minimum amount of seam errors.

Our goal is to improve this result, and in fact we even generalize that. Assume that the maximum number of correct seams that can be achieved by *any* tiling of a given set of patches S is s_{max}. Then for any given ϵ we can tile, in time polynomial in the size of S and ϵ, the patches of S achieving at least $(1 - \frac{\epsilon}{\log n})s_{max}$ correct seams.

3 Rectangle Tiling

We begin by showing that the $\mathcal{NP}$-hardness is dependent on the dimensions of the square. For some rectangle dimensions, the Tiling Problem is polynomial-time computable.

Definition 5. *The* entropy *of multiset m of size $a \cdot b$ is the minimum number of errors obtained by tiling m as an $a \times b$ rectangle.*

Theorem 1. *The entropy of all multisets of size $n \times \frac{\log n}{\epsilon}$ can be computed in time polynomial in ϵ and n.*

Proof: We want to compute the entropy for **all** multisets, so we must first make sure that there are not too many multisets.

Claim. There is a polynomial number of multisets of size $n \times \frac{\log n}{\epsilon}$.

Proof: The alphabet is fixed and finite. Let us assume that $\Sigma = \{1, ..., c\}$, So there are $O(c^4)$ types of tiles.

A multiset is a histogram of the types of tiles that compose it. There are at most $n \cdot \frac{\log n}{\epsilon}$ tiles of each type, while the sum of all types is also $n \cdot \frac{\log n}{\epsilon}$. Combinatorially, the number of different multisets it is equal to the number of combinations to insert n' balls into k' bins, when $n' = n \cdot \frac{\log n}{\epsilon}$ and $k' = c^4$. There are $\binom{n'+k'-1}{n'} = O(n'^{k'-1}) = O((n \cdot \frac{\log n}{\epsilon})^{c^4-1})$ multisets. □

We now show an exact polynomial-time algorithm for a fixed finite alphabet that calculates the entropy for each multiset m of size $n \cdot \frac{\log n}{\epsilon}$. we will need an

auxiliary data structure to keep track of some values for the multisets constructed during the execution of the algorithm. In particular, consider a possible tiling of multiset of size $i \cdot \frac{\log n}{\epsilon}$ into a rectangle of i columns and $\frac{\log n}{\epsilon}$ rows. The rightmost column, rc, of such a tiling is composed of $\frac{\log n}{\epsilon}$ 2×2 patches. For the sake of improving the time complexity, we consider the column composed only of the $\frac{2 \log n}{\epsilon}$ symbols on the *right side* of the patches in the rightmost column. It is possible that different columns of $\frac{\log n}{\epsilon}$ patches have the same right side. For any multiset m and right side r, choose a tiling with the lowest entropy. For that tiling, record in addition to the right side r, also the $\frac{2 \log n}{\epsilon}$ symbols on the *left side* of the patches in the rightmost column.

Auxiliary Data

Define a 2-dimensional array M with the following values:

At iteration i, for each multiset m of size $i \cdot \frac{\log n}{\epsilon}$, that represents a rectangle of i columns and $\frac{\log n}{\epsilon}$ rows, and for each r – the right side of the rightmost column of multiset m:

- $M[m, r].entropy$ holds the minimum entropy of m subject to the constraint that r is the right side column of the $\frac{\log n}{\epsilon} \times i$ rectangle tiling of m.
- $M[m, r].left$ is the left side of the rightmost column in a tiling that obtained the minimum entropy.

  This field's goal is to enable us to reconstruct the optimal tiling of m.

The size of array M is bounded by the product of the number of multisets and the number of possibilities for the right side of column r. The number of multisets for the last iteration was calculated above as $O((n\frac{\log n}{\epsilon})^{c^4-1})$.

Calculation of the number of different possibilities for the right side of a column:

There are c^2 possibilities for each tile, and $\frac{\log n}{\epsilon}$ tiles altogether. Therefore, the right side of a column can be arranged in

$(c^2)^{\frac{\log n}{\epsilon}} = (2^{\log c^2})^{\frac{\log n}{\epsilon}} = (2^{\log n})^{\frac{log c^2}{\epsilon}} = n^{\frac{\log c^2}{\epsilon}} = n^{\frac{2 \log c}{\epsilon}} = z$ different ways.

Thus the size is clearly polynomial.

Algorithm Outline

Initialization phase: initialize all of the *entropy* values of M to ∞.

A column can be considered a Cartesian product of two sides $< l, r >$, so there are z^2 different columns.

Count the number of errors of every possible column c, i.e, an ordered $\frac{\log n}{\epsilon}$ tiles placed one above the other. In this case, m is the multiset comprised by c and r is the right side of column c.

Assume we have computed array M for the first i iterations.

Iteration step: For each combination of (m, r) computed in the previous iteration, the algorithm attaches to r all possibilities as column c. Let C be the multiset of the elements of c. Then each such column c creates a new multiset $m \leftarrow m \cup C$, and new right side $r \leftarrow$ right side of column$c$. The new entropy e' the entropy of the old entry + the number of errors introduced by attaching c to the right of the old entry.

Thus, $M[m, r] \leftarrow \min(M[m, r], e')$

Correctness:
We prove by induction that for each combination of multiset m of size $i \cdot \frac{\log n}{\epsilon}$ and right side of column r, the algorithm finds the entropy of (m, r).

Base Case: For $i = 1$, the assumption holds, because the algorithm goes through all columns and picks the minimum for each combination.

Inductive Step: Assume correctness for i and prove for $i + 1$: Let m_{i+1} be a multiset of size $(i + 1) \cdot \frac{\log n}{\epsilon}$ and r_{i+1} a right side of a column. Any optimal tiling of m_{i+1} as a rectangle such that r_{i+1} is its right side of the rightmost column, is a selection among an optimal tiling of multiset m_i as a rectangle of size $i \times \frac{\log n}{\epsilon}$ such that the right side of the rightmost column is r_i adjacent to column $< l_{i+1}, r_{i+1} >$, such that the number of errors of m_i and $< l_{i+1}, r_{i+1} >$ is smallest. By the induction assumption, the algorithm finds m_i as iteration i. Iteration $i + 1$ attaches all possible columns, particularly $< l_{i+1}, r_{i+1} >$.

Therefore the algorithm finds (m_{i+1}, r_{i+1}). □

Claim. The algorithm's complexity is polynomial.

Proof:
Initialization phase: Considering all combinations of right side of column r and column $< l', r' >$ is $O(z^3) = O(n^{\frac{6 \log c}{\epsilon}})$.

Column construction: We seek after the number of multisets of size $i \cdot \frac{\log n}{\epsilon}$ $\forall i \leq n$. There are $\mathrm{O}((n \cdot \frac{\log n}{\epsilon})^{c^4-1})$ multisets of size exactly $n \cdot \frac{\log n}{\epsilon}$. Therefore, there are $\mathrm{O}(n \cdot (n \cdot \frac{\log n}{\epsilon})^{c^4-1})$ multisets altogether. For each multiset we go through all m^2 columns and perform a constant time work.
Therefore, the algorithm's running time is $\mathrm{O}(n \cdot (n \cdot \frac{\log n}{\epsilon})^{c^4-1} \cdot n^{\frac{4 \log c}{\epsilon}})$.

Finding the optimal tiling: Once a multiset $m =< t_1, ..., t_{c^4} >$ of size $n \cdot \frac{logn}{\epsilon}$, is constructed, it's trivial to reverse the algorithm and tile it as a rectangle with the minimum number of errors. □

4 The Approximation

In the previous section we enumerated all multisets and their entropies. That action could be referred as a pre-processing action, because it does not use the input of the problem, but only exploits the problem size n.

We now partition the n^2 input patches into $\frac{n \cdot \epsilon}{\log n}$ sets of $n \cdot \frac{\log n}{\epsilon}$ tiles, such that the sum of their entropies is minimal. This will also be done by partitioning *all* multisets of size n^2 into such sets.

Algorithm Outline:

Step 1 - Find the entropy for all multisets of size $n \cdot \frac{\log n}{\epsilon}$.

Step 2 - Select a set of multisets, consisting exactly of the input tiles, with minimal entropy.

Implementation:
Step 1: The implementation of step 1 was shown in Section 3.

Step 2: This problem is similar to another $\mathcal{NP}$-hard problem - The *Multidimensional Knapsack problem*. It needs to select a multiset of given objects (or items) in such a way that the total profit of the selected objects is maximized while a set of knapsack constraints are satisfied.

Unfortunately, MDK is $\mathcal{NP}$-hard. It was also shown that finding an FPTAS even for a special case where all profits are the same and equal to 1 and $m = 2$ is $\mathcal{NP}$-hard [10]. Moreover, the problem is $\mathcal{NP}$-hard in the strong sense and thus any dynamic programming approach would result in strictly exponential time bounds [13].

We want an exact algorithm to be polynomial on the one hand, and on the other hand to support multiple knapsacks, utilizing the unique characteristics of our problem.

Let $MS = \{c_1, \ldots, c_x\}$, be the set of all multisets of size $n \cdot \frac{\log n}{\epsilon}$, where x is of size $O((n \cdot \frac{\log n}{\epsilon})^{c^4-1})$. Multiset c_i can be represented by $< w_{i1}, \ldots, w_{ic^4} >$, where w_{ij} is the number of patches of type j there are in multiset c_i. Let e_i be the entropy of multiset c_i. We assume that MS is sorted in non-decreasing order of its entropy, i.e., $\forall i \leq m-1$, $e_i \leq e_{i+1}$.

The input S of the tiling problem is a set of patches S of size n^2. S can be represented by the tuple $< S_1, \ldots, S_{c^4} >$, where $_i$ is the number of input tiles of type i. Our task is to find a multiset of multisets $C \subseteq MS$, such that $\bigcup_{c \in C} c = S$, i.e. $\forall j$, $1 \leq j \leq c^4$, $\sum o_i \cdot w_{ij} = S_j$, where o_i is the number of occurrences of c_i in C.

The Dynamic Programming Matrix:
$T[1..x\ ;\ 1..(n^2)^{c^4}]$ is a matrix with the following values:

Let $< b_1, \ldots, b_{c^4} >$ be a multiset of size between 0 and n^2, and let i be a number $1 \leq i \leq x$.

> If there does not exist a multiset which is the union of sets from $\{c_1, \ldots, c_i\}$, and whose elements are exactly the patches $< b_1, \ldots, b_{c^4} >$, then $T(i, < b_1, \ldots, b_{c^4} >) = \infty$.
> Otherwise, let L be such a multiset where $\sum_{c \in L}(\text{entropy of } c) = E$ is smallest.
> Set $T(i, < b_1, \ldots, b_{c^4} >) = E$.

Algorithm Outline:
The matrix is filled using dynamic programming:

Initialization: The first row can use only c_1, so $\forall \alpha \in \mathbb{N}_0$ every cell that represents $\alpha \cdot c_1$ will have the entropy $\alpha \cdot e_1$. The rest of the cells are infeasible and their entropies are ∞.

Filling the Matrix: For each column $(b_1, \ldots, b_{c^4})$ and row i there are two possibilities:

1. The optimal solution does not use c_i at all in order to achieve $(b_1, \ldots, b_{c^4})$. In that case $T(i, < b_1, \ldots, b_{c^4} >) = T(i-1, < b_1, \ldots, b_{c^4} >)$
2. The optimal solution uses c_i at least one time. In that case $T(i, < b_1, \ldots, b_{c^4} >) = T(i, < b_1, \ldots, b_{c^4} > \setminus c_i) + e_i$

Therefore:

1. $T(i, < b_1, \ldots, b_{c^4} >) \leftarrow \min(T(i, < b_1, \ldots, b_{c^4} >), T(i-1, < b_1, \ldots, b_{c^4} >))$
2. $T(i, < b_1, \ldots, b_{c^4} > \cup\, \alpha \cdot c_i) \leftarrow \min(T(i, < b_1, \ldots, b_{c^4} > \cup c_i), T(i, < b_1, \ldots, b_{c^4} >) + e_i)$

Theorem 2. *The dynamic programming algorithm's running time is* $O((n \cdot \frac{\log n}{\epsilon})^{c^4-1} \cdot (n^2)^{c^4} / (n \cdot \frac{\log n}{\epsilon}))$

Proof: There are $m \cdot (n^2)^{c^4}$ cells in matrix M. The algorithm only handles columns representing multisets of size divisible by $n\frac{\log n}{\epsilon}$. The time to fill each cell is constant. Therefore, the total complexity is $O((n \cdot \frac{\log n}{\epsilon})^{c^4-1} \cdot (n^2)^{c^4} / (n \cdot \frac{\log n}{\epsilon}))$ □

Theorem 3. *Let S be a set and let s be the number of correct seams in the dynamic programming construction of S. Let s_{max} be the maximum number of correct seams for any square tiling of S. Then $s \geq (1 - \frac{\epsilon}{2 \log n}) s_{max}$.*

Proof: Let M be an optimal tiling. Let X be the number of correct vertical seams, and Y the number of correct horizontal seams in M. $s_{max} = X + Y$. (If M has no errors, then $s_{max} = 2n^2 - 2n$).

Without loss of generality we may assume that $X \geq Y$ (otherwise, we rotate all tiles by 90^o). Since the dynamic programming algorithm provides the optimum tiling within the strips of size $\frac{\log n}{\epsilon} \times n$, and the only errors may occur when "putting together" these strips, then it is clear that the number of correct vertical seams decided by our algorithm is no less than the number of vertical seams in the optimum tiling. We need to consider only Y – the number of correct horizontal seams, since our algorithm makes no effort to match the rows between the strips.

We start by identifying the total number of correct horizontal seams at the bottom of the $\frac{\log n}{\epsilon} \times n$ strips of M. Call that number Y_0. Next we consider the strips as moved down by an offset of 1, i.e., assume the first strip is of only one row, and the following strips are of size $\frac{\log n}{\epsilon} \times n$. The total number of correct horizontal seams at the bottom of the $\frac{\log n}{\epsilon} \times n$ strips (with offset 1) we call Y_1. In general, let Y_i the total number of correct horizontal seams at the bottom of the $\frac{\log n}{\epsilon} \times n$ strips with offset i (i.e, the first strip has only i rows, followed by $\frac{\log n}{\epsilon} \times n$ strips.) Formally, $\forall \;\; 0 \leq i \leq \frac{\log n}{\epsilon} - 1$, define $Y_i = \sum_{j=1}^{\frac{n \cdot \epsilon}{\log n}}$ (number of matches between row $j \cdot \frac{\log n}{\epsilon} + i$ and the row below it).

Let Y_{min} be such that $Y_{min} \leq Y_i \forall i,\ 0 \leq i \leq \frac{\log n}{\epsilon} - 1$. We will assume that the worst happened, and all the horizontal seams at the bottom of the $\frac{\log n}{\epsilon} \times n$ strips of the dynamic programming tiling of S are erroneous. However, this number of errors can not exceed Y_i, since the dynamic programming vertical tiling within the strips is *superior* to the optimal tiling. Therefore $s \geq X + Y - Y_{min}$.

However, because of averaging considerations, it is clear that $Y_{min} \leq Y/\frac{\log n}{\epsilon}$. Therefore

$s \geq s_{max} - Y_{min} \geq s_{max} - Y/\frac{\log n}{\epsilon}$. However, $s_{max} = X + Y \geq 2Y$, therefore $Y/\frac{\log n}{\epsilon} \leq s_{max}/\frac{2\log n}{\epsilon}$. Thus $s_{max} - Y/\frac{\log n}{\epsilon} \geq (1 - \frac{\epsilon}{2\log n})s_{max}$. □

5 Conclusions and Open Problems

The idea of using patches for indexing, presented by Amir and Parienty [2], is not viable if tiling can not be done efficiently. In this paper we showed a PTAS for the square tiling problem over fixed finite alphabets. An interesting open question is whether square tiling over an infinite alphabet is also approximable.

An intriguing direction is, perhaps, using rectangles, rather than squares for indexing, since we have shown that rectangle tiling is polynomially computable for "long and skinny" rectangles over a finite fixed alphabet. Indeed, for indexing purposes, the entire square will rarely be sought. Thus the results of this paper bring encouraging evidence to the proposal of utilizing patches for indexing.

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. Amir, A., Parienty, H.: Towards a theory of patches. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 254–265. Springer, Heidelberg (2009)
3. Bergeron, A., Corteel, S., Raffinot, M.: The algorithmic of gene teams. In: Guigó, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 464–476. Springer, Heidelberg (2002)
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. Eres, R., Landau, G.M., Parida, L.: Permutation pattern discovery in biosequences. Journal of Computational Biology 11(6), 1050–1060 (2004)
6. He, X., Goldwasser, M.H.: Identifying conserved gene clusters in the presence of orthologous groups. In: Proc. 8th Annual International Conferences on Research in Computational Molecular Biology (RECOMB), pp. 272–280 (2004)
7. Heber, S., Stoye, J.: Finding all common intervals of k permutations. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 207–218. Springer, Heidelberg (2001)
8. Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A.: Constraint Grammar. A Language Independent System for Parsing Unrestricted Text. Mouton de Gruyter (1995)
9. Lu, G.: Indexing and retrieval of audio: A survey. Multimedia Tools and Applications 15(3), 269–290 (2001)
10. Magazine, M.J., Chern, M.-S.: A note on approximation schemes for multidimensional knapsack problems. Mathematics of Operations Research 9(2), 244–247 (1984)

11. 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)
12. Schmidt, T., Stoye, J.: Quadratic time algorithms for finding common intervals in two and more sequences. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 347–358. Springer, Heidelberg (2004)
13. Srisuwannapa, C., Chamsethikul, P.: An exact algorithm for the unbounded knapsack problem with minimizing maximum processing time. J. of Computer Science 3(3), 138–143 (2007)
14. 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)
15. Uno, T., Yagiura, M.: Fast algorithms to enumerate all common intervals of two permutations. Algorithmica 26(2), 290–309 (2000)

On the Hardness of Counting and Sampling Center Strings

Christina Boucher[1] and Mohamed Omar[2]

[1] David R. Cheriton School of Computer Science,
University of Waterloo
cabouche@cs.uwaterloo.ca
[2] Department of Mathematics,
University of California, Davis
momar@math.ucdavis.edu

Abstract. Given a set S of n strings, each of length ℓ, and a non-negative value d, we define a *center string* as a string of length ℓ that has Hamming distance at most d from each string in S. The #Closest String problem aims to determine the number of unique center strings for a given set of strings S and input parameters n, ℓ, and d. We show #Closest String is impossible to solve exactly or even approximately in polynomial time, and that restricting #Closest String so that any one of the parameters n, ℓ, or d is fixed leads to an FPRAS. We show equivalent results for the problem of efficiently sampling center strings uniformly at random.

1 Introduction

Finding similar regions in multiple DNA, RNA, or protein sequences plays an important role in many applications, including universal PCR primer design [4,16,18,26], genetic probe design [16], antisense drug design [16,3], finding transcription factor binding sites in genomic data [27], determining an unbiased consensus of a protein family [1], and motif-recognition [16,24,25]. The Closest String problem formalizes these tasks and can be defined as follows: given a set of n strings S of length ℓ over the alphabet Σ and parameter d, the aim is determine if there exists a string s that has Hamming distance at most d from each string in S. We refer to s as the *center string* and let $d(x, y)$ be the Hamming distance between strings x and y.

The Closest String was first introduced and studied in the context bioinformatics by Lanctot *et al.* [16]. Frances and Litman *et al.* [11] showed the problem to be NP-complete, even in the special case when the alphabet is binary, implying there is unlikely to be a polynomial-time algorithm for this problem unless P = NP. Since its introduction, the investigation of efficient approximation algorithms and exact heuristics for the Closest String problem has been thoroughly considered [9,10,12,16,17,19,20].

S is *pairwise bounded* if the Hamming distance for each pair strings in S is at most $2d$. The Closest String problem reduces to separating pairwise bounded

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 127–134, 2010.

sets with a center string, and if so, finding one such string, from those that do not. A set of strings S with at least one center string is a *motif set*; S is a *decoy set* if it is pairwise bounded but does not have a center string. We note that a center string for a given set S is not necessarily unique.

A related, uninvestigated problem is determining the computational difficultly in finding the number of center strings for a set of strings. In many biological applications, including the ones listed above, it is useful to identify the all possible center strings, rather than only determining whether one exists. Further, an important relationship between the number of unique center strings for a given set of strings S and the computational difficulty of solving the decision version of #Closest String for an instance S has been shown. Specifically, empirical analysis demonstrates that for sufficiently large n, all motif sets are clustered together and are characterized as having one unique center string, which is a string of length ℓ containing the symbol that occurs most frequently at each position [2]. Imperative to the analytical explanation of this conjecture is the development of an algorithm to efficiently count the number of center strings for a given set of strings.

We give the formal description of this counting problem as follows:

#Closest String
Instance: Parameters n, ℓ and d and a set S of n, length ℓ strings from the alphabet Σ.
Output: The number of distinct strings taken from the alphabet Σ that have distance at most d from each string in S.

Countless sampling and counting problems have been studied, including the sampling and counting versions of the following problems: matchings in a graph [14], the graph-colouring [6,13,21], Hamiltonian path [7], independent set [8], and knapsack [5,22].

This paper focuses on the computational difficulty of counting and sampling center strings exactly or approximately. To our knowledge this is the first consideration of this problem but is motivated by problems addressing the analysis and use of biological data. We show #Closest String is #P-complete, implying it is in the complexity class of hard counting problems.

Given that this problem cannot be solved efficiently, we investigate if it can be reasonably approximated efficiently. Many #P-complete problems have a *fully-polynomial-time randomized approximation scheme (FPRAS)* which produces with high probability an approximation of arbitrarily small error in time that is polynomial with respect to both the size of the problem and accuracy. Jerrum *et al.* [15] showed that every #P-complete problem either has an FPRAS or is impossible to approximate. For sampling problems, the aim is to obtain a *fully polynomial almost uniform sampler (FPAUS)*, which outputs solutions that achieve an approximation to a given distribution of solutions. In absence of the existence of an FPRAS or FPAUS for a general counting or sampling problem, interest remains in showing an FPRAS or FPAUS exists for a restricted version

of the problem (*i.e.*, when one of the problem parameters is fixed). We prove that although there does not exist an FPRAS for the general #Closest String problem, the most natural restricted versions of #Closest String lead to the existence of an FPRAS. Similarly, we show that although there does not exist an FPAUS for sampling center strings uniformly at random (u.a.r.), restricting interest to this sampling problem where one of the parameters is fixed leads to an FPAUS.

2 Hardness Results

We show the #Closest String problem is #P-complete, and that there does not exist an FPRAS for #Closest String under reasonable complexity assumptions. A problem is #P-complete if and only if it is in #P and every problem in #P can be reduced to it by a polynomial-time counting reduction. To prove that #Closest String is #P-hard, it is sufficient to show that #3-SAT, a #P-complete problem, can be reduced to #Closest String. #3-SAT aims to determine for given a 3-CNF formula F, how many satisfying assignments exist for F. Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. A *literal* is either x_i or $\neg x_i$ for some i. We refer to a *3-clause* as a disjunction of three distinct literals, made of three different variables.

Proposition 1. *#Closest String is #P-complete.*

Proof. First, we present the reduction of a single 3-clause, and then extend it to a general 3-CNF formula. For a 3-clause ω over the variables in X we define the string $s = s(1), \ldots, s(2n)$ by:

$$s(2i-1)s(2i) = \begin{cases} 00 & \text{if } \omega \text{ contains the literal } \neg x_i, \\ 11 & \text{if } \omega \text{ contains the literal } x_i, \\ 01 & \text{otherwise} \end{cases}$$

Note that s is defined via its blocks and exactly three of them are repetitions. Let $\phi : X \to \{0,1\}^{2n}$ be the one-to-one mapping from X onto the set of all binary strings of length $2n$ as defined above (*i.e.* the transformation from ω to s). It is shown in [11] show that for any 3-clause over the variable $x_1, \ldots, x_n$, denoted as ω, and assignment $v \in \{0,1\}^n$, v satisfies ω if and only if $\phi(v)$ has distance at most n from s.

Let $F = \omega_1 \wedge \ldots \wedge \omega_t$ be a 3-CNF formula over the variables $x_1, \ldots, x_n$ then $v \in \{0,1\}^n$ is a satisfying assignment to F if and only if $\phi(v)$ has distance less than n from each of the strings in the set $\{s_1, \ldots, s_t\}$, where $s_i = \phi(\omega_i)$. Hence, the number of satisfying formulae to F is parsimonious to the number of center strings to the set $\{s_1, \ldots, s_t\}$. □

Randomization can be quite powerful in achieving an approximation scheme for several #P-complete problems. Jerrum *et al.* show that the problem of counting the number of simple cycles in a directed graph is not approximable by proving

that the existence of an almost uniform generator for this problem, implies the existence of a randomized polynomial time algorithm for determining the existence of a Hamiltonian cycle in a directed graph [15]. FPRAS is the subclass of #P counting problems whose answer, y, is approximable in the following sense: there exists a randomized algorithm that, with probability at least $1 - \delta$, approximates y to within an ϵ multiplicative factor in time polynomial in n (the input size), $1/\epsilon$, and $\log(1/\delta)$.

The results of Jerrum *et al.* [15] imply that every #P-complete problem exhibits an FPRAS or is not approximable. Given a #P-complete problem an important question to answer is if there exists a FPRAS for the problem – since the existence implies the approximability of the problem. Unfortunately, the existence of a FPRAS for #Closest String is unlikely.

Observation 1. *There is no FPRAS for* #Closest String*, unless NP = RP.*

Consider a problem π and I be an instance of the problem π, and let $\#(I)$ denote the number of solutions for I. an FPRAS can be used to distinguish between the case where $\#(I)$ is equal to zero and when $\#(I)$ is greater than zero; hence, providing a randomized polynomial time algorithm for the decision version of the problem π. Therefore, π must be contained in the class BPP. Since it is unlikely that BPP equal to NP, all NP-complete problems are believed not to contain an FPRAS [23, page 309]. Since Closest String problem is NP-complete [11], there exists no FPRAS for #Closest String, unless BPP = NP.

The notions of counting and sampling are closely related. Jerrum *et al.* established the equivalence between the existence of an FPRAS and an FPAUS; namely for self-reducible problems there exists an FPRAS if and only if there exists an FPAUS [15]. It follows that we have the following negative result concerning the approximability of sampling center strings

Observation 2. *There is no FPAUS for sampling center string uniformly at random, unless NP = RP.*

3 Counting and Sampling with Fixed Parameters

Observation 2 is evidence that determining the number of center strings is computationally difficult to approximate and therefore, to achieve progress on the existence of an FPRAS we consider the problem where one of the parameters is fixed. We first determine if the decision problem corresponding to the restricted version of #Closest String can be solved in polynomial-time, since otherwise we could show there does not exist an FPRAS by using similar argument to that for Observation 2. In order for the existence of an FPRAS to be possible for some restricted version of the #Closest String problem, the corresponding decision problem has to be *fixed parameter tractable (FPT)*, meaning there exists an algorithm that is exponential only in the size of a fixed parameter while polynomial in the remaining, unfixed parameters.

Closest String is trivially FPT when the parameter ℓ is fixed since the enumeration algorithm that tries all possible length ℓ strings is polynomial-time

when ℓ is fixed. Gramm *et al.* prove Closest String is FPT when n or d is fixed [12]. These results imply that restricting #Closest String such that at least one of ℓ, d or n is fixed leads to a problem that may have an FPRAS.

When ℓ is fixed the enumeration algorithm that attempts all $|\Sigma|^\ell$ strings is an FPRAS. The $O(d(|\Sigma|\ell e)^d)$ algorithm that determines which strings from the set of strings that have distance at most d from s_1 proves the existence of an FPRAS when d is fixed. These enumeration algorithms prove the existence of a FPAUS for the problem of sampling center strings u.a.r. when ℓ or d is fixed. Next we show there exists both an FPRAS and an FPAUS for the respective problems of counting and sampling center strings when the number of strings is fixed. In fact, we show a stronger result: that there exists an exact polynomial-time algorithm for counting and sampling center strings when n is fixed.

Proposition 2. *When the number of strings is fixed and Σ is the binary alphabet there exists a polynomial-time algorithm for* #Closest String *and for sampling center strings u.a.r.*

Proof. The goal is to give an integer linear programming (ILP) formulation such that the number of variables depends only on the value of n. Let $S = \{s_1, \ldots, s_n\}$ denote a set of n binary strings, each of length ℓ, and denote each string s_j as $s_j(1) \cdots s_j(\ell)$. Let $\mathcal{C}_S$ denote the set of center strings for the set S. Given a set of n strings of length ℓ, we can think of these strings as a $n \times \ell$ matrix. We refer to the *columns* of an instance of #Closest String as the the columns of a matrix. There are 2^n possible number of unique columns. Using the column types, we show how #Closest String restricted to the binary alphabet can be formulated as an ILP with 2^n variables.

Let $\boldsymbol{b} = [b_1, \ldots, b_n]^T$ correspond to one particular column type, and let $\mathcal{P}(\boldsymbol{b}) = \{i \mid (s_i(1), s_i(2), \ldots, s_i(n)) = \boldsymbol{b}\}$ (*i.e.* $\mathcal{P}(\boldsymbol{b})$ is the set of positions in S which are equal to $\boldsymbol{b}$). Therefore, $|\mathcal{P}(\boldsymbol{b})|$ is equal to the number of positions in S that are equal to $\boldsymbol{b}$.

For a string $u = u(1), \ldots, u(\ell)$, let $\rho(\boldsymbol{b})$ be the number of positions that are equal to $\boldsymbol{b}$ where u is equal to 0 (*i.e.* $j \in \mathcal{P}(\boldsymbol{b})$ and $u(j) = 0$). Hence, $|\mathcal{P}(\boldsymbol{b})| - \rho(\boldsymbol{b})$ is the number of positions that are equal to $\boldsymbol{b}$

Therefore, u has distance at most d from s_i if and only if

$$\sum_{\boldsymbol{b}} b_i \rho(\boldsymbol{b}) + (1 - b_i)(|\mathcal{P}(\boldsymbol{b})| - \rho(\boldsymbol{b})) \leq d.$$

Thus, $\mathcal{C}_S$ is nonempty if and only if there is a feasible integer solution to

$$\sum_{\boldsymbol{b}} (2b_i - 1)\rho(\boldsymbol{b}) + (1 - b_i)|\mathcal{P}(\boldsymbol{b})|) \leq d \qquad (1 \leq i \leq n) \tag{1}$$

$$0 \leq \rho(\boldsymbol{b}) \leq \mathcal{P}(\boldsymbol{b}) \tag{2}$$

for the variables $\rho(\boldsymbol{b})$.

Assuming there is a solution, we know the number of strings u corresponding to each such solution. It is exactly

$$\prod_{\boldsymbol{b}} \binom{\mathcal{P}(\boldsymbol{b})}{\rho(\boldsymbol{b})} \tag{3}$$

where the product is over all feasible values of the variables $\rho(\boldsymbol{b})$.

To sample the strings in $\mathcal{C}_S$

(i) generate random values of each of the numbers $\rho(\boldsymbol{b})$, for each $\boldsymbol{b}$, with the correct probabilities — proportional to the formula in (3)

(ii) sample exactly uniformly from the vectors corresponding to the given $\rho(\boldsymbol{b})$ by choosing subsets of given size uniformly at random.

This immediately shows that for fixed n there is a polynomial-time algorithm for perfect sampling, since one can run through all possible values of the set of variables $\rho(\boldsymbol{b})$ (there are at most ℓ values for each of these, hence at most ℓ^{2^n} values in total) and for each one, compute the value of (3). Then choose between one of these polynomially many values with the required probability, and then perform step (ii). □

A slight modification of the proof for the previous proposition leads to the following stronger result that eliminates the requirement that the alphabet is binary. See the Appendix for the details of the proof.

Proposition 3. *When the number of strings is fixed there exists a polynomial-time algorithm for* #Closest String *and for sampling center strings u.a.r.*

4 Conclusion

Counting and sampling from a specific distribution is a well-studied area in discrete mathematics and theoretical computer science that has been useful for the study of combinatorial problems. The problem of counting and sampling center strings has a natural application to several bioinformatic problems, including motif-recognition. We prove #Closest String is #P-complete and does not exist a FPRAS, the problem of sampling center strings u.a.r. does not have a FPAUS, and any natural restriction of these counting and sampling problems yields an FPRAS and FPAUS, respectively. This work suggests some open areas of study, including developing a more efficient sampling and counting algorithms, investigating the existence of a rapidly mixing chain for more restricted sampling problems, or proving hardness results that show the non-existence of a rapidly mixing chain when a single parameter is fixed.

Acknowledgements

The authors would like to thank Nick Wormald for his discussions and insights concerning the results presented in this paper. We gratefully acknowledge research support of the National Sciences and Engineering Research Council of Canada.

References

1. Ben-Dor, A., Lancia, G., Perone, J., Ravi, R.: Banishing bias from consensus strings. In: Hein, J., Apostolico, A. (eds.) CPM 1997. LNCS, vol. 1264, pp. 247–261. Springer, Heidelberg (1997)
2. Boucher, C., Brown, D.G.: Detecting motifs in a large data set: applying probabilistic insights to motif finding. In: Rajasekaran, S. (ed.) BICoB 2009. LNCS, vol. 5462, pp. 139–150. Springer, Heidelberg (2009)
3. Deng, X., Li, G., Li, Z., Ma, B., Wang, L.: Genetic design of drugs without side-effects. SIAM Journal on Computing 32(4), 1073–1090 (2003)
4. Dopazo, J., Rodríguez, A., Sáiz, J.C., Sobrino, F.: Design of primers for PCR amplification of highly variable genomes. Computer Applications in the Biosciences 9, 123–125 (1993)
5. Dyer, M.: Approximate counting by dynamic programming. In: Proc. of STOC, pp. 693–699 (2003)
6. Dyer, M., Frieze, A.: Randomly colouring graphs with lower bounds on girth and maximum degree. In: Proc. of FOCS, pp. 579–587 (2001)
7. Dyer, M., Frieze, A., Jerrum, M.: Approximately counting hamilton paths and cycles in dense graphs. SIAM Journal on Computing 27(5), 1262–1272 (1998)
8. Dyer, M., Frieze, A., Jerrum, M.: On counting independent sets in sparse graphs. SIAM Journal on Computing 31(5), 1527–1541 (2002)
9. Fellows, M.R., Gramm, J., Neidermeier, R.: On the parameterized intractability of CLOSEST SUBSTRING and related problems. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 262–273. Springer, Heidelberg (2002)
10. Fellows, M.R., Gramm, J., Niedermeier, R.: On the parameterized intractability of motif search problems. Combinatorica 26, 141–167 (2006)
11. Frances, M., Litman, A.: On covering problems of codes. Theoretical Computer Science 30(2), 113–119 (1997)
12. Gramm, J., Niedermeier, R., Rossmanith, P.: Fixed-parameter algorithms for closest string and related problems. Algorithmica 37(1), 25–42 (2003)
13. Hayes, T.P., Vigoda, E.: A non-markovian coupling for randomly sampling colorings. In: Proc. of FOCS, pp. 618–627 (2003)
14. Jerrum, M.R., Sinclair, A.: Approximating the permanent. SIAM Journal on Computing 18(6), 1149–1178 (1989)
15. Jerrum, M.R., Valiant, L.G., Vazirani, V.: Random generation of combinatorial structures from a uniform distribution. Theoretical Computer Science 43 (1986)
16. Lanctot, J.K., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. Information and Computation, 41–55 (2003)
17. Li, M., Ma, B., Wang, L.: Finding similar regions in many strings. Journal of Computer and System Sciences 65(1), 73–96 (2002)
18. Lucas, K., Busch, M., Össinger, S., Thompson, J.A.: An improved microcomputer program for finding gene- and gene family-specific oligonucleotides suitable as primers for polymerase chain reactions or as probes. Computer Applications in the Biosciences 7, 525–529 (1991)
19. Ma, B.: A polynomial time approximation scheme for the closest substring problem. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 99–107. Springer, Heidelberg (2000)
20. Ma, B., Sun, X.: More efficient algorithms for closest string and substring problems. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 396–409. Springer, Heidelberg (2008)

21. Molloy, M.: The glauber dynamics on colorings of a graph with high girth and maximum degree. In: Proc. of STOC, pp. 91–98 (2002)
22. Morris, B., Sinclair, A.: Random walks on truncated cubes and sampling 0-1 knapsack solutions. In: Proc. of FOCS, pp. 230–240 (1999)
23. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
24. Pavesi, G., Mauri, G., Pesole, G.: An algorithm for finding signals of unknown length in DNA sequences. Bioinformatics 17, S207–S214 (2001)
25. Pevzner, P., Sze, S.: Combinatorial approaches to finding subtle signals in DNA strings. In: Proc. of 8th ISMB, pp. 269–278 (2000)
26. Proutski, V., Holme, E.C.: Primer master: A new program for the design and analyiss of PCR primers. Computer Applications in the Biosciences 12, 253–255 (1996)
27. Tompa, M., Li, N., Bailey, T.L., Church, G.M., De Moor, B., Eskin, E., Favorov, A.V., Frith, M.C., Fu, Y., Kent, W.J., et al.: Assessing computational tools for the discovery of transcription factor binding sites. Nature Biotechnology 23(1), 137–144 (2005)

Appendix

Proposition 3. When the number of strings is fixed there exists a polynomial-time algorithm for #Closest String and for sampling center strings u.a.r.

Proof. The goal is to give an ILP formulation such that the number of variables depends only on the value of n. Let $S = \{s_1, \ldots, s_n\}$ denote a set of n strings from the alphabet Σ, each of length ℓ, and denote each string s_j as $s_j(1) \cdots s_j(\ell)$. Let $\mathcal{C}_S$ denote the set of center strings for the set S. Let α be a letter in Σ. The nth Bell number is the number of partitions of a set of size n. Without loss of generality, we assume that the first string is equal to α^ℓ, since any set of strings can be trivially converted to an equivalent set where this is true. Using the same terminology defined in the proof of Proposition 2, there exists at most $B_n \leq n!$ unique column types, where B_n is nth Bell number.

Let $\boldsymbol{b} = [b_1, \ldots, b_n]^T$ correspond to one particular column type, and let $\mathcal{P}(\boldsymbol{b}) = \{i \mid (s_i(1), s_i(2), \ldots, s_i(n)) = \boldsymbol{b}\}$. Let $|\mathcal{P}(\boldsymbol{b})|$ be equal to the number of positions in S that are equal to $\boldsymbol{b}$. For a string $u = u(1), \ldots, u(\ell)$, let $\rho(\boldsymbol{b}, \nu)$ be the number of positions that are equal to $\boldsymbol{b}$ where u is equal to ν and $\nu \in \Sigma$.

Hence, $\mathcal{C}_S$ is nonempty if and only if there is a feasible integer solution to

$$\sum_{\boldsymbol{b}} \sum_{\nu \in (\Sigma - \nu(\boldsymbol{b}, i))} \rho(\boldsymbol{b}, \nu) \leq d \qquad (1 \leq i \leq n) \tag{4}$$

$$0 \leq \rho(\boldsymbol{b}, \nu) \leq \mathcal{P}(\boldsymbol{b}) \tag{5}$$

for the variables $\rho(\boldsymbol{b}, \nu)$, where $\nu(\boldsymbol{b}, i)$ is symbol of string i at column $\boldsymbol{b}$. Sampling and counting the solutions to this ILP can be done equivalently to sampling and counting the solutions to the ILP given in the proof of Proposition 2. □

Counting and Verifying Maximal Palindromes

Tomohiro I[1], Shunsuke Inenaga[2], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University
[2] Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishiku, Fukuoka, 819–0395 Japan
tomohiro.i@i.kyushu-u.ac.jp,
inenaga@c.csce.kyushu-u.ac.jp,
{bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. A palindrome is a symmetric string that reads the same forward and backward. Let $Pals(w)$ denote the set of maximal palindromes of a string w in which each palindrome is represented by a pair (c, r), where c is the center and r is the radius of the palindrome. We say that two strings w and z are pal-distinct if $Pals(w) \neq Pals(z)$. Firstly, we describe the number of pal-distinct strings, and show that we can enumerate all pal-distinct strings in time linear in the output size, for alphabets of size at most 3. These results follow from a close relationship between maximal palindromes and parameterized matching. Secondly, we present a linear time algorithm which finds a string w such that $Pals(w)$ is identical to a given set of maximal palindromes.

1 Introduction

1.1 Palindromes in Strings

A palindrome is a symmetric string that reads the same forward and backward. Namely, a string w is a palindrome if $w = xax^R$ where x is a string, x^R is a reversal of x, and a is either a single character or the empty string. Studying palindromic structures in strings have gathered much attention in theoretical computer science and in its applications.

In word combinatorics, palindromic structures of interesting family of words have been extensively studied. For example, palindromic factors of Fibonacci words and Sturmian words were investigated in [11,12,19,26]. A concept called palindrome complexity of infinite words was introduced in [1] and its extension to finite words was proposed in [2]. Palindromic occurrences in ternary square-free words were studied in [10].

In algorithmics, several efficient algorithms to compute palindromes in a string have been proposed. Manacher [27] showed a linear-time algorithm to compute all prefix palindromes of an input string, which can immediately be extended to computing maximal palindromes of all positions of the string within the linear complexity. Another linear-time algorithm for prefix palindromes detection was proposed in the KMP pattern matching algorithm paper [24]. There exist efficient

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 135–146, 2010.

parallel algorithms to find all prefix palindromes or all maximal palindromes of a string [3,4,7]. A polynomial-time algorithm to compute all maximal palindromes from a given compressed string was proposed in [28].

In bioinformatics, some extended concepts of palindromes are known to be important in DNA and RNA sequence analysis [25]. An approximate palindrome, where the first half of the palindrome can be transformed into the reversal of the second half within a predefined edit distance, was introduced in [30]. Gusfield showed a linear-time algorithm to compute maximal palindromes with a fixed gap [20]. Kolpakov and Kucherov proposed linear-time solutions allowing more flexible gaps [25]. In [21] an efficient algorithm to compute all maximal approximate gapped palindromes was developed.

1.2 Our Contribution

It is natural and convenient to represent each maximal palindrome p of a string w by a pair (c, r) such that c is the center of p and r is the radius of p. This way the set of all maximal palindromes can be represented with $O(n)$ space, where n is the length of w. In what follows, we assume that the set of maximal palindromes of a string is represented in this way.

The contribution of this paper is twofold: Firstly, we show new properties of palindromes which are closely related to parameterized matching [5]. That is, if two strings are drawn from an alphabet of size at most 3, then they have the same set of maximal palindromes if and only if they parameterized match. Based on the above result and the results from [29], the number of distinct sets of palindromes for alphabets of size at most 3 can immediately be obtained. Besides, we show that there exists an efficient algorithm to compute a representative string for all distinct sets of maximal palindromes for alphabets of size at most 3.

Secondly, we study the problem of inferring a string from a given set of palindromic structures. Namely, given a set P of pairs (c, r), find a string whose maximal palindromes coincide with P. We propose a linear time solution to this problem, which outputs the lexicographically smallest string over a minimum alphabet.

1.3 Related Work

Inferring a string from other string data structures has been widely studied. An algorithm to find a string having a given border array was presented in [16], which runs in linear time for an unbounded alphabet. A simpler linear-time solution for the same problem for a bounded alphabet was shown in [14]. Linear-time and $O(n^{1.5})$-time inferring algorithms for parameterized versions of border arrays, on a binary alphabet and an unbounded alphabet, respectively, were recently proposed [22,23]. Linear-time inferring algorithms for suffix arrays [15,6], KMP failure tables [13,18], prefix tables [8], cover arrays [9], directed acyclic word graphs [6] and directed acyclic subsequence graphs [6] have been proposed, which provide us with further insight concerning the data structures.

Counting and enumerating some of the above-mentioned data structures have also been studied in the literature [29,31,22,23].

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. The empty substring ε of w is denoted by $w[i:i-1]$ for $1 \leq i \leq n$. For any string w, let w^R denote the reversed string of w, that is, $w^R = w[|w|] \cdots w[2]w[1]$.

A string w is called a *palindrome* if $w = w^R$. If $|w|$ is even, then w is called an *even palindrome*, that is, $w = xx^R$ for some $x \in \Sigma^+$. If $|w|$ is odd, then w is called an *odd palindrome*, that is, $w = xax^R$ for some $x \in \Sigma^*$ and $a \in \Sigma$. The *radius* of a palindrome w is $\frac{|w|}{2}$.

The *center* of a palindromic substring $w[i:j]$ of a string w is $\frac{i+j}{2}$. A palindromic substring $w[i:j]$ is called the *maximal palindrome* at the center $\frac{i+j}{2}$ if no other palindromes at the center $\frac{i+j}{2}$ have a larger radius than $w[i:j]$, i.e., if $w[i-1] \neq w[j+1]$, $i = 1$, or $j = |w|$. In particular, $w[1:j]$ is called a *prefix palindrome* of w, and $w[i:|w|]$ is called a *suffix palindrome* of w.

We denote by $(c,r)_w$ the maximal palindrome of a string w whose center is c and radius is r. We simply write (c,r) when the string w is clear from the context. The set of all maximal even and odd palindromes of a string w is denoted by $Pals(w)$. It is clear that for any string w $Pals(w)$ has exactly $2|w|+1$ elements. Let $SPals(w)$ denote the set of all suffix palindromes of w, that is, $SPals(w) = \{(c,r) \mid (c,r) \in Pals(w), c + r - 0.5 = n\}$.

For example, let $w = \texttt{abbacabbba}$. Then

$$\begin{aligned} Pals(w) = \{&(0.5,0),(1,0.5),(1.5,0),(2,0.5),(2.5,2),(3,0.5),(3.5,0),\\ &(4,0.5),(4.5,0),(5,3.5),(5.5,0),(6,0.5),(6.5,0),(7,0.5),\\ &(7.5,1),(8,2.5),(8.5,1),(9,0.5),(9.5,0),(10,0.5),(10.5,0)\} \text{ and}\\ SPals(w) = \{&(8,2.5),(10,0.5),(10.5,0)\}. \end{aligned}$$

3 Palindromes and Parameterized Matching

In this section we present new properties of palindromic structures in strings, with a tight relationship with parameterized matching which was originally introduced by Baker [5].

For any string w, let σ_w denote the number of distinct characters that appear in w. Any two strings w and z over the alphabet Σ of the same length are said to *parameterized match* (*p-match* in short) if there exists a renaming bijection $f : \Sigma \to \Sigma$ which transforms one string into the other [5], that is, $w = f(z[1])f(z[2]) \cdots f(z[|z|])$. For instance, strings $w = \texttt{abab}$ and $z = \texttt{baba}$ p-match, since w can be transformed to z by applying a renaming function $f : \Sigma \to \Sigma$ such that $f(\texttt{a}) = \texttt{b}$ and $f(\texttt{b}) = \texttt{a}$.

The following intuitive property indeed holds.

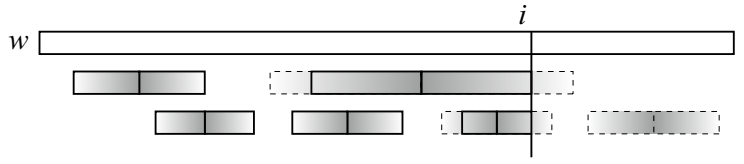


Fig. 1. Illustration for Observation 1

Lemma 1. *If two strings w and z p-match, then $Pals(w) = Pals(z)$.*

Proof. Assume for contrary that $Pals(w) \neq Pals(z)$. Then there exists at least one center c such that $(c, r) \in Pals(w)$, $(c, r') \in Pals(z)$ and $r \neq r'$. Assume w.l.o.g. that $r > r'$. Then it holds that $w[c - r' - 0.5] = w[c + r' + 0.5]$ and $z[c - r' - 0.5] \neq z[c + r' + 0.5]$. Let $\mathtt{a} = w[c - r' - 0.5] = w[c + r' + 0.5]$, $\mathtt{b} = z[c - r' - 0.5]$, and $\overline{\mathtt{b}} = z[c + r' + 0.5]$, where $\overline{\mathtt{b}}$ denotes any character in $\Sigma - \{\mathtt{b}\}$. Then clearly there exists no bijection on the alphabet Σ that can transform w into z, since $\mathtt{a}$ at position $c - r' - 0.5$ needs to be mapped to $\mathtt{b}$ but $\mathtt{a}$ at position $c + r' + 0.5$ needs to be mapped to $\overline{\mathtt{b}}$. This contradicts that w and z p-match. □

The reverse of Lemma 1 is also true if the strings are unary, binary or ternary. To show it, the following observation is useful.

Observation 1. *For any string w of length $n \geq 1$ and for any $i \leq n$,*

$$Pals(w[1:i]) = \{(c, i + 0.5 - c) \mid (c, r) \in Pals(w), c \leq i + 0.5, c + r - 0.5 > i\} \\ \cup \{(c, r) \mid (c, r) \in Pals(w), c + r - 0.5 \leq i\}.$$

(See also Fig. 1)

Lemma 2. *If $Pals(w) = Pals(z)$ and $\sigma_w = \sigma_z \leq 3$, then w and z p-match.*

Proof. **Unary case $\sigma_w = \sigma_z = 1$.** This case is trivial.

Binary case $\sigma_w = \sigma_z = 2$. We prove it by induction on the length i of the strings. When $i = 2$, clearly two strings w and z of length 2 p-match, if $Pals(w) = Pals(z)$ and $\sigma_w = \sigma_z = 2$.

Suppose that the lemma holds for $i = n - 1 \geq 2$. Let w and z be any strings of length n over Σ, such that $Pals(w) = Pals(z)$ and $\sigma_w = \sigma_z = 2$. By Observation 1 $Pals(w[1 : n - 1]) = Pals(z[1 : n - 1])$, and by the induction hypothesis $w[1 : n - 1]$ and $z[1 : n - 1]$ p-match. Let $f : \Sigma \to \Sigma$ be the bijection which transforms $w[1 : n - 1]$ into $z[1 : n - 1]$.

1. When $w[n-1] = w[n]$. If $z[n-1] \neq z[n]$, then $(n-0.5, 1) \in Pals(w)$ and $(n-0.5, 0) \in Pals(z)$. However, this contradicts that $Pals(w) = Pals(z)$, and hence $z[n-1] = z[n]$. Then $f(w[n]) = f(w[n-1]) = z[n-1] = z[n]$, and therefore w and z p-match.

2. When $w[n-1] \neq w[n]$. If $z[n-1] = z[n]$, then $(n-0.5, 0) \in \mathit{Pals}(w)$ and $(n-0.5, 1) \in \mathit{Pals}(z)$. However, this contradicts that $\mathit{Pals}(w) = \mathit{Pals}(z)$, and hence $z[n-1] \neq z[n]$. Let $f(w[n-1]) = z[n-1] = \mathtt{a}$ and $f(w[n]) = \mathtt{b}$. Since $z[n] \neq \mathtt{a}$, $z[n] = \mathtt{b}$. Hence $f(w[n]) = z[n]$ and therefore w and z p-match.

Ternary case $\sigma_w = \sigma_z = 3$**.** We prove it by induction on the length i of the strings. When $i = 3$, clearly two strings w and z of length 3 p-match, if $\mathit{Pals}(w) = \mathit{Pals}(z)$ and $\sigma_w = \sigma_z = 3$.

Suppose that the lemma holds for $i = n-1 \geq 3$. Let w and z be any strings of length n over Σ, such that $\mathit{Pals}(w) = \mathit{Pals}(z)$ and and $\sigma_w = \sigma_z = 3$. By similar arguments to the binary case, $w[1 : n-1]$ and $z[1 : n-1]$ p-match. Let $g : \Sigma \to \Sigma$ be the bijection which transforms $w[1 : n-1]$ into $z[1 : n-1]$.

1. When $w[n-1] = w[n]$. This case can be shown similarly to the binary case.
2. When $w[n-1] \neq w[n]$. By similar arguments to the binary case, we get $z[n-1] \neq z[n]$. Let m be the rightmost position of $w[1 : n-1]$ such that $w[m] \neq w[n-1]$. Let $g(w[n-1]) = z[n-1] = \mathtt{a}$ and $g(w[n]) = \mathtt{b}$.
   (a) When $w[m] = w[n]$. Since $w[m+1 : n-1]$ is unary, $w[m : n]$ is a maximal palindrome of w. Since $z[m : n]$ is a maximal palindrome of z, $z[n] = z[m] = g(w[m]) = \mathtt{b} = g(w[n])$. Hence w and z p-match.
   (b) When $w[m] \neq w[n]$. Let $g(w[m]) = \mathtt{c}$. Since $w[m+1 : n-1]$ is unary, $w[m+1 : n-1]$ is a maximal palindrome of w. Since $z[m+1 : n-1]$ is a maximal palindrome of z, $z[n] \neq z[m] = g(w[m]) = \mathtt{c}$. Additionally, since $z[n] \neq z[n-1] = g(w[n-1]) = \mathtt{a}$, $z[n] = \mathtt{b} = g(w[n])$. Consequently w and z p-match. □

The next proposition follows from Lemma 2.

Proposition 1. *For any string w with $\sigma_w \leq 2$, there exist no string z such that $\mathit{Pals}(w) = \mathit{Pals}(z)$ and $\sigma_z > \sigma_w$.*

It is interesting to see that a similar argument to Lemma 2 does *not* hold if strings contain 4 or more distinct characters. For instance, two strings $w = \mathtt{abbcdaa}$ and $z = \mathtt{accdbcc}$ have the same set of maximal palindromes and $\sigma_w = \sigma_z = 4$, but w and z do not p-match. Also, Proposition 1 does not hold if $\sigma_w \geq 3$. For instance, $w = \mathtt{abcabb}$ and $z = \mathtt{abcdaa}$ have the same set of maximal palindromes, while $\sigma_w = 3$ and $\sigma_z = 4$. It is easy to extend the above examples to infinite sequences of strings with an alphabet of size 4 or more.

Let us define equivalence relations $\equiv_{\mathrm{pal}}$ and $\equiv_{\mathrm{pm}}$ on Σ^* by

$$w \equiv_{\mathrm{pal}} z \iff \mathit{Pals}(w) = \mathit{Pals}(z)$$
$$w \equiv_{\mathrm{pm}} z \iff w \text{ and } z \text{ p-match.}$$

By Lemma 1 and Lemma 2, the two equivalence relations are equivalent for $|\Sigma| \leq 3$. Also, if $|\Sigma| \geq 4$, then $\equiv_{\mathrm{pm}}$ is a refinement of $\equiv_{\mathrm{pal}}$.

Denote by $[w]_{\mathrm{pal}}$ and $[w]_{\mathrm{pm}}$ the equivalence classes with respect to $\equiv_{\mathrm{pal}}$ and $\equiv_{\mathrm{pm}}$, respectively. Define the representative of equivalence class $[w]_{\mathrm{pal}}$ to be the

lexicographically smallest member of $[w]_{\mathrm{pal}}$, and call each representative a *pal-canonical* string. Moore et al. [29] counted the number of *p-canonical* strings, each of which is the representative (i.e., the lexicographically smallest member) of an equivalence class $[\cdot]_{\mathrm{pm}}$. They also presented an algorithm to enumerate all p-canonical strings. From Lemma 1, Lemma 2 and the results of [29], we immediately get the following theorems.

Theorem 1. *Let $p[k,n]$ be the number of distinct sets of maximal palindromes for strings of length n containing exactly k characters. Then $p[k,n] = S(n,k)$ for $1 \leq k \leq 3$ and $p[k,n] < S(n,k)$ for $k \geq 4$, where $S(n,k)$ is the Stirling number of the second kind.*

Theorem 2. *For every pair of integers $k \leq 3$ and $n \geq k$, all pal-canonical strings of length n consisting of exactly k characters can be computed in $O(p[k,n])$ time and space.*

Although the above theorems provide us with new insights and an efficient algorithm, yet they do not immediately help us solve the problem of inferring a string from a given set of maximal palindromes. In the next section, we will provide a linear-time algorithm to solve it.

4 Inferring a String from Maximal Palindromes

4.1 Problem

Let $\mathcal{N}$ be the set of non-negative integers, and let $\mathcal{Q} = \{i \mid i = \frac{j}{2}, j \in \mathcal{N}\}$. In this section we present a linear-time algorithm to solve the following problem.

Problem 1. Given a finite set $P \subset \mathcal{Q} \times \mathcal{Q}$, find a string w such that $P = Pals(w)$ if such exists.

Concerning Lemma 2 and Proposition 1 of the previous section, we try to find the *lexicographically smallest* string over a *minimum* alphabet in solving Problem 1.

4.2 Linear-Time Algorithm to Compute Maximal Palindromes from a String

Let us recall a linear-time algorithm to compute all maximal palindromes in a given string, which is an extended version of Manacher's algorithm that computes all prefix palindromes [27]. A pseudo-code of the algorithm is shown in Algorithm 1. The algorithm is based on the following lemma.

Lemma 3 ([27]). *For any string w, let $(c,r) \in Pals(w)$ and $(c-d, r_\ell) \in Pals(w)$ with some $0 < d \leq r$. Then $(c+d, r_r) \in Pals(w)$, where*

$$r_r = r_\ell \quad \text{if } r_\ell < r - d, \tag{1}$$

$$r_r \geq r - d \quad \text{if } r_\ell = r - d, \tag{2}$$

$$r_r = r - d \quad \text{if } r_\ell > r - d. \tag{3}$$

Algorithm 1. Manacher's algorithm to compute all maximal palindromes in a given string [27]

Input: string w of length n.
Output: compute all maximal palindromes in s.
`/* Let` $w[0]$ `and` $w[n+1]$ `be special symbols that do not match other symbols for convenience.` `*/`
add $(0.5, 0)$ and $(n + 0.5, 0)$ to P;
$i \leftarrow 2$; $c \leftarrow 1$; $r \leftarrow 0.5$;
while $c \leq n$ **do**
  $j \leftarrow 2c - i$; `/* Set` j `to be the mirrored position w.r.t.` c. `*/`
  while $w[i] = w[j]$ **do**
    $i++$; $j--$; $r++$;
  add (c, r) to P;
  $d \leftarrow 0.5$;
  while $d \leq r$ **do**
    let $(c - d, r_\ell) \in P$;
    if $r_\ell = r - d$ **then** break;
    $r_r \leftarrow \min\{r - d, r_\ell\}$;
    add $(c + d, r_r)$ to P;
    $d \leftarrow d + 0.5$;
  if $d > r$ **then** $i++$; $r \leftarrow 0.5$;
  else $r \leftarrow r - d$;
  $c \leftarrow c + d$; `/* Shift the value of` c `by` d. `*/`
return P;

Recall Observation 1. This observation suggests to compute maximal palindromes from left to right in the input string, and therefore Algorithm 1 computes the radius of each center in increasing order of the centers. Let c be the currently focused center, that is, for every center less than c, the corresponding radius has already been computed. Then we compute the radius for c, comparing leftward and rightward substrings of c until a mismatch occurs. Let $(c, r) \in Pals(w)$. The key of the algorithm is that, if Condition 1 or 3 of Lemma 3 holds for a center between c and $c+r$, then the radius of the center can be determined in constant time. If there exists a center $c + d$ with which Condition 2 holds, then we shift the currently focused center to the center $c+d$, and compute the radius for $c+d$. Since in this case the radius for $c+d$ is at least $r-d$, the overall time complexity of Algorithm 1 is linear in the length of the input string.

4.3 Our Algorithm to Compute a String from Maximal Palindromes

Now we consider Problem 1. Any $P \subset \mathcal{Q} \times \mathcal{Q}$ is said to be valid if there exists a string w such that $Pals(w) = P$, and is said to be invalid otherwise. For $P \subset \mathcal{Q} \times \mathcal{Q}$ to be valid, clearly P has to satisfy the following: For each $c = 0.5, 1, 1.5, \ldots, n, n+0.5$, there exists $(c, r) \in P$ with some $r \in \{0, 0.5, 1, 1.5, \ldots, k\}$, where $n = \lfloor |P|/2 \rfloor$

Algorithm 2. Algorithm to compute the lexicographically smallest string over a minimum alphabet which has a given set of maximal palindromes

Input: $P \subset \mathcal{Q} \times \mathcal{Q}$.
Output: The lexicographically smallest string w over a minimum alphabet with $Pals(w) = P$, if such exists.

```
1  w[1] ← a;
2  i ← 2; c ← 1; r̂ ← 0.5;
3  while c ≤ n do
4      j ← 2c − i;          /* Set j to be the mirrored position w.r.t. c. */
5      let (c, r) ∈ P;
6      if r < r̂ then return invalid;
7      while r̂ < r do
8          w[i] ← w[j];
9          i++; j−−; r̂++;
10         clear the list of forbidden characters;
11     add w[j] to the list of forbidden characters for w[i];   /* w[i] ≠ w[j]. */
12     d ← 0.5;
13     while d ≤ r do
14         let (c − d, r_ℓ), (c + d, r_r) ∈ P;
15         if r_ℓ = r − d then break;
16         if r_r ≠ min{r − d, r_ℓ} then return invalid;
17         d ← d + 0.5;
18     if d > r then
19         let x be the lexicographically smallest character not in the list of
           forbidden characters for w[i];
20         w[i] ← x;
21         i++; r̂ ← 0.5;
22         clear the list of forbidden characters;
23     else r̂ ← r − d;
24     c ← c + d;                          /* Shift the value of c by d. */
25 return w[1 : n];
```

and $k = \min\{c-0.5, n+0.5-c\}$. Hence in what follows we only consider as input a set P satisfying the above property.

Algorithm 2 shows our algorithm for Problem 1.

Theorem 3. *Given a valid set $P \subset \mathcal{Q} \times \mathcal{Q}$, Algorithm 2 computes the lexicographically smallest string w over a minimum alphabet such that $Pals(w) = P$, in linear time and space.*

Proof. Let $(c, r) \in P$. If Condition 1 or Condition 3 of Lemma 3 holds for the center c, then P is never rejected w.r.t. c in line 2 of Algorithm 2. Also, if Condition 2 of Lemma 3 holds for the center c, then P is never rejected w.r.t. c in line 2 of Algorithm 2. Therefore, if P is a valid set, then it is verified as valid by Algorithm 2.

In line 2 of Algorithm 2, $w[i]$ is set to $w[j]$ where $j = 2c - i$ is the mirrored position of i w.r.t. c. When i goes "outside" of the radius r of the maximal palindrome (c, r), then $w[j]$ is recorded as a forbidden character for $w[i]$ in line 2, that is, $w[i]$ cannot be equal to $w[j]$. Then in line 2, $w[i]$ is set to be the lexicographically smallest character that is not in the list of forbidden characters. Therefore, for a given valid set P Algorithm 2 computes the lexicographically smallest string w such that $P = Pals(w)$.

The key for time complexity analysis is how to choose the lexicographically smallest character in line 2. We use a bit vector F of length $n = \lfloor |P|/2 \rfloor$ where each $F[h]$ corresponds to the h-th lexicographically smallest character. We initialize F with $F[h] = 0$ for every $1 \leq h \leq n$. If a character $w[j]$ is forbidden for $w[i]$ in line 2 and if $w[j]$ is the h-th lexicographically smallest character, then we set $F[h] = 1$. After finding all forbidden characters for $w[i]$, we find the lexicographically smallest character for $w[i]$ by scanning F from left to right until reaching the smallest index k with $F[k] = 0$. After setting $w[i]$ to be the k-th lexicographically smallest character, we initialize every entry $F[h] = 1$ to $F[h] = 0$.

Since $\sigma_w \leq n$ for any string w of length n, the bit vector F is sufficiently large. For each position i, let $fc(i)$ and $lc(i)$ be the first (leftmost) center and the last (rightmost) center w.r.t. i, respectively. Namely, $w[i]$ is determined right after $lc(i)$ is obtained by shifting $fc(i)$ several times in line 2. Then, the number of forbidden characters for $w[i]$ does not exceed $2(lc(i) - fc(i))$. Hence the total number of forbidden characters for all i is bounded by $|P|$. Consequently we can maintain the bit vector F in a total of linear time and space. □

We remark that Algorithm 2 verifies some invalid sets to be valid. For instance, the following invalid set P is verified to be valid by Algorithm 2:

$$P = \{(0.5, 0), (1, 0.5), (1.5, 1), (2, 0.5), (2.5, 0), (3, 1.5), \\ (3.5, 0), (4, 0.5), (4.5, 1), (5, 0.5), (5.5, 0)\}.$$

Therefore, we firstly use Algorithm 2 as a filter. Consider the case where Algorithm 2 verifies an input set P to be valid. Let w be the output string of Algorithm 2 w.r.t. P. We then run Algorithm 1 over w to compute $Pals(w)$. Finally, we check whether $Pals(w) = P$ or not. Note that P is valid if and only if $Pals(w) = P$. Hence we obtain:

Theorem 4. *Problem 1 can be solved in linear time.*

Reducing Extra Space. Here we consider to reduce extra working space of Algorithm 2. The next lemma is useful to estimate it.

For any string w, let $SPC(w)$ be the set of centers of suffix palindromes of w, that is,

$$SPC(w) = \{c \mid (c, r) \in SPals(w)\}.$$

Lemma 4 ([17,28]). *For any string w of length n, $SPC(w)$ can be represented by $O(\log n)$ arithmetic progressions.*

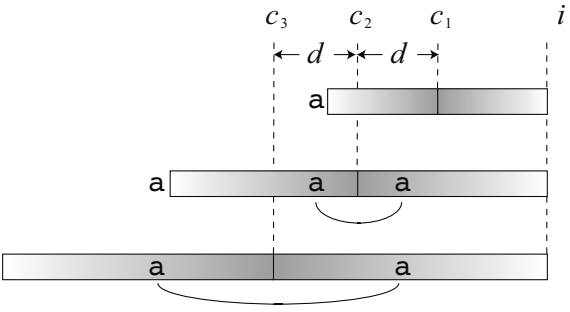


Fig. 2. Illustration for Proposition 2

For any w of length n and $1 \leq i \leq n$, let

$$MC(i) = \{w[2c - i] \mid (c, r) \in Pals(w), i = c + r + 0.5\}.$$

Proposition 2. *Let P be the set of maximal palindromes of some string of length n. Then, there exists a string w over an alphabet of size $O(\log n)$ such that $Pals(w) = P$.*

Proof. Take any three elements from $SPals(w[1 : i-1])$ whose centers belong to the same arithmetic progression, i.e., $(c_1, r_1), (c_2, r_2), (c_3, r_3) \in SPals(w[1 : i-1])$ such that $c_1 = c_2 + d = c_3 + 2d$ for some $d > 0$ (See also Fig. 2). By mirroring $w[2c_2 - i]$ w.r.t. c_3, we get $w[2c_2 - i] = w[2c_3 - (2c_2 - i)] = w[2(c_3 - c_2) + i] = w[i-2d]$. Similarly by mirroring $w[2c_1-i]$ w.r.t. c_2, $w[2c_1-i] = w[2c_2-(2c_1-i)] = w[2(c_2 - c_1) + i] = w[i - 2d]$. Then we get $w[2c_1 - i] = w[i - 2d] = w[2c_2 - i]$. By Lemma 4, $|MC(i)| = O(\log i)$. Since the number of forbidden characters for each position i is at most $|MC(i)|$, we conclude that $O(\log n)$ distinct characters are sufficient in total. □

Since Algorithm 2 always computes a string over a minimum alphabet, a bit vector of size $O(\log n)$ is actually enough for maintaining the forbidden characters for each position i of the output string.

5 Conclusions and Future Work

In this paper we studied the problem of counting the number of distinct sets of maximal palindromes, and that of finding a string from a given set of maximal palindromes. For the first problem, we showed the exact number for an alphabet of size at most 3. We also showed that there exists an algorithm that enumerates all pal-canonical strings for an alphabet of size at most 3, which runs in linear time in the output size. These results follow from the close relationship between maximal palindromes and parameterized pattern matching for alphabets of size at most 3. For the second problem, we presented a linear time algorithm that finds the lexicographically smallest string over a minimum alphabet.

Our future work includes the followings.

1. Counting the number of pal-canonical strings for an alphabet of arbitrary size.
2. Enumerating all distinct sets of maximal palindromes for an alphabet of arbitrary size.
3. Finding a string that has a given set of maximal palindromes and contains exactly k characters, where k is a predefined parameter.

References

1. Allouche, J.P., Baake, M., Cassaigne, J., Damanik, D.: Palindrome complexity. Theoretical Computer Science 292(1), 9–31 (2003)
2. Anisiu, M.C., Anisiu, V., Kása, Z.: Total palindrome complexity of finite words. Discrete Mathematics 310(1), 109–114 (2010)
3. Apostolico, A., Breslauer, D., Galil, Z.: Optimal parallel algorithms for periods, palindromes and squares. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 296–307. Springer, Heidelberg (1992)
4. Apostolico, A., Breslauer, D., Galil, Z.: Parallel detection of all palindromes in a string. Theoretical Computer Science 141, 163–173 (1995)
5. Baker, B.S.: Parameterized pattern matching: Algorithms and applications. Journal of Computer and System Sciences 52(1), 28–42 (1996)
6. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M.: Inferring strings from graphs and arrays. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 208–217. Springer, Heidelberg (2003)
7. Breslauer, D., Galil, Z.: Finding all periods and initial palindromes of a string in parallel. Algorithmica 14(4), 355–366 (1995)
8. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: Proc. STACS 2009, pp. 289–300 (2009)
9. Crochemore, M., Iliopoulos, C., Pissis, S., Tischler, G.: Cover array string reconstruction. In: Amir, A., Parida, L. (eds.) Combinatorial Pattern Matching. LNCS, vol. 6129, pp. 251–259. Springer, Heidelberg (2010)
10. Currie, J.D.: Palindrome positions in ternary square-free words. Theoretical Computer Science 396(1-3), 254–257 (2008)
11. Droubay, X.: Palindromes in the Fibonacci word. Information Processing Letters 55(4), 217–221 (1995)
12. Droubay, X., Pirillo, G.: Palindromes and Sturmian words. Theoretical Computer Science 223(1-2), 73–85 (1999)
13. Duval, J.P., Lecroq, T., Lefebvre, A.: Efficient validation and construction of border arrays and validation of string matching automata. RAIRO - Theoretical Informatics and Applications 43(2), 281–297 (2009)
14. Duval, J.P., Lecroq, T., Lefevre, A.: Border array on bounded alphabet. Journal of Automata, Languages and Combinatorics 10(1), 51–60 (2005)
15. Duval, J.P., Lefebvre, A.: Words over an ordered alphabet and suffix permutations. Theoretical Informatics and Applications 36, 249–259 (2002)
16. Franek, F., Gao, S., Lu, W., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a border array in linear time. J. Comb. Math. and Comb. Comp. 42, 223–236 (2002)
17. Gasieniec, L., Karpinski, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding. In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 392–403. Springer, Heidelberg (1996)

18. Gawrychowski, P., Jez, A., Jez, L.: Validating the Knuth-Morris-Pratt failure function, fast and online. In: Ablayev, F., Mayr, E.W. (eds.) Computer Science – Theory and Applications. LNCS, vol. 6072, pp. 132–143. Springer, Heidelberg (2010)
19. Glen, A.: Occurrences of palindromes in characteristic Sturmian words. Theoretical Computer Science 352(1), 31–46 (2006)
20. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York (1997)
21. Hsu, P.H., Chen, K.Y., Chao, K.M.: Finding all approximate gapped palindromes. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1084–1093. Springer, Heidelberg (2009)
22. I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting parameterized border arrays for a binary alphabet. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 422–433. Springer, Heidelberg (2009)
23. I, T., Inenaga, S., Bannai, H., Takeda, M.: Verifying a parameterized border array in $O(n^{1.5})$ time. In: Amir, A., Parida, L. (eds.) Combinatorial Pattern Matching. LNCS, vol. 6129, pp. 238–250. Springer, Heidelberg (2010)
24. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
25. Kolpakov, R., Kucherov, G.: Searching for gapped palindromes. Theoretical Computer Science 410(51), 5365–5373 (2009)
26. de Luca, A., Luca, A.D.: Palindromes in Sturmian words. In: De Felice, C., Restivo, A. (eds.) DLT 2005. LNCS, vol. 3572, pp. 199–208. Springer, Heidelberg (2005)
27. Manacher, G.: A new linear-time "On-Line" algorithm for finding the smallest initial palindrome of a string. Journal of the ACM 22(3), 346–351 (1975)
28. Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Efficient algorithms to compute compressed longest common substrings and compressed palindromes. Theoretical Computer Science 410(8–10), 900–913 (2009)
29. Moore, D., Smyth, W.F., Miller, D.: Counting distinct strings. Algorithmica 23(1), 1–13 (1999)
30. Porto, A.H.L., Barbosa, V.C.: Finding approximate palindromes in strings. Pattern Recognition 35(11), 2581–2591 (2002)
31. Schürmann, K.B., Stoye, J.: Counting suffix arrays and strings. Theoretical Computer Science 395(2-1), 220–234 (2008)

Identifying SNPs without a Reference Genome by Comparing Raw Reads

Pierre Peterlongo[1], Nicolas Schnel[1], Nadia Pisanti[2], Marie-France Sagot[3], and Vincent Lacroix[3]

[1] INRIA Rennes - Bretagne Atlantique, EPI Symbiose, Rennes, France
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] INRIA Rhône-Alpes, 38330 Montbonnot Saint-Martin, France and Université de Lyon, F-69000 Lyon, Université Lyon 1, CNRS, UMR5558, Laboratoire de Biométrie et Biologie Evolutive, F-69622 Villeurbanne, France

Abstract. Next generation sequencing (NGS) technologies are being applied to many fields of biology, notably to survey the polymorphism across individuals of a species. However, while single nucleotide polymorphisms (SNPs) are almost routinely identified in model organisms, the detection of SNPs in non model species remains very challenging due to the fact that almost all methods rely on the use of a reference genome. We address here the problem of identifying SNPs without a reference genome. For this, we propose an approach which compares two sets of raw reads. We show that a SNP corresponds to a recognisable pattern in the de Bruijn graph built from the reads, and we propose algorithms to identify these patterns, that we call *mouths*. We outline the potential of our method on real data. The method is tailored to short reads (typically Illumina), and works well even when the coverage is low where it reports few but highly confident SNPs. Our program, called kisSnp, can be downloaded here: `http://alcovna.genouest.org/kissnp/`.

1 Introduction

Biology in general, and genomics more particularly, witnessed a revolution in the middle 1970s with the development of rapid DNA sequencing techniques, notably the Sanger method which remained the standard approach for sequencing including whole genomes until the early years of the twenty first century. We have since then been witnessing a second revolution, various orders of magnitude bigger than the first, with the advent of the so-called "next generation sequencers" (NGS for short) which enable to obtain up to several hundred million bases in one single run at increasingly lower costs. These include (not exclusively) the 454 Life Sciences, SOLiD Applied Biosystems and Illumina technologies, each with its own characteristics in terms of read length and error rate. Such characteristics are however evolving extremely fast, faster indeed than the algorithms developed to handle the data such technologies produce.

This incredible acceleration has two implications that motivate the work presented in this paper: first it is now possible to obtain data for various individuals

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 147–158, 2010.

of a same species and thus to investigate the genetic differences among such individuals, and second, increasingly more often this will concern species for which we have no genome of reference, that is no genome already fully sequenced and assembled that could guide the investigation.

The genetic markers that will be of interest in this paper are so-called Single Nucleotide Polymorphisms (SNP for short). These correspond to a DNA sequence variation that occurs when a single nucleotide – A, T, C, or G – in a genome differs among members of a species or between paired chromosomes in an individual. There are two types of SNPs: substitutions or insertions/deletions. We focus here on the first type, that is on substitutions of single nucleotides.

Identifying SNPs in a population may have a wide range of applications that goes from assessing the polymorphism of the population, linking this polymorphism to phenotype information, or selecting SNPs as markers of subpopulations. However, while SNPs are almost routinely identified in model organisms, the detection of SNPs in non model species remains very challenging due to the fact that almost all methods to identify SNPs rely on the use of a reference genome.

Our objective is, given high-throughput read data for a pair of individuals, to identify a set of SNPs with good confidence, without having to perform an assembly of the reads with all the possible mistakes this entails, in a context where we do not have a reference sequence to help the identification.

We are aware of only two publications, dating both from 2010, that deal with the same problem [3,9]. Recognisably, the major difficulty one faces is due to the presence of errors in the reads, which may be mistaken for a SNP. Additionally, the presence of inexact repeats in the genomes of the studied individuals, may further harden the task. In this paper, we restrict to the case where there is only one genomic variant for each individual (we say that the individuals are homozygous). In this context, the issue of repeats is greatly reduced.

Ratan *et al.* [9] first filter the reads in order to remove the repetitive sequences, then create clusters of overlapping reads which they assemble using a short read assembler. The SNPs are finally identified in the micro-assembled regions using a combination of filters, based on the number of reads supporting each variant or the distance of the SNP w.r.t. the end of the contig.

Unlike Ratan *et al.*, we chose not to use an assembler, which we think can make undesired choices as to sequence variants to remove during the assembly. Indeed, the purpose of an assembler is not to identify SNPs but to propose one reference sequence compatible with the data. Similar to Canon *et al.* [3], we work with raw reads, but we go further than a statistical description of the reads and propose to locally reconstruct the de Bruijn graph in order to identify SNPs. The use of a de Bruijn graph in computational biology was introduced by Pevzner *et al.* in 2001 [8] and used since then as a first step by many short read assemblers.

The key point of our method is that a SNP corresponds to a recognisable pattern in the de Bruijn graph, which we call a *mouth*, each *lip* of the mouth representing an individual variant of a same genomic locus.

Our aim is to directly find the mouths that may be reliably associated to a SNP without making use of any preliminary filters that may eliminate repeats.

This is important because, although not explicitely stated, such filters seem to strongly rely on the assumption of an approximately uniform coverage of each sequence position by the reads in the available data. This assumption is usually not true. Moreover, the biases in read coverage may even vary across two sequencing experiments of the same genomic sample. This means that filters may remove sequences which in fact do not belong to repeats.

We thus present in this paper an algorithm which takes as input two sets of short reads (Illumina or AB/SOLiD) and outputs candidate SNPs (i.e. mouths in the de Bruijn graph), without performing any filtering nor using a short read assembler. This is what we call a comparative micro-assembly. This method is new as, as far as we know, no other treats data coming from distinct sequencing experiments. This approach presents the interest of taking advantage of differences in the data directly into the heart of the algorithm and not in a post-treatment step. SNPs are thus detected on raw read data instead of on pre-assembled sequences. We applied our algorithm on data simulated using MetaSim [10], where we show under which sets of parameters the method works best. We finally apply the method to real data for *Escherichia coli*, for which experimentally validated SNPs are available [2], which is very rare. We show that our method successfully identifies the previously known SNPs, but also predicts new SNPs missed by the conservative method used in the original publication [2].

2 Preliminaries

Sequence, k-mers, prefix, suffix. A *sequence* is composed by zero or more symbols from an alphabet Σ containing $|\Sigma|$ distinct characters. A sequence s of length n on Σ is denoted also by $s[0]s[1]\dots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. The length of s is denoted by $|s|$. Finally, we denote by $s[i,j]$ the *substring* $s[i]s[i+1]\dots s[j]$ of s. In this case, we say that the substring $s[i,j]$ occurs at position i in s. We call *k-mer* a substring of length k. If $s = uv$ for $u, v \in \Sigma^*$, we say that v is a *suffix* of s and that u is a *prefix* of s.

De Bruijn graph. Each node of a de Bruijn graph stores exactly one k-mer. An edge connects a node n_0 to a node n_1 if the suffix of length $k-1$ of the k-mer corresponding to node n_0 is equal to the prefix of length $k-1$ of the k-mer corresponding to node n_1.

A category of *de novo* read-assembly methods such as SOAPdenovo [7], Euler [8] and Velvet [11] (to mention a few) uses the de Bruijn graph as a fundamental data structure. In a few words, reads are first divided into overlapping k-mers, then the associated de Bruijn graph is created and finally Eulerian paths are found in the graph for reconstructing the initial genomic sequence, or fragments thereof that are as large as possible (contigs).

One of the main difficulties encountered by such methods comes from the sequencing errors that generate substitutions and insertions/deletions in the data that must then be assembled. Such errors lead to loops in the de Bruijn graph which may hinder the Eulerian path detection. A first step in such algorithms

consists thus in "cleaning the data" by removing suspicious reads and substituting suspect nucleotides. This cleaning step may be problematic when looking for SNPs as it may remove a significant part of them that will be mistakenly considered as sequencing errors.

3 Comparative Micro-assembly Model

Our method compares reads generated by two distinct sequencing experiments, and creates parts of the de Bruijn graph potentially linked to a SNP between these two experiments, thereby detecting such SNPs.

The main idea is that the de Bruijn graph of k-mers stemming from two sequences that contain a SNP presents a mouth shape as shown in Fig. 1. The algorithm described in Section 4 detects and constructs such graph shapes directly from the non-assembled k-mers coming from the sets of reads of two distinct sequencing experiments. It is important to notice that the algorithm does not reconstruct the full de Bruijn graph but focuses only on putative SNPs by building mouths.

Mouth model definition. In a de Bruijn graph of k-mers coming from reads of two sequencing experiments (reads A and reads B), a *mouth* is composed by:

- an upper path of k overlapping k-mers $\{a_0..a_{k-1}\}$ resulting from the reads of at least set A. This path is called the *upper lip* of the mouth;
- a lower path of k overlapping k-mers noted $\{b_0..b_{k-1}\}$ resulting from the reads of at least set B. The a_i's and the b_i's differ by one substitution. This path is called the *lower lip* of the mouth;
- a left (resp. right) node, noted c_{-1} (resp. c_k) that corresponds to a k-mer present in both sets A and B and is connected to both a_0 and b_0 (resp. a_{k-1} and b_{k-1}). These k-mers are called the *closing k-mers* of the mouth.

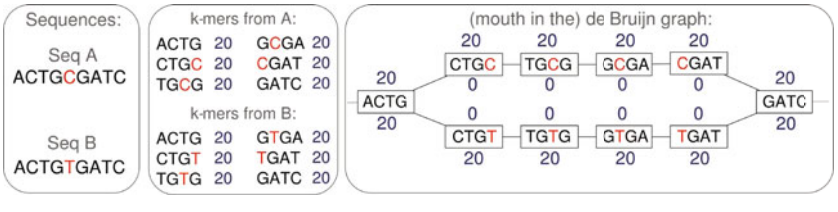


Fig. 1. A SNP between two genome fragments (Seq A and Seq B) generates a mouth shape (rightmost frame) in the de Bruijn graph of the k-mers (here $k = 4$) extracted from Seq A and Seq B. It is assumed in this example that the coverage is exactly 20 (each position of each sequence is covered by 20 reads, thus each position gives rise to 20 k-mers). In the rightmost frame, the number above (resp. below) the nodes indicates the number of occurrences in Seq A (resp. Seq B).

Taking into account the *k*-mer counts. Let us first consider the case where the sequencing is perfect: uniform coverage $\mathcal{C}$ and no sequencing errors nor repeats. In such case, all k-mers from A covering a SNP have $\mathcal{C}$ occurrences more than the same k-mers from B, and *vice-versa*. We considered this theoretical perfect coverage $\mathcal{C} = 20$ in the example of Fig. 1. In practice, the coverage is not uniform and the reads contain errors. The consequence is that the k-mer count difference between experiments will not be constant along the mouth. To account for this, we introduce a parameter called δ, which is meant to capture a deviation from the exact case. Below, we describe the mouth model with counts.

Mouth model integrating k-mer counts. For any k-mer ω, let $mult_A(\omega)$ (resp. $mult_B(\omega)$) be its number of occurrences in the set of reads A (resp. B). We define $diff(\omega) = mult_A(\omega) - mult_B(\omega)$. The SNP mouth model integrates the k-mer number of occurrences as follows. A special k-mer a_{op} called the *opening k-mer* is chosen as reference (see Section 4). If a_{op} is in the upper (resp. lower) lip, for any k-mer a_i contained in a node of the upper (resp. lower) lip, we have $diff(a_i) = diff(a_{op}) \pm \delta$ while for any k-mer b_i contained in a node of the lower (resp. upper) lip, we have $diff(b_i) = -diff(a_{op}) \pm \delta$. The left and right closing k-mers c_{-1} and c_k have no counting properties.

4 Algorithm kisSnp for Mouth Detection

Algorithm outline. The algorithm kisSnp takes as input two sets of reads (A and B) coming from two distinct sequencing experiments. The output is a set of pairs of micro-assembled sequences, each of length $2k-1$, differing by exactly one substitution located at the central position. Those correspond to putative SNPs detected thanks to the mouth model. The algorithm is divided into three main steps:

- For each set A and B, extract the k-mers and their reverse complement and store them in a tree together with their number of occurrences.
- Create the mouths (detailed in Section 4.1):
  - For each possible opening k-mer a_{op}, detect all possible opposite opening k-mers b_{op} distant by one substitution from a_{op} and fulfilling the counting model.
  - For each pair (a_{op}, b_{op}), construct the mouth by extending the k-mers to the right and to the left with coherent k-mers (*i.e.* overlapping on $k-1$ characters and fulfilling the counting model).
  - Stop the right and left extensions once the mouth is closed or no extension can be found.
- Check that the found mouths are coherent with the reads (detailed in Section 4.3).

4.1 Creating the Mouths

Selection of the k-mers opening the mouth. The opening k-mer a_{op} is selected such that $max\,(mult_A(a_{op}), mult_B(a_{op})) < max\,(mult_A(a_i), mult_B(a_i))$ for any k-mer $a_i \neq a_{op}$ and $diff(a_{op}) \neq 0$. In other words, a_{op} is the k-mer having the smallest number of occurrences in either set A or B (possibly zero occurrence in one of the two sets), and is such that it occurs more in a set than in the other, possibly due to a SNP. The rationale for choosing the k-mer with the smallest count is to avoid choosing a k-mer involved in a repeat for opening the mouth. The opposite opening k-mer b_{op} is selected such that a_{op} and b_{op} are distant by exactly one substitution and $diff(b_{op}) = -diff(a_{op}) \pm \delta$. Notice that the substitution position between the k-mers a_{op} and b_{op} may be anywhere. We denote by p this substitution position ($p \in [0, k-1]$). It is worth noticing that, for a given opening k-mer a_{op}, several (at most $(|\Sigma| - 1)k$) distinct k-mers b_{op} may satisfy these conditions. They are all iteratively tested as mouth openers.

Extending and closing the mouth. Once a pair of opening k-mers (a_{op}, b_{op}) is selected, a recursive procedure extends them to the right and left with other k-mers fulfilling the following conditions (also shown in Fig. 2). The k-mers a_{i+1} and b_{i+1} may extend a_i and b_i iff:

- $p \geq 0$ (the closing k-mer c_k has not yet been reached), and
- $a_i[1, k-1] = a_{i+1}[0, k-2]$ and $b_i[1, k-1] = b_{i+1}[0, k-2]$ (the new k-mers overlap on $k-1$ characters with their predecessors), and
- $a_{i+1}[k-1] = b_{i+1}[k-1]$ (the extension is done with a same character), and
- $diff(a_{i+1}) = diff(a_{op}) \pm \delta$ and $diff(b_{i+1}) = -diff(a_{op}) \pm \delta$ (the counting model is fulfilled).

Similar conditions apply to a_{i-1} and b_{i-1} for extending a_i and b_i on the left.

The two lips of a mouth have to be closed. A mouth can be right-closed (resp. left-closed) if there exists a k-mer c_k (resp. c_{-1}) whose prefix (resp. suffix) of length $k-1$ is equal to the suffix (resp. prefix) of length $k-1$ of a_{k-1} (resp. a_0), by definition itself equal to the suffix (resp. prefix) of length $k-1$ of b_{k-1} (resp. b_0). Once the mouth is right- and left- closed, the procedure stops.

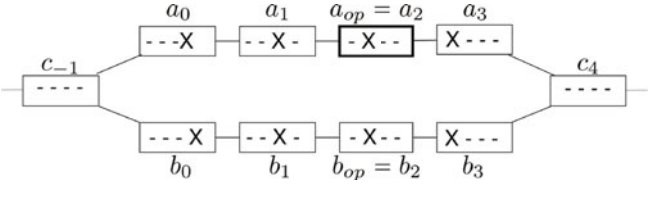


Fig. 2. A mouth with $k = 4$. The symbol $'-'$ stands for a match between two k-mers positions while the symbol $'X'$ stands for a mismatch. The k-mer a_2 is the opening k-mer and the k-mer b_2 is the opposite opening k-mer. Here, $p = 1$ as $a_{op}[1] \neq b_{op}[1]$.

Disabling the use of a same opening k-mer more than once Once a k-mer a_{op} was used for opening a mouth, it is flagged and never used anymore either as an opening or as an extending k-mer. The underlying idea is to avoid detecting twice the same mouth. Moreover, we can safely discard this k-mer because, since it was the one with smallest count, it should not belong to another mouth.

4.2 Complexity

The time complexity can then be divided into two main parts as follows.

- **Indexation:** if N is the total number of distinct k-mers, indexing them into a tree can be done in $O(N \log N)$ time using heap sort. This then provides access in time $O(k)$ to any k-mer information.
- **Mouth creation:** Given one opening k-mer a_{op}, at worst $k \times (|\Sigma| - 1)$ k-mers b_{op} may fulfill the opening conditions. Any k-mer may be opening. Thus at worst, $O(N \times k \times |\Sigma|)$ mouth opening pairs must be tried. Given an opening pair of k-mers a_{op} and b_{op}, in the worst case, for each of the $k-1$ steps of the extension, $|\Sigma|$ k-mers muts be tested. Access to a k-mer information being in time $O(k)$, the time complexity for one mouth extension is thus $O(k|\Sigma|^k)$. Thus, at worst, the time complexity for mouth finding is $O\left(N \times k^2 \times |\Sigma|^{2k}\right)$.

Each k-mer being stored in $O(k)$, the space complexity is $O(Nk)$.

4.3 Checking for Read Coherence

Two k-mers are linked in the de Bruijn graph if they overlap over k-1 characters, without checking whether the created k+1-mer indeed exists in the set of reads. This may lead to false-positive results. In order to remove those k-mers, in a post-treatment step, we check for *read-coherency* of the identified mouths. The upper (resp. lower) lip of a mouth is said to be read-coherent if it is 100% covered by reads from set A (resp. B) and, moreover, if in the upper (resp. lower) lip the SNP position is covered by at least two distinct reads from set A (resp. B). We keep only the mouths for which both lips are read-coherent. The rationale for restricting to mouths covered by at least 2 reads is to minimize the number of sequencing errors that are mistaken for SNPs. Indeed, it is unlikely that a sequencing error occurs at the same nucleotide for 2 distinct reads, as shown in [9].

5 Applications to Simulated Read Data

We developed the algorithm in a program called kisSnp, coded in Java, that was used for testing our approach. kisSnp is available for download at: `http://alcovna.genouest.org/kissnp/`.

To test our approach on controlled datasets, we applied the following procedure. Given an input sequence s_{ref}, we generated a sequence s_{snp} such that s_{ref}

and s_{snp} differ by $\left\lfloor \frac{|s_{ref}|}{1000} \right\rfloor$ substitutions, each considered as a SNP. The average distance between two virtual SNPs is then 1000 nucleotides, in agreement with [4]. The substitutions are randomly distributed over s_{snp}, and each substitution is introduced following the transition/transversion probabilistic model [5].

The sequence s_{ref} and s_{snp} are then virtually sequenced into a set of reads r_{ref} and r_{snp} using the MetaSim [10] program. Among other parameters, MetaSim enables to tune the sequencing errors model as well as the average coverage. In all our experiments, we generated reads of length 62, in agreement with the Illumina technology. kisSnp is then tested using sets r_{ref} and r_{snp}.

5.1 Finding the SNPs

Human chrX portion. We extensively tested the parameters of kisSnp on a small portion of the human chromosome X of length 137897 *bp*, corresponding to a kinesin family member 4A (KIF4A). We applied MetaSim to the pair s_{ref}, s_{snp} distant by 137 simulated SNPs with i) no sequencing errors (Fig. 3(a)) and ii) errors following an empirical Illumina error model (Jean Marc Aury, Genoscope, personal communication – Fig. 3(b) and (c)).

One may start by observing that the quality of the results is relatively robust to the choice of parameters. With a good coverage ($> 4x$), large distinct sets of parameters thus enable to find almost all SNPs with no false positives. The main lessons learnt about such parameters from these results are the following:

- The δ value has not a strong influence for $\delta \geq 20$ (Fig. 3(b) vs Fig. 3(c)). However, for smaller values of δ (data not shown), the specificity decreases rapidly due to sequencing errors and a non uniform coverage.
- As expected, for small values of k (≤ 20), false positives are found. For larger values of k (say, ≥ 30), less SNPs are found as more k-mers involve sequencing errors and/or more positions are not covered by any k-mer.

Concerning the data, coverage is of major importance as a small coverage leads to lower sensitivity (in each case, the lower the coverage, the less sensitive is kisSnp). Illumina sequencing errors have a small influence on the results (Fig. 3(a) vs Fig. 3(b)). One important message is that, using $k = 25$ and $\delta \geq 20$, all experiments obtained 100% specificity (no false positives) for various values of sensitivity, the latter depending in particular on the coverage.

Neisseria meningitidis strain MC58. One main limitation of our mouth model could be the presence of repeats in the genomes considered. To measure the effect of repeats on the performance of kisSnp, we performed experiments on the bacterium *Neisseria meningitidis* (strain *MC*58) of length 2.27 Mbp. The size of the repeated elements in this genome range from 10 bases to more than 2000 bases, and their number may reach more than 200 copies. We performed tests on the original MC58 sequence introducing 2272 SNPs and simulating reads using MetaSim with an Illumina error model. Moreover, we performed exactly the same tests on a randomised sequence obtained from the MC58 sequence (same length and nucleotide frequencies, distribution of nucleotides following a

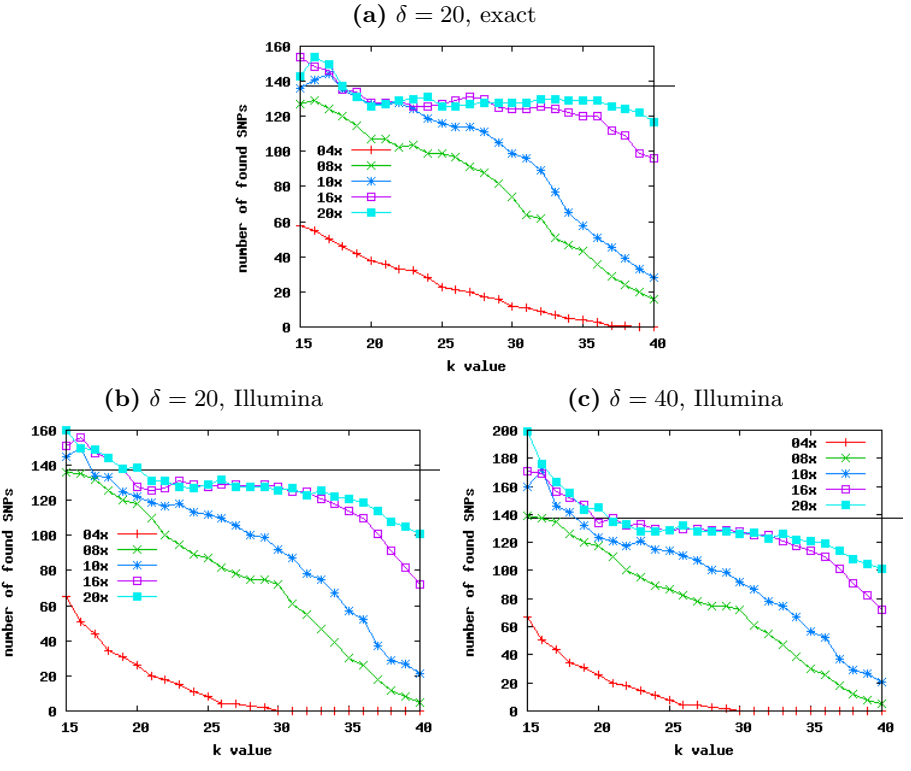


Fig. 3. Number of SNPs (read-coherent mouths) found by kisSnp. Fragment of the human chromosome X, 137 SNPs to find (symbolised by the horizontal line). The "$n\times$" values indicate the coverage used while simulating the reads.

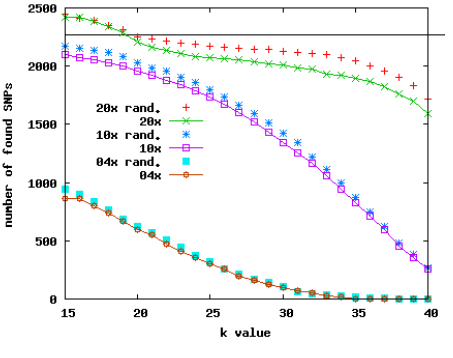


Fig. 4. Results on MC58 and the randomised MC58 sets while looking for 2272 SNPs (horizontal line) with $\delta = 20$. The "$n\times$" values indicate the coverage used while simulating the reads, "*rand.*" stands for results on the randomised MC58 set.

Bernoulli model). Results for both MC58 and the randomised MC58 are presented in Fig. 4.

One may observe that even on a difficult dataset like MC58, a large part of the SNPs are identified (26%, 86% and 97% respectively with coverage 4x, 10x, and 20x with $k = 20$). Another important remark is that the difference between the results obtained on MC58 and the randomised MC58 is small, showing that the algorithm is robust to repeats.

5.2 Execution Time

The kisSnp program, coded in Java, is a prototype and is not yet time optimised. The performance results below enable only to give an idea of the evolution of the running time with different parameters. All tests were done on a DELL laptop, quad-core 2.67GHz with 4Gb memory running under Fedora Core 12.

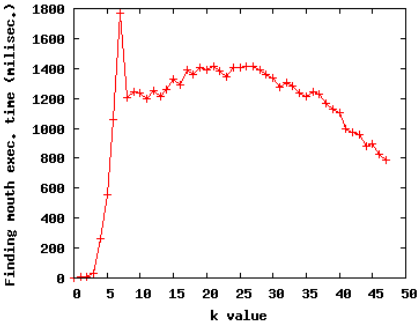


Fig. 5. Influence of k on the execution time. Data: set KIF4A (described Section 5.1), $\delta = 30$.

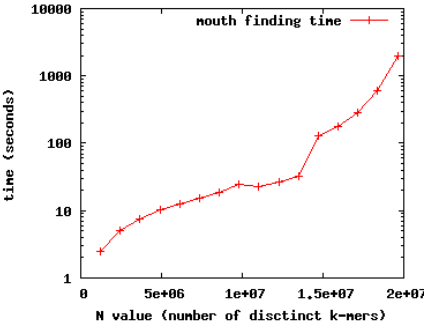


Fig. 6. Execution time with respect to the number N of distinct k-mers of length $k = 25$, with $\delta = 20$, Bernoulli random sequences

We started by testing on the human KIF4A dataset (simulating reads with an Illumina error model), the influence of the δ parameter on the execution time. We observed (data not shown) that this has no influence whatsoever for $\delta \geq 20$.

On the same dataset, we fixed $\delta = 30$ and checked the execution time for values of k varying from 0 to 40. The results, presented in Fig. 5, show that two phases may be distinguished. For k from 0 to 6, we observe an exponential time growth that is in agreement with the theoretical complexity. During this phase, the worst-case behaviour is reached, each mouth extension tests $|\Sigma|^k$ lips. The second phase is observed for bigger values of k, when a large number of possible k-mers are no longer present in the data, thus limiting the number of tested lips extensions. This greatly reduces the execution time, which starts decreasing for $k \geq 25$ as less and less mouths are successfully created.

The execution time also highly depends on the number N of distinct k-mers we have to deal with. We thus performed experiments on random sequences of growing size. The results are presented in Fig. 6. The execution time grows linearly

with N while N remains below a threshold of around 15 million reads. Above this threshold, one observes an exponential growth that could be explained by the fact that the kisSnp prototype uses a hash table instead of a tree for storing and accessing the k-mers information. With a large number of k-mers, the hash table load factor becomes higher than 0.75 increasing the lookup cost, because of an important number of collisions.

6 Applications to Real Read Data

To test our approach on a real dataset now, we used raw reads from the *Escherichia coli Long-term Experimental Evolution Project* [1] whose purpose was to grow *Escherichia coli* during more than 20 years, conserving a sample each 500 generations. The SNPs found over these generations are listed in the Barrick *et. al* paper [2]. An Illumina 1G platform was used for sequencing the samples with reads of length 36 and a high coverage ($50x$). We focused our attention on the raw reads from the first generation sample, and those from the 20.000^{th} generation sample. The existence of experimentally validated SNPs is very rare which is the main reason that led us to work on this dataset, for which true positives are known.

Using a custom-made computational pipeline called BRESEQ, Barrick *et. al* identified 28 SNPs by mapping these two generations of reads against the reference genome CP000819. These 28 SNPs were then experimentally validated.

We used kisSnp for comparing these two sets of reads, forgetting the reference genome. Using as parameters $k = 26$ and $\delta = 20$, 88 SNPs were found. Of these, 27 of the 28 SNPs found by Barrick *et. al* were also identified by us, giving a sensitivity of 96%. Our kisSnp method missed one SNP, located position 430835.

To evaluate the potential interest of the remaining 61 SNPs we identified, we mapped them against the reference genome using Maq [6]. Among the 61, 43 correspond to a SNP structure not detected in the Barrick *et. al* project: the two lips map at the same position with one substitution in the 20.000^{th} generation sequence. The remaining 18 correspond to suspicious SNPs. Indeed, the two lips do not map to the same position in the genome. Without experimental validation, one however cannot conclude on those 61 detected putative SNPs.

The results obtained with kisSnp are very good on this real dataset, as without a reference genome it was able to find back 27 of the 28 experimentally verified SNPs, and 41 additional ones that correspond to real SNP structures.

7 Conclusion and Future Work

We proposed an algorithm for comparing the raw outputs of short reads experiments, typically Illumina ones, with the purpose of finding SNPs between individuals of a same species. This is of particular interest for quickly designing genomic tags without waiting and/or paying for a full genome assembly.

Preliminary results on both simulated and real experimental data are particularly promising. In both cases, kisSnp identifies the SNPs, while not being too

sensitive to the parameters used. On a real dataset, kisSnp enabled to find 96% of the SNPs initially detected by mapping to the reference genome. In addition, we propose new SNPs, which could be tested experimentally.

There is clearly room for improvement. For now, the method does not handle heterozygous SNPs, does not take sequencing qualities into account and is limited to pairwise comparison while sets of more than two individuals may be compared. More generally, the three challenges of SNP identification are that the reads contain errors, the genome contains repeats and the read coverage is not uniform. This last item is usually disregarded whereas we notice that it has a significant impact on the results. We expect that several algorithms in the area of NGS bioinformatics could be improved by taking this observation into account.

Acknowledgments

We are grateful to Jean Marc Aury, who provided the empirical Illumina error model, to Matteo Brilli who pointed out to us the *E.coli* experiment, to the GenOuest platform who supported extensive computations and to the INRIA ARC Alcovna for financial support.

References

1. E. coli long-term experimental evolution project site, http://myxo.css.msu.edu/ecoli/
2. Barrick, J.E., Yu, D.S., Jeong, H., Oh, T.K., Schneider, D., Lenski, R.E., Kim, J.F.: Genome evolution and adaptation in a long-term experiment with escherichia coli. Nature 461, 1243–1247 (2009)
3. Cannon, C., Kua, C.-S., Zhang, D., Harting, J.: Assembly free comparative genomics of short-read sequence data discovers the needles in the haystack. Molecular Ecology 19(Suppl. 1) ,147–161 (2010)
4. Cooper, D.N., Smith, B.A., Cooke, H.J., Niemann, S., Schmidtke, J.: An estimate of unique DNA sequence heterozygosity in the human genome. Hum. Genet. 69, 201–205 (1985)
5. Kimura, M.: A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. J. Mol. Evol. 16, 111–120 (1980)
6. Li, H., Ruan, J., Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Research 18(11), 1851–1858 (2008)
7. Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., Wang, J.: De novo assembly of human genomes with massively parallel short read sequencing. Genome Res 20(2), 265–272 (2010)
8. Pevzner, P., Tang, H., Waterman, M.: An Eulerian path approach to DNA fragment assembly. Proc. Natl. Acad. Sci. 98, 9748–9753 (2001)
9. Ratan, A., Zhang, Y., Hayes, V., Schuster, S., Miller, W.: Calling SNPs without a reference genome. BMC Bioinformatics 11, 130 (2010)
10. Richter, D., Ott, F., Auch, A., Schmid, R., Huson, D.: MetaSim – A Sequencing Simulator for Genomics and Metagenomics. PLoS ONE 3(10), e3373 (2008)
11. Zerbino, D., Birney, E.: Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. 18(5), 821–829 (2008)

Dynamic Z-Fast Tries

Djamal Belazzougui[1], Paolo Boldi[2], and Sebastiano Vigna[2]

[1] Université Paris Diderot—Paris 7, France
[2] Università degli Studi di Milano, Italy

Abstract. We describe a dynamic version of the *z-fast trie*, a new data structure inspired by the research started by the van Emde Boas trees [12] and followed by the development of y-fast tries [13]. The dynamic z-fast trie is a very simple, uniform data structure: given a set S of n variable-length strings, it is formed by a standard compacted trie on S (with two additional pointers per node), endowed with a dictionary of size $n-1$. With this simple setup, the dynamic z-fast trie provides predecessors/successors in time $O(\log \max\{|x|, |x^+|, |x^-|\})$ ($x^\pm$ is the successor/predecessor of x in S) for strings of length linear in the machine-word size w. Prefix queries are answered in time $O(\log |x| + k)$, and range queries in time $O(\log \max\{|x|, |y|, |x^-|, |y^+|\} + k)$, where k is the number of elements in the output and x (and y) represent the input of the prefix (range) queries. Updates are performed within the same bounds in expectation (or with high probability using an appropriate dictionary). We then show a simple modification that makes it possible to handle strings of length up to 2^w; in this case, predecessor/successor queries and updates are supported in $O(|x|/w + \log \max\{|x|, |x^+|, |x^-|\})$ time, (and $O(|x|/B + \log \max\{|x|, |x^+|, |x^-|\})$ I/Os in the cache-oblivious model) with high probability. The space occupied by a dynamic z-fast trie, beside that necessary to store S, is just of $12n$ pointers, n integers and, in the "long string" case, $O(n)$ signatures of $O(w)$ bits each.

1 Introduction

Data structures providing successor and predecessor primitives on strings are divided into *comparison-based* and *digital* structures. The former assume to be able to compare strings in constant time; the latter rely on the actual (binary) digits forming the strings to do their job. The comparison-based data structure of choice is the balanced binary tree, whereas digital structures have been object of a growing interest in the last years starting from the introduction of van Emde Boas trees [12] and fusion trees [5], which provide asymptotically faster alternatives.

In this paper, we revisit the classical dynamic predecessor problem on a set S of size n from a universe of strings of length ℓ. The static version of this problem has been completely solved, with matching upper and lower bounds [11]. If only linear space is available, in a significant number of cases variants of the van Emde Boas tree [12] that answer in time $O(\log \ell)$ are optimal even in the static

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 159–172, 2010.

setting using randomisation. Among such variants we quote Willard's *y-fast trie* and Mehlhorn and Näher [8] structure, which uses dynamic perfect hashing [3].

These two solutions are both two-level data structures using buckets; within each bucket some other standard data structure is used to answer queries. This approach makes these structures complicated, and makes it difficult to decouple the space that is necessary to store the structure and the set S. This is not only a practical issue when we are interested in sets of "long" strings (i.e., shorter than 2^w rather than w, where w is the machine word) of variable length.

The data structure discussed in this paper, the *dynamic z-fast trie*, can be thought of as the first simple optimal digital data structure that is neither recursive nor needs to use bucketing to achieve optimal time and linear space. Moreover, we present a simple variant that provides results with high probability within the same bounds of the standard version for "long" strings, always using a number of words linear in n (rather than in the number of bits that is necessary to represent S). In both cases, our space estimates are *not* asymptotic: we can provide *exact* bounds, which amount to about a dozen machine words per element.

The name "z-fast trie" was used for the first time in [2]. The structure described in this paper is inspired by the idea of *fat binary search* introduced therein, but has little else in common. In particular, the structure described in [2] is a static structure that just manipulates strings and returns strings: a z-fast trie is only able to provide the longest *extent* (see Section 2) in the compacted trie on S that is a prefix of a given query string. The dynamic z-fast trie, instead, is based on a standard compacted trie with two additional pointers per node. On leaves, the additional pointers are used to keep track of the original set in a doubly linked list. On internal nodes, the pointers are called *jump pointers*: they make it possible to jump quickly to the farthest left/right descendant. Additionaly, the z-fast trie uses a dictionary, inspired by the static case, to locate quickly the exit node of a string using a variant of the *fat binary search*. The main combinatorial results we prove (Theorem 3 and 4) highlights a deep connection between jump pointers and fat binary search that make it possible to update the pointers very quickly.

Results. Given a prefix-free set S of n binary strings, we let

$$x^- = \max\{\, y \in S \mid y < x \,\} \qquad \text{(the } \textit{predecessor} \text{ of } x \text{ in } S\text{)}$$
$$x^+ = \min\{\, y \in S \mid y \geq x \,\} \qquad \text{(the } \textit{successor} \text{ of } x \text{ in } S\text{)},$$

where $\leq$ is the lexicographic order. A *predecessor/successor* query is given by a string x, and the answer is $x^{\pm}$. An *existential prefix query* tells whether there are strings in S with given prefix x. An *existential range query* is given by a pair x, y of strings, and tells whether there are strings in S in the (lexicographic) interval $[x\,..\,y)$. Standard prefix (range, respectively) queries report the set of strings (expressed as pointers) with prefix x in S (the set of strings in the interval $[x\,..\,y)$, respectively).

We work in the standard RAM model with a word of length w, allowing multiplications, under a full randomness assumption[1]. We also discuss results in the *cache-oblivious* model [6], which analyses RAM algorithms in the I/O model using an ideal cache replacement strategy.

We prove the following statements:

Theorem 1. *If the string length is bounded by $O(w)$, the dynamic z-fast trie answers predecessor and successor queries in time $O(\log \max(|x|, |x^+|, |x^-|))$, existential prefix queries in time $O(\log |x|)$ and existential range queries in time $O(\log \max\{|x|, |y|\})$. Insertions are performed in time $O(\log \max\{|x|, |x^+|, |x^-\})$, and deletions in time $O(\log |x|)$, in expectation. Besides the space required to store S, the z-fast trie needs $12n$ pointers and n integers.*

For long strings (i.e., strings of length bounded by 2^w) we can provide a simple variant of the structure (the *signed z-fast trie*):

Theorem 2. *If the string length is bounded by 2^w, the signed z-fast trie answers predecessor and successor queries in time $O(|x|/w + \log \max\{|x|, |x^+|, |x^-|\})$ and $O(|x|/B + \log \max\{|x|, |x^+|, |x^-|\})$ I/Os (in the cache-oblivious model)* with high probability. *Existential prefix queries are answered, always w.h.p., in time $O(|x|/w + \log |x|)$ and $O(|x|/B + \log |x|)$ I/Os. Existential range queries are answered, always w.h.p., in time $O(\max\{|x|, |y|\}/w + \log \max\{|x|, |y|\})$ and $O(\max\{|x|, |y|\}/B + \log \max\{|x|, |y|\})$ I/Os. Insertions are performed in time $O(\log \max\{|x|, |x^+|, |x^-|\})$, and deletions in time $O(\log |x|)$, in expectation. Besides the space required to store S, the signed z-fast trie needs $12n$ pointers, n integers, and $3n$ signatures of $O(w)$ bits.*

We can also answer prefix and range queries: if the answer is a set of k strings, then the I/O and time costs are the same as in the existential case, plus $O(k)$ and $O(\log \max\{|x^-|, |y^+|\} + k)$ for prefix and range queries respectively. Note that the space bounds assume a dictionary using $3n$ buckets (e.g., a linear-probing hash table with load factor 2/3 and backing arrays whose length is a power of two, or a variant of Cuckoo hashing [10]). We remark that the only reason of having expectations in the update bounds is the presence of a dictionary: improvements in dynamic dictionaries reflects immediately in improvements to the z-fast trie time and space bounds.

To our knowledge, this is the first paper to propose a digital data structure for variable length strings with optimal performance in the RAM model and good performance in the cache-oblivious model. Nonetheless, we stress the fact that the main feature of dynamic z-fast tries is that they are simple data structures which can be actually implemented, and that the space usage is just about a dozen machine words per element, independently of the string length. A complete implementation is distributed under the GNU Lesser General Public License[2].

Previous work. From a RAM perspective, several structures already provide optimal bounds both in terms of n (e.g., Andersson and Thorup [1] achieve

[1] We remark that hash functions are only used to implement a hash table and to compute signatures in the "long string" case.

[2] `http://sux4j.dsi.unimi.it/`

$O(\sqrt{\log n/\log\log n})$ update/query time) and in term of the string length for fixed-length strings (e.g., y-fast tries [13]). The main contribution of the dynamic z-fast trie is that it is a simple structure that can be easily implemented, and that it works with variable-length strings longer than $O(w)$.

Andersson and Thorup note that by using a trie on the alphabet 2^w and some very careful bookkeeping it is possible to achieve $O(|x|/w + \sqrt{\log n/\log\log n})$ time by storing in each node of the trie a predecessor structure.[3] Although in principle this solution would be faster in the RAM model than ours (for sufficiently small sets), in the cache-oblivious model it would require $(|x|/w + \sqrt{\log n/\log\log n})$ I/Os—in most cases worse than $O(|x|/B + \log\max(|x|, |x^+|, |x^-|))$ I/Os achieved by our data structure. Moreover, the construction is rather involved; finally, while the $O(|x|/w)$ term in our bound is simply due to the evaluation of a hash function and some bit-vector comparisons, the $O(|x|/w)$ term in the trie technique depends on $|x|/w$ accesses to dictionaries, an operation that in practice is orders of magnitude slower.

In the cache-oblivious model, the string B-trees and its variants [4] provide predecessor/successor queries using $O(|x|/B + \log_B n)$ I/Os. We remark that this bound is in general incomparable to ours. Finally, note that the RAM time bound in this case is essentially always worse than ours.

2 Notation and Tools

We use von Neumann's definition and notation for natural numbers: $n = \{0, 1, \ldots, n-1\}$, so $2 = \{0, 1\}$ and 2^* is the set of all binary strings. If x is a string, x juxtaposed with an interval is the substring of x with those indices (starting from 0 in that interval). Thus, for instance, $x[a\,..\,b)$ is the substring of x starting at position a (inclusive) and ending at position b (exclusive). We will write $x[a]$ for $x[a\,..\,a]$. The symbol $\preceq$ denotes prefix order, and $\prec$ is its strict version.

3 The Components of a Z-Fast Trie

A *dynamic z-fast trie* (hereafter referred to simply as z-fast trie) is a compacted trie endowed with two additional pointers per internal node and with a dictionary. More precisely, the z-fast trie can be thought of as an indexing structure built on a set of binary strings S (stored somewhere in memory), and made of two components:

- a compacted trie, whose nodes contain data that will be described later;
- a dictionary (e.g., implemented as a hashtable), whose keys are binary strings and whose values are nodes of the trie.

[3] Using other kinds of digital structures (e.g., the structure described in this paper) it would be possible to achieve similarly $O(|x|/w+\log w)$, but this is of little advantage as $O(|x|/w + \log|x|) = (|x|/w + \log(|x|/w) + \log w) = O(|x|/w + \log w)$.

3.1 The Compacted Trie

We start by defining some basic notation that will be necessary to describe the trie. Consider the compacted trie $T(S)$ [7] built for a prefix-free set of strings $S \subseteq 2^*$. For a given node α of the trie, we define (see Figure 1):

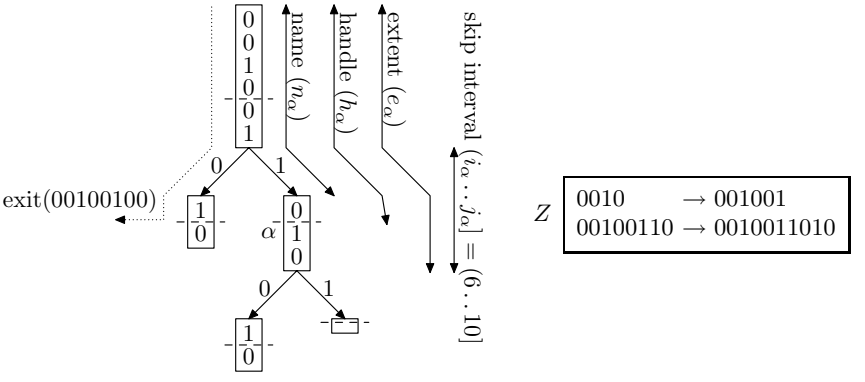


Fig. 1. (above) A compacted trie for S, and the corresponding nomenclature, and its dictionary. where $S = \{001001010, 0010011010010, 00100110101\}$.

- e_α, the *extent of node* α, is the longest common prefix of the strings represented by the leaves that are descendants of α;
- n_α, the *name of node* α, is the string e_α deprived of the string stored at α;
- given a string x, we let exit(x) be the exit node of x, that is, the only node α such that n_α is a prefix of x and either $e_\alpha = x$ or e_α is not a prefix of x; moreover, we call *parex node of* x the *parent* of the exit node of x, or a special symbol $\perp$ if the exit node of x is the root (note that the parex of x is the node with the longest extent that is a *proper* prefix of x);
- the *skip interval* $(i_\alpha \,..\, j_\alpha]$ associated to α is $(0\,..\,|e_\alpha|]$ for the root, and $(|n_\alpha| - 1\,..\,|e_\alpha|]$ for all other nodes.

For the time being we assume that the trie contents are as follows: every internal node α contains a pointer to its two children, the extremes i_α and j_α of its skip interval, its own extent e_α and two additional pointers (called "jump pointers"), J^- and J^+ whose meaning will be explained later. Leaves are organized in a doubly linked list: each leaf, besides a pointer to the corresponding string of S, stores two pointers to the next/previous leaf in lexicographic order.

3.2 The Dictionary

The dictionary associated to a z-fast trie maps certain prefixes of the extents of internal nodes, called *handles*, to the corresponding node. This mapping is sufficient to navigate the trie and find exit(x) in time $O(|x|/w + \log|x|)$. The mapping is similar to that described in [2], where *static* z-fast tries were defined.

Here we are going to prove some new important properties of this mapping (in particular, its rôle in the discovery of *2-fat ancestors*) that are at the basis of the dynamic version. The key definition is the following:

Definition 1 (2-fattest numbers and handles). *The* 2-fattest number *of an interval* $(a\,..\,b)$ *of positive integers is the unique integer in* $(a\,..\,b)$ *that is divisible by the largest power of two, or equivalently, that has the largest number of trailing zeroes in its binary representation. The* handle h_α *of a node* α *is the prefix of* e_α *whose length is 2-fattest number in the skip interval of* α *(see Figure 1). If the skip interval is empty (which can only happen at the root) we define the handle to be the empty string.*

We will in general use "fat" for numbers divisible by large powers of two, and "lean" with the opposite meaning. We remark that if f is 2-fattest in $(a\,..\,b)$, it is also 2-fattest in every subinterval of $(a\,..\,b)$ that still contains f.

We define the map Z from 2^* to the set of *internal* nodes by

$$Z : h_\alpha \mapsto \alpha,$$

for every internal node α. We assume that Z is actually implemented using a hash table, also denoted by Z, and that the entries of this hash table can also be accessed (by their address) if needed (of course, any dictionary allowing satellite data will do the job).

Definition 2 (2-fat jump). *Given a positive integer* $x = a2^b$*, with* a *odd, we call*

$$x' = (a+1)2^b,$$

the (2-fat) jump *of* x*: it is the smallest integer larger than* x *with more trailing zeroes than* x*. We let* $x^{(n)}$ *denote the application of* n *jumps (i.e.,* $x^{(0)} = x$ *and* $x^{(n+1)} = (x^n)'$*). Moreover, the* 2-fat backjump *of* x *is* $x_\prime = (a-1)2^b$*: it is the largest integer smaller than* x *with more trailing zeroes than* x*. Note that* $(2^n)_\prime = 0$*, but* $(2^n)' = 2^{n+1}$*.*

It is an easy observation that iterating n times a jump we always obtain a number larger than or equal to 2^n. This property will be silently used throughout the paper.

Definition 3 (2-fat ancestors). *Given a node* α*, its* 2-fat ancestors *are the nodes* β *such that* $h_\beta \in \{e_\alpha[0\,..\,|h_\alpha|), e_\alpha[0\,..\,|h_\alpha|_\prime), e_\alpha[0\,..\,|h_\alpha|_{\prime\prime}), \ldots\}$.

The above definition may seem a bit cryptic: the idea is that we are looking for nodes on the path leading to α whose handle lengths are fatter than that of α. This property will be essential when updating the trie. Note that not all handles in the definition above necessarily exists. We will need the following lemma, which relates 2-fat ancestors with 2-fattest numbers of certain intervals:

Lemma 1. *Given a positive integer* r*, let* $a_0 = 0$*, and* a_{k+1} *be the 2-fattest number in* $(a_k\,..\,r]$*. Then* $\{a_0, a_1, \ldots, a_t = r\} = \{r, r_\prime, r_{\prime\prime}, \ldots, 0\}$*. Morevoer,* t *is equal to the number of ones in the binary expansion of* r*.*

Proof. First note that 0 belongs to both sets. Now let $p_1 > p_2 > \cdots > p_k$ be the positions of the ones in the binary expansion of r: the i-th backjump of r is exactly $\sum_{j \le k-i} 2^{p_j}$. We prove that $a_i = r - r \bmod 2^{p_i}$. For $i = 1$, a_1 is by definition the 2-fattest number in $(0\,..\,r]$, which is exactly the largest power of two in such interval. For the inductive step, we have defined a_{i+1} as the 2-fattest number in the interval $(r - r \bmod 2^{p_i}\,..\,x]$; observing that $r = \sum_j 2^{p_j}$, we deduce that $a_{i+1} = x - x \bmod 2^{p_{i+1}}$. The thesis follows easily. □

The most important property of Z is that it makes us able to find very quickly all 2-fat ancestors either of the parex or of the exit node of a string, using Algorithm 1, which is inspired by the querying algorithm presented in [2]: in particular, at the end of the algorithm, on the top of the stack we will find parex(x) or exit(x).

Algorithm 1. Querying the z-fast trie: at the end of the execution, on the stack you will find either parex(x) or exit(x) and its 2-fat ancestors

```
Input: a string x
Output: a stack containing the 2-fat ancestors of parex(x) or exit(x)
S ← empty stack
a, b ← 0, |x|
while b − a > 0 do
   f ← the 2-fattest number in (a..b]
   β ← Z(x[0..f))
   if β ≠ ⊥ then
      a ← |e_β| {Move from (a..b] to (|e_β|..b]}
      push β on S
   else
      b ← f − 1 {Move from (a..b] to (a..f − 1]}
   end if
end while
return S
```

The *cutpoint* for a string x with respect to a trie is the length of the longest common prefix bewteen x and exit(x). There are two cases: we say that x *cuts high* if the cutpoint is strictly smaller than $|h_{\mathrm{exit}(x)}|$, *cuts low* otherwise. Note that only in the second case the handle of exit(x) is a prefix of x.

Theorem 3. *Let x be a nonempty string, and*

$$\eta = \begin{cases} \mathrm{parex}(x) & \text{if } x \text{ cuts high or } \mathrm{exit}(x) \text{ is a leaf} \\ \mathrm{exit}(x) & \text{otherwise.} \end{cases}$$

Let $\alpha_0, \alpha_1, \ldots, \alpha_t = \eta$ be the sequence of 2-fat ancestors of η. Let $(a\,..\,b]$ be the interval maintained by Algorithm 1. Before and after each iteration the following invariant is satisfied: $a = 0$ or $a = |e_{\alpha_j}|$ for some j and $|h_\eta| \le b$. Moreover, all values $|e_{\alpha_j}|$ are attained by a (in order of increasing j) at some point of the execution.

Proof. Note that h_η is the longest handle in Z that is a prefix of x. The invariant is trivially true at the start, as the initial interval is $(0\,..\,|x|]$. While $a = 0$ (we follow the notation of Algorithm 1), f goes through the powers of two not larger than $|x|$, starting from the largest one, and whenever we take the negative branch we know that $b = f - 1 \geq |h_\eta|$. The first time we find a node β with a handle of length f that is a prefix of x we take the positive branch of the test (as f is 2-fattest in $(0\,..\,b] \supseteq (0\,..\,|h_\eta|]$, and thus in in $(0\,..\,|e_\eta|]$). By the definition of 2-fat ancestor, $\beta = \alpha_0$.

We now prove by induction that in the rest of execution the invariant is true. At each step we pick the 2-fattest number $f \in (|e_{\alpha_i}|\,..\,b]$, and change interval. Let $\beta = Z(x[0\,..\,f))$. We have two cases:

- If $f \leq |h_\eta|$, $x[0\,..\,f)$ must be the handle of a node β, as f is 2-fattest in $(|e_{\alpha_i}|\,..\,b] \supseteq (|e_{\alpha_i}|\,..\,|h_\eta|]$. Moreover, necessarily $\beta = \alpha_{i+1}$ because of Lemma 1. Thus, by setting $a = |e_\beta| = |e_{\alpha_{i+1}}|$ the invariant is preserved.
- If $f > |h_\eta|$, necessarily $\beta = \bot$. Thus, by setting $b = f - 1 \geq |h_\eta|$ we preserve the invariant. □

A few remarks are in order:

- Notice that $\mathrm{exit}(x)$ can be recovered in constant time: if $\mathrm{parex}(x) = \bot$, $\mathrm{exit}(x)$ is the root; otherwise, $\mathrm{exit}(x)$ is the left or right child of $\mathrm{parex}(x)$ depending on whether $x[|e_{\mathrm{parex}(x)}|]$ is zero or one, respectively.
- Algorithm 1 completes in at most $\log |x|$ iterations.
- If we get $\mathrm{exit}(x)$ and we need $\mathrm{parex}(x)$, we just have to remove $\mathrm{exit}(x)$ from the stack and check whether $\mathrm{parex}(x)$ is at the top of the stack: in this case, we can return the stack as it is; otherwise, we set $a = 0$ if the stack is now empty or $a = |e_{\mathrm{top}(S)}|$, $b = |e_{\mathrm{parex}(x)}|$, and we restart the algorithm: by Theorem 3 the top of the new stack is necessarily $\mathrm{parex}(x)$.
- At the end of Algorithm 1 the returned stack contains the 2-fat ancestors of η in increasing handle-length order.
- Finding the 2-fattest number in an interval requires the computation of the most significant bit[4], but alternatively we can check that $(1 \ll i)\,\&\,a \neq (1 \ll i)\,\&\,b$ for decreasing i: if the test is satisfied, the number is $b\,\&\,{-1} \ll i$, otherwise we decrement i.
- In Definition 1 we had to state separately the case of the empty skip interval, but it is clear that it is just for convenience, as Algorithm 1 will never query Z using the empty string.

3.3 Implementing the Dictionary: Short vs. Long Strings

Each of the $O(\log |x|)$ iterations of Algorithm 1 performs a query to Z with some prefix of x; the time required by such queries depend on the way the dictionary Z is concretely implemented.

[4] More precisely, the 2-fattest number in $(a\,..\,b]$ is $-1 \ll \mathrm{msb}(a \oplus b)\,\&\,b$.

Our description so far assumes that one can determine, for a given string x, whether $Z(x)$ is defined and, if so, what is the associated node (specified through a pointer). If Z is implemented through a constant-time hash table, under standard universal-hashing assumptions [9] queries to the hash table require a worst-case constant number of accesses to the table (whereas modifications yield an expected constant number of accesses), but each access may imply a comparison of keys: if keys are "short" (i.e., if their length is $O(w)$), such comparisons can be performed in constant time, and hence Algorithm 1 requires time $O(\log |x|)$. However, if keys are "long" (more than a constant time the machine-word size), a more complex solution should be devised.

Let us discuss briefly how to obtain results analogous to those described up to here, but only with high probability, when the string length is bounded by 2^w. In this circumstance, we use a *signature-based z-fast trie*: the hashtable Z contains additional signatures of $c \log n + \log w$ bits, and queries are performed to the table using signatures, rather than actual strings; more precisely, when the hashtable is queried, no string comparison is performed, but only comparisons between signatures. Note that signatures can be computed incrementally: after a preprocessing using $O(|x|/w)$ times and $O(|x|/B)$ I/Os, computing a signature for a prefix of x takes constant time.

It is easy to see that the probability of performing w probes without getting a false positive or a colliding key is at most $2/n^c$: indeed, since signatures have size $c \log n + \log w = \log n^c w$, the probability of a colliding key at any given step is at most $\binom{n}{2}(1/n^c w)^2 \leq 1/n^{2c-2}w^2$ and the probability of a false positive is at most $1/n^{c-1}w$; therefore, the probability of an error at some step is at most $w(1/n^{2c-2}w^2 + 1/n^{c-1}w) \leq 2/n^{c-1}$. Thus, Algorithm 1 will end correctly in time $O(|x|/w + \log |x|)$ with high probability. Note that since n is not known in advance, the signatures must actually be $cw + \log w = O(w)$ bits long (we assume, as usual, that the set size is bounded by 2^w).

At the end of the algorithm, we can check in time $O(|x|/w)$ that the length of the longest common prefix of x and $e_{\text{exit}(x)}$ intercepts $\text{exit}(x)$ (which implies that the result is correct), and re-run the algorithm using actual keys instead of signatures if something went wrong (this requires time $O((|x|/w + 1) \log |x|)$ and $O((|x|/B + 1) \log |x|)$ I/Os).[5] We note that if the exit node is correct, then necessarily the stack of 2-fat ancestors is correct, too.

3.4 Jump Pointers

As a last step towards a complete z-fast trie, we endow all internal nodes with two additional pointers, called *jump pointers*. The *left* jump pointer $J^-(\alpha)$ of a node α points somewhere along the path defined recursively by the left child pointers (analogously for *right* jump pointers $J^+(\alpha)$). How far the pointer goes depends

[5] In practice, if the dictionary is implemented by a hash table it is much more efficient and convenient to keep track of the (very few) signatures that are duplicates, and perform exact comparisons when hitting them in the probing phase. Mistakes are then restricted just to false positive. For instance, on a 64-bit machine, 64-bit signatures are sufficient to handle efficiently hundreds of billions of strings.

on some arithmetics related to 2-fattest numbers. Our goal is to position jump pointers so that if the rightmost (largest) leaf under node α contains the string x, in $O(\log |x|)$ steps we can get to leaf. At the same time, when we perform insertions or deletes in a trie we want to be able to update the pointers in time $O(\log |e_{\text{exit}(x)}|)$, where x is the string added to the trie. Remember that a *right (left) descendant* is either the right (left) child, or a descendant of the right (left) child.

Definition 4. *An internal node* α intercepts *an integer* k *if* k *belongs to the skip interval of* α*; a leaf* α *intercepts* k *if* $k > i_\alpha$*, that is, if* k *belongs to the skip interval of* α *or is larger than its right extreme. Given an internal node* α*, its* left (right) jump pointer *points to its unique left (right) descendant that intercepts* $|h_\alpha|'$.

The intuition behind this definition is that jump pointers point near for nodes with lean handles, and point far for nodes with fat handles. Of course, since when we follow a jump pointer of α we land into a node β whose skip interval contains $|h_\alpha|'$, the handle of the β will have strictly fatter length than $|h_\alpha|$: as a consequence, we never need more than $\log |x|$ jumps to reach a leaf containing the string x.

3.5 Space Usage

As described up to now, the z-fast trie is not very space-efficient: the internal nodes must contain their extent, and the dictionary must contain handles, which implies that the space usage is (at most) three times the one necessary to store S. We briefly describe how to avoid storing additional strings at all.

The first observation is that the extent of every internal node is a prefix of some string in S, and such strings are stored in leaves. Thus, we can just store in each internal node the skip interval and a reference to one of its descendant leaves. It is easy to make this relation between internal nodes and leaves an injection (i.e., different nodes always point to different leaves); this way, by storing also in the leaf a backward reference it is possible to perform insertions and deletions with few pointers assignment (the interested reader can easily work out the details).

Moreover, it is not necessary to store both extremes of the skip interval in an internal node: it is enough to store its left extreme i_α, because the right extreme can be deduced from the children.

Finally, in the dictionary every handle is associated with the corresponding internal node, which contains a (now indirect) representation of its extent. From the extent it is also possible to obtain (by truncation) the handle, which implies that we do not need to store the keys of the dictionary explicitly if we use a hash table (although other kind of dictionaries might behave differently).

We can at this point provide a rather precise evaluation of the space required by the z-fast trie, avoiding any asymptotic notation. Each leaf contains four pointers, and internal nodes contains five pointers and an integer: so the trie occupies $(9p + w)n$ bits (where p is the bit size of a pointer). Each dictionary

entry contains a pointer and possibly a signature (only in the case of signature-based z-fast tries); assuming a linear-probing implementation with a load factor of 2/3 and backing array sized as a power of two, we have at most $3n$ entries, the dictionary takes $3(p + s)n$ bits, where $s = 0$ in the exact case and s is the size of a signature in the signature-based case.

4 Queries

We now proceed to describe how the various types of queries discussed in Theorem 1 and 2 are handled by the z-fast trie, and prove the associated time bounds.

Predecessor/successor. In time $O(|x|/w + \log |x|)$ we recover the exit node of x using Algorithm 1. Then, by examining $e_{\text{exit}(x)}$ and x we decide in time $O(|x|/w)$ whether x exits on the left or on the right. By moving through jump pointers to the left or right descendant leaf, respectively, we can find the successor or the predecessor of x, respectively, in time $O(\log \max\{|x^-|, |x^+|\})$; then, possibly after a single access to the doubly linked list of leaves, we can return the result. For example, if x exits on the left and we want to obtain its predecessor x^-, we must follow left jump pointers until reaching a leaf and then move to the previous leaf.

Prefix queries. In time $O(|x|/w + \log |x|)$ we recover the exit node of x using Algorithm 1. We then check in time $O(|x|/w)$ that $x \preceq e_{\text{exit}(x)}$, and, if so, enumerate the leaves in the subtree under $\text{exit}(x)$ (the enumeration requires $O(k)$ time to output k elements). If $x \not\preceq e_{\text{exit}(x)}$ the output is empty. Note that the test $x \preceq e_{\text{exit}(x)}$ is sufficient to answer existential queries.

Range queries. For the range query $[x\,..\,y)$ we first locate $\text{exit}(x)$ and $\text{exit}(y)$ using Algorithm 1 in time $O(\max\{|x|, |y|\}/w + \log \max\{|x|, |y|\})$. Starting from $\text{exit}(x)$ ($\text{exit}(y)$) we then follow left (right, respectively) jump pointers until reaching a node α (β, respectively) such that $|e_\alpha| \geq |y|$ ($|e_\beta| \geq |x|$, respectively). This requires time $O(\log |y|)$ ($O(\log |x|)$, respectively).

Now existential queries can be answered as follows: if x exits on the left the output is empty iff $e_\alpha \geq y$, and if y exits on the right the output is empty iff $e_\beta < x$; both tests can be performed in time $O(\max\{|x|, |y|\}/w$. If x exits on the right and y exits on the left we recover x^- and y^+ in time $O(\log \max\{|x^-|, |y^+|\})$, and then the output is empty iff the successor of x^- is not y^+.

Standard range queries can be answered easily noting that after the range emptiness check either we know x^- (y^+, respectively), or we can recover it in time $O(k)$, where k is the size of the output. For example, if x exits on the left and the output is not empty then the entire subtree under $\text{exit}(x)$ is part of the output, and thus has size and depth bounded by k: we conclude that following left jump pointers up to x^+ has cost $O(k)$ (recovering x^- is then trivial).

We now relieve existential from the dependence on $O(\log \max\{|x^-|, |y^+|\})$ range queries. Note that if x exits on the right and y exits on the left then

$\operatorname{lcp}(x, y)$ is necessarily the extent of some node η that we can recover using Algorithm 1 in time $O(\max\{|x|, |y|\}/w + \log \max\{|x|, |y|\})$. We then follow the right (left, respectively) jump pointers of the left (right, respectively) child of η until reaching a node α (β, respectively) such that $|e_\alpha| \geq |x|$ ($|e_\beta| \geq |y|$, respectively). This requires time $O(\log |x|)$ ($O(\log |y|)$, respectively). The output is not empty iff $x \leq e_\alpha$ or $e_\beta < y$.

I/O bounds. The I/O bounds are easily proved as we scan x (or y) a fixed number of times to preprocess them so to obtain hashes of prefixes in constant time, or to compare them to other strings.

5 Updates

Insertions. We describe how to insert a new string x into the z-fast trie for a set $S \neq \varnothing$. We assume that the 2-fat ancestors of $\operatorname{parex}(x)$ and the exit node of x have been found using Algorithm 1. To simplify the description, we describe in turn how each component is updated. First, however, we state the fundamental property we will use:

Lemma 2 (Parenthesis property). *For positive integers x and y we have that*

$$x_\prime < y < x' \quad \Longrightarrow \quad x_\prime \leq y_\prime < y' \leq x'.$$

Proof. By contradiction, suppose first that $x_\prime < y < x' < y'$, that is, $(2h)2^b < (2k+1)2^c < (2h+2)2^b < (2k+2)2^c$, where $x = (2h+1)2^b$ and $y = (2k+1)2^c$. If $b < c$, dividing by 2^{b+1} we obtain $h < (2k+1)2^{c-(b+1)} < h+1$, which is impossible. Otherwise, if $b \geq c$ dividing by 2^c we obtain $2k+1 < (2h+2)2^{b-c} < 2k+2$, which is again impossible. Reasoning analogously about $y_\prime < x_\prime < y < x'$ completes the proof. □

The compacted trie. Updating the compacted trie by adding a new string x requires the usual constant number pointer updates. We remark that we have also to insert the new leaf into the doubly linked list: to find the predecessor or the successor in S of the new string, we simply follow the right or left jump pointers, respectively, of $\operatorname{exit}(x)$, and then move along the list.

The dictionary. We have to add to the trie a new leaf and a new internal node. If x cuts low we have to add to the dictionary a new entry for the new internal node, and update the entry of the split node if it is not a leaf, as its handle has changed. If x cuts high, we just add to the dictionary the new internal node.

Jump pointers. A node α *crosses* a node β if $e_\alpha \preceq e_\beta$ and $|h_\alpha|' > i_\beta$ (i.e., α potentially jumps to some left or right descendant of β). When inserting a new string with exit node β, only nodes crossing β might need to change their jump pointers. The fundamental combinatorial fact that we will use to update quickly the z-fast trie is the following connection between fat binary search and jump pointers:

Theorem 4. *The nodes crossing β are a subset of the 2-fat ancestors of its parent γ.*

Proof. Indeed, $|h_\gamma|' > i_\beta$ (as $i_\beta = j_\gamma$), so certainly γ crosses β. Now suppose by induction that a 2-fat ancestor ξ crosses β. Because of the parenthesis property, any node with a handle of length $t < |h_\xi|$ must satisfy either $t' \leq |h_\xi|$ or $t' > |h_\xi|'$, and the largest t satisfying the second condition is $|h_\xi|_\prime$. We can iterate this process until $|h_\xi|_{\prime\dots\prime}$ is the handle of some node, which is necessarily the next 2-fat ancestor. □

Not all the jump pointers of 2-fat ancestors need to be updated: actually, we need to update only pointers to nodes that are left (right) descendant of β.

Algorithm 2 exploits the parenthesis property to find such nodes: it just pops nodes out of the 2-fat-ancestors stack and matches them greedily with left (right) descendants of the exit node. As soon as we cannot find a matching descendant, we can stop updating.

Besides updating existing pointers and setting a few other in obvious ways, there are cases in which we create a new node and set somehow its left and right child pointers. In this case, we can easily set the jump pointers by following the jump pointers of the children, and stop as soon as we find a node intercepting the jump length of the new node: the parenthesis property guarantees that this procedure is correct. All these operations require time $O(\log |x|)$.

Algorithm 2. Algorithm to find the nodes that need updates to $J^{\pm}(-)$

Input: a string x
Output: the set of nodes that need jump pointer updates when x is inserted
$R \leftarrow \varnothing$
$S \leftarrow$ stack of 2-fat ancestors of parex(x) (Algorithm 1)
$\beta \leftarrow \text{exit}(x)$
while S not empty **do**
  pop α from S
  while β is internal and $J^{\pm}(\alpha) \neq \beta$ **do**
    $\beta \leftarrow J^{\pm}(\beta)$
  end while
  if $J^{\pm}(\alpha) = \beta$ **then**
    $R \leftarrow R \cup \{\alpha\}$
  else
    break
  end if
end while
return R

Deletions. Deletions mostly follow the insertion steps uneventfully. There is just one important observation to be made: to fix the jump pointers, we need to know the 2-fat ancestors of the parent of parex(x), not of parex(x) (so it is possible that we have to restart Algorithm 1 *twice*).

References

1. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. Assoc. Comput. Mach. 54(3), 1–40 (2007)
2. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In: Proceedings of the 20th Annual ACM-SIAM Symposium On Discrete Mathematics (SOD), pp. 785–794. ACM Press, New York (2009)
3. Dietzfelbinger, M., Karlin, A.R., Mehlhorn, K., auf der Heide, F.M., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput. 23(4), 738–761 (1994)
4. Ferragina, P., Grossi, R.: The string B-tree: A new data structure for string search in external memory and its applications. J. Assoc. Comput. Mach. 46(2), 236–280 (1999)
5. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. System Sci. 47(3), 424–436 (1993)
6. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 1999), pp. 285–297. IEEE Comput. Soc. Press, Los Alamitos (1999)
7. Knuth, D.E.: The Art of Computer Programming. Addison–Wesley, Reading (1973)
8. Mehlhorn, K., Näher, S.: Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. Inf. Process. Lett. 35(4), 183–189 (1990)
9. Pagh, A., Pagh, R., Ruzic, M.: Linear probing with constant independence. In: Johnson, D.S., Feige, U. (eds.) Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC 2007), pp. 318–327. ACM, New York (2007)
10. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001)
11. Pătraşcu, M., Thorup, M.: Randomization does not help searching predecessors. In: SODA 2007: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 555–564. Society for Industrial and Applied Mathematics, Philadelphia (2007)
12. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Theory Comput. Systems 10(1), 99–127 (1976)
13. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Inform. Process. Lett. 17(2), 81–84 (1983)

Improved Fast Similarity Search in Dictionaries*

Daniel Karch, Dennis Luxen, and Peter Sanders

Karlsruhe Institute of Technology
danielkarch@gmail.com, {luxen,sanders}@kit.edu

Abstract. We engineer an algorithm to solve the approximate dictionary matching problem. Given a list of words $\mathcal{W}$, maximum distance d fixed at preprocessing time and a query word q, we would like to retrieve all words from $\mathcal{W}$ that can be transformed into q with d or less edit operations. We present data structures that support fault tolerant queries by generating an index. On top of that, we present a generalization of the method that eases memory consumption and preprocessing time significantly. At the same time, running times of queries are virtually unaffected. We are able to match in lists of hundreds of thousands of words and beyond within microseconds for reasonable distances.

1 Introduction and Previous Results

The problem of searching approximate of matches in a dictionary arises in many fields. For example, Google's 'Did you mean' feature catches typos in search queries. But in some settings, the uncertainty is higher and therefore one is not interested in the best match, but also in other matches which are still within a certain distance from the query.

Each word is represented by a string of characters over a finite alphabet Σ. The Levenshtein distance $ed(a, b)$ defines a metric between two words $a, b \in \Sigma^*$ and is used in this work to compute the distance between two words. Since distance computations are rather expensive, it is natural to find an algorithm that does not compare the input to the entire dictionary, but only a few entries. A so-called *filter* represents a criterion to quickly discard large portions of the search space. The exploitation of the underlying metric space implied by the edit distance [1] is easy. The set of words is partitioned by the distance of each element to a more or less carefully chosen and perhaps random pivot element. By computing the distance to the pivot, the search space is pruned using the triangle inequality. However, this approach has limited effect, e.g. in natural language dictionaries. To cope with the limitations, different schemes were introduced from using multiple pivots to tree-like data structures. The oldest of such trees is the BK-tree data structure proposed by Burkhard and Keller [2], which is built recursively. A root is selected whose subtrees are identified by distance values

* Partially supported by DFG Grant 933/5-1. A full version of the paper is available at http://arxiv.org/abs/1008.1191. We would like to thank Johannes Fischer for the fruitful discussions and the anonymous referees for the detailed reviews.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 173–178, 2010.

to the root. The i-th subtree consists of elements of the dictionary at distance i to the root. The subtrees are recursively built until the number of elements in a subtree is below some threshold. See Chávez et al. [3] for a survey.

The problem of approximately matching words can be categorized into matching elements from a set of words or matching arbitrary patterns in strings [1]. Cole et al. [4] give a solution for the dictionary matching problem using $O(n \log^d n)$ space and answer a query in $O(m \cdot \log \log n + occ)$ for a dictionary of size n, query length m, edit distance d. Here, occ is the number of occurrences of the pattern. Russo et al. [5] propose a compressed index that performs well for $d = 1, 2, 3$, but not for larger d. Mihov and Schulz [6] present a sophisticated but complicated method to solve the problem with universal Levenshtein automata. The best known linear space solution needs $O(m^{d-1} \log n \log \log n + occ)$ query time [7] for error $d \geq 2$. However, this solution is fairly complicated and involves large constant factors, and to our knowledge there aren't any implementations yet.

Practically oriented work focused on filtering algorithms that use linear space, but these do not have strong worst case performance guarantees. The technique of so-called q-grams [8] is popular among practitioners and works for Hamming distance. q-grams are sub words of length q and the q-gram distance (similarity) is defined by the number of q-grams two words share. Taking q letters from a word as before and introducing *don't care* defines a pattern including gaps instead of sub-word. The major difficulties with gapped q-grams is the computation of the smallest number of matching q-grams between a pattern and a text.

2 Approximate Dictionary Matching

Our method can be seen as an implementation of a general approach to approximate matching known as *(lossless) filtering*. This can be formalized as follows: Given a set $\mathcal{S}$ of words over a finite alphabet Σ, a metric $\delta : \Sigma^* \times \Sigma^* \to \mathbb{R}_0$, and an error threshold d, a preprocessing algorithm produces a data structure that allows fast evaluation of a function $F : \Sigma^* \to \mathcal{P}(\mathcal{S})$. For a query word $q \in \Sigma^*$, $F(q)$ computes a set of candidate words from $\mathcal{S}$ such that the set of approximate matches $\{s \in \mathcal{S} : \delta(q, s) \leq d\}$ is a subset of $F(q)$.

Deletion Neighborhood. We improve a filtering technique called *Fast Similarity Search (FastSS)* [9] which is a generalization of the method proposed by Mor and Fraenkel [10]. For integer d and a word $w \in \Sigma^*$ the *d-(deletion-)neighborhood* $\mathcal{N}_d(w)$ is defined as the set of all sub words of w with exactly d deleted positions. Each element of $\mathcal{N}_d(w)$ is called a *residual string*. Furthermore, a string w is called originating string for residual r if and only if $r \in \mathcal{N}_d(w)$. We obtain a lossless filter for a set of words $\mathcal{S}$ by precomputing the d-neighborhoods of strings in $\mathcal{S}$. As a filtering function, we obtain $F(q) = \{s \in \mathcal{S} : \mathcal{N}_d(s) \cap \mathcal{N}_d(q) \neq \emptyset\}$.

Basic Data Structure. A static index data structure is generated in a precomputation phase that can be queried during an on-line phase. We insert a number

of values into a hash table that is part of our data structure. The structure utilizes the hash table to store pointers to originating dictionary entries at the hash values of residual strings. If any hash value has more than one originating dictionary entry then the corresponding pointers are stored in a list.

Query. For an input query q and maximum distance d, the corresponding d-neighborhood and its hash values are computed. If any element of the query's residuals is also an element of the data structure then the pointers to the originating dictionary entries give a set of candidates. Each of those might be an approximate match. Once the candidate set is completely built, it is searched exhaustively by computing the edit distance of each candidate to the query. By removing all elements from the candidate set whose distance is larger than the threshold d we get the set of all dictionary members that are at most a distance d away from query q. Perhaps there exists an additional order on the candidates stemming from the application. The algorithm can be adapted to not only return the best match, but also a list of those candidates that are sufficiently close.

Precomputation. We compute the d-neighborhood of each element of the input dictionary and insert the resulting information into our index data structure. Doing this precomputation naively and storing all residual strings in a data structure takes up an enormous amount of space. Instead, we use hashing and reduce each element of the residual neighborhood into an integer number. We insert pointers to the originating dictionary entries into the hash table at the respective hash values of all residual strings. Therefore, only constant space is needed per residual string regardless of the length of that string.

Algorithmic Generalization. We limit the number of elements that are inserted into the index while staying lossless. We split long input words in half, compute residual strings with half the number of errors, and adapt the query algorithm, which will be explained in this Section. See Section 3 for an analysis of threshold m, which indicates whether or not to split a word. Instead of generating $\binom{|s|}{d}$ hashes we insert only $\binom{|s|}{\lfloor \frac{d}{2} \rfloor} + \binom{|s|}{\lceil \frac{d}{2} \rceil}$ values for any dictionary entry s. The *generalized d-neighborhood* of $w' \in \Sigma^*$ is the set of residuals that is found by computing all combinations of $\lceil \frac{d}{2} \rceil$ deleted characters for both half of w'.

Index generation is simple. But we have to pay some extra care at query time, because insertions and deletions that transform words w into w' can take place at arbitrary positions. As a consequence, we can not rely on the length of a query q to decide whether it has been split or not. Instead of splitting a query q of length l at a fixed position, it is split several times in half at positions in the interval of $\lceil \frac{l}{2} \rceil \pm \lceil \frac{d}{2} \rceil$. Also, the allowed error is halved. If the length of an input word is within $m \pm d$ then the index is also searched for the non-split string.

Consider these definitions. Let $w \in \Sigma^*$ be an entry of dictionary D and d the maximum allowed error. Let $u = p(w)$ and $v = s(w)$ denote the first and second half of the split word w. Prefixes u and suffix v are indexed, while q is the query. Any query q is split at several positions as explained above and we define $\mathcal{P}(w)$ to be the set of first and $\mathcal{S}(w)$ to be the set of second halves. Our method is still

correct since we can show the existence of a common residual string for either the prefix or the suffix of a split query word. See the full paper for the proof.

3 Experimental Results

Implementation Details. We implemented the data structure, the construction and query algorithms in C++ using GCC Compiler version 4.3.2. We hashed all residual strings with the built-in hash function of the Boost library v1.36 to a 32-bit Integer, chained with a simple linear congruence. Our tests were ran on a single core of a Intel Xeon X5550 CPU, running a version 2.6.27 Linux kernel.

The exhaustive search of the candidate set is done by a simple implementation of the Levenshtein distance. It computes a band of width $2d + 1$ only. This way we compute the distance exactly only if it is smaller than d and return otherwise as soon as we get a certificate that the distance is larger than d. Since we need $O(1)$ to fill a cell in the distance table, we can verify a candidate in $O(d \cdot l)$, where l is the length of the shorter word. In the experiments it took less than a microsecond to verify any single candidate. We compare the performance of our optimizations against our own implementation for reasons of fairness.

Test Instances. The word list *mobydick* consists of the 37 924 distinct words (avg. length 9) from Melville's classic novel, the *town* dictionary consists of 47 339 German town names (avg. length 10) extracted from the OpenStreetMap project (`http://www.openstreetmap.org`) in February 2009, the *english* dictionary is an extract of words 213 557 (avg. length 10) from Webster's English Dictionary and the *wikipedia* dictionary is the list of 1 812 365 pairwise distinct words (avg. length 9) from all english Wikipedia (`http://www.wikipedia.org`) titles as of February 2009.

Index Performance. We analyzed the space and query performance of the index for varying values of the split parameter m and error $d = 0, \ldots, 4$. A distance of $d = 3$ is large for natural languages and larger d deliver matches that already look random. Thus, we are able to calibrate the split parameter. The preprocessing was run for all of our data sets and we averaged over 1 000 randomized queries for each m.

In the full paper we experimented with different values of the split parameter m. It turns out that choosing the average word length of the dictionary is a good choice. Splitting large words benefits preprocessing time as well. Memory consumption rises with the split parameter, while the query time decreases as expected. The split parameter functions as an adjusting value to choose between index size and query performance. The query performance is virtually unaffected. Query times rise sublinear with the size of the dictionary and are roughly proportional to the candidate size. For example, for $d = 2$ the query times on the wikipedia dictionary are roughly six times slower than on the mobydick dictionary, while the dictionary size is 45 times larger. See Table 1 for gives an overview of the numbers. See the full paper for an extensive experimental evaluation. When looking at our result and the original experiments of Bocek et al.

Table 1. *Mem* is the size of the index in [MiB], *proc* the preprocessing time in [*s*], *qry* the avg. query duration in microseconds, *cand* the avg. size of the candidate set.

	Mobydick				Town				English				Wikipedia			
d	mem	proc	qry	cand	mem	proc	qry	cand	mem	proc	qry	cand	mem	proc	qry	cand
0	0.25	0.061	2	1	0.46	0.156	0	2	2.36	0.886	0	1	14.41	7.131	1	1
1	1.33	0.320	5	5	1.79	0.576	8	9	8.55	3.450	8	6	55.84	32.287	34	25
2	4.57	1.272	84	61	6.91	2.483	99	99	30.49	12.596	122	46	170.79	107.289	502	702
3	9.78	4.044	553	606	15.18	7.458	644	613	61.37	36.309	644	502	342.18	270.506	7 019	9 900
4	16.09	14.647	2 974	3 376	27.20	28.144	7 250	3 720	105.75	117.970	7 543	4 520	603.35	922.521	$55{\cdot}10^3$	$65{\cdot}10^3$

Table 2. Comparison Against Existing Experiments and BK-tree, best results bold

	$m = \infty$	$m = 10$	Best of Bocek et al.	BK-tree
preprocessing [ms]	2 649	349	5 000 - 7500	**183**
avg. query [μs]	114	**18**	$100–200{\cdot}10^3$	935
dictionary size [MiB]	9.8	1.5	20	**0.25**

[9] in Table 2 we see that our implementation performs better by about an order of magnitude in all important areas. Although we know that our numbers were measured on different hardware, they give an impression on the performance. The experiments were run on the same random dictionary of 10 000 words. Note that the case of $m = \infty$ corresponds to Bocek et al.'s algorithm. They proposed several improvements that either perform fast or have low space consumption but not both at the same time. Since the results of the experiments are only available as plots we had to estimate the values. We did so in a benevolent way and compare the best of their values in each category against our implementation with and without splitting. We see one potential source of performance problems with our experiments as we tested on dictionaries with rather short words that have similar sizes. The higher the allowed error distance d is, the shorter residual strings get. This leads to longer index lists in the hash table, because it is more likely that two distinct words will have common residual strings. This also explains the larger number of candidates for higher values of d.

An experimental evaluation of BK-trees [11] and several variants reports on the sizes of the search space that is visited depending on the allowed error distance. Those experiments were done on a set of 100 000 English words and report on a nearly linear growth of the search space going up from 5% for edit distance 0 to slightly more than 40% for a distance of 4. The size of the visited search space in our experiments is always less than 1% and much less than the search space size for the best BK-tree variant [11]. We confirmed the high number of candidates with our own implementation. The number of candidates is high, i.e. more than 10^3 candidates for a query to the Wikipedia dictionary with $d = 2$.

4 Conclusions and Future Work

We improved a method for approximate string matching in a dictionary. We developed algorithmic optimizations that provide a tuning parameter to choose

between space consumption and running time while having overall lower preprocessing duration. Additionally, the performance has been validated experimentally by comparison against BK-trees and the baseline version of FastSS.

We see possibilities to speed up the verification of the candidate set using bit-parallelism [12] and SIMD instructions of current processors. However, only about half of the time of the algorithm is actually spent in the verification phase with the computation of the edit distance. Likewise there might be opportunities to speed up the precomputation, in particular, using fast, incremental computations of hash functions and using parallelization. Also, it might be interesting to use data compression techniques to further reduce the storage requirements.

References

1. Baeza-Yates, R., Navarro, G.: Fast approximate string matching in a dictionary. In: SPIRE (1998)
2. Burkhard, W.A., Keller, R.M.: Some approaches to best-match file searching. Commun. ACM 16, 230–236 (1973)
3. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. 33, 273–321 (2001)
4. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: 36th ACM Symposium on Theory of Computing (2004)
5. Russo, L.M.S., Navarro, G., Oliveira, A.L., Morales, P.: Approximate string matching with compressed indexes. Algorithms 2(3), 1105–1136 (2009)
6. Mihov, S., Schulz, K.U.: Fast approximate search in large dictionaries. Comput. Linguist. 30, 451–477 (2004)
7. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: Compressed indexes for approximate string matching. Algorithmica (2008)
8. Burkhardt, S., Kärkkäinen, J.: Better filtering with gapped q-grams. Fundamenta Informaticae (2001)
9. Bocek, T., Hunt, E., Stiller, B.: Fast similarity search in large dictionaries. Technical report, Universität Zürich (2007), http://fastss.csg.uzh.ch/
10. Mor, M., Fraenkel, A.S.: A hash code method for detecting and correcting spelling errors. ACM Commun. 25, 935–938 (1982)
11. Motwani, G., Nair, S.G.: Search efficiency in indexing structures for similarity searching. CoRR cs.DB/0403014 (2004)
12. Hyyrö, H., Fredriksson, K., Navarro, G.: Increased bit-parallelism for approximate string matching. ACM Journal of Experimental Algorithmics 10 (2005)

Training Parse Trees for Efficient VF Coding

Takashi Uemura[1], Satoshi Yoshida[1], Takuya Kida[1],
Tatsuya Asai[2], and Seishi Okamoto[2]

[1] Hokkaido University, Kita 14, Nishi 9, Kita-ku, Sapporo 060-0814, Japan
[2] Fujitsu Laboratories Ltd., 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan

Abstract. We address the problem of improving variable-length-to-fixed-length codes (VF codes), which have favourable properties for fast compressed pattern matching but moderate compression ratios. Compression ratio of VF codes depends on the parse tree that is used as a dictionary. We propose a method that trains a parse tree by scanning an input text repeatedly, and we show experimentally that it improves the compression ratio of VF codes rapidly to the level of state-of-the-art compression methods.

1 Introduction

From the viewpoint of speeding up pattern matching on compressed texts, *variable-length-to-fixed-length codes* (VF codes for short) have been reevaluated lately [3,4]. A VF code is a coding scheme that parses an input text into a consecutive sequence of substrings (called blocks) with a dictionary tree, which is called a parse tree, and then assigns a fixed length codeword to each substring. It is quite hard to construct the optimal parse tree that gives the best compression ratio for the input text, since it is equal to or more difficult than NP-complete [4].

Our concern is how to construct parse trees that approximate the optimal tree better. In most VF codes, a frequency of each substring of T is often used as a clue for the approximation, since it could be related to the number of occurrences in a sequence of parsed blocks. This gives a chicken and egg problem as Klein and Shapira stated in [4]; that is, to construct a better dictionary, which decides the partition of T, one has to estimate the number of entries that occurs in the partition.

In this paper we discuss about a method for training a parse tree of a VF code to improve its compression ratio. We propose an algorithm of reconstructing a parse tree based on the merit of each node. We employ a heuristic approach: scanning the input text for estimating the parse tree and then reconstructing it many times. We can control the number of repetition and also we can employ a random sampling technique to reduce the training time. We show experimentally that our method improves VF codes comparable to gzip and the others with a moderate sacrifice of compression time.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 179–184, 2010.

2 Variable-Length-to-Fixed-Length Codes

Let Σ be a finite alphabet. A VF code is a source coding that parses an input text $T \in \Sigma^*$ into a consecutive sequence of variable-length substrings and then assigns a fixed length codeword to each substring. We will describe the brief sketches of two VF codes below.

Tunstall code [7] is an optimal VF code (see also [6]) for a memory-less information source. It uses a parse tree called *Tunstall tree*, which is the optimal tree in the sense of maximizing the average block length. Tunstall tree is an ordered complete k-ary tree that each edge is labelled with a different symbol in Σ, where $k = |\Sigma|$. Let $\Pr(a)$ be an occurrence probability for source symbol $a \in \Sigma$. The probability of string $x_\mu \in \Sigma^+$, which is represented by the path from the root to leaf μ, is $\Pr(x_\mu) = \prod_{\eta \in \xi} \Pr(\eta)$, where ξ is the label sequence on the path from the root to μ (from now on we identify a node in $\mathcal{T}$ and a string represented by the node if no confusion occurs). Then, Tunstall tree $\mathcal{T}^*$ can be constructed as follows:

1. Initialize $\mathcal{T}^*$ as the ordered k-ary tree whose depth is 1, which consists of the root and its children; it has $k + 1$ nodes.
2. Repeat the following while the number of leaves in $\mathcal{T}^*$ is less than 2^k
   (a) Select a leaf v that has a maximum probability among all leaves in $\mathcal{T}^*$.
   (b) Make v be an internal node by adding k children onto v.

A Suffix Tree based VF code [3,4] (STVF code for short[1]) is a coding that constructs a suitable parse tree for the input text by using a suffix tree [8], which is a well-known index structure that stores all substrings and their frequencies in the target text compactly. In STVF codes, a suffix tree for the input text is constructed at first, and then the frequencies of all nodes are precomputed. Since the suffix tree for the input text includes the text itself, the whole tree can not be used as a parse tree. We have to prune it with some frequency-based heuristics to make a compact and efficient parse tree.

We outline the algorithm of constructing the parse tree. The algorithm starts with the initial parse tree that contains the root and its k children in the suffix tree. Then, it repeats choosing a node whose frequency is the highest in the suffix tree but not yet in the parse tree, and putting it into the parse tree. The construction algorithm extends the parse tree on a node-by-node basis.

An internal node u in the parse tree is said to be *complete* if the parse tree contains all the children of u in the suffix tree. We do not need to assign a codeword to any complete node, since the encoding process never fail its traversals at a complete node. In Tunstall codes and the original STVF codes, all the internal nodes are complete; only leaves are assigned codewords. An idea of improving VF codes is to include incomplete nodes in the parse tree, but we have to modify the coding process so that it works in a non-instantaneous way. We omit the detail of the modified coding process for lack of space.

[1] Strictly, the methods of [3] and [4] are slightly different in detail. However, we call them the same name here since the key idea is the same.

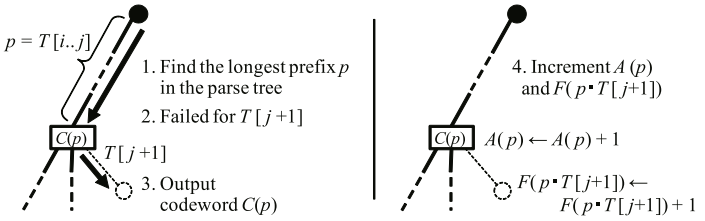


Fig. 1. An example of computing accept counts and failure counts

3 Training Parse Trees

In this section, we present a reconstruction algorithm for a ready-made parse tree to improve the compression ratio. The basic idea is to exchange useless strings in the current parse tree as a result for the other strings that are expected to be frequently used.

We define two measures for evaluating strings. For any string s in the parse tree, the *accept count* of s, denoted by $A(s)$, is defined as the number of that s was used in the encoding. For any string t that is not assigned a codeword, the *failure count* of t, denoted by $F(t)$, is defined as the number of that the prefix $t[1..|t|-1]$ of t was used but the codeword traversal failed at the last character of t. That is, $F(t)$ suggests how often t likely be used if t is in the parse tree. We can embed the computations of $A(s)$ and $F(t)$ in the encoding procedure. When $p = T[i..j]$ is parsed in the encoding, $A(p)$ and $F(p \cdot T[j+1])$ are incremented by one. Figure 1 shows an example of computing these measures.

Comparing the minimum of $A(s)$ and the maximum of $F(t)$, the reconstruction algorithm repeats to exchange s and t if the former is less than the latter; it removes s from the parse tree and enter t instead.

To train a parse tree we apply the algorithm many times. For each iteration, it first encodes the input text with the current parse tree. Next, it evaluates the contribution of each string in the parse tree, and then exchanges some infrequent strings for the other promising strings.

4 Experimental Results

We have implemented Tunstall coding and STVF coding with training approach that we stated in Sec. 3, and compared them with BPEX [5][2], ETDC [2], SCDC [1], gzip, and bzip2. Although ETDC/SCDC are variable-to-variable length codes, their codewords are byte-oriented and designed for compressed pattern matching. We chose 16 as the codeword lengths of both STVF coding and Tunstall coding. Our programs are written in C++ and compiled by g++ of GNU, version 3.4. We ran our experiments on an Intel Xeon (R) 3 GHz and 12 GB of RAM, running Red Hat Enterprise Linux ES Release 4.

[2] This name comes from the program implemented by Maruyama.

Table 1. About text files to be used

Texts	size(byte)	$\|\Sigma\|$	Contents
GBHTG119	87,173,787	4	DNA sequences
DBLP2003	90,510,236	97	XML data
Reuters-21578	18,805,335	103	English texts
Mainichi1991	78,911,178	256	Japanese texts (encoded by UTF-16)

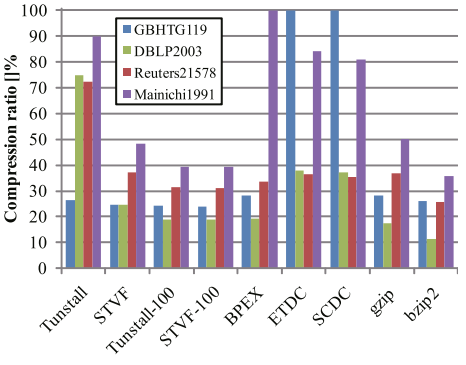


Fig. 2. Compression ratios

We used DNA data, XML data, English texts, and Japanese texts to be compressed (see Table 1). GBHTG119 is a collection of DNA sequences from GenBank[3], which is eliminated all meta data, spaces, and line feeds. DBLP2003 consists of all the data in 2003 from dblp20040213.xml[4]. Reuters-21578(distribution 1.0)[5] is a test collection of English texts. Mainichi1991[6] is from Japanese news paper, *Mainichi-Shinbun*, in 1991.

4.1 Compression Ratios and Speeds

The methods we tested are the following nine: Tunstall (Tunstall codes without training), STVF (STVF codes without training), Tunstall-100 (Tunstall codes with 100 times training), STVF-100 (STVF codes with 100 times training), BPEX, ETDC, SCDC, gzip, and bzip2. Figure 2 shows the results of compression ratios, where every compression ratio includes dictionary informations. We measured the averages of ten executions for Tunstall-100 and STVF-100.

For GBHTG119, STVF, Tunstall-100, and STVF-100 were the best in the compression ratio comparisons. Since ETDC and SCDC are word-based compression, they could not work well for the data that are hard to parse, such as DNA sequences and Unicode texts. Note that, while Tunstall had no advantage to STVF,

[3] http://www.ncbi.nlm.nih.gov/genbank/
[4] http://www.informatik.uni-trier.de/~ley/db/
[5] http://www.daviddlewis.com/resources/testcollections/reuters21578/
[6] http://www.nichigai.co.jp/sales/corpus.html

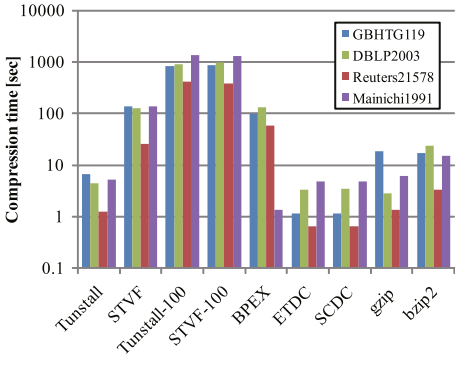


Fig. 3. Compression times. Note that the vertical axis is logarithmic scale.

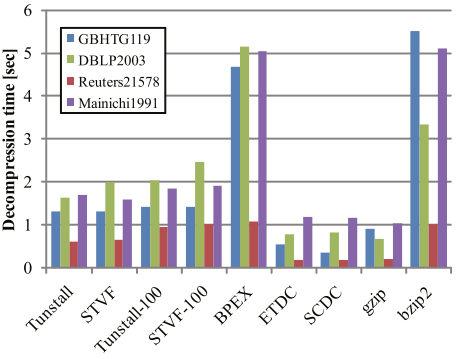


Fig. 4. Decoding times

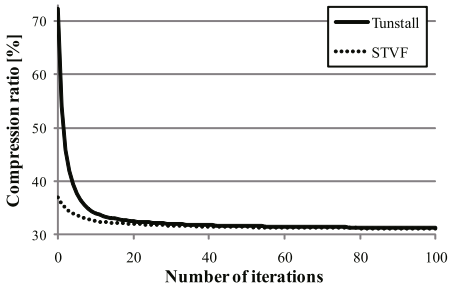


Fig. 5. The effects of training

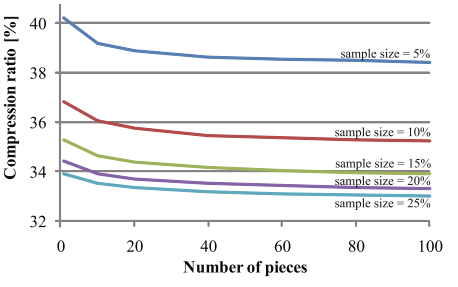


Fig. 6. Training with sampling

Tunstall-100 gave almost the same performance with STVF-100. Moreover, those were between gzip and bzip2.

Figure 3 shows the results of compression times. STVF was much slower than Tunstall and ETDC/SCDC since it takes much time for constructing a suffix tree. As Tunstall-100 and STVF-100 took extra time for training, they were the slowest among all for any dataset.

Figure 4 shows the results of decompression times. Tunstall and STVF were between BPEX and ETDC/SCDC in all the data. Tunstall-100 and STVF-100 became slightly slow.

4.2 Effects of Training

We examined how many times we should apply the reconstruction algorithm for sufficient training. We chose Reuter21578 as the test data in the experiments. Figure 5 shows the results of the effect of training for STVF and Tunstall. We can see that both compression ratios were improved rapidly as the number k of iterations increases. We can also see that they seem to come close asymptotically to the same limit, which is about 32%.

4.3 Speeding-Up by Sampling

In this experiment, we introduce a random sampling technique to save the training time.

Let T be the input text, m be the number of pieces, and B be the length of a piece. For given $m \geq 1$ and $B \geq 1$, we generate a sample text S from T at every iteration as $S = s_1 \cdots s_m$ ($s_k = T[i_k..i_k + B - 1]$ for $1 \leq k \leq m$), where $1 \leq i_k \leq |T| - B + 1$ is a start position of a piece s_k that we select in a uniform random manner for each k. Then, $|S| = mB$.

Figure 6 shows the compression ratios for Tunstall codes with 20 times training. We measured the average of 100 executions for each result. We observed that the compression ratio achieves almost the same limit when the sampling size $|S|$ is 25% of the text and the number m of pieces is 100. Compared with BPEX, Tunstall codes with training overcome in compression ratios when $|S|$ is 20% and $m = 40$. The average compression time at that point was 30.97 seconds, while that of BPEX was 58.77 seconds. Although STVF codes are better than Tunstall codes in compression ratios, it revealed that Tunstall codes with training are also useful from the viewpoint of compression time.

Acknowledgements

This work was partly supported by a Grant-in-Aid for JSPS Fellows (KAKENHI:21002025) and a Grant-in-Aid for Young Scientists (KAKENHI:20700001) of JSPS.

References

1. Brisaboa, N.R., Fariña, A., Navarro, G., Esteller, M.F.: (s, c)-dense coding: An optimized compression code for natural language text databases. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 122–136. Springer, Heidelberg (2003)
2. Brisaboa, N.R., Iglesias, E.L., Navarro, G., Paramá, J.R.: An efficient compression code for text databases. In: Sebastiani, F. (ed.) ECIR 2003. LNCS, vol. 2633, pp. 468–481. Springer, Heidelberg (2003)
3. Kida, T.: Suffix tree based VF-coding for compressed pattern matching. In: Data Compression Conference 2009, p. 449. IEEE Computer Society, Los Alamitos (March 2009)
4. 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)
5. Maruyama, S., Tanaka, Y., Sakamoto, H., Takeda, M.: Context-sensitive grammar transform: Compression and pattern matching. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 27–38. Springer, Heidelberg (2008)
6. Savari, S.A.: Variable-to-fixed length codes for predictable sources. In: Data Compression Conference 1998, pp. 481–490. IEEE Computer Society, Los Alamitos (1998)
7. Tunstall, B.P.: Synthesis of noiseless compression codes. Ph.D. thesis, Georgia Inst. Technol., Atlanta, GA (1967)
8. Weiner, P.: Linear pattern matching algorithms. In: 14th IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

Algorithms for Finding a Minimum Repetition Representation of a String

Atsuyoshi Nakamura[1], Tomoya Saito[1], Ichigaku Takigawa[2], Hiroshi Mamitsuka[2], and Mineichi Kudo[1]

[1] Hokkaido University, Kita 14, Nishi 9, Kita-ku, Sapporo 060-0814, Japan
{atsu@main.,saito@,mine@main.}ist.hokudai.ac.jp
[2] Institute for Chemical Research, Kyoto University, Uji, Kyoto 611-0011, Japan
{takigawa,mami}@kuicr.kyoto-u.ac.jp

Abstract. A string with many repetitions can be written compactly by replacing h-fold contiguous repetitions of substring r with $(r)^h$. We refer to such a compact representation as a *repetition representation string* or RRS, by which a set of disjoint or nested tandem arrays can be compacted. In this paper, we study the problem of finding a *minimum RRS* or MRRS, where the size of an RRS is defined to be the sum of its component letter sizes and the sizes needed to describe the repetitions $(\cdot)^h$ which are defined as $w_R(h)$ using a repetition weight function w_R. We develop two dynamic programming algorithms to solve the problem. One is CMR that works for any repetition weight function, and the other is CMR-C that is faster but can be applied only when the repetition weight function is constant. CMR-C is an $O(w(n+z))$-time algorithm using $O(n+z)$ space for a given string with length n, where w and z are the number of distinct primitive tandem repeats and the number of their occurrences, respectively. Since $w = O(n)$ and $z = O(n \log n)$ in the worst case, CMR-C is an $O(n^2 \log n)$-time $O(n \log n)$-space algorithm, which is faster than CMR by $((\log n)/n)$-factor.

Keywords: tandem repeat, string algorithm.

1 Introduction

A contiguous repeat of a substring embedded in a string is called a *tandem array*, or a *tandem repeat* (or a *square*) for a two-fold repetition. The problem of finding tandem arrays and repeats has been studied for more than two decades in the fields of computer science, mathematics and biology [4]. Efficient algorithms for finding *all* or *primitive* tandem repeats, namely, repeats whose repeated units themselves are not tandem arrays, have been developed so far [2,3,4].

In this paper, we consider *repetition representation strings*, or RRSs, which allows to use notations $(r)^h$ instead of contiguous sequences $rr \cdots r$ of h identical strings r. An RRS can represent many tandem arrays simultaneously but cannot always represent all tandem arrays in a string. An RRS is a string representation for a set of *disjoint* or *nested* tandem arrays in a string, where two substrings of

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 185–190, 2010.

a string are disjoint if they have no intersection, and $(r)^h$ and $(r')^{h'}$ are nested if r contains $(r')^{h'}$. Different RRSs are possible for a string depending on which set of tandem arrays is represented. As an evaluation measure for RRSs, we use their sizes, that is, we consider the problem of finding a *minimum RRS* (MRRS) for a given string. Our expectation that an MRRS represents an essential repetition structure of a string is supported by the well-known MDL principle: "the success in finding regularities can be measured by the length with which the data can be described" (www.mdl-research.org).

We define the size of an RRS as the sum of its component sizes, namely, the sum of its component letter sizes and the sizes needed to describe the repetitions $(\cdot)^h$ which are defined as $w_R(h)$ using a repetition weight function w_R.

We developed two algorithms: CMR and CMR-C. CMR works for any repetition weight function w_R, though it is slow; it runs in $O(n^3)$ time and $O(n^2)$ space. On the other hand, CMR-C works only when the repetition weight function is constant, but it is faster; it is an $O(w(n+z))$-time $O(n+z)$-space algorithm using a sophisticated technique of finding tandem repeats. Here, n is the length of a given string, z is the number of (occurrences of) primitive tandem repeats in the string, and w is the number of distinct strings among them. Considering the worst case setting [1,2]: $w = O(n)$ and $z = O(n \log n)$, these complexities correspond to $O(n^2 \log n)$ time and $O(n \log n)$ space, which are smaller than those of CMR by $((\log n)/n)$-factor. We further note that CMR-C is more effective for smaller w and z.

2 Problem Setting

Let Σ be a finite *alphabet*, whose elements are called *letters*. A *string* s is a sequence of letters with finite length. The length of string s is the number of letters in s which is denoted by $|s|$. More general notion called a *repetition representation string*, or an RRS, is defined as follows. First, all strings are RRSs themselves. If r_1 and r_2 are RRSs, then $r_1 r_2$, concatenation of r_1 and r_2, is also an RRS. If r is an RRS, then $(r)^h$ $(h \geq 2)$ is also an RRS that is another representation of a concatenation $rr \cdots r$ of h identical RRSs r. Note that parentheses '(', ')' and codes for numbers are special symbols not contained in Σ. A substring represented by RRS $(r)^h$ is called a *tandem array*, and it is called a *tandem repeat* when $h = 2$. We say that $(r)^h$ is *expanded* to the concatenation $rr \cdots r$ of h identical RRSs r and reversely $rr \cdots r$ is *reduced* to $(r)^h$. String s without $(\cdot)^h$ notation is said to be *represented by an RRS* r if s is obtained by expanding all the $(\cdot)^h$ in r for any h.

The *size* of an RRS can be calculated using given two non-negative weight functions, *alphabet weight function* w_Σ on Σ and *repetition weight function* w_R on the set of natural numbers at least 2. Given w_Σ and w_R, the size $l(r)$ of an RRS r is recursively defined as follows:

$$
\begin{aligned}
l(\lambda) &= 0 \text{ for empty string } \lambda,\ \ l(a) = w_\Sigma(a) \text{ for } a \in \Sigma, \\
l(r_1 r_2) &= l(r_1) + l(r_2) \text{ for all RRSs } r_1 \text{ and } r_2, \text{ and} \\
l((r)^h) &= l(r) + w_R(h).
\end{aligned}
$$

Remark 1. In most cases, we use constant functions as w_Σ and w_R for simplicity. A constant function w_R in the uniform cost model and $w_R(h) = \log h + c$ in the logarithmic cost model seem natural, where c is a some constant. In practice, w_Σ and w_R should be decided depending on applications.

The problem we deal with in this paper is the following one.

Problem 1. Given a string s, an alphabet weight function w_Σ and a repetition weight function w_R, find a minimum(-sized) RRS r that represents s.

3 Algorithms

In this section, we show algorithms for problem 1. We assume that s is an arbitrary string $a_1 a_2 \cdots a_n$ with length n. For $1 \leq i \leq j \leq n$, we let $s[i..j]$ denote the substring $a_i a_{i+1} \cdots a_j$ of s, and let $r_*[i..j]$ denote a minimum RRS (MRRS) representing $s[i..j]$ in the following subsections.

3.1 General Algorithm

First, we show algorithm CMR that works for any repetition weight function w_R. CMR calculates an MRRS representing a given string $s[1..n]$ by calculating MRRSs representing $s[i..j]$ for all the substrings $s[i..j]$ of $s[1..n]$ using dynamic programming. Its dynamic programming is based on the following proposition, which says that an MRRS representing a string is either a repetition of an MRRS representing its some prefix or a concatenation of two MRRSs representing its two substrings that are made by cutting the string at some position.

Proposition 1. *For all $1 \leq i < j \leq n$, $r_*[i..j]$ is one of the followings:*

(1) $(r_*[i..i+d-1])^h$ for some $d \geq 1$ and $h \geq 2$ with $hd = j - i + 1$,
(2) $r_*[i..i+d]r_*[i+d+1, j]$ for some $0 \leq d < j - i$.

For given $1 \leq i < j \leq n$, assume that $r_*[i'..j']$ is already known for all $1 \leq i' < j' \leq n$ with $j' - i' < j - i$. By the proposition, $r_*[i..j]$ is obtainable by searching an MRRS among at most $\lfloor (j - i + 1)/2 \rfloor$ possibilities of (1) and $j - i$ possibilities of (2). If $l(r_*[i'..j'])$ for all $1 \leq i' < j' \leq n$ with $j' - i' < j - i$ is already calculated, the size for each possible RRS can be calculated in constant time. Thus, such search can be finished in $O(j - i)$ time. This means that an MRRS $r_*[1..n]$ for a given string $s[1..n]$ is obtainable in $O(n^3)$ time and $O(n^2)$ space by dynamic programming using the fact that $r_*[i..i] = s[i..i] = a_i$ for all $1 \leq i \leq n$.

3.2 Algorithm for a Constant Repetition Weight Function

In this subsection, we describe an algorithm faster than CMR in the case with a constant repetition weight function. In this subsection, w_R is always supposed to be a constant function.

First, note that

$$l((\underbrace{rr\cdots r}_{h \text{ times}})^g) = hl(r) + w_R(g) \geq l(r) + w_R(hg) = l((r)^{hg})$$

because w_R is supposed to be a constant function. This means that we only have to check combinations of *primitive* tandem arrays, where a tandem array $(r)^h$ is said to be primitive if r cannot be represented by RRS $(r')^{h'}$ for any string r' and $h' \geq 2$. Such a tandem array is called a *primitive* tandem repeat when $h = 2$. It is known that there are at most $O(n \log n)$ *occurrences* of primitive tandem repeats in a string of length n.

Let $E(i)$ denote the set of primitive tandem repeats ending at the ith letter of s. For each tandem repeat $s[i-2j+1..i]$ in $E(i)$, namely, for each tandem repeat with width j and ending at i, $h_*(i,j)$ denotes an optimal number of repetitions of the last j letters as an RRS-representation of $s[1..i]$:

$$h_*(i,j) \stackrel{\text{def}}{=} \arg\min_g \{l(r(g)) : \mu(r(g)) = s[1..i], \\ r(g) = r_*[1..i-gj](r_*[i-j+1..i])^g\},$$

where $\mu(r)$ denotes an expanded string of r. Then, $r_*[1..i-h_*(i,j)j](r_*[i-j+1..i])^{h_*(i,j)}$ is a minimum RRS representing $s[1..i]$ among the RRSs of the form $r_*[1..i-gj](r_*[i-j+1..i])^g$ $(g \geq 2)$ by the definition, and also a minimum RRS even among all the RRSs of the form $r[1..i-gj](r[i-j+1..i])^g$ $(g \geq 2)$ for any RRSs $r[1..i-gj]$ and $r[i-j+1..i]$ representing $s[1..i-gj]$ and $s[i-j+1..i]$, respectively. Fortunately, $h_*(i,j)$ can be calculated efficiently using dynamic programming.

The following proposition indicates that candidate RRSs for $r_*[1..i]$ can be narrowed to only $|E(i)|+1$ RRSs.

Proposition 2. *One of the RRSs of the following form (1) or (2) is an MRRS representing* $s[1..i]$.

(1) $r_*[1..i-1]a_i$,
(2) $r_*[1..i-h_*(i,j)j](r_*[i-j+1..i])^{h_*(i,j)}$ *for* $s[i-2j+1..i] \in E(i)$.

By the above propositions, $r_*[1..n]$ can be calculated using dynamic programming if an appropriate representation of $E(i)$ is prepared. We call such a dynamic programming algorithm FindBestComb. As an input of FindBestComb, $E(i)$ is assumed to be given by a linked list, where the list entry for tandem repeat $s[i-2j+1..i]$ has the repeat width j, an MRRS $r_*[i-j+1..i]$ and its size, and the pointer to the entry for the tandem repeat $s[i-3j+1..i-j]$. FindBestComb calculates $l(r_*[1..i])$ in the increasing order of i. For each i, it searches an MRRS $r_*[1..i]$ among $|E(i)|+1$ candidates shown in Proposition 2 using already calculated $l(r_*[1..i'])$ for $i' < i$. Note that $h_*(i,j)$ can be calculated using already calculated $h_*(i-j,j)$, which can be accessed through the pointer to the entry for the repeat $s[i-3j+1..i-j]$. An MRRS $r_*[1..n]$ can be constructed by traceback if which one among $|E(i)|+1$ candidates is an MRRS $r_*[1..i]$ is memorized for

CMR-C %[Calculate an MRRS of a string for a Constant repetition weight function]
Input: $s[1..n]$: string,
w_Σ: alphabet weight function, w_R: constant repetition weight function
Output: $r_*[1..n]$: an MRRS representing $s[1..n]$

Step 1 Make the suffix tree T decorated with the endpoints of all primitive tandem repeats in the vocabulary for the reversed string of s using the algorithm developed by Gusfield and Stoye [2].
Step 2 Prepare linked list $E(i)$ for $i = 1, ..., n$ by executing Prepare_E(T's root node).
Step 3 For each entry of tandem repeat $s[i - 2j + 1..i]$ in $E(i)$ $(i = 1, 2, ..., n)$, set an MRRS and its size of $s[i - j + 1..i]$ to the entry by executing Set_Min_Size($s[1..n], w_\Sigma, w_R, E$).
Step 4 Calculate $r_*[1..n]$ by executing FindBestComb($s[1..n], w_\Sigma, w_R, E$).

Fig. 1. Algorithm CMR-C

each $i = 1, 2, ..n$. The time and space complexity of FindBestComb is $O(n \log n)$ because the number of occurrences of primitive tandem repeats is $O(n \log n)$.

Linked lists $E(i)$ for all $i = 1, ..., n$ and the pointers to the entries for tandem repeats $s[i-3j+1..i-j]$ which are set for all entries of primitive tandem repeat $s[i - 2j + 1..i]$ in $E(i)$ $(i = 1, ..., n)$ are calculated in $O(n \log n)$ time using the suffix tree of s decorated with the endpoints of all primitive tandem repeats in the vocabulary, which can be constructed in $O(n)$ time by the algorithm developed by Gusfield and Stoye [2]. The calculation can be done by postorder traversal of the decorated suffix tree for the *reversed string* of s. We call such a recursive algorithm Prepare_E. We also let each entry in $E(i)$ have a pointer to the entry for the leftmost occurrence of the same type tandem repeat, which is also set in algorithm Prepare_E. This pointer is used to save computational cost for occurrences of the same type tandem repeat as described below. Since the number of occurrences of primitive tandem repeats are $O(n \log n)$, the above procedure is done in $O(n \log n)$ time and $O(n \log n)$ space.

There is still one thing we have to do before executing FindBestComb for the whole string. An MRRS $r_*[i - j + 1, ..i]$ and its size $l(r_*[i - j + 1..i])$ in each entry for tandem repeat $s[i - 2j + 1..i]$ must be calculated. There are $O(n \log n)$ occurrences of primitive tandem repeats but MRRSs and their size have to be calculated only for distinct tandem repeats, namely, tandem repeats in the *vocabulary* (the set of different repeat types) [2], the number of which is $O(n)$. Applying FindBestComb to $s[i - j + 1..i]$ for each primitive tandem repeat $s[i - 2j + 1..i]$ in the vocabulary, an MRRS $r_*[i - j + 1..i]$ and its size $l(r_*[i - j + 1..i])$ can be calculated in $O(n \log n)$ time and $O(n \log n)$ space, so the minimum size for all tandem repeats in the vocabulary can be obtained in $O(n^2 \log n)$ time and $O(n \log n)$ space. We call the algorithm for this task Set_Min_Size.

Putting all together, algorithm CMR-C shown in Fig. 1, an algorithm finding an MRRS representing $s[1..n]$, is obtained.

By the argument so far, we have proved the following theorem.

Theorem 1. *$r_*[1..n]$ can be calculated in $O(n^2 \log n)$ time and $O(n \log n)$ space.*

If the number of occurrences of primitive tandem repeats is z, FindBestComb and Prepare_E can be executed in $O(n+z)$ time and $O(n+z)$ space. Furthermore, if the number of primitive tandem repeats in the vocabulary is w, Set_Min_Len can be executed in $O(w(n+z))$ time and $O(n+z)$ space. Thus, the following corollary holds.

Corollary 1. *$r_*[1..n]$ can be calculated in $O(w(n+z))$ time and $O(n+z)$ space when $w > 0$, where w and z is the number of primitive tandem repeats in the vocabulary and the number of their occurrences, respectively.*

4 Concluding Remarks

In this paper, we considered representations called an MRRS for a string, in which a set of disjoint or nested contiguous repeats are compacted, and proposed dynamic programming algorithms CMR and CMR-C to calculate it for a given string. CMR-C was theoretically proved to run in $O(w(n+z))$ time and $O(n+z)$ space, which indicates that it is fast if the number of distinct tandem repeat w and the number of its occurrences z are not large compared to the string length n. According to our experiments on DELL Precision T7500 (cpu: Intel(R) Xeon(R) E5520 [2.27GHz], memory: 2GB), CMC-R is fast enough for large-scale, synthetic datasets on strings; for a random binary string of 1,638,400 letters, 8.2 seconds were enough for CMR-C to find an MRRS. The size of MRRS can be a measure of how well a string is organized in terms of repeated structures. From our experimental result of such a structural complexity analysis applied to DNA sequences or chromosomes of the most major nine species (A. thaliana, C. elegans, zebrafish, fruit fly, chicken, human, mouse, yeast and rat), we found that the MRRS size was unique for each species.

Acknowledgements

This work was partially supported by JSPS KAKENHI 21500128 and the Collaborative Research Program of Institute for Chemical Research, Kyoto University (grant 2010-20).

References

1. Fraenkel, A., Simpson, J.: The exact number of squares in Fibonacci words. Theoretical Computer Science 218, 95–106 (1999)
2. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. Journal of Computer and System Sciences 69, 525–546 (2004)
3. Main, M., Lorentz, R.: An $O(n \log n)$ algorithm for finding all repetitions in a string. Journal of Algorithms 5, 422–432 (1984)
4. Stoye, J., Gusfield, D.: Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree. Theoretical Computer Science 270, 843–856 (2002)

Faster Compressed Dictionary Matching*

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

[1] National Tsing Hua University, Taiwan
{wkhon,thku}@cs.nthu.edu.tw
[2] Louisiana State University, Louisiana, USA
{rahul,svt}@csc.lsu.edu
[3] The University of Kansas, Kansas, USA
jsv@ku.edu

Abstract. Given a set $\mathcal{D}$ of d patterns, the dictionary matching problem is to index $\mathcal{D}$ such that for any query text T, we can locate the occurrences of any pattern within T efficiently. When $\mathcal{D}$ contains a total of n characters drawn from an alphabet of size σ, Hon et al. (2008) gave an $nH_k(\mathcal{D}) + o(n \log \sigma)$-bit index which supports a query in $O(|T|(\log^{\epsilon} n + \log d) + occ)$ time, where $\epsilon > 0$ and $H_k(\mathcal{D})$ denotes the kth order entropy of $\mathcal{D}$. Very recently, Belazzougui (2010) proposed an elegant scheme, which takes $n \log \sigma + O(n)$ bits of index space and supports a query in optimal $O(|T| + occ)$ time. In this paper, we provide connections between Belazzougui's index and the XBW compression of Ferragina et al. (2005), and show that Belazzougui's index can be slightly modified to be stored in $nH_k(\mathcal{D}) + O(n)$ bits, while query time remains optimal; this improves the compressed index by Hon et al. (2008) in both space and time.

1 Introduction

Given a set $\mathcal{D}$ of d patterns, the dictionary matching problem is to index $\mathcal{D}$ such that for any query text T, we can locate the occurrences of any pattern within T efficiently. Such a query is called the *dictionary matching query*. This problem is well-studied [1,2,5,12,16,3], and finds applications in computer virus detection and bio-informatics.

When $\mathcal{D}$ contains a total of n characters drawn from an alphabet of size σ, Aho and Corasick [1] proposed a data structure, now popularly known as the *Aho-Corasick automaton*, which supports the dictionary matching query in $O(|T|+occ)$ time. The automaton consists of a trie structure with $t \leq n+1$ nodes, and can be stored in $O(t \log t)$ bits of space. Alternatively, we may also store the generalized suffix tree [14,17] for the patterns in $\mathcal{D}$, so that the query time remains $O(|T| + occ)$, while the space becomes $O(n \log n)$ bits. Recent research focussed on reducing the index space for this problem. Hon et al. [12] gave an $nH_k(\mathcal{D}) + o(n \log \sigma)$-bit index which supports a query in $O(|T|(\log^{\epsilon} n + \log d) +$

* This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082 and US NSF Grants CCF-1017623 and CCF-0621457.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 191–200, 2010.

occ) time, where $\epsilon > 0$ and $H_k(\mathcal{D})$ denotes the kth order entropy of the text formed by concatenating all the patterns in $\mathcal{D}$. Very recently, Belazzougui [3] proposed an elegant scheme, which takes $tH_0(\mathcal{D}) + O(t)$ bits of index space and supports a query in $O(|T| + occ)$ time without slowdown. Note that $H_k(\mathcal{D}) \leq H_0(\mathcal{D}) \leq \log \sigma$.

In this paper, we provide connections between Belazzougui's index and the XBW compression of Ferragina et al. [6], and show that Belazzougui's index can be slightly modified to be stored in $tH_k(\mathcal{D}) + O(t)$ bits, while query time remains $O(|T| + occ)$; this improves the compressed index by Hon et al. [12] in both space and time. Note that the achieved space bound is the same as that required by performing front coding [18] of the patterns with a subsequent kth-order entropy compression, which is likely to be smaller than the usual $nH_k(\mathcal{D}) + O(n)$ bits obtained by compressing each pattern independently.

The organization of the paper is as follows. In Section 2, we give a review of the Burrows-Wheeler transformation [4] and the XBW compression of Ferragina et al. [6]. Then in Section 3, we solve a related problem called *prefix matching* based on the XBW compression. Section 4 describes how to directly apply the result in Section 3 to obtain our index for the dictionary matching problem. We conclude the paper in Section 5.

2 Preliminaries

2.1 A Review of BWT

Let $T[1..n]$ be a text. The *Burrows-Wheeler transform* (BWT) of T is a permutation of the characters in T, such that the location of $T[i]$ is determined by the rank of the suffix $T[i+1..n]$ among all suffixes of T. (Here, we assume the rank of the empty suffix, $T[n+1..n]$, is 0.) For example, consider Figure 1 which shows a text $T =$ `cababaac`. The rank of each suffix is marked by the node to the left of the suffix. By storing the character preceding the ith smallest suffix in increasing order of i, we obtain the BWT of T.

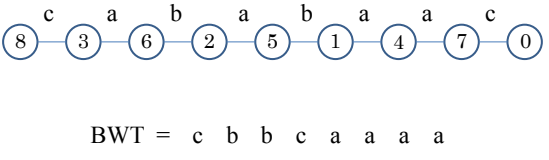


Fig. 1. An example of the Burrows-Wheeler transform

Ferragina and Manzini [8] showed that we can store BWT of T with some auxiliary data structures in $nH_k(T) + O(n)$ bits space, such that we can locate the occurrences of any pattern $P[1..p]$ within T efficiently. Basically, the searching algorithm first identifies all locations in T that match with $P[p..p]$, and then

iteratively identifies the locations matching $P[i..p]$ based on the locations that match with $P[i+1..p]$. For example, consider Figure 2 which shows how to search $P = \texttt{aba}$ in the text T of Figure 1. The searching algorithm starts by finding the locations that match with "a", then the locations that match with "ba", and finally the locations that match with "aba". Note that in each step, the ranks of the matching locations (highlighted nodes) form a contiguous range. The searching algorithm makes use of this fact to *implicitly* represent the matches, and then to compute the desired range of the matches in the next step.

It is shown in [9] that if the alphabet size σ is polylog(n), each step can be done in $O(1)$ time. In general, each step can be performed in $O(\log \sigma / \log \log n)$ time.

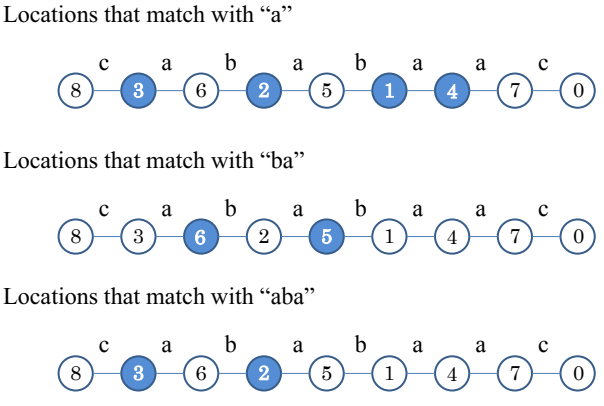


Fig. 2. An example of searching with BWT

2.2 A Review of XBW

Ferragina et al. [6] proposed a generalization of the BWT that is capable for encoding rooted tries. Given a trie $\mathcal{T}$, we encode the tree structure and the edge labels separately, where the latter are stored as a text string XBW analogous to the BWT. Precisely, for each node u in the trie, we define a string r_u, called the *reverse prefix string* of u, formed by concatenating the characters along the path from u to the root. Then each node u is associated with the rank of s_u among all the reverse prefix strings in the trie. Finally, the label on the edge (u, v) is stored in XBW text in (monotonic) increasing order of the rank of r_u. For example, consider Figure 3 which shows a trie. The rank of the reverse prefix of each node is marked inside the node. By storing the character(s) succeeding the node with the ith smallest reverse prefix in increasing order of i, we obtain the XBW of trie. Note that when the trie contains exactly one path, XBW is equivalent to BWT.

Ferragina et al. [6] also extended the notion of entropy compression on a text T to a trie $\mathcal{T}$. For each string ρ, we define cover$[\rho]$ to be the string formed by

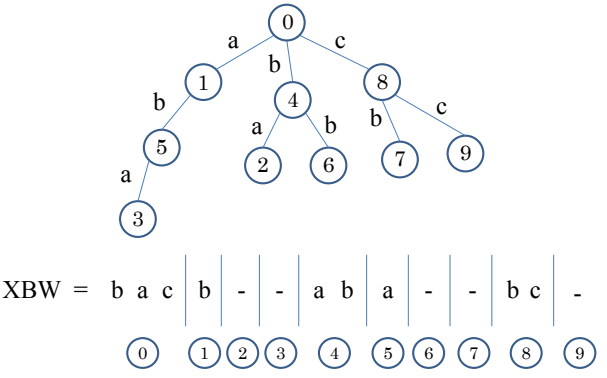


Fig. 3. An example of the XBW transform

concatenating the labels on the edges (u, v) such that r_u is prefixed by ρ. For instance, in the example of Figure 3, cover[b] = aba$--$ and cover[ba] = a (Note that the order of characters among the cover is irrelevant). The kth-order entropy of the trie $\mathcal{T}$, denoted by $H_k(\mathcal{T})$, is defined as:

$$H_k(\mathcal{T}) \equiv \frac{1}{t} \sum_{\text{length-}k \text{ string } \rho} |\text{cover}[\rho]| H_0(\text{cover}[\rho]),$$

where t is the number of nodes in $\mathcal{T}$, and $H_0(s)$ for a string s is the standard zeroth-order entropy of s.[1]

Then, Ferragina et al. showed that we can store XBW of $\mathcal{T}$ with some auxiliary data structures in $tH_k(\mathcal{T}) + O(t)$ bits space, such that for any pattern $P[1..p]$, we can locate all the subpaths (ancestor-descendent paths) within $\mathcal{T}$ efficiently. The searching algorithm is analogous to that of searching BWT, where it first identifies all locations of the subpaths that match with $P[1]$, and then iteratively identifies the locations of the subpaths matching $P[1..i+1]$ based on the locations corresponding to $P[1..i]$.[2]

In each step, the ranks of the matching locations form a contiguous range. Again, we can make use of this fact to *implicitly* represent the matches, and then to compute the desired range of the matches in the next step. It is shown that if the alphabet size σ is polylog(n), each step can be done in $O(1)$ time. For larger alphabet size, each step can be performed in $o((\log\log\sigma)^{1+\epsilon})$ time, for any fixed $\epsilon > 0$ [7].

[1] $H_0(s) \equiv (1/|s|) \sum_{\text{character } c} |n_c| \log(|s|/|n_c|)$, where n_c is the number of character c appearing in s.

[2] Unlike searching with BWT, the pattern P is processed in the *forward* direction during the trie search. It is because we are essentially searching for those nodes whose reverse prefix strings begin with $P[p]P[p-1]...P[1]$, so that we start by matching $P[1]$ first, and then $P[2]P[1]$, and so on.

3 Compressed Prefix Matching with XBW

Suppose that we want to index a trie $\mathcal{T}$, but instead of supporting subpath query, we want to check if a pattern $P[1..p]$ can be searched starting from the root. We call this a *prefix matching* query. Obviously, we can create a dummy root node z and connect z to the original root with a special character λ, and then reduce the prefix matching query to finding the subpath λP in the modified trie. In this way, the prefix matching query can be performed in $o(|P|(\log\log\sigma)^{1+\epsilon})$ time, while the index takes $nH_k(\mathcal{T}) + O(t)$ bits space.

A natural question is whether we can reduce the query time to $O(|P|)$, for any alphabet size σ. Very recently, Belazzougui [3] (implicitly) showed that it is possible. The idea is to find an encoding such that given a node u with rank x and a character c, we can determine if we can extend the node u by the character c in $O(1)$ time, and if so, the rank of the resulting node. His scheme is as follows:

1. For each character c in the alphabet, compute a list with the ranks of all reverse prefixes whose corresponding nodes are succeeded by c.[3] For instance, in the example of Figure 3, we compute
   - the list for `a`: 0, 4, 5;
   - the list for `b`: 0, 1, 4, 8;
   - the list for `c`: 0, 8;
2. Construct an *indexable dictionary* on each of the list, so that for each list L, we can support the following operation in $O(1)$ time:
   - rank(x, L): If x is in L, report its rank among the other elements of L. Otherwise, report -1 indicating "x is not found in L".
3. Construct an auxiliary data structure such that for each character c, we can report in $O(1)$ time the value C$[c]$, which is the number of reverse prefixes lexicographically smaller than c.

Based on the above data structures, we can perform our desired query as follows. Firstly, to check whether we can extend a node with rank x by the character c, we simply call rank(x, L_c) to check if the list L_c for c contains x, which requires $O(1)$ time. Next, suppose we can perform the extension. Then the rank of the desired node must be equal to $C[c] + rank(x, L_c)$, which also requires $O(1)$ time. Consequently, prefix matching can be performed in $O(|P|)$ time.

For the indexable dictionary, we use the result of [15], which takes $\log\binom{t}{n_c} + o(n_c) = n_c \log(t/n_c) + O(n_c)$ bits for the list for c, where n_c denotes the number of entries in the list. Note that n_c is exactly equal to the number of c in the XBW string X of $\mathcal{T}$. In total, the indexable dictionaries for all lists are stored in

$$\sum_{\text{character } c} n_c \log(t/n_c) + O(n_c) = tH_0(X) + O(t) \quad \text{bits.}$$

For the auxiliary data structure that supports computation of $C[c]$, it can be stored in $O(t + \sigma) = O(t)$ bits using Jacobson's constant-time rank and select index [13]. Thus, the overall space is bounded by $tH_0(X) + O(t)$ bits.

[3] When the trie contains only one path, these lists are exactly the Ψ function of the compressed suffix arrays [10].

3.1 Compressing Belazzougui's Scheme

We now show how to modify the above scheme so that the space becomes $tH_k(\mathcal{T}) + O(t)$, for any $k = o(\log_{\sigma+1} t)$.[4] Note that in any case, $H_k(\mathcal{T}) \leq H_0(X) \leq \log \sigma$. The key idea for the compression is to further divide the list into sublists, where each sublist contains the ranks of the reverse prefixes that begin with the same length-k string ρ; then, each sublist is encoded with a separate indexable dictionary. To see why compression can be achieved, we observe that the ranks in the sublist corresponding to ρ must be at least the rank of ρ (say x') among all the reverse prefixes, and can be at most $x' + |\text{cover}[\rho]| - 1$. Thus, we can store the rank x' of ρ, and replace each rank in the sublist with the difference with x'. As the number becomes smaller (between 0 and $|\text{cover}[\rho]| - 1$), we achieve compression. For instance, consider the example of Figure 3 and the case where $k = 1$. Then the ranks 4, 5, 6, and 7 will be referred as 0, 1, 2, 3, respectively, when we encode the sublist for $\rho = \texttt{b}$.

More precisely, suppose that the sublist in the list c that corresponds to ρ contains $n_{c,\rho}$ ranks. Then the corresponding indexable dictionary of [15], which supports $O(1)$ time query, can be stored in at most

$$n_{c,\rho} \log(|\text{cover}[\rho]|/n_{c,\rho}) + O(n_{c,\rho}) \quad \text{bits},$$

so that the sublist in all lists that correspond to ρ can be stored in at most

$$\sum_{\text{character } c} n_{c,\rho} \log(|\text{cover}[\rho]|/n_{c,\rho}) + O(n_{c,\rho}) = |\text{cover}[\rho]| H_0(\text{cover}[\rho]) + O(|\text{cover}[\rho]|) \quad \text{bits}.$$

Consequently, the total space of all sublists for all ρ's can be stored in at most

$$\sum_{\text{length-}k\ \rho} |\text{cover}[\rho]| H_0(\text{cover}[\rho]) + O(|\text{cover}[\rho]|) = tH_k(\mathcal{T}) + O(t) \quad \text{bits}.$$

The indexable dictionaries for the sublists are concatenated according to where it appears in the list. To facilitate the location of a particular dictionary, we use a conceptual bit-vector to mark the boundaries of the dictionaries. Such a bit-vector consists of $(\sigma+1)^{k+1}$ `1`'s and $tH_k(\mathcal{T})+O(t)$ `0`'s, which is stored with the *fully indexable dictionary* of [15] in $o(t)$ bits, and supports finding the desired dictionary for any sublist in $O(1)$ time. We can similarly store the rank of each ρ among the reverse prefixes by using a conceptual bit-vector with $(\sigma+1)^k$ `1`'s and t `0`'s. The idea is to treat ρ as a $k \log \sigma$-bit integer, such that the number of `0`'s preceding the ρth `1` is exactly equal to the desired rank of ρ. The required space is $o(t)$ bits, and the desired rank can be reported in $O(1)$ time. Also, when given a rank x, we can compute in $O(1)$ time the number of `1`'s preceding the xth `0`'s; this corresponds to the rank of the string ρ whose sublist may contain x. Finally, we construct a table such that for each character c and each length-k string ρ, we can report in $O(1)$ time the value $\text{C}[c, \rho]$, which is the number of

[4] With some minor adaptation, we can extend the range of k slightly to become $o(\log_\sigma n)$ instead.

reverse prefixes lexicographically smaller than $c\rho$. The table contains $(\sigma+1)^{k+1}$ entries, each taking $O(\log t)$ bits, so that the total space is bounded by $o(t)$ bits.

Given the above data structures, let us now examine how to perform the extension operation. Suppose we are given a node with rank x and a character c. First, we check which sublist of ρ that x may appear. Next, we check whether list c has a sublist corresponding to ρ. If not, we can conclude such an extension fails. Else, we retrieve the rank x' of ρ among the reverse prefixes, retrieve the indexable dictionary for the sublist $L_{c,\rho}$ of ρ within the list c, and check if $x - x'$ is stored in the sublist. If not, we can again conclude that the extension has failed. Else, we compute the desired rank r of the extended node as:

$$r = \mathrm{rank}(x - x', L_{c,\rho}) + C[c, \rho].$$

In summary, the total space of our index is $tH_k(\mathcal{T}) + O(t)$ bits, and it supports prefix matching of a pattern P in the trie $\mathcal{T}$ in $O(|P|)$ time.

4 Our Index for Compressed Dictionary Matching

The Aho-Corasick automaton [1] is an index for a set $\mathcal{D}$ of patterns that support dictionary matching query in $O(|T| + occ)$ time, where occ denotes the total number of occurrences of the patterns in T. The automaton consists of a trie $\mathcal{T}$ storing all the patterns in $\mathcal{D}$, together with two functions called *failure* and *report* that respectively facilitate the matching algorithm and the occurrence reporting. In particular, the failure function maps a node u to a node v such that its reverse prefix r_v is a proper prefix of r_u, and among all nodes v has the longest r_v. See Figure 4 for an example, where the mapping from each node to its desired node is shown by the dotted arrows. For the report function, it maps a node u to a node v such that (i) its reverse prefix r_v is a proper prefix of r_u, (ii) v itself corresponds to a pattern in $\mathcal{D}$ (that is, the reverse of r_v is a pattern in $\mathcal{D}$), and (iii) among all nodes v has the longest r_v. Note that the report function of a node u may not exist due to condition (ii). The space of the index is $O(t \log t)$ bits, where t is the number of nodes in $\mathcal{T}$.

In [3], Belazzougui showed that if each node u is represented with the rank of the reverse prefix r_u, then both the failure function and the report function can be encoded in $O(t)$ bits, and be computed in the same complexity as those in the

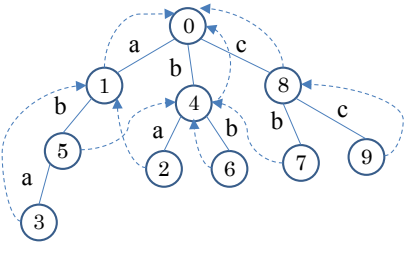


Fig. 4. An example of the failure function

uncompressed Aho-Corasick automaton. Indeed, Belazzougui's idea of encoding the failure function is very elegant: Consider storing all the *reverse prefixes* of $\mathcal{T}$ in a compact trie $\mathcal{C}$, and marking each node in $\mathcal{C}$ that corresponds to a reverse prefix. Let u_C denote the marked node in $\mathcal{C}$ that corresponds to r_u. It is easy to check that the rank of r_u is exactly the pre-order rank of u_C among all the marked nodes in $\mathcal{C}$. See Figure 5 for an example.

Now, to compute the node v mapped by the failure function of u in $\mathcal{T}$, we simply locate u_C, find its lowest marked ancestor v_C (which corresponds to the desired node v in $\mathcal{T}$), and return the pre-order rank of v_C among all the marked nodes. To support the above computation, we need only to store the tree structure of $\mathcal{C}$ (which contains $O(t)$ nodes), a lowest marked ancestor data structure, and a data structure for returning the pre-order rank of a marked node. All these can be stored easily in $O(t)$ bits using the standard technique (see [3] for details) so that the above computation is done in $O(1)$ time.

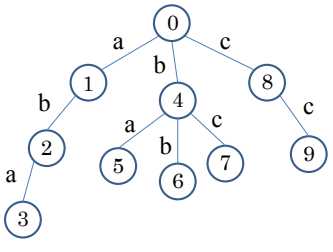


Fig. 5. An example of the compact trie of reverse prefixes. All nodes in this example are marked, and the label of a node shows its pre-order rank among all marked nodes. Note that in general some nodes in the compact trie may not be marked.

As for the report function, Belazzougui showed that it can be represented by a tree with $d = |\mathcal{D}|$ internal nodes and $O(t)$ leaves so that it can be encoded in $O(d \log(t/d) + d) = O(t)$ bits, and the function can be computed in $O(1)$ time. Thus, by combining the above with the prefix matching index in Section 3, where each node can be extended by any character c in $O(1)$ time, dictionary matching can be performed as if we are using the uncompressed Aho-Corasick automaton. This gives the following theorem.

Theorem 1. *The Aho-Corasick automaton with a trie $\mathcal{T}$ can be stored in $tH_k(\mathcal{T}) + O(t)$ bits, such that for any query text T, the dictionary matching query can be performed in $O(|T| + occ)$ time.*

5 Conclusion

We have slightly modified Belazzougui's scheme for encoding an Aho-Corasick automaton so that the index space becomes entropy compressed, while it supports dictionary matching query in $O(|T| + occ)$ time. Nevertheless, it seems

that such a scheme cannot readily support dynamic operations, where a pattern $P[1..p]$ may be inserted to or deleted from the set $\mathcal{D}$ from time to time. If randomized amortized update is allowed, then we remark that we can apply the technique of [11] so that each update requires $O(p + t^{\epsilon})$ randomized amortized time, while query time remains $O(|T| + occ)$; here, $\epsilon > 0$ and t denotes the number of states in the Aho-Corasick automaton. A challenging open question is to support worst-case update operation, while keeping the search time as close to $O(|T| + occ)$ as possible.

References

1. Aho, A., Corasick, M.: Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM 18(6), 333–340 (1975)
2. Amir, A., Farach, M., Matias, Y.: Efficient Randomized Dictionary Matching Algorithms (Extended Abstract). In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 262–275. Springer, Heidelberg (1992)
3. Belazzougui, D.: Succinct Dictionary Matching With No Slowdown. In: Amir, A., Parida, L. (eds.) Combinatorial Pattern Matching. LNCS, vol. 6129, pp. 88–100. Springer, Heidelberg (2010)
4. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA (1994)
5. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed Indexes for Dynamic Text Collections. ACM Transactions on Algorithms 3(2) (2007)
6. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring Labeled Trees for Optimal Succinctness, and Beyond. In: Proceedings of Symposium on Foundations of Computer Science, pp. 184–196 (2005)
7. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and Indexing Labeled Trees, With Applications. Journal of the ACM 57(1) (2009)
8. Ferragina, P., Manzini, G.: Indexing Compressed Text. Journal of the ACM 52(4), 552–581 (2005); A preliminary version appears in FOCS 2000
9. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed Representations of Sequences and Full-Text Indexes. ACM Transactions on Algorithms 3(2) (2007)
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. Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: A Framework for Dynamizing Succinct Data Structures. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 521–532. Springer, Heidelberg (2007)
12. 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)
13. Jacobson, G.: Space-efficient Static Trees and Graphs. In: Proceedings of Symposium on Foundations of Computer Science, pp. 549–554 (1989)
14. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
15. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In: Proceedings of Symposium on Discrete Algorithms, pp. 233–242 (2002)

16. Tam, A., Wu, E., Lam, T.W., Yiu, S.M.: Succinct Text Indexing With Wildcards. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) String Processing and Information Retrieval. LNCS, vol. 5721, pp. 39–50. Springer, Heidelberg (2009)
17. Weiner, P.: Linear Pattern Matching Algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
18. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers, Los Altos (1999)

Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval*

Shanika Kuruppu[1], Simon J. Puglisi[2], and Justin Zobel[1]

[1] National ICT Australia
Department of Computer Science & Software Engineering,
University of Melbourne
{kuruppu,jz}@csse.unimelb.edu.au
[2] School of Computer Science and Information Technology,
Royal Melbourne Institute of Technology, Australia
simon.puglisi@rmit.edu.au

Abstract. *Self-indexes* – data structures that simultaneously provide fast search of and access to compressed text – are promising for genomic data but in their usual form are not able to exploit the high level of replication present in a collection of related genomes. Our 'RLZ' approach is to store a self-index for a base sequence and then compress every other sequence as an LZ77 encoding relative to the base. For a collection of r sequences totaling N bases, with a total of s point mutations from a base sequence of length n, this representation requires just $nH_k(T) + s \log n + s \log \frac{N}{s} + O(s)$ bits. At the cost of negligible extra space, access to ℓ consecutive symbols requires $O(\ell + \log n)$ time. Our experiments show that, for example, RLZ can represent individual human genomes in around 0.1 bits per base while supporting rapid access and using relatively little memory.

1 Introduction

The emergence of high-throughput sequencing technologies, capable of sequencing entire genomes in a single run, has lead to a dramatic change in the number and type of sequencing projects being undertaken. In particular, it is now feasible to acquire and study variations between many individual genomes of organisms from the same species. While the total size of these sets of genomes will be large, individual genomes will not greatly vary, and so these collections represent new challenges for compression and indexing.

In this paper we address the following problem.

Definition 1 ([7]). *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^k| = n$ for $1 \leq k \leq r$ and $\sum_{k=1}^{r} |T^k| = N$, where $T^2, T^3, \ldots, T^r$ are mutated*

* This work was supported by the Australian Research Council and the NICTA Victorian Research Laboratory. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Center of Excellence program.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 201–206, 2010.

copies of the base sequence T^1 *containing overall* s *point mutations, the* repetitive collection indexing problem *is to efficiently store* $\mathcal{C}$ *while allowing queries of the form* display(i, j, k) *to efficiently return the substring* $T^k[i..j]$.

We describe a solution to this problem that requires $nH_k(T^1) + s \log n + s \log \frac{N}{s} + \mathrm{O}(s)$ bits of space and $\mathrm{O}(\ell + \log^{1+\epsilon} n)$ time to return a requested substring of length ℓ for any constant $\epsilon > 0$, assuming a constant alphabet size.

Our approach is to use the base sequence as a dictionary for compression of the other sequences. The collection is parsed into factors in an LZ77 [11] manner, but references to substituted strings are restricted to be in the base sequence only. Arbitrary substrings can then be decoded by reference to the base sequence. This work is inspired by the recent work of Mäkinen et al. [7] who, to our knowledge, were the first to tackle the above problem.

2 Background and Basic Tools

BioCompress [5] was the first compression algorithm to be specific to DNA sequence compression, by simple modifications such as encoding nucleotides with 2 bits per base and detecting reverse complement repeats. Two further variations on the *BioCompress* theme, *Cfact* [10] and *Off-line* [1], both work in rounds, greedily replacing duplicate text with shorter codes. *GenCompress* [3] showed that by considering approximate repeats the results could be improved. Since *GenCompress*, most DNA compression algorithms have been based on efficient methods of approximate repeat detection.

Many of these early DNA compression algorithms are only able to compress small files. Due to the latest advances in DNA sequencing technologies, much larger DNA sequencing projects are underway. As a result, many individual genomes from the same species are being sequenced, creating a large amount of redundancy. Recent algorithms have specifically addressed the issue of compressing DNA data from the same species. Christley et al. [4] compresses variation data from human genomes and encodes the mutations and indels with respect to the human reference sequence and known variations recorded in a SNP database. However, large resources are required to be shared among users of the compressed data (4.2 GB of reference and SNPs in Chirstley et al.'s software). Furthermore, it does not support random access into the sequences.

Before explaining our method we introduce notation and review several pieces of algorithmic machinery on which our results rely.

Strings. A string $T = T[0..n] = T[0]T[1] \dots T[n]$ is a sequence of $n + 1 = |T|$ symbols. The first n symbols of T are drawn from a constant ordered alphabet, Σ. The final character $T[n]$ is a special "end of string" character, \$, distinct from and lexicographically smaller than all the other characters in Σ. We write $T[i..j]$ to represent the ***substring*** $T[i]T[i+1] \cdots T[j]$ of T that starts at position i and ends at position j. For substring $T[i..j]$, if $j = n$ (resp. $i = 0$) we call the substring a suffix (resp. prefix) of T. With $\hat{T}$ we denote the reverse of T.

Suffix Arrays and Self-Indexes. The ***suffix array*** of T, denoted SA_T or just SA, when the context is clear, is an array SA[0..n] that contains a permutation of the integers 0..n such that $T[\mathrm{SA}[0]..n] < T[\mathrm{SA}[1]..n] < \cdots < T[\mathrm{SA}[n]..n]$. In other words, $\mathrm{SA}[j] = i$ iff $T[i..n]$ is the jth suffix of T in ascending lexicographical order. All the positions of occurrence in T of a given pattern, $P[1..m]$, lie in a contiguous range of the suffix array SA[$sp..ep$]. In recent years, successful attempts have been made to compress suffix arrays, and data structures call *self-indexes* have emerged [8]. A self-index of a text T allows the following functionality, all in compressed space: (i) extract (display) any substring $T[s..e]$, (ii) find the range $sp..ep$ for a pattern P, and (iii) return $\mathrm{SA}[i]$ for any i.

Relative Lempel-Ziv Factorization. Given two strings T and S, the Lempel-Ziv factorization (or parsing) of T relative to S, denoted $\mathrm{LZ}(T|S)$, is a factorization $T = w_0 w_1 w_2 \ldots w_z$ where w_0 is the empty string and for $i > 0$ each factor (string) w_i is either: (a) a letter which does not occur in S; or otherwise (b) the longest prefix of $T[|w_0 \ldots w_{i-1}|..|T|]$ that occurs as a substring of S. For example, if $S = abaababa$ and $T = aabacaab$ then in $\mathrm{LZ}(T|S)$ we have $w_1 = aaba$, $w_2 = c$ and $w_3 = aab$. It is convenient to specify the factors not as strings, but as (p_i, ℓ_i) pairs, where p_i denotes the starting position in S of an occurrence[1] of factor w_i (or a letter if w_i is by rule (a)) and ℓ_i denotes the length of the factor (or is zero if w_i is by rule (a)). Thus, in our example: $\mathrm{LZ}(T|S) = (3, 4)(c, 0)(2, 3)$. For convenience, we assume no factors are generated by rule (a) above; that is, if c occurs in T_i for $i \geq 2$ then c also occurs in T_1. If T_1 is not so composed we can simply add the at most $\sigma - 1$ missing symbols to the end of it.

Compressed Integer Sets. We make use of a compressed set representation due to Okanohara and Sadakane [9] (called "sdarray" in their paper). Given a set S of m integers over a universe u this data structure supports the operations: $\mathrm{rank}(S, i)$, returning the number of item in S less than or equal to i; and $\mathrm{select}(S, i)$, returning the value of the ith item in S. The data structure requires $m \log \frac{u}{m} + \mathrm{O}(m)$ bits and $\mathrm{O}(1)$ time for select and $\mathrm{O}(\log \frac{u}{m})$ time for rank.

3 Storing and Accessing Related Genomes

We now describe how to store a collection of related sequences, while still allowing efficient access to arbitrary substrings of any of the constituent sequences.

Lemma 2. *Given a collection $\mathcal{C}$ of r sequences $T^k \in \mathcal{C}$ such that $|T^k| = n$ for $1 \leq k \leq r$ and $\sum_{k=1}^{r} |T^k| = r \cdot n = N$, with*

$$\mathrm{LZ}(T^i|T^1) = (p_1, \ell_1), (p_2, \ell_2), \ldots, (p_{z_i}, \ell_{z_i}),$$

for $2 \leq i \leq r$ and $z = \sum_{i=2}^{r} z_i$, we can store $\mathcal{C}$ in at most $n \log \sigma + z \log n + z \log \frac{N}{z} + \mathrm{O}(z)$ bits such that any substring $T^k[s..e]$ can be output in $\mathrm{O}(e - s + \log \frac{N}{z})$ time.

[1] There may be more than one occurrence; for our purposes here it does not matter to which one p_i refers.

Proof. T^1, the base sequence, is stored in $n \log \sigma$ bits (in the obvious way).

Let $Z^i = \text{LZ}(T^i|T^1) = (p_1, \ell_1), (p_2, \ell_2), \ldots, (p_z, \ell_{z_i})$ for $i > 1$ be the parsing of text T^i relative to T^1. We store Z^i in two pieces. The position components of each factor, $p_1, \ldots, p_{z_i}$ are stored in a table $P_i[1..z_i]$ taking $z_i \log n$ bits. P_i is simply a concatenation of the bits representing $p_1, \ldots, p_z$. Each entry in P_i is $\log n$ bits long, allowing access to any entry $p_j = P_i[j]$ in O(1) time. The length components are stored in a compressed integer set, L_i, containing the values $j = \sum_{k=1}^{u} \ell_k$ for all $u \in 1..z_i$. In other words, the values in L_i are the starting positions of factors in T_i. Via a select query, L_i allows us to access a given p_j as $P_i[\text{select}(L_i, j)]$; and the length of the jth factor, ℓ_j, is simply $\text{select}(L_i, j + 1) - \text{select}(L_i, j)$ for $j \in 1..z - 1$ (because of the terminating sentinel we always have $\ell_z = 1$). Furthermore, the factor that $T^i[j]$ falls in is given by $\text{rank}(L_i, j)$. L_i is stored in the "sdarray" representation [9], which requires $z \log \frac{N}{z} + \text{O}(z)$ bits and allows rank in $\text{O}(\log \frac{N}{z})$ time and select in O(1).

As described, P_i and L_i allow, given j, fast access to p_j and ℓ_j: simply a select query on L_i to get ℓ_j and a lookup on P_i to retrieve p_j. Given a position k in sequence T^i, we can determine the factor in which k lies by issuing a rank query on L_i, in particular $\text{rank}(L_i, k)$. To find a series of consecutive factors we only need to use rank in obtaining the first factor. For the others, the ℓ values can be retrieved using repeated select queries on L_i and the p values by accessing consecutive fields in P_i, both in constant time per factor. This observation allows us to extract any substring $T^i[s..e]$ in $\text{O}(e - s + \log \frac{N}{z})$ time overall. □

We can reduce the $n \log \sigma$ term in the size of the above data structure to nH_k bits by storing T^1 as a self-index instead of in plain form. This increases the cost to access a substring of length ℓ to $\text{O}(\ell \log^{1+\epsilon} n + \log \frac{N}{z})$ worst-case time. It is possible to build our data structure in $\text{O}(n + N \log n)$ time and $n \log \sigma + n \log n$ bits of extra space. The basic idea is to build the suffix array for T_1 and "stream" every other sequence against it to generate the $\text{LZ}(T^i|T^1)$ parsings.

4 Experimental Results

The compression performance of our algorithm, which we call RLZ, is compared to the algorithms `XM` [2], which is known to currently be the best single sequence DNA compression algorithm, Comrad [6], which specialises in compression of large related DNA datasets, and RLCSA, an implementation of a self-index from Mäkinen et al. [7]. The RLZ *display()* function is also compared to RLCSA [7].

Table 1 compares RLZ with RLCSA, Comrad and `XM` by compressing datasets containing many real biological sequences that slightly vary from each other. The datasets are *S. coronavirus* with 141 sequences, *S. cerevisiae* with 39 genomes, *S. paradoxus* with 36 genomes, and *H. sapien* with 4 genomes.

The compression performance of RLZ is varied (Table 2). Unsurprisingly, for the smaller datasets, `XM` has the best compression results. For the larger dataset, RLZ is produces better compression results than Comrad while using less memory (≈45 Mbyte for the yeast sets). RLCSA doesn't perform as well as the other algorithms, but it uses less memory (≈100 Mbyte for yeast) and

Table 1. Compression results for four repetitive collections. The first row is the original size for all datasets (Size in Megabases rather than Mbytes), the remaining rows are the compression performance of RLZ, RLCSA, COMRAD and XM algorithms. The two columns per dataset show the size in Mbytes and the 0-order entropy (in bits per base).

Dataset	S. coronavirus		S. cerevisiae		S. paradoxus		H. sapien	
	Size (Mbyte)	Ent. (bpb)	Size (Mbyte)	Ent. (bpb)	Size (Mbyte)	Ent. (bpb)	Size (Mbyte)	Ent. (bpb)
Original	4.19	1.98	485.87	2.18	429.27	2.12	12066.06	2.18
RLZ	0.08	0.15	17.89	0.29	23.38	0.44	754.43	**0.50**
RLCSA	0.22	0.43	41.39	0.57	47.35	0.88	3834.82	2.54
COMRAD	0.09	0.18	15.29	**0.25**	18.33	0.34	2176	1.44
`XM`	0.03	**0.06**	74.53	1.26	13.17	**0.25**	—	—

Table 2. Compression and decompression times (in seconds)

Dataset	S. coronavirus		S. cerevisiae		S. paradoxus		H. sapien	
	Comp. (sec)	Decom. (sec)	Comp. (sec)	Decom. (sec)	Comp. (sec)	Decom. (sec)	Comp. (sec)	Decom. (sec)
RLZ	**2**	**1**	**213**	**12**	**260**	**10**	**9874**	**172**
RLCSA	6	2	781	312	740	295	34525	14538
COMRAD	10	16	1070	45	1068	50	28442	1666
`XM`	43	62	18990	17926	30580	28920	—	—

Table 3. *display()* times for RLZ and RLCSA for varying query lengths. Query datasets contain 1000 queries each with the same length. The times for each algorithm are in microseconds per character extracted. The times are an average of 5 consecutive runs per dataset. RLZ used 28.71 MByte of memory and RLCSA used 47.35 MByte.

Query Length	10	100	1000	10000	100000
RLCSA (μsec/char)	18.00	2.40	0.85	0.71	0.71
RLZ (μsec/char)	**0.2300**	**0.0250**	**0.0046**	**0.0025**	**0.0022**

is faster than COMRAD and `XM`. Overall, RLZ compress and decompress much faster, and uses drastically less memory, than RLCSA, COMRAD and `XM`.

For the *H. sapien* dataset, each chromosome of each dataset was compressed against the respective reference genome chromosomes for RLZ and RLCSA. We do not report `XM` results since it took nearly 6 hours for a single chromosome 1 sequence to compress. RLZ performed very well on this dataset compared to RLCSA and COMRAD (RLZ used $\approx$1 Gbyte of memory while RLCSA and COMRAD used $\approx$2 and $\approx$16 Gbyte respectively). With the inclusion of the 7-zipped human reference genome, the total dataset of three human genome sequences (summing to 12 Gbase) can be represented in just under 755 MByte with RLZ; of this, 643 Mbyte is the overhead for the base sequence, and we anticipate that roughly 27,000 human genomes could be stored in a terabyte, a massive increase on the 1500 or so that could be stored using methods such as `7-zip`.

Results for the *display(i,s,e)* (retrieve substring $T^i[s..e]$) function are in Table 3. RLZ displays substrings significantly faster than RLCSA for small query lengths and continues its good performance for even larger query lengths, with approximately 0.022 (μsec/c) for RLZ compared to 0.77 (μsec/c) for RLCSA.

Tests were conducted on a 2.6 GHz Dual-Core AMD Opteron CPU with 32Gb RAM and 512K cache running Ubuntu 8.04 OS. The compiler was gcc v4.2.4 with the -O9 option.

5 Discussion

A key issue is choice of a reference sequence. While selecting the reference genome is a simple matter for the *yeast* datasets, it may not be a good representation of the other individual sequences. To observe the compression performance for different reference sequence choices, we compressed the dataset by selecting each genome in turn as the reference. Selecting a genome such as *DBVPG6765* leads to 16.5 MByte as opposed to selecting *UWOPS05_227_2*, which led to 24.5 MByte. We also explored the effect on compression when a reference sequence that is unrelated to the original dataset is selected. 35 genomes of *S. paradoxus* (excluding *REF* genome) was compressed against the *S. cerevisiae REF* genome. The compressed size was 112.09 MByte compared to the result of 2.04 MByte when the *S. paradoxus REF* genome was used. RLZ's effectiveness is at its best when compressing related sequences and care must be taken when selecting a reference sequence.

References

1. Apostolico, A., Lonardi, S.: Compression of biological sequences by greedy off-line textual substitution. In: Proc. IEEE DCC, pp. 143–152 (2000)
2. Cao, M.D., Dix, T., Allison, L., Mears, C.: A simple statistical algorithm for biological sequence compression. In: Proc. IEEE DCC, pp. 43–52 (2007)
3. Chen, X., Kwong, S., Li, M.: A compression algorithm for DNA sequences and its applications in genome comparison. In: Proc. RECOMB, p. 107. ACM, New York (2000)
4. Christley, S., Lu, Y., Li, C., Xie, X.: Human genomes as email attachments. Bioinformatics 25(2), 274–275 (2009)
5. Grumbach, S., Tahi, F.: Compression of DNA sequences. In: Proc. IEEE DCC, pp. 340–350 (1993)
6. Kuruppu, S., Beresford-Smith, B., Conway, T., Zobel, J.: Repetition-based compression of large DNA datasets. In: Poster at RECOMB (2009)
7. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Computational Biology 17(3), 281–308 (2010)
8. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Computing Surveys 39(1) (2007)
9. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. ALENEX. SIAM, Philadelphia (2007)
10. Rivals, E., Delahaye, J., Dauchet, M., Delgrange, O.: A guaranteed compression scheme for repetitive DNA sequences. In: Proc. IEEE DCC, p. 453 (1996)
11. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)

Standard Deviation as a Query Hardness Estimator*

Joaquín Pérez-Iglesias and Lourdes Araujo

Universidad Nacional de Educación a Distancia
Madrid 28040, Spain
joaquin.perez@lsi.uned.es, lurdes@lsi.uned.es

Abstract. In this paper a new *Query Performance Prediction* method is introduced. This method is based on the hypothesis that different score distributions appear for 'hard' and 'easy' queries. Following we propose a set of measures which try to capture the differences between both types of distributions, focusing on the dispersion degree among the scores. We have applied some variants of the classic standard deviation and have studied methods to find out the most suitable size of the ranking list for these measures. Finally, we present the results obtained performing the experiments on two different data-sets.

1 Introduction

Query Performance Prediction (QPP) deals with the problem of estimating the difficulty of a query, where the difficulty degree is commonly measured in terms of the average precision (AP) obtained by the query. Thus, a query which obtains a low AP value is considered as a 'hard' query, while one with a high AP value would be considered as an 'easy' query. A classic application of QPP which has shown some interesting results is *selective query expansion* [1], where the quality prediction is applied to avoid the automatic expansion of those queries which would worsen the overall retrieval quality.

This paper introduces a novel approach for Query Performance Prediction, which falls into the so-called post-retrieval prediction methods. This type of predictors makes use of the information supplied by the search system, once the search has been performed, while pre-retrieval predictors compute the estimation before completing the search. Our approach is focused on the study of the scores assigned to the documents returned by a search system in response to a query. It is based on the hypothesis that there exist differences between the scores distribution of 'hard' and 'easy' queries. The dispersion in the scores of the document ranking list is measured in order to predict the query performance.

* This paper has been funded in part by the Spanish MICINN projects NoHNES (Spanish Ministerio de Educación y Ciencia - TIN2007-68083) and by MAVIR, a research network co-funded by the Regional Government of Madrid under program MA2VICMR (S2009/TIC-1542). Authors want to thank Álvaro Rodrigo-Yuste for his review and comments.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 207–212, 2010.

The rest of this paper is organised as follows. In Section 2 related work in query performance prediction is introduced, with a special emphasis on post-retrieval approaches. Then, in Section 3, a detailed description of the starting hypothesis and the different measures employed to compute the ranking list dispersion is given. Section 4 deals with the specific evaluation performed and the analysis of the obtained results. Finally, the main conclusions are given in Section 5.

2 Related Work

In the last years several techniques dealing with Query Performance Prediction have been proposed. The different prediction methods are usually classified into two main categories: *a*) *pre-retrieval* approaches, which try to estimate query difficulty without using the list of documents obtained from the search engine; *b*) and *post-retrieval*, which use the information obtained after submitting the query to the search engine.

Focusing on post-retrieval methods we can found the classic Clarity Score by Cronen-Townsend et al.[2]. Clarity Score tries to measure the ambiguity of a query with respect to the document collection. The ambiguity of a query is calculated using the Kullback-Leibler divergence (KLD) between the language models of the collection and the top ranked documents. A well performing query would show a high divergence value. Some methods based on Clarity Score appear subsequently, such as *Ranked List Clarity Score*, which replaces ranking scores by the document ranking position, and *Weighted Clarity Score*, which allows to assign a different weight to each query term in order to calculate KLD, both in [3].

Recently a new improved version of Clarity Score has been presented by Hauff et. al [4], which outperforms the original Clarity Score on performance prediction accuracy. A related approach to the one introduced here, where the scores of the ranking list are analysed, was developed by Diaz [5], who applied the similarity between the scores of topically close documents as a prediction value. A similar approach was proposed by Vinay [6], where the prediction is based on the correlation between the actual rank of a document and a computed expected rank, where the expected rank is obtained by modelling the score of a document as a Gaussian random variable.

The following section introduces the proposed post-retrieval predictor based on the scores dispersion in a ranking list.

3 Ranking List Score Dispersion as a Predictor

We based our approach on the hypothesis that some differences between document score distribution for 'hard' and 'easy' queries should be observed. For example, if a ranking list has a high value of dispersion in their document scores, it could indicate that the ranking function has been able to discriminate between relevant and non-relevant documents. On the other hand, if a low level of dispersion appears, it can be interpreted as if it was not able to distinguish between relevant documents from non-relevant ones.

This behaviour can be observed assuming a 'perfect probabilistic model', where documents are scored with a probability of relevance equal to either 1 or 0. Relevant documents are weighted with 1, while 0 is assigned to those considered non-relevant. In this theoretical model, the dispersion among scores will be maximised when an equal number of relevant and non-relevant documents are included within the ranking list. An immediate consequence of the application of dispersion as a query hardness estimator, is the importance of selecting a suitable size k for the document ranking list. The dispersion measured at a wrong ranking list size, which for example includes too many non-relevant documents or not enough relevant documents, will imply a misleading estimation of the query hardness.

An example of the differences in terms of score dispersion can be observed in Figure 1, where the five best (left) and five worst (right) performing queries from Robust 2004 track are represented. As it can be seen in the figure, best queries show a longer distance between maximum and minimum score and a sharp slope. However, queries with a poor performance show a higher similarity in their scores along the ranking list, a softer slope and a smaller distance between maximum and minimum scores.

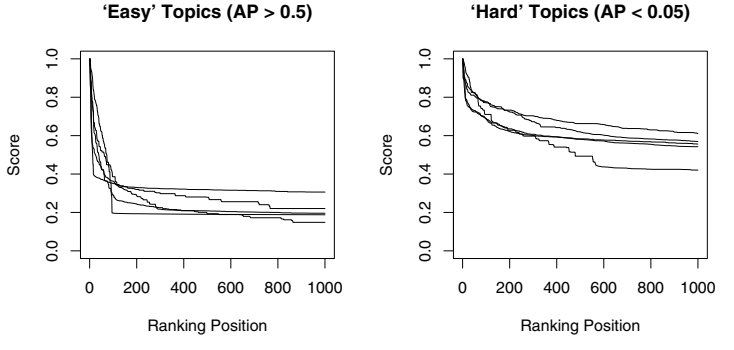


Fig. 1. 5 Best Performing Queries (left) vs 5 Worst Performing Queries (right), from Robust 2004 using BM25 as ranking model. Scores have been normalised in $[0, 1]$, dividing each score by the highest one. The maximum number of retrieved documents is fixed to 1000.

Concerning score distributions, previous works have tried to define how document scores are distributed along a ranking list. In general, it can be assumed that an appropriate model could be a mix of an exponential and a gaussian probability distribution. Exponential for non-relevant documents, and gaussian for relevant documents [7]. Usually most of the retrieved documents are non-relevant (exponential distribution), thus it is likely that a big majority of documents will obtain a low score.

Therefore, the shape of the ranking list exhibits a 'long tail' where most of the non-relevant documents are placed, see Figure 1. An overall effect of the ranking list 'long tail' is a reduction of the accuracy of the applied measures.

Next section describes the measures defined to capture the differences between 'hard' and 'easy' queries.

Proposed Measures

We have tested different measures for capturing the dispersion from the scores of a ranking list. All the proposed measures are based on standard deviation, but with some differences in the techniques used for fixing the most suitable ranking list size k. It should be noted that document scores are normalised by the maximum to allow a fair comparison between queries with different lengths.

Maximum Standard Deviation: This measure tries to minimize the effect of the *ranking list tail* by computing the standard deviation at each point in the ranking list and selecting the maximum value of the standard deviation thus found. Hence, those scores which appear at the *ranking list tail* have no influence in the calculation of the maximum standard deviation.

Standard Deviation at a Common Best k: Computing the standard deviation manually fixing its size k. With the selection of a suitable size k the noise introduced by the set of low scores is removed. Fixing k globally requires the selection of a common k value which maximises the correlation degree for all queries.

Estimating a Cut-Point k Automatically for each Query: The previously introduced measures establish a ranking list size k which is shared by all queries. Here we propose a method aimed at fixing the size k specifically for each query. For this estimation we use the number of documents which are retrieved when each term from the query is considered mandatory, which is equivalent to applying the boolean AND operator for all terms in the query. This estimator has been applied previously [4] to select an appropriate document set size in order to create a top ranking language model.

With the use of the AND operator some queries retrieve no documents at all, while for others the number of retrieved documents is large and similar to that obtained with the OR operator. In order to circumvent this situation a scaling linear function is applied. This function scales the original k value obtained from an AND search to a value closer to the expected median ($\tilde{k}$) for all queries, avoiding the undesired cases described above.

The linear transformation makes use of a free parameter λ, which defines how similar will be the new ranking list size to the original median. In order to ascertain the similarity between the original k value, obtained with the AND operator, and the scaled one k', obtained after the linear transformation, the next conditions have to be fixed:

$$\tilde{k'} = \tilde{k} \text{ and } k'_{max} = \lambda\tilde{k}$$

where k'_{max} is the longest ranking list size expected for the scaled values. Thus the linear transformation is calculated as $k' = ak + b$, where $a = \frac{(\lambda-1)\tilde{k}}{k_{max}-\tilde{k}}$, $b = (1-a)\tilde{k}$ and $\lambda \in [1, \frac{k_{max}}{\tilde{k}}]$.

4 Results

As usual the performance of the predictor is computed by measuring the correlation degree between the predictor and the real search system performance in terms of AP. A significant correlation degree between both measures means an accurate estimation of the query performance. For this purpose the Pearson and Kendall correlation coefficients have been applied. Both correlation coefficients calculate a real number in the range $[-1, 1]$, where 1 means perfect correlation, -1 means a perfect inverse correlation and 0 means no correlation at all. The different measures proposed in this paper have been tested with the set of documents from TREC Disk4 & 5 and GOV2 collections, including topics from 301 to 450 and 700 to 800. Only the field title from each topic has been employed for both collections and the Okapi BM25 [8] has been applied as ranking function.

The results obtained with both datasets are shown in Table 1. Each row corresponds to one of the proposed measures: (σ_{full}) the standard deviation for the whole ranking list (1000 for TREC 4 & 5 and 10000 for GOV2); (σ_{best}) the standard deviation at a common best size k for all topics (100 for TREC 4 & 5 and 1000 for GOV2); (σ_{max}) the maximum standard deviation; and (σ_k) the standard deviation at a specific size k for each query, using the automatic method proposed previously. The last two rows include the results obtained with *Clarity Score* (CS) and the *Improved Clarity Score* (ICS)[1] version proposed by Hauff et al. in [4].

In relation with the capability to capture dispersion of the different proposed measures. Here we can observe how the standard deviation of the whole ranking list obtains the lowest correlation value, as it was expected because of the noise introduced by the *ranking list tail*. The results obtained with the maximum standard deviation are similar to those achieved with the selection of a common optimal ranking list size, but with the advantage of removing the use of a parameter to cut the ranking list and thus avoiding the problem of computing this optimal common k. The best results have been achieved by fixing automatically a suitable ranking list size for each topic.

Table 1. Pearson and Kendall coefficients obtained with the proposed measures

	Pearson					**Kendall**				
	TREC 4 & 5			**GOV2**		**TREC 4 & 5**			**GOV2**	
	301-350	**351-400**	**401-450**	**701-750**	**751-800**	**301-350**	**351-400**	**401-450**	**701-750**	**751-800**
σ_{full}	0.3886	0.3358	0.4367	0.4551	0.1497	0.2721	0.2295	0.2261	0.2959	0.1530
σ_{best}	0.7455	0.5188	0.6363	0.3808	0.2509	0.4693	0.3690	0.3146	0.1921	0.2534
σ_{max}	0.6488	0.4298	0.7854	0.3522	0.2322	0.4761	0.3282	0.4659	0.1802	0.2380
σ_k[4]	0.7802	0.5623	0.7136	0.4957	0.3019	0.5340	0.3996	0.3639	0.3656	0.2193
CS	0.5390	0.3095	0.5727	0.6033	0.4441	0.4198	0.2172	0.3045	0.4149	0.3299
ICS	0.6330	0.5106	0.7064	0.5422	0.5498	0.4998	0.4002	0.5624	0.3723	0.4181

[1] CS and ICS results have been taken from [4].

Finally, the correlation found in the GOV2 collection is not so strong, as for TREC 4 & 5 dataset, although it is almost equivalent to the achieved with CS or ICS. We believe that this decrease is caused by the noise which appears in a Web environment, where for each query the number of retrieved documents that obtain a similar score is higher than for TREC 4 & 5. A similar problem was addressed by Hauff et al. [4] in the application of predictors to a Web environment.

5 Conclusions

A novel query performance predictor has been introduced in this paper. A high correlation value has been found using the proposed measures based on standard deviation, suggesting the hypothesis described on this work about the relation between the scores dispersion and query performance.

The application of the standard deviation as a dispersion measure for a ranking list has shown to be an effective approach if a suitable ranking list size is selected. We have introduced three different techniques for fixing this size, which have improved the results reducing the noise introduced by the *ranking list tail*. Best results have been achieved using a specific ranking list size per topic.

References

1. Amati, G., Carpineto, C., Romano, G., Bordoni, F.U.: Query Difficulty, Robustness and Selective Application of Query Expansion. In: McDonald, S., Tait, J.I. (eds.) ECIR 2004. LNCS, vol. 2997, pp. 127–137. Springer, Heidelberg (2004)
2. Cronen-Townsend, S., Zhou, Y., Croft, W.B.: Predicting query performance. In: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR 2002. ACM Press, New York (2002)
3. Cronen-Townsend, S., Zhou, Y., Croft, W.B.: Precision prediction based on ranked list coherence. Information Retrieval 9(6), 723–755 (2006)
4. Hauff, C., Murdock, V., Baeza-Yates, R.: Improved query difficulty prediction for the web. In: CIKM 2008: Proceedings of the 17th ACM conference on Information and knowledge management, pp. 439–448. ACM, New York (2008)
5. Diaz, F.: Performance prediction using spatial autocorrelation. In: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR 2007, p. 583. ACM Press, New York (2007)
6. Vinay, V., Milic-Frayling, N., Cox, I.: Estimating retrieval effectiveness using rank distributions. In: CIKM 2008: Proceedings of the 17th ACM conference on Information and knowledge management, pp. 1425–1426. ACM, New York (2008)
7. Robertson, S.: On Score Distributions and Relevance. Advances in Information Retrieval 4425, 40–51 (2007)
8. Jones, K.S., Walker, S., Robertson, S.E.: A probabilistic model of information retrieval: development and comparative experiments. Inf. Process. Manage. 36(6), 779–808 (2000)

† Where λ was fixed to 5 for TREC 4 & 5 and 8 for GOV2.

Using Related Queries to Improve Web Search Results Ranking

Georges Dupret[1], Ricardo Zilleruelo-Ramos[2], and Sumio Fujita[3]

[1] Yahoo! Labs
Sunnyvale USA
[2] Yahoo! Research Latin America
Santiago Chile
[3] Yahoo! JAPAN Research
Tokyo Japan

Abstract. There are numerous queries for which search engine results are not satisfactory. For instance, the user may submit an ambiguous or miss-spelled query; or there might be a mismatch between query and document vocabulary, or even character set in some languages. Different automatic methods for query rewriting / refinement have been proposed in the literature, but little work has been done on how to combine the results of these rewrites to find relevant documents. In this paper, we review some techniques efficient enough to be computed online and we discuss their respective assumptions. We also propose and discuss a new model that is theoretically more appealing while still computationally very efficient. Our experiments show that all methods manage to improve the ranking of a leading commercial search engine.

1 Introduction

The search engine query logs provides an invaluable source of information about what a user may find as relevant over the results presented by the search engine and the path he follow to express his information need.

Many queries contain errors or are inherently ambiguous. There is also typically a gap between the language in which users express queries, and the representation built by the search engine to answer them. A possible solution to narrow this gap, is to re-interpret or re-write the user query in a way that is more adequate to the search engine. This is known generically under the name of *query refinement*. Various techniques for query refinement have been developed by exploiting various sources of data. Nevertheless, the question of how to combine the different refinements suggested by these various techniques remains an open question.

In this work, we make use of *search engine query logs* to derive both query rewritings or refinements and document relevance estimates. The logs continuously collect the trace of user interactions with the search engine and provides detailed and valuable information about the issued queries, the URLs presented by the search engine, the clicked documents, and their ranking. It is a poll of

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 213–224, 2010.

millions of users over an enormous variety of topics, and can be thought as users votes in favor of the documents they find interesting.

Click data has been used in different ways to mine user interests and preferences. Examples of applications include Web personalization, Web spam detection, query term recommendation, among others. Also, click data seems the perfect source of information when deciding which documents (or ads) to show in answer to a query. This information can be fed back into the engine, to tune search parameters or even used as direct evidence to influence ranking [11].

However, click data has to be pre-processed in order to be useful. It is very noisy, both including users' mistakes (such as clicks on irrelevant documents by misled users), as well as malicious actions from search engine spammers [3]. Beyond the problems related to data cleaning, there is the problem of how to interpret clicks. For example, the probability of a document being clicked depends not only on its relevance, but on other factors such as its position in the result page, the other documents co-appearing in the result list and the quality of the snippet that summarizes the document. Various methods have been proposed in the literature to evaluate relevance from clicks [8,6,4,9]. We will use the model proposed in [8] to deal with position bias and provide a baseline, and we will focus on another fundamental problem that motivates this work: the sparsity of click data. Users click on the URLs of documents shown on the first page of results, and rarely on the following result pages. Thus, conclusions about the document relevance can only be drawn for a small subset of candidate documents, and a highly relevant document can be totally missed. Radlinski et al. [14] recognize this issue and propose to regularly alter the result list to give other documents a chance of being clicked so as to collect information about them.

Approach. The approach we present in this work is based on the observation that when users are not entirely satisfied with the results presented to them, they often issue new queries [12] until they find relevant documents. When reformulating a query, users can correct the spelling, disambiguate the original query or use a new formulation that better matches the vocabulary of the relevant documents. This is intrinsically an iterative process.

We propose to use these reformulations to improve the original query result list. Consider the following scenario: a user who issues a query q_0, does not click on any result, but reformulates her information need as query q_1. She then clicks on a document on the new result list. Clearly, the presence of the clicked document in the results of q_0 would have improved the user experience. The idea can be extended to ambiguous queries: users' reformulations may reflect different subtopics totally or partially missed by the search engine for the original query.

Pertinent query reformulations can also be obtained in different ways from the click-through data [7] or from different methods, generally referred to as *query refinement* [16]. Consider for example the query "`new deal`". It can be parsed as two distinct terms, or as a phrase query, leading to different sets of documents. The objective of query refinement is to transform the original query into a new query before submitting it to the search engine. Although most work on query refinement aims at discovering one superior reformulation and substitute it the original

query, we argue that maintaining different reformulations of the same query are potentially useful and the final ranking should be able to take *all of them* into account, pondered by their importance. For example, a query like "`apple`" could be reformulated as "`apple computer`" or "`apple growing`". Choosing one of the reformulations exclusively is sure to frustrate the users having the other intent. Other cases arise where the original query has various syntactically different but semantically equivalent reformulations. A final example is related to query segmenation: A three words query "`A B C`" can be segmented in various ways –say `"A B", "C"` or `"A", "B C"`, etc. All these rewrite might be beneficial. We can also consider long queries where different segmentations can co-exist with some query term deletion mechanism.

Contribution. In this paper, we review existing methodologies to take advantage of the various query reformulations to construct the final result list. We make the assumption that these reformulations, along with their probability of being relevant to the original query, have been estimated using a reliable method. We also assume that we know the relevance of document in the result list of the query for which they appear. Various methods have been proposed in the literature to evaluate such probabilities from clicks [8,6,4,9]. We will use a variation on the one described in [8].

In Section 2, we cast the problem in a probabilistic framework. By making some carefull assumptions, the resulting general statistical model is simplified to an efficient and tractable form. We proceed in Section 3 to compare the different models based on numerical experiments.

2 Generative Model

In this section we cast the problem in a general probabilistic framework and discuss hypothesis that lead to a tractable solution. We first introduce some notations:

- **Document-query relevancy.** A document d is relevant to a query q_0 for a user, if when asked "is this document relevant to your search?" the user answers "yes". The relevancy of d to q_0 is the proportion of users finding d relevant to q_0. We use the following notation: $P(d^+|q^+)$ is the probability that document d is relevant to the user (denoted d^+) whenever query q is relevant to the user (similarly denoted q^+). This is the probability of a binary event; denoting the event that the document is not relevant by d^-, we have $P(d^+|q^+)+P(d^-|q^+) = 1$. The probability of being relevant is typically used as a score for ranking documents in accordance to Robertson probabilistic ranking principle [15].
- **Inter-query relevancy.** $P(q_i^+|q_0^+)$ is the probability that the query q_i is relevant to the users for which q_0 is relevant[1]. For example, if we discover

[1] There are different ways to define the relevance of a query. We can state for example that a query is relevant if the user who issued q_0 answers yes to the question "would a correct answer to q_i help to satisfy your information need?".

that 90% of the users who issue the query "`car`" are actually interested in used cars, we would have $P($ "`used car`" is relevant $|$ "`car`" is relevant$) = 0.9$. Note that if the conditioned query possess a close synonym, then the probability of relevance of the synonym query should be equal or similar. In our example, we should have $P($"`second hand car`" is relevant $|$ "`car`" is relevant$) \simeq 0.9$.

The probability of a query being relevant is a binary event and $P(q^+|q_0^+)+P(q^-|q_0^+) = 1$. Unlike random-walk models [5] , here $\sum_{i=0}^{|Q|} P(q_i^+|q_0^+)$ where Q is the set of all queries, is always larger or equal to 1 because we have by definition that $P(q_0^+|q_0^+) = 1$. The difference with the random-walk model is due to a different event definition. In the random-walk model, the event is "the user jumps to node n" and because a user can jump to only one node at a time, the probabilities need to add to one. By contrast, the event we describe here is "the query is relevant to the user", and, as various queries can be relevant simultaneously, the scores need not add to one.

– $P(d^+|q_0^+)$ is the probability of relevance of document d prior to including the knowledge about the relation of q_0 with other queries ($P(d^+|q_i^+)$ and $P(q_i^+|q_0^+)$). The posterior estimate of the relevance of d is denoted $P(d^+|q_0^+;Q)$ where Q is the set of queries that is taken into account to update $P(d^+|q_0^+)$.

The generative model makes the following assumption on the user mental process: the user thinks of an ideal document that would answer his query. Given this document, he decides which queries have the potential to bring up the document in the search engine. Finally, he chooses one query among these queries and issues it to the search engine. This process can be described by a joint distribution $P(d^+, \mathcal{I}_\mathcal{Q}, q_0^+)$ where d is a document, $\mathcal{I}_\mathcal{Q}$ relates to the set of queries and q_0 is relevant. Returning to the "`apple`" query example and denoting "`apple computer`" by q_1 and "`apple tree`" by q_2, we expect intuitively that $\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^+\}$ has a low probability because few users are interested in both the fruit and the computer at the same time. Meanwhile, $P(\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^-\}|q_0)$ should be large because the majority of web search users who issue the query "`apple`" are actually interested in apple computers exclusively.

$P(d^+|\mathcal{I}_\mathcal{Q})$ is the probability that a document is relevant given the set of relevant and irrelevant queries identified by $\mathcal{I}_\mathcal{Q}$. If $\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^-\}$ in the previous example and the document d is `www.apple.com` then $P(d|\mathcal{I}_\mathcal{Q})$ is large. On the other hand, the relevance of d to $\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^+\}$ is low because the web site of Apple Inc. does not provide information about the fruit of the same name.

Because we do not observe $\mathcal{I}_\mathcal{Q}$, we will need to sum over all its possible states:

$$P(d^+|q_0^+;Q) = \sum_{\mathcal{I}_\mathcal{Q} \in \mathcal{P}(\mathcal{Q})} P(\mathcal{I}_\mathcal{Q}|q_0^+) P(d^+|\mathcal{I}_\mathcal{Q}) \tag{1}$$

where $\mathcal{P}(\mathcal{Q})$ is the set of the $2^{|Q|}$ possible $\mathcal{I}_\mathcal{Q}$ configurations.

The remaining of this section examine various possibilities to estimate the terms in the Eq. 1 in terms of what we suppose is known, i.e. $P(q_i^+|q_0^+)$ and

$P(d^+|q_i^+)$. Note that the conventional way to handle this problem is to define instead of $\mathcal{I}_\mathcal{Q}$ a latent variable that spans the latent topics covered by the documents and the queries. The inconvenience of introducing a latent variable is that estimation ends up being iterative and numerically very costly, especially in the context of web retrieval where the number of queries and documents is very large. We will see that the proposed model leads to an efficient approximate method.

2.1 Exclusive Subtopics

In this section, we try to relate the generative model in Eq. 1 with the intuition that the document relevance should be proportional to the sum of the relevance of the related queries, weighted by the probability that those related queries are themselves relevant:

$$P(d^+|q_0^+;Q) \propto \sum_{q_i \in Q} P(q_i^+|q_0^+)P(d^+|q_i) \tag{2}$$

It is important to notice that some of the following assumptions may not reflect the reality, although are necessary to match the intuition from the generative model, and just reveal the weakness behind of this intuitive model.

To define completely the model, we need to express $P(d^+|\mathcal{I}_\mathcal{Q})$ and $P(\mathcal{I}_\mathcal{Q}|q_0^+)$ in terms of $P(q_i^+|q_0^+)$ and $P(d^+|q_i)$ before we can plug these expressions into Eq. 1.

Document Relevance $P(d^+|\mathcal{I}_\mathcal{Q})$. For convenience we define two sets based on $\mathcal{I}_\mathcal{Q}$: Q^+ is the set of queries indicated as relevant in $\mathcal{I}_\mathcal{Q}$ and Q^-, its complement, contains the queries indicated as not relevant. Naturally we have $Q = Q^+ \cup Q^-$ whatever $\mathcal{I}_\mathcal{Q}$.

We first propose to model the probability $P(d^+|\mathcal{I}_\mathcal{Q})$ as the product of the probabilities that the document is relevant to the queries in Q^+:

Assumption 1 (Positive Set Relevance)

$$P(d^+|\mathcal{I}_\mathcal{Q}) = \prod_{q \in Q^+} P(d^+|q)$$

That is, a document is relevant if it is relevant to all the queries in Q^+. The remaining queries are ignored.

Inter-Query Relevance $P(\mathcal{I}_\mathcal{Q}|q_0^+)$. We assume that two queries q_i and q_j, distinct from each other and distinct from q_0 cannot be relevant simultaneously. In other words, each reformulation represents a distinct, mutually exclusive subtopic of the original query q_0. Returning to the "`apple`" example, we assume that no user is interested simultaneously in the search results for "`apple computer`" (q_1) and "`apple tree`" (q_2) and we have $P(\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^+\}|q_0^+) = 0$. This reduces the sets of $\mathcal{I}_\mathcal{Q}$ with a positive probability to: $\{q_0^+, q_1^-, q_2^-\}$, $\{q_0^+, q_1^+, q_2^-\}$ and $\{q_0^+, q_1^-, q_2^+\}$.

More generally the hypothesis states that $P(\mathcal{I}_\mathcal{Q}|q_0^+) > 0$ only for those $\mathcal{I}_\mathcal{Q}$ where only one reformulation is relevant at a time and all other are non-relevant, i.e. $P(\mathcal{I}_\mathcal{Q}|q_0^+) = 0$ unless $Q^+ = \{q_i, q_0\}$ and $Q^- = \{q_j | j \neq i, 0\}$. After Bayesian inversion this leads to the intuitive model of equation 2.

$$P(d^+|q_0; Q) = \sum_{q \in Q} P(q_i^+|q_0) P(d^+|q_i) \tag{3}$$

If Eq. 3 is applied regardless of the distinct sub-query hypothesis, we can run into problems. Consider for example adding a third query q_3, "`apple seeds`", to the previous example. The relevance of a document d_{fruit}, totally non-relevant to a user interested in computers ($P(d^+_{\text{fruit}}|q_2) = 0$), according to Eq. 3, is:

$$P(d^+_{\text{fruit}}|q_0^+; Q) = \\ P(d^+_{\text{fruit}}|q_0^+) + P(q_1^+|q_0^+) P(d^+_{\text{fruit}}|q_1)$$

if we take into account only q_1 (and therefore respect the *exclusive subtopic* hypothesis). If however we include the closely related query q_3, we obtain

$$P(d^+_{\text{fruit}}|q_0^+; Q) = \\ P(d^+_{\text{fruit}}|q_0^+) + P(q_1^+|q_0^+) P(d^+_{\text{fruit}}|q_1) + P(q_3^+|q_0) P(d^+_{\text{fruit}}|q_3)$$

This shows that the model predicted probability of relevance always increase as we add new related queries. In other words, this model might wrongly predict that an obscure document on the topic of apple fruits is more relevant than "`www.apple.com`" provided we add enough queries related to fruits. The ranking predicted by this model depends on the number of synonym queries entering the sum in Eq. 3, rather than the actual relevance of the documents.

In conclusion, the *Exclusive Subtopics* model promotes documents relevant to the particular meaning of the original query that happens to have more rewrites. Using only one rewrite per subtopic is a possible mend, but it is usually not simple to identify a query subtopics or to match a subtopic to a particular rewrite. Another possibility is to restrict updates to a single rewrite –say q_i–, typically the more informative one. Finally, one can select the related query that boosts the most the particular document relevance and ignore the other queries.

2.2 Naive Bayes

We have seen that the hypothesis of the *Exclusive Subtopics* model are rather restrictive. While it handles correctly queries belonging to mutually exclusive subtopics of the original query, its performance is expected to degrade if several reformulations refer to a common topic. In this section we propose a new model that relaxes some of the previous assumptions.

Document Relevance $P(d^+|\mathcal{I}_\mathcal{Q})$. We propose to model the document probability as the product of the probabilities that the document is relevant to the queries in Q^+, multiplied by the product of the probabilities that the document is not

relevant to the queries in Q^-. The main idea behind, is that a document is relevant to a given $\mathcal{I}_\mathcal{Q}$ if it is relevant to the queries in Q^+ and is not to the queries in Q^-.

Assumption 2 (Specific Set Relevance)

$$P(d^+|\mathcal{I}_\mathcal{Q}) = \prod_{q \in Q^+} P(d^+|q) \times \prod_{q \in Q^-} P(d^-|q)$$

The first term in the product is identical to Assumption 1; The second term adds a specificity restriction and promotes documents relevant to Q^+ and not relevant to Q^-.

Inter-Query Relevance $P(\mathcal{I}_\mathcal{Q}|q_0^+)$. The *Exclusive Subtopics* model makes the assumption that the query reformulations refer to mutually exclusive sub-topics. We relax here this condition and instead we express $P(\mathcal{I}_\mathcal{Q}|q_0^+)$ in terms of the known quantities $P(q^+|q_0^+)$ by using the Naive Bayes hypothesis:

Assumption 3 (Naive Bayes)

$$P(\mathcal{I}_\mathcal{Q}|q_0^+) = \prod_{q \in Q^+} P(q^+|q_0^+) \times \prod_{q \in Q^-} P(q^-|q_0^+)$$

This Naive Bayes assumption is not perfect, albeit convenient. As mentionned earlier, the appropriate solution would be to define a latent variable representing the different topics present among documents and query and develop a model similar to PLSA [10] or LDA [1], but these models involve iterative parameter updating and are numerically too expensive for online computation. Also note that Assumptions 2 and 3 together do not respect the Probability Axioms, but they might be allowable as a tractable approximation.

Although the Naive Bayes Assumption is not perfect, its main properties are in line with what we expect. To see this, consider two reformualtions q_1 and q_2, that are synonyms. We have $P(q_1^+|q_0^+) = P(q_2^+|q_0^+)$ and we call this probability p. $P(q_1^+, q_2^-|q_0^+)$ should be zero while the model states that $P(q_1^+, q_2^-|q_0^+) = P(q_1^+|q_0^+)(1 - P(q_2^+|q_0^+)) = p(1-p) \neq 0$. Consider though for a moment that p is larger than $\frac{1}{2}$; i.e. the two synonym queries tend to be relevant when q_0 is, we observe that $P(\mathcal{I}_\mathcal{Q} = \{q_1^+, q_2^+, \dots\}|q_0^+) \propto p^2$ and $P(\mathcal{I}_\mathcal{Q} = \{q_1^+, q_2^-, \dots\}|q_0^+) \propto p(1-p) < p^2$. In other words, the $\mathcal{I}_\mathcal{Q}$ set where both the synonym queries are relevant has a larger influence when computing the document relevance in Eq. 1 than the other $\mathcal{I}_\mathcal{Q}$ sets. Similarly, $\mathcal{I}_\mathcal{Q} = \{q_1^-, q_2^-, \dots\}$ is dominant when p is small. Finally, if $p \simeq \frac{1}{2}$, $P(\{q_1^+, q_2^+, \dots\}|q_0^+) \simeq P(\{q_1^+, q_2^-, \dots\}|q_0^+) \simeq P(\{q_1^-, q_2^-, \dots\}|q_0^+)$ and the two reformulations have, correctly, little importance. In conclusion, although the assumption gives incorrectly some probability to $\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^-\}$ and $\{q_0^+, q_1^-, q_2^+\}$, this probability is comparatively low.

Another argument in favor of the Naive Bayes assumption is that it interacts well with Assumption 2. It is indeed unlikely that a document relevant to q_1 will not be relevant to q_2 if these are synonyms. Hence $P(d^+|\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^+, q_2^-\})$ and $P(d^+|\mathcal{I}_\mathcal{Q} = \{q_0^+, q_1^-, q_2^+\})$ will be small anyway.

2.3 Naive Bayes Generative Model

The assumptions 2 and 3, only involve terms we already know, i.e. $P(q^+|q_0^+)$ and $P(d^+|q)$. They can therefore be plugged into Eq.1 to estimate $P(d^+|q_0^+;Q)$.

$$P(d^+|q_0^+;Q) = \sum_{\mathcal{I}_Q \in \mathcal{P}(Q)} \left(\prod_{q \in Q^+} P(d^+|q)P(q^+|q_0^+) \times \prod_{q \in Q^-} P(d^-|q)P(q^-|q_0^+) \right) \quad (4)$$

Eq.4 involves a summation over $2^{|Q|}$ terms, i.e. a number growing exponentially with the number of queries, but we show in Appendix A how to compute it efficiently (in linear time proportional to $|Q|$) by taking advantage of the particular form of the hypothesis we have made.

$$P(d^+|q_0^+;Q) = \prod_{q \in Q} \left(P(d^+|q)P(q^+|q_0^+) + P(d^-|q)P(q^-|q_0^+) \right) \quad (5)$$

The linear re-expresion of the *Naive Bayes* generative model bring into evidence how the document relevance is estimated. A document will have a high probability of relevance if it is relevant to queries relevant to q_0 and non-relevant for queries that are not relevant to q_0. In other words, in order to be highly relevant, a document needs to be specific to q_0.

3 Numerical Experiments

We compare the different models we have discussed on a set of queries for which we have editorial judgments. We first describe the data and the evaluation measure. The probabilities $P(q_i^+|q_0^+)$ of a query being relevant given that another query is relevant are extracted from one month of a UK search engine query logs using a state of the art algorithm [2]: When the search engine fails to return the expected results, users often reformulate the original query. We make the assumption that these query rewrites are relevant to the original query. When we observe no rewrite from the query q0 to a query q, we assume that $P(q^+|q_0^+) = 0$, that is we assume that q is not relevant to q_0. We don't use direct estimates of the probabilities $P(d^+|q_i^+)$ (Although these could be extracted from the same logs [7]). Instead, we take advantage of editorial judgments that assign one out of five relevance labels to a large set of document & query pairs because they are expected to be of higher quality. We map these labels to a numerical value that reflect the quality of the pair: *Perfect* pairs are assigned a probability of 1., Excellent is mapped to 0.8, Good to 0.6, Fair and Bad are mapped to 0.3 and 0.1 respectively. We use standard *Discounted Cumulated Gain* scores (DCG, see [13]) to compare the models. DCG is a popular metric for search engine evaluation that is computed as the average over a set of test queries as the sum of the document relevances discounted by a factor that increases with the rank of

the document. The discounting factor is 1 at rank 1 and $\frac{1}{log_b r}$ for $r > 1$, where r is the position in the ranking and the basis b of the logarithm is as often set to 2. Editorial labels offer the most reliable $P(d = r|q_i^+)$ values but they are not available for all document & query pairs. In particular, for the computation and the evaluation of the DCG of the different models we need both the quantities $P(d|q_i^+)$ and $P(d|q+)$ for any given document d that appears in the final ranking of q_0. Consequently, we need to remove from our database all the pairs that miss one of these values. This leaves us with a set of 40 queries q_0 for which we can compute the full models in Eqs. 3 and 5.

3.1 Models Comparison

Because we know the labels of the documents, it is easy to define what the ideal ranking would be and compute the corresponding DCG. We compute this upper bound on the score for a given level of expansion as follows: We first identify the set of documents that appear in the ranking of all the queries involved in the expansion and the original query q_0. We then order these documents according to their labels and we compute the DCG up to the 10th document. Because the more queries we include in the expansion, the more documents we will have to chose from, the ideal DCG can never decrease when more queries are added. We examine the effect of adding successively more queries to the expansion set. In practice, this resolves to starting by setting all $P(q_i|q)$ to zero to obtain the model

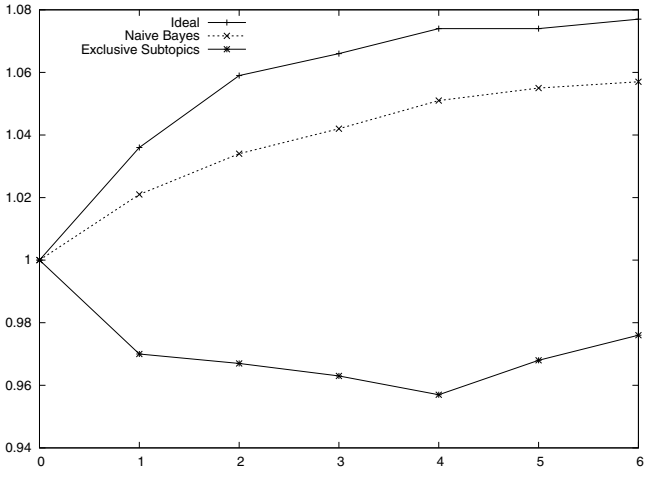


Fig. 1. Relative DCG gain (y axis) as a function of the number of expensions (x axis). We observe that the Exclusive Subtopic model actually impacts negatively the DCG score while the Naive Bayes Model improves the results steadily with the number of expansions. At six expansions, the generative model achieves a relative DCG score of 1.057 compared to 1.077 for the ideal model.

without expansion. Then the query with highest $P(q_i|q_0^+)$ is set to its real value to compute the model with one expansion. The other expansion levels are obtained similarly by adding one by one the remaining queries according to the highest remaining $P(q_i|q_0^+)$ value. To allow a fair comparison, the implementation of the Naive Bayes model of Eq 5 also restricts the number of involved queries. In Figure 1 we take the DCG score with no expansion as the reference and we compute the relative scores of the different models for up to 6 expansions. The same results are reported in Table 1. We see that the *Naive Bayes* model performs better than the *Exclusive Subtopics* model at all levels of expansion and that the DCG score increases with the number of expansions.

Table 1. Average relative DCG (y axis) as a function of the number of expansions (x axis)

Model	0	1	2	3	4	5	6
Exclusive Subtopics	1	0.970	0.967	0.963	0.957	0.968	0.976
Naive Bayes	1	1.021	1.034	1.042	1.051	1.055	1.057
Ideal	1	1.036	1.059	1.066	1.074	1.074	1.077

4 Discussions and Conclusions

Besides the fact that many queries contain errors or are inherently ambiguous, there is also typically a large gap between the natural language of users and the representation search engine use to answer the queries. A possible solution to narrow this gap, known generically under the name of *query refinement*, is to re-interpret or re-write the user query in a way that is more adequate to the search engine. Many different techniques of query refinement have been developed, based on various source of data. Nevertheless, the question of how to combine the different refinements suggested by these various techniques remains an open question. A simple, intuitive method is to issue the different query reformulations to the engine and add the scores that a document achieve in the different result sets. In this work we argue and we show on the basis of numerical experiments that this simple technique is not adequate. Instead, we propose a simple generative model based on the probability of relevance of the different reformulations and we show that it out-performs the intuitive model.

References

1. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. 3, 993–1022 (2003)
2. Boldi, P., Bonchi, F., Castillo, C., Donato, D., Gionis, A., Vigna, S.: The query-flow graph: model and applications. In: CIKM 2008: Proceeding of the 17th ACM conference on Information and knowledge mining, pp. 609–618. ACM, New York (2008)

3. Buehrer, G., Stokes, J.W., Chellapilla, K.: A large-scale study of automated web search traffic. In: AIRWeb 2008: Proceedings of the 4th international workshop on Adversarial information retrieval on the web, pp. 1–8. ACM, New York (2008)
4. Chapelle, O., Zhang, Y.: A dynamic bayesian network click model for web search ranking. In: WWW 2009: Proceedings of the 18th international conference on World wide web, pp. 1–10. ACM, New York (2009)
5. Craswell, N., Szummer, M.: Random walks on the click graph. In: SIGIR 2007: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 239–246. ACM Press, New York (2007)
6. Craswell, N., Zoeter, O., Taylor, M., Ramsey, B.: An experimental comparison of click position-bias models. In: First ACM International Conference on Web Search and Data Mining WSDM 2008 (2008)
7. Dupret, G., Mendoza, M.: Recommending better queries based on click-through data. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 41–44. Springer, Heidelberg (2004)
8. Dupret, G., Piwowarski, B.: A user browsing model to predict search engine click data from past observations. In: Press, A. (ed.) Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval (2008)
9. Guo, F., Liu, C., Kannan, A., Minka, T., Taylor, M., Wang, Y.M., Faloutsos, C.: Click chain model in web search. In: WWW 2009: Proceedings of the 18th international conference on World wide web, pp. 11–20. ACM, New York (2009)
10. Hofmann, T.: Probabilistic Latent Semantic Indexing. In: Proceedings of the 22nd Annual ACM Conference on Research and Development in Information Retrieval, pp. 50–57. Berkeley, California (August 1999)
11. Joachims, T.: Optimizing search engines using clickthrough data. In: KDD 2002: Proceedings of the eighth ACM SIGKDD, pp. 133–142. ACM Press, New York (2002)
12. Jones, R., Fain, D.C.: Query word deletion prediction. In: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval (2003)
13. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of ir techniques. ACM Transactions on Information Systems (ACM TOIS) 20(4), 222–246 (2002)
14. Radlinski, F., Joachims, T.: Active exploration for learning rankings from clickthrough data. In: Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD). ACM, New York (2007)
15. Robertson, S.E.: The probability ranking principle in ir. Journal of Documentation-33(4), 294–304 (1977)
16. Velez, B., Weiss, R., Sheldon, M.A., Gifford, D.K.: Fast and effective query refinement. In: Proceedings of the 20th annual international ACM SIGIR conference on Research and development in information retrieval (1997)

A Fast Evaluation

In order to estimate the document relevance, the *Naive Generative Model* in Eq. 4 involves a sum over an exponential number of terms, which makes it unpractical. Fortunately there is an equivalent formulation which cost is linear in the number of related queries. The basic principle is to add related queries

one at a time: Consider $Q_1 = \{q_0, q_1\}$ and compute $P_1(d^+|q_0; \{q_0, q_1\})$. Then use this last expression as the new $P(d^+|q_0) \leftarrow P_1(d^+|q_0; \{q_0, q_1\})$ and update it using $Q_2 = \{q_0, q_2\}$, etc. We show below that this is indeed equivalent to the original formula.

Let us define $\mathcal{I}_{\mathcal{Q}_n} = \{q_0, q_1, ..., q_{n-1}\}$ as a set of n queries, where q_0 is the original query submitted by the user. If we also introduce $r_{q^+} := P(d^+|q)P(q^+|q_0)$ and $r_{q^-} := P(d^-|q)P(q^-|q_0)$, them the *Naive Generative Model* can be rewritten as

$$P(d^+|q_0; Q) = \sum_{\mathcal{I}_{\mathcal{Q}} \in \mathcal{P}(\mathcal{Q})} \left(\prod_{q^+ \in Q^+} r_{q^+} \times \prod_{q^- \in Q^-} r_{q^-} \right)$$

Suppose $Q_{n+1} = Q \cup q_{n+1}$ where q_{n+1} is a new query related to q_0 ($q_{n+1} \notin Q$). $P(d^+|q_0; Q_{n+1})$ can be rewritten in terms of $P(d^+|q_0; Q)$ as follows:

$$\begin{aligned}
P(d^+|q_0; Q_{n+1}) &= \sum_{\mathcal{I}_{\mathcal{Q}} \in \mathcal{P}(\mathcal{Q}_{n+1})} \prod_{q^+ \in Q^+_{n+1}} r_{q^+} \times \prod_{q^- \in Q^-_{n+1}} r_{q^-} \\
&= \sum_{\mathcal{I}_{\mathcal{Q}} \in \mathcal{P}(\mathcal{Q}_n)} \left(r_{q^+_{n+1}} \prod_{q^+ \in Q^+_n} r_{q^+} \times \prod_{q^- \in Q^-_n} r_{q^-} + r_{q^-_{n+1}} \prod_{q^+ \in Q^+_n} r_{q^+} \times \prod_{q^- \in Q^-_n} r_{q^-} \right) \\
&= (r_{q^+_{n+1}} + r_{q^-_{n+1}}) \sum_{\mathcal{I}_{\mathcal{Q}} \in \mathcal{P}(\mathcal{Q}_n)} \left(\prod_{q^+ \in Q^+_n} r_{q^+} \times \prod_{q^- \in Q^-_n} r_{q^-} \right) \\
&= (r_{q^+_{n+1}} + r_{q^-_{n+1}}) \times P(d^+|q_0; Q_n)
\end{aligned}$$

and because we also have that $P(d^+|q_0; q_1) = r_{q_0^+} \times (r_{q_1^+} + r_{q_1^-})$ we see that $P(d^+|q_0; Q)$ can be computed in $|Q|$ steps.

Evaluation of Query Performance Prediction Methods by Range*

Joaquín Pérez-Iglesias and Lourdes Araujo

Universidad Nacional de Educación a Distancia
Madrid 28040, Spain
joaquin.perez@lsi.uned.es, lurdes@lsi.uned.es

Abstract. During the last years a great number of Query Performance Prediction methods have been proposed. However, this explosion of prediction method proposals have not been paralleled by an in-depth study of suitable methods to evaluate these estimations. In this paper we analyse the current approaches to evaluate Query Performance Prediction methods, highlighting some limitations they present. We also propose a novel method for evaluating predictors focused on revealing the different performance they have for queries of distinct degree of difficulty. This goal can be achieved by transforming the prediction performance evaluation problem into a classification task, assuming that each topic belongs to a unique type based on their retrieval performance. We compare the different evaluation approaches showing that the proposed evaluation exhibits a more accurate performance, making explicit the differences between predictors for different types of queries.

Keywords: Information Retrieval, Evaluation, Query Performance Prediction.

1 Introduction

Research on Query Performance Prediction, or QPP, has attracted growing attention from the information retrieval (IR) community in the last years. This topic is focused on predicting the retrieval effectiveness of a query, that is, the quality of a search engine's response to a submitted query. The relevance of a subset of documents returned by a search engine is usually measured by means of the user's relevance judgements and the Average Precision (AP). Therefore, a query which obtains a high AP value can be considered as 'Easy', since the retrieval model was able to return a relevant subset of documents. Otherwise if the query obtains a low AP it is considered as 'Hard'. Having the ability to predict the performance of a query can help us to apply some specific techniques in order to improve the relevance of the returned documents. This improvement

* This paper has been funded in part by the Spanish MICINN projects NoHNES (Spanish Ministerio de Educación y Ciencia - TIN2007-68083) and by MAVIR, a research network co-funded by the Regional Government of Madrid under program MA2VICMR (S2009/TIC-1542).

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 225–236, 2010.

can be achieved with classic techniques such as relevance or pseudo-relevance feedback, or with the use of a different index whose content is more related to the submitted query.

In the last years several methods have been proposed to deal with QPP, which fall into one of the following groups, depending on the information used to make predictions:

- **Pre-retrieval** methods, where the estimations are computed using query terms statistics, such as collection frequency (CF), document frequency (DF) or query length. An extensive description about this type of prediction methods can be found in the work developed by He and Ounis [1].
- **Post-retrieval** methods, which are usually more complex and are computed using the document ranked list returned by the search engine combined with other collection statistics. The most representative example within this type of prediction methods is Clarity Score. It was proposed by Cronen-Townsend et al. [2] and it is based on measuring the divergence between the relevance language model[1] and the collection language model, where a high divergence suggests a well-performing query.

In general post-retrieval methods achieve better estimations since they use more information to compute predictions, although this entails a considerable increase of the computational cost.

Prediction methods are evaluated by means of correlation coefficients. The goal is to measure for each topic the correlation degree between the estimated value, obtained with the proposed prediction method, and the Average Precision value. Therefore a prediction method is considered more accurate if it obtains a higher value of correlation between the actual values and the generated prediction values.

The correlation degree between two random variables measures the dependence between both variables. This dependence in general is quantified with a real number in the range $[-1, 1]$, where 1 means a perfect direct correlation, -1 means a perfect inverse correlation and 0 means no correlation at all. Therefore, for values of correlation close to zero no dependence between both variables can be observed, although this fact does not imply a total independence between both random variables. Besides, high values of correlation, positives or negatives, suggest, but do not assure, a possible dependence between both variables.

The evaluation based on correlation coefficients usually produces very similar results for the different prediction methods as it can be observed in the related literature [3]. These values are usually hard to interpret, since the differences among the obtained correlation coefficients are sometimes too low.

The evaluation of prediction methods should be focused on their application to specific contexts, and therefore the evaluation framework should help us to decide if a prediction method is suitable for that specific context. Current evaluation, based on correlation, provides a very coarse measure of the method

[1] The relevance language model is built using a subset of the documents returned by the query.

accuracy, ignoring some important details of the real performance. For instance an evaluation more focused on the predictors application should be able to answer questions like: Is a new method able to outperform others in relation with the detection of 'Hard' queries?

In order to stress these differences in terms of prediction performance we propose a new method for measuring the effectiveness of a query performance predictor which provides information for different levels of retrieval quality. This task can be achieved by assuming a discrete classification of topics, that is, assuming that each topic belongs to a unique type based on their retrieval performance. This assumption avoids the drawbacks which arise with the use of the correlation approach, since it allows us to measure the predictor performance partially, i.e. for each type of topic, and globally as it is done by correlation coefficients.

The rest of this paper is organised as follows. In Section 2 current evaluation approaches are introduced, with a special emphasis on describing their main weaknesses. Section 3 is devoted to describe a new evaluation framework for Query Performance Prediction, which we introduce in order to overcome some of the previously analysed limitations. In Section 4 the proposed evaluation method is tested against current approaches using a standard TREC collection, and a detailed analysis of the obtained results is carried out. Finally, the main conclusions drawn from this work appears at Section 5.

2 Current Evaluation Approaches

In the context of Query Performance Prediction *Pearson*(r) and *Kendall*(τ) are the most commonly applied correlation methods[2] to evaluate the accuracy of the estimations.

Pearson is a parametric method, which assumes a linear relationship between both data series indicating the strength and direction of this relationship, whereas *Kendall* computes the correlation value counting the pairwise swappings needed to transform one ranking into the other. *Kendall* is a non-parametric method, and thus does not make assumptions about the input data, providing less information about the data relationship. Therefore, while *Pearson* is focused on establishing if both data series are produced by two linearly dependent functions, *Kendall* measures how similar are the orderings produced by the prediction method in comparison to the one produced by the 'actual' values. The obtained *Kendall's* τ coefficient can be interpreted as a degree of certainty, which indicates if a topic would obtain a higher AP value than other, although the AP value itself cannot be predicted as *Pearson's* r does.

Due to their different nature both methods frequently produce dissimilar values, and thus a direct comparison between both coefficients is not possible.

Pearson drawbacks have been extensively treated in the literature. It is well-known that *Pearson* produces very different results for data series which show

[2] In some works we can find Spearman (ρ) correlation coefficient, although its use remains rare in this context.

strong dependence, whenever a data outlier appears or a small fraction of the data takes values far from the mean. Another known problem occurs when both random variables show a strong dependence but through a non linear relationship. In this case the correlation coefficient r will be low even if the data are strongly correlated. The main problems of the application of *Pearson* correlation coefficient in the context of QPP were previously pointed out by Hauff et al.[4].

Due to the described problems many works on query performance prediction report their results with the *Kendall* correlation coefficient at the expense of using a less informative evaluation measure.

Although *Kendall* is considered as a more appropriate measure to evaluate estimations, in our opinion this measure does not make explicit the real effectiveness of the predictor, since we only obtain a unique value describing the average accuracy, but ignoring the performance for different types of queries.

Kendall is only focused in the number of disordered elements and the distance of them to the position where they should be placed. Therefore *Kendall* gives the same importance to all elements within the data series. On the context of QPP we can be more interested on analysing the performance on topics of different difficulty ('Hard' or 'Easy'), that is, topics placed at the top or bottom of the topic list sorted by AP. For instance a key factor that should be highlighted in the operation of a predictor is its ability to detect those topics that obtain a low AP value. This feature is of extreme importance for tasks such as pseudo-relevance feedback, since it will help us decide in which cases to expand the original user query, as it has been recently studied by He et. al [5].

For instance, let us consider the data series: $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $Y = \{\mathbf{4}, 2, 3, \mathbf{1}, 5, \mathbf{10}, 7, 8, 9, \mathbf{6}\}$, where we consider elements 1 to 5 as the *'best'* elements and the rest as the *'worst'* elements. Since elements 1,4,6 and 10 are not placed at their right position a Kendall correlation of 0.46 is obtained, which suggests a significant but not strong correlation between data. On the other hand if we are only interested on evaluating by type of elements, i.e. *'best'* and *'worst'*, we can conclude that both data series X and Y group both types of elements in the same fashion, as best elements are grouped within the first five positions, and therefore the worst elements are within the last places.

On the other hand, this drawback can not be overcome by measuring the Kendall correlation partially for each type of elements. For instance, if we consider the next data serie $Z = \{6, 7, 8, 9, 10, 1, 2, 3, 4, 5\}$ produced by a predictor, where as before we consider elements 1 to 5 as the *'best'* elements and the rest as the *'worst'* elements. The τ obtained between X and Z for the so-called *'best'* and *'worst'* elements is in both cases equals to 1. However it can be observed that the predictor shows a pretty poor performance since it has predicted the best elements as worst and viceversa.

Some extensions to *Kendall* try to overcome this drawback assigning a relative importance for each element or groups of elements to the final τ value. This family of Kendall variants is known as 'Weighted Kendall', and are usually applied to measure the similarity between responses of different search engines [6,7]. Using this approach it is possible to measure if a method shows a better

performance for a specific type of topics than other, although we must set a suitable weight for each element. Another weakness of this approach is that it is not able to provide partial results for the different types of topics.

3 Evaluating by Range

Previous sections have introduced the idea that using correlation coefficients as a quality measure only provides a global view of the predictor performance, ignoring its specific behaviour on different types of topics, i.e. topics of different difficulty. However, it is expected that different prediction methods show different performance on 'Easy' or 'Hard' topics. Detecting this disparity in terms of prediction effectiveness can be a key factor in order to apply these methods to improve the retrieval quality for a specific type of queries.

In this section we propose a new QPP evaluation framework with a two fold goal: *a*) evaluate the ability of the method to detect 'Easy' or 'Hard' queries; and *b*) to make explicit the accuracy of the method for different types of topics.

For the development of this evaluation framework, we should be able to partition the topic set of study into n blocks of interest, where each topic is uniquely assigned to one of these partitions by its corresponding AP value, which establishes the retrieval quality of the partition. Thus, the best topics in terms of AP would be assigned to the first partition and likewise the *n-th* partition groups the worst topics. The same process is applied using the values provided by the prediction method, instead of AP.

After partitioning the full set of topics, each topic is labelled according to the prediction and the average precision obtained. Therefore, it is possible to test for each topic if the AP value and the estimation belong to the same partition. Thus, the evaluation of the quality of this labelling resembles a problem of classification evaluation.

Grouping Data

An interesting problem which arises with the proposed evaluation is how to group topics by their retrieval performance. The application of a suitable method to group topics is a key task for the evaluation of QPP methods, since different systems gives rises to different retrieval performance.

For instance, we can imagine a hypothetical search system where almost all queries obtain as response a set of documents which satisfy the user. In this case the application of a prediction method is not necessary, since it is known that all queries are correctly answered. A similar case occurs when a search system is unable to respond correctly to a great majority of queries. Therefore, a prediction method is only useful when there is a significant divergence in the quality of the search engine responses. In general, this divergence occurs in actual search systems.

Focusing on the TREC environment, a typical run usually shows an exponential probability distribution[3], as Figure 1 illustrates, where a great number of topics obtain a low AP value.

In order to adapt the different partitions of topics to each search system, we propose to group them following the probability distribution of the AP values, which represents the overall search system performance. For this task the k-means [8] clustering algorithm is a suitable approach, since our goal is to create groups such that the distance between elements in the same group is minimum and the distance among the means of each group is maximum. More formally, given a set of topic AP values $(t_1, t_2, \cdots, t_n)$, the k-means clustering aims to partition the n observations into k sets $(k < n)$ $C = \{C_1, C_2, \ldots, C_k\}$ so as to minimize the within-cluster distance to the mean:

$$\arg\max_{C} \sum_{i=1}^{k} \sum_{t \in C_i} (t - \bar{C}_i)^2$$

where $\bar{C}_i$ is the mean in C_i.

The k-means algorithm is not a deterministic method of clustering because it depends on the initial selection of cluster means. In order to circumvent this problem we propose to set the initial means in such a way that they do not introduce any bias in the construction of the groups. Therefore, initial means should be uniformly distributed along the whole set of topics in such a way that, the number of topics between the proposed means is roughly the same. This method can be implemented computing the percentiles for each desired group based on the next equation: $\bar{C}_i = percentile\left(100 \cdot \frac{i-1}{n+1}\right)$ where $\bar{C}_i$ is the initial mean of the *i-th* cluster, and n is the number of clusters. For instance if n=3, the initial means will be fixed at the *25th*, *50th* and *75th* percentile of the data.

Using k-means in combination with the initialisation method proposed above ensures that the groups created will depend on the data distribution. A proof of this fact is that those groups with a larger number of topics appear where the density function has a maximum, as observed in Figure 1.

The k-means algorithm requires a parameter indicating the desired number of groups in which the input data will be divided. In our opinion a reasonable approach is to set this value manually, since this number only depends on the granularity level desired for the study[4]. For example, a typical setup can include three groups: 'Easy', 'Average' and 'Hard' topics. Obviously, in order to compare different prediction methods the number of partitions must be the same for all the evaluated methods.

The described methodology for grouping topics is also applied to the prediction values. Applying to both data series the same partition method has a strong effect on the results of typical evaluation measures as *precision* or *recall*. In general, those prediction methods whose probability distribution is not similar

[3] Although best runs approach a Gaussian distribution as the MAP value increases.

[4] Although there are several methods which set automatically the number of clusters for the k-means algorithm.

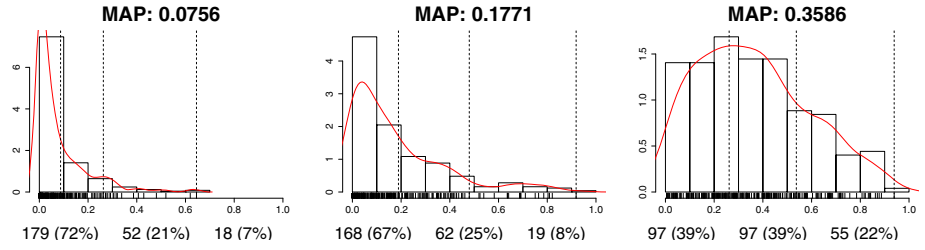


Fig. 1. AP histograms, density function and the obtained partitions applying k-means with k=3 for the worst, average and best run submitted to the Robust 2004 track

to the AP distribution produce poorer results, because this dissimilarity leads to the creation of partitions with different sizes, and this affects the evaluation measures.

In the next section we apply the introduced evaluation framework to a subset of different prediction methods in a standard TREC collection, in order to test the ability of our proposal to circumvent some of the drawbacks of the current approaches for the evaluation of QPP methods.

4 Experiments and Results

For the experimental evaluation of our proposal the set of documents from TREC Disk4 & 5, along with the full set of topics from the Robust 2004 track [9], are employed. We have selected this set of topics since a majority of prediction methods obtain their best results when they are tested with them, as it appears in related literature [3].

All prediction methods are executed against a base run which was obtained using the query likelihood language modeling [10], with a Dirichlet prior smoothing parameter [11] equal to 1500[5]. This run achieves a MAP value of 0.24, which is of the order of a typical TREC run for this collection.

A significant set of prediction methods considered as state of the art have been implemented. For the pre-retrieval case, we have tested some of the methods proposed by He et. al [1], including those based on the query terms IDF or ICTF, such as the maximum IDF, the average ICTF, Simplify Clarity Score (SCS) and the QueryScope method based on the number of documents returned by each query term. On the other hand, the post-retrieval methods tested include Clarity Score [2], for which the number of feedback documents have been fixed to 500, and the ScoreDesv method, which is based on measuring the standard deviation of the top k scores, fixing the top k to 120, as recommended in [12].

The set of topics is grouped into three blocks: 'Easy', for topics with the highest AP; 'Hard', for those with lowest AP; and 'Average', for the rest. The details of the different groups obtained after the application of the k-means

[5] For this task The Lemur Toolkit software was employed.

Table 1. Statistics for 'Easy', 'Average' and 'Hard' topic groups, including number of topics per group, maximum, mean and minimum AP per group and the AP standard deviation per group

	Num	Max	Mean	Min	Sd
Hard	121	.02	.001	.0005	.006
Average	97	.21	.10	.02	.05
Easy	31	.91	.41	.21	.17

algorithm appear in Table 1. As expected, for a typical TREC run the largest group corresponds to the set of 'Hard' topics, while the group with the smallest number of topics corresponds to the 'Easy' partition.

In order to describe in detail the performance obtained by the tested prediction methods, several measures from the classification topic can be applied. The wide range of available measures allow us to define the experimental setup as a function of the desired type of study. For the current setup we have decided to apply the measures described below with the main goal of highlighting the differences in terms of prediction not shown by the correlation coefficients.

The simplest approach to compare the accuracy of the different evaluated methods is to compute the number of hits per partition and the total number of hits. We will employ the classic F-measure, since it is able to combine precision and recall in a single number and can be applied globally and for each defined partition. Formally, the F-measure is defined as $2 \cdot \frac{precision \cdot recall}{precision + recall}$.

In previous measures misclassified elements are equivalent. However, in QPP all errors should not penalise the measure in the same manner. For instance, in a set-up where we have decided to label topics as 'Easy', 'Average' and 'Hard', predicting a topic as 'Hard' when it is actually 'Easy' should imply a greater penalty than if the element had been misclassified as 'Average', a partition closer to 'Easy'. For this purpose we introduce a new measure: the Distance Based Error Measure (DBEM), which is able to indicate the global performance along all the partitions but focused only on the misclassified topics. We define the distance between two partitions C_i and C_j in a set of topics that have been grouped in k partitions as

$$distance(C_i, C_j) = \|i - j\|$$

where $0 < i,j \leq k$. Then

$$DBEM = \frac{\sum_i^n distance(P_{t_i}, C_{t_i})}{\sum_i^n max\left[\forall_{j \in n} distance(P_{t_i}, P_{t_j})\right]}$$

where P_{t_i} is the predicted partition for the topic t_i, C_{t_i} is the AP partition for the topic t_i, and n is the total number of topics. This is, the distance between all topics normalised by the maximum possible distance, where lower distances imply better performances.

Results

Table 2 shows the number of correctly classified topics for the whole set of topics and for each partition. These results indicate that the subset of methods based on IDF is strongly biased toward grouping topics as 'Hard', which leads them to achieve the best result for the 'Hard' partition and very poor results for the 'Average' and 'Easy' partitions. Furthermore, this subset of prediction methods obtain very similar results, as it is shown by the total number of hits achieved by them. However, this similarity in their results is not captured by the *Pearson* and *Kendall* correlation. Moreover, some important differences appear in the correlation coefficients obtained by these methods, suggesting a different performance for these predictors, which has proved false using the simple accuracy. For instance, according to *Pearson* IDFMin (r=0.24) should outperforms clearly IDFAvg (r=0.17), while according to *Kendall* we should conclude exactly the opposite, since IDFMin obtains a τ of 0.16 to be compared with the 0.32 obtained by IDFAvg.

One key factor when comparing prediction methods by their accuracy is to observe the performance not only globally but for each partition too. For example, the obtained global accuracy for the QueryScope method will guide us to the wrong conclusion of a worse performance of this compared to the IDF based methods. But if we check the partial results of QueryScope we can observe a much better performance for this method in the detection of 'Easy' and 'Average' topics in comparison with the IDF based methods. Therefore, we can conclude that the strong performance of the IDF based methods on 'Hard' topics is a consequence of considering more than 95% of topics as 'Hard', which does not correspond to the real performance of the tested run.

The global accuracy is not able to detect the previous situation either, because it is computed as a sum of partial accuracies. However, this weakness is overcome by the F-measure for the whole set of topics, as it can be observed in Table 3 where the IDF based methods obtains the worst results among all predictors.

Table 2. Results for the proposed predictors. The first three columns show the number of hits per type of topic, including in brackets the whole number of topics classified as the type of the column title. Besides, the last three columns show the total accuracy $\frac{hits}{total}$ and the Pearson and Kendall correlation coefficient obtained.

	Hard	Average	Easy	Total	Accuracy	Pearson	Kendall
AVICTF	73(122)	47(110)	9(17)	129	0.52	0.45	0.26
IDFAvg	120(245)	1(3)	1(1)	122	0.49	0.17	0.32
IDFDesv	119(245)	1(3)	1(1)	121	0.48	0.12	0.25
IDFMax	118(243)	1(5)	1(1)	120	0.48	0.15	0.32
IDFMin	121(245)	1(3)	1(1)	123	0.49	0.24	0.16
QScope	91(171)	22(67)	5(11)	118	0.47	0.37	0.18
SCS	60(87)	50(109)	18(53)	128	0.51	-0.45	-0.26
CS	78(110)	54(103)	13(36)	145	0.58	0.51	0.41
ScoreDesv	84(128)	47(91)	17(30)	148	0.59	0.55	0.40

Table 3. Results for the proposed predictors. The first three columns show the F-measure for each type of topic. Besides, the last four columns show the F-measure for the whole set of topics, the Distance Based Error Measure and the Pearson and Kendall correlation coefficient obtained.

	Hard	Average	Easy	Total	DBEM	Pearson	Kendall
AVICTF	0.60	0.45	0.37	0.51	0.32	0.45	0.26
IDFAvg	0.65	0.02	0.06	0.33	0.39	0.17	0.32
IDFDesv	0.65	0.02	0.06	0.33	0.39	0.12	0.25
IDFMAX	0.63	0.02	0.06	0.33	0.39	0.15	0.32
IDFMin	0.66	0.02	0.06	0.36	0.38	0.24	0.16
QScope	0.62	0.26	0.23	0.43	0.35	0.37	0.18
SCS	0.58	0.48	0.43	0.52	0.34	-0.45	-0.26
CS	0.67	0.54	0.39	0.59	0.29	0.51	0.41
ScoreDesv	0.67	0.50	0.56	0.59	0.27	0.55	0.40

Table 4. Confusion Matrix for Clarity Score (left) and ScoreDesv (right), the number of correctly classified topics appears in boldface

	Hard	Average	Easy	Total
Hard	**78**	36	7	121
Average	27	**54**	16	97
Easy	5	13	**13**	31
Total	110	103	36	**145**

	Hard	Average	Easy	Total
Hard	**84**	35	2	121
Average	39	**47**	11	97
Easy	5	9	**17**	31
Total	128	91	30	**148**

A final conclusion that can be extracted from the results in Table 2 is that, as it was expected, the most accurate methods in terms of grouping predictions are CS and ScoreDesv, which outperform clearly the rest of prediction methods.

Table 3, shows the F-measure obtained, which can help us to extract some other conclusions. These results show that in general prediction methods show a better performance detecting 'Hard' topics than 'Average' or 'Easy' topics, something not revealed by means of the correlation coefficients. Besides, while both correlation coefficients suggest an equivalent performance for SCS and AVICTF prediction methods, using the F-measure we observe how SCS outperforms in more than a 16% the AVICTF in relation with the detection of 'Easy' topics.

In relation with the most accurate methods, CS and ScoreDesv, although the correlation methods and the global F-measure suggest a similar performance, a more detailed comparison leads us to observe some interesting differences between them. For instance, with the partial F-measure we can observe that although CS

is slightly better for 'Hard' and 'Average' topics, ScoreDesv improves CS around a 43% detecting 'Easy' topics, which makes ScoreDesv a more reliable option in those contexts where we would expect an accurate detection of 'Easy' topics.

The differences between these last prediction methods are shown as well by the proposed DBEM measure. Although this measure shows a strong correlation with the rest of the global performance measures, as it appears in Table 5, it reveals some important details which are not shown with the rest of measures. Focusing on CS and ScoreDesv, DBEM suggests a minor fail ratio for the last method. This fact can be confirmed observing the confusion matrix of both methods, which appears in Table 4. In this table we observe that while the number of misclassified topics is similar for both methods (104 for CS and 101 for ScoreDesv), CS is labelling 12 topics as 'Easy' when they are actually 'Hard' or viceversa, while these errors only occurs 7 times with ScoreDesv. This error rate implies that the proportion of strong errors by CS is around an 11% against the 6% obtained by the ScoreDesv.

Finally it should be highlighted that the proposed measures applied show a strong correlation with the classic evaluation approach, as it appears in Table 5. Thus the proposed evaluation method provides an information equivalent to correlation approach, but showing a higher level of detail for the topics subset of interest.

Table 5. Pearson correlation coefficient between pairs of proposed measures

	Accuracy	**F-Measure**	**DBEM**
Pearson	0.78	0.77	-0.95
Kendall	0.77	0.66	-0.57

5 Conclusions

In this paper a novel method for the evaluation of Query performance Prediction Methods has been introduced. The goal of this proposal is to avoid some of the drawbacks which appear with the use of correlation coefficients when they are applied to evaluate Query Performance Prediction methods.

Our proposal overcomes previous drawbacks avoiding the use of correlation coefficients, and transforming the performance prediction evaluation into a classification task by assuming that each topic belongs to a unique type based on their retrieval performance. For this task we have proposed an automatic method to group topics based on their retrieval quality, according to the overall retrieval quality of the search system in study.

While the application of correlation coefficients to this topic can hide the specific performance of prediction methods for different types of topics, our proposal makes explicit these differences guiding us to the selection of the more suitable method depending on the application context. Furthermore, each topic is automatically labelled by their retrieval effectiveness according to the prediction

method. Based on this label, a system would be able to decide which is the most suitable technique to improve the quality of the response for this topic, opposite to current approach where this decision is taken based on a numeric value assigned by the prediction method.

The novel evaluation framework has been tested against a set of different prediction methods, providing with a more detailed information about the tested predictors performance. Besides, the proposed measures have shown a strong correlation degree with the current evaluation, which suggests a similar behaviour of our proposal with correlation approach but being at the same time able of revealing some performance differences that are not detected with the current approaches.

References

1. He, B., Ounis, I.: Inferring Query Performance Using Pre-retrieval Predictors. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 43–54. Springer, Heidelberg (2004)
2. Cronen-Townsend, S., Zhou, Y., Croft, W.B.: Predicting query performance. In: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR 2002. ACM Press, New York (2002)
3. Hauff, C.: Predicting the Effectiveness of Queries and Retrieval Systems. PhD thesis, Univ. of Twente, Enschede (January 2010)
4. Hauff, C., Azzopardi, L., Hiemstra, D.: The combination and evaluation of query performance prediction methods. In: ECIR, pp. 301–312 (2009)
5. He, B., Ounis, I.: Studying query expansion effectiveness. In: ECIR, pp. 611–619 (2009)
6. Melucci, M.: Weighted rank correlation in information retrieval evaluation. In: Kuriyama, K. (ed.) AIRS 2009. LNCS, vol. 5839, pp. 75–86. Springer, Heidelberg (2009)
7. Yilmaz, E., Aslam, J.A., Robertson, S.: A new rank correlation coefficient for information retrieval. In: SIGIR 2008: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval, pp. 587–594. ACM, New York (2008)
8. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann Series in Data Management Sys. Morgan Kaufmann, San Francisco (June 2005)
9. Voorhees, E.M.: Overview of the TREC 2004 Robust Retrieval Track. In: Proceedings of the Thirteenth Text REtrieval Conference, TREC (2004)
10. Ponte, J.M., Croft, W.B.: A language modeling approach to information retrieval. In: SIGIR 1998: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval, pp. 275–281. ACM, New York (1998)
11. Zhai, C., Lafferty, J.: A study of smoothing methods for language models applied to ad hoc information retrieval. In: SIGIR 2001: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 334–342. ACM, New York (2001)
12. Pérez-Iglesias, J., Araujo, L.: Ranking list dispersion as a query performance predictor. In: Azzopardi, L., Kazai, G., Robertson, S., Rüger, S., Shokouhi, M., Song, D., Yilmaz, E. (eds.) ICTIR 2009. LNCS, vol. 5766, pp. 371–374. Springer, Heidelberg (2009)

Mining Large Query Induced Graphs towards a Hierarchical Query Folksonomy

Alexandre P. Francisco[1,⋆], Ricardo Baeza-Yates[2], and Arlindo L. Oliveira[1]

[1] INESC-ID / CSE Dept, IST, Tech Univ of Lisbon, Portugal
[2] Yahoo! Research Barcelona, Spain & Santiago, Chile

Abstract. The human interaction through the web generates both implicit and explicit knowledge. An example of an implicit contribution is searching, as people contribute with their knowledge by clicking on retrieved documents. Thus, an important and interesting challenge is to extract semantic relations among queries and their terms from query logs. In this paper we present and discuss results on mining large query log induced graphs, and how they contribute to query classification and to understand user intent and interest. Our approach consists on efficiently obtaining a hierarchical clustering for such graphs and, then, a hierarchical query folksonomy. Results obtained with real data provide interesting insights on semantic relations among queries and are compared with conventional taxonomies, namely the ODP categorization.

1 Introduction

Nowadays the Web is the biggest representation of human knowledge, where people contribute with content either explicitly or implicitly. An example of an implicit contribution is searching, as people contribute with their knowledge by clicking on retrieved documents. Thus, queries submitted to search engines carry implicit knowledge and they can be seen as equivalent to tags associated to clicked documents. An interesting challenge is then to extract relevant semantic relations from query logs, which have several interesting applications. For instance, ranking algorithms, query recommendation systems and advertisement systems integrate such semantic information to improve their results.

In this paper we discuss query classification and meaning, and not URL tagging and folksonomies. We use click-data to infer relationships and similarities among queries. Then, by finding closely related queries, we are able to define a hierarchical query folksonomy associating tags to queries. Note that this approach may associate a tag to a query even if that tag is not part of the query, leading to query contextualization. Our approach relies on graphs to represent relations among queries and on efficient graph mining techniques to uncover relations. According to SearchEngineWatch.com, the number of queries per day is of the order of hundreds of millions, leading to huge query graphs. On the other hand, the number of potential relations and applications is also huge.

⋆ Work done while visiting Yahoo! Research Barcelona. Email: aplf@ist.utl.pt

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 237–242, 2010.

Our study follows recent works on the analysis of query graphs [3,10], which introduce the notion of click induced graph and present several results concerning semantic relations among queries. Here we propose three main contributions: a new heuristic to detect and remove noisy relations among queries mostly caused by multi-topical URLs; an efficient hierarchical clustering method for weighted graphs to extract semantic relations from query graphs; and, given a hierarchical clustering, a method to infer a query folksonomy and semantic relations among queries. We use a sample of a query log of the Yahoo! search engine to evaluate our approach and we compare our results with a query classification obtained by mapping queries over the Open Directory Project (ODP) categories.

2 Related Work

Most of the work on query similarity is related to query expansion or query clustering, common tasks in many applications such as query recommendation systems. Wen *et al* [15] proposed to cluster similar queries using four notions of query distance: (1) based on keywords or phrases of the query; (2) based on string matching of keywords; (3) based on common clicked URLs; and (4) based on the distance of the clicked documents in some predefined hierarchy. As the average number of words in queries is small and the number of clicks in the answer pages is also small [1], notions (1) and (2) generate distance matrices that are very sparse. For notion (4) we need a concept taxonomy and the clicked documents must be classified into that taxonomy as well, something that usually requires direct human intervention and that cannot be done in a large scale. Although notion (3) can generate also sparse distance matrices, the sparsity can be greatly reduced by using large query logs. Previous works have used notion (3) [4], or even variants combining (1) and (3) [16].

Baeza-Yates *et al.* [2] used the content of clicked Web pages to define a term-weight vector model for a query. In their work each term in a clicked URL is weighted according to the number of occurrences of the query and the number of clicks of the documents in which the term appears. Then, the similarity of two queries is given by the cosine similarity of their vector representations. This notion of query similarity is based on common clicked URLs as (3) and has several advantages. It is simple and easy to compute and makes it possible to relate queries that happen to be worded differently but stem from the same topic. More recently, Shen *et al.* [13] also used the notion (3) to cluster similar queries and build a query taxonomy. They also consider the terms in the clicked documents instead of the terms in the queries. In this paper we represent queries in a high dimensional space and we use also the cosine similarity, but each dimension corresponds to a URL. This notion uses common clicked URLs and it was introduced by Baeza-Yates and Tiberi [3].

Chuang *et al.* [5,6,7] also used query logs to build a query taxonomy to also cluster answers. However they do not use any user feedback, like user clicks. Moreover, this is not the same as building a taxonomy of the queries, which is what we call a query taxonomy or folksonomy.

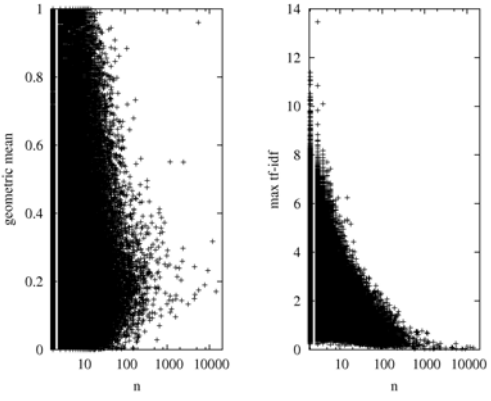


Fig. 1. On left, URL weight contribution geometric mean versus URL coverage size n. On right, given the set of terms associated to the queries covered by each URL, URL maximum tf-idf score versus URL coverage size n.

3 Click Induced Graph

Given a query q, the cover $\mu(q)$ of q is the set of URLs clicked by q. The *click induced graph* $G = (V, E)$ is an undirected graph with queries as nodes, $V = \mathcal{Q}$, and where $(q_1, q_2) \in E$ whenever q_1 and q_2 share at least one common clicked URL, $\mu(q_1) \cap \mu(q_2) \neq \emptyset$. Edges are weighted according to the cosine similarity σ of the queries they connect, with each query q being represented in a high dimensional space. Each dimension corresponds to a unique URL u and the weight is the frequency ratio with which the URL u was clicked for the query q.

In this paper we considered a query log piece from the Yahoo! search engine. The data was collected in April 2005 and contains 2,822,337 queries with at least one clicked URL and 4,927,980 different URLs. From these, only 660,910 URLs were clicked for at least one query. On average, each query has 2.39 distinct clicks and each URL is clicked by 1.37 distinct queries. Click distributions, per query and per URL, follow a power law with exponents 3.50 and 2.59, respectively.

The main purpose of the click induced graph is to represent semantic relations between queries and to enable knowledge extraction. But, for the studied query log, we have that about 75% and 50% of edges have weights bellow 0.5 and 0.273, respectively. Thus, there are many connected queries which are not much similar. Such noisy relations are mostly due to URLs covering dubious topics, several topics or very general topics, *i.e.*, multi-topical URLs.

An approach to remove noise is to ignore contributions from multi-topical URLs. Baeza-Yates and Tiberi [3] suggested that such URLs are the ones that contribute more to edges with low weights. But, we observed that URLs which contribute more to low weighted edges also may contribute more to high weighted edges. In fact there is a positive correlation between the number of queries covered by a URL and its contribution to edge weights (see Fig. 1). We tackle this problem by considering as documents the terms among the queries covered by

each URL and by computing the tf-idf score for each term as usual. We observed that multi-topical URLs have a low average tf-idf score, even when we select only high related queries for which those URLs were clicked. Hence, we propose to compute the maximum tf-idf among the bag of terms associated to each URL and select the URLs with lowest score as multi-topical candidates. In Fig. 1 we have the maximum tf-idf score against URL coverage size for the query log analyzed. In what follows, we ignore 0.05% of the URLs with lowest score, filtering the click induced graph in a conservative way. Note that many of the selected URLs have a large coverage and, maybe unexpectedly, they are not spam URLs.

The resulting click induced graph has 23,177,430 edges, about 6.44% of the size of the full click induced graph. Since we continue having low weighted edges, we remove 10% of the edges with lowest score, which have weights lower than 0.043. The filtered click induced graph has 20,974,257 edges and 1,648,649 connected components. The giant component and the second largest component have 861,903 and 64 vertices, respectively. There are now 1,474,249 singleton vertices. The degree distribution follows a power law with exponent 1.65. Our approach to remove noise and multi-topical URLs dramatically reduces the size of the click induced graph, keeping its structure almost unchanged.

4 Graph Clustering and Induced Query Folksonomy

One of the hardest problems in graph mining is finding graph community structure or graph clustering. Usually, clusters are groups of vertices such that the number of edges within them is higher than the number of edges among different groups. This problem has recently attracted a large interest [9]. In this paper we follow a two stage approach. We find a set of seed sets and, then, we apply a local optimization method. This approach is related to methods based on global partition and local expansion [12,14], but avoiding traditional global partitioning. Since we are interested in forming clusters of similar queries, our approach consists of joining similar queries, *i.e.*, we define cores based on vertex structure similarity. Let $G = (V, E)$ be a graph and $\sigma : E \longrightarrow \mathbb{R}_0^+$ the edge weight function. Given two connected vertices $(v_1, v_2) \in E$, their structural similarity η takes values in $[0, 1]$ and is given by

$$\frac{\sum_{w \in N_{12}} \sigma(v_1, w) + \sigma(v_2, w)}{|N_{12}|} \frac{2\,\sigma(v_1, v_2) + \sum_{w \in N_{12}} \sigma(v_1, w)\sigma(v_2, w)}{\sqrt{1 + \sum_{w \in N_1} \sigma(v_1, w)^2}\sqrt{1 + \sum_{w \in N_2} \sigma(v_2, w)^2}},$$

where N_1 and N_2 are the sets of neighbors of v_1 and v_2, respectively, and $N_{12} = N_1 \cap N_2$. The first term, the weight mean among common neighbors, was introduced because the second term, a cosine similarity based score, takes value 1.0 whenever the vertices v_1 and v_2 share all neighbors, even if they are connected through edges with low weights. Given $\varepsilon > 0$, $C \subseteq V$ is a *core* if C is a connected component composed only of edges with weights higher than ε. We can enumerate the set of cores in a graph for different values of $\varepsilon > 0$ and, by considering the edges in decreasing order of η, we obtain a hierarchy of cores.

Then, we apply the local partition method proposed by Chung [8] to each core, which expands it minimizing the conductance score Φ.

The induced query taxonomy is obtained by associating the most relevant terms to each node in the hierarchical clustering tree. We compute the set of terms associated to each node by grouping together all queries in the underlying subtree and by inspecting the queries. Those sets of terms become our documents and we infer the most relevant terms for each node by computing the tf-idf score for each term. Such terms become the tags for that node. Since click induced graphs are scale-free and have a giant component, we may want to define a threshold on the tf-idf score. Internal nodes corresponding to the giant component, or even to part of it, have usually bad quality tags which do not bring relevant semantic information.

5 Experimental Evaluation

For lack of space, we summarize briefly our results for the click induced graph described above. Full results are available in a technical report [11]. By considering different values for ε, we saw that the method effectively clusters the graph. For instance, with $\varepsilon = 0.4$, the biggest cluster is much smaller, about 1.1% of the original giant component. We note also that values for average conductance Φ are less than 0.1. Nevertheless, we obtain many small clusters for any cut of the hierarchical clustering tree, corresponding to loosely connected clusters that could appear connected if we consider larger query logs. Many are composed of highly specific queries or navigational queries, for which the search engine may return a low number of results and where the user clearly knows what he wants.

As mentioned before, the local optimization may lead to overlaps among clusters, providing interesting information with respect to query ambiguity, context, topic and polysemy. Terms like "windows" and "wine" are examples of polysemic terms that appear within overlaps of rather different clusters. Thus, an approach to identify term polysemy is to compare the bag of terms among overlapping clusters and, if a query is in two clusters but they share few terms, then the query shall be polysemic. Similarly, by analyzing similar words in the same cluster we can detect misspellings.

Given the hierarchical clustering described above, we build the induced query folksonomy for which tf-idf scores become meaningful only for $\varepsilon > 0.3$, *i.e.*, when the giant component vanishes. For $\varepsilon \leq 0.3$, the tf-idf score for the giant component takes values between 0.05 and 0.07, and the most relevant term is "free". Evaluating the query classification is difficult since it is very different from traditional directories. Here we try to compare it with the ODP in order to understand how different are these two ways of expressing knowledge. We mapped all queries over the ODP categories, obtaining several category paths for each query. Then we compare the ODP paths with the induced folksonomy. Since folksonomy labels are not comparable to the categories in the ODP mapping, they are not topic based, we evaluate the clusters by comparing the common ODP path prefix among the queries. Given two queries q_1 and q_2, we select

the two most similar ODP category paths p_1 and p_2, i.e., the ones which share the longest common prefix $\pi(p_1, p_2)$. Then we compute the score $\sigma_{\text{odp}}(p_1, p_2) = |\pi(p_1, p_2)| / \max\{|p_1|, |p_2|\}$, where $|\cdot|$ denotes the path length. The ODP score for a given cluster is the average of the score σ_{odp} for all pairs of queries in that cluster. For all snapshots of the hierarchical clustering for different values of ε, more than 50% of the clusters have an ODP score higher than 0.5 and 30% to 39% of them have an ODP score equal to 1.0. For $\varepsilon > 0.3$, after we cluster the giant component, the ODP score increases with the hierarchical clustering depth, revealing that clusters at higher depths have better quality. This is also supported by the tf-idf scores.

References

1. Baeza-Yates, R.: Applications of web query mining. In: Losada, D.E., Fernández-Luna, J.M. (eds.) ECIR 2005. LNCS, vol. 3408, pp. 7–22. Springer, Heidelberg (2005)
2. Baeza-Yates, R., Hurtado, C., Mendoza, M.: Query clustering for boosting web page ranking. In: Favela, J., Menasalvas, E., Chávez, E. (eds.) AWIC 2004. LNCS (LNAI), vol. 3034, pp. 164–175. Springer, Heidelberg (2004)
3. Baeza-Yates, R.A., Tiberi, A.: Extracting semantic relations from query logs. In: SIGKDD, pp. 76–85. ACM, New York (2007)
4. Beeferman, D., Berger, A.: Agglomerative clustering of a search engine query log. In: SIGKDD. ACM, New York (1999)
5. Chuang, S.L., Chien, L.F.: Towards automatic generation of query taxonomy: A hierarchical query clustering approach. In: IEEE International Conference on Data Mining. IEEE, Los Alamitos (2002)
6. Chuang, S.L., Chien., L.F.: Automatic query taxonomy generation for information retrieval applications. Online Information Review 27(5) (2003)
7. Chuang, S.L., Chien, L.F.: Enriching web taxonomies through subject categorization of query terms from search engine logs. Decision Support System 30(1) (2003)
8. Chung, F.: The heat kernel as the pagerank of a graph. Proceedings of the National Academy of Sciences 104(50), 19735 (2007)
9. Fortunato, S.: Community detection in graphs. Physics Reports 486, 75–174 (2010)
10. Francisco, A.P., Baeza-Yates, R., Oliveira, A.L.: Clique analysis of query log graphs. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 188–199. Springer, Heidelberg (2008)
11. Francisco, A.P., Baeza-Yates, R., Oliveira, A.L.: Mining query logs induced graphs. Tech. Rep. 36/2010, INESC-ID (2010)
12. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-define clusters. arXiv:0810.1355 (2008)
13. Shen, D., Qin, M., Chen, W., Yang, Q., Chen, Z.: Mining Web Query Hierarchies from Clickthrough Data. In: AAAI 2007, pp. 341–346. AAAI Press, Menlo Park (2007)
14. Wei, F., Qian, W., Wang, C., Zhou, A.: Detecting Overlapping Community Structures in Networks. World Wide Web 12(2), 235–261 (2009)
15. Wen, J., Mie, J., Zhang, H.: Clustering user queries of a search engine. In: Proc. of the 10th International World Wide Web Conference. W3C (2001)
16. Zaiane, O.R., Strilets, A.: Finding similar queries to satisfy searches based on query traces. In: Efficient Web-Based Information Systems (EWIS) (2002)

Finite Automata Based Algorithms for the Generalized Constrained Longest Common Subsequence Problems*

Effat Farhana, Jannatul Ferdous, Tanaeem Moosa, and M. Sohel Rahman

AℓEDA Group
Department of Computer Science and Engineering (CSE)
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh
{eva_cse,choity186}@yahoo.com,
{tanaeem,msrahman}@cse.buet.ac.bd

Abstract. The *Longest Common Subsequence* (LCS) problem is a classic and well-studied problem in computer science. Given strings S_1, S_2 and P, the generalized constrained longest common subsequence problem (GC-LCS) for S_1 and S_2 with respect to P is to find a longest common subsequence of S_1 and S_2, which contains (excludes) P as a subsequence (substring). We present finite automata based algorithms with time complexity $O(r(n+m)+(n+m)\log(n+m))$ for a fixed sized alphabet, where r, n and m are the lengths of P, S_1 and S_2 respectively.

1 Introduction

The *Longest Common Subsequence* (LCS) problem is a classic and well-studied problem in Computer Science with extensive applications in diverse areas ranging from spelling error corrections to molecular biology. Given a string, a subsequence of that string is any string such that its symbols appear (not necessarily contiguously) somewhere in the string in the same order. A substring of a string is a subsequence of successive characters within the string. Given two strings X and Y, a common subsequence of X and Y is a subsequence that appears in both the strings. A longest common subsequence is a maximum length common subsequence of the given strings. Given two strings X and Y, the goal of the LCS problem is to compute a longest common subsequence of X and Y.

The classic dynamic programming solution to LCS problem was invented by Wagner and Fischer [16]. The LCS problem and variants thereof have been extensively studied for decades (e.g., [2,3,14,5]). In this paper, we study some relatively new variants of the LCS problem, namely, the Generalized Constrained LCS problems (GC-LCS) as defined below.

* This research work constitutes part of the undergraduate thesis work of the first and second authors under the supervision of the last author. Authors' names are in alphabetic order.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 243–249, 2010.

Problem 1. (**STR-IC-LCS Problem**): Given two strings S_1 and S_2 and a constraint pattern P of lengths n, m and r respectively, the STR-IC-LCS problem is to find an LCS of S_1 and S_2 including P as a substring.

Problem 2. (**SEQ-IC-LCS Problem**): ([4,8,12,15]) Given two strings S_1 and S_2 and a constraint pattern P of lengths n, m and r respectively, the SEQ-IC-LCS problem is to find an LCS of S_1 and S_2 including P as a subsequence.

Problem 3. (**STR-EC-LCS Problem**): Given two strings S_1 and S_2 and a constraint pattern P of lengths n, m and r respectively, the STR-EC-LCS problem is to find an LCS of S_1 and S_2 excluding P as a substring.

Problem 4. (**SEQ-EC-LCS Problem**): Given two strings S_1 and S_2 and a constraint pattern P of lengths n, m and r respectively, the SEQ-EC-LCS problem is to find an LCS of S_1 and S_2 excluding P as a subsequence.

SEQ-IC-LCS was first introduced by Tsai in [15], where an algorithm was presented solving the problem in $O(rn^2m^2)$ time, where $|X| = n, |Y| = m$ and $|P| = r$. Later, Chin [8] and independently, Arslan and Egecioglu [4] presented improved algorithm with $O(mnr)$ time and space complexity. Recently, Iliopouos and Rahman [12] proposed an $O(r\mathcal{R}\log\log n + n)$-time algorithm for this problem, where $\mathcal{R}$ is the total number of ordered pairs of positions at which X and Y match[1]. The other three variants of GC-LCS problems were introduced and studied by Chen and Chao very recently in [6]. They solved all four variants in $O(mnr)$ time. This problem finds its motivation from bioinformatics: in the computation of the homology of two biological sequences it is important to take into account a common specific or putative structure. To compare two sequences, from the above point of view, both inclusive and exclusive constraints seems to be interesting.

Very recently, Gotthilf et al. [9][2] considered the "Restricted LCS" problem which is basically a variant of the SEQ-EC-LCS Problem. In particular, Gotthilf et al. [9] solved the SEQ-EC-LCS Problem for arbitrary number of input and constraint strings in $O(n^{L+\ell})$ time where L = number of input strings, ℓ = number of constrained Patterns. Here for the sake of convenience, it is assumed that all the input strings are of same length n. So, for two input strings and one constraint pattern, the algorihm runs in $O(n^3)$ time.

In this paper, we devise finite automata based efficient algorithms for GC-LCS problems that run in $O(r(n+m) + (n+m)\log(n+m))$ time for a fixed alphabet. Since the GC-LCS problems find their motivation from computational biology and since biological sequences mostly have constant sized alphabets (e.g., DNA/RNA sequences have alphabet size 4 and protein sequences 20), the assumption of a fixed alphabet in this context is perfectly valid. Note that the very recent result of Gotthilf et al. [9] doesn't assume a fixed alphabet. The rest of the paper is organized as follows. In Section 2, we present some preliminary definitions and in Section 3, we present our main results. The time complexity

[1] In [12], without the loss of generality, it is assumed that $|X| = |Y| = n$.

[2] In fact, the result of Gotthilf et al. [9] is presented at the same conference we present our result.

analysis of our algorithms is presented in Section 4. Finally, we conclude briefly in Section 5.

2 Preliminaries

To formally describe our algorithms following definitions are necessary.

Definition 1. DFA. *A deterministic finite automata is represented by 5-tuple notation* $A = (Q, \sum, \delta, q_0, F)$*, where A is the name of the DFA, Q is its set of states,* $\sum$ *its input symbol,* δ *its transition function,* q_0 *its start state and F its set of accepting states.*

Definition 2. DASG. *A DASG for a string A is a DFA that accepts the language of all possible* 2^n *subsequences of A. The DFA is* partial*, that is, each state need not have a transition defined for every symbol.*

Definition 3. DASG for multiple texts. *Let S be a set of texts* $T_1, T_2, \ldots, T_k$*. We say that P is a subsequence of S if and only if there exists* $i \in [1, k]$ *such that P is a subsequence of* T_i*. More formally, DSAG of S is a DFA A which accepts the language* $\mathcal{L}(A)$=$\{w$*:* $i \in [1, k]$*, w is a subsequence of* $T_i\}$*.*

Definition 4. Substring Automata. *Given a string P, substring automata is a DFA A, which accepts the language* $\mathcal{L}(A)$=$\{w$*: w has P as substring*$\}$*.*

Definition 5. Supersequence Automata. *A supersequence automata is a finite automata which accepts the set of all supersequences of a given string.*

3 Finite Automata Based Algorithms

In this section, we present our algorithms to solve the GC-LCS problems. Since, the algorithms we present are closely related to each other, we first present our algorithm for the STR-IC-LCS Problem and then we modify it to solve the other three variants.

3.1 STR-IC-LCS Problem

We first present the algorithm to find an LCS of S_1 and S_2 containing the constrained pattern P as a substring. The main steps of the algorithm are as follows:
Step 1: Construct the *Common Subsequence Automata* M_1 for S_1 and S_2.
Step 2: Construct the *Substring Automata* M_2 for P.
Step 3: Construct the *Intersection Automata* M_3 of M_1 and M_2.
Step 4: Construct the *Maximum length Automata* M from M_3.
Detailed description of each of the steps are given below:

Step 1: Construction of common subsequence automata
On input (S_1, S_2) we can build a DASG $M'_{S_1S_2}$ using $O((n+m)\log(n+m))$ time. The DASG $M'_{S_1S_2}$ has $O(n+m)$ states and $|\sum|(n+m)$ transitions

(for DNA/RNA sequence, fixed alphabet, $|\sum| = 4$), and accepts the language $\mathcal{L}(M'_{S1_S2}) = \{w$: w is a subsequence of S_1 or $S_2\}$. Additionally, each transition in $M'_{S_1S_2}$ is labeled if the transition corresponds to part of a subsequence common to both S_1 and S_2. The construction is given by Baeza-Yates [1], using suffix-trees and table accesses. Then we prune this DASG to create DASG M_1which accepts the language $\mathcal{L}(M_1) = \{w$: w is a common subsequence of S_1 and $S_2\}$. We need to do the following steps:

- We build the DASG for S_1 and S_2, appending to each transition the number of strings that share the matched point. For any state s, we call state s' reachable from s if there is a transition from s to s' that is labeled as being common to both S_1 and S_2.
- We use Depth first search to traverse the DFA $M'_{S1_S2} =(S', \sum, \delta', s_0, F')$, starting at s_0. For all $(s, \sigma, s') \in \delta'$, if state s' is reachable from s, then add (s, σ, s') to δ. Also, add s' to F.
- We return M_1=$(S', \Sigma, \delta, s_0,$ F).

Step 2: Construction of substring automata for P
The deterministic substring automata is similar to pattern matching automata described in [7]. The properties and performance can be expressed by the following Lemma.

Lemma 1 ([7]). *Given a constrained pattern P of length r, we can construct a DFA M_2 accepting language $\mathcal{L}(M_2)$=$\{w$: w has P as substring$\}$ in $O(r)$ time. M_2 has $(r + 1)$ states and $|\sum|(r + 1)$ transitions.*

Step 3: Automata for intersection of language
$\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ are both regular languages [13] and regular languages are closed under Boolean operation [10]. Therefore, $\mathcal{L}(M_1)\bigcap \mathcal{L}(M_2)$ is the language that contains all strings that are both in M_1 and M_2, that is, it accepts all common subsequence of S_1 and S_2 containing P as substring. As DASG is a partial DFA, we have used a variant of standard intersection algorithm that creates only accessible states.

Lemma 2. *Given DFA M_1 and M_2 having $(n+m)$ and r states, DFA M can be constructed accepting language $\mathcal{L}(M) = \mathcal{L}(M_1)\bigcap\mathcal{L}(M_2)$ in $O((n + m)r)$ time. M has at most $(n+m)r$ states and at most $|\sum|(n+m)r$ transitions. Moreover, if M_1 (or M_2) is acyclic, then M is also acyclic.*

Step 4: Maximum Length Automata
We use a maximum length automata (maxlen automata) to find maximum length strings from the intersecting automata. The algorithm can be found in [11]. In brief, the automata is a modification of longest path algorithm for DAGs (Directed Acyclic Graph) that works in $O(E)$ time; where E is the number of edges in the input DAG. If the original automata is $A' = (Q', \sum, \delta', q'_0, F')$, then the resulting maxlen automata is $A = (Q, \sum, \delta, q_0, F)$, where $Q \subseteq Q', \delta \subseteq \delta', F \subseteq F'$.

Lemma 3 ([11]). *Given an acyclic DFA M' with n transitions, DFA M can be built accepting language $\mathcal{L}(M)$=Maxlen(M') in $O(n)$ time. M has at most as many states and at most as many transitions in M'.*

3.2 STR-EC-LCS Problem

Steps are similar as before except for substring automata in **Step 2**. Now we need an automata accepting the complement of language $\mathcal{L}(M_2)$. This can be done by making the accepting states of M_2 into nonaccepting states and vice versa. [10].

3.3 SEQ-IC-LCS Problem

The procedure is same as STR-IC-LCS. But here in **Step 2** instead of substring automata we use supersequence automata, which accepts all strings including constrained pattern P as subsequence. The DFA for accepting language $\mathcal{L}(M) = Super(X)$, for string X, $|X| = n$ can be formally described as follows: $M = (Q, \sum, \delta, q_0, \{F\})$, where $Q = \{q_0, q_1, \ldots, q_n\}, F = \{q_n\}, \delta(q_{i-1}, x_i) = q_i, \delta(q_{i-1}, s) = q_{i-1}$, for $s \in \{\sum \backslash x_i\}$, where $1 \leq i \leq n$ and $\delta(q_n, s) = q_n$. The basic properties and time complexity of this automata can be summarized by the following lemma.

Lemma 4. *Given a string X of length n, a DFA M can be constructed in $O(n)$ time accepting language $\mathcal{L}(\boldsymbol{M})$=Super(x). M has $(n+1)$ states and $|\sum|(n+1)$ transitions.*

3.4 SEQ-EC-LCS Problem

The procedure is same as STR-EC-LCS. But here in **Step 2** instead of substring automata's complement we use supersequence automata's complement. So, after intersecting with common subsequence automata we get an automata which accepts all common subsequences that do not contain P as subsequence. Finally, MAXLEN algorithm is applied on this automata. The resulting automata M accepts language $\mathcal{L}(M)$= maximum length common subsequence of S_1 and S_2 that does not contain P as subsequence.

4 Complexity

In this section, we present the following lemma which proves the correctness of our algorithms and their time complexity.

Lemma 5. *Given strings S_1, S_2 and P of length n, m and r, we can find GC-LCS(S_1, S_2, P) in $O((n+m)\log(n+m) + (m+n)r)$ time.*

Proof. **Correctness:** Common subsequence automata is acyclic and accepts all common subsequences of S_1 and S_2. Substring automata for P is a cyclic automata that accepts all strings which contain P as a substring. In case of STR-IC-LCS, after intersecting common subsequence automata with substring automata, we get an acyclic automata which accepts all common subsequences of S_1 and S_2 that contain P as a substring. Finally, by constructing maxlen automata from this we can get longest common subsequence with P included

as a substring. In case of STR-EC-LCS, we take intersection between common subsequence automata and complement of Substring automata. Complement of Substring automata accepts those strings that do not contain P as a substring. Resulting automata accepts all common subsequences of S_1 and S_2 that do not contain P as a substring. Again, by constructing maxlen automata from this we get longest common subsequence with P not included as substring. Similarly, we can prove correctness of our algorithm for SEQ-IC-LCS and SEQ-EC-LCS.

Time Complexity: Constructing common subsequence automata takes $O((n+m)\log(n+m))$ time. This automata has $O(m+n)$ states. Constructing subsequence automata and substring automata takes $O(r)$ time. This automata has $O(r)$ states. Their intersection takes $O((m+n)r)$ time. Construction of Maxlen automata takes further $O((m+n)r)$ time. So, overall time complexity is $O((n+m)\log(n+m)+(m+n)r)$, which significantly improves existing $O(mnr)$ solution.

5 Conclusion

In this paper, we have studied different variations of the constrained longest common subsequence problem and have given finite automata based solutions that run in $O((n+m)\log(n+m)+(m+n)r)$ time for fixed sized alphabet. However, we can also extend our algorithm for arbitrary number of input strings and constraint patterns as considered in [9]. In that case, the overall time complexity of our extended algorithm will be $O((n-L+2)\times\prod_{i=1}^{k} r_i + Ln\log n)$. It would be interesting to implement these algorithms to compare the performances in practice.

Acknowledgement

This research work was carried out as part of the undergraduate thesis work of Farhana and Ferdous under the supervision of Rahman in the Department of CSE, BUET, Dhaka-1000, Bangladesh.

References

1. Baeza-Yates, R.A.: Searching subsequences. Theoritical Computer Science 78, 363–376 (1991)
2. Aho, A.V., Hirchberg, D., Ullman, J.D.: Bounds on the complexity of longest common subsequence problem. Journal of the ACM 23, 1–12 (1976)
3. Apostolico, A., Guerral, C.: The longest common subsequence problem revisited. Algorithmica 2, 315–336 (1987)
4. Arslan, A.N., Eğecioğlu, Ö: Algorithms for the constrained longest common subsequence problems. International Journal of Foundations Computer Science 16(6), 1099–1109 (2005)

5. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: String Processing and Information Retrieval (SPIRE), pp. 39–48 (2000)
6. Chen, Y.-C., Chao, K.-M.: On the generaized constrained longest common subsequence problem. Journal of Combinatorial Optimization (2009)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT press, McGraw-Hill, New York (2001)
8. Santis, A.D., Chin, F.Y.L., Ferrara, A.L.: A simple algorithm for the constrained longest common sequence problem. Information Processing Letters 90, 175–179 (2003)
9. Gotthilf, Z., Hermelin, D., Landau, G.M., Lewinstein, M.: Restricted lcs. In: Accepted in String Processing and Information Retrieval, SPIRE (2010)
10. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory,Languages, and Computation, 2nd edn. Pearson Education, London (2001)
11. Iliopoulos, C.S., Rahman, M.S., Voráček, M., Vagner, L.: Computing constrained longest common subsequence for degenerate strings using finite automata. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 309–311. Springer, Heidelberg (2007)
12. Iliopoulos, C.S., Rahman, M.S.: New efficient algorithms for the lcs and constrained lcs problem. Information Processing Letters 106, 13–18 (2008)
13. Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)
14. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. Journal of Computer and System Sciences 20(1), 18–31 (1980)
15. Tsai, Y.-T.: The constrained longest common subsequence problem. Information Processing Letters 88(4), 173–176 (2003)
16. Wagner, R.A., Fischer, M.J.: The string to string correction problem. Journal of the ACM 21(1), 168–173 (1974)

Restricted LCS

Zvi Gotthilf[1], Danny Hermelin[2,*], Gad M. Landau[3,**], and Moshe Lewenstein[1,***]

[1] Bar-Ilan University, Israel
gotthiz@cs.biu.ac.il,moshe@cs.biu.ac.il
[2] University of Haifa, Israel
danny@cri.haifa.ac.il
[3] University of Haifa, Israel, and NYU-Poly, USA
landau@cs.haifa.ac.il

Abstract. The *Longest Common Subsequence* (LCS) of two or more strings is a fundamental well-studied problem which has a wide range of applications throughout computational sciences. When the common subsequence must contain one or more *constraint strings* as subsequences, the problem becomes the *Constrained LCS* (CLCS) problem. In this paper we consider the *Restricted LCS* (RLCS) problem, where our goal is finding a longest common subsequence between two or more strings that does not contain a given set of *restriction strings* as subsequences. First we show that in case of two input strings and an arbitrary number of restriction strings the RLCS problem is NP-hard. Afterwards, we present a dynamic programming solution for RLCS and we show that this algorithm implies that RLCS is in FPT when parameterized by the total length of the restriction strings. In the last part of this paper we present two approximation algorithms for the hard variants of the problem.

1 Introduction

Given a set of strings $A_1, \ldots, A_m$, a *common subsequence* of these strings is a string S which appears as a subsequence in each of $A_1, \ldots, A_m$, *i.e.* it can be obtained from each A_i by deleting (possible none) letters. The Longest Common Subsequence (LCS) problem is the problem of determining the (length of the) longest common subsequence between a given set of strings. LCS is a fundamental problem in computer science, and has therefore been thoroughly studied (see *e.g.* [1,6,12,13]). The problem had also been investigated on more general structures such as trees and matrices [2], run-length encoded strings [4], and more.

* Supported by the Adams Fellowship of the Israel Academy of Sciences and Humanities.
** Partly supported by the National Science Foundation Award 0904246, Israel Science Foundation grants 35/05 and 347/09, the Israel-Korea Scientific Research Cooperation, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.
*** Partially supported by the Israel Science Foundation grant 1484/08.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 250–257, 2010.

The *Constrained Longest Common Subsequence* (CLCS) is an extension of the LCS problem, where now we are given a set of constraint strings $B_1, \ldots, B_\ell$ in addition to the set of comparison strings, and the goal is to find the longest common subsequence of the comparison strings that contains each of $B_1, \ldots, B_\ell$ as a subsequence. Quite a few results on the CLCS problem were presented recently [5,8,9,10,11,14,15,18].

In this paper, we consider the "opposite" extension of the LCS problem, namely the Restricted LCS (RLCS) problem:

Definition 1 (Restricted LCS). *Given m input strings $A_1, \ldots, A_m$ and ℓ restriction strings $B_1, \ldots, B_\ell$, the Restricted LCS (RLCS) problem is the problem of computing the (length of the) longest common subsequence of $A_1, \ldots, A_m$ that does not contains each of $B_1, \ldots, B_\ell$ as a subsequence.*

In molecular biology, when biologists find new DNA sequences, they typically would like to discover similarities between this sequence and other known sequences. The classical LCS will provide a good measurement for such similarities. However, in real life, biologists might already be familiar with a set of sequences that are known to be "irrelevant" or "non interesting". Such sets of irrelevant sequences can be based on previous research, in such cases the biologists would like to find the maximum relevant similarity between this new sequence and other known sequences. In other words, biologists would prefer to find major similarities while also reducing unnecessary parts of it.

Similar application can be also found in the data-mining area. While searching for similarities, in most cases people prefer to find a qualitative solution which is compatible to some previous known restrictions or assumptions, than just finding the longest similarity.

Andrejkova [3] also refers to a restricted variant of the classical LCS problem, however, the problem discussed in this paper is completely different. As far as we know, this extension of the LCS problem has not yet been considered. We believe that RLCS might be better suited than its counterpart CLCS for some scenarios, *e.g.* the above mentioned biological or data-mining applications.

1.1 Related Work

One of the goals of this paper is to compare the RLCS problem with CLCS. We therefore briefly describe the state of the art of CLCS. The problem was first introduced by Tsai [18] where he presented a dynamic programming algorithm for the simplest case of two comparison strings and a single constraint string. Improved dynamic programming algorithms were proposed in [5,9]. In [11], fast approximation algorithms were designed for this basic CLCS variant. In [10], it is proven that it is NP-hard to approximate CLCS within any factor, even in case of two input strings and an arbitrary number of constraint strings. Moreover, a factor $\frac{1}{\sqrt{n_{min}|\Sigma|}}$ approximation algorithm is presented for the case of many input strings and a single constraint, where n_{min} denotes the length of the shortest comparison string, and $|\Sigma|$ denotes the number of different letters occurring in both the comparison and constraint strings.

1.2 Our Contribution

We focus on several different settings of RLCS. Section 2 is devoted to fixing some notation and simplifications. Afterwards, in section 3, we show that the problem is NP-hard even in the case of two comparison strings and an arbitrary number of restrictions, each of length at most 2. Moreover, in Section 4, we present an $O(n^{m+\ell})$ dynamic programming solution for the RLCS problem, where m and ℓ respectively denote the number of comparison and restriction strings. We also show in this section that this algorithm implies that RLCS is in FPT when parameterized by the total length of the restriction strings. Finally, in Section 5, we present two simple approximation algorithms for the problem: The first having an approximation ratio of a $\frac{1}{|\Sigma|}$, and is suited for the most general variant of the RLCS problem, and the second having a ratio of $\frac{k_{min}-1}{n_{min}}$, and is relevant only for instances with a constant number of input strings. Here n_{min} and k_{min} are the lengths of the shortest input string and the shortest restriction, respectively, and $|\Sigma|$ is the number of different letters occurring in all strings of the instance.

2 Preliminaries

All strings considered in this paper are defined over some fixed alphabet Σ which can have arbitrary (including infinite) cardinality, for a string S, we use $|S|$ to denote the length of S. For $i \in \{1, \ldots, |S|\}$, we write $S[i]$ for the letter at the i'th position in S, and $S[[i]]$ for the i'th prefix of S, *i.e.* $S[[i]] = S[1] \cdots S[i]$.

We will use $A_1, \ldots, A_m$ and $B_1, \ldots, B_\ell$ to denote the set of *comparison* and *restriction* strings in our input of the RLCS problem. Typically, we will use the letter i to index the comparison strings, and the letter j to index the restriction strings. For $i \in \{1, \ldots, m\}$, we write n_i for $|A_i|$, and for $j \in \{1, \ldots, \ell\}$, we use k_j to denote $|B_j|$. Finally, n is used to denote the total length of the comparison strings, *i.e.* $n = \sum_{i=1}^{m} n_i$, and k to denote the total length of the restriction strings, *i.e.* $k = \sum_{j=1}^{\ell} k_j$.

We will make two assumptions that will not introduce any loss of generality, yet will help in simplifying matters somewhat. The first assumption is that all restriction strings have length at least 2, since if any single character appears as a restriction, we can simply delete all its occurrences from the comparison strings, and proceed without this restriction. The second assumption is that for all $i \in \{1, \ldots, m\}$ and all $j \in \{1, \ldots, \ell\}$, we have $k_j \leq n_i$, since otherwise we can remove the j'th restriction as it will never appear in a common subsequence of $A_1, \ldots, A_m$.

3 Hardness Result

It is known that in case of two input strings the classical LCS problem can be solved easily in polynomial time. Here we show that if we add an arbitrary number of restrictions, each of length 2, then the problem becomes NP-hard:

Theorem 1. *The RLCS problem in case of two comparison strings and an arbitrary number of restrictions, each of length 2, is NP-hard.*

Note that a valid solution to RLCS can always be easily found, as opposed to the CLCS problem where it is NP-hard to determine whether a given instance has any valid solutions. In this sense, proving hardness of approximation for RLCS is somewhat more challenging in comparison to CLCS. Nevertheless, to prove Theorem 1, we deploy a reduction which is similar to the one used for CLCS in [10].

The reduction we use is from the 6-OCC-MAX-2SAT problem which is defined as follows: Given a CNF formula ϕ with clauses of size 2, and where every variable appears in at most 6 clauses, the goal is to find an assignment of the variables that maximizes the number of satisfied clauses in ϕ. Berman and Karpinski [7] showed that 6-OCC-MAX-2SAT is APX-hard.

Given a 6-OCC-MAX-2SAT instance ϕ with variables $x_1, \ldots, x_{n_\phi}$ and clauses $c_1, \ldots, c_\ell$, we construct an RLCS instance $I_\phi = (A_1, A_2, B_1, \ldots, B_\ell)$ over the alphabet $\Sigma = \{c_1, \ldots, c_\ell\} \cup \{s\}$, where s is a special *padding character*. The string A_1 is constructed as follows: For each variable x_i, if $c_{i_1}, \ldots, c_{i_t}$ are the clauses satisfied by setting $x_i = 1$, and $c_{i'_1}, \ldots, c_{i'_{t'}}$ are the clauses satisfied by setting $x_i = 0$, we construct a pair of substrings

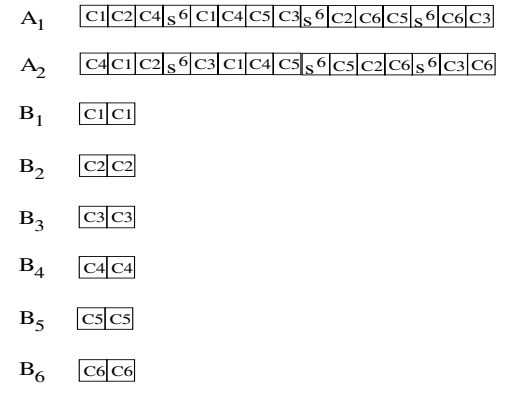
$$X_i = \text{“}c_{i_1}, \ldots, c_{i_t} c_{i'_1}, \ldots, c_{i'_{t'}}\text{”} \text{ and } X'_i = \text{“}c_{i'_1}, \ldots, c_{i'_{t'}} c_{i_1}, \ldots, c_{i_t}\text{”}.$$

The comparison strings A_1 and A_2 are then constructed as

$$A_1 = X_1 s^6 X_2 s^6 \cdots s^6 X_{n_\phi} \text{ and } A_2 = X'_1 s^6 X'_2 s^6 \cdots s^6 X'_{n_\phi},$$

where $s^6 = \text{‘}ssssss\text{’}$. Finally, we let $B_j = \text{“}c_j c_j\text{”}$ for all $j \in \{1, \ldots, \ell\}$.

Given the following 6-OCC-MAX-2SAT formula: $\phi = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\bar{x_2} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2) \wedge (x_2 \vee \bar{x_3}) \wedge (x_3 \vee x_4)$, we construct the following RLCS instance I_ϕ:



Note that the instance I_ϕ satisfies the requirements of the theorem and can be constructed in polynomial time. To complete the proof of Theorem 1, we have the following lemma:

Lemma 1. *w clauses can be satisfied in ϕ iff there is a solution to I_ϕ of length $6(n_\phi - 1) + w$.*

Proof. For simplicity, we assume that there are no clauses in ϕ that contain both x_i and $\bar{x_i}$.

($\Rightarrow$) Let $\varphi : \{x_1, \ldots, x_n\} \to \{0, 1\}$ be some assignment satisfying w clauses in ϕ, and for each i, $1 \leq i \leq n_\phi$, let C_i denote the clauses satisfied by $\varphi(x_i)$. We construct a valid solution to I_ϕ from left to right: Assuming we have already processed $\{C_1, \ldots, C_{i-1}\}$, we append to our solution all clauses in C_i which have not been appended previously, concatenated by s[6].

Note that since we only add clauses satisfied by $x_i = 0$ or $x_i = 1$, but never both, our solution is indeed a common subsequence of A_1 and A_2. Furthermore, notice that every clause has exactly one occurrence in our solution, therefore it does not contain any of $B_1, \ldots, B_\ell$ as a subsequence. Obviously, since w different clauses are satisfied and the separator has $6(n_\phi - 1)$ occurrences, the length of our solution is $6(n_\phi - 1) + w$.

($\Leftarrow$) Let S_ϕ be a solution to I_ϕ of length $6(n_\phi - 1) + w$. Since every X_i contains at most 6 letters, we can conclude that S_ϕ contains exactly $6(n_\phi - 1)$ occurrences of the padding-character s. Moreover, every clause has at most one occurrence in S_ϕ, by construction of the restriction strings $B_1, \ldots, B_\ell$. Now, since all padding characters are in S_ϕ, S_ϕ must contain exactly w different clauses, and for each of these clauses there is an i with the clause selected from X_i in A_1 and X'_i in A_2. By construction of X_i and X'_i, clauses that are satisfied by $x_i = 0$ cannot be selected with clauses satisfied by $x_i = 1$. It follows that there exists an assignment to $x_1, \ldots, x_{n_\phi}$ which satisfies w clauses of ϕ. □

4 Exact Algorithms

In this section we present an exact algorithm for the RLCS problem. We begin with the special case of two input strings and one restriction string, the variant we dub the *basic RLCS* problem. We then extend this algorithm to the general case of m input strings and ℓ restriction strings. We show that this generalization implies that RLCS is FPT when parameterized by the total length of the restriction strings.

For describing our dynamic-programming solution for basic RLCS, we define a dynamic-programming table DP, where the entry $DP[i_1, i_2 : j_1]$, for $(i_1, i_2) \in \{1 \ldots, n_1\} \times \{1, \ldots, n_2\}$ and $j_1 \in \{1, \ldots, k_1\}$, will store the length of the LCS between $A_1[[i_1]]$ and $A_2[[i_2]]$ restricted by $B_1[[j_1]]$. The entry $DP[n_1, n_2 : k_1]$ will store the length of the LCS between A_1 and A_2 restricted by B_1. The computation of $DP[i_1, i_2 : j_1]$ is given by the following recursion:

$$DP[i_1, i_2 : j_1] = \begin{cases} \max \begin{cases} DP[i_1 - 1, i_2 - 1 : j_1 - 1] + 1 \\ DP[i_1, i_2 - 1 : j_1] \\ DP[i_1 - 1, i_2 : j_1] \end{cases} & A_1[i_1] = A_2[i_2] = B_1[j_1], \\ DP[i_1 - 1, i_2 - 1 : j_1] + 1 & A_1[i_1] = A_2[i_2] \neq B_1[j_1], \\ \max \begin{cases} DP[i_1 - 1, i_2 : j_1] \\ DP[i_1, i_2 - 1 : j_1] \end{cases} & \text{otherwise.} \end{cases}$$

It is not difficult to see that the above recursion is correct. In particular, if $A_1[i_1]$ and $A_2[i_2]$ are not both equal to $B_1[j_1]$, then the recursion for $DP[i_1, i_2 : j_1]$ follows the standard recursion for pairwise LCS, since there is no danger in computing a solution which contains $B_1[1] \cdots B_1[j_1]$. On the other hand, if $A_1[i_1] = A_2[i_2] = B_1[j_1]$, then a common subsequence of $A_1[[i_1]]$ and $A_2[[i_2]]$ ending with the letter $A_1[i_1]$ cannot contain $B_1[[j_1 - 1]]$ as a subsequence, and so its length must be equal to $DP[i_1 - 1, i_2 - 1 : j_1 - 1] + 1$.

We next extend the above recursion for the case of m comparison strings $A_1, \ldots, A_m$ and ℓ restriction strings $B_1, \ldots, B_\ell$. Again, we have a dynamic programming table DP, indexed by tuples $(i_1, \ldots, i_m) \in \{1 \ldots, n_1\} \times \cdots \times \{1, \ldots, n_m\}$ and $(j_1, \ldots, j_\ell) \in \{1, \ldots, k_1\} \times \cdots \times \{1, \ldots, k_\ell\}$, where $DP[i_1 \ldots, i_m : j_1, \ldots, j_\ell]$ is equal to the length of the LCS between $A_1[[i_1]], \ldots, A_m[[i_m]]$ restricted by $B_1[[j_1]], \ldots, B_\ell[[j_\ell]]$.

If it is not the case that $A_1[i_1] = \cdots = A_m[i_k]$, then the recursion for $DP[i_1 \ldots, i_m : j_1, \ldots, j_\ell]$ follows the standard recursion for LCS between m strings (*i.e.* the restriction strings can be ignored). If $A_1[i_1], \ldots, A_m[i_k]$ all equal some letter σ, then we compute $DP[i_1, \ldots, i_m : j_1, \ldots, j_\ell]$ by:

$$DP[i_1, \ldots, i_m : j_1, \ldots, j_\ell] = \max \begin{cases} \max \begin{cases} DP[i_1 - 1, \ldots, i_m : j_1, \ldots, j_\ell] \\ . \\ . \\ . \\ DP[i_1, \ldots, i_m - 1 : j_1, \ldots, j_\ell] \end{cases} \\ \\ DP[i_1 - 1, \ldots, i_m - 1 : j_1^*, \ldots, j_\ell^*] \end{cases}$$

Where for all $x \in \{1, \ldots, \ell\}$, we set $j_x^* = j_x - 1$ if $B_x[j_x] = \sigma$, and otherwise $j_x^* = j_x$.

Correctness of this recursion follows from the same arguments used for the recursion for basic RLCS. Thus, since the dynamic programming table DP has $O(n^{m+\ell})$ entries, with each entry computable in constant time, we get:

Lemma 2. *RLCS can be solved in $O(n^{m+\ell})$ time.*

Let k denote the total length of the restriction strings, *i.e.* $k = \sum_{j=1}^{\ell} k_j$. Observe that the number of entries in DP can also be bounded by $O(2^k n^m)$, since the number of prefixes of the restriction strings cannot exceed 2^k. Thus, we have:

Lemma 3. *RLCS is in FPT when parameterized by the total length of the restriction strings.*

5 Approximation Algorithms

We next present two approximation algorithms for the RLCS problem. The first algorithm provides a $\frac{1}{|\Sigma|}$ approximation ratio for the case of both arbitrary number of input strings and arbitrary number of restrictions. Here Σ is the set of letters used in the instance, *i.e.* the actual alphabet of the comparison and

restriction strings. Afterwards, we present an $\frac{k_{min}-1}{n_{min}}$-approximation algorithm, where n_{min} and k_{min} are the lengths of the shortest input string and the shortest restriction, respectively. This algorithm is relevant only for the case of fixed number input strings (and arbitrary number of restrictions). Both algorithms are very simple. This situation should be compared with CLCS, where in general no approximation can be sought unless P=NP, and for the case of a single constraint string only a ratio of $\frac{1}{\sqrt{n_{min}|\Sigma|}}$ is known.

Algorithm I:

1. For every $s \in \Sigma$ and every $i \in \{1, \ldots, m\}$, compute the number of occurrences $Occ_i(s)$ of s in A_i.
2. For every $s \in \Sigma$ compute:
   - $Occ(s)$, the minimum between $Occ_1(s), \ldots, Occ_m(s)$.
   - $Cons(s)$, the length of the shortest restriction string that does not contain any symbol besides s. If no such restriction string exists, $Cons(s) = \infty$.
   - $Val(s)$, the minimum between $Occ(s)$ and $Cons(s) - 1$.
3. Find $s \in \Sigma$ with maximal $Val(s)$ and return $s^{Val(s)}$.

The following lemma is easily established:

Lemma 4. *Given a RLCS instance with arbitrary number input strings and arbitrary number of restrictions, Algorithm I yields an approximation ratio of* $\frac{1}{|\Sigma|}$.

Proof. There is no restricted common subsequence of $A_1, \ldots, A_m$ that contains more than $Val(s)$ occurrences of any $s \in \Sigma$, and thus Algorithm I returns a $\frac{1}{|\Sigma|}$-approximate solution. □

Our second algorithm is even simpler than the first:

Algorithm II:

1. Compute the LCS S of $A_1, \ldots, A_m$.
2. Return the prefix of length $k_{min} - 1$ of S.

Lemma 5. *Given a RLCS instance with fixed number input strings (and arbitrary number of restrictions), Algorithm II yields an approximation ratio of* $\frac{k_{min}-1}{n_{min}}$.

Proof. If the LCS of $A_1, \ldots, A_m$ is shorter than k_{min}, then Algorithm II finds an optimal RLCS. Otherwise, it outputs an RLCS of length $k_{min} - 1$ which yields an approximation ratio of $\frac{k_{min}-1}{n_{min}}$. □

References

1. Aho, A.V., Hirschberg, D.S., Ullman, J.D.: Bounds on the Complexity of the Longest Common Subsequence Problem. Journal of the ACM 23(1), 1–12 (1976)
2. Amir, A., Hartman, T., Kapah, O., Shalom, B.R., Tsur, D.: Generalized LCS. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 50–61. Springer, Heidelberg (2007)

3. Andrejkova, G.: The Longest Restricted Common Subsequence Problem. In: Proceedings, Prague Stringology Club Workshop 1998, pp. 14–25 (1998)
4. Apostolico, A., Landau, G.M., Skiena, S.: Matching for Run-Length Encoded Strings. Journal of Complexity 15(1), 4–16 (1999)
5. Arslan, A.N., Egecioglu, Ö.: Algorithms For The Constrained Longest Common Subsequence Problems. International Journal of Foundations of Computer Science 16(6), 1099–1109 (2005)
6. Bergroth, L., Hakonen, H., Raita, T.: A Survey of Longest Common Subsequence Algorithms. In: Proc. SPIRE 2000, pp. 39–48 (2000)
7. Berman, P., Karpinski, M.: On Some Tighter Inapproximability Results. Electronic Colloquium on Computational Complexity 5(29) (1998)
8. Chen, Y.C., Chao, K.M.: On the generalized constrained longest common subsequence problems. Journal of Combinatorial Optimization, 1–10 (2009)
9. Chin, F.Y.L., De Santis, A., Ferrara, A.L., Ho, N.L., Kim, S.K.: A simple algorithm for the constrained sequence problems. Information Processing Letters 90(4), 175–179 (2004)
10. Gotthilf, Z., Hermelin, D., Lewenstein, M.: Constrained LCS: Hardness and Approximation. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 255–262. Springer, Heidelberg (2008)
11. Gotthilf, Z., Lewenstein, M.: Approximating Constrained LCS. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 164–172. Springer, Heidelberg (2007)
12. Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. Communications of the ACM 18(6), 341–343 (1975)
13. Hirschberg, D.S.: Algorithms for the Longest Common Subsequence Problem. Journal of the ACM 24(4), 664–675 (1977)
14. Iliopoulos, C.S., Rahman, M.S.: New efficient algorithms for the LCS and constrained LCS problems. Information Processing Letters 106(1), 13–18 (2008)
15. Iliopoulos, C.S., Rahman, M.S., Rytter, W.: Algorithms for two versions of the lcs problem for indeterminate strings. Journal of Combinatorial Mathematics and Combinatorial Computing (2008)
16. Maier, D.: The Complexity of Some Problems on Subsequences and Supersequences. Journal of the ACM 25(2), 322–336 (1978)
17. Masek, W.J., Paterson, M.: A Faster Algorithm Computing String Edit Distances. Journal of Computer and System Sciences 20(1), 18–31 (1980)
18. Tsai, Y.-T.: The constrained longest common subsequence problem. Information Processing Letters 88(4), 173–176 (2003)

Extracting Powers and Periods in a String from Its Runs Structure

Maxime Crochemore[1,3], Costas Iliopoulos[1,4], Marcin Kubica[2], Jakub Radoszewski[2,⋆], Wojciech Rytter[2,5], and Tomasz Waleń[2]

[1] King's College London, London WC2R 2LS, UK
maxime.crochemore@kcl.ac.uk, csi@dcs.kcl.ac.uk
[2] Dept. of Mathematics, Computer Science and Mechanics, University of Warsaw, Warsaw, Poland
{kubica,jrad,rytter,walen}@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics, Copernicus University, Toruń, Poland

Abstract. A breakthrough in the field of text algorithms was the discovery of the fact that the maximal number of runs in a string of length n is $O(n)$ and that they can all be computed in $O(n)$ time. We study some applications of this result. New simpler $O(n)$ time algorithms are presented for a few classical string problems: computing all distinct kth string powers for a given k, in particular squares for $k = 2$, and finding all local periods in a given string of length n. Additionally, we present an efficient algorithm for testing primitivity of factors of a string and computing their primitive roots. Applications of runs, despite their importance, are underrepresented in existing literature (approximately one page in the paper of Kolpakov & Kucherov, 1999). In this paper we attempt to fill in this gap. We use Lyndon words and introduce the Lyndon structure of runs as a useful tool when computing powers. In problems related to periods we use some versions of the Manhattan skyline problem.

Keywords: run in a string, square, local period.

1 Introduction

The structure of all runs in a string provides succinct and very useful information about periodic properties of the string. Several basic applications of this structure were given in [14]. We present some other algorithmic applications of runs and simplify already known algorithms.

First we consider the problem of computing all *distinct* kth powers in a string of length n, for a given k. It is a known fact that the number of distinct squares

⋆ Corresponding author. Some parts of this paper were written during the corresponding author's Erasmus exchange at King's College London.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 258–269, 2010.

($k = 2$) does not exceed $2n$ [8,11,12] and for cubes ($k = 3$) there is a $0.8n$ bound [15], which implies same bound for any value $k \geq 4$. Gusfield & Stoye [10] present an $O(n)$ time algorithm for computing all the distinct squares. Unfortunately, this algorithm is complicated and uses suffix trees which are a rather heavyweight data structure and add a logarithmic factor depending on the size of alphabet in most implementations. We present a much simpler $O(n)$ time algorithm which computes all distinct kth powers in a string of length n using suffix arrays instead of suffix trees.

Another application of the runs structure is the computation of local periods which are related to the critical factorizations of a string [5]. The known $O(n)$ time algorithm by Duval et al. [6] employs several different techniques modified in a non-trivial way. We present an equally efficient but simpler algorithm using the solution of the Manhattan Skyline Problem.

Finally, we consider factor-primitivity queries, which consist in checking, for any factor of a given word, whether it is primitive and what is its primitive root. This problem has potential applications in data compression, in particular, in run-length encoding and its derivatives. We provide a solution to this problem with $O(n \log^{\epsilon} n)$ preprocessing time, for any $\epsilon > 0$, and $O(\log n)$ query time.

2 Preliminaries

Let u be a word of length n, $u = u[1\,..\,n]$, over a bounded alphabet Σ. We say that an integer p is the (shortest) *period* of $u[1\,..\,n]$ (notation: $p = \mathsf{per}(u)$) if p is the smallest positive integer such that $u[i] = u[i+p]$ holds for all $1 \leq i \leq n-p$.

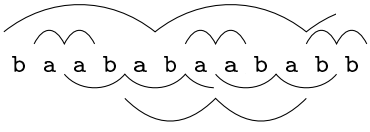


Fig. 1. The structure of runs in the word baababaababb. The word contains 3 runs with period 1, 2 runs with period 2, 1 run with period 3 and 1 run with period 5.

A *run* v (a maximal repetition) in the word u is an interval $[i\,..\,j]$ such that the shortest period $p = \mathsf{per}(v)$ of the associated factor $u[i\,..\,j]$ satisfies $2p \leq j-i+1$, and the interval cannot be extended to the left nor to the right without violating the above property, that is, $u[i-1] \neq u[i+p-1]$ and $u[j-p+1] \neq u[j+1]$, provided that the respective letters exist. Denote by $\mathcal{R}(u)$ the set of all runs in u, each represented as a triple (i, j, p). It is known that $|\mathcal{R}(u)| = O(n)$ [4] and all elements of $\mathcal{R}(u)$ can be computed in $O(n)$ time [14] (a more practical algorithm for computing all runs is given in [2]).

If $w^k = u$ (k is a positive integer) then we say that u is the kth power of the word w. A *square* (*cube*) is the 2nd (3rd) power of a nonempty word. The *primitive root* of a word u, denoted $\mathsf{root}(u)$, is the shortest word w such that

$w^k = u$ for some positive integer k. We call a word u *primitive* if $\mathsf{root}(u) = u$, otherwise it is called *non-primitive*.

Let us recall two useful data structures in string processing.

Suffix Arrays. The suffix array of the word u consists in three tables: SUF, LCP and RANK. The SUF array stores the list of positions in u sorted according to the increasing lexicographic order of suffixes starting at these positions, i.e.:

$$u[\mathsf{SUF}[1]\,..\,n] \;<\; u[\mathsf{SUF}[2]\,..\,n] \;<\; \ldots \;<\; u[\mathsf{SUF}[n]\,..\,n]\ .$$

Thus, indices of SUF are ranks of the respective suffixes in the increasing lexicographic order. The LCP array is also indexed by the ranks of the suffixes, and stores the lengths of the longest common prefixes of consecutive suffixes in SUF. Denote by $lcp(i, j)$ the length of the longest common prefix between $u[i\,..\,n]$ and $u[j\,..\,n]$ (for $1 \le i, j \le n$). Then, we set $\mathsf{LCP}[1] = -1$ and, for $1 < r \le n$, we have:

$$\mathsf{LCP}[r] = lcp(\mathsf{SUF}[r-1], \mathsf{SUF}[r])\ .$$

Finally the RANK table is an inverse of the SUF array:

$$\mathsf{SUF}[\mathsf{RANK}[i]] = i \quad \text{for } i = 1, 2, \ldots, n\ .$$

All tables comprising the suffix array can be constructed in $O(n)$ time [3].

Range Minimum Queries. Define the range minimum query data structure (RMQ, in short) as follows. Assume that we are given an array $A[1\,..\,n]$ of integers. This array is preprocessed to answer the following form of queries: for an interval $[a\,..\,b]$ (for $1 \le a \le b \le n$), find the minimum value $A[k]$ for $a \le k \le b$.

The best known RMQ data structures have $O(n)$ preprocessing time and $O(1)$ query time, using only $O(n)$ bits of space [7,16]. The RMQ data structure on the LCP array enables the computation of longest common extensions, i.e., longest common prefixes between any two suffixes of a string in $O(1)$ time, with $O(n)$ time preprocessing.

3 Lyndon Representations of Runs

Let u be a word of length n. By $rot(u, c)$ let us denote a *cyclic rotation* of the word u obtained by moving (c mod n) first letters of u to its end. We say that the words u and $rot(u, c)$ are *cyclically equivalent*. A word that is both primitive and lexicographically minimal in the class of its cyclic rotations is called a *Lyndon word*. We define the *Lyndon root* of a word u, $\mathsf{lroot}(u)$, as the (only) Lyndon word cyclically equivalent to $\mathsf{root}(u)$. We define the Lyndon root of a run $v = (i, j, p)$ in u, $\mathsf{lroot}(v)$, as $\mathsf{lroot}(u[i\,..\,i + p - 1])$, note that this notion is slightly different from the corresponding notion for words.

Denote by $u_{(a)}$ a prefix of the word u of length a and by $u^{(a)}$ a suffix of u of length a. Each run v can be uniquely represented (*Lyndon representation*) in the following form:

$$v \;\doteq\; \lambda^{(a)} \cdot \lambda^m \cdot \lambda_{(b)} \tag{1}$$

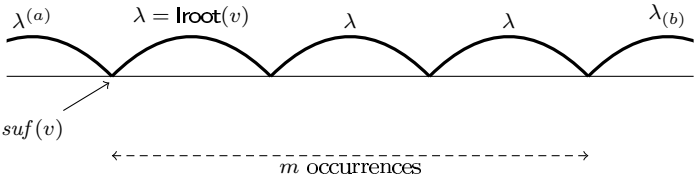


Fig. 2. A graphical view of the Lyndon representation of a run $v = \lambda^{(a)} \cdot \lambda^m \cdot \lambda_{(b)}$

where $\lambda = \mathsf{lroot}(v)$ and $0 \leq a, b < \mathsf{per}(v)$, see Fig. 2. We say that v is a λ-run. We will divide all runs of $\mathcal{R}(u)$ into maximal groups of λ-runs.

For a run $v = (i, j, p)$, define $suf(v)$, $suf(v) \geq i$, as the smallest index for which:

$$u[suf(v) \,..\, suf(v) + p - 1] = \mathsf{lroot}(v) \ ,$$

see Fig. 2. This parameter, together with the period $\mathsf{per}(v)$, provides a unique characterization of the Lyndon root of the run. Additionally define $rank(v) = \mathsf{RANK}[suf(v)]$.

Lemma 1. *The values of $suf(v)$ and $rank(v)$ for any run v in a word u of length n can be computed in $O(1)$ time assuming $O(n)$ time preprocessing.*

Proof. Let $v = (i, j, p)$. The value of $rank(v)$ can be computed using RMQ on the interval $I = [i \,..\, i + p - 1]$ of the table RANK. Indeed, the prefixes of length p of the suffixes $\{u[d \,..\, n] \ : \ d \in I\}$ are exactly all cyclic rotations of $\mathsf{lroot}(v)$. Recall that RMQ for an array of length n can be implemented with $O(n)$ preprocessing time and $O(1)$ query time. Finally, $suf(v) = \mathsf{SUF}[rank(v)]$. □

Theorem 1. *The set $\mathcal{R}(u)$ of all runs within u can be decomposed into pairwise disjoint classes $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_t$ corresponding to runs with equal Lyndon roots in $O(n)$ time, where $n = |u|$.*

Proof. We start the proof of the theorem with the following claim.

Claim 2. *The equality of Lyndon roots of runs (represented as pairs of the form $(\mathsf{per}(v), rank(v))$) in u can be tested in $O(1)$ time with $O(n)$ preprocessing time. Moreover, if $L = v_1, v_2, \ldots, v_a$ is a list of all runs in u with period p sorted in ascending order of the values of parameter rank, then all runs in u with the same Lyndon root λ, $|\lambda| = p$, form a sublist of L composed of a number of consecutive elements.*

Proof. The Lyndon roots of two runs v_1 and v_2 are equal if and only if $\mathsf{per}(v_1) = \mathsf{per}(v_2)$ and the longest common prefix of suffixes at positions $suf(v_1)$ and $suf(v_2)$ is at least $\mathsf{per}(v_1)$. Recall that longest common prefixes of arbitrary suffixes can be computed using RMQ on the LCP array, which proves the first part.

As for the second part of the claim, assume that for three runs v_1, v_2 and v_3 we have $\mathsf{per}(v_1) = \mathsf{per}(v_2) = \mathsf{per}(v_3) = p$, $rank(v_1) < rank(v_2) < rank(v_3)$ and $\mathsf{lroot}(v_1) = \mathsf{lroot}(v_3)$. Then

$$lcp(suf(v_1), suf(v_3)) \;\geq\; p\ ,$$

however due to the rank inequalities we have

$$lcp(suf(v_1), suf(v_3)) \;=\; \min(lcp(suf(v_1), suf(v_2)),\ lcp(suf(v_2), suf(v_3)))\ .$$

Therefore

$$lcp(suf(v_1), suf(v_2)) \;\geq\; p$$

and consequently $\mathsf{lroot}(v_1) = \mathsf{lroot}(v_2) = \mathsf{lroot}(v_3)$. □

Using Claim 2, the requested decomposition of $\mathcal{R}(u)$ can be obtained in $O(n)$ time in the following three steps, recall that $|\mathcal{R}(u)| = O(n)$.

1. Compute the values of $suf(v)$ and $rank(v)$ for all runs in $\mathcal{R}(u)$ — $O(n)$ time in total due to Lemma 1.
2. Represent all runs v in u as pairs $(\mathsf{per}(v), rank(v))$, sort all such pairs lexicographically — $O(n)$ time using radix sort.
3. Group runs with equal Lyndon roots — due to Claim 2 the groups consist in consecutive runs in the sorted order of pairs, and equality of Lyndon roots of runs can be tested in $O(1)$ time with $O(n)$ time preprocessing, what gives $O(n)$ time complexity of this step. □

Define the *compact Lyndon representation* of a run $v = (i, j, p)$ as a tuple:

$$v \;\triangleq\; (i, j, p, a, m, b, \ell) \tag{2}$$

where ℓ is the length of v and a, m, b are defined as in the (ordinary) Lyndon representation (1). Due to the following lemma, the compact Lyndon representations of runs can be computed efficiently:

Lemma 3. *The compact Lyndon representation of runs (represented as (i, j, p)) in a word u of length n can be computed in $O(1)$ time with $O(n)$ time preprocessing.*

Proof. For a run $v = (i, j, p)$ of length $\ell = j - i + 1$, knowing the value of $suf(v)$ the compact Lyndon representation of v can be computed using the following additional formulas:

$$a = suf(v) - i, \qquad m = \lfloor (\ell - a)/p \rfloor\ , \qquad b = \ell - a - mp\ .$$

Hence, the statement is a consequence of Lemma 1. □

4 Inferring Powers from Runs

Denote by $\#powers(u,k)$ the total number of *distinct* kth powers in a string u. In this section we present an algorithm for efficiently computing this function as well as reporting the corresponding powers. By reporting we mean returning the vector $POWERS$ such that, for each i, $POWERS[i]$ is the set of periods of all kth powers which have the last occurrence starting at position i. These sets have cardinality at most two [8,11,12].

Each kth power w^k (for $k \geq 2$) occurring in u corresponds to a run v containing this occurrence for which $\mathsf{per}(v) = |\mathsf{root}(w)|$, we say that w^k is *induced* by the run. If $\mathsf{lroot}(w) = \lambda$ then we call w^k λ-*compatible*. Note that two runs may induce the same power only if their Lyndon roots are equal.

For a λ-run v define $maxpower(v)$ as the maximal natural β such that some cyclic rotation of $\lambda^{k\beta}$ is induced by v.

Observation 4. *If v is a run of length ℓ with period p then $maxpower(v) = \lfloor \ell/(kp) \rfloor$.*

The following lemma shows a correspondence between Lyndon representation of a run and the set of induced distinct kth powers.

Lemma 5. *Let v be a λ-run with period p and let $\beta = maxpower(v)$. Then all powers induced by v are:*

- *all cyclic rotations of $\lambda^{k\alpha}$ for $\alpha < \beta$*
- *cyclic rotations $rot(\lambda^{k\beta}, c)$ for $c \in \mathcal{I}(v)$, where $\mathcal{I}(v) \subseteq [0\,..\,p)$ is a union of at most two intervals.*

Proof. Let $v \doteq \lambda^{(a)} \cdot \lambda^m \cdot \lambda_{(b)}$ be a run of length ℓ with period p.

Note that for a given α the run v induces all cyclic rotations $rot(\lambda^{k\alpha}, c)$ for $c \in [p-a,\ p-a+\ell-kp\cdot\alpha]$. In particular, for $\alpha < \beta$, we obtain all distinct cyclic rotations, since $\ell - kp\cdot\alpha \geq p$. For $\alpha = \beta$, the aforementioned interval for the value of c must be treated modulo p and forms either a single subinterval of $[0\,..\,p)$ or a sum of at most two intervals $\mathcal{I}(v)$. For $\alpha > \beta$, no cyclic rotation of the word $\lambda^{k\alpha}$ is present in v, since $|\lambda^{k\alpha}| > |v|$. □

Let $maxruns(u,\lambda)$ be the set of λ-runs of u with maximal value of $maxpower(v)$. Denote by $\#powers_\lambda(u,k)$ the number of λ-compatible k-powers in u. The following lemma is a consequence of Lemma 5.

Lemma 6. *For a word u let $\beta(\lambda) = \max\{maxpower(v) \ : \ v \in \lambda\text{-}runs(u)\}$. Then*

$$\#powers_\lambda(u,k) \ = \ (\beta(\lambda)-1)\cdot|\lambda| + \left| \bigcup_{v \in maxruns(u,\lambda)} \mathcal{I}(v) \right| ,$$

$$\#powers(u,k) \ = \ \sum_{\lambda} \#powers_\lambda(u,k) \ .$$

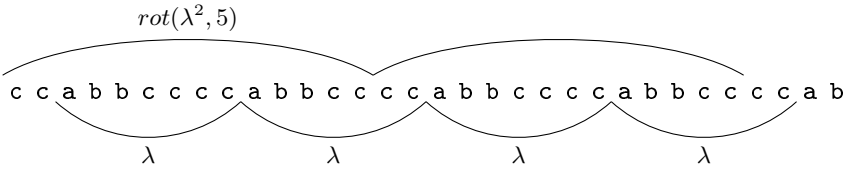


Fig. 3. The run $\lambda^{(2)}\lambda^4\lambda_{(2)}$ with the Lyndon root $\lambda = \texttt{abbcccc}$ induces all possible distinct squares cyclically equivalent to λ^2 and 5 squares cyclically equivalent to λ^4, that is, $maxpower(v) = 2$ and $\mathcal{I}(v) = [0\,..\,2] \cup [5\,..\,6]$

Theorem 2. *For a given word u of length n, the value $\#powers(u, k)$ can be computed and all distinct kth powers in u can be reported in $O(n)$ time.*

Proof. The value $\#powers(u, k)$ can be computed using the formulas from Lemma 6, assuming that we have the decomposition of $\mathcal{R}(u)$ from Theorem 1 and the compact Lyndon representations of all runs, which are necessary to compute the values of $\beta(\lambda)$ and $\mathcal{I}(v)$ (see the formulas in Lemma 5). The only difficulty is to find the size of the union of the sets $\mathcal{I}(v)$ for a given group of λ-runs $\mathcal{R}_y$ in $O(|\mathcal{R}_y|)$ time. Note that this can be performed in a simple way if the sets to be summed form a list of intervals sorted in non-decreasing order (intervals treated as pairs). Due to Lemma 5, each set $\mathcal{I}(v)$ can be divided into a constant number of intervals. Finally, all intervals across all the groups $\mathcal{R}_y$ can be sorted using radix sort in $O(n)$ time.

The algorithm reporting all powers is a natural extension of the algorithm computing $\#powers(u, k)$ using the exact formulas from Lemma 5, we omit the technical description of the algorithm in this version of the paper. □

Denote by $\#occ\text{-}powers(u, k)$ the total number of *occurrences* of kth powers in a string u. We end this section presenting a formula for $\#occ\text{-}powers(u, k)$ which can be evaluated in a straightforward manner to obtain an $O(n)$ time algorithm, where $n = |u|$. Note that the value of the formula can be $\Theta(n^2)$.

Theorem 3

$$\#occ\text{-}powers(u, k) = \sum_{(i,j,p)\in\mathcal{R}(u)} c(i, j, p) \cdot (j - i + 2 - kp/2) - c(i, j, p)^2 \cdot kp/2$$

where

$$c(i, j, p) = \left\lfloor \frac{j - i + 2}{kp} \right\rfloor . \tag{3}$$

The proof of the theorem will be included in the full version of the paper.

5 Computation of Local Periods

By $P = \{p_1, p_2, \ldots, p_{n-1}\}$ we denote the set of inter-positions that are located *between* pairs of consecutive letters of $u[1\,..\,n]$. We say that a square ww is

centered at inter-position p_i of u if both of the following conditions hold, for $x = u[1\,..\,i]$ and $y = u[i+1\,..\,n]$:

- w is a suffix of x or x is a suffix of w
- w is a prefix of y or y is a prefix of w.

We define the *local period* at inter-position p_i (notation: $\mathsf{localper}[i]$) as $|w|$, where ww is the shortest square centered at this inter-position, see also Fig. 4. Clearly, for any p_i there are three possible cases:

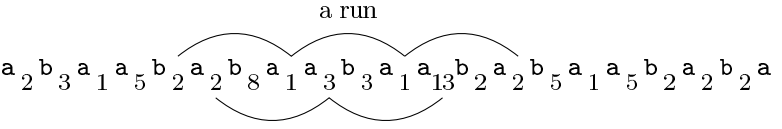


Fig. 4. A Fibonacci string with local periods at all its inter-positions. Local period at inter-position p_9 of the string is 3, since the smallest period q of a run which completely covers the factor of the string corresponding to the interval $[9-q+1\,..\,9+q]$ equals 3.

Case A: $|w| \le \min(|x|, |y|)$, i.e., ww is an *internal* square of u.

Case B: $\min(|x|, |y|) < |w| \le \max(|x|, |y|)$, i.e., ww is a *left-external* square (if $|w| > |x|$) or a *right-external* square (if $|w| > |y|$).

Case C: $\max(|x|, |y|) < |w|$, i.e., ww is a *both-sides-external* square.

We handle Cases A-C separately. In Case A we use the structure of runs in u and perform a reduction to the Manhattan Skyline Problem. In Cases B and C we use the border array from the Morris-Pratt algorithm, which is a simple alternative to a modified Boyer-Moore shift function used for this purpose in [6].

Case A: internal local periods. The problem of the internal local periods can be reduced in $O(n)$ time to the (restricted min-version) of the following problem:

Restricted Manhattan Skyline Problem

Input:
given a set $\mathcal{S}$ of $O(n)$ subintervals of $[1\,..\,n-1]$ with natural heights of size $O(n)$;

Output:
the table $f[t] = \min\{height([i\,..\,j]) : t \in [i\,..\,j],\ [i\,..\,j] \in \mathcal{S}\},\ t \in [1\,..\,n-1]$.

Indeed, note that any internal local period corresponds to a primitively rooted square in u, induced by one of the runs of u, see also Fig. 4. Each run $v = (a, b, q)$ in u induces such squares with root q at inter-positions $p_{a+q-1}, p_{a+q}, \ldots, p_{b-q}$. Thus for each inter-position p_i we need to find the shortest period of a run (i.e., height of an interval from the Manhattan Skyline Problem) inducing a square at this inter-position.

The following two lemmas show how to utilize the described reduction to construct a linear time algorithm for computing internal local periods.

Lemma 7. *Assume initially $X = \emptyset$ and all considered intervals $[i\,..\,j]$ are from the universe $[1\,..\,m]$. Then the sequence of $O(m)$ pairs of operations:*

$$\{ \quad \textit{list-all-elements}([i\,..\,j] \setminus X); \quad X \leftarrow X \cup [i\,..\,j]; \quad \} \tag{4}$$

can be implemented in $O(m)$ time.

Proof. The implementation uses a restricted version of the find/union data structure, in which we are allowed to union only adjacent subintervals. Thus the *structure* of union operations forms a static tree (here it is a path graph) and therefore $O(m)$ find/union operations can be performed in $O(m)$ time [9] (see also [13]).

In the algorithm the universe $[1\,..\,m{+}1]$ (extended to the right by a sentinel) is partitioned into maximal segments of elements of X followed by a single element which is not in X: all elements in such a segment form a single find/union component which stores the index of its rightmost position. The operations (4) are implemented by traversing the components intersecting the interval $[i\,..\,j]$, reporting their rightmost elements and summing them one by one. □

Lemma 8. *The internal local periods can be computed in linear time.*

Proof. We showed that the problem can be reduced to the restricted Manhattan Skyline Problem. This problem can be solved in $O(n)$ time as follows.

Sort intervals from $\mathcal{S}$ according to their heights (in increasing order);
Initialize $X = \emptyset$;
for each interval $[i\,..\,j] \in \mathcal{S}$ (in the sorted order) **do**
  for each $t \in$ list-all-elements$([i\,..\,j] \setminus X)$ **do**
    $f[t] \leftarrow \textit{height}([i\,..\,j])$;
  $X \leftarrow X \cup [i\,..\,j]$;

According to Lemma 7, the set operations in the above pseudocode can be implemented in linear time. This completes the proof. □

Case B: one-side-external local periods. Recall that a word that is both a prefix and a suffix of a word u is called a *border* of the word u; a border of u is called proper if it is shorter than u. Denote by $\mathsf{border}[i]$, for $i = 1, 2, \ldots, n$, the length of the longest proper border of $u[1\,..\,i]$. Recall that the border array can be computed in $O(n)$ time, as in the Morris-Pratt algorithm [5].

The following lemma shows how the border array can be used to compute left-external local periods, the case of right-external local periods is symmetric and can be treated similarly by considering the reversed word u. The proof of the lemma will be present in the full version of the paper.

Lemma 9

(a) If the local period at inter-position p_i is left-external (and not right external) then there exists $j > i$ such that $\mathsf{border}[j] = i$ and $\mathsf{localper}[i] = j - i$.
(b) If $\mathsf{border}[j] = i$ for any $j = 2, 3, \ldots, n$ and $i > 0$ then $\mathsf{localper}[i] \leq j - i$.

Due to Lemma 9, the localper array can be updated in $O(n)$ time by considering all left-external local periods corresponding to the values border$[j]$ for all $j = 1, 2, \ldots, n$.

Case C: both-sides-external local periods. Consider a both-sides-external local period at inter-position p_i of u. If b is the longest overlap between $u[i{+}1\,..\,n]$ and $u[1\,..\,i]$, i.e., the longest suffix of the former word which is also a prefix of the latter word, then localper$[i] = n - b$, see Fig. 5. Note that b is the length of the longest border of u which is not longer than $\min(i, n - i)$.

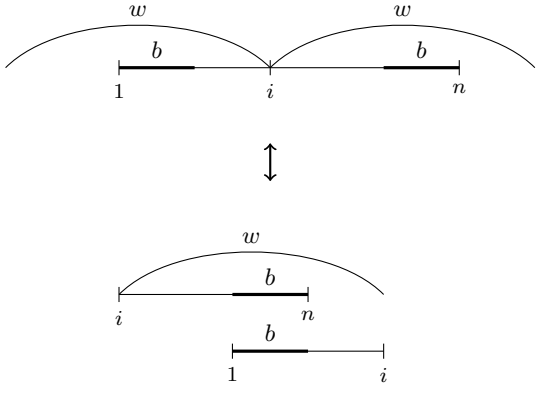


Fig. 5. The correspondence between both-sides-external local periods and borders

Recall that the lengths of all proper borders of u are iterations of the form border$^{(j)}[n]$. This concludes an $O(n)$ time algorithm which updates the localper array obtained after the previous cases considering all both-sides-external local periods, filling the array from its middle to its sides.

Combining the solutions to Cases A-C, we obtain the following result.

Theorem 4. *All local periods of a string u of length n can be computed in $O(n)$ time (in a simple way) using the runs structure of u and the* border *array.*

6 Factor-Primitivity Queries

For a given string u of length n, we define a *factor-primitivity query* as follows: for the indices a, b, $1 \le a \le b < n$, check whether the factor $u[a\,..\,b]$ is primitive, and if not, find the length of its primitive root. Let us introduce a notion relating runs with factor-primitivity queries. We say that a run (i, j, p) *completely covers* an occurrence of a factor $u[a\,..\,b]$ in u if $i \le a, b \le j$.

Lemma 10. *Let p be the minimum period of a run completely covering an occurrence of a factor w in a string u (or $p = \infty$ if no such run exists). If $p < |w|$ and $p \mid |w|$ then $|\mathsf{root}(w)| = p$; otherwise w is primitive.*

Proof. Assume first that $q \stackrel{\text{def}}{=} |\mathsf{root}(w)| < |w|$. Then also $\mathsf{per}(w) = q$, see [5]. Hence, w is completely covered by a run with period q and, obviously, by no run with period smaller than q.

On the other hand, if $|\mathsf{root}(w)| = |w|$ then any run completely covering w and having period p satisfies $p = |w|$ or $p \nmid |w|$. This concludes the proof. □

The conclusion of Lemma 10 can again be interpreted using the notion of Manhattan skyline, see Fig. 6.

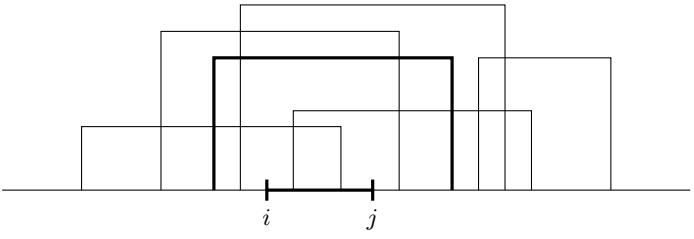


Fig. 6. The buildings in the skyline correspond to runs in a string u and their heights correspond to their periods. When checking primitivity of a factor $w = u[i\,..\,j]$ we look for the lowest building such that w is completely "under its roof".

In our algorithm we utilize yet another interpretation of the problem. To each run (i, j, p) in a word u ($|u| = n$) we assign a point (i, j) in the 2-dimensional plane, and define the value of this point as $f((i, j)) \stackrel{\text{def}}{=} p$. Denote the set of all such points by V. By Lemma 10, to find the primitive root of any factor $u[a\,..\,b]$ of u, it suffices to compute the value

$$\min\{f((i,j)) \;:\; 1 \le i \le a,\ b \le j \le n,\ (i,j) \in V\}\ .$$

This is exactly a *2D range search for minimum* query, which can be answered in the RAM model in: $O(\log^{1+\epsilon} m)$ query time with $O(m)$ preprocessing time, $O(\log m \log\log m)$ query time with $O(m \log\log m)$ preprocessing time, or $O(\log m)$ query time with $O(m \log^{\epsilon} m)$ preprocessing time, where $m = |V| = |\mathcal{R}(u)|$ and ϵ is an arbitrary positive real [1]. Thus we obtain the next result.

Theorem 5. *For a given string u of length n, using the runs structure of u we can answer factor-primitivity queries in $O(n \log^{\epsilon} n)$ preprocessing time, for any $\epsilon > 0$, and $O(\log n)$ query time.*

References

1. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. 17(3), 427–462 (1988)
2. Chen, G., Puglisi, S.J., Smyth, W.F.: Fast and practical algorithms for computing all the runs in a string. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 307–315. Springer, Heidelberg (2007)

3. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
4. Crochemore, M., Ilie, L., Rytter, W.: Repetitions in strings: Algorithms and combinatorics. Theor. Comput. Sci. 410(50), 5227–5235 (2009)
5. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific, Singapore (2003)
6. Duval, J.-P., Kolpakov, R., Kucherov, G., Lecroq, T., Lefebvre, A.: Linear-time computation of local periods. Theor. Comput. Sci. 326(1-3), 229–240 (2004)
7. 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)
8. Fraenkel, A.S., Simpson, J.: How many squares can a string contain? J. of Combinatorial Theory Series A 82, 112–120 (1998)
9. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC), pp. 246–251 (1983)
10. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. 69(4), 525–546 (2004)
11. Ilie, L.: A simple proof that a word of length n has at most $2n$ distinct squares. J. of Combinatorial Theory Series A 112, 163–164 (2005)
12. Ilie, L.: A note on the number of squares in a word. Theoretical Computer Science 380, 373–376 (2007)
13. Itai, A.: Linear time restricted union/find (2006), http://www.cs.technion.ac.il/~itai/Courses/ds2/lectures/lecture.html
14. Kolpakov, R.M., Kucherov, G.: On maximal repetitions in words. J. Discrete Algorithms 1, 159–186 (1999)
15. Kubica, M., Radoszewski, J., Rytter, W., Walen, T.: On the maximal number of cubic subwords in a string. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 345–355. Springer, Heidelberg (2009)
16. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discrete Algorithms 5(1), 12–22 (2007)

On Shortest Common Superstring and Swap Permutations

Zvi Gotthilf[1], Moshe Lewenstein[1,⋆], and Alexandru Popa[2]

[1] Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
{gotthiz,moshe}@cs.biu.ac.il
[2] Department of Computer Science, University of Bristol, UK
popa@cs.bris.ac.uk

Abstract. The Shortest Common Superstring (SCS) is a well studied problem, having a wide range of applications. In this paper we consider two problems closely related to it. First we define the *Swapped Restricted Superstring*(SRS) problem, where we are given a set S of n strings, s_1, s_2, ..., s_n, and a text $T = t_1t_2 \dots t_m$, and our goal is to find a swap permutation $\pi : \{1, \dots, m\} \rightarrow \{1, \dots, m\}$ to maximize the number of strings in S that are substrings of $t_{\pi(1)}t_{\pi(2)} \dots t_{\pi(m)}$. We then show that the *SRS* problem is *NP-Complete*. Afterwards, we consider a similar variant denoted *SRSR*, where our goal is to find a swap permutation $\pi : \{1, \dots, m\} \rightarrow \{1, \dots, m\}$ to maximize the total number of times that the strings of S appear in $t_{\pi(1)}t_{\pi(2)} \dots t_{\pi(m)}$ (we can count the same string s_i as a substring of $t_{\pi(1)}t_{\pi(2)} \dots t_{\pi(m)}$ more than once). For this problem, we present a polynomial time exact algorithm.

1 Introduction

1.1 Motivation

In the shortest common superstring problem we are given a set S of n strings, $s_1, s_2, \dots, s_n$ and we want to find the shortest string that is a superstring on every string in S. We consider its motivation as follows. Given a set of goals (or tasks) which have to be accomplished, we want to find the most cost efficient plan which achieves all the goals. One can notice that in this case we have two major assumptions:

- We have an unlimited set of resources.
- The resources are independent and we can use them in any order.

However, in real life this is never the case: our resources are always limited and might be dependent on each other. Therefore, we ask a more realistic question: given a fixed set of resources with restrictions on their order, how many goals can be achieved ?

To do so, we present the following pattern matching model of the superstring problem while using swap permutations. First we assume to have a fixed set of

⋆ This work was partially supported by the Israel Science Foundation grant 1484/08.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 270–278, 2010.

resources, which are defined as a text $T = t_1 t_2 \dots t_m$. The set of tasks corresponds to the set S of n strings, s_1, s_2, ..., s_n. Then, we limit the way we can use our resources by using swap permutation on the text T. Hence, our goal is to find the best valid arrangement of these resources that leads us to accomplish the maximum number of goals.

It seems that several applications of the *Shortest Common Superstring* problem are more suitable for the case of fixed set of resources and arrangement restriction.

In AI planning research it is very important to exploit the interactions between different parts of plans. This was observed early in the area [15,17,19]. One very important type of interaction is the *merging* of different actions to make the total plan more efficient. Consider the case where a plan already exists and a set of actions is performed according to this plan. We now may want improve the plan to achieve a better performance, however, we do not want to (or we cannot) alter the current plan, thus we are interested in performing local improvements in the existed plan.

Moreover, consider a factory that places a set of machines in a single hall. Now, given a set of products that has to be manufactured, we would like to find the most efficient arrangement of the machines that maximizes the number of manufactured products. The problem is that even if we can find the best arrangement, it is not possible to shift every machine into its optimal position, instead we only use local rotations between adjacent machines.

1.2 Previous Work

In the shortest common superstring problem we are given a set S of n strings, $s_1, s_2, \dots, s_n$ and we want to find the shortest string that is a superstring on every string in S. For arbitrary n the problem is known to be *NP-Complete* [9] and APX-hard [7]. Even for the case of binary alphabet Ott [14] presented approximation ratio lower bounds. The best known approximation ratio so far is 2.5 [12,16].

Given a text T of n symbols and a pattern P, a *swapped version* T' of T is a length n string derived from T by a series of local swaps, (i.e. $t'_\ell \to t_{\ell+1}$ and $t'_{\ell+1} \to t_\ell$) where each element can participate in no more than one swap. The pattern matching with swaps problem is that of finding all the locations i for which there exists a swapped version T' of T where there is an exact matching of P at the location i of T'.

In the last decade several results regarding the pattern matching with swaps problem are presented. In [3], Amir et. al. shows that for special cases the pattern matching with swaps problem can be solved in time $O(n \log^2 m)$. Then, in [1] the first $o(mn)$ algorithm for the pattern matching with swaps problem is presented. In [4], Amir et. al. present an algorithm that counts the number of swaps at every location in running time of $O(n \log m \log \sigma)$, where $\sigma = \min(m, |\Sigma|)$. In [20], the problem of pattern matching with swaps in a weighted sequence is considered. Then, in [6] approximation algorithms for the problem of computing the cyclic swap distance between two n-bit (cyclic) binary strings are considered. In [2], it

is shown that approximate string matching problem with the swap and mismatch as the edit operations, can be computed $O(n\sqrt{m \log m})$ running time. Iliopoulos and Rahman [11] present a new algorithm using a graph-theoretic model for the swap matching problem. Then, in [5], an application for this algorithm is presented.

In [8] it is considered the problem of Shortest Common Permutation Superstring. This turns out to be a very general and difficult problem (hard to approximate within a factor of $n^{1-\epsilon}$, for any $\epsilon > 0$, unless $P = NP$ and remains *NP-hard* even in the case of a binary alphabet). The problems we consider here are interesting restricted variants in the sense that the permutations allowed are restricted to swaps. While these are interesting variants because of their practicality, the theoretical implication is quite surprising as well.

It turns out that there is an inherent difference between the *SRS* problem and the *SRSR* problem. One remains hard while the other is polynomial.

1.3 Our Contributions

In this paper we define two variants of the above described model. First we define the *Swapped Restricted Superstring*(SRS) problem, where our goal is to find the best arrangement of our resources (restricted by swap permutations) that leads us to accomplish the maximum number of different goals. For this variant we show that it is *NP-hard* to find the permutation that maximizes the number of different goals that we can accomplish.

Then, we define the *Swapped Restricted Superstring with Repetitions* (SRSR) problem, where our goal is to find the best arrangement of our resources (restricted by swap permutations) that maximizes the total number of goals achieved (here we can count completion of the same goal more than once). For this variant we present an exact algorithm that finds the permutation that maximizes the total number of goals that we can accomplish (with repetitions).

Here we present the formal definition of both problems:

Problem 1. (Swapped Restricted Superstring) The input consists of a set $S = \{s_1, s_2, \ldots, s_n\}$ of n strings over an alphabet Σ and a text $T = t_1 t_2 \ldots t_m$ over the same alphabet. The goal is to find an ordering of the text T that maximizes the number of different strings from S that are substrings of the ordered text.

We denote this ordering by $\pi : \{1, \ldots, m\} \rightarrow \{1, \ldots, m\}$ such that:

- if $\pi(i) = j$, then $\pi(j) = i$.
- for all i, $\pi(i) \in \{i-1, i, i+1\}$, (only adjacent characters can be swapped).
- if $\pi(i) \neq i$ then $t_{\pi(i)} \neq t_i$.

Example 1. Given a text $T = abcab$ and the set $S = \{abc, cab, cbb, acb, bba\}$. The maximum number of different strings from S that can be a substring of a swapped permutation of T is 3. Such a permutation is $acbba$ which contains the strings acb, cbb, bba as substrings.

Problem 2. (Swapped Restricted Superstring with Repetitions) The input consists of a set $S = \{s_1, s_2, \ldots, s_n\}$ of n strings over an alphabet Σ and a text

$T = t_1 t_2 \dots t_m$ over the same alphabet. The goal is to find an ordering of the text T that maximizes the number of occurrences of strings in S that are substrings of the ordered text, such that the string s_i can be counted as a substring more than once.

We denote this ordering by $\pi : \{1, \dots, m\} \to \{1, \dots, m\}$ exactly as in the *SRS* definition.

Example 2. Given a text $T = ababca$ and the set $S = \{ba, bca, aac\}$. The maximum number of occurrences of strings from S that can be a substring of a swapped permutation of T is 3. Such a permutation is *babaac* which contains the string *ba* twice and the string *aac* once as substrings.

The rest of the paper is organized as follows. In Section 2 we show that the *SRS* problem is *NP-Hard*. In Section 3, we present an exact algorithm for the *SRSR* problem. We first present an algorithm based on dynamic programming with the running time $O(mn^2\ell^2)$, where ℓ is the maximum length of a string in the input for the *SRSR* problem. Then, in Section 3.2, we present guidelines for running time improvements of the above algorithm. We end the paper with some open questions.

2 Hardness of Swapped Restricted Superstring

In this section we prove that the SRS problem is NP-Hard. We present a reduction from the *Fragmentary Pattern Matching* which is known to be NP-Hard [10].

Problem 3 (Fragmentary Pattern Matching (FPM)). Given a text $t = t_1 t_2 \dots t_m$ and a set of strings $P = \{p_1, p_2, \dots, p_n\}$ over the alphabet Σ, find a subset $P' = \{p'_1, p'_2, \dots, p'_k\} \subseteq P$ of maximum cardinality such that there exists a permutation π for which $t = \alpha_1 p'_{\pi(1)} \alpha_2 p'_{\pi(2)} \dots \alpha_k p'_{\pi(k)} \alpha_{k+1}$, where $\alpha_i \in \Sigma^*$ for $1 \leq i \leq k+1$ (i.e. all the k strings match the text t and they do not overlap).

Theorem 1 ([10]). *Fragmentary Pattern Matching is NP-Hard.*

The main result of this section is stated in the following theorem.

Theorem 2. *Swapped Restricted Superstring is NP-Hard.*

Proof. Given an instance of the *FPM* we construct the following instance of the *SRS*:

- The alphabet Σ' of the *SRS* instance is $\Sigma \cup \{x_1, x_2, \dots, x_n, x_{n+1}\}$, where $x_1, x_2, \dots, x_n, x_{n+1} \notin \Sigma$.
- We create the text T by concatenating the string $X = x_1 x_2 \dots x_n x_{n+1}$ after each character of the text t in the *FPM* instance. Thus, given a text $t = t_1 t_2 \dots t_m$ in the *FPM* instance, we set the text T of the corresponding *SRS* to be $T = t_1 X t_2 X \dots t_m X$.
- Denote by X_i the string $x_1 x_2 \dots x_{i-1} x_{i+1} x_i x_{i+2} \dots x_n x_{n+1}$. For every string $p_i = a_1 a_2 \dots a_q$ in the *FPM* instance we construct its corresponding string $s_i = a_1 X_i a_2 X_i \dots a_q X_i$ in the *SRS* instance.

We show that the optimal solution of the *FPM* instance is of the same size as the optimal solution of its corresponding *SRS* instance.

Given an *FPM* solution of size k, we construct a solution of the same size to the corresponding *SRS* instance in the following way. Assume that in the FPM instance the strings are placed on the positions $i_1, i_2, \ldots, i_k$, then in the *SRS* instance we can place the corresponding strings in the text T on the positions $(i_1 - 1)(n + 1) + i_1, (i_2 - 1)(n + 1) + i_2, \ldots, (i_k - 1)(n + 1) + i_k$ and swap the text accordingly.

We now prove the converse part. To do so, we show that in any swap permutation of the text such that s_i and s_j are substrings of it, s_i and s_j cannot overlap. Suppose they do, then the text has to be swapped in a way that matches both X_i and X_j. Since these strings are different, this is impossible (contradiction). Thus, we can recover a solution to the *FPM* problem of the same size. □

Example 3. We are given the following instance of the FPM problem: $t = abaab$ and $P = \{ab, ba\}$. The optimal solution of this instance has size 2 (we can place the first string in the fourth position, and the second string in the second position). The corresponding instance of the SRS problem is:
$T = ax_1x_2x_3bx_1x_2x_3ax_1x_2x_3ax_1x_2x_3bx_1x_2x_3$
$S = \{ax_2x_1x_3bx_2x_1x_3, bx_1x_3x_2ax_1x_3x_2\}$

The optimal solution of the *SRS* instance has also size 2: we place the first string on position 13 and the second one on position 5.

3 Exact Algorithm for Swapped Restricted Superstring with Repetitions

In this Section we present a polynomial time algorithm for the SRSR problem. First we give a dynamic programming algorithm and then we improve the running time of this algorithm using suffix trees.

3.1 A Dynamic Programming Approach

Before we present the algorithm, let us introduce a few notations.

First, we define $|s_k|$ to be the length of the string s_k. We define the function $f(i, k)$, for all $1 \leq i \leq m$ and $1 \leq k \leq n$ to be the optimal solution for the text $T = t_1 \ldots t_i$ with the following restriction: in this solution a copy of the string s_k is placed at position $i - |s_k| + 1$. For a pattern $s_k = a_1a_2...a_\ell$, and an integer $1 \leq i \leq \ell$, let $incl(i, k)$ be the total number of times the strings of S appear in $a_ia_{i+1} \ldots a_\ell$, or have a suffix which is a prefix of $a_ia_{i+1} \ldots a_\ell$ (but are still included in s_k). We say that a pattern s_k swap-matches the text at position i if there exists a swap permutation π of the text such that s_k matches $\pi(T)$ at position i.

We show now how to compute the function f. If s_k does not swap-match the text T at position $i-|s_k|+1$, then $f(i, k) = 0$. Otherwise, $f(i, k)$ is the maximum of the following values:

- $f(a,b) + incl(1,k)$, for any $1 \leq a \leq i - |s_k|, 1 \leq b \leq n$.
- $f(j,q) + incl(j - i + |s_k| + 1, k) + 1$, for each index j between $i - |s_k| + 1$ and i and all the patterns s_q such that $j - |s_q| \leq i - |s_k|$ and the last $j - i + |s_k|$ characters of s_q are the same as the first $j - i + |s_k|$ characters of s_k.

The solution of the *SRSR* problem is $max_{i=1}^{m} max_{j=1}^{n} f(i,j)$. Algorithm 1 describes formally the computation of the function f. We assume that at the beginning all the variables are set to 0.

Theorem 3. *Algorithm 1 computes the optimal solution of the SRSR problem in time $O(mn^2\ell^2)$, where ℓ is the maximum length of a string in the input.*

Proof. First we prove that our algorithm computes the correct result. The proof is by induction over the length of the text. Assume that we want to compute the value $f(i,k)$ and we know that $f(j,q)$ contains the correct value for all $1 \leq j < i$ and $1 \leq q \leq n$. We want to compute what is the maximum number of strings in an optimal solution that are not in conflict with the string s_k. This is performed in the lines $20 - 28$ in the Algorithm 1: if we are at position j and we know that the solution ends with a string s_q that agrees with s_k (i.e. the last $j - i + |s_k|$ characters of s_q are the same as the first $j - i + |s_k|$ characters of s_k) we know that we can place both s_k and all the strings in the solution given by $f(j,q)$. In our solution we also want to count the strings that are substrings of s_k: we do so, by defining the function $incl()$. We add only the strings that are substrings, or have a suffix which is a prefix of $s_k[j+1] \ldots s_k[|s_k|]$, since the others are counted already in $f(j,q)$.

The value $max[i]$ is equal to $max_{j=1}^{i} max_{k=1}^{n} f(j,k)$. Since we need this information each time we compute a value $f(i,k)$ we store it to reduce the running time.

The time complexity of the lines $3 - 11$ is $O(n^2\ell^3)$, where ℓ is the maximum length of a string, even if all the computations are performed naively. The running time of the part between the lines $13 - 33$ is $O(mn^2\ell^2)$ which is greater than $O(n^2\ell^3)$. Therefore the total running time is $O(mn^2\ell^2)$.

3.2 A Faster Solution

In this part we show a few methods to improve the running time of Algorithm 1.

First, we observe that we can improve the running time of the algorithm between the lines $3 - 11$ (which computes the $incl()$ function) to $O(n^2\ell)$ as follows. For any two strings s_i and s_k we find all the occurrences of s_k in s_i using the KMP algorithm [13], and therefore compute faster the values of $match()$. If s_k matches s_i at position j, then we set $match(j + |s_k| - 1)$ to one. Then $incl(j,i) \leftarrow incl(j,i) + \sum_{q=j}^{|s_i|} match(q)$, as we do in Algorithm 1. Since the running time of the KMP algorithm is linear in the length of the input, the total running time of this part is $O(n^2\ell)$. We show later how to improve this part using suffix trees.

Then, we observe that $f(i,k)$ is 0 for all the positions i at which the pattern s_k does not swap-match the text. Therefore, we only have to compute the value

Algorithm 1. Computes the optimal solution to the SRSR problem

Input: Text $t = t_1 t_2 \dots t_m$, strings $s_1, s_2, \dots, s_n$.

for i=1 to n **do**
  for k=1 to n **do**
    for j=1 to $|s_i| - |s_k| + 1$ **do**
      if $s_k = s_i[j]s_i[j+1] \dots s_i[j + |s_k| - 1]$ **then**
        $match(j + |s_k| - 1) \leftarrow 1$
      else
        $match(j + |s_k| - 1) \leftarrow 0$
      end if
    end for
    for j=$|s_i| - 1$ downto 1 **do**
      $match(j) \leftarrow match(j) + match(j+1)$
      $incl(j, i) \leftarrow incl(j, i) + match(j)$
    end for
  end for
end for

for i=1 to m **do**
  $max[i] \leftarrow max[i-1]$
  for k=1 to n **do**
    if s_k does not swap match T at position $i - |s_k| + 1$ **or** $|s_k| > i$ **then**
      continue;
    end if
    $f(i, k) \leftarrow max[i - |s_k|] + incl(1, k)$
    for j=i downto $i - |s_k| + 1$ **do**
      for q=1 to n **do**
        if $s_k[1]s_k[2] \dots s_k[j - i + |s_k|] = s_q[|s_q| - (j - i + |s_k|) + 1] \dots s_q[|s_q|]$
        and $j - |s_q| \leq i - |s_k|$ **then**
          if $f(i, k) < f(j, q) + incl(j - i + |s_k| + 1, k) + 1$ **then**
            $f(i, k) \leftarrow f(j, q) + incl(j - i + |s_k| + 1, k) + 1$
          end if
        end if
      end for
    end for
  end for
  if $f(i, k) > max[i]$ **then**
    $max[i] \leftarrow f(i, k)$
  end if
end for

Output: max[m]

$f(i,k)$ for all the other positions. We can compute all the positions at which a string s_i swap-matches the text in time $O(m\ell^{1/3}\log\ell\log\Sigma)$ [1]. Since there are n strings, the running time is $O(nm\ell^{1/3}\log\ell\log\Sigma)$.

We focus now on the lines $20-27$: for a string s_k and a position i we want to compute $f(i,k)$ as fast as possible. The key observation is that we do not have to check all the positions between $i-|s_k|+1$ and i and all the strings s_q, since two strings s_k and s_q can overlap in a solution only if they agree on the overlap. Therefore, we want to find for a string s_k all the strings s_q that have a suffix which is a prefix of s_k. We introduce the following notation: $Overlap(k) = \{(j,q)|s_k[1]s_k[2]\dots s_k[j] = s_q[|s_q|-j+1]\dots s_q[|s_q|]\}$

We precompute $Overlap(k)$ for all the strings s_k, before we enter the main loop from the line 13, as follows. First, we create the suffix tree for the string $s_1\#s_2\#\dots\#s_n$, where $\#\notin\Sigma$ is a special character. The construction of the suffix tree can be done in linear time (in our case $O(\sum_{i=1}^{n}|s_i|)$ using Ukkonen's algorithm [18]. Then for each string k we can find the set $Overlap(k)$ in time $O(\ell+|Overlap(k)|)$: we traverse the suffix tree following the characters of s_k and we check which other strings have a suffix which is a prefix of s_k.

Using the same suffix tree we can also speed up the computation of the $incl()$ function. For a string s_k, all the nodes in the suffix tree that are below it correspond to suffixes of strings s_i that include s_k (and start with s_k). Therefore the computation can be done in time $O(\sum_{i=1}^{n}\sum_{j=1}^{|s_i|}incl(j,i))$.

4 Open Questions

It would be interesting to answer the following questions:

1. Can we find a faster find a faster algorithm for the *SRSR* problem ?
2. Is there an efficient approximation algorithm for the *SRS* problem ?
3. Can we obtain lower bounds for the *SRS* problem?

Acknowledgements. We thank the anonymous reviewers for their useful comments. The third author is funded by an EPSRC PhD studentship.

References

1. Amir, A., Aumann, Y., Landau, G.M., Lewenstein, M., Lewenstein, N.: Pattern matching with swaps. J. Algorithms 37(2), 247–266 (2000)
2. Amir, A., Eisenberg, E., Porat, E.: Swap and mismatch edit distance. Algorithmica 45(1), 109–120 (2006)
3. Amir, A., Landau, G.M., Lewenstein, M., Lewenstein, N.: Efficient special cases of pattern matching with swaps. Inf. Process. Lett. 68(3), 125–132 (1998)
4. Amir, A., Lewenstein, M., Porat, E.: Approximate swapped matching. Inf. Process. Lett. 83(1), 33–39 (2002)
5. Antoniou, P., Iliopoulos, C.S., Jayasekera, I., Sohel Rahman, M.: Implementation of a swap matching algorithm using a graph theoretic model. In: BIRD, pp. 446–455 (2008)

6. Ardila, Y.J.P., Iliopoulos, C.S., Landau, G.M., Mohamed, M.: Approximation algorithm for the cyclic swap problem. In: Stringology, pp. 190–200 (2005)
7. Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M.: Linear approximation of shortest superstrings. J. ACM 41(4), 630–647 (1994)
8. Clifford, R., Gotthilf, Z., Lewenstein, M., Popa, A.: Restricted common superstring and restricted common supersequence. CoRR, abs/1004.0424v2 (2010)
9. Garey, M.R., Johnson, D.S.: Computers and intractability. A guide to the theory of NP-completeness. W. H. Freeman, New York (1979)
10. Hori, H., Shimozono, S., Takeda, M., Shinohara, A.: Fragmentary pattern matching: Complexity, algorithms and applications for analyzing classic literary works. In: Eades, P., Takaoka, T. (eds.) ISAAC 2001. LNCS, vol. 2223, pp. 719–730. Springer, Heidelberg (2001)
11. Iliopoulos, C.S., Sohel Rahman, M.: A new model to solve the swap matching problem and efficient algorithms for short patterns. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 316–327. Springer, Heidelberg (2008)
12. Kaplan, H., Lewenstein, M., Shafrir, N., Sviridenko, M.: Approximation algorithms for asymmetric tsp by decomposing directed regular multigraphs. J. ACM 52(4), 602–626 (2005)
13. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
14. Ott, S.: Lower bounds for approximating shortest superstrings over an alphabet of size 2. In: Widmayer, P., Neyer, G., Eidenbenz, S. (eds.) WG 1999. LNCS, vol. 1665, pp. 55–64. Springer, Heidelberg (1999)
15. Sacerdoti, E.D.: A structure for plans and behavior. American Elsevier (1977)
16. Sweedyk, Z.: A $2\frac{1}{2}$-approximation algorithm for shortest superstring. SIAM J. Comput. 29(3), 954–986 (1999)
17. Tate, A.: Generating project networks. In: IJCAI, pp. 888–893 (1977)
18. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
19. Wilkins, D.E.: Practical planning: Extending the classical ai planning paradigm. Morgan Kaufmann, CA (1988)
20. Zhang, H., Guo, Q., Iliopoulos, C.S.: String matching with swaps in a weighted sequence. In: Zhang, J., He, J.-H., Fu, Y. (eds.) CIS 2004. LNCS, vol. 3314, pp. 698–704. Springer, Heidelberg (2004)

A Self-Supervised Approach for Extraction of Attribute-Value Pairs from Wikipedia Articles

Wladmir C. Brandão[1], Edleno S. Moura[2], Altigran S. Silva[2], and Nivio Ziviani[1]

[1] Department of Computer Science,
Federal University of Minas Gerais, Belo Horizonte, Brazil
{wladmir,nivio}@dcc.ufmg.br
[2] Department of Computer Science,
Federal University of Amazonas, Manaus, Brazil
{edleno,alti}@dcc.ufam.edu.br

Abstract. Wikipedia is the largest encyclopedia on the web and has been widely used as a reliable source of information. Researchers have been extracting entities, relationships and attribute-value pairs from Wikipedia and using them in information retrieval tasks. In this paper we present a self-supervised approach for autonomously extract attribute-value pairs from Wikipedia articles. We apply our method to the Wikipedia automatic infobox generation problem and outperformed a method presented in the literature by 21.92% in precision, 26.86% in recall and 24.29% in F1.

1 Introduction

Wikipedia started in 2001 aiming to enable collaborative publication and dissemination of ideas and concepts on the web. Since then, people around the world share their knowledge through a platform that allows the organization of information into articles accessible and editable by anyone. Available in several languages, Wikipedia currently has a collection of over 16 million articles and over 24 million registered users. Each Wikipedia article published by one user is reviewed by others that evaluate issues of accuracy and completeness. This evaluation activity makes Wikipedia a valuable source of reliable information on the web.

The feasibility of using Wikipedia as a source of information has been shown to extract entities [28], relationships [20], and attribute-value pairs [1,30,31]. The information extracted from Wikipedia can be used in many information retrieval tasks, such as question answering [12,13], query expansion [17,19], multilingual information retrieval [23], and text categorization [3,29]. The work in [11] illustrates a practical application of an attribute-value pairs extraction approach.

This paper presents WAVE (Wikipedia Attribute-Value Extractor), a self-supervised approach to autonomously extract attribute-value pairs from Wikipedia articles. Our approach is self-supervised in the sense that it uses a priori available information to learn a baseline extractor and the training proceeds repeatedly by using the decisions of the extractor at step s to train the extractor

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 279–289, 2010.

at step $s+1$. WAVE learns how to extract an unlimited number of non-predefined attribute-value pairs from articles represented in the *enriched plain text format*. For example, consider a dataset "University": a possible *attribute* of this dataset could be a "name" with *value* equal to "Stanford University".

To verify the quality of the attribute-value pairs obtained we applied our approach to populate infoboxes with attribute-value pairs extracted from Wikipedia articles, a problem known as automatic infobox generation. An infobox is a special tabular structure that summarizes the content of Wikipedia articles displaying factual information in a set of attribute-value pairs. We considered as baseline the Kylin system [31], which also extracts attribute-value pairs from Wikipedia articles to generate infoboxes. We used precision, recall and F1 as measures to compare the results obtained by WAVE with the results obtained by the baseline. Experimental results show that WAVE outperforms the baseline by 21.92% in precision, 26.86% in recall and 24.29% in F1.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 presents WAVE components. Section 4 compares WAVE with the baseline. Finally, Section 5 concludes our work.

2 Related Work

Several approaches have addressed the problem of information extraction from Wikipedia. The DBPedia system [1] extracts attribute-value pairs from existing infoboxes associated with articles and turns it into an Resource Description Framework (RDF) knowledge base which can be later accessed by users. An RDF [25] extends the linking structure of the Web to use Unified Resource Identifiers (URIs) to name the relationship between things as well as the two ends of the link. This linking structure is usually referred to as a "triple".

The Yago system [28] extends WordNet [16] taxonomy using a mixed suite of heuristics to extract information from Wikipedia's category tags. It also presents a basic data model of entities and a logic-based representation language to allow queries from users. WordNet is a large lexical database for the English language that is widely used in computational linguistics and natural language processing researches.

The work in [20] uses simple heuristics developed by the authors themselves to extract relations from Wikipedia. The heuristics exploit lexical and syntactic patterns and combine them with several techniques such as anaphora resolution, full dependency parsing and subtree mining.

The Kylin system [31] extracts attribute-value pairs from articles and autonomously generates infoboxes. It uses maximum entropy classifiers [21] to associate sentences to attributes and conditional random fields (CRF)[1] [15,26] extractors to take attribute-values pairs from sentences. In a later work, some improvements were proposed in the Kylin system [30] by making it capable of

[1] Conditional random fields (CRF) is a framework for building probabilistic models to segment and label sequence data.

dealing with rare infobox templates. First, the system refers to WordNet's ontology and aggregate attributes from parents to children infobox templates, e.g., knowing that *isA(Actor, Person)* infobox for *Actors* receives prior missing field *BirthPlace*. Second, the system apply TextRunner [10] to the web in order to retrieve additional sentences describing the same attribute-value pairs. Third, the system uses the Google search engine [9] to retrieve additional sentences describing an article. The combination of these techniques improves the recall by 2% to 9% while maintaining increasing precision.

Wave differs from previous approaches in several ways. Differently from DBPedia[1], Wave not only extract attribute-value pairs from existing infoboxes but it can also learn how to extract attribute-value pairs from new articles. Unlike the Yago system [28] and the approach presented in [20], Wave can extract attribute-value pairs for an unlimited number of non-predefined attributes in addition to effectively extracts attribute-value pairs from articles.

The way Kylin system extracts attribute-value pairs is similar to Wave. However, Wave improves Kylin in many aspects. First, Kylin represents the content of articles using a plain text format while Wave represents the content of articles using an *enriched plain text* format. Second, Kylin uses multiple sentences per article to generate CRF extractors, while Wave uses only one sentence per article. Third, Kylin uses a sentence based segmentation model to generate CRF extractors while Wave uses a window based segmentation model. It is important to note that all improvements proposed in the Kylin system by [30] can also be applied to Wave.

3 Extracting Attribute-Value Pairs from Wikipedia

Wave uses the content of Wikipedia articles to learn how to extract attribute-value pairs from it, building extraction models to perform the extraction. In this section we present the Wave system. Figure 1 shows the main components of the system: Wikipedia processor, classifier, filter, and extractor, describing the learning step to obtain extraction models which are then used by Wave to extract attribute-value pairs from Wikipedia.

3.1 Wikipedia Processor

The Wikipedia processor is responsible for extracting articles, infoboxes templates, and attribute-value pairs from Wikipedia corpus. The extracted elements compose a set of training data used by the other components. The Wikipedia processor has five steps:

1. Scan Wikipedia corpus and select articles associated with specific infobox templates. To be selected, an article must contain an infobox with the same name of a given infobox template name.
2. Extract attribute-value pairs from the infoboxes within the selected articles and create infoboxes schemata. An infobox schema is a set of attributes for a specific infobox template.

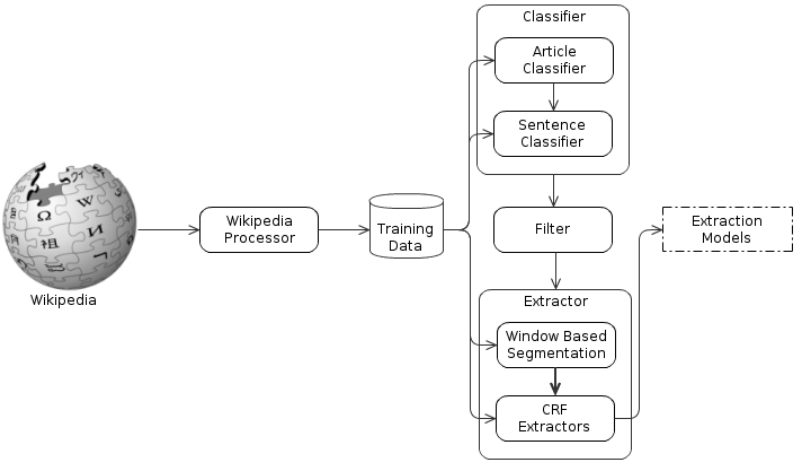


Fig. 1. The WAVE architecture

3. Convert the content of each article from Mediawiki text format to HTML format using the Bliki engine [8], a Java API for parsing and transforming texts. Mediawiki is a Wikipedia syntax used by users to edit articles.
4. Convert the content of each article from HTML format to an *enriched plain text* format. The *enriched plain text* format considers alpha-numeric and punctuation characters. It also considers HTML tags, but discards the attributes of each tag. Figure 2 shows an example of an article content in Mediawiki text format, HTML format, plain-text format and *enriched plain text* format.
5. Segment the content of articles in *enriched plain text* format into sentences using OpenNLP [22], a Java API for natural language processing.

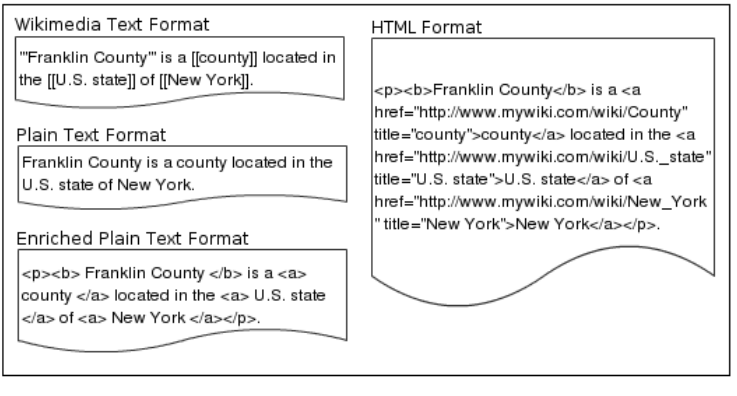


Fig. 2. Example of text formats

Notice that steps 3 and 4 are simply normalization procedures to make it more explicity the syntactic regularity on the text.

3.2 Classifier

Article Classifier. The Wikipedia processor component provides a set of categorized articles, i.e., articles associated with infoboxes templates. This data is used by the article classifier to learn how to associate new articles with infoboxes schemata. We need to associate each new article with exactly one infobox schema in order to determine which attributes should be extracted from the article content. We implement the article classifier using LIBSVM [5], a library for Support Vector Machines (SVM) [4,6], which was chosen because it obtains good prediction performance for text categorization [7]. We use infobox template names, article titles, the content of articles, Wikipedia list names, and Wikipedia category tags as features for the article classifier.

Sentence Classifier. The sentence classifier is responsible for associating article sentences with article attributes. We can divide the processing of the sentence classifier into two phases:

1. Training Phase: Data provided by the Wikipedia processor component is used to build training data for the sentence classifier. For each article, sentences are associated with attributes within the infobox schema of the article. The association is based on simple term matching. If we have terms within any value of an attribute that exactly matches with terms within any sentence, they will be associated.
2. Learning Phase: A maximum entropy classifier [21] learns how to associate sentences with attributes based on training data generated in the previous phase. We use the OpenNLP Maxent library [18] for implementation of the maximum entropy classifier. This classifier was chosen in this step because it is known as a quite competitive alternative for multi-class and multi-label classification.

When a new article arrives, it is segmented into sentences. Then, the classification model learned by the sentence classifier is applied and article sentences are associated with article attributes.

3.3 Filter

The sentence classifier is a multi-class classifier and can obtain a set of sentences associated with the same attribute. The filter component is responsible for choosing the most appropriate sentence in the set.

Considering that we have the same attribute present in several infobox schema within training data, we take all the sentences related with an attribute and group them into clusters using an efficient implementation of the k-means clustering algorithm [14]. To compute the distance between sentences we represent them in a vector space and use the similarity measure as defined by the

vector space model [2]. From the resultant clusters, we select one that presents the greater number of sentences from different infobox schema. This heuristic selection is based on the intuition that the most popular cluster tends to be the one that contains the best value (the best sentence) to be associated with the processed attribute. Then, we use the proximity of the cluster centroid to choose one sentence for each infobox schema. The sentence closer to the centroid is considered as the best option to fill the value of the attribute.

3.4 Extractor

The extractor is responsible for extracting attribute values from text segments and assigning them to attribute-value pairs.

Window Based Segmentation. There is a preprocessing procedure that must be done in an attempt to maximize CRF extractors performance. Each sentence associated with each attribute must be segmented into terms and a term sequence must be selected to compose the text segment to be processed by the CRF extractor.

In more details, for each sentence filtered by the filter component it is possible to determine, with the same term matching procedure used by the sentence classifier, the terms of the sentence that corresponds to the attribute value. We call this terms of "attribute-terms". Using a window size of x, we can extract from each sentence a term sequence composed of x terms before the attribute-terms (pre-terms), the attribute-terms, and x terms after the attribute-terms (post-terms). The extracted term sequences compose a training data for segmentation. It is important to note that the value x of the window size should be obtained empirically.

When a new sentence arrives, it is segmented into terms. Then, we use the similarity between the pre-terms and post-terms extracted from the sentence and the pre-terms and post-terms present in training data to select which terms will be used by CRF Extractor.

CRF Extractor. Extracting attribute values from a text can be viewed as a sequential data-labeling problem. Therefore, the choice of CRF for solving the problem is feasible, since CRF is the state-of-the-art for this task. Our approach trains a different CRF extractor for each attribute and uses a well-known implementation of CRF [24]. For each attribute, the term sequences associated with it by the window based segmentation procedure is used to train the extractor. We label the pre-terms with the "pre" label, the post-terms with the "post" label and each one of the terms on the attribute-terms with three different type of labels in the following way:

1. If the attribute-terms is composed by only one term, this term is labeled with "init" label.
2. If the attribute-terms is composed by only two terms, the first term of the sequence is labeled with "init" label and the last one is labeled with "end" label.

3. If the attribute-terms is composed by more than two terms, the first term of the sequence is labeled with "init" label, the last one is labeled with "end" label and each one of the other terms is labeled with "middle" label.

Each one of the CRF extractors learn a different extraction model and use it to extract values from term sequences. The extracted value is assigned to the attribute generating an attribute-value pair.

4 Experimental Results

As mentioned before, we applied WAVE to address the automatic infobox generation problem and we have chosen Kylin as the baseline. In order to evaluate the performance of WAVE and the baseline, we created four datasets from the 2010.03.12 Wikipedia corpus, one for each of the following popular Wikipedia infobox templates: *U.S. County*, *Airline*, *Actor* and *University*. Table 1 shows the distribution of the 3,610 Wikipedia articles in the four datasets. The size of a dataset is the number of Wikipedia articles within the dataset.

Table 1. Size of the datasets

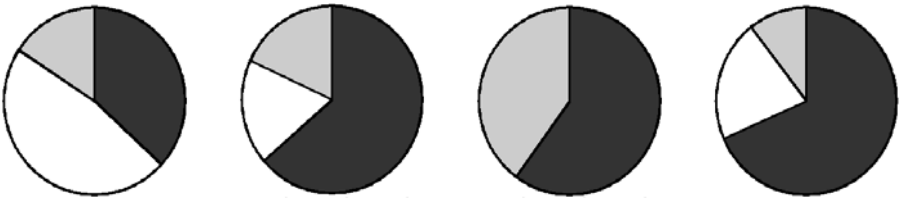
	U.S. County	Airline	Actor	University
Dataset size	1,697	456	312	1,145

Some attributes do not occur frequently in Wikipedia articles and therefore were discarded. For each dataset, we discarded all attributes not present in at least 15% of its articles. The sentence classifier component also discards, for each Wikipedia article, attributes which values do not match with any word sequence in sentences within it. Table 2 shows the number of attributes extracted from Wikipedia articles, the number of attributes discarded due to low frequency and the matching procedure of the sentence classifier, and the total number of attributes actually used in experiments for each dataset.

There are three different types of attribute in each dataset: date, number, and string. Table 2 shows that we used 54 different attributes in the experiments. The most common type of attribute is the string (55.55%), followed by number

Table 2. Number of extracted, discarded, and used attributes for each dataset

Dataset	Number of Attributes			
	Extracted	Discarded		Used
		Low frequency	No matching	
U.S. County	105	75 (71.42%)	11 (10.48%)	19 (18.10%)
Airline	60	40 (66.67%)	9 (15.00%)	11 (18.33%)
Actor	45	34 (75.55%)	6 (13.34%)	5 (11.11%)
University	283	251 (88.70%)	13 (4.59%)	19 (6.71%)



(a) U.S. County (b) Airline (c) Actor (d) University

Fig. 3. Distribution of attribute type per dataset. String is in black, date is in gray and number is in white.

(27.78%), and date attributes (16.67%). Figure 3 show the distribution of the attribute types in the datasets. We observe that only in the *U.S. County* dataset the string type is not majority. In this case, the number of numerical attributes is greater than the number of attributes of the string type.

To perform the experiments, we used the 10-fold cross validation method [27]. Each dataset was randomly split in ten parts, such that, in each run, a different part was used as a training set while the remaining were used as a test set. The split on training and test sets was the same in all experiments. The final results of each experiment represent the average of the ten runs.

The performance of WAVE and the baseline was evaluated using the conventional precision, recall and F1 measures. Precision p is defined as the proportion of correctly extracted attribute-value pairs in the set of all extracted attribute-value pairs. Recall r is defined as the proportion of correctly extracted attribute-value pairs in all of the correctly attribute-value pairs in examples. $F1$ is a combination of precision and recall defined as $2pr/(p + r)$.

We performed preliminary experiments in order to empirically obtain the value of the window size x used by the window based segmentation procedure of the WAVE extractor component. We use $x = 3$ for all experiments.

As mentioned before, WAVE trains a different CRF extractor for each attribute. We perform experiments using the 10-fold cross validation method and, for each attribute, we compute the average precision, average recall, and average F1.

Table 3 shows the overall performance for WAVE and the baseline, for each attribute type. The values presented correspond to the mean average precision, mean average recall and mean average F1 for the group of attributes of each type.

We observe that the difference in performance between WAVE and the baseline is greater in attributes of the string type. When observing the results we realized that attributes of the string type take more advantage of the textual content enrichment made by WAVE. Remember that WAVE enriches the textual content of the articles with HTML structural information, making word patterns more regular, which have increased the performance of the CRF extractors. We observe that the greater the regularity in the word patterns of the HTML tags around the attribute value to be extracted within sentences, the better the performance of the CRF extractor. The gain for numerical numbers are a bit smaller, but can

Table 3. Mean average precision, recall and F1 results achieved by WAVE and the baseline for different attribute type

Attribute Type	Precision (%)			Recall (%)			F1 (%)		
	Baseline	WAVE	Gain	Baseline	WAVE	Gain	Baseline	WAVE	Gain
Date	57.46	63.33	+10.20	52.34	57.34	+9.56	54.67	60.04	+9.82
Number	89.19	92.34	+3.53	88.16	93.04	+5.54	88.58	92.64	+4.58
String	62.58	74.37	+18.85	55.30	67.05	+21.25	58.40	70.02	+19.91

Table 4. Mean average precision, recall and F1 results achieved by WAVE and the baseline for each dataset

Dataset	Precision (%)			Recall (%)			F1 (%)		
	Baseline	WAVE	Gain	Baseline	WAVE	Gain	Baseline	WAVE	Gain
U.S. County	87.73	93.85	+6.97	86.40	93.92	+8.71	87.01	93.87	+7.88
Airline	55.73	67.95	+21.92	49.75	63.11	+26.86	52.40	65.13	+24.29
Actor	65.82	75.31	+14.42	60.57	68.73	+13.48	63.09	71.60	+13.53
University	63.87	71.59	+12.08	56.12	63.33	+12.83	59.27	66.59	+12.34

be also considered as quite important, since the results of the baseline in this case were already quite high, with almost no space for improvements.

Table 4 shows the overall performance for WAVE and the baseline for the datasets. The values presented correspond to the mean average precision, mean average recall and mean average F1 for the group of attributes within each dataset.

We observe that all WAVE results outperform the baseline values. As expected, the gain was greater in datasets with more attributes of the type string, followed by type date. This was expected, since the baseline already presents quite high quality results for numerical attributes. Again, the type string take more advantage of the textual content enrichment made by WAVE.

5 Conclusions

In this paper we have presented WAVE, a self-supervised approach to autonomously extract attribute-value pairs from Wikipedia articles. The extracted attribute-value pairs can be used in many information retrieval tasks, such as question answering, query expansion, multilingual information retrieval, and text categorization.

We applied WAVE to address the automatic infobox generation problem and we have shown that WAVE outperforms the baseline by 21.92% in precision, 26.86% in recall and 24.29% in F1. We have also shown that the greater the regularity in the word patterns of the HTML tags around the attribute value to be extracted within sentences, the better the performance of WAVE.

Future work will concentrate on improving performance of the classifier component by exploiting other Wikipedia features. We intend to analyze the impact

of window size in CRF extractors and also exploit other features, incorporate improvements described by [30], expand the types of values from date, number and string to multivalued slots such as locations and person names, and measure the improvements arising from the application of our approach in some information retrieval tasks.

Acknowledgements

We thank the partial support given by the Brazilian National Institute of Science and Technology for the Web (grant MCT/CNPq 573871/2008-6), Project InfoWeb (grant MCT/CNPq/CT-INFO 550874/2007-0), Project MinGroup (grant CNPq/CT-Amazônia 575553/2008-1), CNPq Grant 30.2209/2007-7 (Edleno S. Moura), CNPq Grant 30.8528/2007-7 (Altigran S. Silva), and CNPq Grant 30.5237/02-0 (Nivio Ziviani).

References

1. Auer, S., Lehmann, J.: What have innsbruck and leipzig in common? extracting semantics from wiki content. In: Proceedings of the 4th European Conference on The Semantic Web, pp. 503–517 (2007)
2. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, Reading (1999)
3. Banerjee, S., Ramanathan, K., Gupta, A.: Clustering short texts using wikipedia. In: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 787–788 (2007)
4. Boser, B.E., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Proceedings of the 5th Annual Workshop on Computational Learning Theory, pp. 144–152 (1992)
5. Chang, C., Lin, C.: Libsvm: a library for support vector machines (2001), http://www.csie.ntu.edu.tw/~cjlin/libsvm/
6. Cortes, C., Vapnik, V.: Support-vector network. Machine Learning 20, 273–297 (1995)
7. Dumais, S., Platt, J., Heckerman, D., Sahami, M.: Inductive learning algorithms and representations for text categorization. In: Proceedings of the 7th International Conference on Information and Knowledge Management, pp. 148–155 (1998)
8. Bliki Engine, http://code.google.com/p/gwtwiki/
9. Google Search Engine, http://www.google.com/
10. Etzioni, O., Banko, M., Soderland, S., Weld, D.S.: Open information extraction from the web. Communications of the ACM 51(12), 68–74 (2008)
11. Hahn, R., Bizer, C., Sahnwaldt, C., Herta, C., Robinson, S., Bürgle, M., Düwiger, H., Scheel, U.: Faceted wikipedia search. In: Wecel, K. (ed.) BIS 2010. LNBIP, vol. 47, pp. 1–11. Springer, Heidelberg (2010)
12. Higashinaka, R., Dohsaka, K., Isozaki, H.: Learning to rank definitions to generate quizzes for interactive information presentation. In: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, pp. 117–120 (2007)
13. Kaisser, M.: The qualim question answering demo: Supplementing answers with paragraphs drawn from wikipedia. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics, pp. 32–35 (2008)

14. Kanungo, T., Mount, D.M., Netanyahu, N.S., Piatko, C.D., Silverman, R., Wu, A.Y.: An efficient k-means clustering algorithm: Analysis and implementation. IEEE Transactions on Pattern Analysis and Machine Intelligence 24(7), 881–892 (2002)
15. Lafferty, J., McCallum, A., Pereira, F.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the 18th International Conference on Machine Learning, pp. 282–289 (2001)
16. WordNet: A lexical database for English, http://wordnet.princeton.edu/.
17. Li, Y., Luk, W.P.R., Ho, K.S.E., Chung, F.L.K.: Improving weak ad-hoc queries using wikipedia as external corpus. In: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 797–798 (2007)
18. OpenNLP Maxent Library, http://maxent.sourceforge.net/
19. Milne, D.N., Witten, I.H., Nichols, D.M.: A knowledge-based search engine powered by wikipedia. In: Proceedings of the 16th ACM Conference on Information and Knowledge Management, pp. 445–454 (2007)
20. Nguyen, D.P., Matsuo, Y., Ishizuka, M.: Exploiting syntatic and semantic information for relation extraction from wikipedia. In: Proceedings of the Workshop on Text-Mining & Link-Analysis (2007)
21. Nigam, K., Lafferty, J., Mccallum, A.: Using maximum entropy for text classification. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence, pp. 61–67 (1999)
22. OpenNLP, http://opennlp.sourceforge.net/
23. Potthast, M., Stein, B., Anderka, M.: Wikipedia-based multilingual retrieval model. In: Proceedings of the 30th European Conference on Information Retrieval Research, pp. 522–530 (2008)
24. CRF Project, http://crf.sourceforge.net/
25. Resource Description Framework (RDF), http://www.w3.org/RDF/
26. Sha, F., Pereira, F.: Shallow parsing with conditional random fields. In: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, pp. 134–141 (2003)
27. Stone, M.: Cross-validation choices and assessment of statistical predictions. Journal of the Royal Statistical Society B36, 111–147 (1974)
28. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A core of semantic knowledge. In: Proceedings of the 16th International Conference on World Wide Web, pp. 697–706 (2007)
29. Wang, P., Hu, J., Zeng, H., Chen, L., Chen, Z.: Improving text classification by using encyclopedia knowledge. In: Proceedings of the 7th IEEE International Conference on Data Mining, pp. 332–341 (2007)
30. Wu, F., Hoffmann, R., Weld, D.S.: Information extraction from wikipedia: Moving down the long tail. In: Proceeding of the 14th ACM International Conference on Knowledge Discovery and Data Mining, pp. 731–739 (2008)
31. Wu, F., Weld, D.S.: Autonomously semantifying wikipedia. In: Proceedings of the 16th ACM Conference on Information and Knowledge Management, pp. 41–50 (2007)

Temporal Analysis of Document Collections: Framework and Applications

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

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

Abstract. As the amount of generated information increases so rapidly in the digital world, the concept of time as a dimension along which information can be organized and explored becomes more and more important. In this paper, we present a temporal document analysis framework for document collections in support of diverse information retrieval and seeking tasks. Our analysis is not based on document creation and/or modification timestamps but on extracting time from the content itself. We also briefly sketch some scenarios and experiments for analyzing documents from a temporal perspective.

1 Time in Information Retrieval

Time is an important dimension of any information space, and it can be very useful in information retrieval tasks [4]. Little work has been done on analyzing and exploiting temporal information embedded in documents as *relevance cues* for the presentation, organization, and in particular the exploration of search results in a temporal context.

In this paper, we outline such a framework that realizes an add-on to search engines in support of time-centric document exploration, visualization, and similarity search tasks. Using an information extraction approach, we show how temporal information embedded in the text of documents is extracted and made explicit for deriving novel *temporal document measures*, such as temporal specificity, richness, and boundaries. Using such base measures, we introduce a histogram based approach, called *temporal coverage*, to compute and visualize what time periods and points in time are mentioned in a document.

A further important contribution of our framework is a *temporal document distance measure*, which is built on the concept of temporal coverage. This functionality is in support of determining documents that are temporally similar to a given document in terms of the time information embedded in the documents. Although our focus is primarily on *topic specific document collections*, some concepts easily translate to Web search scenarios. We also do not focus on topic detection and tracking using temporal information but rather establish a framework that can be used to target these areas as well.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 290–296, 2010.

2 Time Annotated Document Model

As the basis for anchoring documents in time, we assume a discrete representation of time based on the Gregorian Calendar, with a single day being an atomic time interval called *chronon*. Our *base timeline*, denoted T_d, is an interval of consecutive day chronons. We assume the five timelines $\mathcal{T} = \{T_t, T_d, T_w, T_m, T_y\}$ for times (of the day), days, weeks, months, and years, respectively. The composition of granules naturally induces a lattice structure in $\mathcal{T}$. That is, we have the relationship $T_j \gg T_i$ if timeline T_j is (transitively) composed of chronons of timeline T_i. Associated with each timeline $T \in \mathcal{T}$ is a *precedence relationship* $\prec_T$ that allows to compare chronons.

A key aspect of our approach is to take a set of documents $\mathcal{D}$, extract all types of temporal information associated with the documents in $\mathcal{D}$, and use this information for computing temporal document measures. The first type of such information is the *document timestamp*, which appears as the date a document $d \in \mathcal{D}$ has been created and be anchored in the timeline T_d (or T_t).

The second type of temporal information corresponds to concepts that are represented in the document text as sequences of tokens or words, called *temporal expressions*. Similar to the approach by Schilder and Habel [5], we distinguish between explicit, implicit, and relative temporal expressions.

Explicit temporal expressions describe chronons in some timeline, such as an exact time (of a day), day or year. For example, "December 2004" is an explicit expression that is anchored in the timeline T_m. Depending on the capabilities of the entity extraction approach, *implicit temporal expression*, such as names of holidays or events can be anchored in a timeline as well. For example, the token sequence "Columbus Day 2006" in the text of a document can be mapped to the expression "October 12, 2006", which is anchored in T_d. *Relative temporal expressions* represent temporal entities that can only be anchored in a timeline in reference to another explicit or implicit, already anchored temporal expression.

In our time annotated document model, for a document $d \in \mathcal{D}$, the entity extraction applied to d results in a *temporal document profile*, denoted $tdp(d)$. A temporal document profile is a sequence of *chronon/position* pairs $\langle c_i, p_i \rangle$. A chronon c_i is an element from the timelines $\mathcal{T} = \{T_t, T_d, T_w, T_m, T_y\}$, and a position p_i is the absolute document position of the (first token in the) temporal expression that has been mapped to c_i. We assume that all chronons c_i in a temporal document profiles are *normalized*.

3 Temporal Document Measures

Temporal document profiles provide the basis for deriving measures to analyze and better describe documents in terms of their temporal information content. In the following, we assume a single document $d \in \mathcal{D}$ with temporal document profile $tdp(d)$. Let t-seq$(d) = \langle (c_1, p_1), (c_2, p_2), \cdots, (c_k, p_k) \rangle$ denote the chronon/position pairs of the profile. The first measure simply defines the ratio of chronons in the document d to the total number of chronons in the collection D. We call this measure *temporal richness* of d, denoted t-rich(d). The higher this ratio, the richer the document d is in its temporal information content.

The next measure focuses more on the types of chronons in t-seq(d), in particular the frequencies of types. A document d is said to be *temporally most specific* with respect to chronons of type $T \in \mathcal{T}$, if most of the chronons in t-seq(d) are of type T. For example, a document detailing the events of a soccer game might be very specific, using chronons of type time (T_t) whereas a resume document might be more specific with respect to chronons of type year or month (T_y or T_m). In the following, we denote the *temporal specificity* of a document d as t-spec(d). As this measure is simply based on the frequencies of chronons of different types in t-seq(d), it naturally defines an order among the types in $\mathcal{T}$ for a document d, with the first type in that order being the most frequent chronon type in t-seq(d) and so on.

The last aggregate measure considered here describes the *temporal boundaries* of a document. For this, the earliest and the most recent chronons in t-seq(d) are determined. The problem here is that chronons can be of different types and thus the precedence relationship $\prec_T$ might not always be applicable. We therefore make the following assumption: If two chronons $t_i \in T_i$ and $t_j \in T_j$ from t-seq(d) can each be considered the earliest (most recent) chronon, then the one with the coarser granularity is chosen as the earliest (most recent). In the following, we denote the earliest and most recent chronons in t-seq(d) as t-low(d) and t-high(d), respectively. As an example, for Einstein's Wikipedia page, we have t-low = "3/14/1879" (his date of birth), and t-high = "2008".

We use an equiwidth histogram approach to describe the distribution of chronons in a document over a period of time, called *temporal coverage*. The objective is to represent (a) what periods in time are covered by the document's content and (b) what periods in time are emphasized in terms of chronon frequencies. Loosely speaking, for a given document d, the period of time (range) described by its histogram is determined by the temporal boundaries t-low(d) and t-high(d), and the number of values in a bucket is based on the number of chronons that fall in that bucket's subrange (period in time).

First, a range for the histogram's buckets needs to be determined. We choose as temporal unit the type of the chronon in t-seq(d) with the coarsest granularity, even though the temporal specificity of the document might be of a finer granularity. Consider, for example, a document d with t-low(d) = "1950" and t-high(d) = "2003". The histogram for d, denoted t-hist(d) then is made up of 54 buckets, each labeled by a year chronon in $[1950, 2003]$. In the worst case, there might only be one bucket in t-hist(d), namely when all chronons refer to the same year but there is at least one chronon $c \in$ t-seq(d) of type year.

With each bucket in a histogram two important measures are associated. First, for each bucket $b_i, i \in [1, \ell]$, the relative frequency of the chronons placed in b_i, denoted t-freq(b_i), is maintained. The relative frequency for b_i is the ratio between the number of chronons in b_i to the total number of chronons on all the buckets for the histogram. The domain of the chronons' relative frequency is the interval $[0, 1]$, and the histogram is an approximation of the probability density function representing the underlying distribution of the chronons. Second, as for the whole document d, also with each bucket b_i a temporal specificity, denoted t-spec(b_i), can be determined.

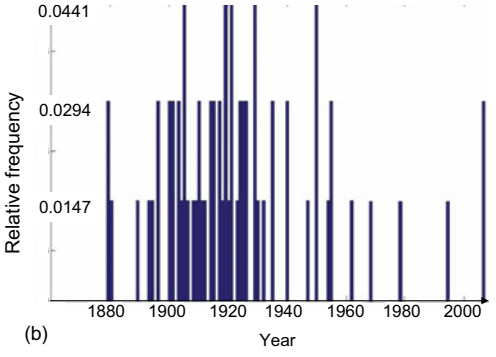


Fig. 1. Temporal coverage in the form of a histogram (Einstein's Wikipedia page)

Figure 1 illustrates an example of the temporal coverage for Einstein's Wikipedia page. Note the temporal "focus" of the document for the time period between 1900 and 1930, something that provides some important information about the temporal content of the document. Note that although the label of a bucket b_i is based on a coarse-grained type T, the bucket might contain several chronons that are of a granularity T' finer than T, i.e., $T' \ll T$. To obtain more fine-grained temporal information from such buckets, a *temporal zoom operation* can be applied as follows. Assume a bucket b_i labeled with a year chronon y. A zoom-in operation partitions this bucket into twelve buckets, each labeled with a month of the year y.

The chronons from b_i are then distributed over these twelve new buckets, based on whether the a new bucket's label covers that chronon. With each new bucket, again a frequency and temporal specificity is associated. As b_i might contain chronons of granularity T, not all chronons can be distributed over the new buckets. This zoom-in operation can be applied in an iterative fashion, exploring individual buckets at an increasingly fine level of time granularity.

4 Temporal Document Similarity

An important feature of our proposed temporal document analysis and exploration framework is to allow users to search for "temporally similar" documents in a hit list or document collection. But what does it mean that two documents are temporally similar? While this question itself warrants an extensive study that goes far beyond the scope of this paper, we propose a simple and easy way to compute the *temporal distance* between two documents.

Assume a hit list L of documents as result to a user query and the user considers a document $d \in L$ as interesting, because d covers a particular period in time in detail (this observation alone can be made from the visual representation of the document's histogram). If the user is now interested in documents from L that are temporally similar to d, she can either look at each document's histogram or simply invoke a distance function that is realized as follows.

The temporal distance is defined as the difference between the distributions of the chronons of two documents. Since we use a histogram as the estimation of the underlying distribution of the chronons, we employ a distance measure for ordinal type histograms described in [3]. We modify this approach to take the characteristics of our type of histograms into account. Assume two documents d and d' with histograms t-hist$(d) = b_0, b_1, \ldots, b_d$ and t-hist$(d') = a_0, a_1, \ldots, a_{d'}$, respectively. In order to apply the technique, the two histograms first need to be normalized in terms of underlying chronon type and temporal boundaries. For this, the histogram based on the coarser type of chronon is chosen and the buckets of the other histogram are aggregated correspondingly. For example, t-hist(d') might be based on month chronons while t-hist(d) is based on year chronons. The buckets in t-hist(d') are then aggregated and new buckets, now labeled with year chronons are created. Next, as the temporal boundaries of the two histograms may differ, they need to be brought to the same length. This is easily achieved by extending the histograms to the left and right, as necessary, with empty buckets so that both histograms have the same t-low and t-high values (c.f. Sect. 3). Assume this normalization results in the two modified histograms t-hist$(d) = b_0, b_1, \ldots, b_n$ and t-hist$(d) = a_0, a_1, \ldots, a_n$ that now have the same lengths n, temporal boundaries, and underlying chronon type. After these two histogram normalization steps, the distance function

$$dist(d, d') = \sum_{i=0}^{n-1} \left| \sum_{j=0}^{i} (\text{t-freq}_n(b_j) - \text{t-freq}_n(a_j)) \right| \quad (1)$$

derived from [3] is applied to the two documents d and d' and associated (modified) histograms. This non-metric distance function basically determines the number of re-arrangements of buckets in one histogram so that the two histograms are trivially identical (recall that because of the normalization, both histograms have the same "number" of elements). The closer the computed value to 0, the more *temporally similar* the documents are.

It should be noted that standard distance functions such as cosine-based similarity are not applicable here (e.g., when histograms are represented as vectors) as they do not consider the shape but only focus on frequencies. Given a document $d \in L$, the distance to all other documents in L is computed and, based on the computed values, the remaining documents in L can in fact also be sorted, presenting the user the temporally most similar documents first.

5 Applications

In the following, we sketch some applications based on the temporal framework described in the previous sections. The first item is to have a temporal annotation infrastructure that can handle different collections. In our case, we rely on the Alembic[1] POS tagger and the GUTime[2] temporal expression tagger to extract

[1] http://www.mitre.org/tech/alembic-workbench/
[2] http://complingone.georgetown.edu/~linguist/

temporal expressions based on the TimeML standard[3]. Having all temporal information embedded in document available in the form of temporal document profiles now gives rise to several document search and exploration scenarios. Here we outline only a few due to space constraints.

First, it is important to note that the temporal content in documents varies form collection to collection. We investigated this aspect using a subset of TREC (FBIS documents) and Wikipedia (featured articles). As shown in Figure 2, the Wikipedia collection is in general much richer in terms of temporal expressions and thus are suitable for temporal exploration applications.

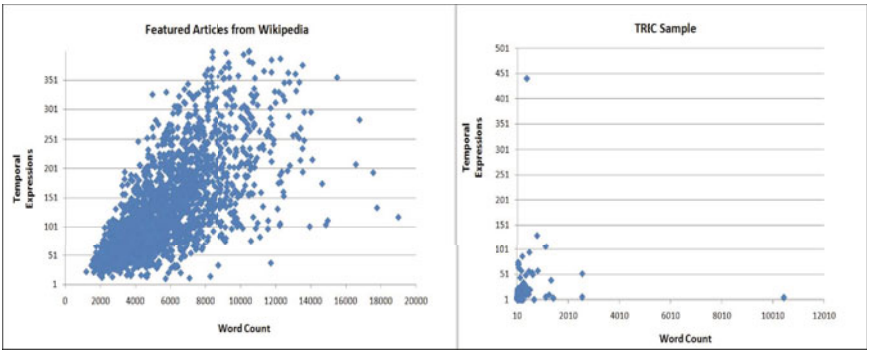


Fig. 2. Distribution of temporal expressions in Wikipedia and TREC samples

We conducted a user study among 30 persons (graduate students and faculty) about our temporal framework. Part of the study consisted of a survey about temporal information in search engines. Of all the respondents, 70% answered that current search engines do not return temporal information and 30% did not know for sure. The other finding is that 65% think that a search engine should present temporal information in a hit list or as a feature (10% answered "No" and 25% "Do not know").

We also performed experiments using the Amazon Mechanical Turk[4] crowdsourcing platform. We ran the same survey about temporal information in search engines. Of all the respondents (20 in total), 80% answered that current search engines do not return temporal information and 20% did not know for sure. Of all respondents 75% think that a search engine should present temporal information in the search results. In summary, for both studies users are interested in using search applications that offer time-aware features.

Typical applications using the temporal information, which we have realized in a prototype, are as follows. Assume one queries a collection for [Einstein], like in a conventional search engine. Using our approach, a hit list is returned that contains the usual document information but also includes information about the temporal coverage. The latter is visualized using sparklines that show the

[3] http://timeml.org
[4] http://www.mturk.com/

temporal coverage as well as the temporal boundaries for each document (see Figure 3). Assume the user is specifically interested in the time period 1905–1920. Using the documents' histogram information, respective documents can be determined and presented to the user.

Based on a document from the hit list, the user can also request documents that are "temporally to this". The application then returns all pages that are temporally similar to the selected one, ranked by the distance to the selected document. Another interesting functionality we realized is to construct *temporal snippets* (Figure 4) for a selected document, which are fragments of the document's text that display two or three sentences with the most relevant chronons [2]. Based on temporal document profile, also recently a method has been presented and integrated into our prototype that enables the temporal clustering of documents in search and exploration tasks [1].

Fig. 3. Enhanced user interface showing temporal information as sparklines

Hey_Jude - Wikipedia
"**Hey Jude**" is a song by the English rock band The Beatles that was recorded in 1968 ... "**Hey Jude**" was released in August 1968 as the first single from The Beatles' record label Apple Records ... "**Hey Jude**" was the top Billboard Hot 100 single for 1968, according to year-end charts ... In 2001, "**Hey Jude**" was inducted into the National Academy of Recording Arts and Sciences Grammy Hall of Fame ...
http://en.wikipedia.org/wiki/Hey_Jude

Fig. 4. Temporal snippet for the query [hey jude]

In summary, one can claim that temporal information embedded in documents in the form of temporal expressions offer an interesting means to further enhance the functionality of current information retrieval applications. In this paper, we have outlined a framework that can be used to make a search application temporally aware by providing more document search and exploration features that leverage time.

References

1. Alonso, O., Gertz, M., Baeza-Yates, R.: Clustering and Exploring Search Results using Timeline Constructions. In: 18th ACM Conf. on Information and Knowledge Management, pp. 97–106 (2009)
2. Alonso, O., Baeza-Yates, R., Gertz, M.: Effectiveness of Temporal Snippets. In: WSSP Workshop, WWW Madrid (2009)
3. Cha, S.-H., Srihari, S.N.: On measuring the distance between histograms. Pattern Recognition (35), 1355–1370 (2002)
4. Mani, I., Pustejovsky, J., Gaizauskas, R. (eds.): The Language of Time. Oxford University Press, Oxford (2005)
5. Schilder, F., Habel, C.: From Temporal Expressions to Temporal Information: Semantic Tagging of News Messages. In: ACL 2001 Workshop on Temporal and Spatial Information Processing, pp. 1–8 (2001)

Text Comparison Using Soft Cardinality

Sergio Jimenez[1], Fabio Gonzalez[1], and Alexander Gelbukh[2]

[1] National University of Colombia
[2] CIC-IPN Mexico
{sgjimenezv,fagonzalezo}@unal.edu.co
www.gelbukh.com

Abstract. The classical set theory provides a method for comparing objects using cardinality and intersection, in combination with well-known resemblance coefficients such as Dice, Jaccard, and cosine. However, set operations are intrinsically crisp: they do not take into account similarities between elements. We propose a new general-purpose method for comparison of objects using a soft cardinality function that show that the soft cardinality method is superior via an auxiliary affinity (similarity) measure. Our experiments with 12 text matching datasets suggest that the soft cardinality method is superior to known approximate string comparison methods in text comparison task.

1 Introduction

Things are usually not either completely equal or completely different. More often than not we need to decide which objects are more similar than others. For example, in information retrieval task we need to find documents most similar to the user query, while none of these documents is exactly equal to it. While the task of exact comparison is well-defined and the corresponding methods are clear and well understood, approximate comparison is a highly heuristic task for which a great variety of methods have been suggested, each one good for some problems and none good for all—which means that the quest for better and more general approximate comparison paradigms is in full swing.

In this paper we propose a new approximate object comparison method based on soft cardinality. While stemming from fundamental elements from the classic set theory, it uses a new cardinality function that allows for flexibility. Despite the simplicity of our method, preliminary results obtained in a text matching task are encouraging as compared with much more elaborated state-of-the-art approximate string matching techniques.

The paper is organized as follows. Section 2 briefly describes some most closely related approaches and compares our method with them. Section 3 presents the notion of soft cardinality and introduces our method. Section 4 gives the experimental results. Finally, Section 5 concludes the paper.

2 Related Work

Binary similarity measures use different strategies to assess commonalities and differences of objects under comparison. Probably the most popular approximate

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 297–302, 2010.

comparison technique uses the well-known resemblance coefficients [1]. Although resemblance coefficients reflect the degree of similarity, they are based on crisp operations such as set intersection, which do not consider degree of similarity between the elements of the sets. This is a drawback in scenarios with uncertainty. Our proposed method addresses the problem of the inability to manage uncertainty by resemblance coefficients: instead, it exploits the redundancy in sets.

Unlike fuzzy sets [2], our approach does not consider uncertainty in the membership of a particular element. It, however, does handle uncertainty as to the contribution of an element to the cardinality of the set if the element presents redundancy with respect to other elements of the same set.

Classic set theory provides an intuitive mechanism for object comparison using set operations such as intersection $\cap$ and union $\cup$, as well as a function $|*| \in \mathbb{N}$ called cardinality for counting the number of different elements in a set. Using only those components, it is possible to compute similarity between two sets A and B using a family of resemblance coefficients [1] such as:

$$Jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|}, \quad cosine(A,B) = \frac{|A \cap B|}{\sqrt{|A| \times |B|}}, \tag{1}$$

as well as a host of other resemblance coefficients widely used in biosciences, economics, social sciences, engineering and computer science, among other fields. All those expressions return 1 if A and B are equal, 0 if A and B have no elements in common, and otherwise a number between 0 and 1.

3 Soft Cardinality

The classic set cardinality function counts the elements in a set in a crisp manner: if an element is duplicated, it is only considered once. However, if two elements in a set are very similar—nearly duplicates but not exactly—our intuition is that together they should contribute less to the total cardinality than a pair of completely different elements.

Consider the set S of animals in Fig. 1(a). The classic crisp cardinality function reports 3 different animals in S. Nevertheless, if an affinity function between the elements of the set is used, Fig. 1(b) shows a better view of the situation. A *soft cardinality* function that reflects this intuition should produce a value less than three but greater than two. Even though the elements of the set shown in Fig. 1 are not sets themselves, the affinity between *tiger* and *lion* induces some type of intersection in terms of the total soft cardinality of the set.

We define a binary affinity function $\alpha(*,*)$ that reflects the similarity between two elements a and b in the set. Obviously,

$$\begin{aligned} &\alpha(a,b) \in [0,1]; \quad &&\alpha(a,b) = \alpha(b,a); \quad &&\alpha(a,a) = 1; \\ &\delta(a,b) = 1 - \alpha(a,b); \quad &&\delta(a,c) \leq \delta(a,b) + \delta(b,c), \end{aligned} \tag{2}$$

where δ is the distance.

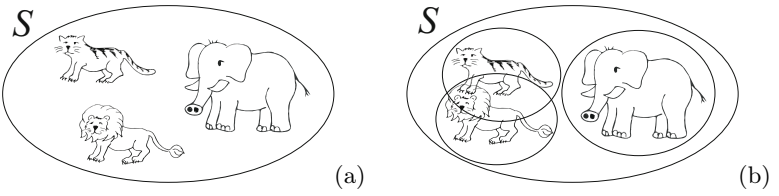


Fig. 1. A set with three elements

Consider Fig. 1(*b*). If the elements of S' are treated as sets, the affinity function α can be considered as an estimation of the intersection between the elements in the set. Assume an affinity value $\alpha(tiger, lion) = 0.7$, the total soft cardinality of the set can be defined as the crisp cardinality $|S| = 3$ minus the overlap between *tiger* and *lion* (which is 0.7), which gives $|S'|_\alpha = 2.3$.

3.1 Estimating Cardinality of Set Union Using Pairwise Intersections

The affinity binary function α has been proposed as an approximation of the cardinality of the intersection of two elements in a set. Although a and b are not sets themselves (they are just elements), the previous assumption allows us to treat them as sets. If we want to calculate the soft cardinality of a set S with only two elements a and b, it can be calculated using (2) properties and recalling $|A \cup B| = |A| + |B| - |A \cap B|$:

$$|S|_\alpha = \alpha(a, a) + \alpha(b, b) - \alpha(a, b) = 2 - \alpha(a, b),$$

where $|S|_\alpha$ stands for the soft cardinality of the set S given the affinity function α. Similarly, the case of a set S with three elements a, b, c can be treated using the following classic set theory expression:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |A \cap C| + |A \cap B \cap C|\,.$$

However, to estimate $|A \cap B \cap C|$, a ternary affinity function $\alpha(*, *, *)$ is needed. In the general case, the soft cardinality of a set S with elements $s_1, s_2, \ldots, s_n$ requires k-ary affinity functions, $k = 1, \ldots, n$. Nevertheless, the most common affinity (similarity) and distance functions are binary, making the construction of such high-arity functions a problem. We propose to estimate $|S|_\alpha$ using only a binary α function.

Consider a matrix $\mathbf{A}$ of pairwise affinity $\alpha(s_i, s_j)$ between the elements of S. It is symmetric and has all 1's in the diagonal. We will construct a function of this matrix that gives n if $\mathbf{A} = \mathbf{I}_n$ ($n \times n$ identity matrix) and 1 if all entries of $\mathbf{A}$ are 1's. The former case corresponds to a set S in which all its elements are completely different (i.e., classical crisp cardinality), while the later case corresponds to a set S in which all elements are identical. On any other symmetric matrix the function will give a real number in the interval $(1, n)$ that approximates the cardinality of $\bigcup_{i=1}^{n} s_i$.

Consider the following expression that satisfies the previous requirements of reproducibility of the classic crisp cardinality:

$$|S|_\alpha = \sum_{i=1}^{n} \frac{1}{\sum_{j=1}^{n} \alpha(s_i, s_j)^p}, \quad p \geq 0 \tag{3}$$

where p is an adjustable parameter. Using $p = 1$ in this expression with our example, a result of $|S|_\alpha = 2.18$ is obtained. This result is close to the value 2.3 obtained previously out of intuitive considerations.

3.2 Decoupling and Combining Affinity and Importance

Until now, we have assumed an inherent individual cardinality of 1 for each element. This makes sense given that the classic crisp cardinality increases by 1 with each different element in a set. This is equivalent to considering elements as sets represented as circles of area equal to 1 in a Venn diagram with among them. However, often it is reasonable to consider elements represented by circles with different areas. For instance, if the elements of the set are words, their discrimination power can be encoded by the area of the circles: stop words coded by small circles and rare words by large circles. This approach allows us to treat separately (i) the affinity between elements and (ii) the intrinsic importance or weight of the element.

In order to integrate importance or "element cardinality" to the soft cardinality method, consider the role of the inner term $(\sum_{j=1} \alpha(s_i, s_j)^p)^{-1}$ in (3). This term represents the contribution of the element s_i to the total soft cardinality of S and thus can be weighted with a factor w_i to reflect the importance ("cardinality") of the element s_i:

$$|S|_\alpha^p = \sum_{i=1}^{n} \frac{w_i}{\sum_{j=1}^{n} \alpha(s_i, s_j)^p} \tag{4}$$

Vector space model represents documents as vectors whose coordinates are dictionary words. Generally, the values of the coordinates are weights associated with the discrimination power of the words such as *tf-idf* [3]: the importance of the word for the document. However, this model assumes independence between index terms (i.e. dimensions), making it impossible to take into account their relatedness. Unlike vector space model, (4) provides the soft cardinality method a mechanism to keep affinity (correlation) and importance (weight) decoupled but naturally combined.

4 Preliminary Results

The aim of our experiments is to explore the utility and potential of the soft cardinality method in a particular text comparison task. The name-matching task consists in comparing two strings and to decide if the strings refer to the same entity or not. For instance, "King Rail Rallus elegans" and "Rail: King (Rale élégant) Rallus elegans" in the *Birds-Scott2* dataset refer to the same bird. All

possible string pairs are compared with a similarity measure and a threshold θ selects matched pairs.

Cohen *et al.* [4] carried out a comparative study of several similarity measures with twelve name matching data sets.They showed that their SoftTF-IDF measure, on average, outperforms all other measures. We compared our soft cardinality method using the same data sets that they used.

4.1 Experimental Setup

The matching problem between two set of strings can be viewed as a classification problem over the Cartesian product of the sets. As performance measure for each experiment (i.e. pairs dataset-similarity measure) we used *interpolated average precision* (IAP) and F_1-score, which are commonly used in information retrieval [5].

We carried out experiments with more than 148 well-known string similarity measures, see [6] for details. For reference purposes, the results for the best performing character level measure (i.e. bigrams) and the best token level measure (i.e. cosine) are reported. The reported similarity measures are described as follow:

bigrams. Bigrams intersection computed as the quotient of the common bigrams between the two strings and the number of bigrams in the longer string.

cosine. Both strings are tokenized (separated into words) and compared using cosine coefficient.

SC1. Both strings are tokenized. Soft cardinality (3) with $p = 1$ is used to compute the cardinality of each set of tokens and the cardinality of the union of both set of tokens. Cardinalities are combined with cosine coefficient (1). The used auxiliary inter-token affinity function α was bigrams.

SC2. It is the same method as SC1, but using $p = 2$ in (3).

SC.3 It is the same method as SC1, but using $p = 2$ and $w_i = idf_i$ in (4).

STI. SoftTF-IDF proposed by Cohen *et al.* [4] SoftTF-IDF is a fuzzified extension to the well-known *tf-idf* vector space model metric.

4.2 Results

Similarly to Cohen *et al.*, we also noticed that STI (SoftTF-IDF) outperforms on average practically all known text similarity measures. As shown in Table 1, the proposed soft cardinality methods SC2 and SC3 slightly outperformed STI, and SC1 reached a close performance. Both average IAP and F_1-score, reported SC3 and SC2 as the best performing measures for the data sets. Note that unlike STI, SC1 is a basic soft cardinality measure without any parameters to be adjusted. Moreover, SC1 is a static measure, that is, it only uses information in the pair of strings being compared. The SC2 measure is also static, with $p = 2$, see (3), reach the same performance of a STI, which also uses the entire corpus as input. In addition, SC3 performs better than STI using the same *idf* coefficients.

Table 1. IAP results for experiments

Data set	bigrams	cosine	SC1	SC2	SC3	STI
Birds-Scott1	0.848	**0.890**	0.877	0.883	0.883	0.879
Birds-Scott2	0.789	**0.907**	0.895	0.905	0.905	0.897
Birds-Kunkel	0.512	0.857	0.723	0.837	0.868	**0.875**
Birds-Nybird	0.715	0.723	0.734	**0.745**	0.740	0.743
Business	0.529	0.533	0.685	0.732	**0.776**	0.704
Game-Demos	0.722	0.754	0.776	0.784	**0.811**	0.798
Parks	0.881	0.846	0.868	0.859	0.890	**0.897**
Restaurants	0.860	**0.906**	0.899	**0.906**	0.903	0.904
UCD-people	0.797	**0.909**	0.889	**0.909**	**0.909**	**0.909**
Animals	0.066	0.117	0.104	0.116	**0.119**	0.118
Hotels	0.610	**0.722**	0.613	**0.722**	0.697	0.671
Census	0.806	0.412	**0.818**	0.768	0.806	0.726
Average IAP	0.678	0.715	0.740	0.764	**0.776**	0.760
Average F_1-score	0.717	0.779	0.776	0.809	**0.827**	0.808

5 Conclusions

We have proposed a new method for object comparison based on classic set theory, and similarity measure used as a metaphor of elements intersection, which we called soft cardinality method. We have explored the modeling ability of the new method with the case study of text comparison applications.

In particular, soft cardinality allows a nice combination of affinity (similarity) between elements and their importance (weights). This property is useful in text applications where approximate string matching measures can be used as affinity function in combination with practical term weighting schemas such as *tf-idf*. Experimental results showed that our approach gives better results than state-of-the-art approximate string comparison methods in a name matching task.

References

1. De Baets, B., De Meyer, H.: Transitivity-preserving fuzzification schemes for cardinality-based similarity measures. European Journal of Operational Research 160, 726–740 (2005)
2. Zadeh, L.: Fuzzy Sets, Fuzzy Logic and Fuzzy Systems. World Scientific, Singapore (1996)
3. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill Book Co., New York (1983)
4. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (2003)
5. Baeza-Yates, R., Ribero-Neto, B.: Modern Information Retrieval. Addison Wesley/ACM Press (1999)
6. Jimenez, S.: A knowledge-based information extraction prototype for data-rich documents in the information technology domain. Master's thesis, National University of Colombia (2008)

Hypergeometric Language Model and Zipf-Like Scoring Function for Web Document Similarity Retrieval

Felipe Bravo-Marquez, Gaston L'Huillier, Sebastián A. Ríos, and Juan D. Velásquez

University of Chile, Department of Industrial Engineering, Santiago, Chile
{fbravo,glhuilli}@dcc.uchile.cl, {srios,jvelasqu}@dii.uchile.cl

Abstract. The retrieval of similar documents in the Web from a given document is different in many aspects from information retrieval based on queries generated by regular search engine users. In this work, a new method is proposed for Web similarity document retrieval based on generative language models and meta search engines. Probabilistic language models are used as a random query generator for the given document. Queries are submitted to a customizable set of Web search engines. Once all results obtained are gathered, its evaluation is determined by a proposed scoring function based on the Zipf law. Results obtained showed that the proposed methodology for query generation and scoring procedure solves the problem with acceptable levels of precision.

1 Introduction

Classic Web search engines have been developed aiming to solve information requirements from users. As proposed in [5], Web search queries can be grouped into three categories: Informational queries, navigational queries, and transactional queries. In [4] a different information requirement is described. We called it, the Web document similarity retrieval problem (WDSRP). This consists of retrieving the most similar documents from the Web using as input a given document instead of a query. Solutions to WDSRP could be applied for plagiarism detection, document impact analysis, or as related ideas retrieval tool. Also, a variation of this problem is known as the Near-Duplicate Detection [3].

Web search engines could perfectly solve the WDSRP allowing users to send complete documents as inputs. However, search engines allow a maximum query length, because long queries take a huge computation time, are hard to cache, and are usually composed of many non relevant terms.

The main contribution of this work is to solve the WDSRP using a probabilistic language model for query generation. Also, a meta search scoring function based on Zipf law is proposed. This function scores considering just the information responded by each search engine, mainly the rankings of retrieved results. This approach aims to avoid costly text processing algorithms over the responded documents. Furthermore, we performed successful experiments in a real world application with promising results.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 303–308, 2010.

2 Previous Work

As described in [7], meta-search engines provide a single unified interface, where a user enters a specific query, the engine forwards it in parallel to a given list of search engines, and results are gathered and ranked into a single list.

The WDSRP has been studied by different researchers [4,9]. These works propose fingerprinting techniques for document representation. In both cases the fingerprint is composed by sentences used as queries. The queries are submitted into a meta-search engine for retrieving an extended list of similar candidate documents.

On the one hand, in [9] document snippets are retrieved and compared with the given document using cosine similarity from the vector space model. On the other hand, Pereira and Ziviani propose in [4] to retrieve the complete text from each Web document, and compare them using text comparison strategies, like Patricia trees and Shingles method.

3 Hypergeometric Language Model for Query Generation

As stated by [5], given a document D, from which a vocabulary V can be extracted, a language model M_D from D is a function that maps a probability measure over strings drawn from V. Language models are used as ranking functions in information retrieval, estimating the probability of generating a query q given a document language model M_D, i.e. $P(q|M_D)$, using a generative expression for the ranking function.

In our query generation task, the probabilistic distribution from the language model is used as a randomized term extraction procedure. The reason for using randomized term permutations, is that similar documents from D do not necessarily contain words in the same order. Furthermore, as a strong but realistic assumption, search engines, where queries will be submitted, treat user natural text queries following a bag of words property [1].

The Hypergeometric Language Model (HLM) is a proposed extension of language models inspired in the multivalued hypergeometric distribution [2], which provides a non replacement property. This means that terms are extracted one by one without replacement. The property is based on the hypothesis that a new term gives more information to a search engine than a repeated term in the generated query, considering that search engines allow a maximum length of input queries.

Consider that the extracted vocabulary is determined by the expression $V = \{t_1, \ldots, t_m\}$, where each term t_i in the document has an assigned positive value w_i stored in vector $\overrightarrow{w} = \{w_1, \ldots, w_m\}$. These values can be determined by several weighting approaches, like Boolean, *tf*, *tf-idf*, among others [8].

A generated query q can be modeled as a list of term pointers extracted from a given vocabulary V, defined by $q = s_1, \ldots s_n$, where each term pointer $s_j \in q$ is an integer taking values in $\{1 \ldots m\} \in V$.

By using the chain rule of probabilities, the probability of generating the query q from a language model M_D, can be defined as, $P(q|M_D) = P(s_1|M_D) \cdot$

$\ldots P(s_n|s_1,\ldots,s_{n-1},M_D)$ and the conditional distribution of extracting the token s_j given an accumulated generated query $q = s_1,\ldots s_n$ and the language model M_D, is determined by,

$$\hat{P}(s_k|q,M_D) = \begin{cases} 0 & \text{if } \exists s_j \in q, s_k = s_j \\ \frac{w_{s_k}}{||\overrightarrow{w}||_1 - \sum_{j=1}^{n} w_{s_j}} & \text{otherwise} \end{cases} \quad (1)$$

HLM also provides a random query generator function. This function generates a sequence of terms using equation 1, giving a higher probability of occurrence to the most relevant terms ranked with the given weighting approach. The function models the term extraction with a multinomial distribution parametrized by $(\frac{\overrightarrow{w}}{||\overrightarrow{w}||_1})$. The without replacement property is modeled by reconstructing the multinomial distribution after each extraction by reducing the dimensionality of $\overrightarrow{w}$ and V by removing the extracted term dimension. Finally, the number of terms extracted is defined by a query *length* parameter.

4 Meta Search Engine and Zipf-Like Scoring Function for Generated Queries

As the coverage of the Web is potentiated by using a set of search engines S, the document retrieval is proposed by the union of the indexed documents presented in each of the search engines used. Assuming that document D is represented by a set of queries Q, generated by HLM (section 3), each query $q \in Q$ is mapped to be submitted to a search engine $s \in S$.

A *queryAnswer* ω is defined as a tuple (s,q,r) , where r represents the ranking assigned by search engine s for query q. Each *queryAnswer* $\omega_{s,q,r}$ will point to a particular document. In our work, all queries originate from the same language model. The hypothesis is that if the set of Web search inverted indexes contains documents with a higher similarity to the given document D, these documents should be founded by many queries in the top of their ranks, and more than once.

After retrieving the set of *queryAnswer* ω objects, they are grouped into *metaAnswer* objects. A *metaAnswer* Ω ($|\Omega| \geq 1$) is a set of ω, where each element points to the same URL or Web document.

Finally, all *metaAnswer* objects are scored and ranked by a proposed approach, based on the Zipf-like distribution function, described below.

The Zipf law, proposed in [10], has been used in the natural language community for the analysis of term frequencies in documents. As stated by [6], if f denotes the popularity of an item and r denotes its relative rank, then f and r are related as $f = \frac{c}{r^\beta}$, where c is a constant and $\beta > 0$. If $\beta = 1$, then f follows exactly the Zipf law, otherwise, is it said to be Zipf-like.

In [6], the Web popularity is modeled as the Zipf law, where the relative frequency with which Web pages are requested for the r^{th} most popular Web page is proportional to $1/r$. Furthermore, in this work we propose to model the relevance of a given *queryAnswer* as a Zipf-like distribution, where the relevance of results presented in a Web search engine are inversely related to their rankings.

All queries are generated by the same language model and have an underlying search intention. If a specific *queryAnswer* appears more than once, the probability that the pointed document is related to the document from which the query was generated increases. Thus, the scoring strategy for a *metaAnswer* Ω is expressed by

$$ZipfLikeScore(\Omega) = \frac{1}{|Q|} \sum_{\omega \in \Omega} \frac{c_s}{r^{\beta_s}} \quad (2)$$

where $c_s \in [0, 1]$ is a constant which represents the average relevance of the best response of a Web search engine s, and β_s represents the decay factor of the results' relevance while the amount of retrieved results increases. With this score measure, we are estimating the relevance of the *queryAnswer* using its ranking and the reliability of search engine results. The score is normalized by the number of queries requested, in order to represent the score by a real value $\in [0, 1]$.

5 Experiments

According to the previously described procedures, a prototype using a term frequency weighting approach and a Spanish stopwords' list was implemented. The prototype allows the client to insert a text without length constraints. However, if a whole document is considered as input, this could increase the number of non relevant results retrieved, because of the randomized query generation process. Therefore, we recommend to use a single paragraph instead of a whole document since it is a self-contained independent information unit.

A hand-crafted set of paragraphs were extracted from a sample of Web documents. For this, a total number of 160 paragraphs were selected from different Spanish written Web sites. The respective URL was stored and classified into three different document types: bibliographic documents or school essays (type 1), blog entries or personal Web pages (type 2), and news (type 3). The distribution of the paragraph types was 77, 53, and 30 respectively. The selected search engines for the experiments were Google, Yahoo!, and Bing. The parameters used for each search engine for the query generation and the ranking procedures were the number of queries generated per paragraph, c_s and β_s, whose assigned values were $(3, 0.95, 0.5)$ for Google, and $(2, 0.93, 0.5)$ for Yahoo! and Bing respectively. Finally the term length of each query was assigned to 13. These values have been assigned by inspection and their formal estimation has been intentionally omitted.

After this, paragraphs were sent to the system as input. The top 15 answers from each paragraph were manually reviewed and classified as relevant or non relevant results (2400 Web documents). The criteria of labeling an answer as relevant was defined that the retrieved document must contain the given paragraph exactly.

The goal of this experiment is to measure the effectiveness of the model at satisfying user information needs. In this case, those needs are related to the WDSRP. The criteria of measuring the relevance of the results is the number of

documents containing exactly the given paragraph (DEPs). The selected evaluation measure was the *precision at k*, which is defined as the number of DEPs retrieved in the top k results divided by the number of documents retrieved in top k results.

6 Results and Discussion

Fig. 1 shows the *precision at k* for results retrieved by the whole set of paragraphs associated with their document types. It is easy to see that the *precision at k* differs with each type of document.

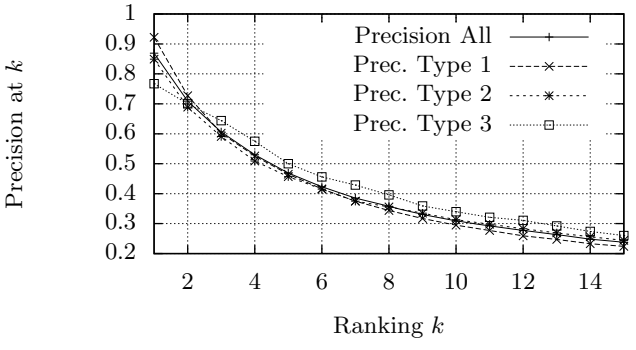


Fig. 1. Precision at k for types of documents

Firstly, type 1 has higher precision at the first values of k than the other types. This is because bibliographic documents are often founded in popular collections like Wikipedia, which are indexed by most Web search engines and usually ranked on top. In this case, a Web document will appear as result for most of generated queries. Secondly, type 2 documents are not as popular as type 1. In this case, blog entries or personal pages are hardly indexed by all Web search engines. Finally, we can observe a lower precision for the first ranked results of type 3 documents. However, a slow decline of the *precision at k* is presented. That is because news contents are repeated in many different Web pages, through the increasing use of content aggregators, so is possible to find a high number of relevant results lower ranked documents.

7 Conclusions

To the best of our knowledge, there are no methods for the Web document similarity retrieval problem (WDSRP) based on randomized query generation and meta-search scoring functions, using mainly the search engines' ranking. The described Zipf-like scoring function can be used as a relevance estimator for a Web search engine result. Also, our probabilistic language model for query generation allows to extract relevant terms, where its weighting approach parameters are sufficient for a key term extraction light-technique.

Acknowledgment

Authors would like to thank continuous support of "Instituto Sistemas Complejos de Ingeniería" (ICM: P-05-004- F, CONICYT: FBO16); FONDEF project (DO8I-1015) entitled, DOCODE: Document Copy Detection (www.docode.cl); and the Web Intelligence Research Group (wi.dii.uchile.cl).

References

1. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
2. Hakerness, W.L.: Properties of the extended hypergeometric distribution. Ann. Math. Statist. 36(3), 938–945 (1965)
3. Henzinger, M.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: SIGIR 2006: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 284–291. ACM, New York (2006)
4. Pereira Jr., A.R., Ziviani, N.: Retrieving similar documents from the web. J. Web Eng. 2(4), 247–261 (2004)
5. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York (2008)
6. Nagaraj, S.V.: Web Caching And Its Applications. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell (2004)
7. Selberg, E., Etzioni, O.: The metacrawler architecture for resource aggregation on the web. IEEE Expert, 11–14 (January–February 1997)
8. Somlo, G.L., Howe, A.E.: Using web helper agent profiles in query generation. In: AAMAS 2003: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pp. 812–818. ACM, New York (2003)
9. Zaka, B.: Empowering plagiarism detection with a web services enabled collaborative network. Journal of Information Science and Engineering 25(5), 1391–1403 (2009)
10. Zipf, G.K.: Human Behavior and the Principle of Least Effort. Addison-Wesley, Reading (1949)

Dual-Sorted Inverted Lists*

Gonzalo Navarro[1] and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Chile, Santiago, Chile
gnavarro@dcc.uchile.cl
[2] School of Comp. Sci. & Inf. Tech., Royal Melbourne Institute of Technology, Melbourne, Australia
simon.puglisi@rmit.edu.au

Abstract. Several IR tasks rely, to achieve high efficiency, on a single pervasive data structure called the *inverted index*. This is a mapping from the terms in a text collection to the documents where they appear, plus some supplementary data. Different orderings in the list of documents associated to a term, and different supplementary data, fit widely different IR tasks. Index designers have to choose the right order for one such task, rendering the index difficult to use for others.

In this paper we introduce a general technique, based on *wavelet trees*, to maintain a single data structure that offers the combined functionality of two independent orderings for an inverted index, with competitive efficiency and within the space of one *compressed* inverted index. We show in particular that the technique allows combining an ordering by decreasing term frequency (useful for ranked document retrieval) with an ordering by increasing document identifier (useful for phrase and Boolean queries). We show that we can support not only the primitives required by the different search paradigms (e.g., in order to implement any intersection algorithm on top of our data structure), but also that the data structure offers novel ways of carrying out many operations of interest, including space-free treatment of stemming and hierarchical documents.

1 Introduction

The last decade has been witness to tremendous progress in the field of compact data strucutres. These data structures mimic the operations of their classical counterparts within much less space and sometimes, surprisingly, offer much wider functionality. Recently, several authors have brought compact data structures to bear on problems in Information Retrieval (IR), in particular ranked document retrieval [14,20]. Although quite different in their details, the common vision of these works is to use breakthroughs in compressed pattern matching as an efficient algorithmic base on which the more sophisticated operations required by IR systems can be built. Our work in this invited paper is complementary to these efforts, applying compact data structures to gain a new perspective on a tool already widely adopted in IR: the inverted index.

* Funded in part by Fondecyt Grant 1-080019, Chile (first author) and by the Australian Research Council (second author).

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 309–321, 2010.

Inverted indexes are an old and simple data structure, yet one of the most successful in IR. They play a central role in any book on the topic [6,31,12,22,11], and are also at the heart of most modern Web search engines.

Given a *text collection* regarded as a set of *documents*, an inverted index is an array of *lists*. Each array entry corresponds to a different *word* or *term* of the collection, and its list points to the documents where that word appears in the text collection. The set of different words is called the *vocabulary*. Empirical laws well accepted in IR [19] establish that the vocabulary is much smaller than the collection size n, more precisely of size $O(n^\beta)$, for some constant $0 < \beta < 1$ that depends on the text type.

Two main variants of inverted indexes exist [5,35]. *Ranked retrieval* is aimed at retrieving documents which are "relevant" to a query, under some criterion. Documents are regarded as vectors, where terms are the dimensions, and the values of the vectors correspond to the relevance of the terms in the documents. The lists point to the documents where each term appears, storing also the weight of the term in that document (i.e., the coordinate value). The query is seen as a set of words, so that retrieval consists of processing the lists of the query words and finding the documents which, considering the weights the query terms have in the document, are predicted to be relevant. Query processing usually involves somehow merging the involved lists, so that documents can be assigned the combined weights over the different terms. Algorithms and different data organizations for this type of query have been intensively studied [25,31,35,3,30]. List entries are usually sorted into order of *descending weight* of the term in the documents.

The second variant is the inverted index for so-called *full-text retrieval* (also known as *boolean retrieval*). These simply find all the documents where the query terms appear. The lists point to the documents where each term appears, usually in *increasing document* order. Queries can be single words, in which case the retrieval consists simply of fetching the list of the word; or disjunctive queries, where one has to fetch the lists of all the query words and merge the sorted lists; or conjunctive queries, where one has to intersect the lists. While intersection can be done also by scanning all the lists in synchronization, it is usually the case that some lists are much shorter than the others [34], and so faster intersection algorithms are possible. These algorithms are especially relevant when many words have to be intersected.

Intersection queries have become extremely popular as Google-like default policies to handle multiword queries. Another important query where intersection is essential is the phrase query, where intersecting the documents where the words appear is the first step. The amount of recent research on intersection of inverted lists witnesses the importance of the problem [15,8,4,7,27,28,13,9]. In particular, in-memory algorithms have received much attention recently, as large main memories and distributed systems make it feasible to hold the inverted index entirely in RAM.

Needless to say, space is an issue in inverted indexes, especially when combined with the goal of operating in main memory. Much research has been carried

out on compressing inverted lists [31,24,35,13], and on the interaction of various compressed representation with different query algorithms, including list intersections. Most of the list compression algorithms for full-text indexes rely on the fact that the document identifiers are increasing, and that the differences between consecutive entries are smaller on the longer lists. The differences are thus represented with encodings that favor small numbers [31,35]. Random access is supported by storing sampled absolute values. For lists sorted by decreasing weights, these techniques can still be adapted, by considering that most documents in a list have small weight values, and within the same weight one can still sort the documents by increasing identifier.

A problem with the current state of the art is that a serious IR system must support both types of retrieval: ranked and full-text. Yet, to maintain reasonable space efficiency, the list must be ordered either by decreasing weights or by increasing document number, but not both. Hence one type of search will be significantly slower than the other, if affordable at all.

In this paper we introduce a data structure that permits, within the same space required for a single compressed inverted index, retrieving the list of documents for any term in either decreasing-weight or increasing-identifier order, thus supporting both types of retrieval. Moreover, we can efficiently support the operations needed to implement any of the intersection algorithms, namely: retrieve the i-th element of a list, retrieve the first element larger than x, retrieve the next element, and several more complex ones. In addition, our structure offers novel ways of carrying out several operations of interest. These include, among others, the support for stemming and for structured document retrieval without any extra space cost. Indeed, the data structure can be generalized to support a combination of any two orderings, not only the two most popular ones.

2 Related Work

2.1 Intersection Algorithms for Inverted Lists

The intersection of two inverted lists can be done in a merge-wise fashion (which is the best choice if both lists are of similar length), or using a set-versus-set (*svs*) approach where the longer list is searched for each of the elements of the shortest, to check if they should appear in the result. Either binary or exponential (also called galloping or doubling) search are typically used for *svs*. The latter checks the list at positions $i + 2^j$ for increasing j, to find an element known to be after position i (but probably close).

Algorithm *bys* [7] is based on binary searching the longer list N for the median of the smallest list M. If the median is found, it is added to the result set. Then the algorithm proceeds recursively on the left and right parts of each list. At each new step the longest sublist is searched for the median of the shortest sublist. It has been shown that *bys* performs about the same number of comparisons as *svs* with binary search. As expected, both *svs* and *bys* improve upon the *merge* algorithm when $|N| >> |M|$ (actually from $|N| \approx 20|M|$).

Multiple lists can be intersected using any pairwise *svs* approach (iteratively intersecting the two shortest lists, and then intersecting the result against the next shortest one, and so on). Other algorithms are based on choosing the first element of the smallest list as an *eliminator* that is searched for in the other lists (usually keeping track of the position where the search ended). If the eliminator is found, it becomes a part of the result. In any case, a new eliminator is chosen. Barbay et al. [9] compared four multi-set intersection algorithms: *i)* a pairwise *svs*-based algorithm; *ii)* an eliminator-based algorithm [8] (called *sequential*) that chooses the eliminator cyclically among all the lists and exponentially searches for it; *iii)* a multi-set version of *bys*; and *iv)* a hybrid algorithm (called *small-adaptive*) based on *svs* and on the so-called *adaptive algorithm* [15], which at each step recomputes the list ordering according to the elements not yet processed, chooses the eliminator from the shortest list, and tries the others in order. In their experimental results [9] the simplest pairwise *svs*-based approach (coupled with exponential search) performed best.

2.2 Data Structures for Inverted Lists

The previous algorithms require that lists can be accessed at any given element (for example those using binary or exponential search) and/or that, given a value, its smallest successor from a list can be obtained. Those needs interact with the methods employed for inverted list compression.

The compression of inverted lists usually represents each list $\langle p_1, p_2, p_3, \ldots, p_\ell \rangle$ as a sequence of d-gaps $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_\ell - p_{\ell-1} \rangle$, and uses a variable-length encoding for these differences, for example γ-codes, δ-codes or Golomb codes [31]. More recent proposals use byte-aligned [29,10,13] or word-aligned [2,33] codes, which lose little compression and are faster at decoding.

Intersection of compressed inverted lists is still possible using a merge-type algorithm. However, approaches that require direct access are not possible as sequential decoding of the d-gaps values is mandatory. This problem can be overcome by sampling the sequence of codes [13,27]. The result is a two-level structure composed of a top-level array storing the absolute values of, and pointers to, the sampled values in the sequence, and the encoded sequence itself.

Assuming $1 \leq p_1 < p_2 < \ldots < p_\ell \leq u$, Culpepper and Moffat [13] extract a sample every $k' = k \log \ell$ values[1] from the compressed list, where k is a parameter. Each of those samples and its corresponding offset in the compressed sequence is stored in the top-level array of pairs $\langle \mathit{value}, \mathit{offset} \rangle$ needing $\lceil \log u \rceil$ and $\lceil \log(\ell \log(u/\ell)) \rceil$ bits, respectively, while retaining random access to the top-level array. Accessing the v-th value of the compressed structure implies accessing the sample $\lceil v/k' \rceil$ and decoding at most k' codes. We call this "(a)-sampling". Results [13] show that intersection using *svs* coupled with exponential search in the samples performs just slightly worse than *svs* over uncompressed lists.

Sanders and Transier [27], instead of sampling at regular intervals of the list, propose sampling regularly at the domain values. We call this a "(b)-sampling".

[1] Our logarithms are in base 2 unless otherwise stated.

The idea is to create buckets of values identified by their most significant bits and then build a top-level array of pointers to them. Given a parameter B (typically $B = 8$), and the value $k = \lceil \log(uB/\ell) \rceil$, bucket b_i stores the values $x_j = p_j \bmod 2^k$ such that $(i-1)2^k \leq p_j < i\,2^k$. Values x_j can also be compressed (typically using variable-length encoding of d-gaps). Comparing with the previous approach [13], this structure keeps only pointers in the top-level array, and avoids the need of searching it (in sequential, binary, or exponential fashion), as $\lceil p_j/2^k \rceil$ indicates the bucket where p_j appears. In exchange, the blocks are of varying length and more values might have to be scanned on average for a given number of samples. The authors also keep track of up to where they have decompressed a given block in order to avoid repeated decompressions.

2.3 Algorithms for Ranked Retrieval

Persin et al. [25] proposed heuristics to solve ranked retrieval problems without scanning all of the lists, and assuming they are sorted by decreasing weight. To fix ideas we will assume, as in their work, that the weight is simply the *term frequency*, that is, the number of times the term appears in the document. This supports various *tf-idf*-like formulas, yet other weights that have been proposed (for example the so-called impacts [3]) can be acommodated as well.

In the *tf-idf* model, the final weight of a document d for a query q is $w(d) = \sum_{t \in q} tf_{t,d} \times idf_t$ summed over all the query terms t. The query retrieves the documents with highest $w(d)$. The term $tf_{t,d}$ is the term frequency of t in d, whereas $idf_t = \log \frac{D}{df_t}$, where D is the total number of documents and df_t is the number of those where t appears. While idf_t (or df_t) is stored in the vocabulary, a term's $tf_{t,d}$ values are stored together with each document d in the inverted list of each term t, and the documents d are sorted by decreasing $tf_{t,d}$ value.

The algorithm retrieves first the shortest list (i.e., with highest idf_t) and stores the documents as candidates for the final answer. Now the other lists are processed in increasing length order. The documents of each list are sought in the set of candidates, and their weight accumulated if found; otherwise they are inserted as new candidates. There is a threshold for continuing processing each list: if the $tf_{t,d}$ values fall below it, the list is abandoned (see Ahn et al. [1] and references therein). There is also a stricter threshold for inserting new elements as candidates. These heuristic thresholds provide a time/quality tradeoff.

3 Wavelet Trees

Let $L[1, N]$ be a sequence of N symbols, where each symbol is in the range $[1, D]$. The wavelet tree of L is a perfect binary tree with D leaves. The leaves are labeled left-to-right with the symbols $[1, D]$ in increasing order. For a given internal node v of the tree, let S_v be the subsequence of L consisting of only the symbols on the leaves in the subtree rooted at v. We store at v a bitvector B_v of $|S_v|$ bits, setting $B_v[i] = 1$ if symbol $S_v[i]$ is appears below the right child of v, and $B_v[i] = 0$ otherwise. Note that S_v is not actually stored, only B_v. Finally, each bitvector

B_v is preprocessed for $O(1)$ rank and select queries [26]: $rank_b(B_v, i)$ returns the number of occurrences of bit b in $B_v[1, i]$; and $select_b(B_v, i)$ returns the position in B_v of the ith occurence of bit b. As we shall see, this preprocessing allows for efficient navigation of the tree when resolving certain range queries on L.

The wavelet tree was originally designed [17] to allow accessing any $S[i]$, as well as computing queries $rank_d(L, i)$ and $select_d(L, i)$ on L for any value d, all in $O(\log D)$ time.

4 Our Data Representation

Let D be the total number of documents in the collection and V the number of different terms. Let $L_t[1, df_t]$ be the list of document identifiers in which term t appears, in decreasing tf order. Let $N = \sum_t df_t$ be the total number of occurrences of *distinct* terms in the documents[2], and $n = \sum_{t,d} tf_{t,d}$ be the total length, in words, of the collection. (Thus $D \leq N \leq \min(DV, n)$.) Finally, let $|q|$ be the number of terms in query q.

We propose to concatenate all the lists L_t into a unique list $L[1, N]$, and store for each term t the starting position s_t of list L_t within L. The sequence L of document identifiers is then represented with a wavelet tree.

The tf values themselves are stored in differential and run-length compressed form in a separate sequence. More precisely, we mark the v_t different $tf_{t,d}$ values of each list in a bitmap $T_t[1, m_t]$, where $m_t = \max_d tf_{t,d}$, and the points in $L_t[1, df_t]$ where value $tf_{d,t}$ changes, in a bitmap $R_t[1, df_t]$. With T_t and R_t preprocessed for $rank$ and $select$ queries we can obtain $tf_{t,L_t[i]} = select_1(T_t, v_t - rank_1(R_t, i) + 1)$.

Finally, the s_t sequence is represented using a bitmap $S[1, N]$, also preprocessed for $rank$ and $select$ queries. Thus $s_t = select_1(S, t)$, and also $rank_1(S, i)$ tells the list $L[i]$ belongs to.

The analysis of wavelet trees [17,23] shows that the space occupied by that of L is $NH_0(L)+o(N \log D)$ bits. Here $NH_0(L) = \sum_d dt_d \log \frac{N}{dt_d} \leq N \log D$, where dt_d is the number of distinct terms in document d. The classical differential encoding of inverted files produces a set of N numbers. If they are sorted by increasing document identifier, these numbers can be represented using $\sum_t df_t \log \frac{N}{df_t} \leq N \log V$ bits plus lower-order terms, by using Elias δ-encoding. If, however, they are sorted by decreasing tf, the analysis is not so clean.

In general the measures are not comparable, yet we remind that our wavelet tree representation will offer the combined functionality of *both* inverted indexes, and more.

The other structures are the tf and the s_t values. The former is encoded with T_t and R_t, which are compressible as they have only v_t bits set. We use a bitmap representation [18, BSGAP, Section 4.3] supporting $rank$ and $select$ in time $O(\log v_t)$ and requiring $v_t \log \frac{m_t}{v_t} + O(v_t \log \log \frac{m_t}{v_t})$ bits for T_t and $v_t \log \frac{df_t}{v_t} + O(df_t \log \log \frac{df_t}{v_t})$ for R_t. This is asymptotically similar to the space needed to

[2] $N = \sum_t df_t$ counts each distinct term once for each document it appears in. This is also the total length of the inverted lists.

represent, in a traditional *tf*-sorted index, each new $tf_{t,d}$ value and the number of entries that share it. The s_t values are represented so that they support constant-time *rank* and *select* [26], requiring $V \log \frac{N}{V} + O(V) + o(N)$ bits, which is less than the usual pointers from the vocabulary to the list of each term. In the worst case the bitmaps add up to $O(N \log \frac{N}{V})$ bits and the time to compute *tf* is $O(\log D)$.

Before considering the classical and extended operations that can be carried out with our data structure, let us raise a couple of issues:

1. *Stemming* is a useful tool to enhance recall [21,32]. A way to provide it is by stemming the terms directly during the parsing, yet in this case the index is unable to provide at the same time non-stemmed searching. One can of course index the stemmed and non-stemmed occurrence of each term, thus increasing the space. We will be able to provide stemmed retrieval without any extra space. All we require is that all the variants of the same stemmed word be contiguous in the vocabulary (this is in many cases automatic as stemmed terms share the same root, or prefix).
2. Most IR systems support a *flat* set of documents, while in XML or file systems, for example, one has a hierarchy of documents and would like to choose, at query time, which level of the hierarchy to use (e.g., to retrieve relevant sections, or relevant chapters, or relevant books), or to carry out ranked IR on a certain subtree. In a temporal (e.g., news archives) or versioned (e.g., Wikipedia) text collection, one might want to search only a range of documents. Our data structure has also support for some queries of this kind without using any extra space.

4.1 Full-Text Retrieval

The full-text index, rather than L_t, requires a list F_t, where the same terms are sorted by increasing document identifier. Different kinds of access operations need to be carried out on F_t. We show now how all these can be carried in $O(\log D)$ time.

Direct retrieval. First, with our wavelet tree representation of L we can find any value $F_t[i]$. This is equivalent to finding the i-th smallest value in $L[st_t, st_{t+1} - 1]$. The algorithm, for a general range $L[l, r]$, is as follows [16]. Let v be the root of the wavelet tree and B_v its bitmap. We count with $n_1 = rank_1(B_v, r) - rank_1(B_v, l-1)$ the number of 1s in $B_v[l, r]$, and with $n_0 = (r-l+1) - n_1$ the number of 0s. If $i \leq n_0$, then there are at least i values d in $L[l, r]$ belonging to the smaller half of the document identifiers, thus we continue recursively on the left child of v, with $l = rank_0(B_v, l-1) + 1$ and $r = rank_0(B_v, r)$. Otherwise, we continue on the right child, with $l = rank_1(B_v, l-1) + 1$, $r = rank_1(B_v, r)$, and $i = i - n_0$. The symbol corresponding to the leaf arrived at is the answer.

We can also extract any segment $F_t[i, i']$, in order, in time $O((i' - i + 1)(1 + \log \frac{D}{i'-i+1}))$. The algorithm is as above, going just by one branch when both i

and i' choose the same, and splitting the interval into two separate searches when they do not. At worst we arrive at $i'-i+1$ leaves of the wavelet tree, but part of the paths to these leaves must be shared. At worst, their paths become all distinct at depth $\log(i'-i+1)$, up to which point we work on all the $O(i'-i+1)$ different wavelet tree nodes, and after then we work on a different path, of length $\log D - \log(i'-i+1)$, for each value.

Another useful operation is to find $F_t[j]$ after having visited $F_t[i]$, for some $j > i$. We show this can be done in amortized time proportional to $\log(j-i+1)$. We need to store $\log D$ numbers m_ℓ, d_ℓ, and b_ℓ, where $m_0 = \infty$ and $d_1 = 0$, and the others are computed as follows when we obtain $F_t[i]$: If, at wavelet tree depth ℓ (the root being depth 1), we must go left, then $m_\ell = \min(m_{\ell-1}, d_\ell + n_0 - i)$ and $d_{\ell+1} = d_\ell$, else $m_\ell = m_{\ell-1}$ and $d_{\ell+1} = d_\ell + n_0$. Here n_0 is the value local to the node. Therefore d_ℓ counts the values skipped to the left, and m_ℓ is the maximum $j-i$ value such that the downward paths to compute $F_t[i]$ and $F_t[j]$ coincide up to depth ℓ. We also set $b_\ell = i$. Now, to compute $F_t[j]$, we consider all the ℓ values, from largest to smallest, until finding the first one such that $j - b_\ell \leq m_\ell$. From there on we recompute the downward path, resetting d_ℓ and m_ℓ accordingly and setting $b_\ell = j$.

Overall, if we carry out this operation k times, across a range $[i, i']$, the cost is $O(\log D + k(1 + \log \frac{i'-i+1}{k}))$, as there can be only $O(1)$ different paths longer than $O(\log(i'-i+1))$ arriving at $i'-i+1$ consecutive leaves, and considering the argument above to analyze the retrieval of $F_t[i, i']$.

Boolean operations. The most important operation for intersecting lists is to be able to find the first j such that $F_t[j] \geq d$, given d. This is usually solved with a combination of sampling and linear, exponential, or binary search. We show now that our representation supports this operation in $O(\log D)$ time.

The operation is as follows. We start at the root v, with bitmap B_v, and the interval $L[l, r]$ with $l = st_t$ and $r = st_{t+1} - 1$. If number d belongs to the first half of the wavelet tree, we descend left, otherwise right. In both cases we update l and r as in the algorithm to retrieve $F_t[i]$. If, at some point, the interval $[l, r]$ becomes empty, then there is no value d in the subtree and we return without a value. If, instead, we arrive at a leaf with a nonempty $[l, r]$ (indeed, it must hold $l = r$ in this case), then the leaf arrived at is d and we return this value. If the recursive step returns no result, then we must look for the first result to the right: If the recursion was to the right child, or it was to the left but there is not any 1 in $B_v[l, r]$, we in turn return with no result. Otherwise we enter the right child looking for the smallest value. From there, we enter recursively the left child only if there is some 0 in $B_v[l, r]$, otherwise we go right. Thus in at most two root-to-leaf traversals we find out the first $d' \geq d$ value in F_t. To obtain j, the position of d' in list F_t, we must add up the n_0 values at all the nodes in the path to d' where we have gone to the right. Note $O(\log D)$ is not far from the time required by a binary search on F_t.

As before, if we know $F_t[j] = d$ and seek for the first value $F_t[j'] \geq d'$, where $d' \geq d$, we can do it in amortized time proportional to $\log(d'-d+1)$. The reason is that, once again, we can redo the work for d' from the corresponding position

of the path used for d (this position is now easier to calculate: it is the first bit at which d and d' differ). For the same reason as before, k searches covering a range $[d, d']$ will cost at most $O(\log D + k \log \frac{d'-d+1}{k})$ time. This is indeed the time required by k successive searches using exponential search.

Finally, our data structure allows us to carry out a particular intersection algorithm. Consider intersecting two lists F_t and $F_{t'}$. This is equivalent to finding the common document numbers in $L[l, r]$ and $L[l', r']$. We proceed as follows. Let v be the wavelet tree root and B_v its bitmap. We descend to the left with $l = rank_0(B_v, l-1) + 1$, $r = rank_0(B_v, r)$, $l' = rank_0(B_v, l'-1) + 1$, and $r' = rank_0(B_v, r')$. We also descend to the right using the same formulas replacing $rank_0$ with $rank_1$. If at any point range $[l, r]$ or range $[l', r']$ is empty, we abort that branch. If we arrive at a leaf d with $l = r$ and $l' = r'$, then we report d as an element in the intersection. This algorithm is indeed a materialization of *bys* [7], where the binary searches have been replaced by constant-time *rank* operations, hence it is an $O(\log D)$ factor faster!

Furthermore, this can be extended to intersecting k terms simultaneously, by maintaining k ranges instead of two, with an $O(k)$ time penalty factor. As soon as one such range disappears, the tree branch is abandoned. This can offer much better performance than the successive pairwise intersections that are currently the best choice in practice. Perhaps more importantly, this scheme can be relaxed to report any document where at least k' out of the k words appear, by abandoning the branches when there are less than k' nonempty intervals. Again, it is not easy to implement this type of search by, say, successive intersections.

We can proceed similarly in case of unions. We start with the k intervals and proceed recursively as long as one of the intervals is nonempty. The cost is $O(M(1 + \log \frac{D}{m}))$, where m is the size of the output and M is the sum, over the returned documents, of the number of intervals where they appeared. The reason is that each interval must be projected to all of its leaves, but again, we arrive at m different leaves overall, but the m paths of length $\log D$ cannot be all different. The classical algorithm is $O(M \log k)$ time, which can be slightly better or worse.

Other operations of interest. If the range of terms $[t, t']$ represent the derivatives of a single stemmed root, we might wish to act as if we had a single list $F_{t,t'}$ containing all the documents where they occur. Indeed, if we apply our previous algorithm to obtain $F_t[i]$ from $L[st_t, st_{t+1} - 1]$, on the range $L[st_t, st_{t'+1} - 1]$, we obtain precisely $F_{t,t'}[i]$, if we understand that a document d may repeat several times in the list if different terms in $[t, t']$ appear in d.

More than that, the algorithms to find the first j such that $F_t[j] \geq d$ can be applied verbatim to obtain the same result for $F_{t,t'}[j] \geq d$ (except that $l = r$ may not hold at the leaves, but rather $r - l + 1$ is the number of times the resulting document appears in $F_{t,t'}$). All the variants of these queries are directly supported as well. Finally, the *bys*-like search can also be applied verbatim in order to intersect stemmed terms (again, at the leaves it may hold that $l \leq r$ and $l' \leq r'$, not necessarily the equality).

Note we can obtain the list of unique documents d for a range of terms $[t, t']$ by using the method that finds the first j and d such that $F_t[j] = d \geq 1$, then $F_t[j'] = d' \geq d + 1$, and so on.

Note also that we have a kind of *summarization* information available. In particular, we can obtain the *local vocabulary* of a document d, that is, the set of different terms that appear in d. By descending to a leaf d, and then locating back all of its occurrences $L[i]$ (via *select* as we move upwards in the wavelet tree), we can find all the i such that $L[i] = d$, and then $rank_1(S, i)$ gives the terms, all in time $O(\log D)$ per term. Moreover, as the occurrences of d within its leaf are a stable sort of the original order in L, we can retrieve the local vocabulary in lexicographic order. This allows, for example, merging in linear time the vocabularies of different documents, or binary searching for a particular term in a particular document (yet, the latter is easier via two *rank* operations on L: $rank_d(L, s_{t+1} - 1) - rank_d(L, s_t - 1)$; then the corresponding position can be obtained by $select_d(L, 1 + rank_d(L, s_t - 1))$).

The way to support hierarchical documents by mapping them to ranges $[d_l, d_r]$ of documents is relatively obvious. It is sufficient to restrict all our wavelet tree traversals to the nodes that contain leaves in this range, disregarding others.

4.2 Ranked Retrieval

We focus now on the operations of interest for ranked retrieval, which are also simulated essentially in $O(\log D)$ time.

Direct access and Persin's algorithm. The L_t lists used for ranked retrieval are directly concatenated in L, so $L_t[i]$ is obtained by extracting symbol $L[s_t + i - 1]$ using the wavelet tree. Recall that the term frequencies *tf* are also available in time $O(\log D)$. Again, a range $L_t[i, i']$ is obtained in $O((i' - i + 1) \log \frac{D}{i' - i + 1})$ time, as follows. Start at the root v with bitmap B_v and let the range to extract be $[l, r]$. Compute the corresponding ranges $[l_0, r_0]$ and $[l_1, r_1]$ for the left and right child, as usual, and descend recursively to both. Stop the recursion when the range is empty. Upon arriving at a leaf d, report d. One can, for example, find with $select_1(R_t, v_t - rank_1(T_t, p) + 1) - 1$ the length of the prefix of L_t where the *tf* values are $> p$, which is useful for Persin's algorithm [25].

This algorithm is correct but it has the problem of retrieving the documents in document order, not in *tf* order as they are in L_t. To recover the correct ordering, we must merge the results at each internal wavelet tree node during the recursion, as they arrive. At node v with results returned by its left and right child, we use $select_0$ and $select_1$, respectively, in B_v to map their positions in the bitmap of v. We advance in both lists so as to build their union in the correct order in B_v prior to sending them upwards. Note that, due to this merging effort, the complexity is again $O((i' - i + 1) \log D)$, but in practice the method is faster than extracting each $L[i]$ one by one.

Note, nevertheless, that retrieving the highest-*tf* documents in document order is indeed beneficial for Persin's algorithm, where a difficulty is how to accumulate results across unordered document sets. One could use the threshold p of the

algorithm to retrieve the relevant documents from the next list to consider, and gracefully merge the result with the current candidates, as all are automatically in increasing document order.

Other operations of interest. Any candidate document d in Persin's algorithm can be directly evaluated, obtaining its weight $w(d)$, by finding it within L_t for each $t \in q$ (with $rank_d$ and $select_d$ on L, as explained), and its tf obtained, all in $O(|q| \log D)$ time.

If we wish to use stemming, we might want to retrieve prefixes of several lists L_t to $L_{t'}$. We may carry out the previous algorithm to deliver all the distinct documents in these prefixes, now carrying on the $t'-t+1$ intervals as we descend in the wavelet tree. When we arrive at the relevant leaves d, the corresponding positions will be contiguous, thus we can naturally return just one occurrence of each d in the union. This is a simplification of the method sketched earlier to obtain the unique documents in $F_{t,t'}$. If we wish to obtain the sum of the tf values for all the stemmed terms in d, we can traverse the wavelet tree upwards for each interval element at leaf d, and obtain its tf upon finding its position i in L. Alternatively, we could store the tf values aligned to the leaves and mark their cumulative values on a compressed bitmap, so as to obtain the sum as the difference of two $select_1$ queries on that bitmap. The space for tf becomes now $O(N \log \frac{n}{N})$ bits. In any case, this delivers the results in document order.

There is also some basic support for hierarchical documents: If we wish to know the total tf of t in a range $[d, d']$ of documents, this range is exactly covered by $O(\log D)$ wavelet tree nodes. We can descend, projecting the range of L_t in L, until those nodes, and then move upwards again to find their positions and individual tf values, to add them all.

More interesting is the fact that we can carry out ranked retrieval restricted to any range of documents $[d, d']$ (e.g., within a particular XML subtree or filesystem directory or range of document versions). Once again, it is sufficient to restrict any of the operations described above so that they do not descend to a node whose range does not intersect $[d, d']$. This automatically yields, for example, Persin's algorithm over a range of documents.

5 Conclusions and Future Work

In this paper we have shown how wavelet trees can be used to achieve dual-ordered inverted lists, that is, lists that are simultaneously sorted by document id (useful for intersections and document retrieval) and by term impact or frequency (useful for ranked retrieval). We are in the process of translating these data structures into practice and verifying them experimentally.

Finally, we emphasize that our approach can be applied to any ordering on the documents. A very different and interesting ordering from the one considered here is that induced by the suffix array (the D array of Culpepper et al. [14]). Applying our data structure and *bys*-like intersection algorithm over this ordering immediately yields efficient "bag-of-strings" queries from suffix arrays, further bridging the gap between IR problems and optimal pattern matching data structures.

References

1. Anh, V., de Kretser, O., Moffat, A.: Vector-space ranking with effective early termination. In: Proc. 24th SIGIR, pp. 35–42 (2001)
2. Anh, V., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. 8(1), 151–166 (2005)
3. Anh, V., Moffat, A.: Pruned query evaluation using pre-computed impacts. In: Proc. 29th SIGIR, pp. 372–379 (2006)
4. Baeza-Yates, R.: A fast set intersection algorithm for sorted sequences. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 400–408. Springer, Heidelberg (2004)
5. Baeza-Yates, R., Moffat, A., Navarro, G.: Searching Large Text Collections, pp. 195–244. Kluwer Academic, Dordrecht (2002)
6. Baeza-Yates, R., Ribeiro, B.: Modern Information Retrieval. Addison-Wesley, Reading (1999)
7. Baeza-Yates, R., Salinger, A.: Experimental analysis of a fast intersection algorithm for sorted sequences. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 13–24. Springer, Heidelberg (2005)
8. Barbay, J., Kenyon, C.: Adaptive intersection and t-threshold problems. In: Proc. 13th SODA, pp. 390–399 (2002)
9. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. ACM J. Exp. Alg. 14, article 7 (2009)
10. Brisaboa, N., Fariña, A., Navarro, G., Esteller, M.: S,C-dense coding: an optimized compression code for natural language text databases. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 122–136. Springer, Heidelberg (2003)
11. Buettcher, S., Clarke, C., Cormack, G.V.: Information Retrieval: Implementing and Evaluating Search Engines. MIT Press, Cambridge (2010)
12. Croft, B., Metzler, D., Strohman, T.: Search Engines: Information Retrieval in Practice. Pearson Education, London (2009)
13. Culpepper, J.S., 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)
14. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: Proc. 18th ESA (2010) (to appear)
15. Demaine, E., Munro, I.: Adaptive set intersections, unions, and differences. In: Proc. 11th SODA, pp. 743–752 (2000)
16. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 1–6. Springer, Heidelberg (2009)
17. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
18. Gupta, A.: Succinct Data Structures. PhD thesis, Duke University, USA (2007)
19. Heaps, H.: Information Retrieval - Computational and Theoretical Aspects. Academic Press, London (1978)
20. Hon, W.-K., Shah, R., Vitter, J.S.: Space-efficient framework for top-k string retrieval problems. In: Proc. 50th IEEE FOCS, pp. 713–722 (2009)
21. Hull, D.A.: Stemming algorithms: A case study for detailed evaluation. J. Amer. Soc. Inf. Sci. 47(1), 70–84 (1996)

22. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge (2008)
23. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)
24. Navarro, G., Moura, E., Neubert, M., Ziviani, N., Baeza-Yates, R.: Adding compression to block addressing inverted indexes. Inf. Retr. 3(1), 49–77 (2000)
25. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. J. Amer. Soc. Inf. Sci. 47(10), 749–764 (1996)
26. 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)
27. Sanders, P., Transier, F.: Intersection in integer inverted indices. In: Proc. 9th ALENEX (2007)
28. Sanders, P., Transier, F.: Compressed inverted indexes for in-memory search engines. In: Proc. 10th ALENEX, pp. 3–12 (2008)
29. Scholer, F., Williams, H., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: Proc. 25th SIGIR, pp. 222–229 (2002)
30. Strohman, T., Croft, B.: Efficient document retrieval in main memory. In: Proc. 30th SIGIR, pp. 175–182 (2007)
31. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes, 2nd edn. Morgan Kaufmann, San Francisco (1999)
32. Xu, J., Croft, W.B.: Corpus-based stemming using cooccurrence of word variants. ACM Trans. Inf. Sys. 16(1), 61–81 (1998)
33. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. 18th WWW, pp. 401–410 (2009)
34. Zipf, G.: Human Behaviour and the Principle of Least Effort. Addison-Wesley, Reading (1949)
35. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comp. Surv. 38(2), article 6 (2006)

CST++

Enno Ohlebusch[1], Johannes Fischer[2], and Simon Gog[1]

[1] Universität Ulm, Institut für Theoretische Informatik, 89069 Ulm, Germany
{enno.ohlebusch,simon.gog}@uni-ulm.de
[2] KIT, Institut für Theoretische Informatik, 76131 Karlsruhe, Germany
johannes.fischer@kit.edu

Abstract. Let A be an array of n elements taken from a totally ordered set. We present a data structure of size $3n + o(n)$ bits that allows us to answer the following queries on A in constant time, without accessing A: (1) given indices $i < j$, find the position of the minimum in $A[i..j]$, (2) given index i, find the first index to the left of i where A is strictly smaller than at i, and (3) same as (2), but to the right of the query index. Based on this, we present a new compressed suffix tree (CST) with $O(1)$-navigation that is smaller than previous CSTs. Our data structure also provides a new (practical) approach to compress the LCP-array.

1 Introduction

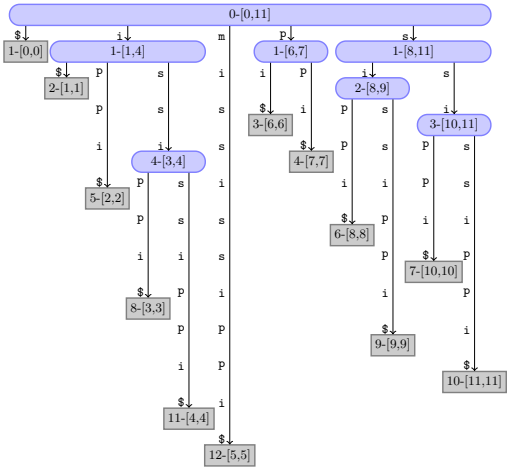
A suffix tree (ST) for a string S of length n is a compact trie storing all the suffixes of S, in the sense that the characters on any root-to-leaf path spell out exactly a suffix. An example is shown in Fig. 1. The ST is an extremely important data structure with applications in exact or approximate string matching, bioinformatics, and document retrieval, to mention only a few examples.

The drawback of STs is their huge space consumption of 20–40 times the text size ($O(n \lg n)$ bits in theory), even when using carefully engineered implementations. To reduce their size, in recent years several authors provided compressed variants of STs (CSTs), both in theory [1,2,3,4,5,6,7] and in practice [8,9,10].

We regard the CST as an abstract data type supporting the following operations (u and v are nodes): Root() yields the root, IsAncestor(u, v) is true iff u is an ancestor of v, Count(u) gives the number of leaves (suffixes) below u, LeafLabel(u) for leaf u yields the position in S where the corresponding suffix begins, SDepth(u) gives u's string-depth (number of characters on root-to-u path), Parent(u)/FChild(u)/NSibling(u) yields the parent/first child/next sibling of u (if existent), SLink(u) gives the unique node v with root-to-v label $\alpha \in \Sigma^\star$ if the root-to-u label is $a\alpha$ for some $a \in \Sigma$, Lca(u, v) yields the lowest common ancestor of u and v, TDepth(u) gives the tree depth of u, and Child(u, a) gives the child v of u such that the label on edge (u, v) starts with $a \in \Sigma$. Here and in the following, Σ denotes the underlying alphabet of size σ.

In all CSTs, there is a trade-off between the *space* it occupies and the *time* it needs to support the operations from the previous paragraph. We first make an extensive survey of existing approaches in the following section (along with some

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 322–333, 2010.



i	SA	LCP	$S_{\mathsf{SA}[i]}$
0	11	-1	$
1	10	0	i$
2	7	1	ippi$
3	4	1	issippi$
4	1	4	ississippi$
5	0	0	mississippi$
6	9	0	pi$
7	8	1	ppi$
8	6	0	sippi$
9	3	2	sissippi$
10	5	1	ssippi$
11	2	3	ssissippi$
12		-1	

Fig. 1. Left: suffix tree for $S = \texttt{mississippi\$}$. Right: suffix- and LCP-array.

simplifications and unifying observations); our technical results and an outline on the rest of this article will be given at the end of that survey.

2 A Guided Tour through Compressed Suffix Trees

Let S_i denote the i'th suffix of S. A CST on S can be divided into three components: (1) the *suffix array* SA, specifying the lexicographic order of S's suffixes, defined by $S_{\mathsf{SA}[0]} < S_{\mathsf{SA}[1]} < \cdots < S_{\mathsf{SA}[n-1]}$ (hence SA captures information about the *leaves*); (2) the *LCP-array* LCP, storing the lengths of the *longest common prefixes* of lexicographically adjacent suffixes: $\mathsf{LCP}[0] = -1 = \mathsf{LCP}[n]$ and for $1 \leq i < n$, $\mathsf{LCP}[i] = \max\{k \geq 0 \ : \ S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[i-1]}$ share a length-k-prefix$\}$ (hence LCP captures information about *internal* nodes); and (3) additional data structures for simulating the *navigational operations* mentioned in the introduction. The goal of a CST is to compress each of these three components.

Compressed Suffix Arrays. There is a wealth of literature on Compressed Suffix Arrays (CSAs), offering different trade-offs between the space they require and the lookup-time t_{SA} they provide. We refer the reader to Navarro's and Mäkinen's survey paper [11] for an in-depth overview of the different alternatives. One of the best choices is the CSA due to Grossi et al. [12], achieving $t_{\mathsf{SA}} = O(\lg^{\epsilon} n)$ with space $(1 + \frac{1}{\epsilon})nH_k + o(n)$ bits (assuming an alphabet of size $\sigma = O(\text{polylg}\, n)$). Here and in the following, H_k denotes the *empirical entropy* of order k of the input text S, which, at least in the realm of text indexing, is used as the *de facto* standard for the compressibility of a text [11].

Compressed LCP-Arrays. There are also different options for compressing the second component (LCP-array), for convenience summarized in Tbl. 1. Although we focus primarily on theoretical results, we also included the simple byte-aligned variant [13] in Tbl. 1, as it is still competitive [10] with recent

Table 1. Trade-offs for storing and accessing the LCP-array. $^{(*)}$worst case $O(n \lg n)$.

ref.	space in bits	t_{LCP}	comment
[14]	$n \lg n$	$O(1)$	uncompressed variant
[13]	$8n^{(*)}$	$O(1)$	practical byte-aligned variant
NEW	$\mathbf{4\text{–}6}\boldsymbol{n}^{\mathbf{(*)}}$	$\boldsymbol{O(1)}$	**only in conjunction with our new CST**
[3]	$2n + o(n)$	$O(t_{\mathsf{SA}})$	
[4]	$nH_k + o(n)$	$O(\lg^{1+\alpha} n)$	constant $0 < \alpha < 1$
[5]	$O(nH_k \lg \frac{1}{H_k})$	$O(t_{\mathsf{SA}})$	
[15]	$O(\frac{n \lg n}{q})$	$O(t_{\mathsf{SA}} \cdot q)$	time amortized; $O(t_{\mathsf{SA}} \cdot n)$ time in the worst case
[7]	$O(\frac{n}{\lg \lg n})$	$O(t_{\mathsf{SA}} \lg^{\beta} n)$	constant $0 < \beta < 1$
[16]	$O(nH_k \lg \frac{1}{H_k})$	$O(t_{\psi} \frac{n}{R^{1-\gamma}})$	R = number of equal-letter runs in BWT of S

practical implementations [8,9]. Apart from the first two [14,13], all other variants exploit the redundancy arising from listing the LCP-values in *text* order (as opposed to *suffix array* order); this is sometimes called the *permuted LCP-array* [15,16]. Note in particular that all LCP-variants with $o(n \lg n)$ bits have non-constant access-time t_{LCP} (assuming an underlying *compressed* SA).

In principle, implementations of suffix- and LCP-arrays are interchangeable in CSTs. Hence, combining the different variants mentioned above already yields a rich variety of CSTs, although some CSTs favor certain suffix- or LCP-arrays.

Succinct Tree Navigation. The real difference between the various known CSTs lies in the way they support the navigational operations (third component above), which we are going to discuss next. There are two main lines of research:

(a) Storing the tree topology *explicitly* by a sequence of balanced parentheses (either BPS [17] or DFUDS [18]).
(b) Deriving the tree topology *implicitly* from intervals in the LCP-array as follows: let v be a node in ST such that the labels on the root-to-v path form the string ω of length ℓ. Then there is an interval $[v_l..v_r]$ in the suffix array such that ω is the longest common prefix of $S_{\mathsf{SA}[v_l]}, \ldots, S_{\mathsf{SA}[v_r]}$, and ω in not a prefix of any other suffix. In the LCP-array, this interval is called an *LCP-interval of LCP-value* ℓ [13]. It has the properties $\mathsf{LCP}[v_l] < \ell$, $\mathsf{LCP}[v_r + 1] < \ell$, and $\mathsf{LCP}[k] \geq \ell$ for all k with $v_l < k \leq v_r$. The indices $i_1 < i_2 < \cdots < i_m$ in $[v_l..v_r]$ with $\mathsf{LCP}[i_j] = \ell$ (where $1 \leq m \leq \sigma - 1$) are called LCP-indices, and the child intervals of $[v_l..v_r]$ are $[v_l..i_1 - 1]$, $[i_1..i_2 - 1], \ldots, [i_m..v_r]$, where some of them may be singleton intervals $[i_j..i_j]$.

Tbl. 2 summarizes all known existing CSTs, and shows those with type-(a)-navigation in the left half [1,2,3,4], and those of type (b) in the right half [5,6].

Approach (a), pursued either verbatim [1,2,3] or on a subset of certain sampled nodes [4], has the advantage that the rich results on navigation in general-purpose succinct trees (see [17,18,19,20] and references therein) can be re-used for CSTs. Hence, whenever a new operation for BPS or DFUDS is discovered, this automatically carries over to CSTs using this representation.

Table 2. Comparison of different CSTs (space in bits on top of CSA & LCP). The $O(\cdot)$ is omitted in all operations. Trees with type-(a) navigation are in the left half.

space	[1,2,3] $4n$	[4] $o(n)$	[5] $o(n)$	[6] $2n$	NEW $\mathbf{3n}$
Root, IsAncestor,Count	1	1	1	1	**1**
LeafLabel	t_{SA}	$\lg^{1+\alpha} n$	t_{SA}	t_{SA}	$\boldsymbol{t_{\mathsf{SA}}}$
SDepth	t_{LCP}	$\lg^{1+\alpha} n$	t_{LCP}	t_{LCP}	$\boldsymbol{t_{\mathsf{LCP}}}$
Parent	1	$\lg^{1+\alpha} n$	t_{LCP} polylglg n	$t_{\mathsf{LCP}} \lg \sigma$	**1**
FChild, NSibling	1	$\lg^{1+\alpha} n$	t_{LCP} polylglg n	t_{LCP}	**1**
SLink, Lca	1	$\lg^{1+\alpha} n$	t_{LCP} polylglg n	$t_{\mathsf{LCP}} \lg \sigma$	**1**
TDepth	1	$\lg^{2+2\alpha} n$	t_{LCP} polylglg n	1	**1**
Child	$t_{\mathsf{SA}} \lg \sigma$	$\lg \sigma + \lg^{1+\alpha} n$	$t_{\mathsf{SA}} \lg \sigma$	$t_{\mathsf{SA}} \lg \sigma$	$\boldsymbol{t_{\mathsf{SA}} \lg \sigma}$

The most prominent CST from group (a) is the one due to Sadakane [3], called Sad-CST henceforth. Sad-CST supports very fast operations, but its disadvantage is that it needs up to $4n + o(n)$ bits for the sequence of parentheses, as the ST consists of n leaves and up to $n-1$ additional internal nodes (the $o(n)$-term comes from the extra data structures for navigation). There are at least two explanations why the resulting $4n$ bits are a waste of space. First, a suffix tree is not an arbitrary tree, but a *compact* one, meaning that it does not contain nodes of out-degree 1. Thus, in principle it should be possible to represent such a compact tree on n' nodes with less than $2n'$ bits ($n' < 2n$ is the number of nodes in ST). The second (and even more convincing) reason is that the LCP-array itself already captures the topology of the ST by means of LCP-intervals. Hence, in theory no space at all has to be spent for the topology, as the LCP-array is already part of any CST.

Motivated by the observations from the previous paragraph, more recent CSTs [5,6] base their navigation on intervals in the LCP-array LCP. CSTs from this (b)-group express all navigational operations by a combination of three basic queries on LCP: (i) range minimum queries (RMQ), where for two given indices $i \leq j$ in LCP one seeks the (first) position of the minimum in $\mathsf{LCP}[i, j]$, (ii) previous smaller value queries (PSV), where for an index i in LCP one searches for the rightmost position to the left of i where LCP is strictly smaller than at position i, and (iii) next smaller value queries (NSV), which are defined analogously for the sub-array to the right of the query point.

One could use separate data structures for each of the three queries mentioned above (RMQ/PSV/NSV), amounting to $6n + o(n)$ bits in total: $2n + o(n)$ for RMQs [21] and $2n + o(n)$ each for PSV and NSV [5]. This, however, would constitute a disadvantage over the $4n + o(n)$ bits used by the methods storing the explicit topology (a). To cope with this situation, Fischer et al. [5] proposed to "sparsify" the RMQ/PSV/NSV-structures to use only $o(n)$ bits in total, thereby giving up constant-time retrieval of RMQ/PSV/NSV-values. This, nonetheless, constitutes no theoretical slowdown for the suffix-tree operations, as they have to make at least one lookup to LCP, which already costs $\Omega(t_{\mathsf{SA}}) = \Omega(\lg^{\epsilon} n)$ at the very best in the presence of a compressed LCP-array (see Tbl. 1).

A different idea to reduce the $6n$-bit term is to use a *combined* data structure for RMQ/PSV/NSV. This is the idea of Ohlebusch and Gog's OG-CST [6], who noted that a $2n$-bit balanced parentheses representation BPR of LCP's Super-Cartesian Tree [22] provides this functionality, at least for RMQ and NSV. Although BPR is not defined in mathematically rigorous terms, the authors provide an algorithm that constructs BPR in linear time [6, Alg. 3]. It works by scanning LCP from left to right, and writing '$)^k($' in step i if LCP$[i]$ is the NSV of k preceding positions. This results in a $2n$-bit sequence BPR that supports RMQ and NSV in $O(1)$ time (using additional $o(n)$ bits); PSV-queries are supported in $O(t_{\mathsf{LCP}} \lg \sigma)$ time by a binary search over the at most σ closing parentheses (which are consecutive), taking the LCP-values of the corresponding positions as search keys. A different (practical) proposal [9] of a combined data structure is based on the *range min-max tree* [20].

The parentheses sequence BPR from OG-CST has an interesting connection to a seemingly different data structure, the 2d-Min-Heap [21]: BPR is DFUDS of the 2d-Min-Heap of LCP read from right to left (reversed). To see why this is so, recall the definition of the 2d-Min-Heap $\mathcal{M}_A$ of an array A [21, Def. 1]: it is an ordered tree on nodes $\{1, 2, \ldots, n\}$, defined such that i is the parent of j iff $A[i]$ is the PSV of $A[j]$. Writing the DFUDS of $\mathcal{M}_A$, where a node with k children is encoded as '$(^k)$', results in a sequence of parentheses where node i appears as '$(^k)$' if $A[i]$ is the PSV of k succeeding positions (as opposed to NSV in BPR). Given these similarities, it is also not surprising that the construction algorithms for BPR [6, Alg. 3] and the DFUDS of $\mathcal{M}_{\mathsf{LCP}}$ [21, Sect. 4] are strikingly similar.

Unifying View. The separation between CSTs with navigation of type (a) and those of type (b) is actually not as strict as it may seem at first sight. Indeed, if for (a) we restrict ourselves to the BPS, then the resulting sequence $U[1, 2n']$ *implicitly* lists the depths of the ST-nodes as visited in an Euler-Tour. For such a sequence U, Sadakane and Navarro [20] showed that the efficient support of operations very similar to (if not the same as) RMQ/PSV/NSV yields all known navigational operations in the underlying tree. The only conceptual difference is that the underlying BPS U lists the *tree*-depths of ST, whereas type-(b)-navigation works on LCP, which lists the *string*-depths (a similar observation is made by [9]).

Construction. Particular emphasis has been put on efficient construction algorithms for all three components of CSTs. Here, "efficiency" encompasses both construction *time* and *space*, as the latter can cause a significant memory bottleneck. Most CSAs can be built in $O(n)$ time and $O(n)$ bits (constant alphabet), or $O(n \lg \lg \sigma)$ time using $O(n \lg \sigma)$ bits (arbitrary alphabet size σ) of space [23].

The $2n + o(n)$-bit LCP-array [3] can be constructed with no extra space in addition to CSA and the text [8]. Other smaller LCP-variants from Tbl. 1 would need these $2n$ bits as their intermediate working space.

Concerning the navigational component, the most space-consuming part is the sequence of balanced parentheses, either that of the ST (Sad-CST), or of the Super-Cartesian Tree (OG-CST). For Sad-CST, it has been shown how to

Algorithm 1. Space-efficient construction of the DFUDS U of a suffix tree

```
1  push(n + 1)          /* LCP[n + 1] = −1 */
2  for i ← n downto 1 do
3    write ')' to U's current beginning          /* accounts for leaf SA[i] */
4    while LCP[i] < LCP[top()] do
5      write '((' to U's current beginning       /* node with ≥ 2 children */
6      λ ← pop()
7      while LCP[λ] = LCP[top()] do
8        write '(' to U's current beginning      /* additional children */
9        λ ← pop()
10   push(i)
11 write '(' to U's current beginning            /* to make U balanced */
```

construct the BPS in $O(n)$ time using $O(n)$ bits of working space [24, 8]. However, these algorithms are quite complex (using Elias δ-codes, batched updates, ...) and involve large big-O-constants, and are therefore not used in existing implementations [8, 9].

There is a simpler way if we construct ST's DFUDS U instead of its BPS, shown in Alg. 1.[1] As in previous approaches [24, 8], we construct U from the LCP-array of S. We scan LCP from *right to left* and build the DFUDS from *back to front*. Note the similarity of our algorithm to the construction of the balanced parentheses representation of LCP's Super-Cartesian Tree [6, Alg. 3] (and to the algorithm for constructing the DFUDS of LCP's 2d-Min-Heap [21, Sect. 4.1]).

The correctness follows from the fact that if v is a node in ST with k children $v_1, \ldots, v_k$, then there are $k-1$ LCP-indices $p_1, \ldots, p_{k-1}$ in v's LCP-interval $[v_l..v_r]$. These indices p_j must have v_l as their PSV; hence, when scanning position $i = v_l$ in Alg. 1, we write k opening parentheses (2 in the outer and $k-2$ in the inner while-loop). The outer while-loop accounts for the fact that v_l could be the left delimiter of several nested LCP-intervals with decreasing LCP-values.

The only drawback of Alg. 1 is that the stack might still use $O(n \lg n)$ bits in the worst case. To cope with such a situation, Fischer [21, Sect. 4.2] shows how to represent a stack containing at most n in- or decreasing elements from $[1, n]$ with a bit-vector of length n, assuming that elements are only pushed in a right-to-left (or left-to-right) manner. To retain constant-time access to the elements on the stack, we need three further tables of size at most $O(\frac{n \lg\lg n}{\lg n}) = o(n)$ bits. Thus, $n + o(n)$ bits of working space suffice for constructing ST's DFUDS.

It is interesting to note that if one substitutes the '((' by a simple '(' in line 3 of Alg. 1, then the result is again the DFUDS of LCP's 2d-Min-Heap [21]! This shows yet another interesting connection between the triumvirate suffix tree/Super-Cartesian Tree/2d-Min-Heap.

[1] Note that BPS and DFUDS support the same set of operations [25], so we can choose whatever representation is more suitable for our purposes.

2.1 Our Results in Context

In Sect. 3, we first show a general result that is independent from CSTs:

Theorem 1. *Given an array A of n elements taken from a totally ordered set, there is a data structure of size $3n+o(n)$ bits that supports* RMQ/PSV/NSV*-queries on A in $O(1)$ time, without needing access to A at query time.*

In Sect. 4, we then use Thm. 1 to improve upon OG-CST:

Theorem 2. *Let S be a text of size n with characters from an alphabet of size σ. Given S's suffix array with access time t_{SA} and its LCP-array with access time t_{LCP}, there is a CST with additional $3n+o(n)$ bits that supports* Count, IsAncestor, Parent, FChild, NSibling, SLink, Lca *and* TDepth *in $O(1)$,* LeafLabel *in t_{SA},* SDepth *in $O(t_{\mathsf{LCP}})$, and* Child *in $O(t_{\mathsf{SA}} \lg \sigma)$ time.*

This latter result should be seen in the context of other CSTs; see again Tbl. 2. In terms of space, our new CST resides between Sad-CST with $4n$ additional bits [3] and OG-CST with $2n$ bits [6]. However, it is equally fast as the larger of these (Sad-CST). Clearly, there are smaller variants of CSTs [4,5], but due to their increased navigation time they are incomparable to Thm. 2.

Finally, we briefly sketch how the data structure from Thm. 2 yields a practicable and small LCP-array.

3 A New Representation of Super-Cartesian Trees

In this section, we prove Thm. 1 (with linear construction time).

Definition 1. *Let $A[1..n]$ be an array of elements of a totally ordered set $(M, \leq)$. To deal with boundary cases, we add an element $-\infty$ to M which is smaller than any other element of M, and define $A[0] = -\infty = A[n+1]$. For an index $1 \leq i \leq n$, we define:*

$$\texttt{FEV}(i) = \min\{k \mid \texttt{PSV}(i) < k < \texttt{NSV}(i) \text{ and } A[k] = A[i]\}$$
$$\texttt{LEV}(i) = \max\{k \mid \texttt{PSV}(i) < k < \texttt{NSV}(i) \text{ and } A[k] = A[i]\}$$

An interval $[i..j]$, where $1 \leq i \leq j \leq n$, is called mound-interval *if $A[k] > \max\{A[i-1], A[j+1]\}$ for all k with $i \leq k \leq j$.*

Let us call the interval $[\texttt{PSV}(i)+1..\texttt{NSV}(i)-1]$ the mound-interval of i. Then, $\texttt{FEV}(i)$ ($\texttt{LEV}(i)$) is the first (last) index in the mound-interval of i at which a value equal to $A[i]$ can be found. As an example consider index 5 in the array A from Fig. 2: Its mound-interval is $[1..11]$ because $\texttt{PSV}(5) = 0$ and $\texttt{NSV}(5) = 12$. Furthermore, $\texttt{FEV}(5) = 1$ and $\texttt{LEV}(5) = 8$.

The definition of the Super-Cartesian tree is taken from [22]; cf. Fig. 3.

Definition 2. *Let $A[l..r]$ be an array of elements of a totally ordered set $(M, \leq)$ and suppose that the minima of $A[l..r]$ appear at positions $p_1 < p_2 < \cdots < p_k$ for some $k \geq 1$. The* Super-Cartesian tree $\mathcal{C}^{sup}(A[l..r])$ *of $A[l..r]$ is recursively constructed as follows:*

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$A[i]$	-1	0	1	1	4	0	0	1	0	2	1	3	-1

Fig. 2. $A[1..11]$ is an array of natural numbers. We choose -1 as the element that is smaller than every natural number and set $A[0] = -1 = A[12]$.

- *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. 3, 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.*

Ohlebusch and Gog [6] showed that the Super-Cartesian tree of $A[1..n]$ can be represented by a sequence of balanced parentheses BPR, and they give a construction algorithm that is solely based on A. However, the sequence BPR lacks some information, as the cases "right child" and "right sibling" are treated in the same fashion, and the array A itself is required to compensate for this. Here, we will compensate for the lack of information by enhancing the BPR with a bitstring B of length $n + 2$. A closing parentheses corresponding to a node that is a right sibling is marked with a 0, otherwise it is marked with a 1. The construction of the enhanced BPR of the Super-Cartesian tree of array A starts at the root of the tree and proceeds as follows (see Fig. 4 for an example):

1. Write the balanced parentheses sequence of the left child.
2. Write an opening parenthesis.
3. Write the balanced parentheses sequence of the right child/sibling.
4. Write a closing parenthesis. If the node under consideration is a right sibling, append 0 to B; otherwise append 1 to B.

Similar to [6, Alg. 3], the enhanced BPR of (the Super-Cartesian tree of) an array can be constructed *without* knowing the Super-Cartesian tree itself; see Alg. 2. Again, the stack can be implemented with $n + o(n)$ bits [21, Sect. 4.2].

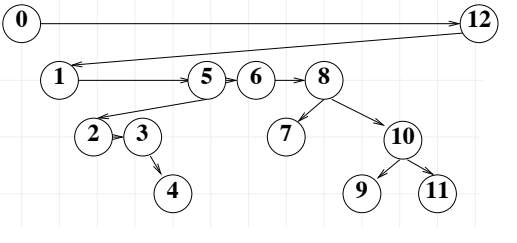


Fig. 3. The Super-Cartesian tree of the array A from Fig. 2

					1	0	1				1			1			1	1	0	0	0	1		0	1
(	(	(	(	(	)	)	)	(	(	(	)	(	(	)	(	(	)	)	)	)	)	)	(	)	)
0	1	2	3	4	4	3	2	5	6	7	7	8	9	9	10	11	11	10	8	6	5	1	12	12	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Fig. 4. Enhanced BPR of the Super-Cartesian tree of Fig. 3. The lower row of numbers shows the positions of the parentheses in the sequence. The row below BPR is only for illustrative purposes: The opening parentheses are numbered consecutively and a closing parenthesis has the number of its matching opening parenthesis. The row above the BPR shows the bitstring B.

A few thoughts on the space of our data structure: For an array of n elements, there are $2n + 4$ parentheses and the bitstring has length $n + 2$. Note that the class of Super-Cartesian trees is isomorphic to the class of Schröder Trees, whose number is given by the n'th Small Schröder Number $\mathcal{C}_n$, and $\lg \mathcal{C}_n \approx 2.54n - \Theta(\lg n)$ [22]. Hence, although our $3n$-bit representation does not meet this lower bound, it is also not too far away from it.

Given a balanced parentheses sequence, the following operations can be supported in $O(1)$ time with $o(n)$ bits of extra space (see [17] and references therein): $rank_(i)$ returns the number of opening parentheses up to position i; $select_(i)$ returns the position of the i-th opening parenthesis; $findclose(i)$ returns the position of the closing parenthesis matching the opening parenthesis at position i; and $enclose(i)$ for an opening parenthesis at position i returns the position j of the opening parenthesis such that $(j, findclose(j))$ encloses $(i, findclose(i))$ most tightly. Operations $rank_)(i)$, $select_)(i)$, and $findopen(i)$ are defined analogously.

Let us denote the i-th opening parenthesis by ${}_i($ and the matching closing parenthesis by $)_i^b$, where b is its mark (bit in B). Note that ${}_i($ and $)_i^b$ occur at positions $ipos = select_(i)$ and $cipos = findclose(ipos)$ in BPR, respectively. Moreover, b occurs at position $bcipos = rank_)(cipos)$ in the bitstring B. Vice versa, given the position $bpos$ in B, its corresponding index in A can be determined by $rank_(findopen(select_)(bpos)))$.

Algorithm 2. Construction of the enhanced BPR of an array A.

```
push(0)          /* A[0] = −∞ */
B ← ε            /* bitstring B is initially empty */
write '('
for i ← 1 to n + 1 do
  while A[i] < A[top()] do
    λ ← pop()
    write ')'
    if A[λ] = A[top()] then append 0 to the bitstring B
    else append 1 to the bitstring B
  push(i) and write '('
write '))' and append 01 to the bitstring B        /* for A[0] and A[n + 1] */
```

Algorithm 3. Finding $\texttt{FEV}(i)$ and $\texttt{LEV}(i)$ in constant time.

$cipos \leftarrow findclose(select_{(}(i))$
$bcipos \leftarrow rank_{)}(cipos)$
if $\mathsf{BPR}[cipos-1] = \text{"("}$ **or** $(\mathsf{BPR}[cipos-1] = \text{")"}$ **and** $B[bcipos-1] = 1)$ **then**
  $\texttt{LEV}(i) \leftarrow i$
else /* $\mathsf{BPR}[cipos-1] = \text{")"}$ **and** $B[bcipos-1] = 0$ */
  $blpos \leftarrow select_1(rank_1(bcipos)-1)+1$
  $\texttt{LEV}(i) \leftarrow rank_{(}(findopen(select_{)}(blpos)))$
if $B[bcipos] = 1$ **then**
  $\texttt{FEV}(i) \leftarrow i$
else /* $B[bcipos] = 0$ */
  $brpos \leftarrow select_1(rank_1(bcipos)+1)$
  $\texttt{FEV}(i) \leftarrow rank_{(}(findopen(select_{)}(brpos)))$

The enhanced BPR has the following crucial property. If $j_1 < j_2 < \cdots < j_m$ are the indices in the mound-interval of i such that $A[i] = A[j_k]$, then $)^0_{j_m} \ldots)^0_{j_2})^1_{j_1}$ form a contiguous subsequence in BPR (note that the order is reversed, so that $\texttt{LEV}(i) = j_m$ is first and $\texttt{FEV}(i) = j_1$ is last), where the first $m-1$ closing parentheses are marked 0 and the last one is marked 1. This allows us to compute $\texttt{FEV}(i)$ and $\texttt{LEV}(i)$ by a case distinction (see Alg. 3): If $)^b_i$ is preceded by $_i($ or by $)^1_j$, then $\texttt{LEV}(i) = i$. Otherwise, $)^b_i$ is preceded by $)^0_j$. In this case, we search for the position $bpos$ of the first 1 in B that is left to $bcipos$ (to ensure that there is always such a 1, an additional 1 is added at the beginning of B). The index corresponding to $blpos = bpos + 1$ is $\texttt{LEV}(i)$. If $b = 1$ (i.e., $B[bcipos] = 1$), then $\texttt{FEV}(i) = i$. Otherwise, $b = 0$ and we search for the position $brpos$ of the first 1 in B that is right to $bcipos$. The index corresponding to $bcipos$ is $\texttt{FEV}(i)$.

As a matter of fact, Alg. 3 also allows us to determine the *number* of values that are equal to $A[i]$ in the mount interval of i: this is $brpos - blpos + 1$, and we can access each of them in constant time!

It follows from the construction of the enhanced BPR that the values $\texttt{PSV}(i)$ and $\texttt{NSV}(i)$ can also be computed in constant time:

- $\texttt{PSV}(i) = rank_{(}(enclose(select_{(}(\texttt{FEV}(i))))$
- $\texttt{NSV}(i) = rank_{(}(findclose(select_{(}(i))) + 1$

Furthermore, it is proved in [6] that a (general) RMQ can be computed in constant time on the BPR. However, the hidden constant is quite large, so that this operation is rather slow in practice. In other words, one should avoid its use whenever possible. Specific RMQ's (on mound-intervals) can be answered quicker as we shall see next. The proof of the following lemma is straightforward.

Lemma 1. *If $[i..j]$ is a mound-interval in array A, then $A[i-1] > A[j+1]$ if $\texttt{NSV}(i-1) = j+1$ and $A[i-1] \leq A[j+1]$ otherwise.*

At first glance, Lemma 1 is sort of weird. However, if the array A is compressed and access to its entries takes more time than the computation of $\texttt{NSV}(i-1)$, then a use of Lemma 1 makes perfect sense.

Lemma 2. *Let $[i..j]$ be a mound-interval in array A. Then,*

$$\texttt{RMQ}(i,j) = \begin{cases} rank_{(}(findopen(findclose(select_{(}(i-1))-1)), & \text{if } \texttt{NSV}(i-1)=j+1 \\ rank_{(}(findopen(select_{(}(j+1)-1)), & \text{otherwise} \end{cases}$$

Proof. Let $k = \texttt{RMQ}(i,j)$. If $\texttt{NSV}(i-1) = j+1$, then $A[i-1] > A[j+1]$ by Lemma 1. In this case, k is the right child of $i-1$ in the Super-Cartesian tree of the array A. In the BPR, $)_k$ is directly followed by $)_{i-1}$. Thus, $k = rank_{(}(findopen(findclose(select_{(}(i-1))-1))$. Otherwise, $A[i-1] \leq A[j+1]$. In this case, k is the left child of $j+1$ in the Super-Cartesian tree of A. In the BPR, $)_k$ is directly followed by $(_{j+1}$. Therefore, $k = rank_{(}(findopen(select_{(}(j+1)-1))$.

4 New Compressed Suffix Tree

We now show how to use the result from Sect. 3 for CSTs. Our basis is OG-CST [6], but we use the enhanced BPR for RMQ/PSV/NSV. It remains to show how to simulate the operations that access LCP in OG-CST; all other operations (including level ancestor queries, if needed) can be taken from [6,5]. Note that an LCP-interval is a mound-interval that includes its leftmost delimiting point.

- PARENT$([v_l..v_r])$ returns $\perp$ if $v = [v_l..v_r]$ is the root. If v is not the root, it returns $[\texttt{PSV}[k]..\texttt{NSV}[k]-1]$, where $k = v_l$ if $\texttt{NSV}[v_l] = v_r+1$ (i.e., $\texttt{LCP}[v_l] > \texttt{LCP}[v_r+1]$ by Lemma 1) and $k = v_r+1$ otherwise.
- FCHILD$([v_l..v_r])$ returns $\perp$ if $v_l = v_r$; otherwise it returns $[v_l..k-1]$, where $k = \texttt{RMQ}(v_l+1, v_r)$ is calculated according to Lemma 2.
- NSIBLING$([v_l..v_r])$ first determines $\texttt{NSV}(v_l)$. If $\texttt{NSV}(v_l) = v_r+1$, then $\texttt{LCP}[v_l] > \texttt{LCP}[v_r+1]$ by Lemma 1. In this case $\texttt{LCP}[v_l]$ is the last LCP-index of the parent interval; so it returns $\perp$ because $[v_l..v_r]$ has no sibling. Otherwise we know from $\texttt{LCP}[v_l] \leq \texttt{LCP}[v_r+1]$ that $i = v_r+1$ is an LCP-index of the parent interval. There is a succeeding LCP-index j iff $)_i$ is preceded by $)_j^0$. If so, it returns $[v_r+1..j-1]$, otherwise it returns $[v_r+1..\texttt{NSV}(v_r+1)-1]$.

Moreover, the enhanced BPR provides a new (practical) approach to compress the LCP-array: Because we can access the first LCP-index of an LCP-interval $[i..j]$ in constant time (starting from $[i..j]$ or an arbitrary LCP-index) and all LCP-indices have the same LCP-value ℓ, it suffices to store ℓ solely at the first LCP-index. Combining this with the byte-aligned LCP-array [13] yields LCP-arrays of size $4n$ to $6n$ bits on texts from `pizzachili.dcc.uchile.cl` with $t_{\texttt{LCP}} = O(1)$.

References

1. Munro, J.I., Raman, V., Rao, S.S.: Space efficient suffix trees. J. Algorithms 39(2), 205–222 (2001)
2. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)

3. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
4. Russo, L.M.S., Navarro, G., Oliveira, A.L.: Fully-compressed suffix trees. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 362–373. Springer, Heidelberg (2008)
5. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comput. Sci. 410(51), 5354–5364 (2009)
6. Ohlebusch, E., Gog, S.: A compressed enhanced suffix array supporting fast string matching. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 51–62. Springer, Heidelberg (2009)
7. Fischer, J.: Wee LCP. Inform. Process. Lett. 110(8-9), 117–120 (2010)
8. Välimäki, N., Mäkinen, V., Gerlach, W., Dixit, K.: Engineering a compressed suffix tree implementation. ACM J. Experimental Algorithmics 4, Article no.2 (2009)
9. Cánovas, R., Navarro, G.: Practical compressed suffix trees. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 94–105. Springer, Heidelberg (2010)
10. Gog, S., Fischer, J.: Advantages of shared data structures for sequences of balanced parentheses. In: Proc. DCC, pp. 406–415. IEEE Press, Los Alamitos (2010)
11. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article No. 2 (2007)
12. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. SODA, pp. 841–850. ACM/SIAM (2003)
13. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms 2(1), 53–86 (2004)
14. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
15. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
16. Sirén, J.: Sampled longest common prefix array. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 227–237. Springer, Heidelberg (2010)
17. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
18. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
19. Jansson, J., Sadakane, K., Sung, W.K.: Ultra-succinct representation of ordered trees. In: Proc. SODA, pp. 575–584. ACM/SIAM (2007)
20. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA, pp. 134–149. ACM/SIAM (2010)
21. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
22. Fischer, J., Heun, V.: Finding range minima in the middle: Approximations and applications. Mathematics in Computer Science 3(1), 17–30 (2010)
23. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. 38(6), 2162–2178 (2009)
24. Hon, W.K., Sadakane, K.: Space-economical algorithms for finding maximal unique matches. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 144–152. Springer, Heidelberg (2002)
25. Farzan, A., Raman, R., Rao, S.S.: Universal succinct representations of trees? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 451–462. Springer, Heidelberg (2009)

Succinct Representations of Dynamic Strings*

Meng He and J. Ian Munro

Cheriton School of Computer Science, University of Waterloo, Canada
{mhe,imunro}@uwaterloo.ca

Abstract. The `rank` and `select` operations over a string of length n from an alphabet of size σ have been used widely in the design of succinct data structures. In many applications, the string itself must be maintained dynamically, allowing characters of the string to be inserted and deleted. Under the word RAM model with word size $w = \Omega(\lg n)$, we design a succinct representation of dynamic strings using $nH_0 + o(n) \cdot \lg \sigma + O(w)$ bits to support `rank`, `select`, `insert` and `delete` in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time[1]. When the alphabet size is small, i.e. when $\sigma = O(\texttt{polylog}(n))$, including the case in which the string is a bit vector, these operations are supported in $O(\frac{\lg n}{\lg \lg n})$ time. Our data structures are more efficient than previous results on the same problem, and we have applied them to improve results on the design and construction of space-efficient text indexes.

1 Introduction

Succinct data structures provide solutions to reduce the storage cost of modern applications that process large data sets, such as web search engines, geographic information systems, and bioinformatics applications. First proposed by Jacobson [17], the aim is to encode a data structure using space close to the information-theoretic lower bound, while supporting efficient navigation operations in them. This approach was successfully applied to many abstract data types, including bit vectors [17,7,23], strings [14,3,13], binary relations [2,3], (unlabeled and labeled) trees [17,20,8,2,3,24], graphs [17,20,1] and text indexes [14,5,13].

A basic building block for most succinct data structures is the pair of operations `rank` and `select`. In particular, we require a highly space-efficient representation of a string S of length n over an alphabet of size σ to support the fast evaluation of the following operations:

- $\texttt{access}(S, i)$, which returns the character at position i in the string S;
- $\texttt{rank}_\alpha(S, i)$, which returns the number of occurrences of character α in $S[1..i]$;
- $\texttt{select}_\alpha(S, i)$, which returns the position of the i^{th} occurrence of character α in the string S.

* This work was supported by NSERC of Canada and the Canada Research Chairs program.

[1] $\lg n$ denotes $\log_2 n$.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 334–346, 2010.

This problem has many applications such as designing space-efficient text indexes [14,13], as well as representing binary relations [2,3], labeled trees [8,2,3] and labeled graphs [1]. The case in which the string is a bit vector whose characters are 0's and 1's (i.e. $\sigma = 2$) is even more fundamental: A bit vector supporting `rank` and `select` is a key structure used in several approaches of representing strings succinctly [14,2,3,19], and it is also used in perhaps most succinct data structures [17,20,8,2,3,1].

Due to the importance of strings and bit vectors, researchers have designed various succinct data structures for them [17,23,14,3,13] and achieved good results. For example, the data structure of Raman *et al.* [23] can encode a bit vector using $nH_0 + o(n)$ bits, where H_0 is the zero-order entropy of the bit vector[2], to support `access`, `rank` and `select` operations in constant time. Another data structure called wavelet tree proposed by Grossi *et al.* [14] can represent a string using $nH_0 + o(n) \cdot \lg \sigma$ bits to support `access`, `rank` and `select` in $O(\lg \sigma)$ time.

However, in many applications, it is not enough to have succinct static data structures that allow data to be retrieved efficiently, because data in these applications are also updated frequently. In the case of strings and bit vectors, the following two update operations are desired in many applications in addition to `access`, `rank` and `select`:

- $\texttt{insert}_\alpha(S, i)$, which inserts character α between $S[i-1]$ and $S[i]$;
- $\texttt{delete}(S, i)$, which deletes $S[i]$ from S.

In this paper, we design succinct representations of dynamic strings and bit vectors that are more efficient than previous results. We also present several applications to show how advancements on these fundamental problems yield improvements on other data structures.

1.1 Related Work

Blandford and Blelloch [4] considered the problem of representing ordered lists succinctly, and their result can be used to represent a dynamic bit vector of length n using $O(nH_0)$ bits to support the operations defined in Section 1 in $O(\lg n)$ time (note that $H_0 \leq 1$ holds for a bit vector). A different approach proposed by Chan *et al.* [5] can encode dynamic bit vectors using $O(n)$ bits to provide the same support for operations. Later Chan *et al.* [6] improved this result by providing $O(\lg n / \lg \lg n)$-time support for all these operations while still using $O(n)$ bits of space. Mäkinen and Navarro [19] reduced the space cost to $nH_0 + o(n)$ bits, but their data structure requires $O(\lg n)$ time to support operations. Recently, Sadakane and Navarro [24] designed a data structure for dynamic trees, and their main structure is essentially a bit vector that supports

[2] The zero-order (empirical) entropy of a string of length n over an alphabet of size σ is defined as $H_0 = \sum_{i=1}^{\sigma} (\frac{n_i}{n} \lg \frac{n}{n_i})$, where n_i is the number of times that the i^{th} character occurs in the string. Note that we always have $H_0 \leq \lg \sigma$. This definition also applies to a bit vector, for which $\sigma = 2$.

more types of operations. Their result can be used to represent a bit vector using $n + o(n)$ bits to support the operations we consider in $O(\lg n / \lg\lg n)$ time.

For the more general problem of representing dynamic strings of length n over alphabets of size σ, Mäkinen and Navarro [19] combined their results on bit vectors with the wavelet tree structure of Grossi *et al.* [14] to design a data structure of $nH_0 + o(n) \cdot \lg\sigma$ bits that supports `access`, `rank` and `select` in $O(\lg n \log_q \sigma)$ time, and `insert` and `delete` in $O(q \lg n \log_q \sigma)$ time for any $q = o(\sqrt{\lg n})$. Lee and Park [18] proposed another data structure of $n\lg\sigma + o(n) \cdot \lg\sigma$ to support `access`, `rank` and `select` in $O(\lg n(\frac{\lg\sigma}{\lg\lg n} + 1))$ worst-case time which is faster, but `insert` and `delete` take $O(\lg n(\frac{\lg\sigma}{\lg\lg n} + 1))$ amortized time. Finally, González and Navarro [13] improved the above two results by designing a structure of $nH_0 + o(n) \cdot \lg\sigma$ bits to support all the operations in $O(\lg n(\frac{\lg\sigma}{\lg\lg n} + 1))$ worst-case time.

Another interesting data structure is that of Gupta *et al.* [15]. For the same problems, they aimed at improving query time while sacrificing update time. Their bit vector structure occupies $nH_0 + o(n)$ bits and requires $O(\lg\lg n)$ time to support `access`, `rank` and `select`. It takes $O(\lg^\epsilon n)$ amortized time to support `insert` and `delete` for any constant $0 < \epsilon < 1$. Their dynamic string structure uses $n\lg\sigma + \lg\sigma(o(n) + O(1))$ bits to provide the same support for operations (when $\sigma = O(\texttt{polylog}(n))$, `access`, `rank` and `select` take $O(1)$ time).

1.2 Our Results

We adopt the word RAM model with word size $w = \Omega(\lg n)$. Our main result is a succinct data structure that encodes a string of length n over an alphabet of size σ in $nH_0 + o(n) \cdot \lg\sigma + O(w)$ bits to support `access`, `rank`, `select`, `insert` and `delete` in $O(\frac{\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all these operations can be supported in $O(\frac{\lg n}{\lg\lg n})$ time. Note that the $O(w)$ term in the space cost exists in all previous results, and we omit them in Section 1.1 for simplicity of presentation (in fact many papers simply ignore them). Our structure can also encode a bit vector of length n in $nH_0 + o(n) + O(w)$ bits to support the same operations in $O(\frac{\lg n}{\lg\lg n})$ time, matching the lower bound in [12]. Our solutions are currently the best to the problem, for both the general case and the special case in which the alphabet size is $O(\texttt{polylog}(n))$ or 2 (i.e. the string is a bit vector). The only previous result that is not comparable is that of Gupta *et al.* [15], since their solution is designed under the assumption that the string is queried frequently but updated infrequently.

We also apply the above results to design a succinct text index for a dynamic text collection to support text search, and the problem of reducing the required amount of working space when constructing a text index. Our dynamic string representation allows us to improve previous results on these problems [19,13].

2 Preliminaries

Searchable Partial Sums. Raman *et al.* [22] considered the problem of representing a dynamic sequence of integers to support `sum`, `search` and `update` operations. To achieve their main result, they designed a data structure for the following special case in which the length of the sequence is small, and we will use it to encode information stored as small sequences of integers in our data structures:

Lemma 1. *There is a data structure that can store a sequence, Q, of $O(\lg^\epsilon n)$ nonnegative integers of $O(\lg n)$ bits each*[3]*, for any constant $0 \leq \epsilon < 1$, using $O(\lg^{1+\epsilon} n)$ bits to support the following operations in $O(1)$ time:*

- `sum`(Q, i)*, which computes* $\sum_{j=1}^{i} Q[j]$*;*
- `search`(Q, x)*, which returns the smallest i such that* `sum`$(Q, i) \geq x$*;*
- `update`(Q, i, δ)*, which updates $Q[i]$ to $Q[i] + \delta$, where $|\delta| \leq \lg n$.*

The data structure can be constructed in $O(\lg^\epsilon n)$ time, and it requires a precomputed universal table of size $O(n^{\epsilon'})$ bits for any fixed $\epsilon' > 0$.

Collections of Searchable Partial Sums. A key structure in the dynamic string representation of González and Navarro [13] is a data structure that maintains a set of sequences of nonnegative integers, such that `sum`, `search` and `update` can be supported on any sequence efficiently, while `insert` and `delete` are performed simultaneously on all the sequences at the same given position, with the restriction that only 0's can be inserted or deleted. More precisely, let $C = Q_1, Q_2, \cdots, Q_d$ to be a set of dynamic sequences, and each sequence, Q_j, has n nonnegative integers of $k = O(\lg n)$ bits each. The *collection of searchable partial sums with insertions and deletions (CSPSI)* problem is to encode C to support:

- `sum`(C, j, i), which computes $\sum_{p=1}^{i} Q_j[p]$;
- `search`(C, j, x), which returns the smallest i such that `sum`$(C, j, i) \geq x$;
- `update`(C, j, i, δ), which updates $Q_j[i]$ to $Q_j[i] + \delta$;
- `insert`(C, i), which inserts 0 between $Q_j[i-1]$ and $Q_j[i]$ for all $1 \leq j \leq d$;
- `delete`(C, i), which deletes $Q_j[i]$ from sequence Q_j for all $1 \leq j \leq d$, and to perform this operation, $Q_j[i] = 0$ must hold for all $1 \leq j \leq d$.

González and Navarro [13] designed a data structure of $kdn(1 + O(\frac{1}{\sqrt{\lg n}} + \frac{d}{\lg n}))$ bits to support all the above operations in $O(d + \lg n)$ time. This structure becomes succinct (i.e. using $dk(n + o(n))$ bits if $d = o(\lg n)$, when the operations can be supported in $O(\lg n)$ time. A careful reading of their technique shows that these results only work under the word RAM model with word size $w = \Theta(\lg n)$ (see our discussions after Lemma 5). We improve this data structure for small d, which is further used to design our succinct string representations.

[3] Raman *et al.* [22] required each integer to fit in one word. However, it is easy to verify that their proof is still correct if each integer requires $O(\lg n)$ bits, i.e. each integer can require up to a constant number of words to store.

3 Collections of Searchable Partial Sums

We follow the main steps of the approach of González and Navarro [13] to design a succinct representation of dynamic strings, but we make improvements in each step. We first improve their result for the CSPSI problem (Section 3), and then combine it with other techniques to improve their data structure for strings over small alphabets (Section 4). Finally, we extend the result on small alphabets to general alphabets (Section 5). Our main strategy of achieving these improvements is to divide the sequences into superblocks of appropriate size, and store them in the leaves of a B-tree (instead of the red-black tree in [13]). Similar ideas were applied to data structures for balanced parentheses [6,24]. Our work is the first that successfully adapts it to integer sequences and character strings, and we have created new techniques to overcome some difficulties. Proofs omitted from this paper due to space constraints can be found in the full version [16].

In this section, we consider the CSPSI problem defined in section 2. We assume that $d = O(\lg^{\eta} n)$ for any constant $0 < \eta < 1$, and for the operation $\texttt{update}(C, j, i, \delta)$, we assume $|\delta| \leq \lg n$. Under these assumptions, we improve the result in [13] under the word RAM model with word size $\Omega(\lg n)$.

Data Structures. Our main data structure is a B-tree constructed over the given collection C. Let $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. Each leaf of this B-tree stores a *superblock* whose size is between (and including) $L/2$ and $2L$ bits, and each superblock stores the same number of integers from each sequence in C. More precisely, the content of the leftmost leaf is $Q_1[1..s_1]Q_2[1..s_1]\cdots Q_d[1..s_1]$, the content of the second leftmost leaf is $Q_1[s_1+1..s_2]Q_2[s_1+1..s_2]\cdots Q_d[s_1+1..s_2]$, and so on, and the indices $s_1, s_2, \cdots$ satisfy the following conditions because of requirement on the sizes of superblocks: $L/2 \leq s_1 kd \leq 2L, L/2 \leq (s_2 - s_1)kd \leq 2L, \cdots$.

Let $f = \lg^{\lambda} n$, where λ is a positive constant number less than 1 that we will fix later. Each internal node of the B-tree we construct has at least f and at most $2f$ children. We store the following $d+1$ sequences of integers for each internal node v (let h be the number of children of v):

- A sequence $P(v)[1..h]$, in which $P(v)[i]$ is the number of positions stored in the leaves of the subtree rooted at the i^{th} child of v for any sequence in C (note that this number is the same for all sequences in C);
- A sequence $R_j(v)[1..h]$ for each $j = 1, 2, \cdots, d$, in which $R_j(v)[i]$ is the sum of the integers from sequence Q_j that are stored in the leaves of the subtree rooted at the i^{th} child of v.

We use Lemma 1 to encode each of the $d+1$ sequences of integers for v.

We further divide each superblock into *blocks* of $\lceil \lceil \lg n \rceil^{3/2} \rceil$ bits each, and maintain the blocks for the same superblock using a linked list. Only the last block in the list can be partially full; any other block uses all its bits to store the data encoded in the superblock. This is how we store the superblocks physically.

To analyze the space cost of the above data structures, we have:

Lemma 2. *The above data structures occupy $kd(n+o(n))$ bits if the parameters λ and η satisfy $0 < \lambda < 1 - \eta$.*

Supporting `sum`, `search` *and* `update`. We discuss these three operations first because they do not change the size of C.

Lemma 3. *The data structures in this section can support* `sum`, `search` *and* `update` *in* $O(\frac{\lg n}{\lg\lg n})$ *time with an additional universal table of* $o(n)$ *bits.*

Proof. To support $\texttt{sum}(C, j, i)$, we perform a top-down traversal in the B-tree. In our algorithm, we use a variable r that is initially 0, and its value will increase as we go down the tree. We have another variable s whose initial value is i. Initially, let v be the root of the tree. As $P(v)$ stores the number of positions stored in the subtrees rooted at each child of v, the subtree rooted at the c^{th} child of v, where $c = \texttt{search}(P(v), i)$, contains position i. We also compute the sum of the integers from the sequence Q_j that are stored in the subtrees rooted at the left siblings of the c^{th} child of v, which is $y = \texttt{sum}(R_j(v), c-1)$, and we increase the value of r by y. We then set v to be its c^{th} child, decrease the value of s by $\texttt{sum}(P(v), c-1)$, and the process continues until we reach a leaf. At this time, r records the sum of the integers from the sequence Q_j that are before the first position stored in the superblock of the leaf we reach. As the height of this B-tree is $O(\frac{\lg n}{\lg\lg n})$, and the computation at each internal node takes constant time by Lemma 1, it takes $O(\frac{\lg n}{\lg\lg n})$ time to locate this superblock.

It now suffices to compute the sum of the first s integers from sequence Q_j that are stored in the superblock. This can be done by first going to the block storing the first integer in the superblock that is from Q_j, which takes $O(\frac{\sqrt{\lg n}}{\lg\lg n})$ time (recall that each block is of fixed size and there are $O(\frac{\sqrt{\lg n}}{\lg\lg n})$ of them in a superblock), and then read the sequence in chunks of $\lceil \frac{1}{2}\lg n \rceil$ bits. For each $\lceil \frac{1}{2}\lg n \rceil$ bits we read, we use a universal table A_1 to find out the sum of the $z = \lfloor \lceil \frac{1}{2}\lg n \rceil / k \rfloor$ integers stored in it in $O(1)$ time (the last $a = \lceil \frac{1}{2}\lg n \rceil \bmod k$ bits in this block are concatenated with the next $\lceil \frac{1}{2}\lg n \rceil - a$ bits read for table lookup). This table simply stores the result for each possible bit strings of length $\lceil \frac{1}{2}\lg n \rceil$. The last chunk we read may contain integers after $Q_j[i]$. To address the problem, we augment A_1 so that it is a two dimensional table $A_1[1..2^{\lceil \frac{1}{2}\lg n \rceil}][1..z]$, in which $A[b][g]$ stores for the b^{th} lexicographically smallest bit vector of length $\lceil \frac{1}{2}\lg n \rceil$, the sum of the first g integers of size k stored in it. This way the computation in the superblock can be done in $O(\frac{\lg n}{\lg\lg n})$ time, and thus $\texttt{sum}(C, j, i)$ can be supported in $O(\frac{\lg n}{\lg\lg n})$ time. The additional data structure we require is table A_1, which occupies $O(2^{\lceil \frac{1}{2}\lg n \rceil} \times \lfloor \lceil \frac{1}{2}\lg n \rceil / k \rfloor \times \lg n) = O((\sqrt{n}\lg^2 n)/k)$ bits. The operations `search` and `update` can be supported in a similar manner. □

Supporting `insert` *and* `delete`. We give a proof sketch of the following lemma on supporting `insert` and `delete`:

Lemma 4. *When* $w = \Theta(\lg n)$*, the data structures in this section can support* `insert` *and* `delete` *in* $O(\frac{\lg n}{\lg\lg n})$ *amortized time.*

Proof (sketch). To support $\texttt{insert}(C, i)$, we locate the leaf containing position i as we do for `sum`, updating $P(v)$'s along the way. We insert a 0 before the

i^{th} position of all the sequences by creating a new superblock, copying the data from the old superblock contained in this leaf to this new superblock in chunks of size $\lceil \lg n \rceil$, and adding 0's at appropriate positions when we copy. This takes $O(\frac{\lg n}{\lg \lg n} + d) = O(\frac{\lg n}{\lg \lg n})$ time. If the size of the new superblock exceeds $2L$, we split it into two superblocks of roughly the same size. The parent of the old leaf becomes the parent, v, of both new leaves, and we reconstruct the data structures for $P(v)$ and $R_j(v)$'s in $O(df) = o(\frac{\lg n}{\lg \lg n})$ time. This may make a series of internal nodes to overflow, and in the amortized sense, each split of the leaf will only cause a constant number of internal nodes to overflow. Thus we can support `insert` in $O(\frac{\lg n}{\lg \lg n})$ amortized time. The support for `delete` is similar.

Each `insert` or `delete` changes n by 1. This might change the value $\lceil \lg n \rceil$, which will in turn affect L, the size of blocks, and the content of A_1. As $w = \Theta(\lg n)$, L and the block size will only change by a constant factor. Thus if we do not change these parameters, all our previous space and time analysis still applies. The $o(n)$ time required to reconstruct A_1 each time $\lceil \lg n \rceil$ changes can be charged to at least $\Theta(n)$ `insert` or `delete` operations. □

As we use a B-tree, a new problem is to deamortize the support for `insert` and `delete`. We also need to consider the case in which the word size is $w = \Omega(\lg n)$. The following lemma presents our solution to the CSPSI problem:

Lemma 5. *Consider a collection, C, of d sequences of n nonnegative integers each ($d = O(\lg^{\eta} n)$ for any constant $0 < \eta < 1$), in which each integer requires k bits. Under the word RAM model with word size $\Omega(\lg n)$, C can be represented using $O(kdn + w)$ bits to support* `sum`, `search`, `update`, `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time with a buffer of $O(n \lg n)$ bits (for the operation* `update`*(C, j, i, δ), we assume $|\delta| \leq \lg n$).*

Proof (sketch). To deamortize the algorithm for `insert` and `delete`, we first observe that the table A_1 can be built incrementally each time we perform `insert` and `delete`. Thus the challenging part is to re-balance the B-tree (i.e. to merge and split its leaves and internal nodes) after insertion and deletion. For this we use the *global rebuilding* approach of Overmars and van Leeuwen [21]. By their approach, if there exist two constant numbers $c_1 > 0$ and $0 < c_2 < 1$ such that after performing $c_1 n$ insertions and/or $c_2 n$ deletions without re-balancing the B-tree, we can still perform query operations in $O(\frac{\lg n}{\lg \lg n})$ time, and if the B-tree can be rebuilt in $O(f(n) \times n)$ time, we can support insertion or deletion in $O(\frac{\lg n}{\lg \lg n} + f(n))$ worse-case time using additional space proportional to the size of our original data structures and a buffer of size $O(n \lg n)$ bits. We first note that if we do not re-balance the B-tree after performing `delete` $c_2 n$ times for any $0 < c_2 < 1$, the time required to answer a query will not change asymptotically. This is however different for `insert`, and we use the approach of Fleischer [11] as in [24]. Essentially, in his approach, at most one internal node and one leaf is split after each insertion, which guarantees that the degree of any internal node does not exceed $4f$. This way after $\Theta(n)$ insertions, query operations can still be performed in $O(\frac{\lg n}{\lg \lg n})$ time. Finally, it takes $O(nd)$ time to construct the B-tree, so we can support `insert` and `delete` in $O(d + \frac{\lg n}{\lg \lg n}) = O(\frac{\lg n}{\lg \lg n})$ time.

To reduce the space overhead when $w = \omega(\lg n)$, we allocate a memory block whose size is sufficient for the new structure until another structure has to be built, and this increases the space cost by a constant factor. Then we can still use pointers of size $O(\lg n)$ bits (not $O(w)$ bits), and $O(w)$ bits are required to record the address of memory blocks allocated. □

Section 2 states that González and Navarro [13] designed a data structure of $kdn(1 + O(\frac{1}{\sqrt{\lg n}} + \frac{d}{\lg n}))$ bits. This is more compact, but it only works for the special case in which $w = \Theta(\lg n)$. González and Navarro [13] actually stated that their result would work when $w = \Omega(\lg n)$. This requires greater care than given in their paper. Their strategy is to adapt the approach of Mäkinen and Navarro [19] developed originally for a dynamic bit vector structure. To use it for the CSPSI problem, they split each sequence into three subsequences. The split points are the same for all the sequences in C. The set of left, middle, and right subsequences constitute three collections, and they build CSPSI structures for each of them. For each insertion and deletion, a constant number of elements is moved from one collection to another, which will eventually achieve the desired result with other techniques. Moving one element from one collection to another means that the first or the last integers of all the subsequences in one collection must be moved to the subsequences in another collection. However, their CSPSI structure only supports insertions and deletions of 0's at the same position in all subsequences in $O(\lg n)$ time, so moving one element cannot be supported fast enough. Thus their structure only works when $w = \Theta(\lg n)$. Their result can be generalized to the case in which $w = \Omega(\lg n)$ using the approach in Lemma 5, but the space will be increased to $O(kdn + w)$ bits and a buffer will be required.

4 Strings over Small Alphabets

In this section, we consider representing a dynamic string $S[1..n]$ over an alphabet of size $\sigma = O(\sqrt{\lg n})$ to support `access`, `rank`, `select`, `insert` and `delete`.

Data Structures. Our main data structure is a B-tree constructed over S. We again let $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. Each leaf of this B-tree contains a *superblock* that has at most $2L$ bits. We say that a superblock is *skinny* if it has fewer than L bits. The string S is initially partitioned into substrings, and each substring is stored in a superblock. We number the superblocks consecutively from left to right starting from 1. Superblock i stores the i^{th} substring from left to right. To bound the number of leaves and superblocks, we require that there do not exist two consecutive skinny superblocks. Thus there are $O(\frac{n \lg \sigma}{L})$ superblocks.

Let $b = \sqrt{\lg n}$, and we require that the degree of each internal node of the B-tree is at least b and at most $2b$. For each internal node v, we store the following data structures encoded by Lemma 1 (let h be the number of children of v):

- A sequence $U(v)[1..h]$, in which $U(v)[i]$ is the number of superblocks contained in the leaves of the subtree rooted at the i^{th} child of v;

– A sequence $I(v)[1..h]$, in which $I(v)[i]$ stores the number of characters stored in the leaves of the subtree rooted at the i^{th} child of v.

As in Section 3, each superblock is further stored in a list of blocks of $\lceil \lceil \lg n \rceil^{3/2} \rceil$ bits each, and only the last block in each list can have free space.

Finally for each character α, we construct an integer sequence $E_\alpha[1..t]$ in which $E_\alpha[i]$ stores the number of occurrences of character α in superblock i (t denotes the number of superblocks). We create σ integer sequences in this way, and we construct a CSPSI structure, E, for them using Lemma 5. Note that the buffered required in Lemma 5 to support operations on E takes $o(n)$ bits here as the length of the sequences in E is $O(n/L)$.

The above data structures occupy $n \lg \sigma + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}})$ bits.

Supporting `access`*,* `rank` *and* `select`*.* We first support query operations.

Lemma 6. *The data structures in this section can support* `access`*,* `rank` *and* `select` *in* $O(\frac{\lg n}{\lg \lg n})$ *time with a universal table of* $O(\sqrt{n}\texttt{polylog}(n))$ *bits.*

Proof. To support $\texttt{access}(S, i)$, we perform a top-down traversal in the B-tree to find the leaf containing $S[i]$. During this traversal, at each internal node v, we perform `search` on $I(v)$ to decide which child to traverse, and perform `sum` on $I(v)$ to update the value i. When we find the leaf, we follow the pointers to find the block containing the position we are looking for, and then retrieve the corresponding character in constant time. Thus `access` takes $O(\frac{\lg n}{\lg \lg n})$ time.

To compute $\texttt{rank}_\alpha(S, i)$, we first locate the leaf containing position i using the same process for `access`. Let j be the number of the superblock contained in this leaf, which can be computed using $U(v)$ during the top-down traversal. Then $\texttt{sum}(E, \alpha, j-1)$ is the number of occurrences of α in superblocks $1, 2, \cdots j-1$, which can be computed in $O(\frac{\lg n}{\lg \lg n})$ time by Lemma 5. To compute the number of occurrences of α up to position i inside superblock j, we read the content of this superblock in chunks of size $\lceil \frac{1}{2} \lg n \rceil$ bits. As with the support for `sum` in the proof of Lemma 3, this can be done in $O(\frac{\lg n}{\lg \lg n})$ time using a precomputed table A_2 of $O(\sqrt{n}\texttt{polylog}(n))$ bits. Our support for $\texttt{select}_\alpha(S, i)$ is similar. □

Supporting `insert` *and* `delete`*.* Careful tuning of the techniques for supporting insertions and deletions for the CSPSI problem yields the following lemma:

Lemma 7. *When* $w = \Theta(\lg n)$*, the data structures in this section can support* `insert` *and* `delete` *in* $O(\frac{\lg n}{\lg \lg n})$ *amortized time.*

To deamortize the support for `insert` and `delete`, we cannot directly use the global rebuilding approach of Overmars and van Leeuwen [21] here, since we do not want to increase the space cost of our data structures by a constant factor, and the use of a buffer is also unacceptable. Instead, we design an approach that deamortizes the support for `insert` and `delete` completely, and differently from the original global rebuilding approach of Overmars and van Leeuwen [21], our approach neither increases the space cost by a constant factor, nor requires

any buffer. We thus call our approach *succinct global rebuilding*. This approach still requires us to modify the algorithms for `insert` and `delete` so that after $c_1 n$ insertions ($c_1 > 0$) and $c_2 n$ deletions ($0 < c_2 < 1$), a query operation can still be supported in $O(\frac{\lg n}{\lg \lg n})$ time. We also start the rebuilding process when the number of `insert` and `delete` operations performed exceeds half the initial length of the string stored in the data structure. The main difference between our approach and the original approach in [21] is that during the process of rebuilding, we never store two copies of the same data, i.e. the string S. Instead, our new structure stores a prefix, S_p, of S, and the old data structure stores a suffix, S_s, of S. During the rebuilding process, each time we perform an insertion or deletion, we perform such an operation on either S_p or S_s. After that, we remove the first several characters from S_s, and append them to S_p. By choosing parameters and tuning our algorithm carefully, we can finish rebuilding after at most $n_0/3$ update operations, where n_0 is the length of S when we start rebuilding. During this process, we use both old and new data structures to answer queries in $O(\frac{\lg n}{\lg \lg n})$ time.

To reduce the space overhead when $w = \omega(\lg n)$, we adapt the approach of Mäkinen and Navarro [19]. We finally use the approach of González and Navarro [13] to compress our representation. Since their approach is only applied to the superblocks (i.e. it does not matter what kind of tree structures are used, since additional tree arrangement operations are not required when the number of bits stored in a leaf is increased due to a single update in their solution), we can use it here directly. Thus we immediately have:

Lemma 8. *Under the word RAM model with word size $w = \Omega(\lg n)$, a string S of length n over an alphabet of size $\sigma = O(\sqrt{\lg n})$ can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* `access`, `rank`, `select`, `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time.*

5 Strings over General Alphabets

We follow the approach of Ferragina *et al.* [10] that uses a generalized wavelet tree to extend results on strings over small alphabets to general alphabets. Special care must be taken to avoid increasing the $O(w)$-bit term in Lemma 8 asymptotically. We now present our main result:

Theorem 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a string S of length n over an alphabet of size σ can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* `access`, `rank`, `select`, `insert` *and* `delete` *operations in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all the operations can be supported in $O(\frac{\lg n}{\lg \lg n})$ time.*

The following corollary is immediate:

Corollary 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a bit vector of length n can be represented using $nH_0 + o(n) + O(w)$ bits to support* `access`, `rank`, `select`, `insert` *and* `delete` *in $O(\frac{\lg n}{\lg \lg n})$ time.*

6 Applications

Dynamic Text Collection. Mäkinen and Navarro [19] showed how to use a dynamic string structure to index a text collection N to support string search. For this problem, n denotes the length of the text collection N when represented as a single string that is the concatenations of all the text strings in the collection (a separator is inserted between texts). González and Navarro [13] improved this result in [19] by improving the string representation. If we use our string structure, we directly have the following lemma, which improves the running time of the operations over the data structure in [13] by a factor of $\lg\lg n$:

Theorem 2. *Under the word RAM model with word size $w = \Omega(\lg n)$, a text collection N of size n consisting of m text strings over an alphabet of size σ can be represented in $nH_h + o(n)\cdot\lg\sigma + O(m\lg n + w)$ bits*[4] *for any $h \le (\alpha\log_\alpha n) - 1$ and any constant $0 < \alpha < 1$ to support:*

- *the counting of the number of occurrences of a given query substring P in N in $O(\frac{|P|\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time;*
- *After counting, the locating of each occurrence in $O(\lg^2 n(\frac{1}{\lg\lg n} + \frac{1}{\lg\sigma}))$ time;*
- *Inserting and deleting a text T in $O(\frac{|T|\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time;*
- *Displaying any substring of length l of any text string in N in $O(\lg^2 n(\frac{1}{\lg\lg n} + \frac{1}{\lg\sigma}) + \frac{l\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time.*

Compressed Construction of Text Indexes. Researchers have designed space-efficient text indexes whose space is essentially a compressed version of the given text, but the construction of these text indexes may still require a lot of space. Mäkinen and Navarro [19] used their dynamic string structure to construct a variant of FM-index [9] using as much space as what is required to encode the index. Their result was improved by González and Navarro [13], and the construction time can be further improved by a factor of $\lg\lg n$ using our structure:

Theorem 3. *A variant of a FM-index of a text string $T[1..n]$ over an alphabet of size σ can be constructed using $nH_h + o(n)\cdot\lg\sigma$ bits of working space in $O(\frac{n\lg n}{\lg\lg n}(\frac{\lg\sigma}{\lg\lg n} + 1))$ time for any $h \le (\alpha\log_\alpha n) - 1$ and any constant $0 < \alpha < 1$.*

7 Concluding Remarks

In this paper, we have designed a succinct representation of dynamic strings that provide more efficient operations than previous results, and we have successfully applied it to improve previous data structures on text indexing. As a string structure supporting rank and select is used in the design of succinct representations of many data types, we expect our data structure to play an important role in the future research on succinct dynamic data structures.

[4] H_h is the h^{th}-order entropy of the text collection when represented as a single string.

We have also created some new techniques to achieve our results. We particularly think that the approach of succinct global rebuilding is interesting, and expect it to be useful for deamortizing algorithms on other succinct data structures.

References

1. Barbay, J., Castelli Aleardi, L., He, M., Munro, J.I.: Succinct representation of labeled graphs. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 316–328. Springer, Heidelberg (2007)
2. Barbay, J., Golynski, A., Munro, J.I., Rao, S.S.: Adaptive searching in succinctly encoded binary relations and tree-structured documents. Theoretical Computer Science 387(3), 284–297 (2007)
3. Barbay, J., He, M., Munro, J.I., Rao, S.S.: Succinct indexes for strings, binary relations and multi-labeled trees. In: SODA, pp. 680–689 (2007)
4. Blandford, D.K., Blelloch, G.E.: Compact representations of ordered sets. In: SODA, pp. 11–19 (2004)
5. Chan, H.L., Hon, W.K., Lam, T.W.: Compressed index for a dynamic collection of texts. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 445–456. Springer, Heidelberg (2004)
6. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
7. Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: SODA, pp. 383–391 (1996)
8. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM 57(1) (2009)
9. 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)
10. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
11. Fleischer, R.: A simple balanced search tree with o(1) worst-case update time. Int. J. Found. Comput. Sci. 7(2), 137–150 (1996)
12. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: STOC, pp. 345–354 (1989)
13. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science 410(43), 4414–4422 (2009)
14. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850 (2003)
15. Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: A framework for dynamizing succinct data structures. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 521–532. Springer, Heidelberg (2007)
16. He, M., Munro, J.I.: Succinct representations of dynamic strings. CoRR abs/1005.4652 (2010)
17. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS, pp. 549–554 (1989)
18. Lee, S., Park, K.: Dynamic rank/select structures with applications to run-length encoded texts. Theoretical Computer Science 410(43), 4402–4413 (2009)
19. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3) (2008)

20. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
21. Overmars, M.H., van Leeuwen, J.: Worst-case optimal insertion and deletion methods for decomposable searching problems. Inf. Process. Lett. 12(4), 168–173 (1981)
22. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)
23. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), 43 (2007)
24. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA, pp. 134–149 (2010)

Computing Matching Statistics and Maximal Exact Matches on Compressed Full-Text Indexes

Enno Ohlebusch, Simon Gog, and Adrian Kügel

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

Abstract. Exact string matching is a problem that computer programmers face on a regular basis, and full-text indexes like the suffix tree or the suffix array provide fast string search over large texts. In the last decade, research on compressed indexes has flourished because the main problem in large-scale applications is the space consumption of the index. Nowadays, the most successful compressed indexes are able to obtain almost optimal space and search time simultaneously. It is known that a myriad of sequence analysis and comparison problems can be solved efficiently with established data structures like the suffix tree or the suffix array, but algorithms on compressed indexes that solve these problem are still lacking at present. Here, we show that *matching statistics* and *maximal exact matches* between two strings S^1 and S^2 can be computed efficiently by matching S^2 backwards against a compressed index of S^1.

1 Introduction

The suffix tree of a string S of length n is an index structure that can be computed and stored in $O(n)$ time and space; see the seminal paper of Weiner [1]. 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 (about $20n$ bytes). This is a drawback in actual implementations. Thus, nowadays many string algorithms are based on suffix arrays instead. The suffix array specifies the lexicographic ordering of all suffixes of S, and it was introduced by Manber and Myers [4]. Almost a decade ago, Ferragina and Manzini [5] invented the FM-index, which is based on the Burrows-Wheeler transform [6] and the LF-mapping, and showed that it is possible to search a pattern backwards in the suffix array SA of string S using the FM-index instead of SA. In contrast to traditional data structures like the suffix tree, this compressed index supports backward search much better than forward search. Needless to say that one needs new algorithms to exploit this. In this paper, we present the first algorithms for computing matching statistics and maximal exact matches that use backward search instead of forward search. Matching statistics were introduced by Chang and Lawler [7] to solve the approximate string matching problem. Among other things, they were used in the computation of string kernels [8] and the design of DNA chips [9]. Matching statistics can also be used in a space-efficient computation of longest common substrings [3, Section 7.9]

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 347–358, 2010.

i	SA	LCP	PSV	NSV	BWT	$S_{\mathsf{SA}[i]}$
1	11	-1			t	\$
2	3	0	1	12	c	$aaacatat$\$
3	4	2	2	4	a	$aacatat$\$
4	1	1	2	8	\$	$acaaacatat$\$
5	5	3	4	6	a	$acatat$\$
6	9	1	2	8	t	at\$
7	7	2	6	8	c	$atat$\$
8	2	0	1	12	a	$caaacatat$\$
9	6	2	8	10	a	$catat$\$
10	10	0	1	12	a	t\$
11	8	1	10	12	a	tat\$
12		-1				

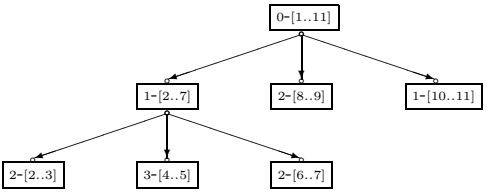


Fig. 1. Left: The suffix array of the string $S = acaaacatat\$$ and auxiliary arrays. Right: The lcp-interval tree (without singleton intervals) of the lcp-array.

and in the preprocessing phase of an algorithm that determines longest common prefix queries in constant time [3, Section 9.1]. Maximal exact matches play a key role in genome-genome comparisons (see e.g. [10,11]) and recently they were used to seed alignments of high-throughput reads for genome assembly. Our experiments show that the new algorithms outperform the current state-of-the-art algorithms based on forward search.

2 Preliminaries

Let Σ be an ordered alphabet whose smallest element is the so-called sentinel character \$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the ith 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 $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. In the following, SA^{-1} denotes the inverse of the permutation SA. In 2003, it was shown independently and contemporaneously by three research groups that a direct linear time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms are known; see [12].

The Burrows and Wheeler transform converts a string S into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. In virtually all cases, the Burrows-Wheeler transformed string compresses much better than the original string; see [6]. The permutation LF, defined by $LF(i) = \mathsf{SA}^{-1}[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$

Algorithm 1. Given $c \in \Sigma$ and an ω-interval $[i..j]$, $backwardSearch(c, [i..j])$ returns the $c\omega$-interval if it exists, and $\perp$ otherwise

```
backwardSearch(c, [i..j])
  i ← C[c] + Occ(c, i − 1) + 1
  j ← C[c] + Occ(c, j)
  if i ≤ j then return [i..j]
  else return ⊥
```

and $LF(i) = 1$ otherwise, is called LF-mapping. The LF-mapping can be implemented by

$$LF(i) = C[c] + Occ(c, i), \text{ where } c = \mathsf{BWT}[i],$$

$C[c]$ is the overall number (of occurrences) of characters in S which are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$.

In the following, the ω-interval of a substring ω of S is the interval $[i..j]$ in the suffix array SA such that ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leq k \leq j$, but ω is not a prefix of any other suffix of S. For example, the ca-interval in the suffix array of Fig. 1 is the interval [8..9]. Ferragina and Manzini [5] showed that it is possible to search a pattern character-by-character backwards in the suffix array SA of string S, without storing SA; see Algorithm 1.

Searching backwards in the string $S = acaaacatat\$$ for the pattern ca works as follows. By definition, backward search for the last character of the pattern starts with the ε-interval $[1..n]$, where ε denotes the empty string. In our example $n = 11$ and $backwardSearch(a, [1..11])$ returns the a-interval [2..7] because $C[a] + Occ(a, 1 - 1) + 1 = 1 + 0 + 1 = 2$ and $C[a] + Occ(a, 11) = 1 + 6 = 7$. Similarly, $backwardSearch(c, [2..7])$ delivers the ca-interval [8..9] because $C[c] + Occ(c, 2 - 1) + 1 = 7 + 0 + 1 = 8$ and $C[c] + Occ(c, 7) = 7 + 2 = 9$.

A space efficient data structure that supports backward search and the LF-mapping (plus a certain navigational operation in the lcp-interval tree as detailed below) will be called *compressed full-text index* in this paper. In our implementation, we use the wavelet tree of Grossi et al. [13] but there are alternatives; see the review paper of Navarro and Mäkinen [14] for details. With the wavelet tree, both a backward search step and the computation of $LF(i)$ take constant time; see [13].[1] The wavelet tree uses $n \log |\Sigma| + o(n \log |\Sigma|)$ bits of space.

The suffix array SA is often enhanced with the so-called lcp-array LCP containing the lengths of longest common prefixes between consecutive suffixes in SA; see Fig. 1. Formally, the lcp-array is an array such that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n + 1]$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings u and v. Kasai et al. [15] showed that the lcp-array can be computed in linear time from the suffix array and

[1] Strictly speaking, it takes $\mathcal{O}(\log |\Sigma|)$ time, but here we assume a constant alphabet. In fact it is possible to get rid of the $\log |\Sigma|$ factor, trading space for time; see [14].

its inverse. Sadakane [16] describes an encoding of the lcp-array that uses only $2n+o(n)$ bits. Abouelhoda et al. [17] introduced the concept of lcp-intervals. An interval $[i..j]$, where $1 \leq i < j \leq n$, in an lcp-array LCP 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. $\mathsf{LCP}[j+1] < \ell$.

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); see Fig. 1.

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 parent interval of an lcp-interval $[i..j] \neq [1..n]$ with $\mathsf{LCP}[i] = p$ and $\mathsf{LCP}[j+1] = q$ can be determined as

$$parent([i..j]) = \begin{cases} p\text{-}[\mathsf{PSV}[i]..\mathsf{NSV}[i]-1] & , \text{if } p \geq q \\ q\text{-}[\mathsf{PSV}[j+1]..\mathsf{NSV}[j+1]-1], & \text{if } p < q \end{cases}$$

where, for any index $2 \leq i \leq n$,

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

3 Matching Statistics by Backward Search

In the following, let S^1 and S^2 be strings of length n_1 and n_2, respectively. We tacitly assume that S^1 has the sentinel character at the end (and nowhere else). As already mentioned, Chang and Lawler [7] introduced matching statistics in the context of approximate string matching. For each position p_2 in the string S^2, they searched for the longest match of $S^2[p_2..n_2]$ with a substring of S^1 by matching S^2 in *forward* direction against the *suffix tree* of S^1. This takes only linear time if suffix links are used as shortcuts in the traversal of the suffix tree; cf. [3, Section 7.8]. By contrast, we here match S^2 in *backward* direction against a compressed index of S^1. Our algorithm does not rely on suffix links but on the ability to determine parent intervals of lcp-intervals efficiently. Sadakane's [16] compressed suffix tree requires $4n+o(n)$ bits and allows one to determine a parent interval in constant time. Recently, we presented a balanced parentheses data structure that can be constructed in linear time from the lcp-array, uses only $2n+o(n)$ bits, and—in combination with the lcp-array—allows us to compute $\mathsf{NSV}[i]$ in constant time and $\mathsf{PSV}[i]$ in $\mathcal{O}(\log|\Sigma|)$ time [18]. In other words, a parent interval can be computed in constant time (for a constant alphabet).

Algorithm 2. Computing matching statistics by backward search

```
p2 ← n2
(q, [i..j]) ← (0, [1..n1])
while p2 ≥ 1 do
  [lb..rb] ← backwardSearch(S^2[p2], [i..j])
  if [lb..rb] ≠ ⊥ then
    q ← q + 1
    ms[p2] ← (q, [lb..rb])
    [i..j] ← [lb..rb]
    p2 ← p2 − 1
  else if [i..j] = [1..n1] then
    ms[p2] ← (0, [1..n1])
    p2 ← p2 − 1
  else
    q-[i..j] ← parent([i..j])
```

An even more space-efficient implementation was proposed by Fischer et al. [19]. Their method to identify parent intervals also uses PSV and NSV values and runs in sublogarithmic time (in n). Either of the three methods can be used in our algorithms.

Before explaining our new Algorithm 2, we define matching statistics slightly more general than Chang and Lawler did.

Definition 1. *A* matching statistics *of S^2 w.r.t. S^1 is an array ms such that for every entry $ms[p_2] = (q, [lb..rb])$, $1 \leq p_2 \leq n_2$, the following holds:*

1. *$S^2[p_2..p_2 + q - 1]$ is the longest prefix of $S^2[p_2..n_2]$ which is substring of S^1.*
2. *$[lb..rb]$ is the $S^2[p_2..p_2 + q - 1]$-interval in the suffix array of S^1.*

Algorithm 2 shows how matching statistics can be computed by backward search. To exemplify it, we match the string $S^2 = caaca$ backwards against the compressed full-text index of $S^1 = acaaacatat\$$; cf. Fig. 1. Starting with the last character of S^2 and the ε-interval $[1..n_1]$, backward search returns the a-interval $[2..7]$, and Algorithm 2 sets $ms[5] = (1, [2..7])$. Similary, it determines $ms[4] = (2, [8..9])$, $ms[3] = (3, [4..5])$, and $ms[2] = (4, [3..3])$. The procedure call $backwardSearch(c, [3..3])$ returns $\perp$, indicating that $S^2[1..5] = caaca$ is not a substring of S^1. In this case—if a mismatch occurs—the algorithm determines the parent interval of the current interval $[3..3]$, which in our example is the aa-interval 2-$[2..3]$. The next procedure call $backwardSearch(c, [2..3])$ returns the caa-interval $[8..8]$, and $ms[1]$ is set to $(3, [8..8])$.

We prove the correctness of Algorithm 2 by finite induction on the length $n_2 - p_2 + 1$ of the suffix $S^2[p_2..n_2]$ of S^2. If the length equals 1, i.e., $p_2 = n_2$ then there are two possibilities. The character $c = S^2[n_2]$ either (a) occurs in S^1 or (b) it does not. In case (a), Algorithm 2 sets $ms[n_2] = (1, [lb..rb])$, where $[lb..rb]$ is the c-interval. This is certainly correct. In case (b), Algorithm 2 sets $ms[n_2] = (0, [1..n_1])$, where

$[1..n_1]$ is the ε-interval. This is also correct. As induction hypothesis, we may assume that for some fixed position $p_2 + 1$ with $1 \leq p_2 < n_2$, Algorithm 2 correctly computed the matching statistic $ms[p_2 + 1] = (q, [i..j])$, i.e.,

1. $\omega = S^2[p_2 + 1..p_2 + q]$ is the longest prefix of $S^2[p_2 + 1..n_2]$ that occurs as a substring of S^1.
2. $[i..j]$ is the ω-interval in the suffix array of S^1.

In the inductive step, we must show that Algorithm 2 correctly computes $ms[p_2]$. Let $c = S^2[p_2]$. If $c\omega$ is a substring of S^1, then $backwardSearch(c, [i..j])$ yields the $c\omega$-interval $[lb..rb]$ in the suffix array of S^1. It is readily verified that $c\omega = S^2[p_2..p_2 + q]$ is the longest prefix of $S^2[p_2..n_2]$ that occurs as a substring of S^1. Consequently, $ms[p_2] = (q + 1, [lb..rb])$. Otherwise, if $c\omega$ is not a substring of S^1, then $backwardSearch(c, [i..j])$ returns $\bot$. We consider the two subcases (a) $[i..j] = [1..n_1]$ and (b) $[i..j] \neq [1..n_1]$.
(a) If $[i..j] = [1..n_1]$, i.e., $\omega = \varepsilon$, then the character c does not occur in S^1. This means that the longest prefix of $S^2[p_2..n_2]$ that occurs as a substring of S^1 is the empty string ε and $ms[p_2] = (0, [1..n_1])$.
(b) If $[i..j] \neq [1..n_1]$, then $\omega \neq \varepsilon$. Because $c\omega$ is not a substring of S^1, we must search for the longest prefix u' of ω such that cu' is a substring of S^1. Let $[i'..j']$ be the parent lcp-interval of $[i..j]$. The lcp-interval $[i'..j']$ is the u-interval of a proper prefix u of ω. Suppose that b is the character immediately following u in ω, i.e., $\omega = ubv$ for some string v. Because the u-interval $[i'..j']$ is the parent lcp-interval of the ω-interval $[i..j]$, every substring ω' of S^1 that has ub as a prefix must also have ω as a prefix. We claim that the string cub cannot occur in S^1. To prove the claim, suppose to the contrary that cub is a substring of S^1. Because every substring ω' of S^1 that has ub as a prefix must also have ω as a prefix, it follows that $c\omega$ must be a substring of S^1. This contradicts the fact that $c\omega$ is not a substring of S^1 and thus proves the claim that the string cub cannot occur in S^1. Consequently, u is the longest prefix of ω such that cu is a possible substring of S^1. Observe that the algorithm checks in the next iteration of the while-loop whether or not cu is indeed a substring of S^1. If so, then u is the longest prefix of ω such that cu is a substring of S^1. If not, the algorithm continues with the parent interval of the u-interval $[i'..j']$, and so on, until either backward search succeeds or the interval $[1..n_1]$ is found. In both cases $ms[p_2]$ is assigned correctly.

We use an amortized analysis to derive the worst-case time complexity of Algorithm 2. Each statement in the while-loop takes only constant time (assuming a constant alphabet). We claim that the number of iterations of the while-loop over the entire algorithm is bounded by $2n_2$. In each iteration of the while-loop, either the position p_2 in S^2 is decreased by one or the search interval $[i..j]$ is replaced with its parent interval. Clearly, p_2 is decreased n_2 times and we claim that at most n_2 many search intervals can be replaced with its parent interval. To see this, let the search interval $[i..j]$ be the ω-interval and let $[i'..j']$ denote its parent interval. The lcp-interval $[i'..j']$ is the u-interval of a proper prefix u of ω. Consequently, each time a search interval is replaced with its parent interval, the length of the search string ω is shortened by at least one. Since the overall

length increase of all search strings is bounded by n_2, the claim follows. Thus, in our implementation, Algorithm 2 has a worst-case time complexity of $O(n_2)$.

4 Computing Maximal Exact Matches by Backward Search

The starting point for any comparison of large genomes is the computation of exact matches between their DNA sequences S^1 and S^2. In our opinion, maximal exact matches—exact matches that cannot be extended in either direction towards the beginning or end of S^1 and S^2 without allowing for a mismatch—are most suitable for this task.

Definition 2. *An exact match between two strings S^1 (where S^1 ends with \$) and S^2 of lengths n_1 and n_2 is a triple (q, p_1, p_2) such that $S^1[p_1..p_1 + q - 1] = S^2[p_2..p_2 + q - 1]$. An exact match is called right maximal if $p_2 + q - 1 = n_2$ or $S^1[p_1+q] \neq S^2[p_2+q]$. It is called left maximal if $p_2 = 1$ or $\mathsf{BWT}[p_1] \neq S^2[p_2-1]$. A left and right maximal exact match is called maximal exact match (MEM).*

In genome comparisons, one is merely interested in MEMs (q, p_1, p_2) that exceed a user-defined length threshold ℓ, i.e., $q \geq \ell$. In the software-tool CoCoNUT [11], maximal exact matches between S^1 and S^2 are computed by matching S^2 in *forward* direction against an enhanced suffix array of S^1. The bottleneck in large-scale applications like genome comparisons is often the space requirement of the software-tool. If the index structure (e.g. an enhanced suffix array) does not fit into main memory, then it is worthwhile to use a compressed index structure instead. Our new Algorithm 3 computes maximal exact matches by matching S^2 in *backward* direction against a *compressed full-text index* of S^1.

Algorithm 3 proceeds as in the computation of the matching statistics by backward search, i.e., for each position p_2 in S^2, it computes the longest match of $S^2[p_2..n_2]$ with a substring of S^1 of length q, and the matching $S^2[p_2..q-1]$-interval $[lb..rb]$. This time, however, it keeps track of the longest matching path. To be precise, it stores the matching statistics $ms[p_2] = (q, [lb..rb])$ of each position p_2 satisfying $q \geq \ell$ as a triple $(q, [lb..rb], p_2)$ in a list called *path* until a mismatch occurs (i.e., until backward search returns $\perp$). Then, it computes MEMs from the triples in the list *path* (in its outer for-loop). If all elements of the list *path* have been processed, it computes the next longest matching path, and so on.

By construction (or more precisely, by the correctness of Algorithm 2), if the triple $(q, [lb..rb], p_2)$ occurs in some matching path, then $ms[p_2] = (q, [lb..rb])$ and $q \geq \ell$. (Note that for each position p_2 in S^2 at most one triple $(q, [lb..rb], p_2)$ appears in the matching paths.) Clearly, this implies that each $(q, \mathsf{SA}[k], p_2)$ is a longest right maximal exact match at position p_2 in S^2, where $lb \leq k \leq rb$. Now Algorithm 3 tests left maximality by $\mathsf{BWT}[k] \neq S^2[p_2 - 1]$. If $(q, \mathsf{SA}[k], p_2)$ is left maximal, then it is a maximal exact match between S^1 and S^2 with $q \geq \ell$, and the algorithm outputs it. After that, it considers the parent lcp-interval of $[lb..rb]$. Let us denote this parent interval by q'-$[lb'..rb']$. For each k with $lb' \leq k < lb$ or

Algorithm 3. Computing MEMs of length $\geq \ell$ by backward search

```
p_2 ← n_2
(q, [i..j]) ← (0, [1..n_1])
while p_2 ≥ 1 do
  path ← [ ]
  [lb..rb] ← backwardSearch(S^2[p_2], [i..j])
  while [lb..rb] ≠ ⊥ and p_2 ≥ 1 do
    q ← q + 1
    if q ≥ ℓ then
      add(path, (q, [lb..rb], p_2))
    [i..j] ← [lb..rb]
    p_2 ← p_2 − 1
    [lb..rb] ← backwardSearch(S^2[p_2], [i..j])
  for each (q', [lb'..rb'], p'_2) in path do
    [lb..rb] ← ⊥
    while q' ≥ ℓ do
      for each k ∈ [lb'..rb'] \ [lb..rb] do
        if p'_2 = 1 or BWT[k] ≠ S^2[p'_2 − 1] then
          output (q', SA[k], p'_2)
      [lb..rb] ← [lb'..rb']
      q'-[lb'..rb'] ← parent([lb'..rb'])
  if [i..j] = [1..n_1] then
    p_2 ← p_2 − 1
  else
    q-[i..j] ← parent([i..j])
```

$rb < k \leq rb'$, the triple $(q', \mathsf{SA}[k], p_2)$ is a right maximal exact match because $S^1[\mathsf{SA}[k]..\mathsf{SA}[k]+q'-1] = S^2[p_2..p_2+q'-1]$ and $S^1[\mathsf{SA}[k]+q'] \neq S^2[p_2+q']$. So if $q' \geq \ell$ and $\mathsf{BWT}[k] \neq S^2[p_2-1]$, then the algorithm outputs $(q', \mathsf{SA}[k], p_2)$. Then it considers the parent lcp-interval of $[lb'..rb']$ and so on. To sum up, Algorithm 3 checks every right maximal exact match exceeding the length threshold ℓ for left maximality. It follows as a consequence that it detects every maximal exact match of length $\geq \ell$.

We exemplify the algorithm by matching the string $S^2 = caaca$ backwards against the compressed full-text index of $S^1 = acaaacatat\$$. For the length threshold $\ell = 2$, the first matching path is $(2, [8..9], 4)$, $(3, [4..5], 3)$, $(4, [3..3], 2)$. The triple $(2, [8..9], 4)$ yields no output, but for the triple $(3, [4..5], 3)$, the algorithm outputs the MEM $(3, 1, 3)$. (Note that the parent intervals of $[8..9]$ and $[4..5]$ are not considered because their lcp-value is smaller than $\ell = 2$.) The triple $(4, [3..3], 2)$ yields the output $(4, 4, 2)$, and when its parent interval 2-$[2..3]$ is considered, the algorithm does not output the right maximal exact match $(2, 3, 2)$ because it is not left maximal. Now all triples in the matching path have been considered, and the algorithm computes the next longest matching path starting at position $p_2 = 1$ and the parent interval 2-$[2..3]$ of $[i..j] = [3..3]$. This new path consists of the triple $(3, [8..8], 1)$ resulting in the output $(3, 2, 1)$ and $(2, 6, 1)$.

Let us analyse the worst-case time complexity of Algorithm 3. If the outer for-loop was not there, it would run in $O(n_2)$ time; see the run-time analysis of Algorithm 2. In each execution of the while-loop within the outer for-loop, Algorithm 3 tests a right maximal exact match of length $\geq \ell$ for left maximality by $\mathsf{BWT}[k] \neq S^2[p'_2 - 1]$. As a matter of fact, it is not necessary to store the Burrows-Wheeler transformed string $\mathsf{BWT}[1..n_1]$ of S^1. This is because the wavelet tree allows to access the LF-mapping without it, and we have $\mathsf{BWT}[k] \neq c$ if and only if $LF(k) \notin [i..j]$, where $[i..j]$ is the c-interval (e.g., $backwardsearch(c, [1..n_1])$ returns $[i..j]$). In other words, the test $\mathsf{BWT}[k] \neq S^2[p'_2 - 1]$ in Algorithm 3 can be replaced with the test $LF(k) \notin [i..j]$, where $[i..j]$ is the $S^2[p'_2 - 1]$-interval, and this test takes only constant time. Therefore, the algorithm runs in $O(n_2 + z + occ \cdot t_{\mathsf{SA}})$ time, where occ (z) is the number of (right) maximal exact matches of length $\geq \ell$ between the strings S^1 and S^2, and t_{SA} is the access time to the compressed full-text index.

5 Experimental Results

Our implementations are available under the GNU General Public License at `http://www.uni-ulm.de/in/theo/research/seqana`. As already mentioned, the wavelet tree of a string S^1 of length n_1 needs only $n_1 \log |\Sigma| + o(n_1 \log |\Sigma|)$ bits (about $1.25 n_1 \log |\Sigma|$ bits in practice). The balanced parentheses data structure [18] to determine parent intervals requires $2n_1 + o(n_1)$ bits (about $3n_1$ bits in practice). The lcp-array is stored as suggested in [17], i.e., values < 255 are stored in one byte and larger values are stored in an index sorted array—so that larger values are retrieved in logarithmic time. Thus, the lcp-array uses n_1 to $4n_1$ bytes (n_1 to $2n_1$ bytes in practice). Alternatively, one could use Sadakane's [16] encoding of the lcp-array, which uses only $2.1n_1$ bits in practice. However, this slows down the computation considerably. So here is room for improvement.

We conducted experiments to compare our algorithms using backward search with "standard" algorithms using forward search. (Very recently, Russo et al. [20] presented algorithms based on forward search for parallel and distributed compressed indexes. To the best of our knowledge, implementations of their algorithms are not available.)

Test setup: All programs were compiled using `g++` version 4.1.2 with options `-O3 -DNDEBUG` on a 64 bit Linux (Kernel 2.6.16) system equipped with a Dual-Core AMD Opteron processor with 3 GHz and 3GB of RAM. For each test set we measure the real runtime (user time plus system time) of the programs in order to show the effects of swapping, which occurs when a program does not fit into main memory. All programs construct an index (suffix array, wavelet tree) in a first phase and then perform their task based on the index. Because (a) the index can be reused and (b) different programs are used to construct the index, we solely focus on the second phase.

Matching statistics: In machine learning, string kernels in combination with SVM provide string classification algorithms. After a preprocessing phase, an

Size S^1	backwardSK	sask-sl
40MB	144MB	840MB
80MB	277MB	1680MB
120MB	412MB	2520MB
160MB	539MB	3360MB

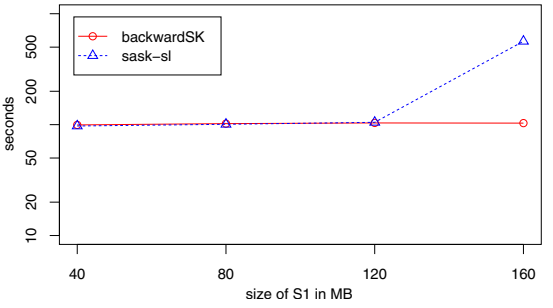


Fig. 2. Left: The amount of memory used by the programs (without the construction phase because the programs use different construction algorithms). Right: Runtime in seconds of the programs to calculate the matching statistics. Note that the y-axis is in log scale.

efficient computation of a string kernel boils down to the computation of matching statistics; see [8]. In this context, Teo and Vishwanathan [8] implemented the linear time algorithm of Abouelhoda et al. [17]. Their program sask-sl is available at `http://users.cecs.anu.edu.au/~chteo/SASK.html`. It uses $18n_1$ to $21n_1$ bytes (n_1 bytes for the string S^1, $4n_1$ bytes for the suffix array, n_1 to $4n_1$ bytes for the lcp-array, $4n_1$ bytes for the child table, and $8n_1$ bytes for the suffix links). In our experiments it used $21n_1$ bytes. We compared their implementation of matching statistics computation with our implementation *backwardSK*. As data we used a concatenation of books from project Gutenberg. Fig. 2 shows the memory usage and the real runtime of the programs to compute the matching statistics, given precalculated supporting data structures of S^1. We evaluated the effect of increasing the size of S^1 while keeping S^2 constant (S^2 has size 20 MB). For sizes up to 120 MB, the runtimes of the programs do not differ significantly. When the size of S^1 reaches 160 MB, however, sask-sl slows down drastically because it does not fit into main memory anymore (so parts of it are swapped out of main memory).

Maximal exact matches: Besides the data structures mentioned above, we used a compressed suffix array based on a wavelet tree that occupies $(n_1 \log n_1)/k$ bits [14]. Its size and the access time t_{SA} to it depend on the parameter $k \geq 1$. For $k = 1$, it is the uncompressed suffix array and the access time is constant. For $k > 1$, only every kth entry of the suffix array is stored and the remaining entries are reconstructed in $k/2$ steps (on average) with the LF-mapping (which can be computed with the wavelet tree). In order to compare our implementation *backwardMEM* with other software-tools computing MEMs, we chose the one developed by Khan et al. [21], called *sparseMEM* (`http://compbio.cs.princeton.edu/mems`). Their method computes MEMs between S^1 and S^2 by matching S^2 in *forward* direction against a *sparse* suffix array of S^1, which stores every Kth suffix of S^1, where K is a user-defined parameter. (The reader should be aware of the difference between a *sparse* and

a *compressed* suffix array: A sparse suffix array stores each Kth suffix of S^1, while a compressed suffix array stores each kth entry of the suffix array of S^1.) Our choice is justified by the fact that the sequential version of *sparseMEM* beats the open-source software-tool *MUMmer* [10] and is competitive with the closed-source software-tool *vmatch* (`http://www.vmatch.de/`).

For a fair comparison, we used the sequential version of *sparseMEM* and the same input and output routines as *sparseMEM*. In the experiments, we used the program parameters `-maxmatch -n -l` ℓ, which set the length threshold on the MEMs to ℓ. We ran both programs on DNA-sequences of different species. In the uncompressed case ($k = K = 1$), the memory consumption of *backwardMEM* is smaller than that of *sparseMEM*, but *sparseMEM* is faster. In the compressed cases, *backwardMEM* performs quite impressively, in most cases much better than *sparseMEM*. For example, *backwardMEM* takes only 57s (using 235 MB for $k = 16$) to compare the human chromosome 21 with the mouse chromosome 16, whereas *sparseMEM* takes 10m34s (using 255 MB for $K = 8$); see Table 1.

The space consumption of *sparseMEM* decreases faster with K as that of *backwardMEM* with k, but its running time also increases faster. While the experiments show a clear space-time tradeoff for *sparseMEM*, this is fortunately not the case for *backwardMEM*. Sometimes its running time increases with increasing compression ratio, and sometimes it does not. This is because the algorithm is output-sensitive. More precisely, before a MEM $(q, \mathsf{SA}[i], p_2)$ can be output, the algorithm first has to determine the value $\mathsf{SA}[i]$. While this takes only constant time in an uncompressed suffix array, it takes t_{SA} time in a compressed suffix array, and the value of t_{SA} crucially depends on the compression ratio.

Table 1. For each pair of DNA-sequences (cf. http://compbio.cs.princeton.edu/mems), the time (in minutes and seconds) and space consumption (in MByte) of the programs are shown (without the construction phase). We tested different values of K and k to demonstrate the time-space tradeoff of the algorithms. The value ℓ is the length threshold on the MEMs.

S^1	$\lvert S^1\rvert$	S^2	$\lvert S^2\rvert$	ℓ						
sparseMEM	Mbp		Mbp		$K=1$		$K=4$		$K=8$	
A.fumigatus	29.8	*A.nidulans*	30.1	20	23s	307	3m59s	108	6m13s	74
M.musculus16	35.9	*H.sapiens21*	96.6	50	1m15s	430	10m52s	169	19m56s	163
H.sapiens21	96.6	*M.musculus16*	35.9	50	32s	957	5m08s	362	10m34s	255
D.simulans	139.7	*D.sechellia*	168.9	50	2m17s	1489	21m09s	490	49m34s	326
D.melanogaster	170.8	*D.sechellia*	168.9	50	2m37s	1861	28m49s	588	55m43s	386
D.melanogaster	170.8	*D.yakuba*	167.8	50	2m49s	1860	32m57s	587	61m39s	384
backwardMEM	Mbp		Mbp		$k=1$		$k=8$		$k=16$	
A.fumigatus	29.8	*A.nidulans*	30.1	20	43s	187	49s	89	50s	82
M.musculus16	35.9	*H.sapiens21*	96.6	50	2m09s	261	2m09s	142	2m16s	134
H.sapiens21	96.6	*M.musculus16*	35.9	50	51s	576	59s	258	57s	235
D.simulans	139.7	*D.sechellia*	168.9	50	5m42s	859	17m35s	399	32m39s	366
D.melanogaster	170.8	*D.sechellia*	168.9	50	4m33s	1074	11m19s	504	20m38s	464
D.melanogaster	170.8	*D.yakuba*	167.8	50	3m50s	1068	5m18s	502	7m35s	463

References

1. Weiner, P.: Linear pattern matching algorithms. Proc. 14th IEEE Annual Symposium on Switching and Automata Theory. 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. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
6. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center (1994)
7. Chang, W., Lawler, E.: Sublinear approximate string matching and biological applications. Algorithmica 12(4/5), 327–344 (1994)
8. Teo, C., Vishwanathan, S.: Fast and space efficient string kernels using suffix arrays. In: Proc. 23rd Conference on Machine Learning, pp. 929–936. ACM Press, New York (2003)
9. Rahmann, S.: Fast and sensitive probe selection for DNA chips using jumps in matching statistics. In: Proc. 2nd IEEE Computer Society Bioinformatics Conference, pp. 57–64 (2003)
10. Kurtz, S., Phillippy, A., Delcher, A., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.: Versatile and open software for comparing large genomes. Genome Biology 5, R12 (2004)
11. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: CoCoNUT: An efficient system for the comparison and analysis of genomes. BMC Bioinformatics 9, 476 (2008)
12. Puglisi, S., Smyth, W., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2), 1–31 (2007)
13. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article 2 (2007)
15. 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)
16. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41, 589–607 (2007)
17. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
18. Ohlebusch, E., Gog, S.: A compressed enhanced suffix array supporting fast string matching. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 51–62. Springer, Heidelberg (2009)
19. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoretical Computer Science 410(51), 5354–5364 (2009)
20. Russo, L., Navarro, G., Oliveira, A.: Parallel and distributed compressed indexes. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 348–360. Springer, Heidelberg (2010)
21. Khan, Z., Bloom, J., Kruglyak, L., Singh, M.: A practical algorithm for finding maximal exact matches in large sequence data sets using sparse suffix arrays. Bioinformatics 25, 1609–1616 (2009)

The Gapped Suffix Array: A New Index Structure for Fast Approximate Matching

Maxime Crochemore[1,2] and German Tischler[1,3]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, UK
{maxime.crochemore,german.tischler}@kcl.ac.uk
[2] Université Paris-Est, France
[3] Newton Fellow

Abstract. Approximate searching using an index is an important application in many fields. In this paper we introduce a new data structure called the gapped suffix array for approximate searching in the Hamming distance model. Building on the well known filtration approach for approximate searching, the use of the gapped suffix array can improve search speed by avoiding the merging of position lists.

1 Introduction

Pattern matching in textual data is a much studied question in Computer Science and large parts of books on algorithms on strings and sequences are devoted to the question (see e.g. [2,6]). Several types of applications require approximate matching rather than exact matching of the patterns. This is typically the situation for motif search and inference in biological molecular sequences because they allow some diversity without altering the basic information they carry. But this is not by far the only domain demanding approximate matching solutions. A main technique to deal with the question is the notion of alignment, which admits a considerable number of variants and is parameterised by the costs of allowed elementary operations (see e.g. [1]).

However there are actually two sub-problems depending on which are known first, patterns to be searched for or data to be searched. They admit totally different types of solutions. The paradigm solution for searching for approximate occurrences of a fixed pattern under the notion of Levenshtein operations is due to Landau and Vishkin [8], and the same authors designed a simpler version when only mismatches are considered [7]. The second type of solution appears when the data is to be searched for multiple patterns. It is then appropriate to index the data for accelerating their future inspection and analysis.

Indexing for approximate searches is the problem we address in the article, but with one important restriction: patterns are of fixed size. Moreover the proposed solution accommodates only few mismatches to be feasible with reasonable resources.

In this paper we introduce what we call the gapped Suffix Array. It is a data structure enhancing the standard suffix array and tailored to accept searches for patterns up to some mismatches.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 359–364, 2010.

2 Definitions

Let throughout this paper Σ be a finite ordered alphabet and let Σ^* denote the set of all finite strings over Σ. We denote the empty string by ϵ and the length of a string u by $|u|$. Let $y = y[0] \dots y[n-1]$ denote a string of finite length n over Σ which we call the text. We denote the factor starting at position i and ending at position j of some string x by $x[i\,..\,j]$. It is defined by $x[i\,..\,j] = x[\max(0,i)]x[\max(0,i)+1] \dots x[\min(j,|x|-1)]$ for $\max(0,i) \le \min(j,|x|-1)$ and $x[i\,..\,j] = \epsilon$ otherwise. A prefix of a string x is $x[0\,..\,i]$ for any position i and a suffix of x is $x[i\,..\,|x|-1]$ for any position i. A string $u \in \Sigma^*$ is lexicographically smaller than a string $v \in \Sigma^*$ (which we denote by $u < v$), if $u \neq v$ and either $u = \epsilon$ or $u \neq \epsilon \neq v$ and $u[0] < v[0]$ or $u \neq \epsilon \neq v$ and $u[0] = u[0]$ and $u[1\,..\,|u|-1] < v[1\,..\,|v|-1]$. The array SA of length n is defined by $\mathsf{SA}[r]$ being the start position of the r'th lexicographically smallest non empty suffix of y, i.e. we obtain the relation

$$y[\mathsf{SA}[0]\,..\,n-1] < y[\mathsf{SA}[1]\,..\,n-1] < \ldots < y[\mathsf{SA}[n-1]\,..\,n-1] \ .$$

The array SA can be computed from the string y in linear time if $|\Sigma| \in O(n^c)$ for some constant c (in particular for $c = 0$, i.e. alphabets of constant size, cf. [2]). Let the array ISA be defined by $\mathsf{ISA}[\mathsf{SA}[r]] = r$ for $0 \le r < n$. The length of the longest common prefix of $u, v \in \Sigma^*$, which we denote by is $\mathsf{lcp}(u,v)$, is the largest $l \le \max(|u|,|v|)$ such that $u[0\,..\,l-1] = v[0\,..\,l-1]$. We denote the length of the longest common prefix of the two suffixes starting at position i and j of y by $\mathsf{lcp}_{\mathrm{y}}(i,j)$ and we define $\mathsf{lcp}_{\mathrm{r}}(r,q) = \mathsf{lcp}_{\mathrm{y}}(\mathsf{SA}[r],\mathsf{SA}[q])$ for $0 \le r,q < n$. We define the LCP array by $\mathsf{LCP}[r] = \mathsf{lcp}_{\mathrm{r}}(r-1,r)$ for $1 \le r < n$ and by $\mathsf{LCP}[0] = 0$. It is well known that the identity

$$\mathsf{lcp}_{\mathrm{r}}(r,q) = \min\{\mathsf{LCP}[r+1], \mathsf{LCP}[r+2], \ldots, \mathsf{LCP}[q]\}$$

holds for $0 \le r < q < n$. The LCP array can be computed from the string y and the array SA in linear time (cf. [2]). The pair of arrays $(\mathsf{SA}, \mathsf{LCP})$ is commonly known as the *suffix array* of the string y.

For two strings u, v such that $|u| = |v| = m$ the Hamming distance $d(u,v)$ of u and v is defined as the number of differences between u and v. For sake of completeness we define $d(u,v) = \infty$ for strings u, v such that $|u| \neq |v|$.

3 Approximate String Matching

We consider approximate string matching by the well known procedure called *filtration* or *partitioning into exact matches* (cf. [10,9]). Let $x \in \Sigma^*$ be a pattern of length m. We want to find occurrences of x in the text y with up to k mismatches under the Hamming distance. Partitioning into exact matches works as follows. We partition x into $q > k$ fragments $x_0, \dots x_{q-1} \in \Sigma^+$. We search the lists occurrences X_i of x_i. For each of the possibilities of choosing $q-k$ of the q fragments, we merge the respective lists of positions using the respective position

offsets. This provides us with $\binom{q}{q-k}$ candidate position lists. The union X of these merged lists is a superset of the positions of occurrences of x in y with up to k mismatches. We obtain the list of occurrences of x in y by filtering X using an online algorithm for testing if the candidate positions designate occurrences with at most k mismatches.

As an example consider a pattern x we partition into three fragments x_0, x_1 and x_2 for searching its occurrences with 1 mismatch. We have to consider three pairs of fragments: (x_0, x_1), (x_1, x_2) and (x_0, x_2). The first two combinations are easily found using an index for y. We need only search for the patterns x_0x_1 and x_1x_2. The third requires merging of lists in the conventional scheme. If we have an index supporting searching patterns with gaps however, merging is no longer necessary. Supporting the search of patterns with gaps is the purpose of the gapped suffix array.

4 The Gapped Suffix Array

4.1 Definitions

We define a generalisation of the notion of *lexicographical order* which we call (g_0, g_1)-*lexicographical order*, where $g_0, g_1 \in \mathbb{N}$. A string $u \in \Sigma^*$ is (g_0, g_1)-lexicographically smaller than a string $v \in \Sigma^*$

- if $u[0\,..\,g_0 - 1] \neq v[0\,..\,g_0 - 1]$ then iff $u < v$
- otherwise $(u[0\,..\,g_0 - 1] = v[0\,..\,g_0 - 1])$, if $\min(|u|, |v|) > g_0 + g_1$ then iff $u[0\,..\,g_0 - 1]u[g_0 + g_1\,..\,|u| - 1] < v[0\,..\,g_0 - 1]v[g_0 + g_1\,..\,|v| - 1]$
- otherwise $(u[0\,..\,g_0 - 1] = v[0\,..\,g_0 - 1], \min(|u|, |v|) \leq g_0 + g_1)$, iff $|u| < |v|$

Informally the definition means that we compare u and v ignoring the presence of the letters in the position interval $[g_0, g_0 + g1)$, where we have to take some care about those strings which end inside the gap area. The (g_0, g_1)-lexicographical order is a total order on a set of strings such that each string has a different length. We define the (g_0, g_1)-*gapped suffix array* of y, which we denote by $(g_0, g_1) - \mathsf{gSA}$ (or shorter gSA, if the parameters g_0 and g_1 are clear from the context), as the array containing the starting positions of the non-empty suffixes of y in (g_0, g_1)-lexicographically ascending order. The (g_0, g_1)-*prefix* of the string $u \in \Sigma^*$, which we denote by $D(g_0, g_1, u)$, is defined as $u[0\,..\,g_0 - 1]$ if $|u| \leq g_0 + g_1$ and as $u[0\,..\,g_0 - 1]u[g_0 + g_1\,..\,|u| - 1]$ if $|u| > g_0 + g_1$. We define the (g_0, g_1)-gLCP array (we use the shorter notation gLCP, if the parameters g_0 and g_1 are clear from the context) for the string y based on its (g_0, g_1)-gSA array by

$$\mathsf{gLCP}[r] = \mathsf{lcp}(D(g_0, g_1, y[\mathsf{gSA}[r-1]\,..\,n-1]), D(g_0, g_1, y[\mathsf{gSA}[r]\,..\,n-1]))$$

for $r > 0$ and $\mathsf{gLCP}[r] = 0$ for $r = 0$.

4.2 Searching Using the Gapped Suffix Array

Assume we are given a query x of length $m > g_0 + g_1$ and we want to find all occurrences of patterns in $x[0\,..\,g_0 - 1]\Sigma^{g_1}x[g_0 + g_1\,..\,m - 1]$ in a text y using

the array (g_0, g_1)-gSA for y. The search method we use is analogous to the one we would use for searching an ungapped pattern using the array SA. The only major difference is that we suitably substitute the lexicographic order by the (g_0, g_1)-lexicographic order in the binary search for the interval of gapped suffix matching x. Thus the time required to report the *occ* gapped occurrences of x in y is $O((m - g_1) \log n + occ)$ if we do not use an adjoint (g_0, g_1)-gLCP array and $O((m - g_1) + \log n)$ if we do.

4.3 Computing the Gapped Suffix Array

For the rest of the section assume we have fixed two natural numbers g_0 and g_1 and want to compute the arrays (g_0, g_1)-gSA (short gSA) and (g_0, g_1)-gLCP (short gLCP). We now show how to deduce the sorting in gSA in linear time $O(n)$ from the suffix array of y.

Let $\mathsf{GRANK}[r]$ be defined as the number of ranks $r' < r$ such that $\mathsf{LCP}[r] < g_0$. GRANK contains the ranks of factors of y with length up to g_0. The largest number we can find in GRANK is n. If $\mathsf{GRANK}[\mathsf{ISA}[i]] < \mathsf{GRANK}[\mathsf{ISA}[j]]$ for two positions i, j, then the suffix at position i is lexicographically and (g_0, g_1)-lexicographically smaller than the one at position j. Thus the order of the suffixes between SA and gSA can only differ if $\mathsf{GRANK}[r] = \mathsf{GRANK}[q]$ for two ranks r and q. If $\mathsf{GRANK}[r] = \mathsf{GRANK}[q]$, then we can determine the order of the respective gapped suffixes in gSA by checking $\mathsf{ISA}[\mathsf{SA}[r] + g_0 + g_1]$ and $\mathsf{ISA}[\mathsf{SA}[q] + g_0 + g_1]$, given that these two are defined. A problem occurs for such ranks r where $\mathsf{SA}[r]+g_0+g_1 \geq n$ because the obtained value is not a valid position on y and thus ISA is not defined for it. According to the definition of the (g_0, g_1)-lexicographic order, this problem can be solved by sorting along the array HRANK given by

$$\mathsf{HRANK}[r] = \begin{cases} \mathsf{ISA}[\mathsf{SA}[r] + g_0 + g_1] + g_0 + g_1 & \text{if } SA[r] + g0 + g1 < n \\ n - 1 - \mathsf{SA}[r] & \text{otherwise} \end{cases}$$

The range of numbers found in the array HRANK is $[0, n+g_0+g_1-1]$, in particular the upper bound is $O(n)$. We can compute a representation of the array gSA by sorting the sequence of ranks $0, \ldots, n-1$ by the pair $(\mathsf{GRANK}[r], \mathsf{HRANK}[r])$. This can be performed efficiently in linear time using a two stage radix sort, where we first sort by HRANK and then by GRANK. The concrete array gSA can then be obtained from this intermediate representation by mapping each rank on the suffix array to the respective position.

Theorem 1. *Given a string y of length n and its suffix sorting SA, the gapped suffix array (g_0, g_1)-gSA of y can be computed in linear time $O(n)$.*

4.4 Computing the Gapped LCP Array gLCP

We show how to compute the gapped LCP array gLCP from the suffix array and the array $(g_0, g_1) - \mathsf{gSA}$ (short gSA) in linear time. We require a constant time solution of the range minimum query (RMQ) problem after linear time

preprocessing (see e.g. [3]). We can obtain the array $(g_0, g_1)-\mathsf{gLCP}$ (short gLCP) by setting

$$\mathsf{gLCP}[r] = \begin{cases} \mathsf{LCP}[r] \text{ if } \mathsf{LCP}[r] < g_0 \\ g_0 \quad \text{if } \max(\mathsf{gSA}[r] + g_0 + g_1, \mathsf{gSA}[r-1] + g_0 + g_1) \geq n \\ g_0 + l \quad \text{otherwise, where } l = \min(\mathsf{LCP}[p] + 1, \ldots, \mathsf{LCP}[q]) \text{ for} \\ \qquad p' = \mathsf{ISA}[\mathsf{gSA}[r-1] + g_0 + g_1] \\ \qquad q' = \mathsf{ISA}[\mathsf{gSA}[r] + g_0 + g_1] \\ \qquad p = \min(p', q') \text{ and } q = \max(p', q') \end{cases}$$

As every single step in the computation takes constant time and we have n steps, the runtime for computing gLCP is $O(n)$.

Theorem 2. *Given a string y of length n and its suffix sorting SA, the gapped LCP array (g_0, g_1)-gLCP of y can be computed in linear time $O(n)$.*

5 Representing the Array gSA in Reduced Space

The uncompressed version of the gSA array requires $n\lceil \log n \rceil$ bits. The text however can be stored in $n\lceil \log |\Sigma| \rceil$ bits. In applications the size of the alphabet is fixed an small. Thus the space taken by the gSA will often be much larger than the space required for the text. The array SA is compressible (cf. [5]). Unfortunately, methods for compressing SA cannot be applied for compressing the array gSA, as the compression of SA requires the sorting of the suffixes according to the lexicographical order, which in general is not the same as the (g_0, g_1)-lexicographical order.

We provide a simple method for storing the array gSA using less than $n \log n$ bits space on average. Decoding the compressed representation of gSA will require the array SA. We limit our description to the aspects necessary for searching using the array gSA, i.e. our description allows accessing values in gSA corresponding to a provided query string. The more general case of accessing gSA for a given rank r without knowing a corresponding string can be facilitated using some additional succinct data structures. We omit the description for lack of space. We assume that the query string has a length of at least g_0. For shorter strings searching on the suffix array is sufficient. This may enumerate occurrences in a different order. However, this is not critical in most applications.

Let $R[r] = \{r' | \mathsf{HRANK}[r'] = r\}$. Each $R[r]$ is given as an interval of ranks. Observe that the sequences found in SA and gSA in each such interval are permutations of each other. On average we can expect each interval to have a size of $\frac{n}{|\Sigma|^{g_0}}$. Thus the permutation transforming gSA into SA can be stored using $\lceil \log |R[r]| \rceil$ bits per number for each interval r. Each interval $R[r]$ is assigned to a unique prefix $u(r) \in \Sigma^*$ of length at most g_0. The left bound $\mathsf{low}(r)$ and right bound $\mathsf{up}(r)$ of the interval $R[r]$ can be obtained by searching $u(r)$ on the suffix array. Knowing $\mathsf{low}(r)$ and $\mathsf{up}(r)$ we can compute the number of bits $b(r) = \lceil \log \mathsf{up}(r) - \mathsf{low}(r) + 1 \rceil$ used to store the numbers in the interval. Let L be defined by $L(r) = b(R^{-1}[r])$. L can be stored and indexed for rank queries via

a wavelet tree (cf. [4]) using $n\lceil\log\lceil\log n\rceil\rceil + o(n \log\log n)$ bits. Let $C[r]$ denote the permutation mapping the portion $R[r]$ of gSA to SA and let C_i denote the concatenation of all $C[r]$ such that $b(r) = i$. We can obtain $\mathsf{gSA}[r]$ for the query v as $C_{b(u^{-1}(v[0..g_0-1]))}[\mathsf{rank}_{b(u^{-1}(v[0..g_0-1]))}(r)]$ in time $O(\log\log n)$. The size of the data structure on average is $n(\log n - g_0 \log|\Sigma|) + n\log\log n + o(n\log\log n)$ bits. Using a space efficient wavelet tree data structure, the size is dominated by the first two terms in practice.

6 Conclusion

In this paper we have presented the gapped suffix array as a new efficient data structure for approximate matching under the Hamming distance. We obtained the same query time as for the conventional suffix array. The gapped suffix array can be derived in linear time from a text and its suffix sorting. Open problems include an improved query time independent of the text size, a succinct representation in $n\log\Sigma + o(n\log\Sigma)$ space and whether the gLCP array can be computed in linear time without using RMQ queries.

References

1. Böckenhauer, H.-J., Bongartz, D.: Algorithmic Aspects of Bioinformatics. Springer, Heidelberg (2007)
2. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings, 392 p. Cambridge University Press, Cambridge (2007)
3. 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)
4. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA 2003: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 841–850. Society for Industrial and Applied Mathematics, Philadelphia (2003)
5. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM J. Comput. 35(2), 378–407 (2005)
6. Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, Cambridge (1997)
7. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theor. Comput. Sci. 43, 239–249 (1986)
8. Landau, G.M., Vishkin, U.: Fast string matching with k differences. J. Comput. Syst. Sci. 37(1), 63–78 (1988)
9. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)
10. Navarro, G., Baeza-Yates, R.A., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Eng. Bull. 24(4), 19–27 (2001)

Parameterized Searching with Mismatches for Run-Length Encoded Strings
(Extended Abstract)

Alberto Apostolico[1,⋆], Péter L. Erdős[2,⋆⋆], and Alpár Jüttner[3,⋆⋆⋆]

[1] College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30318, USA and Dipartimento di Ingegneria dell' Informazione, Università di Padova Padova, Via Gradenigo 6/A, 35131 Padova, Italy
axa@cc.gatech.edu

[2] A. Rényi Institute of Mathematics, Hungarian Academy of Sciences, Budapest, P.O. Box 127, H-1364 Hungary
elp@renyi.hu

[3] Department of Operations Research, Eötvös University of Sciences, Pázmány Péter sétány 1/C, Budapest, H-1117 Hungary
alpar@cs.elte.hu

1 Introduction

Two strings $\mathbf{y}$ and $\mathbf{y}'$ of equal length over respective alphabets Σ_y and $\Sigma_{y'}$ are said to *parameterized match* if there exists a bijection $\pi : \Sigma_y \to \Sigma_{y'}$ such that $\pi(\mathbf{y}) = \mathbf{y}'$, i.e., renaming each character of $\mathbf{y}$ according to its corresponding element under π yields $\mathbf{y}'$. (Here we assume that all symbols of both alphabets are used somewhere.) Two natural problems are then *parameterized matching*, which consists of finding all positions of some text $\mathbf{x}$ where a pattern $\mathbf{y}$ parameterized matches a substring of $\mathbf{x}$, and *approximate parameterized matching*, which seeks, at each location of $\mathbf{x}$, a bijection π maximizing the number of parameterized matches at that location.

The first variant was introduced and studied by B. Baker [2,3] and others, motivated by issues of program compaction in software engineering. In [2,3], optimal, linear time algorithms were given under the assumption of constant size alphabets. A tight bound for the case of an alphabet of unbounded sizes was later presented in [1].

We study approximate variants of the problem where a (possibly controlled) number of mismatches is allowed. Hence, we are concerned with the second

⋆ Work carried out in part while visiting P.L. Erdős at the Rényi Institute, with support from the Hungarian Bioinformatics MTKD-CT-2006-042794, Marie Curie Host Fellowships for Transfer of Knowledge. Additional partial support was provided by the United States - Israel Binational Science Foundation (BSF) Grant No. 2008217, and by the Research Program of Georgia Tech.

⋆⋆ This work was supported in part by the Hungarian NSF, under contract NK 78439 and K 68262.

⋆⋆⋆ This work was supported by OTKA grant CK80124.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 365–371, 2010.

variant. Formally, we seek to find, for given text $\mathbf{x} = x_1 x_2 \dots x_n$ and pattern $\mathbf{y} = y_1 y_2 \dots y_m$ over respective alphabets Σ_t and Σ_p, the injection π_i from Σ_p to Σ_t maximizing the number of matches between $\pi_i(\mathbf{y})$ and $x_i x_{i+1} \dots x_{i+m-1}$ $(i = 1, 2, \dots n - m + 1)$. The general version of the problem can be solved in time $O\left(nm(\sqrt{m} + \log n)\right)$ by reduction to bipartite graph matching (refer to, e.g., [4]): each mutual alignment defines one graph in which edges are weighed according to the number of effacing characters and the problem is to choose the set of edges of maximum weight. An $O\left(nk\sqrt{k} + mk \log m)\right)$ time algorithm for parameterized matching with at most k mismatches was given in [5].

In this paper, we are interested in particular in a more general version where both strings are run-length encoded. This case was previously examined in [4], further restricted to the case where one of the alphabets is binary. For this special case, the authors gave a construction working in time $O\left(n + (r_p \times r_t)\alpha(r_t) \log r_t\right)$, where r_p and r_t denote the number of runs in the corresponding encodings for p and t, respectively and α is the inverse of Ackerman's function. This complexity actually reduces to $O(n + (r_p \times r_t))$ when both alphabets are binary.

Here we turn our interest to a more general case: we still assume run-length encoded text and pattern, however we relax the constraints on the the size of both alphabets. We give two algorithms, both having a time complexity of the form $O\Big((r_t \times r_p) \times F_1 \times F_2\Big)$, where F_1 and F_2 are polynomials of substantial degree in the alphabet size. The first one will compute the parameterized matching with mismatches between two run-length encoded strings giving values throughout the positions of the text; the second will report the positions where such a match is achieved within a preassigned bound k.

2 Problem Description

We assume that $\mathbf{x}$ and $\mathbf{y}$ are presented in their run-length encodings, denoted $X = X_1 X_2 \dots X_{r_t}$ and $Y = Y_1 Y_2 \dots Y_{r_p}$, respectively. The generic run, say X_k corresponds to a maximal substring $x_i x_{i+1} \dots x_{i+\ell-1}$ of consecutive occurrences of the same symbol, and is encoded by the pair $[\sigma, L_k]$ where $\sigma = x_i$, we set x_{n+1} to the empty word, and L_k is the k-th element of the *left-end list*, $L_1 = 1, L_2, \dots, L_{r_t+1} = n + 1$ of $\mathbf{x}$. This notation is extended to Y in analogy.

Consider the left-end list $L_1, \dots, L_{r_t+1}$ of the text and assume that we want to compute the approximate parameterized matching of the pattern beginning at location i of $\mathbf{x}$. It is convenient to view this alignment as a shift of the text $i-1$ positions to the left. The *i-shift list* is the list $L_1 - (i-1), L_2 - (i-1), \dots, L_{r_t+1} - (i-1)$. At position i, we are interested only in the portion of the text facing the pattern, that is, in the portion of the i-shift list containing the first $|\mathbf{y}| = m$ positive elements.

Definition 1. *The* i-fusion *(or* fusion *when this causes no ambiguity) is the sequence of intervals defined by the left-end list resulting from the merge of the left-end list of* $\mathbf{y}$ *with the i-shift list of* $\mathbf{x}$.

Depending on its origin, an element L_k of a fusion is said to be either a *pattern element* or a *text element*. Two elements of the same value coalesce in a single item and are said to form a *bump*.

As mentioned, the problem of finding an optimal injection from Σ_p to Σ_t at position i can be re-formulated in terms of the following standard graph theoretic problem.

We are given a weighted bipartite graph G_i with classes Σ_t and Σ_p, which draws its edge-weights from all possible bijections π_i, as follows: for each edge u, v ($u \in \Sigma_p$ and $v \in \Sigma_t$) the weight $w_{u,v}$ is the number of matches induced by accepting $\pi_i(u) = v$.

Under this formulation, an optimal approximate parametrized matching at position i corresponds to a *maximum weighted matching* (*MWM* for short) in a bipartite graph G.. There are several standard methods to determine the best weighted matching in a bipartite graph. However, the complexity of these algorithms is $O\left(V^2 \log V + VE\right)$ (see [8]), which would make the iterated application to our case prohibitive. In what follows, we show an approach that resorts to MWM more sparsely.

We begin by examining the effect of shifting the text by one position to the left. Clearly, this might change the weight $w_{u,v}$ for every pair. Let $\delta_{u,v}$ be the value of this change, which could be either negative or positive. The new weights after the shift will be in the form $w_{u,v} + \delta_{u,v.}$ Observe that as long as no bump occurs each consecutive shift will cause the same changes in the weights. Within such a regimen, we could calculate the new weights in our graph following every individual shift, each time at a cost of $O\left(|\Sigma_t||\Sigma_p|\right)$ time. But we could as well just use the linear functions $w_{u,v} + \alpha\delta_{u,v}$ to determine the weights of the maximum weighted matching achievable throughout, without computing every intermediate solution.

Whenever a bump occurs, we have to recalculate the δ functions. Each recalculation should take care of all characters that are actually affected by the bump. However, the number of function recalculations cannot exceed $r_t \times r_p$, the maximum number of of bumps.

In conclusion, our task can be subdivided into two interrelated, but computationally distinct, steps:

1. At every bump we have to (re)calculate the function Δ in order to quickly update the weights on the bipartite graph.
2. Within bumps, we have to update the weight function following each unit shift and determine whether or not a change in the matching function is necessary.

3 Parameterized String Matching via Parametric Graph Matching

For our intended treatment, we need to neglect for a moment the fact that the "weight" and "difference" functions (w and Δ, respectively) take integer values

and even that the relative shifts between pattern and text take place in a stepwise discrete fashion.

Definition 2. *Let $G = (A, B, E)$ be a bipartite graph with node sets A and B and edge set E. Assume that $|A| \leq |B|$. A set of independent edges is called* (graph) matching*, and a matching is* full *if it covers each vertex in A.*

Let $\mathcal{M}$ denote the set of all full matchings. Let $w : E \longrightarrow \mathbb{R}$ and $\Delta : E \longrightarrow \mathbb{R}$ be two given functions on the edges. For some $\lambda \in \mathbb{R}_+$ and for an arbitrary function $z : E \longrightarrow \mathbb{R}$ let $z_\lambda := z + \lambda\Delta$. Furthermore, let $L(z) := \max\{z(M) : M \in \mathcal{M}\}$ and $\mathcal{M}_z := \{M \in \mathcal{M} : z(M) = L(z)\}$. For the sake of simplicity we use the notations $L(\lambda) := L(w_\lambda)$ and $\mathcal{M}_\lambda := \mathcal{M}_{w_\lambda}$. A fundamental (but simple) property of the function L is the following

Claim 1. *$L(\lambda)$ is a convex piecewise linear function.* □

A function $\pi : A \cup B \longrightarrow \mathbb{R}$ is called a *potential* if $\pi(b) \geq 0$ for all $b \in B$. Let $z : E \longrightarrow \mathbb{R}$ be again an arbitrary weight function on the edges. Then a potential is called *z-feasible* or shortly *feasible* if $z(uv) \leq \pi(u) + \pi(v)$ holds for all $uv \in E$. Finally, let Π_z denote the set of z-feasible potentials. Then, Π_z is a closed convex polyhedron in $\mathbb{R}^{A \cup B}$.

The following duality theorem is well known (see e.g. [7]):

Theorem 1

$$L(z) = \min\left\{ \sum_{v \in A \cup B} \pi(v) : \pi \in \Pi_z \right\}.$$

If $\pi^* \in \Pi_z$ is an arbitrary minimizing feasible potential, then a full matching M is z-minimal if and only if $z(uv) = \pi^*(u) + \pi^*(v)$ holds for all $uv \in M$.

From the linearity of the objective function we get the following

Claim 2. *Let $[\alpha, \beta]$ be a linear segment of $L(\lambda)$. Then $\mathcal{M}_{\lambda_1} = \mathcal{M}_{\lambda_2}$ for all $\lambda_1, \lambda_2 \in (\alpha, \beta)$.* □

Definition 3. *Let $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a convex function. A vector $s \in \mathbb{R}^n$ is a* subgradient *of the function f in the point $u \in \mathbb{R}^n$ if $f(v) \geq f(u) + \langle s, v - u\rangle$ holds for all $v \in \mathbb{R}^n$.*

Let $\partial f(u)$ denote the set of the subgradients of f in u, i.e

$$\partial f(u) := \left\{ s \in \mathbb{R}^n : f(v) \geq f(u) + \langle s, v - u\rangle \quad \forall v \in \mathbb{R}^n \right\}. \tag{1}$$

Obviously $\partial f(u)$ is never empty and $|\partial f(u)| = 1$ if and only if f is differentiable in u.

Theorem 2. *For any $\lambda \geq 0$, the value of $L(\lambda)$ and a subgradient of the function L in the point λ can be computed using the max weight matching algorithm.*

Proof. It is easy to see that for any $M \in \mathcal{M}_\lambda$, $\Delta(M)$ is a subgradient of the function L in the point λ. In fact all the subgradients can be obtained in this way, i.e.

$$\partial L(\lambda) := \{\Delta(M) : M \in \mathcal{M}_\lambda\}.$$

Assuming now that a threshold value $\gamma \in \mathbb{R}_+$ is assigned, we look for the set

$$\Gamma := \{\lambda \in \mathbb{R}_+ : L(\lambda) \leq \gamma\}. \tag{2}$$

Due to the convexity of L, the set Γ is a closed interval. Moreover, it is also easy to see that starting the following Newton-Dinkelbach method from an upper and a lower bounds of Γ gives us the endpoints of Γ in finitely many steps. (See Figure 1 demonstrating the execution of the algorithm.)

```
Procedure Maxl(w,d,lstart)
begin
  l:=lstart;
  do
    M:=max_matching(w+l*d);
    l:=(w(M)-gamma)/d(M);
  while (w+l*d)(M)!=0;
  return l;
end
```

Using a technique originally developed by Radzik[6], it can be shown that

Theorem 3. *The above method terminates in* $O\left(|E| \log^2 |E|\right)$ *iterations, thus the full running time is* $O\left(|B||E|^2 \log^2 |E| + |B|^3 |E| \log^3 |E|\right)$.

Due to the space limitations the proof of this theorem is deferred to the full paper.

Note that the number of iterations (therefore the running time) is independent from the distance of the initial starting points and from the w and Δ values in the input. It solely depends on the size of the underlying graph.

We now apply the above treatment to our string searching problem. As it has already been mentioned in Section 2, our problem can be considered as a sequence of weighted matching problems over special auxiliary graphs, where

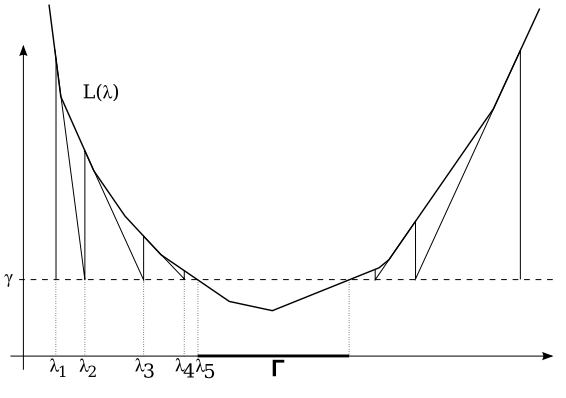


Fig. 1. The steps of Newton-Dinkelback method

an optimal matching in the auxiliary graph represents a best mapping of the pattern alphabet at that position. It has further been noticed that the edge weights change linearly between two bumps, therefore the problem breaks up $r_t r_p$ pieces of parametric bipartite graph matching problems (over the integral domain).

We mention that restricting ourselves to integer solutions does not cause any problem, as it suffices to round up the solutions into the right direction at the end of the algorithm.

Now, let us analyze the running time. The nodes of the graph represent the characters of the alphabets, therefore $|A| = |\Sigma_p|$ and $|B| = |\Sigma_t|$, whereas $|E| = |A||B| = |\Sigma_p||\Sigma_t|$. Thus the running time needed to solve a single instance of the parametric weighted matching problem is

$$O\left(|B||A|^2|B|^2\log^2(|A||B|) + |B|^3|A||B|\log^3(|A||B|)\right) = O\left(|\Sigma_p||\Sigma_t|^4\log^3|\Sigma_t|\right).$$

Note that this is simply a constant time algorithm if the size of the alphabets are constant. Thus for any fixed size alphabets the full running time of the algorithm is simply the number of bumps, i.e., $O(r_p r_t)$. If the size of the alphabet is part of the input, then the full running time is $O\left(r_p r_t|\Sigma_p||\Sigma_t|^4\log^3|\Sigma_t|\right)$.

4 Conclusion

We have presented a method for computing the parameterized matching on run-length encoded strings over alphabets of arbitrary size. The approach extends to alphabet of arbitrary yet constant size the $O\left(|r_p| \times |r_t|\right)$ performance previously available only for binary alphabets. For general alphabets, the bound obtained by the present method exhibits a substantial polynomial dependency on the alphabet size. This, however, should be contrasted with the general version of the problem, that can be solved in time $O(nm(\sqrt{m} + \log n))$. In other words, although the exponents are quite high in our expression, the overall complexity depends – in contrast with the convolution based approaches – on the run-length encoded lengths of the input and it is still polynomial in the size of the alphabets. The problem of designing an alphabet independent $O(|r_p| \times |r_t|)$ time algorithm for this problem is still open.

References

1. Amir, A., Farach, M., Muthukrishnan, S.: Alphabet Dependence in Parameterized Matching. Information Processing Letters 49, 111–115 (1994)
2. Baker, B.S.: Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. SIAM Journal of Computing 26(5), 1343–1362 (1997)
3. Baker, B.S.: Parameterized Pattern Matching: Algorithms and Applications. Journal Computer System Science 52(1), 28–42 (1996)
4. Apostolico, A., Erdős, P.L., Lewenstein, M.: Parameterized Matching with Mismatches. J. Discrete Algorithms 5(1), 135–140 (2007)

5. Hazay, C., Lewenstein, M., Sokol, D.: Approximate Parameterized Matching. ACM Transactions on Algorithms 3 (3), Article 29 (2007)
6. Radzik, T.: Fractional combinatorial optimization. In: Du, D., Pardalos, P. (eds.) Handbook of Combinatorial Optimization, vol. 1. Kluwer Academic Publishers, Dordrecht (1998)
7. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
8. Fredman, M.L., Tarjan, R.E.: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. Journal of ACM 34(3), 596–615 (1987)

Fast Bit-Parallel Matching for Network and Regular Expressions

Yusaku Kaneta, Shin-ichi Minato, and Hiroki Arimura

Hokkaido University, N14, W9, Sapporo 060-0814, Japan
{y-kaneta,minato,arim}@ist.hokudai.ac.jp

Abstract. In this paper, we extend the SHIFT-AND approach by Baeza-Yates and Gonnet (CACM 35(10), 1992) to the matching problem for network expressions, which are regular expressions without Kleene-closure and useful in applications such as bioinformatics and event stream processing. Following the study of Navarro (RECOMB, 2001) on the extended string matching, we introduce new operations called Scatter, Gather, and Propagate to efficiently compute ε-moves of the Thompson NFA using the Extended SHIFT-AND approach with integer addition. By using these operations and a property called the bi-monotonicity of the Thompson NFA, we present an efficient algorithm for the network expression matching that runs in $O(ndm/w)$ time using $O(dm)$ preprocessing and $O(dm/w)$ space, where m and d are the length and the depth of a given network expression, n is the length of an input text, and w is the word length of the underlying computer. Furthermore, we show a modified matching algorithm for the class of regular expressions that runs in $O(ndm \log(m)/w)$ time.

1 Introduction

Recent emergence of massive text and sequence data in networks has attracted much attention to string processing technologies [1,3,4,12,15,17,19]. In this paper, we study the *regular expression matching problem*, which is one of the most important problems in string processing. Especially, for the last decades, approaches based on efficient NFA simulation have been extensively studied for restricted subclasses of regular expressions, namely, the four-russian approach for the class REG of regular expressions [4,12]; the SHIFT-AND approach for the class STR of strings [3,19], and the SHIFT-ADD approach for the classes of k-mismatch string patterns [3,8]. In particular, Navarro and Raffinot [14,15] presented efficient bit-parallel approach, called *Extended SHIFT-AND approach* tailored to a restricted but useful subclass EXT of *extended string patterns*, which are regular expressions in linear form, such as $R_0 = (\texttt{[AB]}^+)(\texttt{B.}\{\texttt{1,3}\})(\texttt{[BC]?})(\texttt{.}^*)\texttt{C}$, that consists of letters $a \in \Sigma$, wildcards ".", classes of letters $\alpha = [ab\cdots]$, optional letters $\alpha?$, bounded repeats $\alpha\{x, y\}$, and unbounded repeats α^* and α^+, where $\alpha \subseteq \Sigma$. In this approach, Navarro and Raffinot [15] nicely extended the original approach of [3,19] by introducing a new bit-parallel simulation technique, called the propagation, with the use of integer addition "+" (or subtraction "−") in addition to usual Boolean operations on RAM to deal with a special case of

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 372–384, 2010.

ε-closure caused by optional letters $\alpha?$ and bounded repeats $\alpha\{x, y\}$ in extended string patterns as well as unbounded repeats α^* with the use of an extended letter mask.

In this paper, inspired by the work by Navarro and Raffinot [15], we study the pattern matching problem for a special class NET of regular expressions, called *network expressions*, which are introduced in Myers [13]. A network expression (over strings) in NET is a regular expression without Kleene-closure, that is, an expression constructed recursively from strings in STR applying ε-edges, concatenation, and union. Similarly, we can define the class EXNET of *extended network expressions*, which are network expressions over extended string patterns in EXT. For example, $R_1 = \mathtt{A(BA|CD)(CD|AB)B}$ and $R_2 = \mathtt{A(AB|B?)(B?.^*|AB)C}$ are examples of expressions in NET and EXNET, respectively. Network expressions and extended network expressions are widely used in applications in the various fields including such as bioinformatics [13], event stream processing [1], and network intrusion detection systems [17].

As main results in this paper, we show the followings. Let $RAM(op)$ denote a unit-cost random access machine equipped with a set op of arithmetic operations in addition to the standard Boolean operations "&", "|", "$\sim$", and "$\oplus$". We present an efficient algorithm that solves the regular expression matching problem for the classes NET and EXNET in $O(nd\lceil m/w\rceil)$ time using $O(dm + |\Sigma|\lceil m/w\rceil)$ preprocessing and $O(d\lceil m/w\rceil + |\Sigma|\lceil m/w\rceil)$ space on $RAM(+)$, where Σ is a fixed alphabet, m and d are the length and the depth of an input expression R, and n is the length of an input text T over Σ. Furthermore, we show that the regular expression matching problem for the full class REG can be solved in $O(nd\lceil m/w\rceil \log m)$ time using $O(dm \log m + |\Sigma|\lceil m/w\rceil)$ preprocessing and $O(d\lceil m/w\rceil \log m + |\Sigma|\lceil m/w\rceil)$ space on $RAM(+)$. If we allow the reversal of bitmasks inv as a primitive, then the problem can be solved in the same time, preprocessing, and space complexities as NET and EXNET on $RAM(+, inv)$.

To obtain above results, we devise the following techniques to achieve efficient bit-parallel simulation of Thompson NFA (TNFA, for short) for classes NET and EXNET. A key of NFA simulation for the full class REG is an efficient simulation of ε-closure in TNFA as mentioned in the previous works [4,12]. Hence, by extending the previous SHIFT-AND [3,19] and Extended SHIFT-AND [15] approaches, we introduce a set of new bit-parallel simulation operations, called Scatter, Gather and Propagate operations to deal with the long succession and the branching of ε-edges caused by concatenation and union in network expressions in NET and EXNET. Furthermore, we also devise a transformation technique of a given TNFA into a special form of NFA that satisfies a property called "*bi-monotonicity*" of ε-moves by attaching new ε-edges to all subexpressions whose initial and final states are ε-reachable in the original expression. Furthermore, we introduce the *barrel shifter* technique for implementing backward ε-edges for REG based on a well-known technique in the VLSI circuit design.

The advantages of our approach to regular expression matching are summarized as follows. (i) Simple and efficient: Since our algorithm naturally exploits the composition structure of TNFAs and does not use complex module decompositions as

in [4], it is particularly efficient for regular expressions with small depth. (ii) Hardware friendly: Since it uses only simple bit-operations and addition/subtraction and avoids the heavy use of table-lookup, it has potential to be implemented on modern parallel hardwares with simple structure, such as GPGPUs or FPGAs. To confirm the above observations, we developed a hardware implementation of a multiple regular expression matching system on FPGA based on the proposed algorithm. The experimental results showed that the system could match 256 patterns at the same time against a text stream with throughput of 1.6Gbps and 0.5Gbps in total for NET and EXNET, respectively.

Related works. There are a number of researches on the regular expression matching problem for REG other than the Extended SHIFT-AND approach. In the Table-Lookup approach, Myers [12] developed an $O(nm/\log n)$ time and space algorithm. Improving the space complexity of [12], Bille and Thorup [5] presented $O(nm(\log\log n)/(\log n)^{3/2} + n + m)$ time and $O(m)$ space algorithm. For DFA simulation by Brute force determinization, Navarro and Raffinot [16] proposed an $O(n)$ time and $O(m2^m)$ bits space algorithm using DFA simulation of Glushkov's NFAs, while Wu and Manber [19] presented an $O(n)$ time and $O(m2^{2m})$ bits space algorithm based on the DFA simulation of Thompson's NFAs. Champarnaud *et al.* [7] improves this result by obtaining an expected exponential reduction of the space complexity. Papers [6,13,14] study pattern matching with bounded and unbounded gaps.

Organization of this paper. In Section 2, we give basic definitions and notations. In Section 3, we present our algorithm for the class NET of network expressions as well as extended network expressions EXNET. In Section 4, we give a modified algorithm for the full class REG of regular expressions. In Section 5, we show experimental results on the hardware implementation of the proposed algorithms. In Section 6, we conclude this paper. For details, please consult the full paper [10] and the companion paper [11].

2 Preliminary

In this section, we give basic definitions and notations in the regular expression matching problem according to [2,12,15].

Regular expression matching problem. Let $\mathbf{N} = \{0, 1, 2, \ldots\}$. For $i \leq j$, we define $[i..j] = \{i, i+1, \ldots, j\}$. Let Σ be a finite alphabet of *letters*. A *string* on Σ is a sequence $S = s_1 \cdots s_n$ of letters, where $s_i \in \Sigma$ for every i. For every $1 \leq i \leq j \leq n$, We denote by $S[i] = s_i \in \Sigma$, by $S[i..j]$ the substring $s_i \cdots s_j$, and by ε the *empty string*. If $i > j$, we define $S[i..j] = \varepsilon$. For a string S, we denote by $|S|$ the *length* (or the *size*) of S.

The class REG of *regular expressions* on Σ is defined recursively as follows: (1) If $a \in \Sigma \cup \{\varepsilon\}$ then $a \in$ REG. (2) If $R_1, \ldots, R_n \in$ REG then $(R_1 \cdots R_n)$, $(R_1 | \cdots | R_n)$, $(R_1)^* \in$ REG. In this paper, regular expressions are *unbounded*, i.e., $n \geq 1$, while $n = 2$ in the standard definition [4,15]. The *length* (or the *size*) of R is defined by the number $||R||$ of symbols from $\Sigma \cup \{\varepsilon, \cdot, |, *\}$ appearing in R.

For a regular expression R, the *parse tree* T_R, the *language* $L(R) \subseteq \Sigma^*$, and the *depth* (or the *height*) $d(R)$, respectively, are defined in the standard way [15]. Let $\mathcal{C} \subseteq \mathsf{REG}$ be any subclass of REG. A *pattern* is any regular expression $R \in \mathcal{C}$ and a *text* is a string $T \in \Sigma^*$ over Σ. We say that a regular expression R of length m *occurs* in a text T of length n if there exist some $i \leq j$ such that $T[i..j] \in L(R)$ holds. Then, the index j is called the *end position* of R in T. Now, we state our problem below. *The regular expression matching problem for a class* $\mathcal{C} \subseteq \mathsf{REG}$ is, given a regular expression $R \in \mathcal{C}$ of length m and an input text T of length n, to output the set of all end positions of R in T.

Subclasses of regular expressions. We introduce the classes STR, EXT, NET, and EXNET of string patterns, extended string patterns, network expressions, and extended network expressions, respectively, as follows. A *string pattern* over Σ is a string $R \in \Sigma^*$. An *extended string pattern* [14] over Σ is a regular expression $R = r_1 \cdots r_m$ $(m \geq 0)$, where for every $1 \leq i \leq m$, r_i is one of the following forms: (i) letters $a \in \Sigma$, (ii) wildcards ".", (iii) classes of letters $\alpha = [ab\cdots]$, (iv) optional letters $\alpha?$, (v) bounded repeats $\alpha\{x, y\}$, and (vi) unbounded repeats α^* and α^+, where $\alpha \subseteq \Sigma$. The semantics of the additional operations is given by the notational equivalence: "." $\equiv \Sigma$, $\alpha? \equiv (\alpha|\varepsilon)$, $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$, and $\alpha^+ \equiv (\alpha\,\alpha^*)$.

A *network expression* (over strings) in NET [13] is a regular expression over strings, that is, a regular expression obtained from strings, ε-edges, concatenation, and union. An *extended network expression* in EXNET [13] is a network expression over extended string patterns in EXT. For example, $R_0 =$ `A(BA|C?)(C*|AB)B`, $R_1 =$ `A(BA|CD)(CD|AB)B`, and $R_2 =$ `A(AB|B?)(B?.*|AB)C` are examples of expressions over $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ in EXT, NET, and EXNET, respectively.

Model of computation. As the model of computation, we assume a *unit-cost RAM* with word length w [2]. For any bitmask length $L \geq 0$, A *bitmask* is a vector $X = b_L \cdots b_1 \in \{0,1\}^L$ of L bits. for a bit $b \in \{0,1\}$, we denote by b^k the bitmask consisting of k copies of b. For bitmasks with $L \leq w$, we assume that the following Boolean and arithmetic operations are executed in $O(1)$ time: *bitwise* AND "&", *bitwise* OR "|", *bitwise* NOT "$\sim$", *bitwise* XOR "$\oplus$", *left shift* "$\ll$", *right shift* "$\gg$" on $RAM()$, *integer addition* "+" and *integer subtraction* "$-$" on $RAM(+)$. The space complexity is measured in the number of words.

3 Fast Bit-Parallel Algorithm for Extended Network Expressions

In this section, we present an efficient algorithm that receives any input extended network expression R in NET or EXNET with length m and depth d and an input text T on Σ with length n, and finds all the occurrences of R in T in $O(nd\lceil m/w \rceil)$ time using $O(dm + |\Sigma|\lceil m/w \rceil)$ preprocessing and $O(d\lceil m/w \rceil + |\Sigma|\lceil m/w \rceil)$ space on $RAM(+)$. In what follows, we assume an input regular expression R with length m and depth d and the input text T with length n.

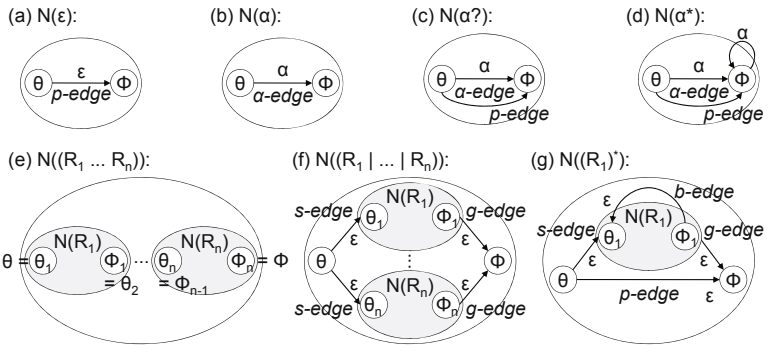


Fig. 1. The construction of Thompson automata (TNFAs)

3.1 Basic NFA Simulation Algorithm

We show the outline of our algorithm BP-Match. First, in the preprocessing phase, we construct a set of the bitmasks M_R from a given extended network expression $R \in \mathsf{EXNET}$, and then, in the runtime phase, we search for all the end positions of R in an input text T based on NFA simulation of N_R.

Algorithm. BP-Match($T \in \Sigma^*$: an input text, $R \in \mathsf{EXNET}$: an extended network expression)

PREPROCESS:

(1) Transform R to its expanded form $\mathsf{Expand}(R)$.
(2) Construct the TNFA N_R from $\mathsf{Expand}(R)$.
(3) Construct a set M_R of the bitmasks from N_R.

RUNTIME:

(4) Simulate N_R on T by using M_R

Transformation of a regular expression to its expanded form. As preprocessing, we first expand all the occurrences of bounded repeats $\alpha\{x, y\}$ and unbounded repeats α^+ in an input expression R using the equivalence $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$ and $\alpha^+ \equiv (\alpha\alpha^*)$, respectively. Furthermore, we apply the operation, called *bypassing*, that replaces all the subexpressions S in R such that $\varepsilon \in L(S)$ with the expression $S' \equiv (S \mid \varepsilon)$. This bypassing does not change the language $L(R)$. We denote by $\mathsf{Expand}(R)$ the resulting extended network expression. The properties of $\mathsf{Expand}(R)$ will be examined later.

Construction of TNFA. We construct the parse tree T_R of R as shown in Fig. 2. By the construction in Fig. 1, we compute the Thompson NFA (TNFA, for short) $N(R) = (V, E, \theta, \phi)$ of $\mathsf{Expand}(R)$ as shown in Fig. 3, where $V = \{0, \ldots, L\}$ for $L \geq 0$. As a special case, for the subexpression $S' \equiv (S|\varepsilon)$ introduced by the bypassing, we add the ε-edge to S directly connecting from θ_S to ϕ_S instead of rule (f). In Fig. 3, we show an example of TNFA for

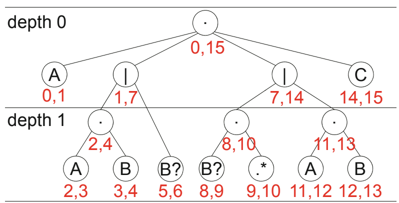


Fig. 2. The parse tree of an extended network expression $R_2 = \mathtt{A(AB|B?)(B?.^*|AB)C}$

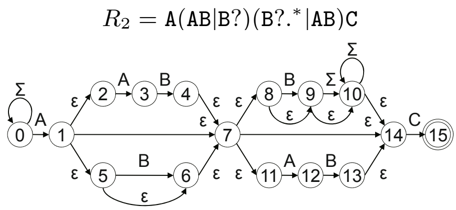


Fig. 3. An extended network expression R_2 and its TNFA $N_2 = N(R_2)$

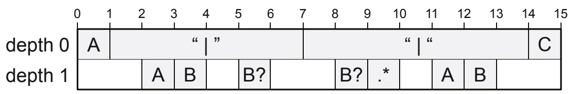


Fig. 4. The bit-position assignment for subexpressions of $\mathsf{Expand}(R_2)$, where the depth is compressed by ignoring internal nodes labeled with concatenation "·"

$R_2 = \mathtt{A(AB|B?)(B?.^*|AB)C}$. For each node v of T_R, let $S = S(v)$ be the subexpression of $\mathsf{Expand}(R)$ associated with v and $N_S = N(S) = (V_S, E_S, \theta_S, \phi_S)$ be its corresponding TNFA, called the *component TNFA* for v, with a state set V_S, an edge set E_S, initial and final states θ_S and ϕ_S. By depth-first search of T_R from left to right, we assign the set $V(v) = \{\theta_S, \phi_S\} \subseteq [0..L]$ of the initial and final states of S to each node v of T_R as in Fig. 2, and define the depth $d(v)$ by the number of non-concatenation nodes on the path from the root to v. For each $x \in V(v)$, we define $d(x) = d(v)$, and for each subexpression $S = S(v)$, we associate the interval $I_S = [\theta_S..\phi_S] \subseteq [0..L]$. In Fig. 4, we show a bit-position assignment related to T_R in Fig. 2. A labeled edge $e = (u, \beta, v) \in E_S$ is an α*-edge* if $\beta \subseteq \Sigma$, and is an ε*-edge* if $\beta = \varepsilon$.

Efficient NFA simulation. Next, we describe the standard *NFA simulation* method developed by Thompson [2,18] that most of the previous regular expression matching algorithms [3,4,12,15,19] employ. In Thompson's algorithm [18], the current status of the TNFA $N_R = (V_R, E_R, \theta_R, \phi_R)$ is represented by a set $D \subseteq V_R$ of *active states*. Then, we define the following operations: Init_N returns the set $\{\theta_R\}$; Accept_N returns the set $\{\phi_R\}$; For any letter $c \in \Sigma$, $\mathsf{Move}_N(D, c)$ returns the set $\{y \in V_R \mid y$ is reachable from some $x \in D$ by exactly one α-edge such that $c \in \alpha\ \}$; $\mathsf{EpsClo}_N(D)$ returns the set $\{y \in V_R \mid y$ is reachable from some $x \in D$ by zero or more ε-edges $\}$, called the ε*-closure* of D.

In Fig. 5, we show the algorithm $\mathsf{RunTNFA}$ that simulates the computation of the TNFA N_R on an input text T. We can show the following lemma [18].

Lemma 1 (Thompson [18]). *For any input text T, the algorithm* $\mathsf{RunTNFA}$ *in Fig. 5 correctly solves the regular expression matching problem for* REG.

Algorithm RunTNFA($T = t_1 \cdots t_n$: an input text, $N(R)$: a TNFA)

```
 1: D ← Init_N; //initial state set
 2: D ← EpsClo_N(D); //ε-closure
 3: for i ← 1, ..., n do
 4:    if D ∩ Accept_N ≠ ∅ then
 5:       report "match at i − 1";
 6:    D ← Move_N(D, t_i); //α-edges
 7:    D ← EpsClo_N(D); //ε-closure
 8: end for
 9:
10:
```

Fig. 5. The algorithm RunTNFA for NFA simulation in the runtime phase

Procedure $\mathsf{EpsClo}_N(D$: the state set for a TNFA $N(R))$

```
 1: for k ← d(R), ..., 1 do
 2:    D ← Propagate(D, k);
 3:    D ← Gather(D, k − 1);
 4: end for
 5: D ← Propagate(D, 0);
 6: for k ← 1, ..., d(R) do
 7:    D ← Scatter(D, k − 1);
 8:    D ← Propagate(D, k);
 9: end for
10: return D;
```

Fig. 6. The procedure EpsClo_N for computing ε-closure

Fine classification of ε-moves. It is not hard to efficiently implement Move_N either by using *table-lookup* [12] or SHIFT-AND approach [3,19], while it is not straightforward to efficiently implement EpsClo_N since we have to compute ε-closure. The key of our algorithm is an efficient implementation of EpsClo_N based on a set of new bit-parallel operations Scatter, Gather, and Propagate defined as follows.

In the construction (a)–(g) of TNFA in Fig. 1, we categorize ε-edges in a component TNFA $N(S)$ into four types: (i) $e = (\theta, \varepsilon, \theta_i)$ in (f) or (g) is a *scatter edge* (s-edge) with depth $d(\theta)$, (ii) $e = (\phi_i, \varepsilon, \phi)$ in (f) or (g) is a *gather edge* (g-edge) with $d(\phi)$, (iii) $e = (\theta, \varepsilon, \phi)$ in (a), (c), (d), or (g) is a *propagate edge* (p-edge) with $d(\theta) = d(\phi)$, and (iv) $e = (\phi_i, \varepsilon, \theta_i)$ in (g) is a *back edge* (b-edge) with $d(\theta_i) = d(\phi_i)$, where θ and ϕ are the initial and the final states of $N(S)$. We classify the ε-edge introduced by bypassing as a propagate edge. For scatter, gather, propagate, and back edges in $N(S)$, we assign the depth of the outermost node θ or ϕ of $N(S)$. The next lemma gives a characterization of ε-edges.

Lemma 2. *If $e = (u, \varepsilon, v)$ is an ε-edge in the TNFA N_R for $R \in \mathsf{EXNET}$, then $\Delta = d(v) - d(u) \in \{+1, 0, -1\}$ holds. Moreover, (i) if $\Delta = +1$, e is a scatter edge, (ii) if $\Delta = -1$, e is a gather edge, and (iii) if $\Delta = 0$, e is a propagate edge.*

For any set $D \subseteq V$ and any $k = 0, \ldots, d(R)$, we define $\mathsf{Scatter}(D, k)$ (or $\mathsf{Gather}(D, k)$) the sets of states from some states in D reachable by exactly one scatter edge (or one gather edge, resp.) with depth k. On the other hand, the set $\mathsf{Propagate}(D, k)$ is defined by the ε-closure of D restricted by the propagate edges with depth k. For any component TNFA S, an *ε-block* $B \subseteq V_S$ is a set of states that induces a maximal connected component consisting only of propagate edges. By construction of TNFA and bypassing, we can see that any such ε-block forms a chain. Clearly, all states in B have the same depth d, which is called the *depth* of B. For example, an expression $R_2 = \mathtt{A(AB|B?)(B?.^*|AB)C}$ in Fig. 3 has three ε-blocks, $B_1 = \{1, 7, 14\}$, $B_2 = \{5, 6\}$, and $B_3 = \{8, 9, 10\}$.

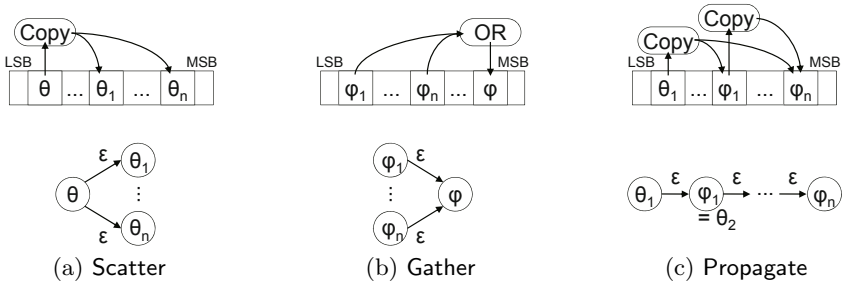


Fig. 7. The bit-operations and the corresponding parts of TNFAs

Now, we show the key lemma, called the bi-monotonicity lemma on bypassing transformation. For any $d, d' \in \mathbf{N}$, we define $d \leq_1 d'$ if $|d - d'| \leq 1$ holds. For any states x, y in TNFA $N(R)$, a ε-path $\pi = (x_1 = x, \ldots, x_n = y) \in (V_R)^*$ is said to be *bi-monotone* if there exists some state x_k $(1 \leq k \leq n)$ such that $d(x_1) \leq_1 \cdots \leq_1 d(x_k)$ and $d(x_k) \geq_1 \cdots \geq_1 d(x_n)$ hold, that is, the depth sequence for the first half is non-decreasing and the latter half is non-increasing. By induction on the construction of TNFA, we can show the next lemma.

Lemma 3 (bi-monotonicity lemma). *Let x, y be any states in* Expand(R). *If π be any ε-path from x to y, then there also exists some bi-monotone ε-path from x to y in* Expand(R).

Based on the bi-monotonicity of an expanded version of TNFA, we present in Fig. 6 the procedure EpsClo_N that computes the ε-closure for EXNET.

Lemma 4. *Suppose that* Scatter, Gather, *and* Propagate *operations are correctly implemented for $R \in$ EXNET with depth $d(R)$. Then, the algorithm* EpsClo *in Fig. 6 correctly computes the ε-closure* $\mathsf{EpsClo}_{N(R)}(D)$ *of any state set D.*

Proof. The soundness is obvious from construction. The completeness follows that if a state y is ε-reachable from a state x, then the applications of operators in the order of the regular expression (Propagete.Gather)* Propagete (Scatter.Propagete)* moves x to y by the existence of a bi-monotone ε-path by Lemma 3. Since this is what EpsClo_N does, the lemma is proved. □

3.2 Bit-Parallel Implementation

To simulate the TNFA N_R for an extended network expression Expand(R), we use a set M_R of bitmasks of L bits $\mathtt{CHR}[c]$, $\mathtt{REP}[c]$, $\mathtt{BLK}_\tau[k]$, $\mathtt{SRC}_\tau[k]$, and $\mathtt{DST}_\tau[k] \in \{0,1\}^L$, for every $c \in \Sigma$, $0 \leq k \leq d(R)$, and $\tau \in \{\mathtt{S}, \mathtt{G}, \mathtt{P}\}$, where L is the number of the states of N_R. Then, by further generalizing the Extended SHIFT-AND approach, we simulate the ε-closure operations Scatter, Gather, and Propagate as follows. Let $N_S = (V, E, \theta, \phi)$ be any component TNFA in depth k.

Simulation of Move operation. *Preprocess*: Let $e = (\theta, \alpha, \phi) \in E$ be an α-edge of N_S, where $\alpha \subseteq \Sigma$. To implement the Move operation, we precompute the

following bitmasks. For every letter $c \in \alpha$ and every N_S, we define: (M.1) $\texttt{CHR}[c]$ has 1 in the bit-position $j = \phi$. (M.2) $\texttt{REP}[c]$ has 1 in the bit-position $j = \phi$ such that $\theta = \phi$ holds, that is, an α-edge e is a self-loop, equivalently, either $S = \alpha^*$ or $S = \alpha^+$.

Runtime: To simulate the $\mathsf{Move}(D, t_i)$, we perform

$$D \leftarrow (((D \ll 1)\ \&\ \texttt{CHR}[t_i])\mid 1)\mid (D\ \&\ \texttt{REP}[t_i]); \tag{1}$$

where $t_i \in \Sigma$ be an input letter. This code is the same as the code for α-moves in the Extended SHIFT-AND approach [15]. For the details, see [15].

Simulation of Scatter **operation.** *Preprocess*: Let $e = (\theta, \varepsilon, \theta_i) \in E$ be a scatter edge of N_S. To implement the Scatter operation, we precompute the following bitmasks. For every depth k and every N_S, we define: (S.1) $\texttt{BLK}_\texttt{S}[k]$ has 1 in the bit-position $j = \phi - 1$. (S.2) $\texttt{SRC}_\texttt{S}[k]$ has 1 in the bit-position $j = \theta$. (S.3) $\texttt{DST}_\texttt{S}[k]$ has 1 in the bit-position j iff $j = \theta_i$ for all i (Fig. 7).

Runtime: To simulate the $\mathsf{Scatter}(D, k)$, we perform

$$D \leftarrow D \mid ((\texttt{BLK}_\texttt{S}[k] - \{D\ \&\ \texttt{SRC}_\texttt{S}[k]\})\ \&\ \texttt{DST}_\texttt{S}[k]); \tag{2}$$

Firstly, by the formula $(D\ \&\ \texttt{SRC}_\texttt{S}[k])$, we extract the values of source bits from D. Then, by subtracting the values from $\texttt{BLK}_\texttt{S}[k]$, all the destination bits are set to 1 if the source bits is 1, and all to 0 otherwise. Note that this is done by carry propagation of subtraction "$-$". Finally, we extract the destination bits by AND-ing the result with $\texttt{DST}_\texttt{S}[k]$, and put all the destination bits to D.

Simulation of Gather **operation.** *Preprocess*: Let $e = (\phi_i, \varepsilon, \phi) \in E$ be a gather edge of N_S. For every depth k and every N_S, we define: (G.1) $\texttt{BLK}_\texttt{G}[k]$ has 1 in the bit-position $j \in [\theta + 1..\phi - 1]$. (G.2) $\texttt{SRC}_\texttt{G}[k]$ has 1 in the bit-position j iff $j = \phi_i$ for all i. (G.3) $\texttt{DST}_\texttt{G}[k]$ has 1 in the bit-position $j = \phi$ (Fig. 7).

Runtime: To simulate the $\mathsf{Gather}(D, k)$, we do the following

$$D \leftarrow D \mid ((\texttt{BLK}_\texttt{G}[k] + \{D\ \&\ \texttt{SRC}_\texttt{G}[k]\})\ \&\ \texttt{DST}_\texttt{G}[k]); \tag{3}$$

Since this code is similar to one of Scatter except that Gather uses addition, while Scatter uses subtraction, we omit the details.

Simulation of Propagate **operation.** *Preprocess*: Let $e = (\theta, \varepsilon, \phi) \in E$ be a propagate edge of N_S. For every k and ε-block B with depth k, we define: (P.1) $\texttt{BLK}_\texttt{P}[k]$ has 1 in the bit-position $i \in B$. (P.2) $\texttt{SRC}_\texttt{P}[k]$ has 1 in the bit-position $i = min(B)$. (P.3) $\texttt{DST}_\texttt{P}[k]$ has 1 in the bit-position $i = max(B)$ (Fig. 7).

Runtime: To simulate the $\mathsf{Propagate}(\mathsf{D}, k)$, we perform the following

$$A \leftarrow (D\ \&\ \texttt{BLK}_\texttt{P})\mid \texttt{DST}_\texttt{P}[k]; \tag{4}$$

$$D \leftarrow D \mid (\texttt{BLK}_\texttt{P}[k]\ \&\ ((\sim (A - \texttt{SRC}_\texttt{P}[k])) \oplus A)); \tag{5}$$

The above code works since any ε-block is a chain in the same depth, and thus the propagation of the carry bits in the subexpression $(A - \texttt{SRC}_\texttt{P}[k])$ correctly implements the ε-closure on a chain as shown in [14,15].

3.3 Main Results

From Navarro and Raffinot (Sec. 1.3.1, [15]), we know that the integer addition and subtraction can be executed in $O(\lceil m/w \rceil)$ time and space by simulating carry propagation. Combining this and the arguments in the previous section, we have the following lemma.

Lemma 5. *By the above construction, the* Move(D, c), Scatter(D, k), Gather (D, k), *and* Propagate(D, k) *operations for* $N(R)$ *are correctly implemented to run in* $O(\lceil m/w \rceil)$ *time on* $RAM(+)$, *where* $c \in \Sigma$, $0 \leq k \leq d(R)$, D *is any* m*-bit mask,* m *is the number of states in* $N(R)$ *and* w *is the word length.*

From Lemma 5, in the *large automata* case with $m > w$, we can use inexpensive simulation of primitive operations on $RAM(+)$ instead of expensive module decomposition technique used tabling-based algorithms as in [4,12]. This will be an advantage of our algorithm in implementing it on parallel hardwares such as GPGPUs and FPGAs. Now, we show the main result of this paper.

Theorem 1. *The algorithm* BP-Match *solves the regular expression matching problem for* NET *and* EXNET *of network and extended network expressions in* $O(nd\lceil m/w \rceil)$ *time using* $O(dm + |\Sigma|\lceil m/w \rceil)$ *preprocessing and* $O(d\lceil m/w \rceil + |\Sigma|\lceil m/w \rceil)$ *space, where* $n = |T|$, $m = ||R||$, $d = d(R)$, w *is the word length.*

Proof. The correctness follows from Lemma 1, Lemma 4, and Lemma 5. Then, the result immediately follows from that the for-loop is executed at most $d(R)$ times and each code can be executed in $O(\lceil m/w \rceil)$ time from Lemma 5. □

4 Extension for General Regular Expressions

To generalize our algorithm in Sec. 3 for the full class REG in the Extended SHIFT-AND approach, we need to simulate backward ε-edges corresponding to the Kleene-closure "$*$". However, the backward ε-edges from lower to higher bits seems hard to compute on $RAM(+)$. To overcome this difficulty, we introduce a technique called *barrel shifter* as follows.

The idea is to decompose each backward ε-moves from higher to lower bits having the length J bits into a series of right-shifts "$\gg$" having the widths $2^0 = 1, 2^1 = 2, \ldots, 2^\ell$, where $\ell = \lceil \log \delta \rceil$ and $\delta = O(m)$ is the maximum length of the backward ε-edges in TNFA R. More precisely, for each back edge e in a certain depth of R, if the edge e has the width $J \geq 0$, we have the unique binary expansion $bin(J) = J_{\ell-1} \cdots J_0 \in \{0,1\}^L$ such that $J = \sum_{i=0}^{\ell-1} J_i 2^i$. For each $k = 0, \ldots, d(R)$ and $i = 0, \ldots, \ell - 1$, the bitmask $\mathtt{BLK_B}[k][i]$ is defined by: for each back edge $e = (\phi_i, \varepsilon, \theta_i)$ in depth k, we fill the interval $I_e = [\theta_i..\phi_i]$ with 1's if $J_k = 1$ and with 0's if $J_k = 0$. In run-time, we set $jmp \leftarrow 0$, and repeatedly perform $D \leftarrow (D \;\&\; \sim \mathtt{BLK_B}[k][i]) \mid (D \;\&\; \mathtt{BLK_B}[k][i]) \gg jmp); jmp \leftarrow jmp \ll 1$. From the construction, this operation can be implemented in $O(d\lceil m/w \rceil \log m)$ time using $O(dm \log m)$ preprocessing and $O(d\lceil m/w \rceil \log m)$ space.

Table 1. Summary of experimental results on the hardware implementation, where #op, #add, #reg, #bram, and #slice, are the numbers of 32 bit operations, 32 bit integer additions, registers, block RAM lines, resp., per PMM. #pat and #char are the number and the total size of input patterns, resp.

Class	#op	#add	#reg	#bram	#slice	frequency	throughput	load time	#pat	#char
STR	5	0	3	256	54	363 MHz	2.9 Gbps	0.182 ms	256	8,192
EXT	11	1	6	512	123	202 MHz	1.6 Gbps	0.328 ms	128	4,096
EXNET	20	9	24	512	736	65 MHz	0.5 Gbps	1.055 ms	128	4,096

Theorem 2. *The regular expression matching problem for the class* REG *can be solved in* $O(nd\lceil m/w\rceil \log m)$ *time using* $O(dm \log m + |\Sigma|\lceil m/w\rceil)$ *preprocessing and* $O(d\lceil m/w\rceil \log m + |\Sigma|\lceil m/w\rceil)$ *space.*

As an alternative, if there are at most constant number of back edges with mutually distinct lengths, then we can replace the $O(\log m)$ term with $O(1)$. As other option, if the $O(1)$-bit-reversal inv is available, we can also replace the $O(\log m)$ term with $O(1)$ on $RAM(+, inv)$ by simulating backward ε-moves by Scatter (or Gather) and inv. Thus, we obtain the same complexity as Theorem 1.

5 Experimental Results

To evaluate the performance, we implemented our regular expression matching algorithm on *FPGA* in Verilog-HDL for STR, EXT, and EXEXT. We designed the algorithm as a collection of up to 256 pattern matching modules (PPMs) working simultaneously [11], where the word length is $w = 32$ bits, and masks are stored in block RAMs and a set of registers. We used the Xilinx ISE Design Suite 10.1 and Synopsys VCS development tools. Having targetted an *FPGA* device, Xilinx Virtex-5 LX330 with -1 speed grade, which had 51,840 slices and 288 block RAMs with 36 Kbits, we could install up to 256 PPMs. For more details of the experiments, see the companion paper [11]. Table. 1 shows the summary of the experimental results on our hardware. The #bram is given by the number of block RAMs times $|\Sigma| = 256$. Then, we can observe that our hardware achieves the high throughput of 0.5 Gbps for the class EXNET and of 1.6 Gbps for the class EXT, which is hard to achieve by software implementation on the current general CPUs. Hence, our algorithm is suitable to hardware implementation.

6 Conclusion

In this paper, we presented an efficient bit-parallel algorithm that solves the regular expression matching problem for the class EXNET of extended network expressions in $O(nd\lceil m/w\rceil)$ time using $O(d\lceil m/w\rceil)$ space and $O(dm)$ preprocessing by extending the Extended SHIFT-AND approach [15]. Furthermore, we show that the problem for the full class REG of regular expressions is solvable

in $O(nmd \log w/w)$ time on $RAM(+)$. Experiments on its hardware implementation showed that the proposed algorithm is suitable to parallel execution on hardwares. Other advantage is the guaranteed worst-case time complexity. Thus, it may be useful as a base algorithm for other approaches such as filtration as mentioned in [8,15]. Application of the Extended SHIFT-AND to tree and XML matching [9] will be an interesting future research.

Acknowledgements

The authors would like to thank Osamu Watanabe, Ryuhei Uehara, Takashi Horiyama, Yoshio Okamoto, Thomas Zeugmann, Shinobu Nagayama, Ayumi Shinohara, Masayuki Takeda, for their discussions and valuable comments, and to Shingo Yoshizawa, and Yoshikazu Miyanaga for their warm encouragements and constant supports on VLSI circuit design. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, and interdisciplinary research project "*high performance FPGA-based string matching hardwares*" under MEXT/JSPS Global COE Program, FY2007–2011.

References

1. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proc. ACM SIGMOD 2008, pp. 147–160 (2008)
2. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
3. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Communications of the ACM 35(10), 74–82 (1992)
4. Bille, P.: New algorithms for regular expression matching. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 643–654. Springer, Heidelberg (2006)
5. Bille, P., Thorup, M.: Faster regular expression matching. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 171–182. Springer, Heidelberg (2009)
6. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proc. SODA 2010, pp. 1297–1308 (2010)
7. Champarnaud, J.-M., Coulon, F., Paranthoën, T.: Compact and fast algorithms for safe regular expression search. Int. J. Comput. Math. 81(4), 383–401 (2004)
8. Grabowski, S., Fredriksson, K.: Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time. Inf. Proccess. Lett. 105(5), 182–187 (2008)
9. Kaneta, Y., Arimura, H.: Faster bit-parallel algorithm for unordered pseudo-tree matching and tree homeomorphism. In: Proc. of IWOCA 2010 (July 2010); also appeared as Hokkaido U., TCS-TR-A-10-43 (May 2010)
10. Kaneta, Y., Minato, S., Arimura, H.: Fast bit-parallel matching for network and regular expressions. Hokkaido U., TCS-TR-A-10-47 (August 2010)
11. Kaneta, Y., Yoshizawa, S., Minato, S., Arimura, H., Miyanaga, Y.: Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching. Hokkaido U., TCS-TR-A-10-45 (June 2010) (submitting)

12. Myers, E.W.: A four-russian algorithm for regular expression pattern matching. Journal of the ACM 39(2), 430–448 (1992)
13. Myers, E.W.: Approximate matching of network expessions with spacers. J. Computatinal Biology 3(1), 33–51 (1996)
14. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In: Proc. RECOMB 2001, pp. 231–240 (2001)
15. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences, Cambridge (2002)
16. Navarro, G., Raffinot, M.: New techniques for regular expression searching. Algorithmica 41, 89–116 (2005)
17. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: Proc. IEEE FCCM 2001, pp. 227–238 (2001)
18. Thompson, K.: Programming techniques: regular expression search algorithm. Communications of the ACM 11(6), 419–422 (1968)
19. Wu, S., Manber, U.: Fast text searching: allowing errors. Communications of the ACM 35(10), 83–91 (1992)

String Matching with Variable Length Gaps

Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind

Technical University of Denmark

Abstract. We consider string matching with variable length gaps. Given a string T and a pattern P consisting of strings separated by variable length gaps (arbitrary strings of length in a specified range), the problem is to find all ending positions of substrings in T that match P. This problem is a basic primitive in computational biology applications. Let m and n be the lengths of P and T, respectively, and let k be the number of strings in P. We present a new algorithm achieving time $O((n+m)\log k+\alpha)$ and space $O(m+A)$, where A is the sum of the lower bounds of the lengths of the gaps in P and α is the total number of occurrences of the strings in P within T. Compared to the previous results this bound essentially achieves the best known time and space complexities simultaneously. Consequently, our algorithm obtains the best known bounds for almost all combinations of m, n, k, A, and α. Our algorithm is surprisingly simple and straightforward to implement.

1 Introduction

Given integers a and b, $0 \leq a \leq b$, a *variable length gap* $g\{a, b\}$ is an arbitrary string over Σ of length between a and b, both inclusive. A *variable length gap pattern* (abbreviated VLG pattern) P is the concatenation of a sequence of strings and variable length gaps, that is, P is of the form

$$P = P_1 \cdot g\{a_1, b_1\} \cdot P_2 \cdot g\{a_2, b_2\} \cdots g\{a_{k-1}, b_{k-1}\} \cdot P_k \ .$$

A VLG pattern P *matches* a substring S of T iff $S = P_1 \cdot G_1 \cdots G_{k-1} \cdot P_k$, where G_i is any string of length between a_i and b_i, $i = 1, \ldots, k-1$. Given a string T and a VLG pattern P, the *variable length gap problem* (VLG problem) is to find all ending positions of substrings in T that match P.

Example 1. As an example, consider the problem instance over the alphabet $\Sigma = \{A, G, C, T\}$:

$$T = \text{ATCGGCTCCAGACCAGTACCCGTTCCGTGGT}$$
$$P = \text{A} \cdot g\{6, 7\} \cdot \text{CC} \cdot g\{2, 6\} \cdot \text{GT}$$

The solution to the problem instance is the set of positions $\{17, 28, 31\}$. For example the solution contains 17, since the substring ATCGGCTCCAGACCAGT, ending at position 17 in T, matches P.

Variable length gaps are frequently used in computational biology applications [15,13,16,7,8]. For instance, the PROSITE data base [5,9] supports searching for proteins specified by VLG patterns.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 385–394, 2010.

1.1 Previous Work

We briefly review the main worst-case bounds for the VLG problem. As above, let $P = P_1 \cdot g\{a_1, b_1\} \cdot P_2 \cdot g\{a_2, b_2\} \cdots g\{a_{k-1}, b_{k-1}\} \cdot P_k$ be a VLG pattern consisting of k strings, and let T be a string. To state the bounds, let $m = \sum_{i=1}^{k} |P_i|$ be the sum of the lengths of the strings in P and let n be the length of T.

The simplest approach to solve the VLG problem is to translate P into a regular expression and then use an algorithm for regular expression matching. Unfortunately, the translation produces a regular expression significantly longer than P, resulting in an inefficient algorithm. Specifically, suppose that the alphabet Σ contains σ characters, that is, $\Sigma = \{c_1, \ldots, c_\sigma\}$. Using standard regular expression operators (union and concatenation), we can translate $g\{a, b\}$ into the expression

$$g\{a, b\} = \overbrace{C \cdots C}^{a} \overbrace{(C|\epsilon) \cdots (C|\epsilon)}^{b-a},$$

where C is shorthand for the expression $(c_1 \mid c_2 \mid \ldots c_\sigma)$. Hence, a variable length gap $g\{a, b\}$, represented by a constant length expression in P, is translated into a regular expression of length $\Omega(\sigma b)$. Consequently, a regular expression R corresponding to P has length $\Omega(B\sigma + m)$, where $B = \sum_{i=1}^{k-1} b_i$ is the sum of the upper bounds of the gaps in P. Using Thompson's textbook regular expression matching algorithm [19] this leads to an algorithm for the VLG problem using $O(n(B\sigma + m))$ time. Even with the fastest known algorithms for regular expression matching this bound can only be improved by at most a polylogarithmic factor [14,17,2,3].

Several algorithms that improve upon the direct translation to a regular expression matching problem have been proposed [15,13,6,16,11,12,18,7,8,4]. Some of these are able to solve more general versions of the problem, such as searching for patterns that also contain character classes and variable length gaps with negative length. Most of the algorithms are based on fast simulations of non-deterministic finite automata. In particular, Navarro and Raffinot [16] gave an algorithm using $O(n(\frac{m+B}{w} + 1))$ time, where w is the number of bits in a memory word. Fredrikson and Grabowski [7,8] improved this bound for the case when all variable length gaps have lower bound 0 and identical upper bound b. Their fastest algorithm achieves $O(n(\frac{m \log\log b}{w} + 1))$ time. Very recently, Bille and Thorup [4] gave an algorithm using $O(n(k\frac{\log w}{w} + \log k) + m \log m + A)$ time and $O(m + A)$ space, where $A = \sum_{i=1}^{k-1} a_i$ is the sum of the lower bounds on the lengths of the gaps. Note that if we assume that the nk term dominates and ignore the $w/\log w$ factor, the time bound reduces to $O(nk)$.

An alternative approach, suggested independently by Morgante et al. [12] and Rahman et al. [18], is to design algorithms that are efficient in terms of the total number of occurrences of the k strings $P_1, \ldots, P_k$ within T. Let α be this number, e.g., in Example 1 A, CC, and GT occur 5, 5, and 4 times in T. Hence, $\alpha = 5 + 5 + 4 = 14$. Rahman et al. [18] gave an algorithm using

$O((n+m)\log k + \alpha \log(\max_{1\leq i<k}(b_i - a_i)))$ time[1]. Morgante et al. [12] gave a faster algorithm using $O((n+m)\log k + \alpha)$ time. Each of the k strings in P can occur at most n times and therefore $\alpha \leq nk$. Hence, in the typical case when the strings occur less frequently, i.e, $\alpha = o(n(k\frac{\log w}{w} + \log k))$, these approaches are faster. However, unlike the automata based algorithm that only use $O(m + A)$ space, both of these algorithm use $\Theta(m + \alpha)$ space. Since α typically increases with the length of T, the space usage of these algorithms is likely to quickly become a bottleneck for processing large biological data bases.

1.2 Our Results

We address the basic question of whether is it possible to design an algorithm that simultaneously is fast in the total number of occurrences of the k strings and uses little space. We show the following result.

Theorem 1. *Given a string T and a VLG pattern P with k strings, we can solve the variable length gaps matching problem in time $O((n+m)\log k+\alpha)$ and space $O(m + A)$. Here, α is the number of occurrences of the strings of P in T and A is the sum of the lower bounds of the gaps.*

Hence, we match the best known time bounds in terms of α and the space for the fastest automata based approach. Consequently, whenever $\alpha = o(n(k\frac{\log w}{w} + \log k))$ the time and space bounds of Theorem 1 are the best known. Our algorithm uses a standard comparison based version of the Aho-Corasick automaton for multi-string matching [1]. If the size of the alphabet is constant or we use hashing the $\log k$ factor in the running time disappears. Furthermore, our algorithm is surprisingly simple and straightforward to implement.

In some cases, we may also be interested in outputting not only the ending positions of matches of P, but also all of the possible combinations of strings in P that imply an occurrence in T. For instance, after we have identified a particularly interesting section in T using Theorem 1. Note that there can be exponentially many of these. Morgante et al. [12] showed how to encode all of these in a graph of size $O(\alpha)$. We can similarly extend our algorithm to produce such an encoding at the cost of using $O(\alpha)$ additional space.

1.3 Technical Overview

The previous work by Morgante et al. [12] and Rahman et al. [18] find all of the α occurrences of the strings $P_1, \ldots, P_k$ of P in T using a standard multi-string matching algorithm (see Section 2.1). From these, they construct a graph of size $\Omega(\alpha)$ to represent possible combinations of string occurrences that can be combined to form occurrences of P.

Our algorithm similarly finds all of the occurrences of the strings of P in T. However, we show how to avoid constructing a large graph representing the

[1] The bound stated in the paper does not include the $\log k$ factor, since they assume that the size of the alphabet is constant. We make no assumption on the alphabet size and therefore include it here.

possible combinations of occurrences. Instead we present a way to efficiently represent sufficient information to correctly find the occurrences of P, leading to a significant space improvement from $O(m + \alpha)$ to $O(m + A)$. Surprisingly, the algorithm needed to achieve this space bound is very simple, and only requires maintaining a set of sorted lists of disjoint intervals. Even though the algorithm is simple the space bound achieved by it is non-obvious. We give a careful analysis leading to the $O(m + A)$ space bound.

2 Algorithm

In this section we present the algorithm. For completeness, we first briefly review the classical Aho-Corasick algorithm for multiple string matching in Section 2.1. We then define the central idea of *relevant occurrences* in Section 2.2. We present the full algorithm in Section 2.3 and analyze it in Section 3.

2.1 Multi-string Matching

Given a set of pattern strings $\mathcal{P} = \{P_1, \ldots, P_k\}$ of total length m and a text T of length n the *multi-string matching problem* is to report all occurrences of each pattern string in T. Aho and Corasick [1] generalized the classical Knuth-Morris-Pratt algorithm [10] for single string matching to multiple strings. The *Aho-Corasick automaton* (AC-automaton) for $\mathcal{P}$, denoted $\text{AC}(\mathcal{P})$, consists of the trie of the patterns in $\mathcal{P}$. Hence, any path from the root of the trie to a state s corresponds to a prefix of a pattern in $\mathcal{P}$. We denote this prefix by $\text{path}(s)$. For each state s there is also a special *failure transition* pointing to the unique state s' such that $\text{path}(s')$ is the longest prefix of a pattern in $\mathcal{P}$ matching a proper suffix of $\text{path}(s)$. Note that the depth of s' in the trie is always strictly smaller for non-root states than the depth of s.

Finally, for each state s we store the subset $\text{occ}(s) \subseteq \mathcal{P}$ of patterns that match a suffix of $\text{path}(s)$. Since the patterns in $\text{occ}(s)$ share suffixes we can represent $\text{occ}(s)$ compactly by storing for s the index of the longest string in $\text{occ}(s)$ and a pointer to the state s' such that $\text{path}(s')$ is the second longest string if any. In this way we can report $\text{occ}(s)$ in $O(|\text{occ}(s)|)$ time.

The maximum outdegree of any state is bounded by the number of leaves in the trie which is at most k. Hence, using a standard comparison-based balanced search tree to index the trie transitions out of each state we can construct $\text{AC}(\mathcal{P})$ in $O(m \log k)$ time and $O(m)$ space.

To find the occurrences of $\mathcal{P}$ in T, we read the characters of T from left-to-right while traversing $\text{AC}(\mathcal{P})$ to maintain the longest prefix of the strings in $\mathcal{P}$ matching T. At a state s and character c we proceed as follows. If c matches the label of a trie transition t from s, the next state is the child endpoint of t. Otherwise, we recursively follow failure transitions from s until we find a state s' with a trie transition t' labeled c. The next state is then the child endpoint of t'. If no such state exists, the next state is the root of the trie. For each failure transition traversed in the algorithm we must traverse at least as many trie

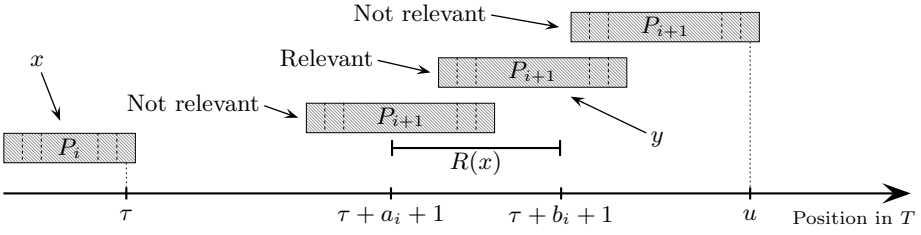


Fig. 1. In this figure x is an occurrence of P_i in T reported at position τ. The first and last occurrence of P_{i+1} start outside $R(x)$ thereby violating the ith gap constraint, so these occurrences are not relevant compared to x. The second occurrence y of P_{i+1} starts in $R(x)$, so if x is itself relevant, then y is also relevant.

transitions. Therefore, the total time to traverse AC($\mathcal{P}$) and report occurrences is $O(n \log k + \alpha)$, where α is the total number of occurrences.

Hence, the Aho-Corasick algorithm solves multi-string matching in $O((n + m) \log k + \alpha)$ time and $O(m)$ space.

2.2 Relevant Occurrences

For a substring x of T, let *startpos*(x) and *endpos*(x) denote the start and end position of x in T, respectively. Let x be an occurrence of P_i with $\tau =$ *endpos*(x) in T, and let $R(x)$ denote the range $[\tau + a_i + 1; \tau + b_i + 1]$ in T. An occurrence y of P_i in T is a *relevant occurrence* of P_i iff $i = 1$ or *startpos*$(y) \in R(x)$, for some relevant occurrence x of P_{i-1}. See Fig. 1 for an example. Relevant occurrences are similar to the *valid occurrences* defined in [18]. The difference is that a valid occurrence is an occurrence of P_{i+1} that is in $R(x)$ for *any* occurrence x of P_i in T, i.e., x need not be a valid occurrence itself.

From the definition of relevant occurrences, it follows directly that we can solve the VLG problem by finding the relevant occurrences of P_k in T. Specifically, we have the following result.

Lemma 1. *Let S be a substring of T matching the VLG pattern $S_1 \cdot g\{a_1, b_1\} \cdot S_2 \cdot g\{a_2, b_2\} \cdots S_k$. Then, startpos$(S_{i+1}) \in R(S_i)$ for all $i = 1, \ldots, k-1$.*

2.3 The Algorithm

Algorithm 1 computes the relevant occurrences of P_k using the output from the AC automaton. The idea behind the algorithm is to keep track of the ranges defined by the relevant occurrences of each subpattern P_i, such that we efficiently can check if an occurrence of P_i is relevant or not. More precisely, for each subpattern P_i, $i = 2, \ldots, k$, we maintain a sorted list L_i containing the ranges defined by previously reported relevant occurrences of P_{i-1}. When an occurrence of P_i is reported by the AC automaton, we can determine whether it is relevant by checking if it starts in a range contained in L_i (step 2b). Initially, the lists

Algorithm 1. Algorithm solving the VLG problem for a VLG pattern P and a string T

1. Build the AC-automaton for the subpatterns $P_1, P_2, \ldots, P_k$.
2. Process T using the automaton and each time an occurrence x of P_i is reported at position $\tau = \mathit{endpos}(x)$ in T do:
   (a) Remove any dead ranges from the lists L_i and L_{i+1}.
   (b) If $i = 1$ or $\tau - |P_i| = \mathit{startpos}(x)$ is contained in the first range in L_i do:
      i. If $i < k$: Append the range $R(x) = [\tau + a_i + 1; \tau + b_i + 1]$ to the end of L_{i+1}. If the range overlaps or adjoins the last range in L_{i+1}, the two ranges are merged into a single range.
      ii. If $i = k$: Report τ.

$L_2, L_3, \ldots, L_k$ are empty. When a relevant occurrence of P_i is reported, we add the range defined by this new occurrence to the end of L_{i+1}. In case the new range $[s, t]$ overlaps or adjoins the last range $[q, r]$ in L_{i+1} $(s \leq r + 1)$ we merge the two ranges into a single range $[q, t]$.

Let τ denote the current position in T. A range $[a, b] \in L_i$ is *dead at position* τ iff $b < \tau - |P_i|$. When a range is dead no future occurrences y of P_i can start in that range since $\mathit{endpos}(y) \geq \tau$ implies $\mathit{startpos}(y) \geq \tau - |P_i|$. In Fig. 1 the range $R(x)$ defined by x dies, when position u is reached. Our algorithm repeatedly removes any dead ranges to limit the size of the lists $L_2, L_3, \ldots, L_k$. To remove the dead ranges in step 2a we traverse the list and delete all dead ranges until we meet a range that is not dead. Since the lists are sorted, all remaining ranges in the list are still alive. See Fig. 2 for an example.

3 Analysis

We now show that Algorithm 1 solves the VLG problem in time $O((n+m) \log k + \alpha)$ and space $O(m + A)$, implying Theorem 1.

3.1 Correctness

To show that Algorithm 1 finds exactly the relevant occurrences of P_k, we show by induction on i that the algorithm in step 2b correctly determines the relevancy of all occurrences of P_i, $i = 1, 2, \ldots, k$, in T.

Base case: All occurrences of P_1 are by definition relevant and Algorithm 1 correctly determines this in step 2b.

Inductive step: Let y be an occurrence of P_i, $i > 1$, that is reported at position τ. There are two cases to consider.

1. y is relevant. By definition there is a relevant occurrence x of P_{i-1} in T, such that $\mathit{startpos}(y) = \tau - |P_i| \in R(x)$. By the induction hypothesis x was correctly determined to be relevant by the algorithm. Since $\mathit{endpos}(x) < \tau$, $R(x)$ was appended to L_i earlier in the execution of the algorithm. It remains to show that the range containing $\mathit{startpos}(y)$ is

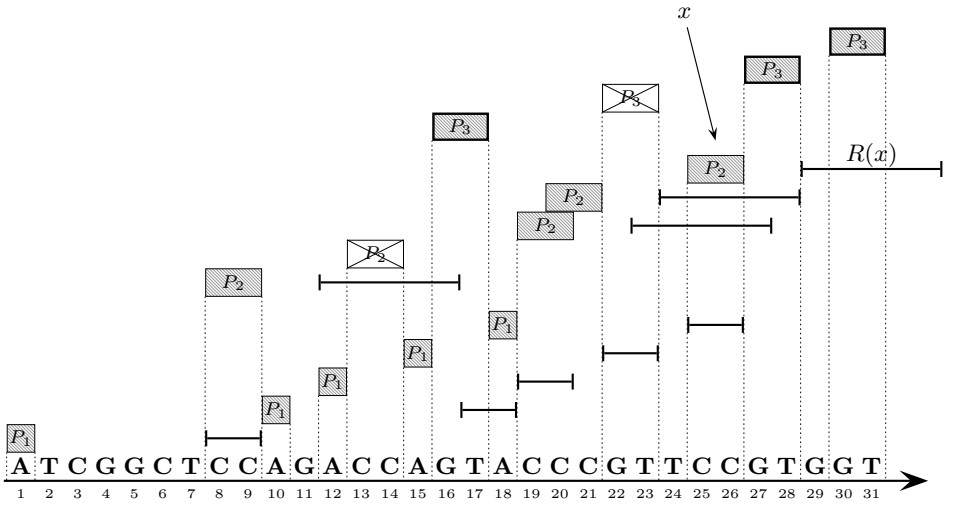


Fig. 2. The occurrences of the subpatterns $P_1 = \text{A}$, $P_2 = \text{CC}$ and $P_3 = \text{GT}$ and the ranges they define in the text T from Example 1. Occurrences which are not relevant are crossed out. The bold occurrences of P_3 are the relevant occurrences of P_k and their end positions 17,28 and 31 constitute the solution to the VLG problem. Consider the point in the execution of the algorithm when the occurrence x of P_2 at position $\tau = 26$ is reported by the Aho-Corasick automaton. At this time $L_2 = [\,[17;20], [22;23], [25;26]\,]$ and $L_3 = [\,[23,28]\,]$. The ranges $[17;20]$ and $[22;23]$ are now dead and are removed from L_2 in step 2a. In step 2b the algorithm determines that x is relevant and $R(x) = [29;33]$ is appended to L_3: $L_3 = [\,[23;33]\,]$.

the first range in L_i in step 2b. When removing the dead ranges in L_i in step 2a, all ranges $[a,b]$ where $b < \tau - |P_i|$ are removed. Therefore the range containing $\tau - |P_i| = \mathit{startpos}(y)$ is the first range in L_i after step 2a. It follows that the algorithm correctly determines that y is relevant.

2. y is not relevant. Then there exists no relevant occurrence x of P_{i-1} such that $\mathit{startpos}(y) \in R(x)$. By the induction hypothesis there is no range in L_i containing $\mathit{startpos}(y)$, since the algorithm only append ranges when a relevant occurrence is found. Consequently, the algorithm correctly determines that y is not relevant.

3.2 Time and Space Complexity

The AC automaton for the subpatterns $P_1, P_2, \ldots, P_k$ can be built in time $O(m \log k)$ using $O(m)$ space, where $m = \sum_{i=1}^{k} |P_i|$. For each of the α occurrences of the strings $P_1, P_2, \ldots, P_k$ Algorithm 1 first removes the dead ranges from L_i and L_{i+1} and performs a number of constant-time operations. Since both lists are sorted, the dead ranges can be removed by traversing the lists

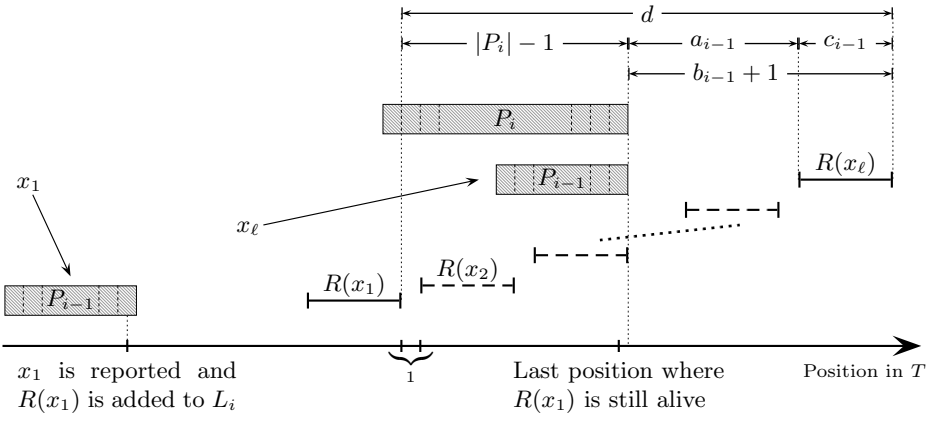


Fig. 3. The worst-case situation where ℓ, the maximum number of ranges are present in L_i. The figure only shows the first and the last occurrence of P_{i-1} (x_1 and x_ℓ) defining the ℓ ranges.

from the beginning. At most α ranges are ever added to the lists, and therefore the algorithm spends $O(\alpha)$ time in total on removing dead ranges. Since the AC automata runs in time $O((n+m)\log k + \alpha)$, the total running time is $O((n+m)\log k + \alpha)$.

To prove the space bound, we first show the following lemma.

Lemma 2. *At any time during the execution of the algorithm we have*

$$|L_i| \leq \left\lfloor \frac{2c_{i-1} + |P_i| + a_{i-1}}{c_{i-1} + 1} \right\rfloor = O\left(\frac{|P_i| + a_{i-1}}{b_{i-1} - a_{i-1} + 2} \right),$$

for $i = 2, 3, \ldots, k$, *where* $c_i = b_i - a_i + 1$.

Proof. Consider list L_i for some $i = 2, \ldots, k$. Referring to Algorithm 1, the size of the list L_i is only increased in step 2(b)i, when a range $R(x_j)$ defined by a relevant occurrence x_j of P_{i-1} is reported and $R(x_j)$ *does not* adjoin or overlap the last range in L_i.

Let $R(x_1) = [s, t]$ be the first range in L_i at an arbitrary time in the execution of the algorithm. We bound the number of additional ranges that can be added to L_i from the time $R(x_1)$ became the first range in L_i until $R(x_1)$ is removed. The last position where $R(x_1)$ is still alive is $\tau_a = t + |P_i| - 1$. If a relevant occurrence x_ℓ of P_{i-1} ends at this position, then the range $R(x_\ell) = [\tau_a + a_{i-1} + 1; \tau_a + b_{i-1} + 1]$ is appended to L_i. Hence, the maximum number of positions d from t to the end of $R(x_\ell)$ is

$$\begin{aligned} d &= \tau_a + b_{i-1} + 1 - t \\ &= (t + |P_i| - 1) + b_{i-1} + 1 - t \\ &= |P_i| + b_{i-1} \\ &= |P_i| + a_{i-1} + c_{i-1} - 1 \ . \end{aligned}$$

In the worst case, all the ranges in L_i are separated by exactly one position as illustrated in Fig. 3. Therefore at most $\lfloor d/(c_{i-1}+1) \rfloor$ additional ranges can be added to L_i before $R(x_1)$ is removed. Counting in $R(x_1)$ yields the following bound on the size of L_i

$$|L_i| \leq \left\lfloor \frac{d}{c_{i-1}+1} \right\rfloor + 1 = \left\lfloor \frac{2c_{i-1} + |P_i| + a_{i-1}}{c_{i-1}+1} \right\rfloor = O\left(\frac{|P_i| + a_{i-1}}{b_{i-1} - a_{i-1} + 2} \right) .$$

□

By Lemma 2 the total number of ranges stored at any time during the processing of T is at most

$$O\left(\sum_{i=2}^{k} \frac{|P_i| + a_{i-1}}{b_{i-1} - a_{i-1} + 2} \right) = O\left(\sum_{i=1}^{k-1} \frac{|P_{i+1}|}{b_i - a_i + 2} + \sum_{i=1}^{k-1} \frac{a_i}{b_i - a_i + 2} \right) = O\left(m + A\right).$$

Each range can be stored using $O(1)$ space, so this is an upper bound on the space needed to store the lists $L_2, \ldots, L_k$. The AC-automaton uses $O(m)$ space, so the total space required by our algorithm is $O(m + A)$.

In summary, the algorithm uses $O((n+m)\log k + \alpha)$ time and $O(m+A)$ space. This completes the proof of Theorem 1.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Bille, P.: New algorithms for regular expression matching. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 643–654. Springer, Heidelberg (2006)
3. Bille, P., Thorup, M.: Faster regular expression matching. In: Proc. 36th ICALP, pp. 171–182 (2009)
4. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proc. 21st SODA (2010)
5. Bucher, P., Bairoch, A.: A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In: Proc. 2nd ISMB, pp. 53–61 (1994)
6. Crochemore, M., Iliopoulos, C., Makris, C., Rytter, W., Tsakalidis, A., Tsichlas, K.: Approximate string matching with gaps. Nordic J. of Computing 9(1), 54–65 (2002)
7. Fredriksson, K., Grabowski, S.: Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. Inf. Retr. 11(4), 335–357 (2008)
8. Fredriksson, K., Grabowski, S.: Nested counters in bit-parallel string matching. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 338–349. Springer, Heidelberg (2009)
9. Hofmann, K., Bucher, P., Falquet, L., Bairoch, A.: The prosite database, its status in. Nucleic Acids Res. (27), 215–219 (1999)
10. Knuth, D.E., James, J., Morris, H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)

11. Lee, I., Apostolico, A., Iliopoulos, C.S., Park, K.: Finding approximate occurrences of a pattern that contains gaps. In: Proc. 14th AWOCA, pp. 89–100 (2003)
12. Morgante, M., Policriti, A., Vitacolonna, N., Zuccolo, A.: Structured motifs search. J. Comput. Bio. 12(8), 1065–1082 (2005)
13. Myers, E.W.: Approximate matching of network expressions with spacers. J. Comput. Bio. 3(1), 33–51 (1992)
14. Myers, E.W.: A four-russian algorithm for regular expression pattern matching. J. ACM 39(2), 430–448 (1992)
15. Myers, G., Mehldau, G.: A system for pattern matching applications on biosequences. CABIOS 9(3), 299–314 (1993)
16. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. J. Comput. Bio. 10(6), 903–923 (2003)
17. Navarro, G., Raffinot, M.: New techniques for regular expression searching. Algorithmica 41(2), 89–116 (2004)
18. Rahman, M.S., Iliopoulos, C.S., Lee, I., Mohamed, M., Smyth, W.F.: Finding patterns with variable length gaps or don't cares. In: Chen, D.Z., Lee, D.T. (eds.) COCOON 2006. LNCS, vol. 4112, pp. 146–155. Springer, Heidelberg (2006)
19. Thompson, K.: Regular expression search algorithm. Commun. ACM 11, 419–422 (1968)

Approximate String Matching with Stuck Address Bits

Amihood Amir[1,4], Estrella Eisenberg[1], Orgad Keller[1],
Avivit Levy[2,3], and Ely Porat[1]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
{amir,kellero,porately}@cs.biu.ac.il
[2] Department of Software Engineering, Shenkar College,
12 Anna Frank, Ramat-Gan, Israel
avivitlevy@shenkar.ac.il
[3] CRI, University of Haifa, Mount Carmel, Haifa 31905, Israel
[4] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218

Abstract. A string $S \in \Sigma^m$ can be viewed as a set of pairs $\{(s_i, i) \mid s_i \in S,\ i \in \{0, \ldots, m-1\}\}$. We follow the recent work on *pattern matching with address errors* and consider approximate pattern matching problems arising from the setting where errors are introduced to the location component (i), rather than the more traditional setting, where errors are introduced to the content itself (s_i). Specifically, we continue the work on string matching in the presence of address bit errors. In this paper, we consider the case where bits of i may be stuck, either in a consistent or transient manner. We formally define the corresponding approximate pattern matching problems, and provide efficient algorithms for their resolution.

1 Introduction

Background. Over 30 years ago, one of the co-authors of this paper was busy writing a program that points an antenna to a given moving location. Having written a program that converts latitude and longitude to the appropriate azimuth, taking all geodesic information into consideration, the program was finally tested.

The frustrated programmer noticed that the antenna was pointing to the west, when it was supposed to point north. In those days, de-bugging meant halting the computer and looking at the memory contents through a panel register. The programmer halted the program after it loaded the bus with the azimuth and immediately prior to giving the device the signal to load the azimuth, and checked the value. To his surprise, the value matched his calculations. He then resumed running the program and the antenna pointed exactly to the required direction. However, running the program from beginning to end again achieved a wrong result.

The problem was that some bits on the bus settled on their value faster than others, thus when those bits had a 0 value and the value was changed to 1, it

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 395–405, 2010.

took longer to settle than when a 1 was changed to a 0, or when the value was not changed. A short wait helped.

1.1 Pattern Matching with Address Errors

Motivation. An important implicit assumption in the traditional view of pattern matching was that there may indeed be errors in the *content* of the data, but the *order* of the data is inviolate. Consider a text $T = t_0 \dots t_{n-1}$ and pattern $P = p_0 \dots p_{m-1}$, both over an alphabet Σ. Traditional pattern matching regards T and P as *sequential* strings, provided and stored in sequence (e.g., from left to right). However, some non-conforming problems have been gnawing at the basis of this assumption. An example is the *swap* error, motivated by the common typing error where two adjacent symbols are exchanged [20,7,8,11], which does not assume error in the content of the data, but rather, in the order.

Computational biology has also added several problems wherein the "error" is in the order, rather than the content. During the course of evolution areas of genome may be shifted from one location to another. Considering the genome as a string over the alphabet of genes, these cases represent a situation where the difference between the original string and resulting one is in the locations rather than contents of the different elements. Several works have considered specific versions of this biological setting, primarily focusing on the sorting problem (*sorting by reversals* [13,14], *sorting by transpositions* [12], and *sorting by block interchanges* [15]).

The inherently distributed nature of the web is already causing (in Bit Torrent and Video on Demand) the phenomenon of transmission of a stream of data in tiny pieces from different sources. This creates the problem of putting scrambled data back together again.

Finally, in computer architecture, address errors are of no less concern than content errors [17]. It is by no means taken for granted that when seeking a word from a given address, no errors will occur in the address bits. This problem is relevant even when reading a buffer of consecutive words since these words are not necessarily consecutive in the disk or in an interleaved cache.

Therefore, the traditional view of strings is becoming, at times, too restrictive.

The Model. In such cases, it is more natural to view the string as a set of pairs (σ, i), where i denotes a location in the string, and σ is the value appearing at this location. Given this view of strings, the problem of *approximate pattern matching* has been reconsidered in the last few years, and a new pattern matching paradigm – *pattern matching with address errors* – was proposed in [2]. In this model, the pattern *content* remains intact, but the relative positions (addresses) may change. Efficient algorithms for several different natural types of rearrangement errors were presented [3,9,4,19] (see also [10]). These types of address errors were inspired by biology, i.e., pattern elements exchanging their locations due to some external process.

Address Bit Errors. Another broad class of address errors inspired by computer architecture was studied by [1,6]. They consider errors which arise from a process

of flipping some or all of the bits in the binary representation of $[1, m]$. Such errors represent situations where the text and the pattern are generated by two different systems, which may use different naming conventions. The error processes are inspired by address errors resulting from failures in the wires of the address bus, the wires connecting the CPU and the memory which are used to transmit the address of operands (see Figure 1), or failure in the transmitted address bits. The errors handled by [1,6] were all bi-directional, jogging the memory of our programmer. Discussions with old cronies who, over the years, continued grappling with parallel transmissions over wires, resulted in the desire to study the situation where the "badness" of the bits means being "stuck" on a value, rather than changing it.

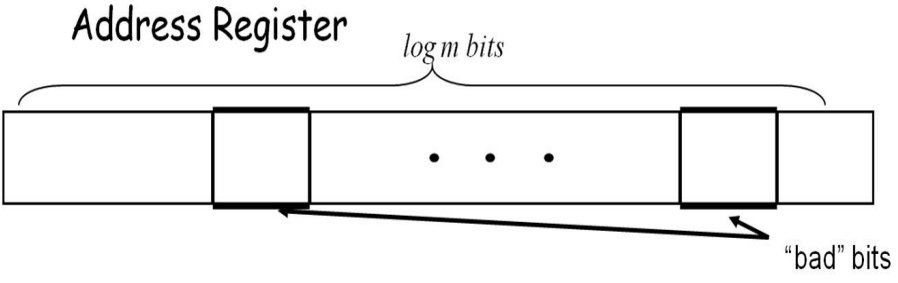


Fig. 1. Failures in the address bus due to 'bad' bits cause wrong addresses to be stored in the address register

Our Contribution. This paper follows the work of [6], but studies the situation of address bit errors caused in the presence of stuck bits (defined below), that was not considered by previous work. The contributions of this paper are two-fold:

1. to enhance the nascent body of work on pattern matching with rearrangements. In particular this paper requires a non-trivial use of network flow to solve one version of our problem. This is definitely not a technique in the traditional Pattern Matching tool kit.
2. to continue the study of pattern matching under address bit errors that was begun in [5]. This paper is still only a beginning. Discussions with practitioners suggest further directions, as will be presented in Sect. 4.

1.2 Problem Definition

Consider a string $S \in \Sigma^m$. Using the definition of [5], the string is regarded as a set of pairs, $S = \{(\sigma, i) \mid \sigma \in \Sigma,\ i \in \{0,1\}^{\log m}\}$. We consider two types of errors in the bits of the i entries:

Stuck bits. There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each i, all bits in positions $f \in F$ are either always changed to zero (i.e. 1 is turned into a 0 and 0 remains 0) or always changed to one (i.e. 0 is turned into a 1 and 1 remains 1).

For example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{1\}$, a resulting string is $S' = \{(1, 00), (2, 01), (3, 00), (4, 01)\}$.

Transient stuck bits. There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each i, the bits in positions $f \in F$ *may* remain unchanged, or may be changed to a "1" (of course the original string changes only if the intention was to output a "0").

As an example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{1\}$, the resulting string may be $S' = \{(1, 10), (2, 01), (3, 10), (4, 11)\}$ (the bit was changed to one for address 1 but not for address 2).

Note that the resulting set is actually a multi-set, and may not represent a valid string, as some locations may appear multiple times, while others not at all.

We consider approximate pattern matching problems associated with each of the above types of errors. Specifically, given a pattern P and text T, we wish to find:

- the smallest set F such that if the bits of F are consistently stuck, then P has a match in T. We call this problem the *stuck bits* problem.
- the smallest set F such that if the bits of F may be transiently stuck, then P has a match in T. We call this problem the *transient stuck bits* problem.

Following [5], we focus on developing efficient solutions for the case that the text and the pattern are both of length m. We discuss the situation of text longer than pattern in Sect. 4.

1.3 Our Results

We provide the following results:

- an $O(m \log m)$ time solution for pattern matching with stuck bits, which also reports the stuck bits positions, where m is the length of both text and pattern.
  (Theorem 1)

- a simple $O(m^{2.5})$ time solution for pattern matching with transient bits, which also reports the stuck bits positions. This algorithm is based on a reduction to finding perfect matching in a bipartite graph.
  (Corollary 2)

- a flow-based $O(m^{2.2156} \log^2 m)$ time solution for pattern matching with transient bits, which also reports the stuck bits positions.
  (Theorem 2)

Paper Organization. The rest of the paper is organized as follows. In Sect. 2 we study the stuck bits problem and prove Theorem 1. In Sect. 3 we study the transient stuck bits problem and prove Corollary 2 and Theorem 2.

2 The Stuck Bits Problem

The nature of the stuck bits problem, as opposed to the flipped bits problem of [5], is that a stuck bit necessarily *deletes* addresses and creates addresses with multiple symbols. We show below that it is possible to not only compute the number of stuck bits by a considering the address sets, but also to easily compute the stuck bits' positions.

Let T be a length-m text and $P = \{(\sigma, i) \mid \sigma \in \Sigma, \quad i \in \{0,1\}^{\log m}\}$ be a length-m pattern. Define $IP = \{i \mid \exists\sigma, \ (\sigma, i) \in P\}$ to be the set of character positions given in P.

Observation 1. *Assume m is a power of 2. Let n be the number of stuck bits and $\ell = \log m - n$. Then $|IP| = 2^{\ell}$.*

Algorithm StuckBits(P) below constructs the set IP from input P, and outputs a binary string k of length $\log m$, where $k[i] = 0$ if i is a stuck bit, and $k[i] = 1$ otherwise. The algorithm uses the boolean operator $\oplus$ – the *exclusive or* operation. Specifically, $a \oplus b$ is 0 if $a = b$ and 1 if $a \neq b$, for $a, b \in \{0,1\}$. The definition below extends the boolean operation to strings in the natural manner.

Definition 1. *Let $s, t \in \{0,1\}^{\ell}$ i.e. $s = s[1], \ldots, s[\ell]$, $t = t[1], \ldots t[\ell]$ $s[i], t[i] \in \{0,1\}$, $i = 1, \ldots, \ell$, and let $\oplus$ be a boolean operator, exclusive or. Define $s \oplus t$ as:*

$$(s \oplus t)[i] = s[i] \oplus t[i]$$

for $i = 1, \ldots, \ell$.

Algorithm 1. StuckBits(P)

```
1 let k be a log m-length bit-vector;
2 let i_0 be the lexicographic minimum i such that i ∈ IP;
3 if ∄j ∈ IP,  j ≠ i_0 then return 1^{log m};
4 foreach j ∈ IP such that j ≠ i_0 do
5     k_j ← j ⊕ i_0;
6 end foreach
7 k ← ⋀_{j≠i_0} k_j;
8 return k;
```

Lemma 1. *Every location in k – the output vector of StuckBits(P) – that equals zero, is a stuck bit.*

Proof. A bit i is stuck iff every location i has the same value (zero or one) in all addresses (second component) of P iff the exclusive or of all addresses of P has a zero in location i. ∎

Example 1. Given the pattern $P = \{(1, 00), (2, 01), (3, 00), (4, 01)\}$, algorithm StuckBits calculates $(00 \oplus 01)$, and therefore returns 01. We conclude that in this case the most significant bit is a stuck bit.

2.1 Pattern Matching under Stuck Bits Errors

For two addresses $i, j \in \{0, \ldots, m\}$, we say i *is equivalent to* j *under possible stuck bits*, and write $i \equiv j$, if all bits where i differs from j are stuck bits. Formally, $i \equiv j$ iff $(i \oplus j) \wedge \text{StuckBits}(P) = 0^{\log m}$. The following algorithm decides if there exist a set of bits that, if stuck, cause text T to become pattern P. In this case we say that P *matches* T *under stuck bits errors*. The idea of the algorithm is to gather all text symbols in locations whose addresses are indistinguishable due to the stuck bits, and compare these sets to the sets provided by the pattern. The algorithm works efficiently since Lemma 1 allows us to identify the locations of the stuck bits.

Algorithm 2. StuckMatch(T, P)

```
1 foreach i ∈ IP do
2     B_i^P ← {σ | (σ, i) ∈ P};
3     sort B_i^P;
4     B_i^T ← {T[j] | j ≡ i};
5     sort B_i^T;
6 end foreach
7 B ← ⋀_{i∈IP}(B_i^P = B_i^T);
8 return B;
```

Example 2. Given the pattern $P = \{(1, 00), (2, 01), (3, 00), (4, 01)\}$, algorithm StuckBits$(P)$ returns 01. In this case, $T = \langle 1, 2, 3, 4\rangle$, $B_0^P = \langle 1, 3\rangle$, $B_1^P = \langle 2, 4\rangle$, $B_0^T = \langle 1, 3\rangle$, and $B_1^T = \langle 2, 4\rangle$. Therefore, algorithm StuckMatch(T, P) returns 1.

2.2 Total Time for the Stuck Bits Problem

We obtain the following:

Theorem 1. StuckMatch(T, P) *can be solved in* $O(m \log m)$ *time, where* m *is the length of both text and pattern. For finite alphabets, or alphabet* $\{1, ..., m\}$, StuckMatch(T, P) *can be solved in linear time.*

Proof. Correctness follows from the above discussion. For the time complexity, we assume constant time operations on words of size $O(\log m)$ bits. Finding the stuck bits requires $\Theta(m)$ time. The B_i^P and B_i^T can be constructed in time $\Theta(m)$ as well. Sorting each of the B_i^P and B_i^T can be done in time $O(m \log m)$ in general, and by bucket sort for finite alphabets or alphabet $\{1, ..., m\}$. Finally, B is calculated in $\Theta(m)$ time. We conclude that the overall time is $O(m \log m)$, and linear in the case of finite alphabets, or alphabet $\{1, ..., m\}$. ∎

3 Transient Stuck Bits Problem

Similarly to the flipped bit problem of [5], the first step is comparing the histogram of characters in the text and pattern, i.e., for each alphabet symbol σ, the number of occurrences of σ in T and P needs to be equal, otherwise there can be no matching. Assume, therefore, that the histograms match.

3.1 Verifying the Existence of a Transient Stuck Bit Matching

Let $T \in \Sigma^m$ and $P = \{(\sigma, i) \mid \sigma \in \Sigma, \quad i \in \{0,1\}^{\log m}\}$ of length m.

The following is a simple solution, reducing the problem to perfect bipartite matching. For a given $\sigma \in \Sigma$, we are seeking a bijection from the pattern pairs (σ, i) to the text locations where there are σ's, in a manner that if (σ, i) is matched to j then every bit location b that has a 1 in j has a 1 in i.

Such a bijection can be constructed via maximum perfect matching in the following bipartite graph:

Definition 2. [The Bipartite Graph] *Let $G_\sigma = (V_1, V_2, E)$ where each of V_1 and V_2 has m elements. The elements of V_1 are labeled by the pairs $(T[j], j)$, $j = 1, ..., m$. We label V_2 by the bijection $\ell : V_2 \to P$.*

Put an edge between node $v_1 \in V_1$ and node $v_2 \in V_2$ if their labels have the same symbols and if the address of v_1's label can be translated to the address of v_2's label via transient stuck bit *errors. Formally, $E = \{(v_1, v_2) |$ where $v_1 = (\sigma_1, i)$, $v_2 = (\sigma_2, j)$, $\sigma_1 = \sigma_2$ and for every 1 in bit location b in i there is a 1 in bit location b in j $\}$.*

Example 3. Given $T = ABAB$ and $P = \{(A, 01), (A, 10), (B, 11), (B, 11)\}$. Figure 2 shows the bipartite graph constructed from the text and pattern as well as a maximal perfect matching.

Now all we need to do is verify if there is a perfect matching in G_σ. The Hopcroft and Karp algorithm [18] (denoted as *HopcroftKarp* in the pseudo-code below) finds the size of the maximum matching in time: $\Theta(E\sqrt{V_1 + V_2})$. In our case, since $V_1 = V_2 = m$ the time is $\Theta(E\sqrt{m})$, and in the worst case, $O(m^{2.5})$.

3.2 Finding the Stuck Bits Location

The remaining task is identifying the locations of the transient stuck bits.

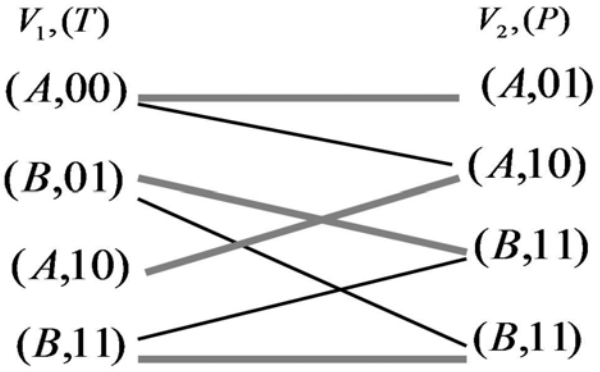


Fig. 2. The bipartite graph with a maximal matching

Algorithm 3. TransientBitsMatch(T, P)

```
1 Construct bipartite graph G_σ.
2 if |HopcroftKarp(G)| = m then return 1;
3 else return 0;
```

Denote the sum of 1's in bit location b of the addresses in the text (V_1) as T_b, and the sum of 1's in bit location b of the addresses in the pattern (V_2) as P_b.

Consider node $v_1 \in V_1$ and assume there is a 1 in bit location b of its address. The node $v_2 \in V_2$ that was matched to v_1 by the perfect matching must have a 1 in bit location b of its address. Therefore, if the sum of 1's in bit location b of all nodes in V_1 (T_b) equals the sum of 1's in bit location b of all nodes in V_2 (P_b) then bit b can not be a stuck bit.

On the other hand if $P_b > T_b$ then it means that b is sometimes stuck. Moreover, the number of times it is stuck is $P_b - T_b$.

It can never be the case that $T_b > P_b$ since then there is no possible perfect matching of size m.

Corollary 1. *The stuck bits are all the bit locations b where $P_b > T_b$ and the number of times it is stuck is $P_b - T_b$.*

Example 4. In example 3, $T_0 = 2$ and $P_0 = 3$ which means that bit 0 was stuck once. $T_1 = 2$ and $P_1 = 3$ which means that bit 1 was also stuck once.

3.3 Faster Verification of Transient Stuck Bit Matching

Assuming a transient stuck bit matching exists, Subsection 3.2 finds the location in time $O(m \log m)$. Thus the time complexity bottleneck is the transient stuck bits matching verification. In this subsection, we show a faster verification algorithm, based on network flow.

Let V_1 be as in Definition 2 and V_3 be the set of all *distinct* pairs $(\sigma, i) \in P$. Construct the following flow network.

Definition 3. [The Flow Network] *Let $G_{T,P} = (V, E)$ where $V = V_1 \cup V_3 \cup \{s\} \cup \{f\}$. s is the* source *and f is the* sink.

E is constructed as follows. $\forall v \in V_1$ there is an edge $\overrightarrow{sv}$ (for every node v in V_1 there is an edge from the source to v). $\forall w \in V_3$ $\exists$edge $\overrightarrow{wf}$. For every $v = (\sigma_1, i) \in V_1$ and $w = (\sigma_2, j) \in V_3$, if $\sigma_1 = \sigma_2$ and for every 1 in bit location b in i there is a 1 in bit location b in j then there is an edge $\overrightarrow{vw}$. (This last condition is the same as that of Definition 2.)

We now define the edge capacities. Let $v = (\sigma, i) \in V_3$ and assume that v occurs c times in P. Then the capacity of edge $\overrightarrow{vf}$ is c. The capacity of every edge from the source to V_1 or from V_1 to V_3 is 1.

Lemma 2. *$G_{T,P}$ has a flow of value m iff there is a transient stuck bits matching between T and P.*

Proof. It is easy to see that a transient stuck bits matching defines a flow. Conversely, if there is a flow whose value is m, assign the nodes as defined by the flow. ∎

Max flow can be determined, using the Goldberg-Rao binary blocking flow algorithm [16], in time $O(|E| \min(|V|^{2/3}, \sqrt{|E|}) \log(|V|^2/|E|) \log U)$, where U is the network capacity. In our case we have $|V| = U = \Theta(m)$.

Corollary 2. *The transient stuck bits matching problem can be solved in time $O(|E| \min(m^{2/3}, \sqrt{|E|}) \log^2 m)$.*

We need to determine the value of $|E|$.

Lemma 3. *$G_{T,P}$ has at most $O(3^{\log m}) = O(m^{\log_2 3}) \approx O(m^{1.5489})$ edges.*

Proof. There are always exactly m nodes from s to V_1, and between 1 and m nodes from V_3 to f. We consider the case where $|V_1| = |V_3|$, because that is the case with the most number of edges from V_1 to V_3. Every other case has a subset of nodes, thus a subset of edges from V_1 to V_3.

If $V_1 = V_3$ then for every node in V_1, if it has t zeroes, it has 2^t outgoing edges, to all possible nodes in V_3 that a stuck bit can send it. Therefore, the total number of outgoing edges from V_1 is:

$|E| = \sum_{t=0}^{\log m} \binom{\log m}{t} \cdot 2^t = 3^{\log m} \approx m^{1.5489}$ ∎

3.4 Total Time for the Transient Stuck Bits Problem

From the above discussion we obtain the following:

Theorem 2. TransientStuckBits(T, P) *runs in* $\Theta(m^{\log_2 3+2/3} log^2 m) = O(m^{2.2157} \log^2 m)$ *time, where m is the length of both text and pattern.*

4 Conclusions and Open Problems

This paper follows up recent work on a new paradigm for approximate pattern-matching that, instead of content errors, considers location errors or rearrangement errors. Specifically, the problems of finding a match under stuck bits and transient stuck bits were studied and efficient solutions for these problems are provided. Most importantly, apart from the specific algorithmic results, this paper gives another evidence of the richness of the research field that is opened with the new paradigm.

We have solved the stuck bit problem only for the case where the text and pattern are of the same length. It can clearly be extended to a pattern matching setting where the text is of greater length and we would like to find the number of stuck bits for a every text location where there is a stuck bit matching. Our algorithm can, of course, be run for every text location separately. It would be interesting to know if a faster solution than $O(nm^{2.2156}\log^2 m)$ can be found.

This direction of research leads to more challenging questions. In reality, various types of address bit errors can occur. Some were considered in [5], some in this paper, and there are more. Different types of errors have different probabilities of occurrence. In some hardware configurations, even the "stuck" bits have a different probability of occurrence depending on where in the register they are located. It would be important to integrate various different errors into the Pattern Matching model and, in future work, consider the probabilities of the various errors as well.

References

1. Amir, A.: Asynchronous pattern matching. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 1–10. Springer, Heidelberg (2006) (invited talk)
2. Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: Rearrangement distances. In: Proc. 17th ACM-SIAM Symp. on Discrete Algorithms, SODA (2006)
3. Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: Rearrangement distances. Journal of Computer and System Sciences 75(6), 359–370 (2009)
4. Amir, A., Aumann, Y., Indyk, P., Levy, A., Porat, E.: Efficient computations of ℓ_1 and ℓ_∞ rearrangement distances. Theoretical Computer Science 410(43), 4382–4390 (2009)
5. Amir, A., Aumann, Y., Kapah, O., Levy, A., Porat, E.: Approximate string matching with address bit errors. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 118–129. Springer, Heidelberg (2008)
6. Amir, A., Aumann, Y., Kapah, O., Levy, A., Porat, E.: Approximate string matching with address bit errors. Theoretical Computer Science 410(51), 5334–5346 (2009); Special Issue of CPM 2008 Best Papers
7. Amir, A., Cole, R., Hariharan, R., Lewenstein, M., Porat, E.: Overlap matching. Information and Computation 181(1), 57–74 (2003)
8. Amir, A., Eisenberg, E., Porat, E.: Swap and mismatch edit distance. Algorithmica 45(1), 109–120 (2006)

9. Amir, A., Hartman, T., Kapah, O., Levy, A., Porat, E.: On the cost of interchange rearrangement in strings. SIAM Journal on Computing 39(4), 1444–1461 (2009)
10. Amir, A., Levy, A.: String rearrangement metrics: A survey. In: Algorithms and Applications, pp. 1–33 (2010)
11. Amir, A., Lewenstein, M., Porat, E.: Approximate swapped matching. Information Processing Letters 83(1), 33–39 (2002)
12. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM J. on Discrete Mathematics 11, 221–240 (1998)
13. Berman, P., Hannenhalli, S.: Fast sorting by reversal. In: Hirschberg, D.S., Myers, E.W. (eds.) CPM 1996. LNCS, vol. 1075, pp. 168–185. Springer, Heidelberg (1996)
14. Carpara, A.: Sorting by reversals is difficult. In: Proc. 1st Annual Intl. Conf. on Research in Computational Biology (RECOMB), pp. 75–83. ACM Press, New York (1997)
15. Christie, D.A.: Sorting by block-interchanges. Information Processing Letters 60, 165–169 (1996)
16. Goldberg, A., Rao, S.: Beyond the flow decomposition barrier. J. of the ACM 45(5), 783–797 (1998)
17. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann, San Francisco (2002)
18. Hopcroft, J., Karp, R.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Computing 2(4), 225–231 (1973)
19. Kapah, O., Landau, G.M., Levy, A., Oz, N.: Interchange rearrangement: The element-cost model. Theoretical Computer Science 410(43), 4315–4326 (2009)
20. Lowrance, R., Wagner, R.A.: An extension of the string-to-string correction problem. J. of the ACM, 177–183 (1975)

Erratum to: Range Queries over Untangled Chains

Francisco Claude, J. Ian Munro, and Patrick K. Nicholson

David R. Cheriton School of Computer Science, University of Waterloo, Canada
{fclaude,imunro,p3nichol}@cs.uwaterloo.ca

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 82–93, 2010.

DOI 10.1007/978-3-642-16321-0_42

In the original version of this paper the bound cited in the abstract is incorrect. It should read "The running time for a query is $O(\lg k \lg n + k' \lg n + m)$, where k is the number of non-crossing monotonic chains in which we can partition the set of points, k' leq k is a value dependent on the query, and m is the size of the output."

This bound also appears at the end of Section 2.

The original online version for this chapter can be found at
http://dx.doi.org/10.1007/978-3-642-16321-0_8

Author Index