Fabio Crestani
Paolo Ferragina
Mark Sanderson (Eds.)

String Processing and Information Retrieval

LNCS 4209

13th International Conference, SPIRE 2006
Glasgow, UK, October 2006
Proceedings

Springer

Lecture Notes in Computer Science 4209

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

Fabio Crestani Paolo Ferragina
Mark Sanderson (Eds.)

String Processing and Information Retrieval

13th International Conference, SPIRE 2006
Glasgow, UK, October 11-13, 2006
Proceedings

Springer

Volume Editors

Fabio Crestani
University of Strathclyde
Department of Computer and Information Sciences
16 Richmond Street, Glasgow G12 0NX, UK
E-mail: f.crestani@cis.strath.ac.uk

Paolo Ferragina
University of Pisa
Department of Computer Science
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
E-mail: ferragina@di.unipi.it

Mark Sanderson
University of Sheffield
Department of Information Studies
Regent Court, 211 Portobello St, Sheffield, S1 4DP, UK
E-mail: m.sanderson@shef.ac.uk

Library of Congress Control Number: 2006932965

CR Subject Classification (1998): H.3, H.2.8, I.2, E.1, E.5, F.2.2

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

ISSN 0302-9743
ISBN-10 3-540-45774-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-45774-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11880561 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 13th International Symposium on String Processing and Information Retrieval (SPIRE), held October 11-13, 2006, in Glasgow, Scotland.

The SPIRE annual symposium provides an opportunity for both new and established researchers to present original contributions to areas such as string processing (dictionary algorithms, text searching, pattern matching, text compression, text mining, natural language processing, and automata-based string processing); information retrieval languages, applications, and evaluation (IR modelling, indexing, ranking and filtering, interface design, visualization, cross-lingual IR systems, multimedia IR, digital libraries, collaborative retrieval, Web-related applications, XML, information retrieval from semi-structured data, text mining, and generation of structured data from text); and interaction of biology and computation (sequencing and applications in molecular biology, evolution and phylogenetics, recognition of genes and regulatory elements, and sequence-driven protein structure prediction).

The papers in this volume were selected from 102 papers submitted from over 20 different countries in response to the Call for Papers. A total of 26 submissions were accepted as full papers, yielding an acceptance rate of about 25%. In view of the large number of good-quality submissions the Program Committee decided to accept 5 short papers, that have also been included in the proceedings. SPIRE 2006 also featured two talks by invited speakers: Jamie Callan (Carnegie Mellon University, USA) and Martin Farach-Colton (Rutgers University, USA).

The Organizing Committee would like to thank all the authors who submitted their work for consideration and the participants of SPIRE 2006 for making the event a great success.

Special thanks are due to the members of the Program Committee who worked very hard to ensure the timely review of all the submitted manuscripts, and to the invited speakers, Jamie Callan and Martin Farach-Colton, for their inspiring presentations. We also would like to thank the sponsoring institutions, EPSRC (Engineering and Physical Sciences Research Council) , Yahoo! Research, the Kelvin Institute, the BCS-IRSG (British Computer Society - Information Retrieval Specialist Group), and the University of Strathclyde, for their generous financial and institutional support, and Glasgow City Council for civic hospitality.

Thanks are due to the editorial staff at Springer for their agreement to publish the colloquium proceedings as part of the *Lecture Notes in Computer Science* series.

Thanks are also due to the local team of student volunteers (in particular Mark Baillie, Murat Yakici and Emma Nicol), the secretaries (Carol-Ann Seath and Linda Hunter), and the information officer (Paul Smith), whose efforts ensured the smooth organization and running of the event.

Finally, we would like to thank Ricardo Baeza-Yates, who, on behalf of the Steering Committee, invited us to organize SPIRE 2006 and supported us at every step of the way.

October 2006

Fabio Crestani
Paolo Ferragina
Mark Sanderson

SPIRE 2006 Organization

Organizing Institution

SPIRE 2006 was organized by the Department of Computer and Information Sciences of the University of Strathclyde and held at the Teacher Building in Glasgow, Scotland, UK.

Sponsoring Institutions

Engineering and Physical Sciences Research Council, UK.
Yahoo! Research, Barcellona, Spain.
Kelvin Institute, Glasgow, Scotland, UK.
British Computer Society - Information Retrieval Specialist Group, UK.
Glasgow City Council, Glasgow, Scotland, UK.

Organizing Committee

General Chair: Fabio Crestani (University of Strathclyde, UK)

Program Committee Chairs: Paolo Ferragina (University of Pisa, Italy) and Mark Sanderson (University of Sheffield, UK)

Program Committee

Gianni Amati (Fondazione Ugo Bordoni, Italy)
Amihood Amir (University Bar-Ilan, Israel and Georgia Tech, USA)
Alberto Apostolico (Georgia Tech, USA and University of Padua, Italy)
Ricardo Baeza-Yates (Yahoo! Research, Spain and Chile)
Michael Bender (Stony Brook University, USA)
Mohand Boughanem (University of Tolouse, France)
Giorgio Brajnik (University of Udine, Italy)
Gerth S. Brodal (University of Aarhus, Denmark)
Paul Browne (Imperial College, UK)
Chris Buckley (Sabir Research, USA)
Mariano Consens (University of Toronto, Canada)
Nick Craswell (Microsoft Research, UK)
Maxime Crochemore (University of Marne-la-Vallée, France)
Bruce Croft (University of Massachusetts at Amherst, USA)
Erik Demaine (MIT, USA)
Martin Farach-Colton (Rutgers University, USA)
Edward Fox (Virginia Tech, USA)

Norbert Fuhr (University of Duisburg-Essen, Germany)
Eric Gaussier (Xerox-RCE, France)
Raffaele Giancarlo (University of Palermo, Italy)
Mark Girolami (University of Glasgow, UK)
Nazli Goharian (IIT, USA)
Enrique Herrera-Viedma (University of Granada, Spain)
Costas Iliopoulos (King's College London, UK)
Joemon Jose (University of Glasgow, UK)
Juha Kärkkäinen (University of Helsinki, Finland)
Jussi Karlgren (SICS, Sweden)
Mounia Lalmas (Queen Mary, University of London, UK)
Gadi Landau (University of Haifa, Israel and Polytechnic University, NY, USA)
Hans-Peter Lenhof (University of Saarbrücken, Germany)
Moshe Lewenstein (University Bar-Ilan, Israel)
Stefano Lonardi (University of California Riverside, USA)
David Losada (University of Santiago de Compostela, Spain)
Andrew MacFarlane (City University, London, UK)
Veli Mäkinen (University of Helsinki, Finland)
Giovanni Manzini (University of Piemonte Orientale, Italy)
Paul McNamee (JHU, USA)
Massimo Melucci (University of Padova, Italy)
Alistair Moffat (University of Melbourne, Australia)
Gonzalo Navarro (University of Chile, Chile)
Paul Ogilvie (CMU, USA)
Arlindo Oliveira (INESC-ID/Technical University of Lisbon, Portugal)
Pietro Pala (University of Firenze, Italy)
Gabriella Pasi (Università degli Studi di Milano Bicocca, Italy)
Mathieu Raffinot (CNRS, France)
Rajeev Raman (Leicester University, UK)
Andreas Rauber (Technical University of Vienna, Austria)
Crawford Revie (University of Strathclyde, UK)
Keith van Rijsbergen (University of Glasgow, UK)
Ian Ruthven (University of Strathclyde, UK)
Kunihiko Sadakane (Kyushu University, Japan)
Marie-France Sagot (INRIA Rhone-Alpes, France)
Falk Scholer (RMIT, Australia)
Steven Skiena (Stony Brook University, USA)
Ian Soboroff (NIST, USA)
Jens Stoye (University of Bielefeld, Germany)
Tassos Tombros (Queen Mary, University of London, UK)
Andrew Trotman (Otago, New Zealand)
Andrew Turpin (RMIT, Australia)
Sebastiano Vigna (Università degli Studi di Milano, Italy)
Arjen P. de Vries (CWI, The Netherlands)
Peter Widmayer (ETH Zurich, Switzerland)

Yi Zhang (University of California Santa Cruz, USA)
Nivio Ziviani (Federal University of Minas Gerais, Brazil)
Roelof van Zwol (Utrecht University, Netherlands)

Additional Reviewers

José Augusto Amgarten Quitzao, Vo Ngoc Anh, Diego Arroyuelo, Elham Ashoori, Claudine Badue, Bodo Billerbeck, Guillaume Blin, Ciccio Bozza, Pavel Calado, Ana Cardoso-Cachopo, Carlos Castillo , Jean-Marc Champarnaud, Massi Ciaramita, Raphael Clifford, Luís Coelho, Roberto Cornacchia, Thierson Couto Rosa, Marco Antonio Cristo, J. Shane Culpepper, Fabiano Cupertino Botelho, Shiri Dori, Gudrun Fischer, Matthias Fitzi, Ingo Frommholz, Lilia Greenenko, Sàndor Héman, MohammadTaghi Hajiaghayi, Danny Hermelin, Andreas Hildebrandt, Jan Holub, Sarvnaz Karimi, Shahar Keret, Tsvi Kopelowits, Adrian Kosowski, Thierry Lecroq, Liat Leventhal, Kan Liu, Sabrina Mantaci, Rudolf Mayer, Laurent Mouchard, Joong Chae Na, Nitsan Oz, Andreas Pesenhofer, Georg Poelzlbauer, Simon Puglisi, James F. Reid, Eric Rivals, Luís Russo, Klaus-Bernd Schürmann, Marinella Sciortino, Edleno Silva de Moura, Lynda Tamine, Theodora Tsikrika, Alexandra Uitdenbogerd, Marion Videau, Newton Jose Vieira, Jun Wang, Oren Weimann, YongHui Wu.

Previous Venues of SPIRE

The first four editions focused primarily on *string processing* and were held in South America. At the time SPIRE was called WSP (South American Workshop on String Processing). Starting in 1998, the focus of the workshop was broadened to include the area of *information retrieval* due to its increasing relevance and its inter-relationship with the area of string processing, changing to its current name. In addition, since 2000, the symposium started to alternate between Europe and Latin America, being held in Spain, Chile, Portugal, Brazil, and Italy in the last years. This is the first time that SPIRE was held in the United Kingdom.

2005: Buenos Aires, Argentina
2004: Padova, Italy
2003: Manaus, Brazil
2002: Lisboa, Portugal
2001: Laguna San Rafael, Chile
2000: A Coruna, Spain
1999: Cancun, Mexico
1998: Santa Cruz, Bolivia
1997: Valparaso, Chile
1996: Recife, Brazil
1995: Valparaso, Chile
1993: Belo Horizonte, Brazil

Table of Contents

Web Clustering and Text Categorization

Strings

User Behavior

Web Search Algorithms

Compression

Correction

Information Retrieval Applications

Bio Informatics

Web Search Engines

Short Papers

MP-Boost: A Multiple-Pivot Boosting Algorithm and Its Application to Text Categorization

Andrea Esuli, Tiziano Fagni, and Fabrizio Sebastiani

Istituto di Scienza e Tecnologia dell'Informazione
Consiglio Nazionale delle Ricerche
Via Giuseppe Moruzzi 1 – 56124 Pisa, Italy
{andrea.esuli, tiziano.fagni, fabrizio.sebastiani}@isti.cnr.it

Abstract. AdaBoost.MH is a popular supervised learning algorithm for building multi-label (aka n-of-m) text classifiers. AdaBoost.MH belongs to the family of "boosting" algorithms, and works by iteratively building a committee of "decision stump" classifiers, where each such classifier is trained to especially concentrate on the document-class pairs that previously generated classifiers have found harder to correctly classify. Each decision stump hinges on a specific "pivot term", checking its presence or absence in the test document in order to take its classification decision. In this paper we propose an improved version of AdaBoost.MH, called MP-Boost, obtained by selecting, at each iteration of the boosting process, not one but *several* pivot terms, one for each category. The rationale behind this choice is that this provides highly individualized treatment for each category, since each iteration thus generates, for each category, the best possible decision stump. We present the results of experiments showing that MP-Boost is much more effective than AdaBoost.MH. In particular, the improvement in effectiveness is spectacular when few boosting iterations are performed, and (only) high for many such iterations. The improvement is especially significant in the case of macroaveraged effectiveness, which shows that MP-Boost is especially good at working with hard, infrequent categories.

1 Introduction

Given a set of textual documents D and a predefined set of *categories* (aka *labels*) $C = \{c_1, \ldots, c_m\}$, *multi-label* (aka n-of-m) *text classification* is the task of approximating, or estimating, an unknown *target function* $\Phi : D \times C \rightarrow \{-1, +1\}$, that describes how documents ought to be classified, by means of a function $\hat{\Phi} : D \times C \rightarrow \{-1, +1\}$, called the *classifier*, such that Φ and $\hat{\Phi}$ "coincide as much as possible". Here, "multi-label" indicates that the same document can belong to zero, one, or several categories at the same time.

AdaBoost.MH [1] is a popular supervised learning algorithm for building multi-label text classifiers. AdaBoost.MH belongs to the family of "boosting" algorithms (see [2] for a review), which have enjoyed a wide popularity in the text categorization and filtering community because of their state-of-the-art

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 1–12, 2006.

effectiveness and of the strong justifications they have received from computational learning theory. AdaBoost.MH works by iteratively building a committee of "decision stump" classifiers[1], where each such classifier is trained to especially concentrate on the document-category pairs that previously generated classifiers have found harder to correctly classify. Each decision stump hinges on a specific "pivot term", and takes its classification decision based on the presence or absence of the pivot term in the test document.

We here propose an improved version of AdaBoost.MH, called MP-Boost, obtained by selecting, at each iteration of the boosting process, not one but *several* pivot terms, one for each category. The rationale behind this choice is that this provides highly individualized treatment for each category, since each iteration generates, for each category, the best possible decision stump. The result of the learning process is thus not a single classifier committee, but a set of such committees, one for each category.

The paper is structured as follows. In Section 2 we concisely describe boosting and the AdaBoost.MH algorithm. Section 3 describes in detail our MP-Boost algorithm and the rationale behind it. In Section 4 we present experimental results comparing AdaBoost.MH and MP-Boost. Section 5 concludes.

2 An Introduction to Boosting and AdaBoost.MH

AdaBoost.MH [1] (see Figure 1) is a *boosting* algorithm, i.e. an algorithm that generates a highly accurate classifier (also called *final hypothesis*) by combining a set of moderately accurate classifiers (also called *weak hypotheses*). The input to the algorithm is a training set $Tr = \{\langle d_1, C_1\rangle, \ldots, \langle d_g, C_g\rangle\}$, where $C_i \subseteq C$ is the set of categories to each of which d_i belongs.

AdaBoost.MH works by iteratively calling a *weak learner* to generate a sequence $\hat{\Phi}_1, \ldots, \hat{\Phi}_S$ of weak hypotheses; at the end of the iteration the final hypothesis $\hat{\Phi}$ is obtained as a sum $\hat{\Phi} = \sum_{s=1}^{S} \hat{\Phi}_s$ of these weak hypotheses. A weak hypothesis is a function $\hat{\Phi}_s : D \times C \rightarrow \mathbf{R}$. We interpret the sign of $\hat{\Phi}_s(d_i, c_j)$ as the prediction of $\hat{\Phi}_s$ on whether d_i belongs to c_j, i.e. $\hat{\Phi}_s(d_i, c_j) > 0$ means that d_i is believed to belong to c_j while $\hat{\Phi}_s(d_i, c_j) < 0$ means it is believed not to belong to c_j. We instead interpret the absolute value of $\hat{\Phi}_s(d_i, c_j)$ (indicated by $|\hat{\Phi}_s(d_i, c_j)|$) as the strength of this belief.

At each iteration s AdaBoost.MH tests the effectiveness of the newly generated weak hypothesis $\hat{\Phi}_s$ on the training set and uses the results to update a distribution D_s of weights on the training pairs $\langle d_i, c_j\rangle$. The weight $D_{s+1}(d_i, c_j)$ is meant to capture how effective $\hat{\Phi}_1, \ldots, \hat{\Phi}_s$ have been in correctly predicting whether the training document d_i belongs to category c_j or not. By passing (together with the training set Tr) this distribution to the weak learner, AdaBoost.MH forces this latter to generate a new weak hypothesis $\hat{\Phi}_{s+1}$ that concentrates on the pairs with the highest weight, i.e. those that had proven harder to classify for the previous weak hypotheses.

[1] A *decision stump* is a decision tree of depth one, i.e. consisting of a root node and two or more leaf nodes.

Input: A training set $Tr = \{\langle d_1, C_1\rangle, \ldots, \langle d_g, C_g\rangle\}$
where $C_i \subseteq C = \{c_1, \ldots, c_m\}$ for all $i = 1, \ldots, g$.

Body: Let $D_1(d_i, c_j) = \dfrac{1}{gm}$ for all $i = 1, \ldots, g$ and for all $j = 1, \ldots, m$
For $s = 1, \ldots, S$ do:

- pass distribution $D_s(d_i, c_j)$ to the weak learner;
- get the weak hypothesis $\hat{\Phi}_s$ from the weak learner;
- set $D_{s+1}(d_i, c_j) = \dfrac{D_s(d_i, c_j)\exp(-\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j))}{Z_s}$

  where $Z_s = \sum_{i=1}^{g}\sum_{j=1}^{m} D_s(d_i, c_j)\exp(-\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j))$

  is a normalization factor chosen so that $\sum_{i=1}^{g}\sum_{j=1}^{m} D_{s+1}(d_i, c_j) = 1$

Output: A final hypothesis $\hat{\Phi}(d, c) = \sum_{s=1}^{S}\hat{\Phi}_s(d, c)$

Fig. 1. The AdaBoost.MH algorithm

The initial distribution D_1 is uniform. At each iteration s all the weights $D_s(d_i, c_j)$ are updated to $D_{s+1}(d_i, c_j)$ according to the rule

$$D_{s+1}(d_i, c_j) = \frac{D_s(d_i, c_j)\exp(-\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j))}{Z_s} \tag{1}$$

where

$$Z_s = \sum_{i=1}^{g}\sum_{j=1}^{m} D_s(d_i, c_j)\exp(-\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j)) \tag{2}$$

is a normalization factor chosen so that D_{s+1} is in fact a distribution, i.e. so that $\sum_{i=1}^{g}\sum_{j=1}^{m} D_{s+1}(d_i, c_j) = 1$. Equation 1 is such that the weight assigned to a pair $\langle d_i, c_j\rangle$ misclassified by $\hat{\Phi}_s$ is increased, as for such a pair $\Phi(d_i, c_j)$ and $\hat{\Phi}_s(d_i, c_j)$ have different signs and the factor $\Phi(d_i, c_j)\cdot\hat{\Phi}_s(d_i, c_j)$ is thus negative; likewise, the weight assigned to a pair correctly classified by $\hat{\Phi}_s$ is decreased.

2.1 Choosing the Weak Hypotheses

In AdaBoost.MH each document d_i is represented as a vector $\langle w_{1i}, \ldots, w_{ri}\rangle$ of r binary weights, where $w_{ki} = 1$ (resp. $w_{ki} = 0$) means that term t_k occurs (resp. does not occur) in d_i; $T = \{t_1, \ldots, t_r\}$ is the set of terms that occur in at least one document in Tr.

In AdaBoost.MH the weak hypotheses generated by the weak learner at iteration s are decision stumps of the form

$$\hat{\Phi}_s(d_i, c_j) = \begin{cases} a_{0j} & \text{if } w_{ki} = 0 \\ a_{1j} & \text{if } w_{ki} = 1 \end{cases} \tag{3}$$

where t_k (called the *pivot term* of $\hat{\Phi}_s$) belongs to $\{t_1, \ldots, t_r\}$, and a_{0j} and a_{1j} are real-valued constants. The choices for t_k, a_{0j} and a_{1j} are in general different for each iteration s, and are made according to an error-minimization policy described in the rest of this section.

Schapire and Singer [3] have proven that the a reasonable (although suboptimal) way to maximize the effectiveness of the final hypothesis $\hat{\Phi}$ is to "greedily" choose each weak hypothesis $\hat{\Phi}_s$ (and thus its parameters t_k, a_{0j} and a_{1j}) in such a way as to minimize the normalization factor Z_s.

Schapire and Singer [1] define three different variants of AdaBoost.MH, corresponding to three different methods for making these choices. In this paper we concentrate on one of them, AdaBoost.MH *with real-valued predictions* (hereafter simply called AdaBoost.MH), since it is the one that, in [1], has been experimented most thoroughly and has given the best results; the modifications that we discuss in Section 3 straightforwardly apply also to the other two variants. AdaBoost.MH chooses weak hypotheses of the form described in Equation 3 by the following algorithm.

Algorithm 1 (The AdaBoost.MH weak learner)

1. *For each term $t_k \in \{t_1, \ldots, t_r\}$, select, among all the weak hypotheses $\hat{\Phi}$ that have t_k as the "pivot term", the one (indicated by $\hat{\Phi}_{best(k)}$) for which Z_s is minimum.*
2. *Among all the hypotheses $\hat{\Phi}_{best(1)}, \ldots, \hat{\Phi}_{best(r)}$ selected for the r different terms in Step 1, select the one (indicated by $\hat{\Phi}_s$) for which Z_s is minimum.*

Step 1 is clearly the key step, since there are a non-enumerable set of weak hypotheses with t_k as the pivot term. Schapire and Singer [3] have proven that, given term t_k and category c_j,

$$\hat{\Phi}_{best(k)}(d_i, c_j) = \begin{cases} \frac{1}{2} \ln \frac{W_{+1}^{0jk}}{W_{-1}^{0jk}} & \text{if } w_{ki} = 0 \\ \frac{1}{2} \ln \frac{W_{+1}^{1jk}}{W_{-1}^{1jk}} & \text{if } w_{ki} = 1 \end{cases} \tag{4}$$

where

$$W_b^{xjk} = \sum_{i=1}^{g} D_s(d_i, c_j) \cdot [\![w_{ki} = x]\!] \cdot [\![\Phi(d_i, c_j) = b]\!] \tag{5}$$

for $b \in \{-1, +1\}$, $x \in \{0, 1\}$, $j \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, r\}$, and where $[\![\pi]\!]$ indicates the characteristic function of predicate π (i.e. the function that returns 1 if π is true and 0 otherwise.

The output of the final hypothesis is the value $\hat{\Phi}(d_i, c_j) = \sum_{s=1}^{S} \hat{\Phi}_s(d_i, c_j)$ obtained by summing the outputs of the weak hypotheses.

2.2 Implementing AdaBoost.MH

Following [4], in our implementation of AdaBoost.MH we have further optimized the final hypothesis $\hat{\Phi}(d_i, c_j) = \sum_{s=1}^{S} \hat{\Phi}_s(d_i, c_j)$ by "compressing" the weak hypotheses $\hat{\Phi}_1, \ldots, \hat{\Phi}_S$ according to their pivot term t_k. In fact, note that if $\{\hat{\Phi}_1, \ldots, \hat{\Phi}_S\}$ contains a subset $\{\hat{\Phi}_1^{(k)}, \ldots, \hat{\Phi}_{q(k)}^{(k)}\}$ of weak hypotheses that all hinge on the same pivot term t_k and are of the form

$$\hat{\Phi}_r^{(k)}(d_i, c_j) = \begin{cases} a_{0j}^r \text{ if } w_{ki} = 0 \\ a_{1j}^r \text{ if } w_{ki} = 1 \end{cases} \tag{6}$$

for $r = 1, \ldots, q(k)$, the collective contribution of $\hat{\Phi}_1^{(k)}, \ldots, \hat{\Phi}_{q(k)}^{(k)}$ to the final hypothesis is the same as that of a "combined hypothesis"

$$\hat{\Phi}^{(k)}(d_i, c_j) = \begin{cases} \sum_{r=1}^{q(k)} a_{0j}^r \text{ if } w_{ki} = 0 \\ \sum_{r=1}^{q(k)} a_{1j}^r \text{ if } w_{ki} = 1 \end{cases} \tag{7}$$

In the implementation we have thus replaced $\sum_{s=1}^{S} \hat{\Phi}_s(d_i, c_j)$ with $\sum_{k=1}^{\Delta} \hat{\Phi}^{(k)}(d_i, c_j)$, where Δ is the number of different terms that act as pivot for the weak hypotheses in $\{\hat{\Phi}_1, \ldots, \hat{\Phi}_S\}$.

This modification brings about a considerable efficiency gain in the application of the final hypothesis to a test example. For instance, the final hypothesis we obtained on Reuters-21578 with AdaBoost.MH when $S = 1000$ consists of 1000 weak hypotheses, but the number of different pivot terms is only 766 (see Section 4.2). The reduction in the size of the final hypothesis which derives from this modification is usually larger when high reduction factors have been applied in a feature selection phase, since in this case the number of different terms that can be chosen as the pivot is smaller.

3 MP-Boost, an Improved Boosting Algorithm with Multiple Pivot Terms

We here propose an improved version of AdaBoost.MH, dubbed AdaBoost.MH *with multiple pivot terms* (here nicknamed MP-Boost), that basically consists in modifying the form of weak hypotheses and how they are generated. Looking at Equation 3 we may note that, at each iteration s, choosing a weak hypothesis means choosing (i) a pivot term t_k, *the same for all categories*, and (ii) for each category c_j, a pair of constants $\langle a_{0j}, a_{1j} \rangle$. We contend that the fact that, at iteration s, the same term t_k is chosen as the pivot term on which the binary classifiers for all categories hinge, is clearly suboptimal. At this iteration term t_k may be a very good discriminator for category c', but a very poor discriminator for category c'', which means that the weak hypothesis generated at this iteration would contribute very little to the correct classification of documents under c''. We claim that choosing, at every iteration s, a different pivot term $t_{\langle s,j \rangle}$ for each category c_j allows the weak hypothesis to provide customized treatment to each

individual category. In MP-Boost the weak hypotheses generated by the weak learner at iteration s are thus of the form

$$\hat{\Phi}_s(d_i, c_j) = \begin{cases} a_{0j} \text{ if } w_{\langle s,j\rangle i} = 0 \\ a_{1j} \text{ if } w_{\langle s,j\rangle i} = 1 \end{cases} \tag{8}$$

where term $t_{\langle s,j\rangle}$ is the pivot term chosen for category c_j at iteration s. To see how MP-Boost chooses weak hypotheses of the form described in Equation 8, let us first define a *weak c_j-hypothesis* as a function

$$\hat{\Phi}^j(d_i) = \begin{cases} a_{0j} \text{ if } w_{ki} = 0 \\ a_{1j} \text{ if } w_{ki} = 1 \end{cases} \tag{9}$$

that is only concerned with classifying documents under c_j; a weak hypothesis is the union of weak c_j-hypotheses, one for each $c_j \in C$. At each iteration s, MP-Boost chooses a weak hypothesis $\hat{\Phi}_s$ by means of the following algorithm.

Algorithm 2 (The MP-Boost weak learner)

1. *For each category c_j and for each term $t_k \in \{t_1, \ldots, t_r\}$, select, among all weak c_j-hypothesis $\hat{\Phi}^j$ that have t_k as the pivot term, the one (indicated by $\hat{\Phi}^j_{best(k)}$) which minimizes*

$$Z_s^j = \sum_{i=1}^{g} D_s(d_i, c_j) \exp(-\Phi(d_i, c_j) \cdot \hat{\Phi}^j(d_i)) \tag{10}$$

2. *For each category c_j, among all the hypotheses $\hat{\Phi}^j_{best(1)}, \ldots, \hat{\Phi}^j_{best(r)}$ selected in Step 1 for the r different terms, select the one (indicated by $\hat{\Phi}^j_s$) for which Z_s^j is minimum;*
3. *Choose, as the weak hypothesis $\hat{\Phi}_s$, the "union", across all $c_j \in C$, of the weak c_j-hypotheses selected in Step 2, i.e. the function such that $\hat{\Phi}_s(d_i, c_j) = \hat{\Phi}^j_s(d_i)$.*

Note the difference between Algorithm 1, as described in Section 2.1, and Algorithm 2; in particular, Step 2 of Algorithm 2 is such that weak c_j-hypotheses based on different pivot terms may be chosen for different categories c_j.

For reasons analogous to the ones discussed in Section 2.1, Step 1 is the key step; it is important to observe that $\hat{\Phi}^j_{best(k)}$ is still guaranteed to have the form described in Equation 4, since the weak hypothesis generated by Equation 8 is the same that Equation 3 generates when $m = 1$, i.e. when C contains one category only.

Note also that the "outer" algorithm of Figure 1 is untouched by our modifications, except for the fact that a normalization factor Z_s^j local to each category c_j is used (in place of the "global" normalization factor Z_s) in order to obtain the revised distribution D_{s+1}; i.e. $D_{s+1}(d_i, c_j) = \frac{D_s(d_i,c_j) \exp(-\Phi(d_i,c_j) \cdot \hat{\Phi}^j(d_i))}{Z_s^j}$. The main difference in the algorithm is thus in the "inner" part, i.e. in the weak

hypotheses that are received from the weak learner, which now have the form of Equation 8, and in the way they are generated.

Concerning the optimizations discussed in Section 2.2, obtained by merging into a single weak hypothesis all weak hypotheses that share the same pivot term, note that in MP-BOOST these must be done on a category-by-category basis, i.e. by merging into a single weak c_j-hypothesis all weak c_j-hypotheses that share the same pivot term. The effect of this is that the different categories $c_1, \ldots, c_m$ may be associated to final hypotheses consisting of different numbers $\Delta_1, \ldots, \Delta_m$ of weak hypotheses.

Last, let us note that one consequence of switching from ADABOOST.MH to MP-BOOST is that *local* feature selection (i.e. choosing different reduced feature sets for different categories) can also be used in place of *global* feature selection (i.e. choosing the same reduced feature set for all categories). In fact, since in MP-BOOST the choice of pivot terms is category-specific, the vectorial representations of documents can also be category-specific. This allows the designer to select, ahead of the learning phase and by means of standard feature selection techniques, the terms that are the most discriminative for a given category c_j, and are thus highly likely to be chosen as pivot terms for the c_j-hypotheses. This can be done separately for each individual category, and thus allows the use of even higher reduction factors; from the standpoint of efficiency this is advantageous, given that the computational cost of MP-BOOST has a linear dependence on the number of features used (see Section 3).

In an extended version of this paper [5] we discuss the computational cost of MP-BOOST, proving that:

- At training time both ADABOOST.MH and MP-BOOST are $O(grm)$.
- At testing time, at a first approximation, ADABOOST.MH can be shown to be $O(S)$, while MP-BOOST is instead $O(mS)$. In practice, since weak hypotheses are "compressed", as described in Section 2.2, for both learners the cost linearly depends on Δ, the number of distinct pivot terms selected during the training process (for MP-BOOST, we take Δ to be an average of the category-specific Δ_i values). For a given value of S the value of Δ tends to be much smaller for MP-BOOST than for ADABOOST.MH, since the "good" pivot terms for a specific category tend to be few. As a result, for the testing phase the differential in cost between the two algorithms is, in practice, much smaller than the upper bounds discussed above seem to suggest.

4 Experiments

4.1 Experimental Setting

In our experiments we have used the REUTERS-21578 and RCV1-V2 corpora.

"REUTERS-21578, Distribution 1.0" is currently the most widely used benchmark in multi-label text categorization research[2]. It consists of a set of 12,902

[2] `http://www.daviddlewis.com/resources/testcollections/~reuters21578/`

news stories, partitioned (according to the "ModApté" split we have adopted) into a training set of 9,603 documents and a test set of 3,299 documents. The documents are labelled by 118 categories; in our experiments we have restricted our attention to the 115 categories with at least one positive training example.

Reuters Corpus Volume 1 version 2 (RCV1-v2)[3] is a more recent text categorization benchmark made available by Reuters and consisting of 804,414 news stories produced by Reuters from 20 Aug 1996 to 19 Aug 1997. In our experiments we have used the "LYRL2004" split, defined in [6], in which the (chronologically) first 23,149 documents are used for training and the other 781,265 are used for test. Of the 103 "Topic" categories, in our experiments we have restricted our attention to the 101 categories with at least one positive training example.

In all the experiments discussed in this paper, stop words have been removed, punctuation has been removed, all letters have been converted to lowercase, numbers have been removed, and stemming has been performed by means of Porter's stemmer. Feature selection has been performed by scoring features by means of *information gain*, defined as $IG(t_k, c_i) = \sum_{c \in \{c_i, \overline{c}_i\}} \sum_{t \in \{t_k, \overline{t}_k\}} P(t, c) \cdot \log \frac{P(t,c)}{P(t) \cdot P(c)}$. The final set of features has been chosen according to Forman's *round robin* technique, which consists in picking, for each category c_i, the v features with the highest $IG(t_k, c_i)$ value, and pooling all of them together into a category-independent set [7]. This set thus contains at most vm features, where m is the number of categories; it usually contains strictly fewer than vm features, since some features are among the best v features for more than one category. We have set v to 48 (for Reuters-21578) and 177 (for RCV1-v2); these are the values that bring about feature set sizes of 2,012 (Reuters-21578) and 5,509 (RCV1-v2), thus achieving 90% reduction with respect to the original sets which consisted of 20,123 (Reuters-21578) and 55,051 (RCV1-v2) terms.

As a measure of effectiveness that combines the contributions of *precision* (π) and *recall* (ρ) we have used the well-known F_1 function, defined as $F_1 = \frac{2\pi\rho}{\pi+\rho} = \frac{2TP}{2TP+FP+FN}$, where TP, FP, and FN stand for the numbers of true positives, false positives, and false negatives, respectively. We compute both microaveraged F_1 (denoted by F_1^μ) and macroaveraged F_1 (F_1^M). F_1^μ is obtained by (i) computing the category-specific values TP_i, (ii) obtaining TP as the sum of the TP_i's (same for FP and FN), and then (iii) applying the $F_1 = \frac{2\pi\rho}{\pi+\rho}$ formula. F_1^M is obtained by first computing the category-specific F_1 values and then averaging them across the c_i's.

4.2 Results

The results of our experiments are reported in Table 1 for some key values of the number of iterations S; Figure 2 reports the same results in graphical form for any value of S comprised in the [1..1000] interval. It is immediate to observe that, for any value of S, MP-Boost always improves on AdaBoost.MH, in terms of both F_1^μ and F_1^M.

[3] `http://trec.nist.gov/data/reuters/reuters.html`

Let us discuss the results obtained on REUTERS-21578 (the ones obtained on RCV1-v2 are qualitatively similar)[4]. For small values of S the improvement in effectiveness of MP-BOOST wrt ADABOOST is spectacular: F_1^μ goes up by +69.47% for $S = 5$, by +57.07% for $S = 10$, and by +30.07% for $S = 20$. As the value of S grows, the margin between the two learners narrows: we obtain +4.34% for $S = 1,000$ and +4.20% for $S = 10,000$. This fact may be explained by noting that in ADABOOST.MH, if the final hypothesis consists of a few weak hypotheses only, it is likely that only few categories have been properly addressed (i.e. that the pivot terms used in the committee have a high discrimination power for few categories only). When the number of weak hypotheses gets larger, it is more likely that many (or most of the) categories have been properly catered for. Conversely, MP-BOOST has already used the best pivot terms for each category right from the very first iterations; this explains the fact that MP-BOOST is highly effective even for small values of S.

Note that the improvement brought about by the individualized treatment of categories implemented by MP-BOOST is not recovered by ADABOOST.MH even by pushing S to the limit. For instance, note that not even in 10,000 iterations ADABOOST.MH manages to obtain the F_1^μ values obtained by MP-BOOST in just 50 iterations: MP-BOOST with $S = 50$ obtains a slightly superior effectiveness (+1.4%) than ADABOOST.MH with $S = 10,000$, in less than 1% the training time and in about 10% the testing time of this latter.

These effectiveness improvements are even more significant when considering macroaveraged effectiveness. In this case, we obtain a relative improvement in F_1^M that ranges from a minimum of +21.13% (obtained for $S = 10,000$) to a maximum of +124,70% (obtained for $S = 5$). Again, not even in 10,000 iterations ADABOOST.MH obtains the F_1^M values obtained by MP-BOOST in just 5 iterations. This may be explained by recalling the well-known fact that macroaveraged effectiveness especially rewards those classifiers that perform well also on infrequent categories (i.e. categories with few positive training examples); indeed, unlike ADABOOST.MH, MP-BOOST places equal emphasis on all categories, regardless of their frequency, thus picking the very best pivot terms for the infrequent categories too right from the first iterations.

Let us now discuss the relative efficiency of the two learners. As expected, for both learners the time required to generate the final committees grows linearly with the number of boosting iterations S. We also observed an almost constant ratio between the running times of the two learners, with MP-BOOST being about 9% slower than ADABOOST.MH. A profiling session on the applications has pointed out that this difference is due to the larger (by a constant factor)

[4] The reader might notice that the best performance we have obtained from ADABOOST.MH on REUTERS-21578 ($F_1^\mu = .808$) is inferior to the one reported in [1] for the same algorithm ($F_1^\mu = .851$). There are several reasons for this: (a) [1] actually uses a different, much older version of this collection, called REUTERS-21450 [8]; (b) [1] only uses the 93 categories which have at least 2 positive training examples and 1 positive test example, while we also use the categories that have just 1 positive training example and those that have no positive test example. This makes the two sets of ADABOOST.MH results difficult to compare.

Table 1. Comparative performance of AdaBoost.MH and MP-Boost on the Reuters-21578 and RCV1-v2 benchmarks, with (i) a full feature set and with (ii) a reduced feature set obtained with a round-robin technique and 90% reduction factor. S indicates the number of boosting iterations; F_1^μ and F_1^M indicate micro- and macro-averaged F_1, respectively; $\tau(Tr)$ and $\tau(Te)$ indicate the time (in seconds) required for training and testing, respectively.

		AdaBoost.MH				MP-Boost				MP-Boost wrt AdaBoost.MH			
	S	F_1^μ	F_1^M	$\tau(Tr)$	$\tau(Te)$	F_1^μ	F_1^M	$\tau(Tr)$	$\tau(Te)$	F_1^μ	F_1^M	$\tau(Tr)$	$\tau(Te)$
Reuters-21578 full feature set	5	0.416	0.235	12.1	0.1	0.704	0.529	13.2	0.2	+69.47%	+124.70%	+9.09%	+100.0%
	10	0.483	0.271	24.2	0.1	0.759	0.556	26.4	0.3	+57.07%	+105.52%	+9.09%	+183.3%
	20	0.611	0.325	48.4	0.1	0.795	0.586	52.8	0.5	+30.07%	+80.44%	+9.09%	+266.6%
	50	0.723	0.392	96.8	0.2	0.822	0.589	105.7	1.1	+13.79%	+50.44%	+9.19%	+324.0%
	100	0.776	0.454	193.6	0.4	0.837	0.608	211.3	1.7	+7.91%	+34.06%	+9.14%	+326.8%
	200	0.798	0.461	387.1	0.8	0.843	0.600	422.7	3.1	+5.68%	+30.16%	+9.20%	+297.4%
	500	0.811	0.485	774.2	2.0	0.848	0.604	845.3	6.3	+4.51%	+24.62%	+9.18%	+216.1%
	1000	0.811	0.482	1548.4	3.7	0.846	0.603	1690.6	9.2	+4.34%	+25.06%	+9.18%	+150.1%
	10000	0.810	0.497	15483.9	10.0	0.844	0.602	16906.2	20.6	+4.20%	+21.13%	+9.18%	+106.0%
RCV1-v2 full feature set	5	0.361	0.037	34.5	21.8	0.519	0.306	37.3	54.0	+43.89%	+720.57%	+8.12%	+147.9%
	10	0.406	0.070	69.1	25.5	0.588	0.367	74.7	91.8	+44.80%	+421.88%	+8.10%	+260.7%
	20	0.479	0.131	138.1	32.7	0.646	0.418	149.4	148.5	+34.96%	+218.09%	+8.18%	+354.7%
	50	0.587	0.239	276.2	54.6	0.700	0.455	298.7	286.2	+19.24%	+90.63%	+8.15%	+423.8%
	100	0.650	0.333	552.4	87.5	0.726	0.474	597.5	472.5	+11.75%	+42.33%	+8.16%	+439.8%
	200	0.701	0.396	1104.8	161.5	0.745	0.487	1194.9	837.0	+6.20%	+23.00%	+8.16%	+418.3%
	500	0.735	0.435	2209.7	516.1	0.761	0.495	2389.9	1698.3	+3.58%	+13.74%	+8.15%	+229.1%
	1000	0.745	0.442	4419.3	1014.4	0.768	0.496	4779.7	2478.6	+2.99%	+12.21%	+8.16%	+144.4%
	10000	0.754	0.459	44192.3	2831.4	0.765	0.485	47796.2	5772.4	+1.46%	+5.66%	+8.16%	+103.9%
Reuters-21578 red. feature set	5	0.416	0.235	9.3	0.1	0.704	0.515	10.2	0.2	+69.23%	+119.15%	+9.68%	+133.3%
	10	0.483	0.271	18.5	0.1	0.760	0.560	20.4	0.3	+57.35%	+106.64%	+10.27%	+200.0%
	20	0.611	0.325	37.1	0.1	0.794	0.567	40.7	0.5	+29.95%	+74.46%	+9.70%	+307.7%
	50	0.723	0.392	74.1	0.2	0.826	0.596	81.4	1.0	+14.25%	+52.04%	+9.85%	+325.0%
	100	0.773	0.457	148.3	0.4	0.839	0.614	162.9	1.7	+8.54%	+34.35%	+9.84%	+315.0%
	200	0.790	0.474	296.5	0.7	0.845	0.623	325.8	2.9	+6.96%	+31.43%	+9.88%	+288.0%
	500	0.811	0.485	593.0	1.9	0.846	0.617	651.5	5.8	+4.32%	+27.22%	+9.87%	+202.1%
	1000	0.806	0.484	1186.0	3.2	0.839	0.619	1303.0	8.2	+4.09%	+27.89%	+9.87%	+153.2%
RCV1-v2 red. feature set	5	0.361	0.037	28.2	21.1	0.519	0.307	30.5	49.6	+43.77%	+729.73%	+8.16%	+135.6%
	10	0.406	0.070	56.4	24.4	0.587	0.365	61.0	78.0	+44.58%	+421.43%	+8.16%	+219.3%
	20	0.479	0.131	112.7	31.2	0.646	0.416	122.1	125.6	+34.86%	+217.56%	+8.34%	+302.1%
	50	0.587	0.239	225.4	54.6	0.701	0.458	244.2	247.4	+19.42%	+91.63%	+8.34%	+352.8%
	100	0.650	0.333	450.9	84.6	0.727	0.478	488.4	442.3	+11.85%	+43.54%	+8.32%	+422.7%
	200	0.701	0.396	901.8	154.4	0.744	0.493	976.8	896.9	+6.13%	+24.49%	+8.32%	+481.0%
	500	0.734	0.431	1803.5	495.9	0.760	0.503	1953.5	2133.1	+3.54%	+16.71%	+8.32%	+330.2%
	1000	0.747	0.445	3607.0	974.7	0.764	0.505	3907.0	3500.6	+2.28%	+13.48%	+8.32%	+259.2%

size of weak hypotheses in MP-Boost (see Section 3), which generates a small overhead in memory management. In terms of testing time, instead, it turns out that MP-Boost is, for equal numbers S of boosting iterations, from one to four times slower than AdaBoost.MH (see Table 1). This is due to the fact that AdaBoost.MH selects, for the same value S, a number Δ of distinct pivot terms smaller than the number $\sum_{i=1}^{m} \Delta_i$ that MP-Boost selects (see Section 2.2), and to the fact that the classifier tests all the values of these terms in the document. However, note that for MP-Boost this loss in testing efficiency is more than compensated by the large gain in effectiveness. Also, with MP-Boost trained on the full feature set with $S = 1000$ (a value at which effectiveness peaks) the time required for classifying all the 781,265 RCV1-v2 test documents is about 79 minutes, which is more than acceptable.

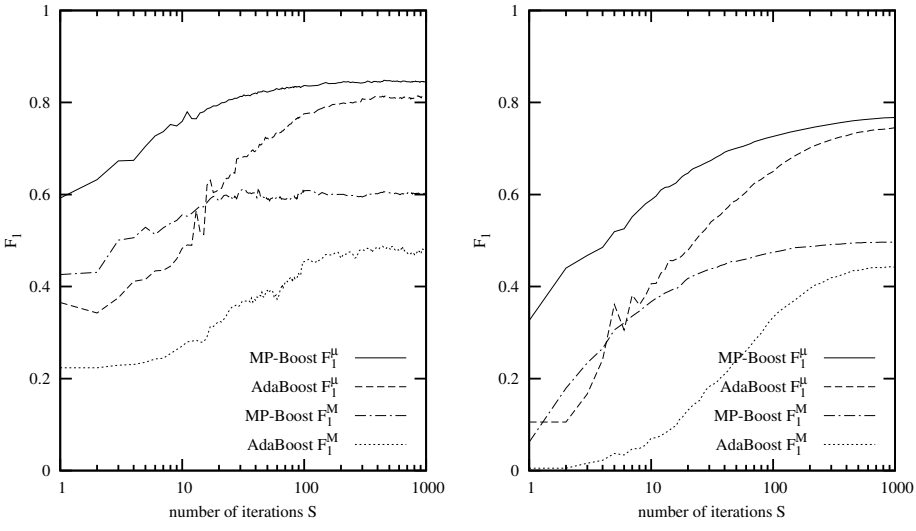


Fig. 2. Effectiveness of AdaBoost.MH and MP-Boost on Reuters-21578 (left) and RCV1-v2 (right) as a function of the number S of iterations. The X axis is displayed on a logarithmic scale.

Last, let us note that the experiments run with the reduced feature set (see Table 1) have produced practically unchanged effectiveness results wrt those obtained with the full feature set, but (as expected – see Section 3) at the advantage of dramatically smaller training times and substantially smaller testing times. That feature selection does not reduce effectiveness might seem surprising in the context of a boosting algorithm, since feature selection brings about smaller degrees of freedom in the choice of the best pivot term; quite evidently, IG is very effective at discarding the terms that the boosting algorithm would not choose anyway as pivots.

5 Conclusion

We have presented MP-Boost, a novel algorithm for multi-label text categorization that improves upon the well-known AdaBoost.MH algorithm by selecting multiple pivot terms at each boosting iteration, we have provided (training time and testing time) complexity results for it, and we have thoroughly tested it on two well-known benchmarks, one of which consisting of more than 800,000 documents. The results allow us to conclude that MP-Boost is a largely superior alternative to AdaBoost.MH since, at the price of a tolerable decrease in classification efficiency, it yields speedier convergence, superior microaveraged effectiveness, and dramatically superior macroaveraged effectiveness. This latter fact makes it especially suitable to categorization problems in which the distribution of training examples across the categories is highly skewed.

References

1. Schapire, R.E., Singer, Y.: BoosTexter: a boosting-based system for text categorization. Machine Learning **39**(2/3) (2000) 135–168
2. Meir, R., Rätsch, G.: An introduction to boosting and leveraging. In Mendelson, S., Smola, A.J., eds.: Advanced lectures on machine learning. Springer Verlag, Heidelberg, DE (2003) 118–183
3. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. Machine Learning **37**(3) (1999) 297–336
4. Sebastiani, F., Sperduti, A., Valdambrini, N.: An improved boosting algorithm and its application to automated text categorization. In: Proceedings of the 9th ACM International Conference on Information and Knowledge Management (CIKM'00), McLean, US (2000) 78–85
5. Esuli, A., Fagni, T., Sebastiani, F.: MP-Boost: A multiple-pivot boosting algorithm and its application to text categorization. Technical Report 2006-TR-56, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, IT (2006) Submitted for publication.
6. Lewis, D.D., Li, F., Rose, T., Yang, Y.: RCV1: A new benchmark collection for text categorization research. Journal of Machine Learning Research **5** (2004) 361–397
7. Forman, G.: A pitfall and solution in multi-class feature selection for text classification. In: Proceedings of the 21st International Conference on Machine Learning (ICML'04), Banff, CA (2004)
8. Apté, C., Damerau, F.J., Weiss, S.M.: Automated learning of decision rules for text categorization. ACM Transactions on Information Systems **12**(3) (1994) 233–251

TreeBoost.MH: A Boosting Algorithm for Multi-label Hierarchical Text Categorization

Andrea Esuli, Tiziano Fagni, and Fabrizio Sebastiani

Istituto di Scienza e Tecnologia dell'Informazione
Consiglio Nazionale delle Ricerche
Via Giuseppe Moruzzi 1 – 56124 Pisa, Italy
{andrea.esuli, tiziano.fagni, fabrizio.sebastiani}@isti.cnr.it

Abstract. In this paper we propose TreeBoost.MH, an algorithm for multi-label *Hierarchical Text Categorization* (HTC) consisting of a hierarchical variant of AdaBoost.MH. TreeBoost.MH embodies several intuitions that had arisen before within HTC: e.g. the intuitions that both feature selection and the selection of negative training examples should be performed "locally", i.e. by paying attention to the topology of the classification scheme. It also embodies the novel intuition that the weight distribution that boosting algorithms update at every boosting round should likewise be updated "locally". We present the results of experimenting TreeBoost.MH on two HTC benchmarks, and discuss analytically its computational cost.

1 Introduction

Hierarchical text categorization (HTC) is the task of generating text classifiers that operate on classification schemes endowed with a hierarchical structure. Notwithstanding the fact that most large-sized classification schemes for text (e.g. the ACM Classification Scheme[1]) indeed have a hierarchical structure, so far the attention of text classification (TC) researchers has mostly focused on algorithms for "flat" classification, i.e. algorithms that operate on non-hierarchical classification schemes. These algorithms, once applied to a hierarchical classification problem, are not capable of taking advantage of the information inherent in the class hierarchy. On the contrary, many researchers have argued that by leveraging on the hierarchical structure of the classification scheme, heuristics of various kinds can be brought to bear that make the classifier more efficient and/or more effective. Many of these heuristics have been used in close association with a specific learning algorithm; the most popular choices in this respect have been naïve Bayesian methods [1, 2, 3, 4, 5, 6], neural networks [7, 8, 9], support vector machines [10, 11], and example-based classifiers [11].

Within this literature, the absence of "boosting" methods is conspicuous: to the best of our knowledge, we do not know of any HTC method belonging to the boosting family. This is somehow surprising, (i) because of the high applicative interest of HTC, (ii) because boosting algorithms are well-known for their

[1] http://info.acm.org/class/1998/ccs98.html

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 13–24, 2006.

interesting theoretical properties and for their high accuracy, and (iii) because, given their relatively high computational cost, they would definitely benefit by the added efficiency that consideration of the hierarchical structure can bring about.

In this paper we try to fill this gap by proposing TreeBoost.MH, a multi-label HTC algorithm that consists of a hierarchical variant of AdaBoost.MH, a very well-known member of the family of boosting algorithms; here, *multi-label* (ML) means that a document can belong to zero, one, or several categories at the same time. TreeBoost.MH embodies several intuitions that had arisen before within HTC: e.g. the intuitions that both feature selection and the selection of negative training examples should be performed "locally", i.e. by paying attention to the topology of the classification scheme. TreeBoost.MH also incorporates the novel intuition that the weight distribution that boosting algorithms update at every boosting round should likewise be updated "locally". All these intuitions are embodied within TreeBoost.MH in an elegant and simple way, i.e. by defining TreeBoost.MH as a recursive algorithm that uses AdaBoost.MH as its base step, and that recurs over the tree structure.

The paper is structured as follows. In Section 2 we give a concise description of boosting and the AdaBoost.MH algorithm. Section 3 describes TreeBoost.MH. In Section 4 we present experiments comparing AdaBoost.MH and TreeBoost.MH on two well-known HTC benchmarks. Section 5 discusses related work. Section 6 concludes.

2 An Introduction to Boosting and AdaBoost.MH

AdaBoost.MH [12] is a *boosting* algorithm, i.e. an algorithm that generates a highly accurate classifier $\hat{\Phi}$ (also called *final hypothesis*) by combining a set of moderately accurate classifiers $\hat{\Phi}_1, \ldots, \hat{\Phi}_S$ (also called *weak hypotheses*). The input to the algorithm is a set of pairs $C = \{\langle c_1, Tr^+(c_1)\rangle, \ldots, \langle c_m, Tr^+(c_m)\rangle\}$ each consisting of a category and its set of positive training examples. For each $c_j \in C$, we define the set $Tr^-(c_j)$ of its negative training examples simply as the union of the sets of the positive training examples of the other categories, minus $Tr^+(c_j)$.

AdaBoost.MH works by iteratively calling a *weak learner* to generate a sequence $\hat{\Phi}_1, \ldots, \hat{\Phi}_S$ of weak hypotheses; at the end of the iteration the final hypothesis $\hat{\Phi}$ is obtained as a sum $\hat{\Phi} = \sum_{s=1}^{S} \hat{\Phi}_s$ of these weak hypotheses. A weak hypothesis is a function $\hat{\Phi}_s : D \times C \rightarrow \mathbb{R}$. We interpret the sign of $\hat{\Phi}_s(d_i, c_j)$ as the prediction of $\hat{\Phi}_s$ on whether d_i belongs to c_j, i.e. $\hat{\Phi}_s(d_i, c_j) > 0$ means that d_i is believed to belong to c_j while $\hat{\Phi}_s(d_i, c_j) < 0$ means it is believed not to belong to c_j. We instead interpret the absolute value of $\hat{\Phi}_s(d_i, c_j)$ (indicated by $|\hat{\Phi}_s(d_i, c_j)|$) as the strength of this belief.

At each iteration s AdaBoost.MH tests the effectiveness of the newly generated weak hypothesis $\hat{\Phi}_s$ on the training set and uses the results to update a distribution D_s of weights on the training pairs $\langle d_i, c_j\rangle$. The weight $D_{s+1}(d_i, c_j)$ is meant to capture how effective $\hat{\Phi}_1, \ldots, \hat{\Phi}_s$ have been in correctly predicting

whether the training document d_i belongs to category c_j or not. By passing (together with the training set Tr) this distribution to the weak learner, AdaBoost.MH forces this latter to generate a new weak hypothesis $\hat{\Phi}_{s+1}$ that concentrates on the pairs with the highest weight, i.e. those that had proven harder to classify for the previous weak hypotheses.

The initial distribution D_1 is uniform. At each iteration s all the weights $D_s(d_i, c_j)$ are updated to $D_{s+1}(d_i, c_j)$ according to the rule

$$D_{s+1}(d_i, c_j) = \frac{D_s(d_i, c_j) \exp(-\Phi(d_i, c_j) \cdot \hat{\Phi}_s(d_i, c_j))}{Z_s} \quad (1)$$

where the *target function* $\Phi(d_i, c_j)$ is defined to be 1 if $c_j \in C_i$ and -1 otherwise, and $Z_s = \sum_{i=1}^{g} \sum_{j=1}^{m} D_s(d_i, c_j) \exp(-\Phi(d_i, c_j) \cdot \hat{\Phi}_s(d_i, c_j))$ is a normalization factor chosen so that $\sum_{i=1}^{g} \sum_{j=1}^{m} D_{s+1}(d_i, c_j) = 1$, i.e. so that D_{s+1} is in fact a distribution. Equation 1 is such that the weight assigned to a pair $\langle d_i, c_j \rangle$ misclassified by $\hat{\Phi}_s$ is increased, as for such a pair $\Phi(d_i, c_j)$ and $\hat{\Phi}_s(d_i, c_j)$ have different signs and the factor $\Phi(d_i, c_j) \cdot \hat{\Phi}_s(d_i, c_j)$ is thus negative; likewise, the weight assigned to a pair correctly classified by $\hat{\Phi}_s$ is decreased.

2.1 Choosing the Weak Hypotheses

In AdaBoost.MH each document d_i is represented as a vector $\langle w_{1i}, \ldots, w_{ri} \rangle$ of r binary weights, where $w_{ki} = 1$ (resp. $w_{ki} = 0$) means that term t_k occurs (resp. does not occur) in d_i; $T = \{t_1, \ldots, t_r\}$ is the set of terms that occur in at least one document in Tr.

In AdaBoost.MH the weak hypotheses generated by the weak learner at iteration s are decision stumps of the form

$$\hat{\Phi}_s(d_i, c_j) = \begin{cases} a_{0j} & \text{if } w_{ki} = 0 \\ a_{1j} & \text{if } w_{ki} = 1 \end{cases} \quad (2)$$

where t_k (called the *pivot term* of $\hat{\Phi}_s$) belongs to T, and a_{0j} and a_{1j} are real-valued constants. The choices for t_k, a_{0j} and a_{1j} are in general different for each iteration s, and are made according to an error-minimization policy described in the rest of this section.

Schapire and Singer [13] have proven that a reasonable (although suboptimal) way to maximize the effectiveness of the final hypothesis $\hat{\Phi}$ is to "greedily" choose each weak hypothesis $\hat{\Phi}_s$ (and thus its parameters t_k, a_{0j} and a_{1j}) in such a way as to minimize Z_s.

Schapire and Singer [12] define three different variants of AdaBoost.MH, corresponding to three different methods for making these choices. In this paper we concentrate on one of them, AdaBoost.MH *with real-valued predictions* (hereafter simply called AdaBoost.MH), since it is the one that, in [12], has been experimented most thoroughly and has given the best results:

Algorithm 1 (The AdaBoost.MH weak learner)

1. *For each term* $t_k \in \{t_1, \ldots, t_r\}$ *select, among all weak hypotheses* $\hat{\Phi}$ *that have* t_k *as the "pivot term", the one (indicated by* $\hat{\Phi}_{best(k)}$*) for which* Z_s *is minimum.*
2. *Among all the hypotheses* $\hat{\Phi}_{best(1)}, \ldots, \hat{\Phi}_{best(r)}$ *selected for the* r *different terms in Step 1, select the one (indicated by* $\hat{\Phi}_s$*) for which* Z_s *is minimum.*

Step 1 is clearly the key step, since there are a non-enumerable set of weak hypotheses with t_k as the pivot. Schapire and Singer [13] have proven that, given term t_k and category c_j,

$$\hat{\Phi}_{best(k)}(d_i, c_j) = \begin{cases} \frac{1}{2} \ln \frac{W_{+1}^{0jk}}{W_{-1}^{0jk}} & \text{if } w_{ki} = 0 \\ \frac{1}{2} \ln \frac{W_{+1}^{1jk}}{W_{-1}^{1jk}} & \text{if } w_{ki} = 1 \end{cases} \tag{3}$$

where $W_b^{xjk} = \sum_{i=1}^{g} D_s(d_i, c_j) \cdot [\![w_{ki} = x]\!] \cdot [\![\Phi(d_i, c_j) = b]\!]$ for $b \in \{1, -1\}$, $x \in \{0, 1\}$, $j \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, r\}$, and where $[\![\pi]\!]$ indicates the function that returns 1 if π is true and 0 otherwise.

3 A Hierarchical Boosting Algorithm Multi-label TC

In this section we describe a version of AdaBoost.MH, called TreeBoost.MH, that is explicitly designed to work on tree-structured sets of categories, and is capable of leveraging on the information inherent in this structure. TreeBoost.MH is fully illustrated in Figure 1.

Let us first fix some notation and definitions. Let H be a tree-structured set of categories, let r be its root category, and let $L = \langle\langle l_1, Tr^+(l_1)\rangle, \ldots, \langle l_m, Tr^+(l_m)\rangle\rangle\rangle$ be the set of leaf categories of H together with their sets of positive training examples. For each category $c_j \in H$, we will use the following abbreviations:

Symbol	Meaning
$\uparrow(c_j)$	the parent category of c_j
$\downarrow(c_j)$	the set of children categories of c_j
$\Uparrow(c_j)$	the set of ancestor categories of c_j
$\Downarrow(c_j)$	the set of descendant categories of c_j
$\leftrightarrow(c_j)$	the set of sibling categories of c_j

We assume that documents can belong to zero, one, or several leaf categories in L, and that leaf categories are the only categories to which documents can belong (so that categories corresponding to internal nodes are just aggregations of "real" categories). The notion of set of positive/negative training examples is naturally extended to nonleaf categories via the following definition.

Definition 1. *Given a nonleaf category* c_j*, its set of positive training examples* $Tr^+(c_j)$ *is defined as* $Tr^+(c_j) = \bigcup_{c \in \Downarrow(c_j)} Tr^+(c)$*, i.e. as the union of the sets of positive training examples of all its descendant (leaf) categories.*

Input: A triplet $\langle H, r, L\rangle$ where
  H is a tree-structured set of categories,
  r is the root category of H,
  $L = \langle\langle l_1, Tr^+(l_1)\rangle, \ldots, \langle l_m, Tr^+(l_m)\rangle\rangle$ is the (possibly empty) set of leaf categories of H together with their sets of positive training examples;
Body: **if** r is a leaf category **then** do nothing
else begin
  let $\downarrow(r) = \{\langle\downarrow_1(r), Tr^+(\downarrow_1(r))\rangle \ldots, \langle\downarrow_{k(r)}(r), Tr^+(\downarrow_{k(r)}(r))\rangle\}$ be the $k(r)$ children categories of r together with their sets of positive training examples;
  run a ML feature selection algorithm on $\downarrow(r)$;
  run AdaBoost.MH on $\downarrow(r)$;
  for $q = 1, \ldots, k(r)$ **do**
    begin
      let T_q be the subtree of H rooted at $\downarrow_q(r)$;
      let $L_q = \{\langle l_{q(1)}, Tr^+(l_{q(1)})\rangle, \ldots, \langle l_{q(z)}, Tr^+(l_{q(z)})\rangle\}$ be the (possibly empty) set of leaf categories of T_q together with their sets of positive training examples;
      run TreeBoost.MH on $\langle T_q, \downarrow_q(r), L_q\rangle$;
    end
end
Output: For each nonleaf category $c_t \in H$, a final hypothesis $\hat{\Phi}^{(t)}(d, c) = \sum_{s=1}^{S} \hat{\Phi}_s^{(t)}(d, c)$ for $c \in \downarrow(c_t)$

Fig. 1. The TreeBoost.MH algorithm

Definition 2. *Given a nonleaf category c_j, its set of negative training examples $Tr^-(c_j)$ is defined as $Tr^-(c_j) = \left(\bigcup_{c\in\leftrightarrow(c_j)} Tr^+(c)\right) - Tr^+(c_j)$, i.e. as the union of the sets of positive training examples of all its sibling (leaf or nonleaf) categories, minus its own positive training examples.*

3.1 The Rationale

TreeBoost.MH embodies several intuitions that had arisen before within HTC.

The first, fairly obvious intuition (which lies at the basis of practically all HTC algorithms proposed in the literature) is that, in a hierarchical context, the classification of a document d_i is to be seen as a descent through the hierarchy, from the root to the leaf categories where d_i is deemed to belong. In ML classification, this means that each nonroot category c_j has an associated binary classifier $\hat{\Phi}_j$ which acts as a "filter" that prevents unsuitable documents to percolate to lower levels. All test documents that a classifier $\hat{\Phi}_j$ deems to belong to c_j are passed as input to all the binary classifiers corresponding to the categories in $\downarrow(c_j)$, while the documents that $\hat{\Phi}_j$ deems not to belong to c_j are "blocked" and analysed no further. Each document may thus reach zero, one, or several leaf categories, and is thus classified under them.

The second intuition is that the training of $\hat{\Phi}_j$ should be performed "locally", i.e. by paying attention to the topology of the classification scheme. To see this, note that, during classification, if the classifier for $\uparrow(c_j)$ has performed correctly, $\hat{\Phi}_j$ will only (or mostly) be presented with documents that belong to the subtree rooted in $\uparrow(c_j)$, i.e. with documents that belong to c_j and/or to some of the categories in $\leftrightarrow(c_j)$. As a result, the training of $\hat{\Phi}_j$ should be performed by using, as negative training examples, the union of the positive training examples of the categories in $\leftrightarrow(c_j)$ (with the obvious exception of the

documents that are also positive training examples of c_j); in particular, training documents that only belong to leaf categories other than those in $\Downarrow(c_j)$ need not be used. The rationale of this choice is that the chosen documents are "quasi-positive" examples of c_j [14], i.e. are the negative examples that are closest to the boundary between the positive and the negative region of c_j (a notion akin to that of "support vectors" in SVMs), and are thus the most informative negative examples that can be used in training. This is beneficial also from the standpoint of (both training and classification time) efficiency, since fewer training examples and fewer features are involved. This intuition lies at the basis of Definition 2 above; in a similar form, it had first been presented in [15].

The third intuition is similar, i.e. that feature selection should also be performed "locally", by paying attention to the topology of the classification scheme. As above, if the classifier for $\uparrow(c_j)$ has performed correctly, $\hat{\Phi}_j$ will only (or mostly) be presented with documents that belong to the subtree rooted in $\uparrow(c_j)$. As a consequence, for the classifiers corresponding to c_j and its siblings, it is cost-effective to employ features that are useful in discriminating among them, and only among them; features that discriminate among categories lying outside the subtree rooted in $\uparrow(c_j)$ are too general, and features that discriminate among the subcategories of c_j, or among the subcategories of one of its siblings, are too specific. This intuition, albeit in a different form, was first presented in [2].

TreeBoost.MH also embodies the novel intuition that the weight distribution that boosting algorithms update at every boosting round should likewise be updated "locally". In fact, the two previously discussed intuitions indicate that hierarchical ML classification is best understood as consisting of several independent (flat) ML classification problems, one for each internal node of the hierarchy: for each such node c_j we must generate a number of binary classifiers, one for each $c_q \in \downarrow(c_j)$. In a boosting context, this means that several independent distributions, each one "local" to an internal node and its children, should be generated and updated by the process. In this way, the "difficulty" of a category c_q will only matter *relative* to the difficulty of its sibling categories.

3.2 The Algorithm

TreeBoost.MH incorporates these four intuitions by factoring the hierarchical ML classification problem into several "flat" ML classification problems, one for every internal node in the tree. TreeBoost.MH learns in a recursive fashion, by identifying internal nodes c_j and calling AdaBoost.MH to generate a ML (flat) classifier for the set of categories $\downarrow(c_j)$. Alternatively (and more conveniently), this process may be viewed as generating, for each nonroot category $c_j \in H$, a binary classifier $\hat{\Phi}$ for c_j, by means of which hierarchical classification can be performed as described in Section 3.1.

Learning in TreeBoost.MH proceeds by first identifying whether a leaf category has been reached (line 6 of Figure 1), in which case nothing is done, since the classifiers are generated only at internal nodes.

If an internal node c_j has been reached, a ML feature selection process may (optionally) be run (line 10) to generate a reduced feature set on which the ML classifier for $\downarrow(c_j)$ will operate. This may be dubbed a "glocal" feature selection

policy, since it takes an intermediate stand between the well-known "global" policy (in which the same set of features is selected for all the categories in H) and "local" policy (in which a different set of features is chosen for each different category). The glocal policy selects a different set of features for each maximal set of sibling categories in H, thus implementing a view of feature selection as described in Section 3.1[2]. Any of the standard feature scoring functions (e.g. information gain, chi-square) can be used, as well as any of the standard feature score globalization methods (e.g. max, weighted average, Forman's [16] round robin). Note that all these functions require a precise notion of what the positive and negative training examples of a category are; this notion is well-defined for leaf categories (see beginning of Section 2), and is catered for by Definitions 1 and 2 for internal node categories.

After the reduced feature set has been identified, TreeBoost.MH calls upon AdaBoost.MH (line 11) to solve a ML (flat) classification problem for the categories in $\downarrow(c_j)$; here too, what counts as a positive and as a negative training example of a category comes from Definitions 1 and 2, which implements the "quasi-positive" policy for the choice of negative training examples discussed in Section 3.1. Note that restricting the AdaBoost.MH call to the categories in $\downarrow(c_j)$ implements the view, discussed in Section 3.1, of several independent, "local" distributions being generated and updated during the boosting process.

Finally, after the ML classifier for $\downarrow(c_j)$ has been generated, for each category $c_q \in\downarrow(c_j)$ a recursive call to TreeBoost.MH is issued (lines 12–18) that processes the subtree rooted in c_q in the same way. The final result is a hierarchical ML classifier in the form of a tree of binary classifiers, one for each nonroot node, each consisting of a committee of S decision stumps.

In an extended version of this paper [17] we discuss the computational cost of TreeBoost.MH, proving that (at least in the idealized case of a "fully grown", perfectly balanced tree of constant ariety a):

- at training time TreeBoost.MH is $O(grah)$, while AdaBoost.MH is $O(grm)$;
- at testing time TreeBoost.MH is $O(Sah)$, while AdaBoost.MH is $O(Sm)$.

Since $m = a^h$, this means that TreeBoost.MH is cheaper than AdaBoost.MH by an exponential factor, both at training time and at testing time.

4 Experiments

4.1 Experimental Setting

The first benchmark we have used in our experiments is the "Reuters-21578, Distribution 1.0" corpus[3]. In origin, the Reuters-21578 category set is not

[2] Note that a local policy would also implement this view, but is not made possible by AdaBoost.MH, since this latter uses the same set of features for all the categories involved in the ML classification problem. This means that we need to use the same set of features for all categories in $\downarrow(c_j)$.

[3] `http://www.daviddlewis.com/resources/testcollections/~reuters21578/`

hierarchically structured, and is thus not suitable "as is" for HTC experiments; we have thus used a hierarchical version of it generated in [5] by the application of hierarchical agglomerative clustering on the 90 REUTERS-21578 categories that have at least one positive training example and one positive test example. The original REUTERS-21578 categories are thus "leaf" categories in the resulting hierarchy, and are clustered into four "macro-categories" whose parent category is the root of the tree. Conforming to the experiments of [5], we have used (according to the ModApte split) the 7,770 training examples and 3,299 test examples that are labelled by at least one of the selected categories.

The second benchmark we have used is REUTERS CORPUS VOLUME 1 version 2 (RCV1-v2)[4], consisting of 804,414 news stories. In our experiments we have used the "LYRL2004" split defined in [18], in which the (chronologically) first 23,149 documents are used for training and the other 781,265 are used for testing. Out of the 103 "Topic" categories, in our experiments we have restricted our attention to the 101 categories with at least one positive training example. The RCV1-v2 hierarchy is four levels deep (including the root, to which we conventionally assign level 0); there are four internal nodes at level 1, and the leaves are all at the levels 2 and 3.

In all the experiments discussed in this section, punctuation has been removed, all letters have been converted to lowercase, numbers have been removed, stop words have been removed, and stemming has been performed by means of Porter's stemmer. As a measure of effectiveness that combines the contributions of *precision* (π) and *recall* (ρ) we have used the well-known F_1 function, defined as $F_1 = \frac{2\pi\rho}{\pi+\rho} = \frac{2TP}{2TP+FP+FN}$, where TP, FP, and FN stand for the numbers of true positives, false positives, and false negatives, respectively. We compute both microaveraged F_1 (denoted by F_1^μ) and macroaveraged F_1 (F_1^M). F_1^μ is obtained by (i) computing the category-specific values TP_i, (ii) obtaining TP as the sum of the TP_i's (same for FP and FN), and then (iii) applying the $F_1 = \frac{2\pi\rho}{\pi+\rho}$ formula. F_1^M is obtained by first computing the category-specific F_1 values and then averaging them across the c_i's.

4.2 Results

The results of our experiments are reported in Table 1.

In a first experiment we have compared ADABOOST.MH and TREEBOOST.MH using a full feature set. We have then switched to reduced feature sets, obtained according to a "global" feature selection policy in which (i) feature-category pairs have been scored by means of *information gain*, defined as $IG(t_k, c_i) = \sum_{c\in\{c_i,\overline{c}_i\}} \sum_{t\in\{t_k,\overline{t}_k\}} P(t,c) \cdot \log \frac{P(t,c)}{P(t)\cdot P(c)}$ and (ii) the final set of features has been chosen according to Forman's *round robin* technique, which consists in picking, for each category c_i, the v features with the highest $IG(t_k, c_i)$ value, and pooling all of them together into a category-independent set [16]. This set thus contains a number of features $q \leq vm$, where m is the number of categories; it usually contains strictly fewer than them, since some features are among the best v features for more than one category. We have set v to 60 for REUTERS-21578 and to 43 for

[4] `http://trec.nist.gov/data/reuters/reuters.html`

Table 1. AdaBoost.MH and TreeBoost.MH on Reuters-21578 (top 5 rows) and RCV1-v2 (bottom 5 rows). In each square, the first figure from top is F_1^μ, the second is F_1^M, the third is training time (inclusive of the time required to perform feature selection, if any), and the fourth is testing time.

	5 iterations	10 iterations	20 iterations	50 iterations	100 iterations	200 iterations	500 iterations	1000 iterations
AdaBoost.MH (Full)	.533 .033 34.0 11.1	.597 .075 68.1 14.3	.664 .160 136.3 18.2	.724 .255 340.7 35.1	.783 .332 681.5 66.2	.798 .361 1362.9 129.9	.804 .377 3407.3 274.0	.808 .379 6814.6 464.3
TreeBoost.MH (Full)	.596 (+11.9%) .100 (+197.4%) 16.9 (-50.4%) 13.0 (+16.8%)	.699 (+17.2%) .187 (+148.4%) 33.8 (-50.4%) 12.6 (-11.8%)	.745 (+12.2%) .286 (+78.9%) 67.6 (-50.4%) 17.2 (-5.5%)	.795 (+9.9%) .416 (+62.6%) 169.0 (-50.4%) 20.6 (-41.4%)	.810 (+3.5%) .425 (+28.2%) 337.9 (-50.4%) 29.9 (-54.9%)	.827 (+3.7%) .454 (+25.6%) 675.8 (-50.4%) 48.5 (-62.7%)	.830 (+3.1%) .460 (+22.0%) 1689.4 (-50.4%) 96.6 (-64.7%)	.826 (+2.3%) .479 (+26.4%) 3378.9 (-50.4%) 151.3 (-67.4%)
AdaBoost.MH (Global)	.533 .034 24.6 8.8	.597 .075 49.2 12.4	.664 .160 98.4 17.0	.724 .256 246.1 32.8	.783 .332 492.1 59.6	.799 .354 984.3 112.2	.811 .373 2460.8 255.2	.801 .362 4921.5 386.0
TreeBoost.MH (Global)	.596 (+11.9%) .100 (+197.4%) 11.5 (-53.4%) 9.1 (+3.2%)	.699 (+17.2%) .187 (+148.4%) 23.0 (-53.4%) 9.6 (-22.1%)	.744 (+11.9%) .285 (+78%) 45.9 (-53.4%) 11.3 (-33.5%)	.800 (+10.5%) .437 (+70.8%) 114.7 (-53.4%) 17.2 (-47.7%)	.809 (+3.4%) .427 (+28.7%) 229.5 (-53.4%) 26.3 (-55.9%)	.815 (+2.0%) .457 (+28.8%) 459.0 (-53.4%) 42.4 (-62.2%)	.828 (+2.1%) .457 (+22.4%) 1147.4 (-53.4%) 82.3 (-67.7%)	.821 (+2.5%) .473 (+30.6%) 2294.7 (-53.4%) 131.3 (-66.0%)
TreeBoost.MH (Glocal)	.596 (+11.9%) .100 (+197.4%) 12.1 (-50.7%) 10.7 (+21.6%)	.699 (+17.2%) .187 (+148.4%) 24.2 (-50.7%) 14.9 (+20.9%)	.744 (+11.9%) .285 (+77.9%) 48.5 (-50.7%) 14.1 (-16.7%)	.794 (+9.7%) .401 (+56.9%) 121.2 (-50.7%) 23.0 (-29.8%)	.812 (+3.8%) .430 (+29.8%) 242.5 (-50.7%) 28.3 (-52.4%)	.817 (+2.2%) .460 (+29.8%) 485.0 (-50.7%) 46.2 (-58.9%)	.824 (+1.6%) .465 (+24.7%) 1212.4 (-50.7%) 94.6 (-62.9%)	.825 (+3.0%) .465 (+28.3%) 2424.8 (-50.7%) 142.9 (-63.0%)
AdaBoost.MH (Full)	.361 .037 181.3 3346.2	.406 .070 362.7 3576.4	.479 .131 725.3 5168.2	.587 .239 1813.3 9524.7	.650 .333 3626.5 16827.2	.701 .396 7253.1 33608.9	.735 .435 18132.7 83951.2	.745 .442 36265.5 170720.3
TreeBoost.MH (Full)	.391 (+8.4%) .097 (+159.0%) 78.3 (-56.8%) 2774.5 (-17.1%)	.460 (+13.3%) .128 (+81.5%) 156.5 (-56.8%) 2813.5 (-21.3%)	.543 (+13.5%) .211 (+60.8%) 313.1 (-56.8%) 3081.1 (-40.4%)	.658 (+12.1%) .341 (+42.8%) 782.7 (-56.8%) 3963.2 (-58.4%)	.705 (+8.4%) .409 (+22.7%) 1565.3 (-56.8%) 6044.2 (-64.1%)	.734 (+4.6%) .447 (+12.9%) 3130.6 (-56.8%) 9328.3 (-72.2%)	.752 (+2.3%) .476 (+9.4%) 7826.6 (-56.8%) 21847.4 (-74.0%)	.761 (+2.1%) .486 (+9.8%) 15653.1 (-56.8%) 38342.9 (-77.5%)
AdaBoost.MH (Global)	.361 .037 107.8 1598.4	.406 .070 215.5 2223.7	.479 .131 431.1 3741.0	.587 .239 1077.7 8012.8	.650 .332 2155.5 16206.9	.700 .398 4310.9 31907.7	.736 .443 10777.3 74292.3	.749 .457 21554.7 147354.1
TreeBoost.MH (Global)	.391 (+8.4%) .097 (+159.0%) 33.8 (-68.7%) 1830.2 (+14.5%)	.460 (+13.3%) .128 (+81.6%) 67.5 (-68.7%) 1422.0 (-36.1%)	.545 (+13.8%) .213 (+62.6%) 135.1 (-68.7%) 1901.0 (-49.2%)	.657 (+12.0%) .340 (+42.4%) 337.6 (-68.7%) 2772.1 (-65.4%)	.702 (+8.0%) .409 (+23.4%) 675.3 (-68.7%) 4836.0 (-70.2%)	.732 (+4.6%) .448 (+12.5%) 1350.5 (-68.7%) 8156.6 (-74.4%)	.753 (+2.3%) .484 (+9.4%) 3376.4 (-68.7%) 18384.5 (-75.3%)	.760 (+1.4%) .495 (+8.3%) 6752.8 (-68.7%) 33648.3 (-77.2%)
TreeBoost.MH (Glocal)	.391 (+8.4%) .097 (+159.0%) 41.3 (-61.7%) 2374.9 (+48.6%)	.460 (+13.3%) .128 (+81.5%) 82.6 (-61.7%) 2432.7 (+9.4%)	.543 (+13.5%) .211 (+61.1%) 165.3 (-61.7%) 2499.9 (-33.2%)	.658 (+12.1%) .340 (+42.6%) 413.1 (-61.7%) 3645.9 (-54.5%)	.703 (+8.1%) .408 (+23.0%) 826.3 (-61.7%) 5020.3 (-69.0%)	.735 (+5.1%) .450 (+12.9%) 1652.6 (-61.7%) 8372.9 (-73.8%)	.753 (+2.3%) .476 (+7.6%) 4131.4 (-61.7%) 18173.9 (-75.5%)	.762 (+1.7%) .490 (+7.2%) 8268.9 (-61.7%) 33149.0 (-77.5%)

RCV1-v2, which are the values that, for each corpora, best approximate a total number of features of 2,000; in fact, the reduced feature sets consist of 2,012 features for Reuters-21578 (11% of the 18,177 original ones) and 2,029 for RCV1-v2 (3.7% of the 55,051 original ones).

We have also run an experiment in which we have used the "glocal" feature selection policy described in Section 3.2, consisting in selecting a different feature subset (of the same cardinalities as in the global policy) for the set of children of each different internal node. Note that the results obtained by means of this policy are reported only for TreeBoost.MH, since this policy obviously is not applicable to AdaBoost.MH.

We will now comment on the Reuters-21578 results[5]; the RCV1-v2 are qualitatively similar. The first observation we can make is that, in switching from AdaBoost.MH to TreeBoost.MH, effectiveness improves substantially. F_1^μ improves from +2.3% to +17.2%, depending on the number S of boosting iterations. F_1^M improves even more substantially, from +22.0% to +197.4%; this

[5] The reader might notice that the best performance we have obtained from AdaBoost.MH on Reuters-21578 ($F_1^\mu = .808$) is inferior to the one reported in [12] for the same algorithm ($F_1^\mu = .851$). There are several reasons for this: (a) [12] actually uses a different, much older version of this collection, called Reuters-21450 [19]; (b) [12] only uses the 93 categories which have at least 2 positive training examples and 1 positive test example, while we also use the categories that have just 1 positive training example and those that have no positive test example. This makes the two sets of AdaBoost.MH results difficult to compare.

means that TreeBoost.MH is especially suited to categorization problems in which the distribution of training examples across the categories is highly skewed. For both F_1^μ and F_1^M, the improvements tend to be more substantial for low values of S, showing that TreeBoost.MH converges to optimum performance more rapidly than AdaBoost.MH. Altogether, these effectiveness improvements are somehow surprising, since it is well-known that hierarchical TC can introduce a deterioration of effectiveness due to classification errors made high up in the hierarchy, which cannot be recovered anymore [2, 4]. The improvements thus show that the "filters" placed at the internal nodes work well, likely due to the fact that they their training benefits from using only the "quasi-positive" examples of local interest as negative training examples.

In terms of efficiency, we can observe that training time is +50.4% smaller, irrespectively of the number of iterations, a reduction that confirms the theoretical findings discussed in Section 3.2 (and that might likely be even more substantial in classification problems characterized by a deeper, more articulated hierarchy). Classification time is also generally reduced; aside from an isolated case in which it increases by 16.8%, it is reduced from +5.5% to +67.4%, with higher reductions being obtained for high values of S; this is likely due to the fact that, since high values of S bring about more effective classifiers, the classifiers placed at internal nodes are more effective at "blocking" unsuitable documents from percolating down to leaves which would reject them anyway.

The experiments run after global feature selection qualitatively confirm the results above. Note that the effectiveness values are practically unchanged wrt the full feature set experiment; this is especially noteworthy for the RCV1-v2 experiments, in which more than 96% of the original features have been discarded with no loss in effectiveness. Effectiveness does not change also when using "glocal" feature selection. This is somehow surprising, since an effectiveness improvement might have been expected here, due to the generation of feature sets customized to each internal node. It is thus likely that the values of v chosen when applying the global policy were large enough to allow the inclusion, for each internal node, of enough features customized to it.

5 Related Work

HTC was first tackled in [9], in the context of a TC system based on neural networks. The intuition that it could be useful to perform feature selection locally by exploiting the topology of the tree is originally due to [2]. However, this work dealt with 1-of-n text categorization, which means that feature selection was performed relative to the set of children of each internal node; given that we are in a m-of-n classification context, we instead do it relative to each individual child of any internal node. The intuition that the negative training examples for training the classifier for category c_j could be limited to the positive training examples of categories close to c_j in the tree is due to [15]. The notion that, in a m-of-n classification context, classifiers at internal nodes act as "filters" informs much of the HTC literature, and is explicitly discussed at least in [7], which proposes a HTC system based on neural networks.

Other works in HTC focus on other specific aspects of the learning task. For instance, the "shrinkage" method presented in [4] attempts to improve parameter estimation for data-sparse leaf categories in a 1-of-n HTC system based on a naïve Bayesian method. Incidentally, the naïve Bayesian approach seems to have been the most popular among HTC researchers, since several other HTC models are hierarchical variations of naïve Bayesian learning algorithms [1, 3, 5, 6].

6 Conclusion

We have presented TreeBoost.MH, a recursive algorithm for hierarchical text categorization that uses AdaBoost.MH as its base step and that recurs over the category tree structure. We have given complexity results in which we show that TreeBoost.MH, by leveraging on the hierarchical structure of the category tree, is exponentially cheaper to train and to test than AdaBoost.MH. These theoretical intuitions have been confirmed by thorough empirical testing on two standard benchmarks, on which TreeBoost.MH has brought about substantial savings at both learning time and classification time. TreeBoost.MH has also shown to bring about substantial improvements in effectiveness wrt AdaBoost.MH, especially in terms of macroaveraged effectiveness; this feature makes it extremely suitable to categorization problems characterized by a skewed distribution of the positive training examples across the categories.

References

1. Chakrabarti, S., Dom, B.E., Agrawal, R., Raghavan, P.: Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. Journal of Very Large Data Bases **7**(3) (1998) 163–178
2. Koller, D., Sahami, M.: Hierarchically classifying documents using very few words. In: Proceedings of the 14th International Conference on Machine Learning (ICML'97), Nashville, US (1997) 170–178
3. Gaussier, É., Goutte, C., Popat, K., Chen, F.: A hierarchical model for clustering and categorising documents. In: Proceedings of the 24th European Colloquium on Information Retrieval Research (ECIR'02), Glasgow, UK (2002) 229–247
4. McCallum, A.K., Rosenfeld, R., Mitchell, T.M., Ng, A.Y.: Improving text classification by shrinkage in a hierarchy of classes. In: Proceedings of the 15th International Conference on Machine Learning (ICML'98), Madison, US (1998) 359–367
5. Toutanova, K., Chen, F., Popat, K., Hofmann, T.: Text classification in a hierarchical mixture model for small training sets. In: Proceedings of the 10th ACM International Conference on Information and Knowledge Management (CIKM'01), Atlanta, US (2001) 105–113
6. Vinokourov, A., Girolami, M.: A probabilistic framework for the hierarchic organisation and classification of document collections. Journal of Intelligent Information Systems **18**(2/3) (2002) 153–172
7. Ruiz, M., Srinivasan, P.: Hierarchical text classification using neural networks. Information Retrieval **5**(1) (2002) 87–118
8. Weigend, A.S., Wiener, E.D., Pedersen, J.O.: Exploiting hierarchy in text categorization. Information Retrieval **1**(3) (1999) 193–216

9. Wiener, E.D., Pedersen, J.O., Weigend, A.S.: A neural network approach to topic spotting. In: Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval (SDAIR'95), Las Vegas, US (1995) 317–332
10. Dumais, S.T., Chen, H.: Hierarchical classification of web content. In: Proceedings of the 23rd ACM International Conference on Research and Development in Information Retrieval (SIGIR'00), Athens, GR (2000) 256–263
11. Yang, Y., Zhang, J., Kisiel, B.: A scalability analysis of classifiers in text categorization. In: Proceedings of the 26th ACM International Conference on Research and Development in Information Retrieval (SIGIR'03), Toronto, CA (2003) 96–103
12. Schapire, R.E., Singer, Y.: BoosTexter: a boosting-based system for text categorization. Machine Learning **39**(2/3) (2000) 135–168
13. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. Machine Learning **37**(3) (1999) 297–336
14. Schapire, R.E., Singer, Y., Singhal, A.: Boosting and Rocchio applied to text filtering. In: Proceedings of the 21st ACM International Conference on Research and Development in Information Retrieval (SIGIR'98), Melbourne, AU (1998) 215–223
15. Ng, H.T., Goh, W.B., Low, K.L.: Feature selection, perceptron learning, and a usability case study for text categorization. In: Proceedings of the 20th ACM International Conference on Research and Development in Information Retrieval (SIGIR'97), Philadelphia, US (1997) 67–73
16. Forman, G.: A pitfall and solution in multi-class feature selection for text classification. In: Proceedings of the 21st International Conference on Machine Learning (ICML'04), Banff, CA (2004)
17. Esuli, A., Fagni, T., Sebastiani, F.: TreeBoost.MH: A boosting algorithm for multi-label hierarchical text categorization. Technical Report 2006-TR-56, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, IT (2006) Submitted for publication.
18. Lewis, D.D., Li, F., Rose, T., Yang, Y.: RCV1: A new benchmark collection for text categorization research. Journal of Machine Learning Research **5** (2004) 361–397
19. Apté, C., Damerau, F.J., Weiss, S.M.: Automated learning of decision rules for text categorization. ACM Transactions on Information Systems **12**(3) (1994) 233–251

Cluster Generation and Cluster Labelling for Web Snippets

Filippo Geraci[1,2], Marco Pellegrini[1], Marco Maggini[2], and Fabrizio Sebastiani[3]

[1] Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Via G Moruzzi 1, 56124 Pisa, Italy
{f.geraci, m.pellegrini}@iit.cnr.it
[2] Dipartimento di Ingegneria dell'Informazione, Università di Siena, Via Roma 56, 53100 Siena, Italy
maggini@ing.unisi.it
[3] Istituto di Scienza e Tecnologia dell'Informazione, Consiglio Nazionale delle Ricerche, Via G Moruzzi 1, 56124 Pisa, Italy
fabrizio.sebastiani@isti.cnr.it

Abstract. This paper describes Armil, a meta-search engine that groups into disjoint labelled clusters the Web snippets returned by auxiliary search engines. The cluster labels generated by Armil provide the user with a compact guide to assessing the relevance of each cluster to her information need. Striking the right balance between running time and cluster well-formedness was a key point in the design of our system. Both the clustering and the labelling tasks are performed on the fly by processing only the snippets provided by the auxiliary search engines, and use no external sources of knowledge. Clustering is performed by means of a fast version of the furthest-point-first algorithm for metric k-center clustering. Cluster labelling is achieved by combining intra-cluster and inter-cluster term extraction based on a variant of the information gain measure. We have tested the clustering effectiveness of Armil against Vivisimo, the *de facto* industrial standard in Web snippet clustering, using as benchmark a comprehensive set of snippets obtained from the Open Directory Project hierarchy. According to two widely accepted "external" metrics of clustering quality, Armil achieves better performance levels by 10%. We also report the results of a thorough user evaluation of both the clustering and the cluster labelling algorithms.

1 Introduction

An effective search interface is a fundamental component in a Web search engine. In particular, the quality of presentation of the search results often represents one of the main keys to the success of such systems. Most search engines present the results of a user query as a ranked list of Web snippets. *Meta-search engines* (MSEs) integrate the items obtained from multiple "auxiliary" search engines, with the purpose of increasing the coverage of the results. However, without an accurate design, MSEs might in principle even worsen the quality of the information access experience, since the user is typically confronted with an even larger set of results. Thus, key issues to be faced by MSEs concern the exploitation of effective algorithms for merging the ranked lists of results (while at the same time removing the duplicates), and the design of advanced user interfaces based on a structured organization of the results. This latter aspect is

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 25–36, 2006.

usually implemented by grouping the results into homogeneous groups by means of clustering or categorization algorithms.

This paper describes the Armil system[1], a meta-search engine that organizes the Web snippets retrieved from auxiliary search engines into disjoint clusters and automatically constructs a title label for each cluster by using only the text excerpts available in the snippets. Our design efforts were directed towards devising a fast clustering algorithm able to yield good-quality homogeneous groups, and a distillation technique for selecting appropriate and useful labels for the clusters. The speed of the two algorithms was a key issue in our design, since the system must organize the results on the fly, thus minimizing the latency between the issuing of the query and the presentation of the results. Second-level clustering is also performed at query time (i.e. not on demand) to minimize latency. In Armil, an equally important role is played by the clustering component and by the labelling component. Clustering is accomplished by means of an improved version of the furthest-point-first (FPF) algorithm for k-center clustering [1]. To the best of our knowledge this algorithm had never been used in the context of Web snippet clustering or text clustering. The generation of the cluster labels is instead accomplished by means of a combination of intra-cluster and inter-cluster term extraction, based on a modified version of the information gain measure. This approach tries to capture the most significant and discriminative words for each cluster.

One key design feature of Armil is that it relies only on the information returned by the auxiliary search engines, i.e. the snippets; this means that no external source of information, such as ontologies or lexical resources, is used. We thus demonstrate that such a lightweight approach, together with carefully crafted algorithms, is sufficient to provide a useful and successful clustering-plus-labelling service. Obviously, this assumption relies on the hypothesis that the quality of the results and of the snippets returned by the auxiliary search engines is satisfactory. We have tested the clustering effectiveness of Armil against Vivisimo, the *de facto* industrial standard in Web snippet clustering, using as benchmark a comprehensive set of snippets obtained from the Open Directory Project hierarchy. According to two metrics of clustering quality that are normalized variants of the Entropy and the Mutual Information [2], Armil achieves better performance levels by 10%. Note that, since the normalization reduces the ranges of these measures in the interval $[0, 1]$, an increase of 10% is noteworthy. We also report the results of a thorough user evaluation of both the clustering and the cluster labelling algorithms.

Outline of the clustering algorithm. Clustering and labelling are both essential operations for a Web snippet clustering system. However, each previously proposed such system strikes a different balance between the two aspects. Some systems (e.g. [3, 4]) view label extraction as the primary goal, and clustering is a by-product of the label extraction procedure. Other systems (e.g. [5, 6]) view instead the formation of clusters as the most important step, and the labelling phase is considered as strictly dependent on the clusters found. We have followed this latter approach. In order to cluster the snippets in the returned lists, we map them into a vector space endowed with a distance function, which we treat as a metric; then a modified furthest-point-first algorithm (M-FPF) is applied to generate the clusters. The M-FPF algorithm generates the same clusters of the "standard" FPF algorithm, but uses filters based on the triangular inequality to speed up the

[1] The Armil system can be freely accessed at `http://armil.iit.cnr.it/`.

computation. As such, M-FPF inherits a very important property of the FPF algorithm, i.e. it is within a factor 2 of the optimal solution for the k-center problem [7]. The second interesting property of M-FPF is that it does not compute centroids of clusters. Centroids tend to be dense vectors and, as such, their computation and/or update in high-dimensional space is a computational burden. M-FPF relies instead only on pairwise distance calculations between snippets, and as such better exploits the sparsity of the snippet vector representations.

Outline of the cluster labelling algorithm. The cluster labelling phase aims at extracting from the set of snippets assigned to each cluster a sequence of words highly descriptive of the corresponding group of items. The quality of the label depends on its well-formedness (i.e. whether the text is syntactically and semantically plausible), on its descriptive power (i.e. how well it describes what is contained in the cluster), and on its discriminative power (i.e. how well it differentiates what is contained in the cluster with respect to what is contained in other clusters). The possibility to extract good labels directly from the available snippets is strongly dependent on their quality and, obviously, on the homogeneity of the produced clusters. In order to pursue a good tradeoff between descriptive and discriminative power, we select candidate words for each cluster by means of IG_m, a modified version of the *Information Gain* measure [2]. For each cluster, IG_m allows the selection of those words that are most representative of its contents and are least representative of the contents of the other clusters. Finally, in order to construct plausible labels, rather than simply using the list of the top-scoring words (i.e. the ones that maximize IG_m), the system looks within the titles of the returned Web pages for the substring that best matches the selected top-scoring words.

Once each cluster has been assigned a set of descriptive and discriminative words (we call such set the cluster *signatures*), all the clusters that share the same signature are merged. This reduces the arbitrariness inherent in the choice of their number k, that is fixed *a priori* independently of the query.

Outline of the paper. The paper is organized as follows. In Section 2 we review related work on techniques for the automatic re-organization of search results. Section 3 introduces the data representation adopted within Armil and sketches the properties of the M-FPF clustering algorithm and of the cluster labelling algorithm. The results of the system evaluation are reported in Sections 5 and 4. Finally, in Section 6 conclusions and prospective future research are discussed. A full version of this paper with more details is in [8].

2 Previous Work

Tools for clustering Web snippets have recently become a focus of attention in the research community. In the past, this approach has had both critics [9, 10] and supporters [11], but the proliferation of commercial Web services such as Copernic, Dogpile, Groxis, iBoogie, Kartoo, Mooter, and Vivisimo seems to confirm the validity of the approach. Academic research prototypes are also available, such as Grouper [12, 6], EigenCluster [13], Shoc [14], and SnakeT [3]. Generally, details of the algorithms underlying the commercial Web services are not in the public domain.

Maarek et al. [15] give a precise characterization of the challenges inherent in Web snippet clustering, and propose an algorithm based on complete-link

hierarchical agglomerative clustering that is quadratic in the number n of snippets. They introduce a technique called "lexical affinity" whereby the co-occurrence of words influences the similarity metric.

Zeng et al. [16] tackle the problem of detecting good cluster names as preliminary to the formation of the clusters, using a supervised learning approach. Note that the methods considered in our paper are instead all unsupervised, thus requiring no labelled data.

The EigenCluster [13], Lingo [17], and Shoc [14] systems all tackle Web snippet clustering by performing a singular value decomposition of the term-document incidence matrix[2]; the problem with this approach is that SVD is extremely time-consuming, hence problematic when applied to a large number of snippets. Zamir and Etzioni [12, 6] propose a Web snippet clustering mechanism (Suffix Tree Clustering – STC) based on suffix arrays, and experimentally compare STC with algorithms such as k-means, single-pass k-means [18], Backshot and Fractionation [19], and Group Average Hierarchical Agglomerative Clustering. They test the systems on a benchmark obtained by issuing 10 queries to the Metacrawler meta-search engine, retaining the top-ranked 200 snippets for each query, and manually tagging the snippets by relevance to the queries. They then compute the quality of the clustering obtained by the tested systems by ordering the generated clusters according to precision, and by equating the effectiveness of the system with the average precision of the highest-precision clusters that collectively contain 10% of the input documents. Interestingly, the authors show that very similar results are attained when full documents are used instead of their snippets, thus validating the snippet-based clustering approach.

Lawrie and Croft [4] view the clustering/labelling problem as that of generating multilevel summaries of the set of documents (in this case the Web snippets returned by a search engine). The technique is based on first building off-line a statistical model of the background language (e.g. the statistical distribution of words in a large corpus of the English language), and on subsequently extracting "topical terms" from the documents, where "topicality" is measured by the contribution of a term to the Kullback-Leibler divergence score of the document collection relative to the background language. Intuitively, this formula measures how important this term is in measuring the distance of the collection of documents from the distribution of the background language. The proposed method is shown to be superior (by using the KL-divergence) to a naive summarizer that just selects the terms with highest $tf * idf$ score in the document set.

Kammamuru et al. [5] propose a classification of Web snippet clustering algorithms into *monothetic* (in which the assignment of a snippet to a cluster is based on a single dominant feature) and *polythetic* (in which several features concur in determining the assignment of a snippet to a cluster). The rationale for proposing a monothetic algorithm is that the single discriminating feature is a natural label candidate. The authors propose such an algorithm in which the snippets are seen as sets of words and the next term is chosen so as to maximize the number of newly covered sets while minimizing the hits with already covered sets. The paper reports empirical evaluations and user studies over two classes of queries, "ambiguous" and "popular". The users were asked to compare 3 clustering algorithms over the set of queries and, for each query, were asked to answer 6 questions of a rather general nature on the generated hierarchy.

[2] The Eigencluster system is available on-line at `http://www-math.mit.edu/cluster/`

Ferragina and Gulli [3] propose a method for hierarchically clustering Web snippets, and produce a hierarchical labelling based on constructing a sequence of labelled and weighted bipartite graphs representing the individual snippets on one side and a set of labels (and corresponding clusters) on the other side. Data from the Open Directory Project (ODP)[3] is used in an off-line and query-independent way to generate predefined weights that are associated on-line to the words of the snippets returned by the queries. Data is collected from 16 search engines as a result of 77 queries chosen for their popularity among Lycos and Google users in 2004. The snippets are then clustered and the labels are manually tagged as relevant or not relevant to the cluster to which they have been associated. The clusters are ordered in terms of their weight, and quality is measured in terms of the number of relevant labels among the first n labels, for $n \in \{3, 5, 7, 10\}$. Note that in this work the emphasis is on the quality of the labels rather than on that of the clusters, and that the ground truth is defined "a posteriori", after the queries are processed.

3 The Clustering Algorithm and the Labelling Algorithm

The clustering algorithm. We approach the problem of clustering Web snippets as that of finding a solution to the classic k-center problem: *Given a set S of points in a metric space M endowed with a metric distance function D, and given a desired number k of resulting clusters, partition S into non-overlapping clusters $C_1, \ldots, C_k$ and determine their "centers" $\mu_1, \ldots, \mu_k \in M$ so that the radius $\max_j \max_{x \in C_j} D(x, \mu_j)$ of the widest cluster is minimized.* The k-center problem can be solved approximately using the furthest-point-first (FPF) algorithm [7, 20], which we now describe. Given a set S of n points, FPF builds a sequence $T_1 \subset \ldots \subset T_k = T$ of k sets of "centers" (with $T_i = \{\mu_1, \ldots, \mu_i\} \subset S$) in the following way.

1. At the end of iteration $i-1$ FPF holds the mapping μ defined for every point $p_j \in S \setminus T_{i-1}$ as: $\mu(p_j) = \arg\min_{\mu_s} D(p_j, \mu_s)$ i.e. the center in T_{i-1} closest to p_j; $\mu(p_j)$ is called the *leader* of p_j. Note that this mapping is established in the first iteration in time $O(n)$.
2. At iteration i, among all points p_j, FPF picks $\mu_i = \arg\max_{p_j} D(p_j, \mu(p_j))$ i.e. the point for which the distance to its leader is maximum, and makes it a new center, i.e. adds it to T_{i-1}, thus obtaining T_i. This selection costs $O(n)$.
3. Compute the distance of μ_i to any point in $S \setminus T_i$ and update the mapping μ if needed. Thus μ is now correct for the beginning of iteration $i+1$. This update phase costs $O(n)$.

The final set of centers $T = \{\mu_1, \ldots, \mu_k\}$ defines the resulting k-clustering, since each center μ_i implicitly identifies a cluster C_i as the set of data points whose leader is μ_i. Note that T_1 is initialized to contain a single point chosen at random from S; this random choice is due to the fact that, in practice, both the effectiveness and the efficiency of the algorithm can be seen experimentally to be insensitive to this choice.

Most of the computation is actually devoted to computing distances and updating the auxiliary mapping μ: this takes $O(n)$ time per iteration, so the total

[3] `http://www.dmoz.org/`

computational cost of the algorithm is $O(nk)$. In [1] we have thus defined an improved version of this algorithm that exploits the triangular inequality in order to filter out useless distance computations. This modified algorithm (M-FPF), which we now describe, works in any metric space, hence in any vector space[4].

Consider, in the FPF algorithm, any center $\mu_x \in T_i$ and its associated set of closest points $N(\mu_x) = \{p_j \in S \setminus T_i \mid \mu(p_j) = \mu_x\}$. We store $N(\mu_x)$ as a ranked list, in order of decreasing distance from μ_x. When a new center μ_y is added to T_i, in order to identify its associated set of closest points $N(\mu_y)$ we scan every $N(\mu_x)$ in decreasing order of distance, and stop scanning when, for a point $p_j \in N(\mu_x)$, it is the case that $D(p_j, \mu_x) \leq \frac{1}{2} D(\mu_y, \mu_x)$. By the triangular inequality, any point p_j that satisfies this condition cannot be closer to μ_y than to μ_x. This rule filters out from the scan points whose leader cannot possibly be μ_y, thus significantly speeding up the identification of leaders. Note that all distances between centers in T_i must be available; this implies an added $O(k^2)$ cost for computing and maintaining these distances, which is anyhow dominated by the term $O(nk)$.

Using medoids. The M-FPF is applied to a random sample of size $\sqrt{nk}$ of the input points (this sample size is suggested in [21]). Afterwards the remaining points are associated to the closest (according to the Generalized Jaccard Distance) center. We obtain improvements in quality by making an iterative update of the "center" when a new point is associated to a cluster. Within a cluster C_i we find the point a_i furthest from μ_i and the point b_i furthest from a_i (intuitively this is a good approximation to a diametral pair). The medoid m_i is the point in C_i that has the minim value of the function $|D(a_i, x) - D(b_i, x)| + |D(a_i, x) + D(b_i, x)|$, over all $x \in C_i$.[5] When we add a new point to C_i, we check if the new point should belong to the approximate diametral pair (a_i, b_i), and if so we update m_i accordingly. The association of the remaining points is done with respect to the medoids, rather than the centers. The application of M-FPF plus the iterative recomputation of medoids gave us a clustering of better quality than simply using M-PFP on the whole input set.

The distance function. Each snippet is turned into a "bag of words" after removing stop words and performing stemming. In [1] we report experiments using, as a distance function, (i) the cosine distance measure (i.e. the complement to 1 of the cosine similarity function) applied to vectors of terms weighted by $tf*idf$, and (ii) a slight modification of the standard Jaccard Distance, which we call *Weighted Jaccard Distance* (WJD); in those experiments, (ii) has performed at the same level of accuracy as (i), but has proven much faster to compute. In this paper we improve on the results of [1] by using the Generalized Jaccard Distance described in [22]. Given two "bag-of-words" snippet vectors $s_1 = (s_1^1, ...s_1^h)$ and $s_2 = (s_2^1, ...s_2^h)$, the *Generalized Jaccard Distance* is: $D(s_1, s_2) = 1 - \frac{\sum_i \min(s_1^i, s_2^i)}{\sum_i \max(s_1^i, s_2^i)}$. The term weights s_a^i consist of "weighted term frequencies", obtained as weighted sums of the numbers of occurrences of the term in the snippet, where weight 3 is assigned to a term occurring in the page title, weight 1 to a term occurring in the text fragment, and weight 0 is assigned to a term occurring in the URL (since,

[4] We recall that any vector space is also a metric space, but not vice-versa.

[5] This formula mimics in a discrete setting the task of finding the cluster point closest to the median point to the segment (a_i, b_i).

in previous experiments we had run, the text of the URL had proven to give no contribution in terms of cluster quality). Note that, when using unit weights only, the Generalized Jaccard Distance coincides with the standard Jaccard Distance.

The candidate words selection algorithm. We select candidate terms for labelling the generated clusters through a modified version of the information gain function [2]. For term t and category c, information gain is defined as $IG(t,c) = \sum_{x\in\{t,\bar{t}\}}\sum_{y\in\{c,\bar{c}\}} P(x,y)\log\frac{P(x,y)}{P(x)P(y)}$ Intuitively, IG measures the amount of information that each argument contains about the other; when t and c are independent, $IG(t,c)=0$. This function is often used for feature selection in text classification, where, if $IG(t,c)$ is high, the presence or absence of a term t is deemed to be highly indicative of the membership or non-membership in a category c of the document containing it. In the text classification context, the rationale of including in the sum, aside from the factor that represents the "positive correlation" between the arguments (i.e. the factor $P(t,c)\log\frac{P(t,c)}{P(t)P(c)} + P(\bar{t},\bar{c})\log\frac{P(\bar{t},\bar{c})}{P(\bar{t})P(\bar{c})}$), also the factor that represents their "negative correlation" (i.e. the factor $P(\bar{t},c)\log\frac{P(\bar{t},c)}{P(\bar{t})P(c)} + P(t,\bar{c})\log\frac{P(t,\bar{c})}{P(t)P(\bar{c})}$), is that, if this latter factor has a high value, this means that the absence (resp. presence) of t is highly indicative of the membership (resp. non-membership) of the document in c. That is, the term is useful anyway, although in a "negative" sense.

However, in our context we are interested in terms that *positively describe* the contents of a cluster, and are thus only interested in positive correlation. Therefore, we drop the factor denoting negative correlation from the IG formula, yielding the modified version $IG_m(t,c) = P(t,c)\log\frac{P(t,c)}{P(t)P(c)} + P(\bar{t},\bar{c})\log\frac{P(\bar{t},\bar{c})}{P(\bar{t})P(\bar{c})}$ that coincides with the positive correlation factor of IG. We use IG_m to select, for each cluster, words that are *representative* of the cluster and, at the same time, allow to discriminate among clusters.

4 Experimental Evaluation of the Clustering Algorithm

The baseline. As baseline against which to compare the clustering capabilities of Armil, we have chosen Vivisimo[6]. Vivisimo is considered an industrial standard in terms of clustering quality and user satisfaction, and in 2001 and 2002 it has won the "best meta-search-award" assigned annually by the on-line magazine SearchEngineWatch.com. Vivisimo thus represents a particularly difficult baseline, and it is not known if its clustering quality only depends on an extremely good clustering algorithm, or rather on the use of external knowledge or custom-developed resources. To the best of our knowledge, this is the first published experiment comparing the clustering quality of an academic prototype and Vivisimo. Vivisimo's advanced searching feature allows a restriction of the considered auxiliary search engines to a subset of a range of possible auxiliary search engines. For the purpose of our experiment we restrict our source of snippets to the ODP directory.

Measuring clustering quality. Following a consolidated practice, in this paper we measure the effectiveness of a clustering system by the degree to which it is able to "correctly" re-classify a set of pre-classified snippets into exactly the same

[6] `http://vivisimo.com/`

categories without knowing the original category assignment. In other words, given a set $C = \{c_1, \ldots, c_k\}$ of categories, and a set Θ of n snippets pre-classified under C, the "ideal" term clustering algorithm is the one that, when asked to cluster Θ into k groups, produces a grouping $C' = \{c'_1, \ldots, c'_k\}$ such that, for each snippet $s_j \in \Theta$, $s_j \in c_i$ if and only if $s_j \in c'_i$. The original labelling is thus viewed as the latent, hidden structure that the clustering system must discover.

The measure we use is *normalized mutual information* (see e.g. [23, page 110]), i.e. $NMI(C, C') = \frac{2}{\log |C||C'|} \sum_{c \in C} \sum_{c' \in C'} P(c, c') \cdot \log \frac{P(c, c')}{P(c) \cdot P(c')}$ where $P(c)$ represents the probability that a randomly selected snippet s_j belongs to c, and $P(c, c')$ represents the probability that a randomly selected snippet s_j belongs to both c and c'. The normalization, achieved by the $\frac{2}{\log |C||C'|}$ factor, is necessary in order to account for the fact that the cardinalities of C and C' are in general different [2]. Higher values of NMI mean better clustering quality. The clustering produced by Vivisimo has partially overlapping clusters (in our experiments Vivisimo assigned roughly 27% of the snippets to more than one cluster), but NMI is designed for non-overlapping clustering. Therefore, in measuring NMI we eliminate from the ground truth, from the clustering produced by Vivisimo, and from that produced by Armil, the snippets that are present in multiple copies.

However, in order to also consider the ability of the two systems to "correctly" duplicate snippets across overlapping clusters, we have also computed the *normalized complementary entropy* [23, page 108], in which we have changed the normalization factor so as to take overlapping clusters into account. The entropy of a cluster $c'_l \in C'$ is $E(c'_l, C) = \sum_{k=1}^{|C|} -\frac{|c'_l \cap c_k|}{|c_k|} \log \frac{|c'_l \cap c_k|}{|c_k|}$. The normalized complementary entropy of c'_l is $NCE(c'_l, C) = 1 - \frac{E(c'_l, C)}{\log |C|}$. NCE ranges in the interval $[0, 1]$, and a greater value implies better quality of c'_l. The complementary normalized entropy of C' is the weighted average of the contributions of the single clusters in C'. Let $n' = \sum_{l \in 1}^{|C'|} |c'_l|$ be the sum of the cardinalities of the clusters of C'. Note that when clusters may overlap it holds that $n' \geq n$. Thus $NCE(C', C) = \sum_{l \in 1}^{|C'|} \frac{|c'_l|}{n'} NCE(c'_l, C)$. NCE values reported below are thus obtained on the full set of snippets returned by Vivisimo.

Establishing the ground truth. Following [24], we have made a series of experiments using as input the snippets resulting from queries issued to the Open Directory Project (ODP – see Footnote 3). The ODP is a searchable Web-based directory consisting of a collection of a few million Web pages (as of today, ODP claims to index 5.1M Web pages) pre-classified into more than 590K categories by a group of volunteer human experts. The classification induced by the ODP labelling scheme gives us an objective "ground truth" against which we can compare the clustering quality of Vivisimo and Armil. In ODP, documents are organized according to a hierarchical ontology. For any snippet we obtain a label for its class by considering only the first two levels of the path on the ODP category tree. This coarsification is needed in order to balance the number of classes and the number of snippets returned by a query.

Queries are submitted to Vivisimo, asking it to retrieve pages only from ODP. This is done to ensure that Vivisimo and Armil operate on the same set of snippets, hence to ensure full comparability of the results. The resulting set of snippets

Table 1. Results of the comparative evaluation

	Vivisimo	Armil(40)	Armil(30)
NCE	0.667	0.735 (+10.1%)	0.683 (+2.3%)
NMI	0.400	0.442 (+10.5%)	0.406 (+1.5%)

is parsed and given as input to Armil. Since Vivisimo does not report the ODP category to which a snippet belongs, for each snippet we perform a query to ODP in order to establish its ODP-category.

Outcome of the comparative experiment. The queries used in this experiment are the last 30 of those reported in Appendix A (the first 5 have been excluded since too few related snippets are present in ODP). On average, ODP returned 41.2 categories for each query. In Table 1 we report the NMI and NCE values obtained by Vivisimo and Armil on these data. Vivisimo produced by default about 40 clusters; therefore we have run Armil with a target of 40 clusters (thus with a choice close to that of Vivisimo, and to the actual average number of ODP categories per query) and with 30 (this number is the default used in the user evaluation).

The experiments indicate an substantial improvement of about 10% in terms of cluster quality of Armil(40) with respect to Vivisimo.[7] This improvement is an important result since, as noted in 2005 in [3], "[T]he scientific literature offers several solutions to the web-snippet clustering problem, but unfortunately the attainable performance is far from the one achieved by Vivisimo." It should be noted moreover that Vivisimo uses a proprietary algorithm, not in the public domain, which might make extensive use of external knowledge. In contrast our algorithm is open and disclosed to the research community.

5 User Evaluation of the Cluster Labelling Algorithm

Assessing "objectively" the quality of a cluster labelling method is a difficult problem, for which no established methodology has gained a wide acceptance. For this reason a user study is the standard testing methodology. We have set up a user evaluation of the cluster labelling component of Armil in order to have an independent and measurable assessment of its performance. We performed the study on 22 volunteer master students, doctoral students and post-docs in computer science at our departments. The volunteers have all a working knowledge of the English language.

The user interface of Armil has been modified so as to show clusters one-by-one and proceed only when the currently shown cluster has been evaluated. The queries are supplied to the evaluators in a round robin fashion from a list of 35 predefined queries. For each query the user must first say whether the query is meaningful to her; an evaluator is allowed to evaluate only queries meaningful to her. For each cluster we propose three questions: (a) Is the label syntactically well-formed?; (b) Can you guess the content of the cluster from the label?; (c) After inspecting the cluster, do you retrospectively consider the cluster as well described

[7] For the sake of replicating the experiments all the search results have been cached and are available at `http://psp1.iit.cnr.it/~mcsoft/armil` .

Table 2. Correlation tables of questions row-(a) and column-(b) (left), row-(b) and column-(c) (middle), row-(a) and column-(c) (right). Entries in the top part give the percentage over all answers, and entries in the bottom part give percentage over rows.

	Yes	Sort-of	No	Yes	Sort-of	No	Yes	Sort-of	No
Yes	42.67%	12.81%	5.11%	33.52%	12.81%	3.72%	35.98%	18.93%	5.68%
Sort-of	5.74%	15.27%	4.41%	11.36%	16.85%	3.66%	8.64%	12.81%	3.97%
No	1.64%	3.78%	8.52%	2.14%	8.90%	7.00%	2.39%	6.81%	4.73%
Yes	70.41%	21.14%	8.43%	66.96%	25.59%	7.44%	59.37%	31.25%	9.37%
Sort-of	22.58%	60.04%	17.36%	35.64%	52.87%	11.48%	33.99%	50.37%	15.63%
No	11.76%	27.14%	61.08%	11.88%	49.30%	38.81%	17.19%	48.86%	33.93%

by the label? The evaluator must choose one of three possible answers (Yes; Sort-of; No), and her answer is automatically recorded in a database. Question (a) is aimed at assessing the gracefulness of the label produced. Question (b) is aimed at assessing the quality of the label as an instrument predictive of the cluster content. Question (c) is aimed at assessing the correspondence of the label with the content of the cluster. Note that the user cannot inspect the content of the cluster before answering (a) and (b).

Selection of the queries. Similarly to [3, 5], we have randomly selected 35 of the most popular queries submitted to Google in 2004 and 2005[8]; from the selection we have removed queries (such as e.g. "Spongebob", "Hilary Duff") that, referring to someone or something of regional interest only, were unlikely to be meaningful to our evaluators. The queries are listed in Appendix A.

Discussion of the results. Each of the 35 queries has been evaluated by two different evaluators, for a total of 70 query evaluations and 1584 cluster evaluations. The results are displayed in the following table:

	Yes	Sort-of	No
(a)	60.5%	25.5%	14.0%
(b)	50.0%	32.0%	18.0%
(c)	47.0%	38.5%	14.5%

By checking the percentages of No answers, we can notice that sometimes labels considered non-predictive are nonetheless considered well descriptive of the cluster; we interpret this fact as due to the discovery of meanings of the query string previously unknown to the evaluator. The correlation matrices in Table 2 show more precisely the correlation between syntax, predictivity and representativeness of the labels. The data in Table 2 (left) show that there is a strong correlation between syntactic form and predictivity of the labels, as shown by the fact that in a high percentage of cases the same answer was returned to questions (a) and (b). The middle and right part of Table 2 confirms that while for the positive or mildly positive answers (Yes, Sort-of) there is a strong correlation between the answers returned to the different questions, it is often the case that a label considered not predictive of the content of the cluster can still be found, after inspection of the cluster, to be representative of the content of the cluster.

Running times. Our system runs on an AMD Athlon (1Ghz Clock) processor with 750Mb RAM and operating system FreeBSD 4.11 - STABLE. The code

[8] `http://www.google.com/press/zeitgeist.html`

was developed in Python V. 2.4.1. Excluding the time needed to download the snippets from the auxiliary search engines, the 35 queries have been clustered and labelled in 0.72 seconds on average; the slowest query took 0.92 seconds.

6 Conclusions and Future Work

Why is Armil not "yet another clustering search engine"? The debate on how to improve the performance of search engines is at the core of the current research in the area of Web studies, and we believe that so far only the surface of the vein has been uncovered. The main philosophy of the system/experiments we have proposed follows these lines: (i) principled algorithmic choices are made whenever possible; (ii) clustering is clearly decoupled from labelling; (iii) attention is paid to the trade-off between response time and quality while limiting the response time within limits acceptable by the user; (iv) a comparative study of Armil and Vivisimo has been performed in order to assess the quality of Armil's clustering phase by means of effectiveness measures commonly used in clustering studies; (v) a user study has been set up in order to obtain an indication of user satisfaction with the produced cluster labelling; (vi) no use of external sources of knowledge is made.

Further research is needed in two main areas. First, we plan to assess to what extent a modicum of external knowledge can improve the system's performance without speed penalties. Second, it is possible to introduce in the current pipeline (input snippets are clustered, candidates are extracted, labels are generated) of the architecture a feedback loop by considering the extracted candidates/labels as predefined categories, thus examining which snippets in different clusters are closer to the generated labels. Snippets close to the label of cluster C_x but in a different cluster C_y could be shown on the screen as related also to C_x. This would give the benefits of soft clustering without much computational overload.

References

1. Geraci, F., Pellegrini, M., Pisati, P., Sebastiani, F.: A scalable algorithm for high-quality clustering of Web snippets. In: Proceedings of SAC-06, 21st ACM Symposium on Applied Computing, Dijon, FR (2006) 1058–1062
2. Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons, New York, US (1991)
3. Ferragina, P., Gulli, A.: A personalized search engine based on Web-snippet hierarchical clustering. In: Special Interest Tracks and Poster Proceedings of WWW-05, 14th International Conference on the World Wide Web, Chiba, JP (2005) 801–810
4. Lawrie, D.J., Croft, W.B.: Generating hierarchical summaries for Web searches. In: Proceedings of SIGIR-03, 26th ACM International Conference on Research and Development in Information Retrieval. (2003) 457–458
5. Kummamuru, K., Lotlikar, R., Roy, S., Singal, K., Krishnapuram, R.: A hierarchical monothetic document clustering algorithm for summarization and browsing search results. In: Proceedings of WWW-04, 13th International Conference on the World Wide Web, New York, NY (2004) 658–665
6. Zamir, O., Etzioni, O., Madani, O., Karp, R.M.: Fast and intuitive clustering of Web documents. In: Proceedings of KDD-97, 3rd International Conference on Knowledge Discovery and Data Mining, Newport Beach, US (1997) 287–290
7. Gonzalez, T.F.: Clustering to minimize the maximum intercluster distance. Theoretical Computer Science **38**(2/3) (1985) 293–306

8. Geraci, F., Pellegrini, M., Sebastiani, F., Maggini, M.: Cluster generation and cluster labelling for web snippets: A fast and accurate hierarchical solution. Technical Report IIT TR-1/2006, Institute for Informatics and Telematics of CNR (2006)
9. Kural, Y., Robertson, S., Jones, S.: Clustering information retrieval search outputs. In: Proceedings of the 21st BCS IRSG Colloquium on Information Retrieval, Glasgow, UK (1999)
10. Kural, Y., Robertson, S., Jones, S.: Deciphering cluster representations. Information Processing and Management **37** (1993) 593–601
11. Tombros, A., Villa, R., van Rijsbergen, C.J.: The effectiveness of query-specific hierarchic clustering in information retrieval. Information Processing and Management **38**(4) (2002) 559–582
12. Zamir, O., Etzioni, O.: Web document clustering: A feasibility demonstration. In: Proceedings of SIGIR-98, 21st ACM International Conference on Research and Development in Information Retrieval, Melbourne, AU (1998) 46–54
13. Cheng, D., Kannan, R., Vempala, S., Wang, G.: On a recursive spectral algorithm for clustering from pairwise similarities. Technical Report MIT-LCS-TR-906, Massachusetts Institute of Technology, Cambridge, US (2003)
14. Zhang, D., Dong, Y.: Semantic, hierarchical, online clustering of Web search results. In: Proceedings of APWEB-04, 6th Asia-Pacific Web Conference, Hangzhou, CN (2004) 69–78
15. Maarek, Y., Fagin, R., Ben-Shaul, I., Pelleg, D.: Ephemeral document clustering for Web applications. Technical Report RJ 10186, IBM, San Jose, US (2000)
16. Zeng, H.J., He, Q.C., Chen, Z., Ma, W.Y., Ma, J.: Learning to cluster Web search results. In: Proceedings of SIGIR-04, 27th ACM International Conference on Research and Development in Information Retrieval, Sheffield, UK (2004) 210–217
17. Osinski, S., Weiss, D.: Conceptual clustering using Lingo algorithm: Evaluation on Open Directory Project data. In: Proceedings of IIPWM-04, 5th Conference on Intelligent Information Processing and Web Mining, Zakopane, PL (2004) 369–377
18. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. Volume 1. (1967) 281–297
19. Cutting, D.R., Pedersen, J.O., Karger, D., Tukey, J.W.: Scatter/Gather: A cluster-based approach to browsing large document collections. In: Proceedings of SIGIR-92, 15th ACM International Conference on Research and Development in Information Retrieval, Kobenhavn, DK (1992) 318–329
20. Hochbaum, D.S., Shmoys, D.B.: A best possible approximation algorithm for the k-center problem. Mathematics of Operations Research **10**(2) (1985) 180–184
21. Indyk, P.: Sublinear time algorithms for metric space problems. In: Proceedings of STOC-99, ACM Symposium on Theory of Computing. (1999) 428–434
22. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: Proceedings of STOC-02, 34th Annual ACM Symposium on the Theory of Computing, Montreal, CA (2002) 380–388
23. Strehl, A.: Relationship-based Clustering and Cluster Ensembles for High-dimensional Data Mining. PhD thesis, University of Texas, Austin, US (2002)
24. Haveliwala, T.H., Gionis, A., Klein, D., Indyk, P.: Evaluating strategies for similarity search on the Web. In: Proceedings of WWW-02, 11th International Conference on the World Wide Web, Honolulu, US (2002) 432–442

A Queries Used in the User Evaluation

skype, winmx, nintendo revolution, pamela anderson, twin towers, wallpaper, firefox, ipod, tsunami, tour de france, weather, matrix, mp3, new orleans, notre dame, games, britney spears, chat, CNN, iraq, james bond, harry potter, simpsons, south park, baseball, ebay, madonna, star wars, tiger, airbus, oscars, london, pink floyd, armstrong, spiderman.

Principal Components for Automatic Term Hierarchy Building

Georges Dupret and Benjamin Piwowarski

Yahoo! Research Latin America
gdupret@yahoo-inc.com
bpiwowar@yahoo-inc.com

Abstract. We show that the singular value decomposition of a term similarity matrix induces a term hierarchy. This decomposition, used in Latent Semantic Analysis and Principal Component Analysis for text, aims at identifying "concepts" that can be used in place of the terms appearing in the documents. Unlike terms, concepts are by construction uncorrelated and hence are less sensitive to the particular vocabulary used in documents. In this work, we explore the relation between terms and concepts and show that for each term there exists a latent subspace dimension for which the term coincides with a concept. By varying the number of dimensions, terms similar but more specific than the concept can be identified, leading to a term hierarchy.

Keywords: Term hierarchy, principal component analysis, latent semantic analysis, information retrieval.

1 Introduction

Automated management of digitalized text requires a computer representation of the information. A common method consists in representing documents by a bag-of-words or set of features, generally a subset of the terms present in the documents. This gives rise to the vector space model where documents are points in an hyperspace with features as dimensions: The more important a feature in a document, the larger the coordinate value in the corresponding dimension [12].

Clearly, much information is lost when discarding the term order but the more significant limitation is that only the presence and the co-occurrence of terms are taken into account, not their meaning. Consequently, synonyms appear erroneously as distinct features and polysemic terms as unique features. This serious limitation is an avatar of the feature independence assumption implicit in the vector representation.

In the more general *statistical models* [20] (OKAPI) representations of queries and documents are clearly separated. Relevance of a document to a query is estimated as the product of individual term contributions. The corresponding assumption is not much weaker than strict independence.

Term dependence is taken into account in Language Models like *n-grams* and their applications to Information Retrieval [18], but generally within windows of

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 37–48, 2006.

two or three terms. The Bayesian network formalism [19] also allows for term dependence, but its application to a large number of features is unpractical.

Principal Component Analysis (PCA) [5] for text (and the related Latent Semantic Analysis method) offers a different approach: Uncorrelated linear combinations of features –the latent "concepts"– are identified. The lack of correlation is taken to be equivalent to independence as a first approximation, and the latent "concepts" are used to describe the documents. This work shows that more than a list of latent concepts, Principal Component Analysis uncovers a hierarchy of terms that share a "*related and more specific than*" relation.

Together with [7], this work extends the results of [6] beyond the context of Latent Semantic Analysis and PCA to all type of symmetric similarity measures between terms and hence documents and proposes a theoretical justification of the results. The main contribution, a method to derive a term hierarchy, is presented in Sect. 3. Numerical experiments in Sect. 4 validate the method while a review of automatic hierarchy generation methods is proposed in Sect. 5.

2 Term Similarity Measure

The estimation of the similarity between terms in Information Retrieval is generally based on term co-occurrences. Essentially, if we make the assumption that each document of a collection covers a single topic, two terms that co-occur frequently in the documents necessarily refer to a common topic and are therefore somehow similar. If the documents are not believed to refer to a single topic, it is always possible to divide them into shorter units so that the hypothesis is reasonably verified.

The Pearson correlation matrix $\mathbf{S}$ associated to the term by document matrix $\mathbf{A}$ is a common measure of term similarity. Nanas et al. [15] count the number of term co-occurrence in sliding windows of fixed length, giving more weight to pairs of terms appearing close from each other. Park et al. [17] use a Bayesian network. The method we present here does not rely on a particular measure of similarity or distance. The only requirement is an estimate of the similarity between any two index terms, represented by a symmetric matrix $\mathbf{S}$.

In the vector space representation of documents, index terms correspond to the base vectors of an hyperspace where documents are represented by points. If to each term j corresponds a base vector $\mathbf{e}_j$, an arbitrary document d is represented by $\mathbf{d} = \sum_{j=1}^{N} \omega_j \mathbf{e}_j$ where ω_j is the weight of term j in the document d. Weights are usually computed using the well known *tf.idf* formula and then normalized. The inconvenient of this representation stems from the implicit assumption of independence between terms: Consider two documents d_a and d_b each composed of a different single term. Independently of whether the single terms are synonyms, unrelated or antonyms, the documents similarity in the hyperspace representation is null because their representations coincide with two different base vectors. A more desirable result would be a non null similarity between terms u and v. This can be achieved by redefining the similarity measure

between documents: We will use the dot product in base $\mathbf{S}$ between the normalized document vectors[1].

$$\frac{\mathbf{d}_a^T}{|\mathbf{d}_a|}\mathbf{S}\frac{\mathbf{d}_b}{|\mathbf{d}_b|} = S_{u,v}$$

Alternatively, we can define an *extended representation* of a document d as $(1/|\mathbf{d}|)\mathbf{d}^T\sqrt{\mathbf{S}}$ and use the traditional cosine similarity measure[2].

The idea of introducing the similarity between terms to compute document similarity is closely related to Latent Semantic Analysis and Principal Component Analysis for text [6]. In the latter, the similarity between a set of documents and a query is computed as $\mathbf{r}(k) = \mathbf{A}^T\mathbf{S}(k)\mathbf{q}$ where $\mathbf{A}$ is the matrix formed by the space vector representation of the documents and q is a query. The i^{th} component of $\mathbf{r}(k)$, noted $r_i(k)$, is the similarity of document i with the query. The analogy with the *extended document representation* is clear, but instead of using the original similarity matrix $\mathbf{S}$, we use the rank k approximation of its eigenvalue decomposition. The matrix $\mathbf{S}$ can be decomposed into a product including the orthonormal matrix $\mathbf{V}$ and the diagonal matrix $\mathbf{\Sigma}$ of its eigenvalues σ_ℓ in decreasing value order: $\mathbf{S} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^T$. The best approximation following the Frobenius norm of the matrix $\mathbf{S}$ in a subspace of dimensionality $k < N$ is obtained by setting to zero the eigenvalues σ_ℓ for $\ell > k$. Noting $\mathbf{V}(k)$ the matrix composed of the k first columns of $\mathbf{V}$ and $\mathbf{\Sigma}(k)$ the diagonal matrix of the first k eigenvalues, we have $\mathbf{S}(k) = \mathbf{V}(k)\mathbf{\Sigma}(k)\mathbf{V}(k)^T$.

We can now represent in extended form a document $\mathbf{t}_u$ formed of a unique index term u in the rank k approximation of the similarity matrix:

$$\mathbf{t}_u^T = \mathbf{e}_u^T\sqrt{\mathbf{S}} = \mathbf{e}_u^T\mathbf{V}(k)\mathbf{\Sigma}(k)^{1/2} = \mathbf{V}_{u,\cdot}(k)\mathbf{\Sigma}(k)^{1/2} \quad (1)$$

where $\mathbf{\Sigma}(k)^{1/2}$ is the diagonal matrix of the square root of the eigenvalues in decreasing order and $\mathbf{V}_{u,\cdot}(k)$ is the u^{th} row of $\mathbf{V}(k)$. By analogy with the terminology introduced by Latent Semantic Analysis, the columns of $\mathbf{V}(k)$ represent latent concepts. The documents in general as well as the single term documents are represented with minimal distortion[3] as points in the k dimensional space defined by the k first columns – i.e. the eigenvectors – of $\mathbf{V}$ instead of the N dimensional space of index terms. This is possible only if the selected eigenvectors summarize the important features of the term space, hence the idea that they represent latent concepts.

In the next sections, we analyze the properties of the rank k approximation of the similarity matrix for different ranks and show how a hierarchy can be deduced.

3 The Concepts of a Term

We explore in this section the relation between terms and concepts. Sending a similarity matrix onto a subspace of fewer dimensions implies a loss of

[1] $\mathbf{S}$ being symmetric, but not necessarily full rank, this dot product introduces a quasi-norm [10].

[2] $\sqrt{\mathbf{S}}$ always exists because the singular values of $\mathbf{S}$ are all positive or null.

[3] according to the Frobenius norm.

information. We will see that it can be interpreted as the merging of terms meanings into a more general concept that encompasses them. We first examine the conditions under which a term coincides with a concept. Then we use the results to deduce a hierarchy.

A similarity matrix row $\mathbf{S}_{j,.}$ and its successive approximations $\mathbf{S}(k)_{j,.}$ represent a single term document $\mathbf{t}_j$ in terms of its similarity with all index terms. We seek a representation that is sufficiently detailed or encompass enough information for the term to be correctly represented. A possible way is to require that a single term document is more similar to itself than to any other term document:

Definition 1 (Validity). *A term is correctly represented in the k-order approximation of the similarity matrix only if it is more similar to itself than to any other term. The term is then said to be valid at rank k.*

If we remember that the normalized single term documents correspond to the base vectors, $\mathbf{e}_u$, the definition of validity requires: $\mathbf{e}_u^T\mathbf{S}(k)\mathbf{e}_u > \mathbf{e}_u^T\mathbf{S}(k)\mathbf{e}_v \ \forall u \neq v$ or equivalently $\mathbf{t}_u^T\mathbf{t}_u > \mathbf{t}_u^T\mathbf{t}_v \ \forall u \neq v$. This is verified if the diagonal term of $\mathbf{S}$ corresponding to u is larger than any other element of the same column, i.e. if $\mathbf{S}(k)_{u,u} > \mathbf{S}(k)_{u,v} \quad \forall v \neq u$. In other words, even though the diagonal element corresponding to term i is not equal to unity –which denotes perfect similarity by convention, it should be greater than the non-diagonal elements of the same row[4] to be correctly represented.

It is useful to define the rank below which a term ceases to be valid:

Definition 2 (Validity Rank). *A term* $\mathtt{t}$ *is optimally represented in the k-order approximation of the similarity matrix if it is valid at rank k and if* $k-1$ *is the largest value for which it is not valid. Rank k is the validity rank of term* $\mathtt{t}$ *and is denoted* $\mathrm{rank}(\mathtt{t})$.

In practice it might happen for some terms that validity is achieved and lost successively for a short range of ranks. It is not clear whether this is due to a lack of precision in the numerically sensitive eigenvalue decomposition process or to more fundamental reasons. The definition of validity was experimentally illustrated in [6] and a theoretical justification can be found in [2].

At a given rank k, if a term $\mathtt{a}$ is more similar to a valid term $\mathtt{c}$ than to itself, the representation of term $\mathtt{c}$ represents a meaning more general than $\mathtt{a}$: We say that $\mathtt{a}$ is generalised by the concept $\mathtt{c}$.

Definition 3 (Concept of a Term). *A term* $\mathtt{c}$ *is a concept of term* $\mathtt{a}$ *if* $\mathrm{rank}(\mathtt{c}) < \mathrm{rank}(\mathtt{a})$ *and if for some rank k such that* $\mathrm{rank}(\mathtt{c}) \leq k < \mathrm{rank}(\mathtt{a})$, $\mathtt{a}$ *is more similar to* $\mathtt{c}$ *than to itself.*

The requirement that $\mathrm{rank}(\mathtt{c}) < \mathrm{rank}(\mathtt{a})$ ensures that $\mathtt{a}$ is never a concept of $\mathtt{c}$ if $\mathtt{c}$ is a concept of $\mathtt{a}$.

It is possible to determine at each rank k the concepts of a term. To derive a hierarchy, we incrementally reconstruct the similarity matrix based on its decomposition $\mathbf{S} = \sum_{k=1}^{N} \sigma_k \mathbf{V}_{k,.}\mathbf{V}_{k,.}^T$. We collect for each k the links between concepts – i.e. terms whose representation is valid at rank k – and invalid terms.

[4] $\mathbf{S}(k)$ is symmetric and the condition can be applied indifferently on rows or columns.

Table 1. The first 30 direct links in *Shopping* and *Science* databases, ordered by decreasing coverage and limited to the stable links. Links in bold are in the ODP database.

Shopping		Science	
alberta	→ **canada**	**humidor**	→ **cigar**
monorail	→ lighting	**cuban**	→ **cigar**
criminology	→ sociology	**alberta**	→ **canada**
prehistory	→ **archaeology**	**cuckoo**	→ **clock**
romanian	→ slovenian	**grandfather**	→ **clock**
gravitation	→ relativity	fudge	→ chocolate
forensics	→ forensic	**soy**	→ **candle**
aztec	→ maya	**putter**	→ **golf**
karelian	→ finnish	**quebec**	→ **canada**
oceania	→ asia	racquetball	→ racket
transpersonal	→ **psychology**	**tasmania**	→ **australia**
etruscan	→ greek	airbed	→ mattress
barley	→ wheat	**glycerin**	→ **soap**
papuan	→ eastern	**snooker**	→ **billiard**
quebec	→ **canada**	**housebreaking**	→ **dog**
cryobiology	→ cryonics	waterbed	→ mattress
soho	→ solar	oceania	→ asia
catalysis	→ **chemistry**	tincture	→ herbal
geotechnical	→ engineering	**gunsmithing**	→ **gun**
iguana	→ **lizard**	chrysler	→ chevrolet
sociologist	→ **sociology**	equestrian	→ horse
olmec	→ maya	flamenco	→ guitar
oceanographer	→ **oceanography**	**pistachio**	→ **nut**
canine	→ dog	condiment	→ sauce
neptunium	→ plutonium	appraiser	→ estate
lapidary	→ mineral	**salsa**	→ **sauce**
raptor	→ **bird**	**ontario**	→ **canada**
ogham	→ irish	volkswagen	→ volvo
governmental	→ **organization**	arthropod	→ insect
forestry	→ forest	bulldog	→ terrier

There is a typically a range of ranks between rank(c) and rank(a) where a term a points to its concept c. This motivates the following definition:

Definition 4 (Coverage of a Link). *Define* $k_{\min}$ *and* $k_{\max}$ *as the minimum and maximum* k *that verify* $\text{rank}(\texttt{c}) \leq k < \text{rank}(\texttt{a})$ *and for which* c *is a concept of term* a*. The coverage of the link between the two concepts is the ratio*

$$\textit{coverage} = \frac{k_{\max} - k_{\min} + 1}{\text{rank}(\texttt{a}) - \text{rank}(\texttt{c})}$$

The coverage has values in $]0, 1]$.

The coverage reflects "how long" with respect to the possible range defined by rank(c) and rank(a), the valid term was a concept for the other term. We will see when we illustrate the hierarchy building procedure in Section 4 that the coverage is a good predictor of interesting links.

4 Numerical Experiments

There are no standard procedures to evaluate hierarchies although some attempts have been made [13]. Beyond the fact that evaluation is difficult even

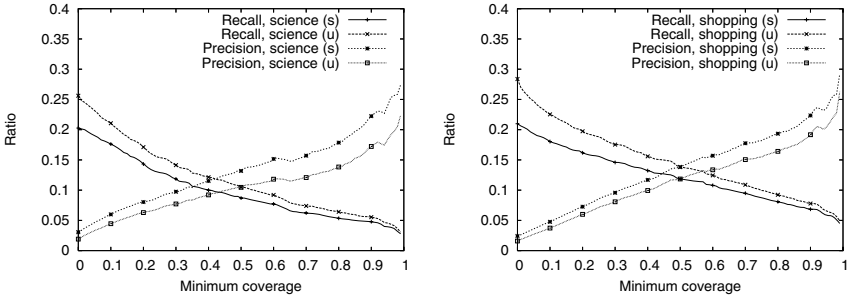


Fig. 1. Comparison of the PCA stable (s) and unstable (u) links with the ODP hierarchy on the Science and Shopping topics. *Recall* is the proportion of links in the original ODP hierarchy rediscovered by PCA. *Precision* is the proportion of ODP links among those retrieved by PCA. The x-axis is the coverage ratio: For a given value c, the PCA links we consider are those whose coverage is superior to c.

when a group of volunteers is willing to participate, it also depends on the task the hierarchy is designed for. For example, the measure used in [13] could not be applied here as the scoring is based on an estimate of the time it takes to find all relevant documents by calculating the total number of menus –this would be term nodes in this work– that must be traversed and the number of documents that must be read, which bears no analogy to this work.

We expect PCA to uncover two main types of relation between terms: The first one is semantic and can be found in dictionaries like WordNet[5]. These are relations that derives from the definition of the terms like "cat" and "animal" for example. The other kind of relation we expect to uncover is more circumstantial but equally interesting like, for example, "Rio de Janeiro" and "Carnival". These two words share no semantic relation, but associating them make sense. To evaluate the PCA hierarchy, we chose to compare the links it extracts from the document collection associated with the Open Directory Project[6] to the original, edited hierarchy. To identify the ability of PCA to extract "semantic" relations, we performed some experiments with WordNet which are not presented in this article due to a lack of space.

The *Open Directory Project* (ODP) is the most comprehensive human edited directory of the Web. We extracted two topics from this hierarchy, namely *Shopping* and *Science*. Out of the 104,276 and 118,584 documents referred by these categories, we managed to download 185,083 documents to form the database we use.

Documents were processed with a language independent part-of-speech tagger[7] and terms replaced by their lemmata. We extracted only adjectives and substantives to form the bag-of-word representations. Low and high frequency

[5] `http://wordnet.princeton.edu/`

[6] www.dmoz.org

[7] the TreeTagger home page can be found at `http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/`

terms as well as stopwords were discarded unless they appeared in the ODP hierarchy. Original documents were divided in parts of 25 consecutive terms to form new, shorter documents. The objective is to reduce the confusion of topics inside a same document (Section 2).

A path in the ODP hierarchy is composed as a series of topics, from the most generic to the most specific. An example of such a path is "Health/Beauty/-Bath_and_Body/Soap". We discard concepts described as a sequence of terms. For example, the previous sequence is transformed into "Health/Beauty/Soap". The hierarchy is then decomposed into direct links – i.e. relations that exist between adjacent terms – and indirect links where relations between terms belong to the same path. The direct links in our example are Health ← Beauty and Beauty ← Soap and the set of transitive links is composed of the former links more Health ← Beauty.

In order to test the stability of the discovered links, we *bootstrapped* [8] the document database. The method consists in picking randomly with replacement 185,083 documents from the original database to form a new correlation matrix before deducing a new set of links. This process is repeated ten times. The number of replications where a particular link appears reflects its stability with respect to variations in the database. We say that a link is *stable* when the relationship between the two terms held the ten times, and in the opposite case it is said to be *unstable*. For the science and shopping topics, half of the links are stable.

From this set of links between two terms we can construct a hierarchy of terms. Although cycles can appear among unstable links, they are absent by construction from the stable links. It would also be interesting to consider links that always appear in each bootstrap replication but with a different direction: This could be a good indicator of a symmetric relationship between two concepts.

With respect to the complexity of the algorithm, the term by term matrix is not sparse and the computation of the singular value decomposition is of order $O(n^3)$. This becomes rapidly intractable on regular desktop PC unless the number of terms is restricted to a range of between 5.000 and 10.000 terms and less frequent terms are discarded. This need not be a problem, given that the similarity of infrequent terms will be poorly estimated anyway.

In the remaining of this section, we compute the proportion of direct and indirect links present in ODP that we retrieve automatically with our Principal Component Analysis method. We also study the impact of link stability and coverage (Definition 4). Note that a large intersection between human and automatically generated links increases the confidence on the validity of the automatic method, but it does not invalidate the automatic links absent from edited hierarchy because documents and topics can be organized in a variety of equally good ways. This is corroborated in Table 1 where links absent from ODP are in normal font.

4.1 Coverage and Stability of Direct Links

Coverage is perceived as a relevant indicator of link quality because it reflects the strength that unite the two terms linked by a hierarchical relation. In Table 2,

Table 2. Number of links discovered by PCA in *Science* documents as a function of the coverage and, in parenthesis, the size of the intersection with the 2,151 *Science* ODP links

coverage	stable	unstable
0%	14,266 (436)	28,859 (551)
20%	3,832 (308)	5,850 (368)
40%	1,867 (251)	2,831 (261)
60%	1,095 (166)	1,676 (198)
80%	644 (115)	998 (138)
99%	218 (59)	294 (65)

the number of links discovered from the *Science* documents are reported as a function of the minimum coverage in both the stable and unstable cases. We see that 70% and 80% of the links have a coverage lower than 20%. Discarding all the links below this level of coverage results in the lost of only 30% and 33% of ODP links.

The stability is also an important selection criteria. We observe that if we consider all the PCA links, stable or not, we retrieve 551 of the original 2,151 ODP links present in Science. If we select only the stable links, we retrieve 436 ODP links, but the total number of PCA links is divided by two from 28,859 to 14,266. Some of the links present in the ODP hierarchy are lost, but more than half of the PCA links are discarded. A similar conclusion holds when varying the coverage minimum threshold. This justifies stability as an important criteria for selecting a link.

By analogy with the Information Retrieval measures, we define *recall* as the proportion of links in the original hierarchy that the PCA method manages to retrieve automatically. The *precision* is defined as the proportion of ODP links present in the set of PCA links. If we denote by H the set of links in the ODP human edited hierarchy and by A the set in the PCA automatic hierarchy, these measures become $recall = |H \cap A|/|H|$ and $precision = |H \cap A|/A$. Recall answers the question "How many ODP link do I retrieve automatically?", while precision answers "What is the concentration of ODP links among all the PCA links?"

Fig. 1 offers a global view of the impacts of stability and coverage on recall and precision for topics *Science* on the left and *Shopping* on the right. The portion of common links is significantly larger when the coverage is closer to its maximum. On both graphs, if we select only links with a coverage superior to 0.8, one tenth of the links in A are present in ODP. These results are good since the number of links in ODP is quite high in comparison with the number of relevant documents in the ad-hoc task of Information Retrieval, thus penalizing the recall. Moreover, ODP is not a gold standard and links not present in this hierarchy might still be useful.

When varying the coverage threshold from 0 to 1, precision increases and recall decreases almost always. This means that coverage is a good predictor of the link "relevance". This was verified empirically as well by inspecting some part of the discovered links ordered by coverage.

Summarizing, stability and coverage are both important predictors of link quality and PCA is able to identify a significant number of ODP links.

4.2 Transitive Links

Some links present in ODP might appear as combination of links in PCA and vice-versa. We already explained how ODP was processed to obtain these links. For PCA, we create a link between two terms if there is a path from one term to another. A link is said to be direct if it appears in the original hierarchy, and indirect if it was discovered by transitivity. A set of links is transitive if it includes both direct and indirect links.

The coverage being a good indicator of the link quality, we tried to extend this notion to transitive links. We found experimentally that the minimum coverage of all the traversed links led to the best results: An indirect link is penalized if all the paths between the two terms traverse a link with a low coverage.

A study of the effect of coverage and stability on precision and recall is reported in Fig. 2 where we aggregated the results over the science and shopping topics, and compared the direct and transitive ODP and PCA links. The results being similar for both topics, there is no need to treat them separately. Precision and recall when both links set are either transitive or direct (PCA, ODP and PCA+, ODP+ curves on Fig. 2) are very similar: This shows that precision is not much affected by the new PCA indirect links (around 38% more links, from 31,611 to 43,632) while recall is not much affected by the new ODP links (around 126% more links, from 4,153 to 9391). It is interesting also to observe that among the 2,297 links common to the transitive PCA and ODP sets, 1,998 are present in the direct PCA set. This is reflected on Fig. 2 (ODP+, PCA plot) where the corresponding precision curve is significantly superior while recall is less affected. This suggests that the indirect links of PCA did not contribute much.

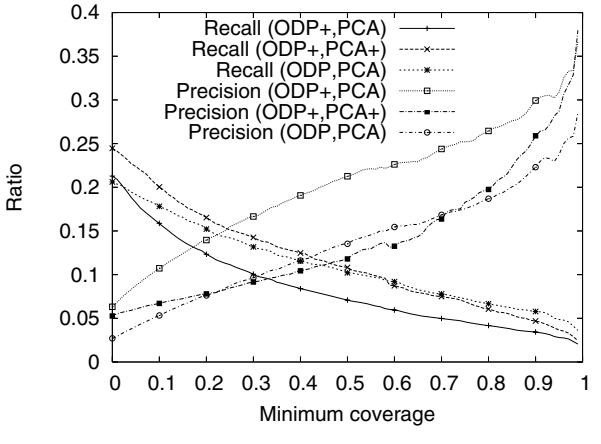


Fig. 2. Comparison of the PCA direct links (PCA) and transitive links (PCA+) with the ODP direct links (ODP) and transitive links (ODP+)

In conclusion, the new definition of coverage as the minimum on the path of traversed links proves a good selection indicator, as the precision increases with the coverage threshold. The manually derived and the PCA hierarchies share a significant amount of links and it seems that PCA is successful in discovering relations between terms. This is a specially good results given that the ODP directory is only one among numerous possible ways of organizing the documents in the database.

5 Related Work

Different fully automatic hierarchy discovery methods have already been proposed. The most popular one, from Sanderson and Croft [21], uses the co-occurrence information to identify a term that subsumes other terms. We tried various values of the unique parameter without succeeding in getting acceptable results. We suspect that a part of the problem stems from the heterogeneity of the corpus we used.

Njike-Fotzo and Gallinari [16] cluster documents prior to applying the Sanderson and Croft algorithm. This probably helps and will be used in future works. Nanas et al. [15] also proposed a method similar to Sanderson and Croft, but a subsumption relation is accepted if the terms involved are also correlated. The correlation is measured for terms appearing in windows of fixed length, and depends on the distance between them.

Hyponymy relations are derived from lexico-syntactic rules rather than plain co-occurrence in [11]. Another approach is to rely on frequently occurring words within phrases or lexical compounds. The creation of such *lexical hierarchies* has been explored and compared with subsumption hierarchies in [13]. In addition to the above two approaches, the same authors have investigated the generation of a concept hierarchy using a combination of a graph theoretic algorithm and a language model.

Glover et al. [9] base their hierarchy discovering algorithm on three categories: If a term is very common in a cluster of documents, but relatively rare in the collection, then it may be a good "self" term. A feature that is common in the cluster, but also somewhat common in the entire collection, is a description of the cluster, but is more general and hence may be a good "parent" feature. Features that are common in the cluster, but very rare in the general collection, may be good "child" features because they only describe a subset of the positive documents.

Application of traditional data mining and machine learning methods have also been tested. In [14], the learning mechanism is based on the Srikant and Agrawal [22] algorithm for discovering generalized association rules. A Bayesian network approach is proposed in [17]. Hierarchical clustering algorithm [1, 3] can be used to derive relations between terms, but cluster labelling is a challenging task. In [4] clustering is explicitly used to derive synonyms, hyperonyms and hyponyms relations.

6 Conclusion

We showed that the term similarity matrix induces a hierarchical relation among the terms. We computed this hierarchy based on the set of documents associated with two topics of the Open Directory Project hierarchy and observed significant similarities with the human edited original hierarchy.

We investigated different selection criteria and identified stability and coverage as good predictors of link quality. The coverage is especially interesting since it allows to order the links prior to selection. We also studied transitive links and showed that it is possible to extend to them the notion of coverage.

In conclusion, we observe that the hierarchy discovered by PCA is surprisingly good, especially if one considers only the stable links with a high coverage. The vast majority of links make sense and relations are uncovered than one would not expect to deduce from a simple co-occurence representation of documents.

References

1. L. D. Baker and A. K. McCallum. Distributional clustering of words for text classification. In W. B. Croft, A. Moffat, C. J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *SIGIR-98, 21st ACM*, pages 96–103, Melbourne, AU, 1998. ACM Press, New York, US.
2. H. Bast and D. Majumdar. Understanding spectral retrieval via the synonymy graph. In *SIGIR-05, 28th ACM*, 2005.
3. C. Y. Chung and B. Chen. Cvs: a correlation-verification based smoothing technique on information retrieval and term clustering. In *KDD '02: Eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 469–474, New York, NY, USA, 2002. ACM Press.
4. C. Y. Chung, R. Lieu, J. Liu, A. Luk, J. Mao, and P. Raghavan. Thematic mapping - from unstructured documents to taxonomies. In *CIKM '02*, pages 608–610, New York, NY, USA, 2002. ACM Press.
5. S. Deerwester, S. Dumais, G. Furnas, and T. Landauer. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41:391–407, 1990.
6. G. Dupret. Latent concepts and the number orthogonal factors in latent semantic analysis. In *SIGIR-03, 26th ACM*, pages 221–226. ACM Press, 2003.
7. G. Dupret and B. Piwowarski. Deducing a term taxonomy from term similarities. In *Second International Workshop on Knowledge Discovery and Ontologies, Porto, Portugal*, 2005.
8. B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC, May, 15 1993.
9. E. Glover, D. M. Pennock, S. Lawrence, and R. Krovetz. Inferring hierarchical descriptions. In *CIKM '02*, pages 507–514, New York, NY, USA, 2002. ACM Press.
10. D. Harville. *Matrix Algebra from a Statistician's Perspective*. Springer-Verlag, New York, 1997. 14.
11. M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *14th conference on Computational linguistics*, pages 539–545, Morristown, NJ, USA, 1992. Association for Computational Linguistics.

12. W. P. Jones and G. W. Furnas. Pictures of relevance: a geometric analysis of similarity measures. volume 38, pages 420–442, New York, NY, USA, 1987. John Wiley & Sons, Inc.
13. D. Lawrie and W. Croft. Discovering and comparing topic hierarchies. In *Proceedings of RIAO 2000*, 2000.
14. A. Maedche and S. Staab. Discovering conceptual relations from text. pages 321–325, 2000.
15. N. Nanas, V. Uren, and A. D. Roeck. Building and applying a concept hierarchy representation of a user profile. In *SIGIR-03, 26th ACM*, pages 198–204, New York, NY, USA, 2003. ACM Press.
16. H. Njike-Fotzo and P. Gallinari. Learning generalization/specialization relations between concepts - application for automatically building thematic document hierarchies. In *RIAO 2004*, Apr. 2004.
17. Y. C. Park, Y. S. Han, and K.-S. Choi. Automatic thesaurus construction using bayesian networks. In *CIKM '95*, pages 212–217, New York, NY, USA, 1995. ACM Press.
18. J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR-98, 21st ACM*, pages 275–281, New York, NY, USA, 1998. ACM Press.
19. B. A. N. Ribeiro and R. Muntz. A belief network model for ir. In *SIGIR-96: 19th ACM*, pages 253–260, New York, NY, USA, 1996. ACM Press.
20. S. Robertson and K. S. Jones. Simple proven approaches to text retrieval. Technical report tr356, Cambridge University Computer Laboratory, 1997.
21. M. Sanderson and B. Croft. Deriving concept hierarchies from text. In *SIGIR-99, 22th ACM*, pages 206–213, New York, NY, USA, 1999. ACM Press.
22. R. Srikant and R. Agrawal. Mining generalized association rules. *Future Generation Computer Systems*, 13(2–3):161–180, 1997.

Computing the Minimum Approximate λ-Cover of a String

Qing Guo[1], Hui Zhang[2], and Costas S. Iliopoulos[2]

[1] Department of Computer Science and Engineering, Zhejiang University, Hangzhou, Zhejiang 310027, China
guoqing@tiansign.com
[2] Department of Computer Science, King's College London Strand, London WC2R 2LS, England
{hui, csi}@dcs.kcl.ac.uk

Abstract. This paper studies the minimum approximate λ-cover problem of a string. Given a string x of length n and an integer λ, the minimum approximate λ-cover problem is to find a set of λ substrings of equal length that covers x with the minimum error, under a variety of distance models including the Hamming distance, the edit distance and the weighted edit distance. We present an algorithm that can solve this problem in polynomial time.

1 Introduction

String regularities mainly concern the repetitive structures of strings. Typically, a substring w is a *period* of a given string x if x is a prefix of a string constructed by concatenations of w. By allowing superpositions as well as concatenations, the notion of periods is generalized, called *covers*.

In the literature, a tremendous amount of research has been done on computing the covers of a given string x of length n. Apostilico, Farach and Iliopoulos [1] first introduced the notion of covers and presented a linear-time algorithm to test superprimitivity. Breslauer [2] described a linear time on-line algorithm to compute the shortest cover of every prefix of x. Moore and Smyth [8], and recently Li and Smyth [7] both gave a solution to find all the covers of x. In parallel computation, Iliopoulos and Park [5] gave a work-time optimal $O(\log \log n)$ algorithm for the shortest cover problem of x.

Extending the definition of covers in the sense that a set of substrings instead of a single substring of x are examined, Zhang et al.[11] introduced the notion of λ-covers. Given an integer λ, the λ-cover problem attempts to find all the sets of λ substrings each of equal length that cover x. A general algorithm that can solve this problem in $O(n^2)$ time was also presented.

Advances in multimedia technology and computational biology has shown that it could be of significant benefit to relax the basis for regularities. For instance, one seldom expects to find exact repetitions in molecular sequence analysis, but *approximate regularities* that allow errors to some extent. In this case, we consider a string "matching" a given pattern if the distance between them is

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 49–60, 2006.

within allowed bounds under a predefined metric. The most classic metrics are *Hamming distance*, *edit distance* and *weighted edit distance*.

Sim, Iliopoulos, Park and Smyth [9] introduced the notion of *approximate period* and provided polynomial time algorithms for finding approximate periods. Then, Sim, Park, Kim and Lee [10] solved the *approximate covers* problem in polynomial time as well.

Inspired by the idea of λ-covers, we introduce in this paper the notion of *approximate λ-covers*, approximate version of λ-covers. The motivation comes from the need for biological sequence analysis and its interaction with the alignment problem as solved by BLAST. It could be considered as a preliminary operation before the alignment of two sequences because it discovers motifs having adequate properties for that. We focus on solving the minimum approximate λ-cover problem of a string. To avoid triviality, we simply consider the case $\lambda > 1$.

2 Preliminaries

A *string* is a sequence of zero or more symbols over an alphabet Σ. A string x of length n is represented by an array $x[1..n] = x[1]x[2]\cdots x[n]$, where $x[i]$ is the i-th symbol of $x(x[i] \in \Sigma$ for $1 \leq i \leq n)$. The *empty string* is the empty sequence (of zero length) denoted by ε. The set of all strings over the alphabet Σ(including the empty string) is denoted by Σ^*.

A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$; x is equivalently a *superstring* of w. A substring of length p is called a *p-substring* for short. For a nonempty substring $w = x[i..j]$, we say that w occurs at position i and i is an occurrence of w in x. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $w = uw$ for $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1..n]$ and $y = y[1..m]$ such that $x[n-i+1..n] = y[1..i]$ for some $i \geq 1$, the string $x[1..n]y[i+1..m]$ is a *superposition* of x and y with i overlaps, we say that x and y are *overlapping*. A substring u is said to be a *cover* of x if x can be constructed by concatenations and superpositions of u. For example, the string $x = abababa$ has a cover aba.

The distance $\delta(x, y)$ between two strings x and y indicates the minimum cost to transform x into y. The cost arises from a sequence of operations, which is the sum of the cost of the individual operations. The most frequent operations that are used for string transformation mainly consist of:

- *Insertion*: inserting an extraneous character a, denoted by $\varepsilon \to a$.
- *Deletion*: deleting a character a, denoted by $a \to \varepsilon$.
- *Substitution*: Replacing a character a by another character b $(b \neq a)$, denoted by $a \to b$.

The following commonly used distance functions rest on above operations:

- *Hamming distance*: allows only substitutions, costing 1 for each operation. Note that, this model is restricted to two strings of the same length.

- *Levenshtein or edit distance*: allows insertions, deletions and substitutions. In this model, we count the number of edit operations, the cost of each equal to 1.
- *weighted edit distance*: is a generalized model of edit distance, where each edit operation has a different cost, stored in a *penalty matrix*.

Taking the lengths of strings x and y into account, a distance function $\delta(x, y)$ is called a *relative distance function*, otherwise an *absolute distance function*. The following two methods are used to define a relative distance function between x and y: The *error ratio with respect to* x is defined to be $d/|x|$; the *symmetric error ratio* is defined to be d/l, where d is an absolute distance between x and y, and $l = (|x|+|y|)/2$. Obviously, the Hamming distance and the edit distance are both examples of absolute distance. The weighted edit distance can be viewed as a relative distance function since the penalty matrix contains arbitrary costs.

3 Problem Definitions

As we mentioned above, the notion of approximate λ-covers of strings is extended from the notion of approximate covers, which can be formulated as below:

Definition 1. *Let x and w be strings over Σ^*, δ be a distance function and t be an integer, we say that w is a t-approximate cover of x if and only if there exist strings $w_1, w_2, \ldots, w_\xi$ $(w_i \neq \varepsilon)$ such that:*

(1) *$\delta(w, w_i) \leq t$ for $1 \leq i \leq \xi$, and*
(2) *x can be constructed by concatenations or superpositions of the strings $w_1, w_2, \ldots, w_\xi$.*

Considering a set of substrings instead of a single string to cover x, we give the following definitions:

Definition 2. *Let $u_1, u_2, \ldots, u_k$ be substrings of x, we say that the set $U = \{u_1, u_2, \ldots, u_k\}$ is a combination of k substrings drawn from x, called for short a k-combination of x, and a (k, p)-combination of x if each u_i $(1 \leq i \leq k)$ is of length p.*

Definition 3. *Given a string x of length n, integers λ and p, we say that a (λ, p)-combination $U = \{u_1, u_2, \ldots, u_\lambda\}$ is a (λ, p)-cover of x if and only if every position of x lies within an occurrence of some u_i $(1 \leq i \leq \lambda)$.*

Definition 4. *Let $U = \{u_1, u_2, \ldots, u_\lambda\}$ be a (λ, p)-combination of string x, δ be a distance function and t be an integer, we say that U is a t-approximate (λ, p)-cover of x if and only if there exist strings $w_1, w_2, \ldots, w_\xi$ $(w_i \neq \varepsilon)$ such that:*

(i.) for every w_i $(1 \leq i \leq \xi)$, there exists at least a u_j $(1 \leq j \leq \lambda)$ that suffices to: $\delta(u_j, w_i) \leq t$, and
(ii.) x can be constructed by concatenating or overlapping copies of the strings $w_1, w_2, \ldots, w_\xi$.

In this paper we mainly study the minimum approximate λ-cover problem, which can be formally stated as below:

Problem 1. Let x be a string of length n, λ be an integer and δ be a distance function, the minimum approximate λ-cover problem is to find a set $U = \{u_1, u_2, \ldots, u_\lambda\}$ of substrings of x such that:

(i.) $|u_1| = |u_2| \cdots = |u_\lambda|$;
(ii.) U is an approximate λ-cover of x with the minimum error.

4 Method and Solution

The solution to the minimum approximate λ-cover problem is based on the solution to the following fundamental problem:

Problem 2. Given a string x of length n, a (λ, p)-combination $U = \{u_1, u_2, \ldots, u_\lambda\}$ and a distance function δ, find the minimum integer t such that U is a t-approximate (λ, p)-cover of x.

In this problem, the set U is given a priori. As p, the size of each u_i is given as an input, it makes no difference whether δ is an absolute distance function or a relative one. The method for solving this problem is similar to the method for solving the *Smallest Distance Approximate Cover* problem defined as below:

Problem 3. Given a string x of length n, a string w of length m and a distance function δ, find the minimum integer t such that w is a t-approximate cover of x.

Sim, Park, Kim and Lee [10] as well as Christodoulakis, Iliopoulos, Park and Sim [3] presented polynomial time algorithms for Problem 3. We first give a brief description for the algorithm.

4.1 Main Idea of the Algorithm for Problem 3

The algorithm consists of two steps:

1) *Compute the distance between w and every substring of x.*

We denote the distance between w and $x[i..j]$ for $1 \leq i \leq j < n$ by:

$$d_{i,j} = \delta(x[i..j], w) \tag{1}$$

The computation of $d_{i,j}$ depends on the distance function we are using. If the Hamming distance is used, we simply consider those substrings of x of length m, executing character-by-character comparison between them and w.

If the edit distance (weighted or not) is used, we use a dynamic programming table, called for short the *D-table* to compute $d_{i,j}$. For two strings x and y, a D-table is a matrix of size $(|x| + 1) \times (|y| + 1)$. Each entry $D(i, j)$, $0 \leq i \leq |x|$

and $0 \leq j \leq |y|$, stores the minimum cost of converting $x[1..i]$ to $y[1..j]$. Initially, $D(0,0) = 0$, $D(i,0) = D(i-1,0) + \delta(x[i], \varepsilon)$, $D(0,j) = D(0,j-1) + \delta(\varepsilon, y[j])$. Then we can compute all the entries of the D-table in $O(|x||y|)$ time by the following recurrence:

$$D[i,j] = min \begin{cases} D[i-1,j] + \delta(x[i], \varepsilon) \\ D[i,j-1] + \delta(\varepsilon, y[j]) \\ D[i-1,j-1] + \delta(x[i], y[j]) \end{cases}$$

where $\delta(a,b)$ is the cost of substituting character a with character b, $\delta(a,\varepsilon)$ is the cost of deleting a and $\delta(\varepsilon,a)$ is the cost of inserting a.

2) Compute the minimum t such that w is a t-approximate cover of x.

Let t_i be the minimum value such that w is a t_i-approximate cover of $x[1..i]$. Initially, $t_0 = 0$. Then for $i = 1$ to n, using dynamic programming, we iteratively compute:

$$t_i = \min_{h_{min} \leq h \leq h_{max}} \{\max\{\min_{h \leq j < i} \{t_j\}, d_{h+1,i}\}\} \tag{2}$$

The value t_n is the minimum t we are looking for. We now explain how to compute t_i using this equation, assuming that we have already obtained the previous values: $t_1, t_2, \ldots, t_{i-1}$. For every position h, we cover a suffix $x[h+1..i]$ of $x[1..i]$ with one copy of w with error $d_{h+1,i}$. What is left to be covered is $x[1..h]$, which involves the following ways: cover either $x[1..h]$ with error t_h, or $x[1..h+1]$ with error t_{h+1}, ..., or $x[1..i-1]$ with error t_{i-1}. We choose the $x[1..j]$ that contributes the smallest error. Fig. 1 shows the way that w covers $x[1..i]$, where the shaded box is the best $x[1..j]$ with the minimum error.

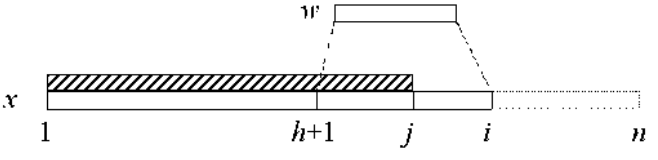


Fig. 1. Computing the minimum error

The number of positions to be checked, h_{min} and h_{max}, depends on the distance function we are using. When the Hamming distance is used, the fact that two strings of equal length are defined for this model allows us to examine only one position $h = i - m$. When the edit distance is used, only those $2m$ suffixes of $x[1..i]$ of length not greater than $2m$ are considered. However, all the suffixes of $x[1..i]$ have to be considered when δ is weighted edit distance.

Theorem 1. *The Smallest Distance Approximate Cover problem can be solved in $O(mn)$ time when the Hamming or edit distance is used for δ, and in $O(mn^2)$ time when δ adopts a weighted edit distance.*

The proof of this theorem can be referred to the original paper[10].

4.2 Our Algorithm for Problem 2

Compared with Problem 3, Problem 2 concentrates on a set of substrings of x of equal length that "cooperatively" cover x. Our algorithm for solving Problem 2 is similar to that for Problem 3, with the major distinction that every element of the set should be comprehensively considered rather than only one pattern as we described above. Therefore, we simply explain the differences between our algorithm and the original one.

Recall that in Problem 2, a (λ, p)-combination $U = \{u_1, u_2, \ldots, u_\lambda\}$ of x is given, where λ is a constant and $p = |u_k|$ $(1 \leq k \leq \lambda)$ is fixed. There are still two steps involved in our algorithm:

1) *For every u_k $(1 \leq k \leq \lambda)$, compute the distance between u_k and every substring of x.*

To denote the distance between u_k and $x[i..j]$ for $1 \leq i \leq j < n$, we modify Equation (1) to:

$$d_{i,j}(k) = \delta(x[i..j], u_k) \tag{3}$$

The computation of $d_{i,j}(k)$ depends on the distance function we are using:

- *Hamming distance*: We simply consider those p-substrings of x, that is, those $x[i..j]$'s satisfying $j - i + 1 = p$. For each of exactly $n - p + 1$ such substrings, we compute $d_{i,j}(k)$ through character-by-character comparison with u_k in p units of time. Hence, it takes $(n - p + 1)p = O(pn)$ time for each u_k and $O(\lambda pn) = O(pn)$ time for all λ patterns.
- *Edit distance*: We consider those $x[i..j]$'s for which $j - i + 1 \leq 2p$. Utilizing the incremental algorithm [6], we initially create the D-table for the distance between $x[1..2p]$ and u_k, then iteratively update the D-tables. It costs $O(pn)$ time to get all the D-tables relating to u_k, and $O(\lambda pn) = O(pn)$ time to compute $d_{i,j}$'s for λ patterns.
- *Weighted edit distance*: The distance between two characters in this model is given by arbitrary values, instead of 1 or 0 in the edit distance model. Therefore, in a more generally way, we build a D-table of size $(p+1) \times (n-i)$ for the distance between $x[i..n-i]$ and u_k for each position i of x. The last row of this table gives the values of $d_{i,i}$, $d_{i,i+1}$, ..., $d_{i,n}$. Thus, the overall time complexity for n positions is $pn(n-1)/2 = O(pn^2)$ for each u_k and the same for all u_k's.

2) *Compute the minimum t such that U is a t-approximate (λ, p)-cover of x.*

Let t_i be the minimum value such that U is a t_i-approximate (λ, p)-cover of $x[1..i]$. Equation (2) is changed to:

$$t_i = \min_{h_{min} \leq h \leq h_{max}} \{\max\{\min_{h \leq j < i} \{t_j\}, \min_{1 \leq k \leq \lambda} \{d_{h+1,i}(k)\}\}\} \tag{4}$$

The value t_n is the minimum t we are seeking for, which can be computed in the same manner with the algorithm described in Section 4.1. The difference

is that, we compute the distance $d_{h+1,i}(k)$ between $x[h+1..i]$ with every u_k ($k \in [1, \lambda]$) for every position h, then keep the minimum one. In other words, we find the best way to cover $x[h+1..i]$, with the minimum error. Then we cover $x[1..h]$ with the smallest error. The values of h_{min} and h_{max} depend on the distance function we are using.

- *Hamming distance*: $h_{min} = h_{max} = i - p$. The inner *min* loop runs in $O(p)$ time, since $min\{t_j\}$ and $min\{d_{h+1,i}(k)\}$ requires p units of time and λp units of time respectively. Thus the total time complexity for step 2 is $O(pn)$.
- *Edit distance*: $h_{max} = i - 1$, $h_{min} = i - 2p$. Occupying $O(n)$ additional memory, we can compute $\min_{h \le j < i}\{t_j\}$ in constant time as follows: At stage i, we store $m[h] = \min_{h \le j < i}\{t_j\}$ for $h \in [0, i-1]$; then at stage $i+1$, we obtain $\min_{h \le j < i+1}\{t_j\} = \min\{m[h], t_{i+1}\}$. Note that, this small trick works for the weighted edit distance as well. Thus, the computation runs in $2p\lambda n$ units of time, i.e. $O(pn)$ in total.
- *Weighted edit distance*: $h_{max} = i - 1$, $h_{min} = 0$. computing the minimum t requires $O(n^2)$ time.

Corollary 1. *Problem 2 can be solved in $O(pn)$ time when the Hamming distance or the edit distance is used for δ, and in $O(pn^2)$ time when δ uses a weighted edit distance.*

This directly follows from Theorem 1 and our analysis above.

4.3 Computing the Minimum Approximate λ-Cover

Now we turn to investigate the minimum approximate λ-cover problem. In this problem, the (λ, p)-combination U is not given, thus any (λ, p)-combination of x for $1 \le p < n/\lambda$ can be viewed as a candidate approximate λ-cover. The reason for this assumption is because if $p \ge n/\lambda$, we can trivially obtain exact (λ, p)-covers of x that are composed of $x[1..p]$, $x[n-p+1..n]$ and other $\lambda - 1$ or $\lambda - 2$ p-substrings, which easily lead to approximate (λ, p)-covers with small errors.

For a certain p, as a (λ, p)-combination $U = \{u_1, u_2, \ldots, u_\lambda\}$ is a set of different p-substrings of x, we need to record all the distinct substrings of x and efficiently list all the (λ, p)-combinations. We use Crochemore's partitioning [4] to compute all the substrings in a string. The main idea of this algorithm rests on the following definition of the equivalence relation over the positions of the string:

Definition 5. *Given a string $x[1..n] \in \Sigma^*$, two positions $i, j \in \{1, \ldots, n-p+1\}$ of x, then $(i, j) \in E_p$ iff $x[i...i+p-1] = x[j...j+p-1]$, noted iE_pj.*

That is, two positions i and j in x belong to the same one E_p-class when two p-substrings starting at i and j in x are identical. Clearly every E_p-class represents a distinct p-substring of x. Since E_{p+1} is a refinement of E_p, the algorithm first computes E_1, then iteratively builds E_2, E_3, ... until all classes are singletons. Utilizing the "smaller-half trick" [4], the partitioning takes $O(n \log n)$ time in total. For more details readers can refer to [4].

Our algorithm for computing all the λ-combinations works as follows: Initially, the algorithm tests valid (λ, p)-combinations of x for induction in base case, which corresponds to the first p such that the number of distinct substrings is greater than λ. Then it works iteratively to deduce all the (λ, p)-combinations of x according to the $(\lambda, p-1)$-combinations.

The iterative process relies on the construction of Equivalence Class Tree (ECT), which expresses the relationship between each E_{p-1}-class and its corresponding E_p-classes. As we mentioned above, each E_{p-1}-class can be partitioned into several E_p-classes by extending one more character to the right. Let $\{C_1, \ldots, C_k\}$ be the E_{p-1}-classes, we can create ECT as follows: The root of ECT has label 0. There are k nodes of depth $p-1$, each of which is denoted by the index of $C_i (1 \leq i \leq k)$. The sons of the node corresponding to C_i are the indices of E_p-classes partitioned by C_i. For the convenience of explanation, we label every node by the substring itself instead of the index of its equivalence class. Fig. 2 constructs ECT of a string $x = abbabbaababbab$.

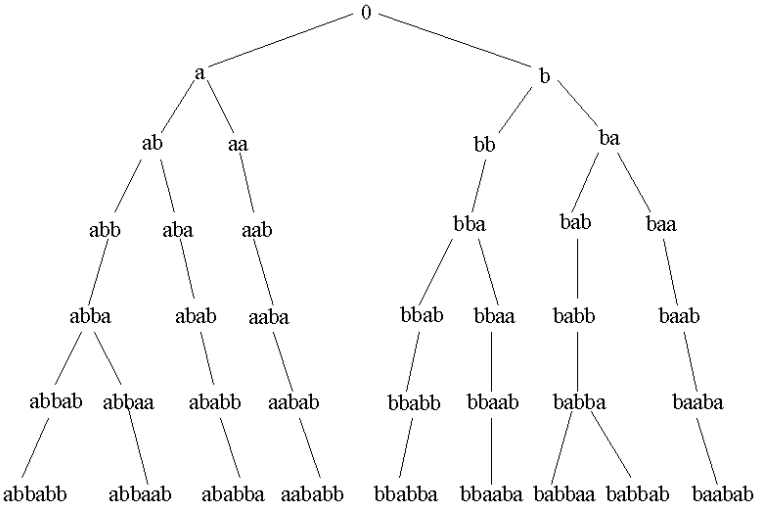


Fig. 2. ECT of string $x = abbabbaababbab$

Note that ECT is built along with the partitioning of equivalence classes. When all the equivalence classes for p-substrings are computed, the corresponding nodes of depth p are added into ECT. Thus the construction of ECT costs the same time with the partitioning, that is, $O(n\ logn)$ time.

Suppose that we have created ECT for x. Let m be the number of E_p-classes, that is, the number of distinct substrings of length p in x. Our algorithm can be formally stated as follows:

(1) *Base case*:

Initially consider $p = 1$, the number of E_1-classes is at most $m = \min(|\Sigma|, n)$. If $m > \lambda$, the combinations of m items taken λ are at most C_m^λ. If $m \leq \lambda$, we

have to turn to $p = 2, 3, \cdots$, then base case is the first p such that $m > \lambda$, which allows us to find qualified (λ, p)-combinations for deduction.

(2) *Inductive step*:

Assume that at stage $p-1$, we have stored all the $(\lambda, p-1)$-combinations of x. Our target is to iteratively deduce all the (λ, p)-combinations of x according to the $(\lambda, p-1)$-combinations. Consider a certain $(\lambda, p-1)$-combination $U = \{u_1, u_2, \ldots, u_\lambda\}$, it might produce a series of (λ, p)-combinations as a result of the partitioning of each of λ E_{p-1}-classes according to ECT. Let the number of sons of u_i in ECT be r_i. From $i = 1$ to $i = \lambda$, we successively substitute l_i among these r_i p-substrings respectively for u_i. Every current combination obtained after u_i being processed is saved to be further updated, denoted by S_i.

First consider u_1. As the number of sons of u_1 in ECT is r_1, the relevant p-substrings can be denoted by $s_1^1, s_1^2, \ldots, s_1^{r_1}$, clearly $r_1 \leq |\Sigma|$. We update U by replacing u_1 with l_1 among r_1 p-substrings related to u_1. When $l_1 > 0$, we need to remove $l_1 - 1$ from the other $\lambda - 1$ u_i's $(i > 1)$ to compose λ-combinations, which leads to $\sum C_{r_1}^{l_1} C_{\lambda-1}^{\lambda-l_1}$ S_1's; when $l_1 = 0$, we keep the $(\lambda-1)$-combination $\{u_2, u_3, \ldots, u_\lambda\}$ as an eligible S_1 to be further processed in the following stages. Some examples of S_1 are listed below:

$$\{s_1^1, s_1^2, \ldots, s_1^{l_1}, u_2, u_3, \ldots, u_{\lambda-l_1+1}\}$$
$$\{s_1^1, s_1^2, \ldots, s_1^{l_1}, u_{l_1+1}, \ldots, u_{\lambda-1}, u_\lambda\}$$
$$\{s_1^{r-l_1+1}, \ldots, s_1^{r-1}, s_1^r, u_2, u_3, \ldots, u_{\lambda-l_1+1}\}$$

(i) $r_1 < \lambda$: As $1 \leq l_1 \leq r_1$ and $r_1 \leq \min(|\Sigma|, \lambda - 1)$, the cardinality of $\{S_1\}$ is at most $\sum_{l_1=1}^{min(|\Sigma|,\lambda-1)} C_{min(|\Sigma|,\lambda-1)}^{l_1} C_{\lambda-1}^{\lambda-l_1} + 1$, a constant dependent only on λ and $|\Sigma|$.

(ii) $r_1 \geq \lambda$: In this case, $1 \leq l_1 \leq \lambda$, $r_1 \leq |\Sigma|$. There are at most $\sum_{l_1=1}^{\lambda} C_{|\Sigma|}^{l_1} C_{\lambda-1}^{\lambda-l_1} + 1$ S_1's, which is also a constant independent of p.

Next, assume that we have obtained $\{S_j\}(1 \leq j \leq \lambda - 1)$, a set of S_{j+1}'s can be obtained by updating every S_j as follows:

(i) If S_j contains u_{j+1} as a member: Keeping all those members of S_j located before u_{j+1}, namely, $l_1 + l_2 + \ldots + l_j$ p-substrings produced respectively by u_1, u_2, …, u_j unchanged, we process u_{j+1} in the same manner as we did on u_1. The following cases need to be discussed.

(a) u_{j+1} is not the last element of S_j: Replace u_{j+1} by l_{j+1} among r_{j+1} p-substrings partitioned by u_{j+1} according to ECT, then remove $l_{j+1} - 1$ from the remaining u_i's $(i > j+1)$ in the case of $l_{j+1} \geq 1$; or delete u_{j+1} and reserve all the remaining u_i's $(i > j+1)$ in the case of $l_{j+1} = 0$. Let $\lambda' = \lambda - (l_1 + l_2 + \ldots + l_j)$, then the upper limit of l_{j+1} is:

$$\max\{l_{j+1}\} = \begin{cases} \lambda', & r_{j+1} \geq \lambda'; \\ \min(|\Sigma|, \lambda' - 1), & \text{otherwise.} \end{cases}$$

(b) u_{j+1} is the last element of S_j: Replace u_{j+1} with exactly λ' among r_{j+1} p-substrings partitioned by u_{j+1} in the case of $r_{j+1} \geq \lambda'$; or delete this S_j otherwise.

(ii) If S_j does not contain u_{j+1}: $S_{j+1} \leftarrow S_j$.

Evidently, the values of l_i and r_i $(1 \leq i \leq j+1)$ in ECT directly account for the number of S_{i+1}'s, each of which rests on $|\Sigma|$ and λ as we analyzed above. That is, the cardinality of $\{S_{i+1}\}$ is a constant independent of p.

Finally, after u_λ is processed, we obtain $\{S_\lambda\}$ consisting of all the (λ, p)-combinations associated with the given $(\lambda, p-1)$-combination U, which has constant size. Consequently, the total number of (λ, p)-combinations is linear to the number of $(\lambda, p-1)$-combinations, which amounts to $O(n)$ since in base case there are at most $O(n)$ λ-combinations to be processed. The inductive step stops until p reaches n/λ, thus there sums to $O(n^2)$ such combinations to be used as candidate approximate λ-covers.

Applying each of $O(n^2)$ candidates into our algorithm for problem 2, we can find the minimum approximate λ-cover of x. Since the length p is not fixed, we use a relative distance function rather than an absolute one. For instance, we adopt as δ the error ratio with respect to x in the case of the Hamming and edit distance, or a weighted edit distance.

Theorem 2. *The minimum approximate λ-cover problem can be solved in $O(n^4)$ time when a relative Hamming or edit distance is used for δ, in $O(n^5)$ time when a weighted edit distance is used for δ.*

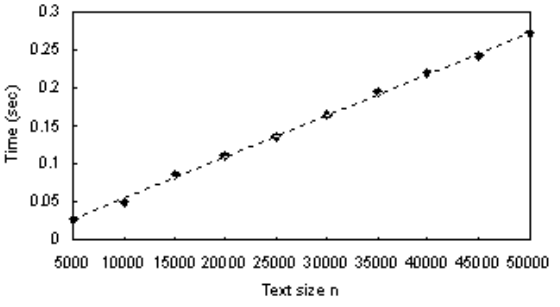
5 Experimental Results

To verify the running times of our algorithms, we implemented our algorithm, programmed in C++, for computing the approximate λ-covers when δ uses a Hamming distance. The experiment environment is a Pentium-4M CPU 1.8GHz system, with 256MB of RAM, under the Microsoft Windows XP operating system (SP2).

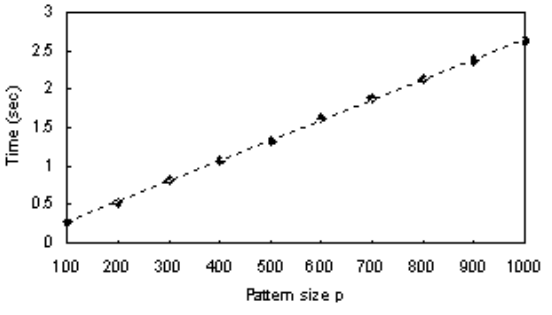
In our experiments, the text string x are pseudo random strings on the alphabet $\{a, b\}$. We first test the time complexity of our solution to Problem 2. In this case, a (λ, p)-combination U is randomly selected from λ-combinations of p-strings over $\{a, b\}$, where λ is set to be 3. For each test, we execute our program for 5 times and count the average value.

Observe Fig. 3, which shows the time consumption of the algorithm for Problem 2 when the Hamming distance is used. Fig. 3(a) gives the time consumption with respect to the text size n when p is fixed to be 100. It is easily noticed that time increases linearly when n increases. Similarly, the running time is proportional to the pattern size p, as presented in Fig. 3(b) where n is fixed to be 50000. Thus it is obvious that, Problem 2 can be computed in $O(pn)$ time.

In the case of the minimum approximate λ-cover problem, we choose the candidates for U to be random 3-combinations of substrings of x of length 5. Fig. 4 shows the the time consumption of the algorithm for this problem. The time complexity is $O(n^4)$ when the Hamming distance is used.



(a) time spent with respect to n



(b) time spent with respect to p

Fig. 3. Problem 2 when the Hamming distance is used

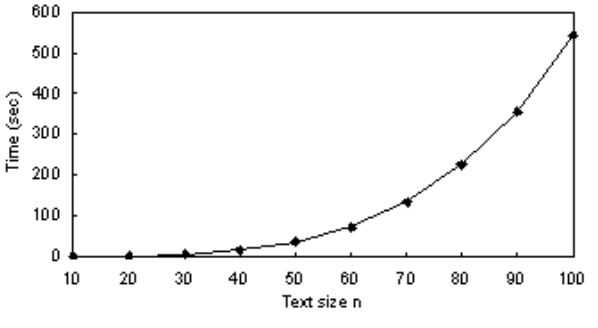


Fig. 4. The minimum approximate λ-cover problem when the Hamming distance is used

6 Conclusions

In this paper we presented a polynomial time algorithm for solving the minimum approximate λ-cover problem of a string. Compared with the corresponding

approximate covers problem relating to a single substring of x, our algorithm does not increase the time complexity, despite the fact that it considers a set of substrings of x. Moreover, our algorithm can also be used to solve the minimum approximate λ-seed problem. Our future direction is focused on generalizing the approximate λ-cover problem in the sense that a set of substrings of different lengths are to be considered.

References

1. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings, Information Processing Letters, 39, (1991) 17-20.
2. Breslauer, D.: An on-line string superprimitivity test, Information Processing Letters, 44, (1992) 345-347.
3. Christodoulakis, M., Iliopoulos, C.S., Park, K., Sim, J.S.: Approximate seeds of strings, Proc. of the 2003 Prague Stringology Conference (PSC'03), (2003) 25-36.
4. Crochemore, M.: An Optimal Algorithm for Computing the Repetitions in a Word, Information Processing Letters, 12 (5), (1981) 244-250.
5. Iliopoulos, C.S., Park, K.: An optimal $O(\log \log n)$ time algorithm for parellel superprimitivity testing, J. of the Korean Information Science Society, 21(8), (1994) 1400-1404.
6. Kim, S., Park, K.: A dynamic edit distance table, Proc. 11th Symp. Combinatorial Pattern Matching, Springer, Berlin, volume 1848, (2000) 60-68.
7. Yin Li, Smyth, W.F.: Computing the cover array in linear time, Algorithmica, 32(1), (2002) 95-106.
8. Moore, D.W.G., Smyth, W.F.: A correction to "Computing the covers of a string in linear time", Information Processing Letters, 54, (1995) 101-103.
9. Sim, J.S., Iliopoulos, C.S., Park, K., Smyth, W.F.: Approximate periods of strings, Theoretical Computer Science, 262, (2001) 557-568.
10. Sim, J.S., Park, K., Kim, S., Lee, J.: Finding approximate covers of strings, Journal of Korea Information Science Society, 29(1), (2002) 16-21.
11. Zhang, H., Guo, Q., Iliopoulos, C.S.: Computing the λ-covers of a string. Accepted by AWOCA2006

Sparse Directed Acyclic Word Graphs

Shunsuke Inenaga[1,2] and Masayuki Takeda[2,3]

[1] Japan Society for the Promotion of Science
[2] Department of Informatics, Kyushu University, Fukuoka 812-8581, Japan
{shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp
[3] SORST, Japan Science and Technology Agency (JST)

Abstract. The suffix tree of string w is a text indexing structure that represents all suffixes of w. A sparse suffix tree of w represents only a subset of suffixes of w. An application to sparse suffix trees is composite pattern discovery from biological sequences. In this paper, we introduce a new data structure named *sparse directed acyclic word graphs* (*SDAWGs*), which are a sparse text indexing version of directed acyclic word graphs (DAWGs) of Blumer et al. We show that the size of SDAWGs is linear in the length of w, and present an on-line linear-time construction algorithm for SDAWGs.

1 Introduction

Text indexing is a classical technique for pattern matching. Probably, the most widely known structure for text indexing is suffix trees [1, 2]. Indeed, quite a lot of applications for suffix trees have been introduced so far, and many problems are efficiently solved by using suffix trees [3]. Some of those problems require 'variants' of suffix trees that are modified for the specific purposes. This paper focuses on one of such problems, named the *sparse text indexing problem*, described as follows:

Input: Pattern string p, text string t of length n, and subset $S \subseteq \{1, \ldots, n\}$ of all positions in t.
Output: Whether or not there is any occurrence of p in t which begins at a position in S.

Kärkkäinen and Ukkonen [4] introduced the *sparse suffix tree* which stores only the suffixes of t beginning at the positions in S. Sparse suffix trees enable us to solve the above problem in time proportional to the pattern length (for fixed alphabets). An example of applications to sparse suffix trees is composite pattern discovery from biological sequences [5, 6, 7]. It is not difficult to see that sparse suffix trees can be constructed in $O(n)$ time and space, in such a way that we firstly construct a normal suffix tree of t in $O(n)$ time, and then prune the leaves for the suffixes which do *not* begin with positions in S. Now, a natural interest is whether or not the sparse suffix tree for t w.r.t. S is directly constructible in $O(n)$ time using $O(k)$ space, where $k = |S|$. (Note that $O(n)$ time consumption is unavoidable due to the necessity of reading entire t at least once.) To the best of our knowledge, this is still an open problem.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 61–73, 2006.

However, another representation of S and some simple alteration to t make it possible to construct its sparse suffix tree efficiently. Let # be a unique symbol not appearing anywhere in t, and let us insert # into t at the positions listed in S. Now we get a string of length $n + k$, and let us denote this string by w. Remark that since $k \leq n$, w is at most twice long as t. Now, if we are able to construct a sparse suffix tree representing only the suffixes of w which begin immediately after #, then this tree is an alternative to the sparse suffix tree for t. At a matching phase of pattern p we simply ignore any #'s in the edge labels of the tree.

This tree is also known as the *word suffix tree* whose concept was first introduced in [8]. Let Σ be an alphabet, and let $D = \Sigma^*\#$ be a dictionary of words over Σ, each followed by #. Now assume that w is a string in D^+, namely, w is a sequence $w_1 \cdots w_k$ of k words in D. Then, the word suffix tree of w w.r.t. D is a tree structure which represents only the k suffixes in the form of $w_i \cdots w_k$. We can associate the special symbol # with, e.g., a 'word separator' such as the blank symbol in the European languages, and then the word suffix tree of w represents only the suffixes of w starting at the beginning of words. In this way, we can avoid unwanted matches such as 'other' in 'mother'. Andersson et al. [9] introduced an algorithm to build the word suffix tree for w w.r.t. D with $O(k)$ space, but in $O(n)$ *expected* time. Very lately, we invented an algorithm that constructs word suffix trees with $O(k)$ space and in $O(n)$ time *in the worst cases* [10].

In this paper, we introduce a new data structure named *sparse directed acyclic word graphs* (*SDAWGs*) as an alternative to the word suffix trees and to the sparse suffix trees thereby. SDAWGs are a sparse text indexing version of *directed acyclic word graphs* (*DAWGs*) of Blumer et al. [11]. We give a formal definition of SDAWGs based on an equivalence relation on string w and dictionary D, and show the size of SDAWGs to be linear in n. One might concern that $O(n)$ space consumption is a disadvantage of SDAWGs against sparse suffix trees requiring only $O(k)$ space, but we recall that the edge labels of sparse suffix trees are implemented as pairs of pointers to the positions of w, and therefore the input string w has to be kept stored. Consequently, the total space requirement is also $O(n)$ for using the sparse suffix tree. On the other hand, any edge label of SDAWGs is a single symbol, and thus the input string w can be discarded after the SDAWG is constructed.

Finally, we present an on-line linear-time algorithm for constructing SDAWGs. Our algorithm is based on, and generalizes, the on-line construction algorithm for DAWGs invented by Blumer et al. [11]. Our algorithm directly constructs SDAWGs without building normal DAWGs as intermediate structures, by using the minimum DFA accepting dictionary D as suggested in [10]. We emphasize that SDAWGs can be obtained by first constructing the corresponding normal DAWGs, removing the unnecessary suffixes from the DAWGs, and then minimizing the resulting graphs. However, since these are non-tree DAGs, removing only one suffix may take linear time. Therefore, the algorithm presented in this paper is the only known one capable of building SDAWGs in $O(n)$ time.

2 Preliminaries

2.1 Notations

Let Σ be a finite set of symbols, called an *alphabet*. Throughout this paper we assume that Σ is fixed. A finite sequence of symbols is called a *string*. We denote the length of string w by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let Σ^* be the set of strings over Σ. For any symbol $a \in \Sigma$, we define a^{-1} such that $a^{-1}a = \varepsilon$.

Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. A prefix, substring, and suffix of string w are said to be *proper* if they are not w. Let $\mathit{Prefix}(w)$ be the set of the prefixes of string w, and let $\mathit{Prefix}(S) = \bigcup_{w \in S} \mathit{Prefix}(w)$ for set S of strings.

Definition 1 (Prefix property). *A set L of strings is said to satisfy the* prefix property *if no string in L is a proper prefix of another string in L.*

The i-th symbol of string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. For any strings $x, w \in \Sigma^*$, let

$$\mathit{Endpos}_w(x) = \{j \mid x = w[j - |x| + 1..j]\}.$$

Let D be a set of strings called a *dictionary*. A *factorization* of string w w.r.t. D is a list $w_1, \ldots, w_k$ of strings in D such that $w = w_1 \cdots w_k$ and $w_i \in D$ for each $1 \leq i \leq k$. In the rest of the paper, we assume that $D = \Sigma^*\#$ where $\#$ is a special symbol not belonging to Σ, and that $w \in D^+$. Then, a factorization of w w.r.t. D is always unique, since D clearly satisfies the prefix property because of $\#$ not being in Σ. Let M_D denote the minimum DFA which accepts $D = \Sigma^*\#$. It is easy to see that M_D requires only constant space (refer to the left of Fig. 2).

Let

$$\mathit{Suffix}_D(w) = \{w_i \cdots w_k \mid 1 \leq i \leq k + 1\}.$$

Then, $\mathit{Suffix}_D(w)$ consists only of the original string w, the suffixes which immediately follow $\#$ in w, and the empty string ε intended by $w_{k+1}w_k$. We define set $\mathit{Wordpos}_D(w)$ of the word-starting positions in w as follows:

$$\mathit{Wordpos}_D(w) = \{|w| - |s| + 1 \mid s \in \mathit{Suffix}_D(w)\}.$$

2.2 Equivalence Class on Strings over D

For set S of integers and integer i, we denote $S \oplus i = \{j + i \mid j \in S\}$ and $S \ominus i = \{j - i \mid j \in S\}$. Now we define the end-equivalence relation $\equiv_w$ on $w \in D^+$ by:

$$\begin{aligned} x \equiv_w y \Leftrightarrow \mathit{Endpos}_w(x) \cap (\mathit{Wordpos}_D(w) \oplus |x| \ominus 1) \\ = \mathit{Endpos}_w(y) \cap (\mathit{Wordpos}_D(w) \oplus |y| \ominus 1). \end{aligned}$$

We note that the above end-equivalence relation is a 'word-position-sensitive' version of the equivalence relation introduced by Blumer et al. [11], where the intersection with $\mathit{Wordpos}_D(w)$ makes it word-position-sensitive. We denote by $[x]_w$ the equivalence class of x w.r.t. $\equiv_w$.

Proposition 1. *All strings that are not in* $\mathit{Prefix}(\mathit{Suffix}_D(w))$ *form one equivalence class under* $\equiv_w$*, called the* degenerate class.

Proof. Since for any string $x \notin \mathit{Prefix}(\mathit{Suffix}_D(w))$ we have $\mathit{Wordpos}_D(w) = \emptyset$, we consequently obtain $\mathit{Endpos}_w(x) \cap (\mathit{Wordpos}_D(w) \oplus |x| \ominus 1) = \emptyset$. Moreover, for any string $y \in \mathit{Prefix}(\mathit{Suffix}_D(w))$, it is easy to observe that $\mathit{Endpos}_w(y) \cap (\mathit{Wordpos}_D(w) \oplus |x| \ominus 1) \neq \emptyset$. □

It follows from the definition of $\equiv_w$ that if two strings x, y are in a same non-degenerate equivalence class under $\equiv_w$, then either x is a suffix of y, or vice versa. Thus, each non-degenerate equivalence class under $\equiv_w$ has a unique longest member, which is called the *representative* of it. The representative of $[x]_w$ is denoted by $\overleftarrow{x}^w$.

3 Sparse Directed Acyclic Word Graphs

3.1 Definitions

Here we define the *sparse directed acyclic word graphs* (*SDAWGs* in short) as edge-labeled DAGs (V, E) with $E \subseteq V \times \Sigma^+ \times V$ where the second component of each edge represents its label.

Definition 2 (Sparse directed acyclic word graph). *The* sparse directed acyclic word graph *of string* $w \in D^+$*, denoted by* $\mathit{SDAWG}_D(w)$*, is a DAG* (V, E) *such that*

$$V = \{[x]_w \mid x \in \mathit{Prefix}(\mathit{Suffix}_D(w))\},$$
$$E = \{([x]_w, a, [xa]_w) \mid x, xa \in \mathit{Prefix}(\mathit{Suffix}_D(w)) \text{ and } a \in \Sigma \cup \{\#\}\}.$$

$\mathit{SDAWG}_D(w)$ has single *source node* $[\varepsilon]_w$ of in-degree zero, and single *sink node* $[w]_w$ of out-degree zero.

We associate each node $[x]_w$ of $\mathit{SDAWG}_D(w)$ with $\mathit{length}([x]_w) = |\overleftarrow{x}^w|$. For any edge $([x]_w, a, [xa]_w)$, if $\mathit{length}([xa]_w) = \mathit{length}([x]_w) + 1$, this edge is called *primary*; otherwise, it is called *secondary*.

Fig. 1 shows $\mathit{SDAWG}_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$, together with the normal $\mathit{DAWG}(w)$ representing all the suffixes of w. Observe that $\mathit{SDAWG}_D(w)$ only represents the suffixes $\mathtt{a\#b\#a\#bab\#}$, $\mathtt{b\#a\#bab\#}$, $\mathtt{a\#bab\#}$, $\mathtt{bab\#}$, and ε, all from $\mathit{Suffix}_D(w)$.

Also, observe that substrings $\mathtt{a\#b}$ and $\mathtt{b}$ are in distinct nodes of $\mathit{DAWG}(w)$, while they are in the same node of $\mathit{SDAWG}_D(w)$. It is because $\mathit{Endpos}_w(\mathtt{a\#b}) = \{3, 7\} \neq \mathit{Endpos}_w(\mathtt{b}) = \{3, 7, 9\}$, but $\mathit{Endpos}_w(\mathtt{a\#b}) \cap (\mathit{Wordpos}_D(w) \oplus 2) =$

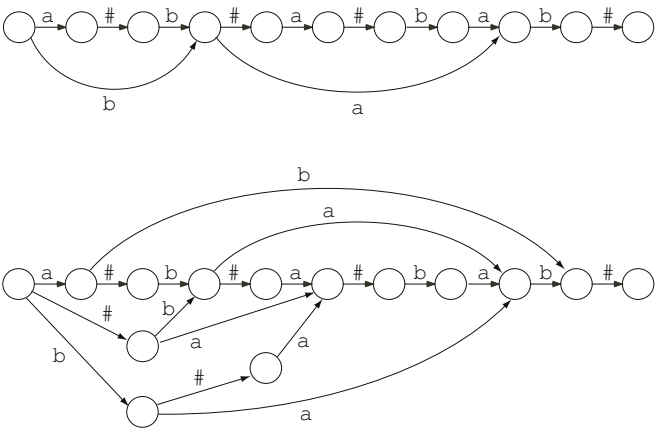


Fig. 1. $SDAWG_D(w)$ with $w = \mathtt{a\#b\#a\#bab\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}^*\#$ is shown on the upper, and normal $DAWG(w)$ is shown on the lower for comparison. Observe that $SDAWG_D(w)$ contains only suffixes of $Suffix_D(w)$, while $DAWG(w)$ has all the suffixes of $Suffix(w)$. For instance, $\mathtt{ab\#}$ is a suffix of w and is in normal $DAWG(w)$, but is not in $SDAWG_D(w)$.

$Endpos_w(\mathtt{b}) \cap (Wordpos_D(w) \oplus 0) = \{3, 7\}$, as $Wordpos_D(w) = \{1, 3, 5, 7\}$. Similar discussion holds for the pair of strings $\mathtt{a\#b\#}$ and $\mathtt{b\#}$.

Now we define the *suffix links* of $SDAWG_D(w)$, which are extensively used for on-line linear-time construction algorithm to be given later on. Also, they play an important role to bound the size of $SDAWG_D(w)$ within $O(n)$ space. For any string $x \in Prefix(Suffix_D(w))$, we consider a partition $x = x_1x_2$ such that $x_1 \in D^*$ and x_2 is a *proper* prefix of some string in D. Then, it is easy to see that the partition x_1x_2 is unique for any $x \in Prefix(Suffix_D(w))$.

The following proposition is clear from the definition of the end-equivalence.

Proposition 2. *For any $x, y \in Prefix(Suffix_D(w))$ such that $x \equiv_w y$ and $|x| > |y|$, we have $x_1 = vy_1$ with $v \in D^+$ and $x_2 = y_2$.*

Definition 3 (Suffix links of SDAWGs). *For any node $[x]_w$ of $SDAWG_D(w)$, let $x' = x'_1x'_2$ be the shortest member of $[x]_w$.*

1. *If $x'_1 \in D^+$, the suffix link from node $[x]_w$ goes to node $[u]_w$ such that $\overleftarrow{u}^w = u = u_1u_2$, $u_1 \in D^*$, $u_2 = x'_2$, and $x'_1 = hu_1$ for some $h \in D$;*
2. *Otherwise (If $x'_1 = \varepsilon$), the suffix link from $[x]_w$ goes to the initial state of M_D.*

Fig. 2 displays $SDAWG_D(w)$ and its suffix links, with $w = \mathtt{a\#b\#a\#bab\#}$. For instance, see Node 8 that is $[x]_w = \{\mathtt{a\#b\#a\#b}, \mathtt{b\#a\#b}\}$, where $x' = \mathtt{b\#a\#b}$, $x'_1 = \mathtt{b\#a\#}$, and $x'_2 = \mathtt{b}$. The suffix link of Node 8 goes to Node 4 that is $[u]_w = \{\mathtt{a\#b}, \mathtt{b}\}$, where $\overleftarrow{u}^w = \mathtt{a\#b}$, $u_1 = \mathtt{a\#}$, and $u_2 = \mathtt{b}$. Observe that $h = \mathtt{b\#}$, $x'_1 = hu_1 = \mathtt{b\#a\#}$ and $x'_2 = u_2 = \mathtt{b}$.

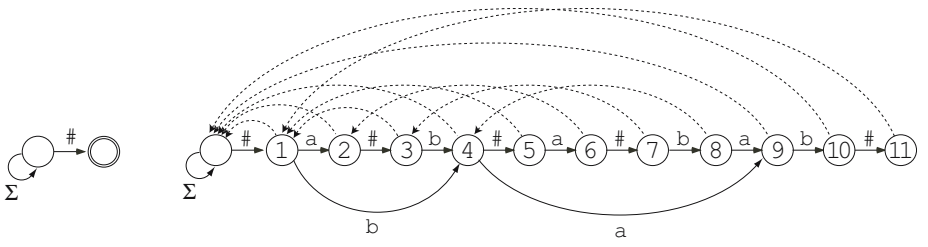


Fig. 2. To the left is the minimum DFA M_D accepting $D = \Sigma^*\#$, and to the right is $SDAWG_D(w)$ for $w = \texttt{a\#b\#a\#bab\#}$, with M_D and its suffix links (broken arrows) attached. Nodes 3, 5, 6, 7, 8, and 11 are in Group 1 of Definition 3, and nodes 1, 2, 4, 9, and 10 are in Group 2.

3.2 Size Bound

Here we analyze the size of $SDAWG_D(w)$. Firstly, we show that any distinct prefixes of w are associated with distinct nodes of $SDAWG_D(w)$.

Lemma 1. *For any strings $x, xa \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$, $x \not\equiv_w xa$.*

Proof. By the length argument, $|x| \in Endpos_w(x)$ but $|x| \notin Endpos_w(xa)$. Since $1 \in Wordpos_D(w)$ for any $w \in D^+$, $|x| \in Endpos_w(x) \cap (Wordpos_D(w) \oplus |x| \ominus 1)$ but $|x| \notin Endpos_w(xa) \cap (Wordpos_D(w) \oplus |xa| \ominus 1)$. Thus we have $x \not\equiv_w xa$. □

According to the above lemma, $SDAWG_D(w)$ has at least $n + 1$ nodes, each corresponding to a certain prefix of w. In addition, for any proper prefix x of w, there exists primary edge $([x]_w, a, [xa]_w)$ in $SDAWG_D(w)$ with $xa \in Prefix(w)$.

To show the upper bound for the size of $SDAWG_D(w)$, we consider the suffix link tree $T_D(w) = (V \cup \{q_s\}, E_\ell)$ where q_s is the initial state of M_D and is the root of $T_D(w)$, and E_ℓ is the set of the 'reversed' suffix links of $SDAWG_D(w)$.

The following lemma is critical to bound the size of $SDAWG_D(w)$ within linear space w.r.t. n.

Lemma 2. *If $\overleftarrow{x}^w \notin Prefix(w)$, node $[x]_w$ of $T_D(w)$ is branching (has at least two children).*

Proof. Since $\overleftarrow{x}^w \notin Prefix(w)$, there exist some distinct strings $u, v \in D$ such that

- both $u\overleftarrow{x}^w$ and $v\overleftarrow{x}^w$ are substrings of w,
- $Endpos_w(u\overleftarrow{x}^w) \cap (Wordpos_D(w) \oplus |u\overleftarrow{x}^w| \ominus 1) \neq \emptyset$,
- $Endpos_w(v\overleftarrow{x}^w) \cap (Wordpos_D(w) \oplus |v\overleftarrow{x}^w| \ominus 1) \neq \emptyset$, and
- $Endpos_w(u\overleftarrow{x}^w) \cap (Wordpos_D(w) \oplus |u\overleftarrow{x}^w| \ominus 1) \neq$
  $Endpos_w(v\overleftarrow{x}^w) \cap (Wordpos_D(w) \oplus |v\overleftarrow{x}^w| \ominus 1)$.

Then, no two strings of $u\overleftarrow{x}^w$, $v\overleftarrow{x}^w$, or $\overleftarrow{x}^w$ belong to the same end-equivalence class. By Proposition 2 and Definition 3, the suffix links of $[u\overleftarrow{x}^w]_w$ and $[v\overleftarrow{x}^w]_w$ both go to $[\overleftarrow{x}^w]_w = [x]_w$. □

Now we show the upper bound of the size of SDAWGs, based on a similar idea to the case of DAWGs by Blumer et al. [11].

Theorem 1. *For any string $w \in D^+$ of length n, $SDAWG_D(w)$ has $O(n)$ nodes and edges.*

Proof. By Lemmas 1 and 2, $T_D(w)$ can have at most $n+1$ leaves which correspond to the nodes having the prefixes of w. Since $T_D(w)$ is a tree, it can have at most n branching nodes, and therefore the total number of nodes in $SDAWG_D(w)$ is bounded by $O(n)$.

Now we bound the number of edges in $SDAWG_D(w)$. It is not difficult to see that for any $w \in D^+$, $SDAWG_D(w)$ has a spanning tree rooted at the source node $[\varepsilon]_w$, and let us fucus on one such spanning tree. With each edge of $SDAWG_D(w)$ *not* in the spanning tree, we associate one of the $k-1$ non-empty proper suffixes of $Suffix_D(w)$. This suffix can be obtained by spelling out a path from the source through the spanning tree until one of its leaves, across the omitted edge, and finally to the sink node $[w]_w$ in any convenient way. Then, distinct omitted edges are associated with distinct non-empty suffixes of $Suffix_D(w)$, since they are associated with distinct source-to-sink paths (the paths differ in the first edge traversed outside the spanning tree). Thus, the number of edges not in the spanning tree is bounded by $k-1$, and the total number of edges in $SDAWG_D(w)$ by $O(n)$. □

One might concern that Theorem 1 suggests a disadvantage of the SDAWGs against the sparse (word) suffix trees which have only $O(k)$ nodes and edges, but we recall that the edge labels of those suffix trees are implemented as pairs of pointers to the positions of w. To do so, the input string w has to be kept stored and therefore the total space requirement for using those suffix trees is also $O(n)$. On the other hand, any edge label of SDAWGs is a single symbol from $\Sigma \cup \{\#\}$, and therefore the input string w can be discarded after $SDAWG_D(w)$ is constructed.

3.3 On-Line Linear-Time Construction Algorithm

In this section we present our on-line linear-time construction algorithm for SDAWGs. Since our algorithm is on-line, it sequentially processes the input string $w \in D^+$ from left to right. To discuss this on-line construction, we extend the definition of $Suffix_D(u)$ to any prefix u of $w \in D^+$, as follows. For any prefix u of $w = w_1 \dots w_k \in D^+$ such that $u = w_1 \cdots w_\ell v$, $1 \le \ell < k$, and v is a proper prefix of $w_{\ell+1}$, let $u_i = w_i \cdots w_\ell v$. For convenience, let $u_{\ell+1} = v$ and $u_{\ell+2} = \varepsilon$. Now, let

$$Suffix_D(u) = \{u_i \mid 1 \le i \le \ell + 2\}.$$

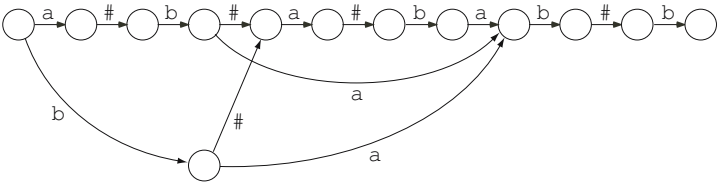


Fig. 3. The SDAWG for string a#b#a#bab#b w.r.t. $D = \Sigma^*\#$. Compare this with the SDAWG for string a#b#a#bab# w.r.t. D in Fig. 1 (upper).

Then, the definitions of $Wordpos_D(u)$, the end-equivalence relation $\equiv_u$, and $SDAWG_D(u)$ are naturally extended to any prefix u of w.

The following proposition and lemma state how to update the nodes of $SDAWG_D(u)$ when we read a new symbol a and construct $SDAWG_D(ua)$.

Proposition 3. *Let $w \in D^+$ and $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Then,*

$$Wordpos_D(ua) = \begin{cases} Wordpos_D(u) \cup \{|ua|\}, & \text{if } u[|u|] = \#; \\ Wordpos_D(u), & \text{otherwise.} \end{cases}$$

Also, for any string $x \in (\Sigma \cup \{\#\})^*$,

$$Endpos_{ua}(x) = \begin{cases} Endpos_u(x) \cup \{|ua|\}, & \text{if } x \in Suffix(ua); \\ Endpos_u(x), & \text{otherwise.} \end{cases}$$

Lemma 3. *Let $w \in D^+$, and let $u, ua \in Prefix(w)$ with $a \in \Sigma \cup \{\#\}$. Let z be the longest string in $Suffix_D(ua) \cap Prefix(Suffix_D(u))$. Then, for any $x \in Prefix(Suffix_D(u))$, we have*

$$[x]_u = \begin{cases} [\overleftarrow{x}^u]_{ua} \cup [z]_{ua}, & \text{if } z \in [x]_u \text{ and } z \neq \overleftarrow{x}^u; \\ [x]_{ua}, & \text{otherwise.} \end{cases}$$

Proof (Sketch). By Proposition 3, it is not difficult to see that only if $z \in [x]_u$ and $z \neq \overleftarrow{x}^u$, it happens that $[x]_u \neq [x]_{ua}$. In any other cases, we have $[x]_u = [x]_{ua}$. By Proposition 3, for any string $s \in [x]_u$ with $|s| > |z|$, $Endpos_{ua}(s) = Endpos_u(s)$, and for any string $t \in [x]_u$ with $|t| \leq |z|$, $Endpos_{ua}(t) = Endpos_u(t) \cup \{|ua|\}$. No matter if $Wordpos_D(ua) = Wordpos_D(u) \cup \{|ua|\}$ or $Wordpos_D(ua) = Wordpos_D(u)$, we have $[x]_u = [\overleftarrow{x}^u]_{ua} \cup [z]_{ua}$. □

To see a concrete example of the above lemma, compare $SDAWG_D(u)$ of Fig. 1 (upper) and $SDAWG_D(u\mathtt{b})$ of Fig. 3, where $u =$ a#b#a#bab#. Observe that $z =$ b. Now, node $[\mathtt{a\#b}]_u$ of $SDAWG_D(u)$ is split when it is updated to $SDAWG_D(u\mathtt{b})$, as follows:

$$[\mathtt{a\#b}]_u = \{\mathtt{a\#b}, \mathtt{b}\} = \{\mathtt{a\#b}\} \cup \{\mathtt{b}\} = [\mathtt{a\#b}]_{u\mathtt{b}} \cup [\mathtt{b}]_{u\mathtt{b}}.$$

For any other nodes $[x]_u$, we have $[x]_u = [x]_{u\mathtt{b}}$.

```
Input:   w = w[1..n] ∈ D^+ and M_D with initial state q_s and final state q_f.
Output:  SDAWG_D(w).
{
    length(q_f) = 0;   length(q_s) = −1;
    source = q_f;   link(source) = q_s;
    sink = source;
    for (i = 1; i ≤ n; i++)  sink = Update(sink, i);
}

node Update(sink, i) {
    c = w[i];
    create new node newsink;   length(newsink) = i;
    create new edge (sink, c, newsink);
    for (s = link(sink); no c-edge from s; s = link(s))
        create new edge (s, c, newsink);
    s' = SplitNode(s, c);
    link(newsink) = s';
    return newsink;
}

node SplitNode(s, c) {
    let s' be the head of the c-edge from s;
    if (length(s') == length(s) + 1) return s';
    create node r' as a duplication of s' with the out-going edges;
    link(r') = link(s');   link(s') = r';
    length(r') = length(s) + 1;
    do {
        replace edge (s, c, s') by edge (s, c, r');
        s = link(s);
    } while the head of the c-edge from s is s';
    return r';
}
```

Fig. 4. SDAWG construction algorithm. For any node v, $link(v)$ indicates the node to which the suffix link of v goes. Only the initialization steps using M_D is different from the normal DAWG construction algorithm by Blumer et al. [11].

Fig. 4 shows a pseudo code of our on-line algorithm to build SDAWGs, with the help of the DFA M_D and the suffix links of Definition 3. The only difference between our algorithm and the algorithm of Blumer et al. [11] for constructing normal DAWGs is the initialization steps of the main routine where we set the source of the SDAWG to the final state q_f of M_D and the suffix link of the source to the initial state q_s of M_D. These simple modifications make a difference in the resulting data structures. In Fig. 5 we illustrate on-line construction of $SDAWG_D(w)$ with $w = \texttt{ab\#b\#ba\#}$ and $D = \{\texttt{a}, \texttt{b}\}\#$.

We remark that our algorithm generalizes the normal DAWG construction algorithm of Blumer et al. [11]. Assume just for now $D = \Sigma$, and consider a DFA which accepts Σ with only two states that are a single initial state and a

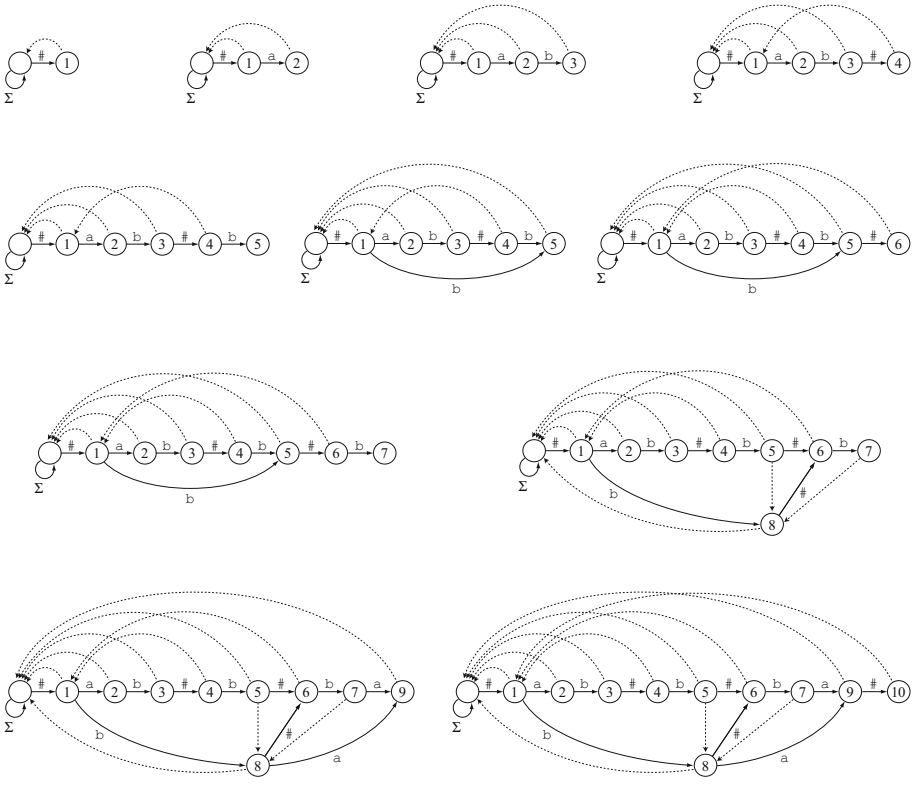


Fig. 5. A snapshot of on-line construction of $SDAWG_D(w)$ with $w = \mathtt{ab\#b\#ba\#}$ and $D = \{\mathtt{a}, \mathtt{b}\}\#$. The broken arrows represent suffix links. The update from $SDAWG_D(\mathtt{ab\#})$ to $SDAWG_D(\mathtt{ab\#b})$ is shown in two rounds, as two new edges are created here. Also, the update from $SDAWG_D(\mathtt{ab\#b\#})$ to $SDAWG_D(\mathtt{ab\#b\#b})$ is shown in two rounds, as a node is here split into two nodes.

single final state. Then this DFA plays the same role as the auxiliary '$\perp$' node used in Ukkonen's on-line suffix tree construction algorithm [12], and this alters our algorithm so that it builds normal DAWGs.

Theorem 2. *The algorithm of Fig. 4 correctly constructs $SDAWG_D(w)$ for any string $w \in D^+$.*

To establish the above correctness theorem, we show the following claim:

Claim. Let $w \in D^+$ and $w_1, \ldots, w_k$ be a unique factorization of w w.r.t. D. Let $u = w_1 \cdots w_\ell v$ be the prefix of w of length j, where v is a proper prefix of $w_{\ell+1}$. After the j-th call of the *Update* function, we have $SDAWG_D(u)$ representing $Suffix_D(u)$ together with the suffix links of Definition 3.

Proof. By induction on $j = |u|$. When $|u| = 0$, the lemma trivially holds. We now consider $|u| > 0$. Let $u_i = w_i \cdots w_\ell v$ for $1 \leq i \leq \ell$, and for convenience, let

$u_{\ell+1} = v$, $u_{\ell+2} = \varepsilon$ and $u_{\ell+3} = c^{-1}$. For the induction hypothesis, assume that, after the j-th call of the *Update* function, we have $SDAWG_D(u)$ representing

$$Suffix_D(u) = \{u_i \mid 1 \leq i \leq \ell + 2\}.$$

At the $(j+1)$-th call of *Update*, due to Lemmas 1 and 3, $sink = [u]_u = [u]_{uc}$ and thus $newsink$ is created as node $[uc]_{uc}$, together with edge $(sink, c, newsink) = ([u]_{uc}, c, [uc]_{uc})$. Now let h $(1 < h \leq \ell + 3)$ be the smallest integer satisfying $w_h \cdots w_\ell vc = u_h c \in Prefix(Suffix_D(u))$. Note that such h always exists, since $u_{\ell+3}c = c^{-1}c = \varepsilon$ is always in $Prefix(Suffix_D(u))$. Then, $u_h c$ is the longest element of $Suffix_D(uc)$ represented by $SDAWG_D(u)$. In the iteration of the **for** loop, we traverse the suffix links starting from $sink$, each time creating edge $([u_i]_{uc}, c, [uc]_{uc})$ for $i = 2, \ldots, h-1$. (Note that for some consecutive i's, the strings u_i may belong to the same end-equivalence class under u, and in this case only one edge is created for all such consecutive i's.) This is justified by the definition of the end-equivalence relation, as we do have $u_i c \equiv_{uc} uc$ for all $i = 1, \ldots, h-1$. Hence, the current DAG represents $Suffix_D(uc)$.

For the suffix link of $newsink$, there are two possible cases to happen:

- When $u_h c = \overleftarrow{u_h c}^{u}$. In this case, $([u_h]_u, c, [u_h c]_u)$ is a primary edge. Due to Lemma 3, we have $[u_h c]_u = [u_h c]_{uc}$ and the suffix link from $newsink = [uc]_{uc}$ is set to node $[u_h c]_{uc}$. This operation is justified by Definition 3.
- When $u_h c \neq \overleftarrow{u_h c}^{u}$. In this case, $([u_h]_u, c, [u_h c]_u)$ is a secondary edge. Due to Lemma 3, we have $[u_h c]_u = [y u_h c]_{uc} \cup [u_h c]_{u_h c}$ where $\overleftarrow{u_h c}^{u} = y u_h c$ and $y \in D^+$. This is done by the function *SplitNode*, and the suffix link of $[u_h c]_{uc}$ is set to $link([u_h c]_u)$, and that of $[y u_h c]_{uc}$ is set to $[u_h c]_{uc}$. Then, the suffix link of $newsink = [uc]_{uc}$ is also set to node $[u_h c]_{uc}$. These operations are justified by Definition 3.

Judging whether $u_h c = \overleftarrow{u_h c}^{u}$ or not, namely, whether edge $([u_h]_u, c, [u_h c]_u)$ is primary or secondary, is done by the **if** condition in *SplitNode* checking the lengths of the nodes $[u_h]_u$ and $[u_h c]_u$. The resulting structure is $SDAWG_D(uc)$ with its suffix links. □

Now the only remaining matter is the time complexity of the algorithm.

Let $u, ua \in Prefix(w)$ with $w \in D^+$ and $a \in \Sigma \cup \{\#\}$. For any $x \in Prefix(Suffix_D(u))$ with $\overleftarrow{x}^{u} = x$, let $SC_u(x)$ be the list of nodes contained in the suffix-link path from node $[x]_u$ to the root of $T_D(u)$. We can establish the following lemma, similarly to [11].

Lemma 4. *Let $u \in Prefix(w)$ with $w \in D^+$. Assume that there is a primary edge $([x]_u, a, [xa]_u)$ in $SDAWG_D(u)$, with $\overleftarrow{x}^{u} = x$ and $\overleftarrow{xa}^{u} = xa$. Then $|SC_u(xa)| = |SC_u(x)| - m + 1$ where m is the number of secondary edges from nodes in $SC_u(x)$ to nodes in $SC_u(xa)$.*

We are ready to prove the following theorem, based on a similar idea to [11].

Theorem 3. *The execution time of the algorithm of Fig. 4 is linear in the input string length.*

Proof. Let $w \in D^+$ be the input string and let $u, ua \in \mathit{Prefix}(w)$ with $a \in \Sigma \cup \{\#\}$. Consider a single call to *Update* creating new node *newsink*, where $\mathit{sink} = [u]_u$ and $\mathit{newsink} = [ua]_{ua}$. Let l be the total number of iterations by the **for** and **do while** loops, except for the first execution of the **do while** loop that generates the primary edge from node s to r'. In any other iteration of either of these loops, a secondary edge is created from a node in $SC_{ua}(u)$ to either $[ua]_{ua}$ or $[z]_{ua}$, where z is the longest string in $\mathit{Suffix}_D(ua) \cap \mathit{Prefix}(\mathit{Suffix}_D(u))$. Since $z \in \mathit{Suffix}_D(ua)$, we have $[z]_{ua} \in SC_{ua}(ua)$. Therefore, by Lemma 4, we have $|SC_{ua}(ua)| \leq |SC_{ua}(u)| - l + 1$.

Moreover, consider the special case where $z \in [u]_u$ and $z \neq \overleftarrow{u}^u$. Recall that the occurrence of z as a suffix of ua immediately follows # in ua, namely, $ua[|ua| - |z|] = \#$. Since now $z \in \mathit{Suffix}_D(u)$, by the periodicity of z, this special case can happen only when $z = \#^{|z|}$. By Lemma 3, node $[u]_u$ is split into two nodes $[u]_{ua}$ and $[z]_{ua}$, increasing $|SC_{ua}(u)|$ by one from $|SC_u(u)|$. Consequently, we obtain $|SC_{ua}(ua)| \leq |SC_{ua}(u)| - l + 2$.

The above formula implies that at each call to *Update*, the suffix chain of *newsink* of $\mathit{SDAWG}_D(ua)$ can grow by at most two from the suffix chain of *sink* of $\mathit{SDAWG}_D(u)$. On the other hand, at each call to *Update*, the length of this suffix chain decreases by l, which is the number of iterations of the **for** and **do while** loops minus one. Note that the length of this suffix chain never gets zero, since at the beginning of the construction, $SC_D(\varepsilon)$ already has the initial state q_s of M_D. Hence, the total number of iterations of these loops when we have processed the entire string w is linear in $|w|$. □

4 Conclusions and Further Work

In this paper we introduced a new data structure $\mathit{SDAWG}_D(w)$ which supports a sparse text indexing of string w w.r.t. dictionary $D = \Sigma^*\#$. Namely, for any string $w = w_1 \cdots w_k$ with $w_i \in D$ for each $1 \leq i \leq k$, $\mathit{SDAWG}_D(w)$ represents the suffixes of w of the form $w_i \cdots w_k$. A typical application to SDAWGs is word- and phrase-level search on texts written in natural languages such as the European languages, where the blank character can be regarded as #. Also, by using SDAWGs, the sparse text indexing problem stated in Section 1 is solvable in time proportional to the pattern length. Further, we showed that $\mathit{SDAWG}_D(w)$ has $O(n)$ nodes and edges, where $n = |w|$, and finally we presented an on-line algorithm that constructs $\mathit{SDAWG}_D(w)$ in $O(n)$ time and space.

Our future work includes the followings: The first one is to show the exact, tight bound on the size of SDAWGs. Blumer et al. [11] showed that for any string $w \in \Sigma^*$, $\mathit{DAWG}(w)$ has at most $2n - 1$ nodes and $3n - 4$ edges, where $n = |w|$. Since SDAWGs are a sparse version of DAWGs, SDAWGs should have strictly less nodes and edges, but this has to be explored in more details.

The second one is to extend this work to compact directed acyclic word graphs (CDAWGs) [13]. The idea of using the minimum DFA M_D is applicable to the

on-line construction algorithm for CDAWGs [14], yielding a sparse text indexing version of CDAWGs. We will then need to define this new data structure, show its size bound, and prove that the modified algorithm correctly constructs the desired structure in linear time.

References

1. Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory. (1973) 1–11
2. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
4. Kärkkänen, J., Ukkonen, E.: Sparse suffix trees. In: Proc. 2nd International Computing and Combinatorics Conference (COCOON'96). Volume 1090 of Lecture Notes in Computer Science., Springer-Verlag (1996) 219–230
5. Inenaga, S., Kivioja, T., Mäkinen, V.: Finding missing patterns. In: Proc. 4th Workshop on Algorithms in Bioinformatics (WABI'04). Volume 3240 of Lecture Notes in Bioinformatics., Springer-Verlag (2004) 463–474
6. Bannai, H., Hyyrö, H., Shinohara, A., Takeda, M., Nakai, K., Miyano, S.: An $O(N^2)$ algorithm for discovering optimal boolean pattern pairs. IEEE/ACM Transactions on Computational Biology and Bioinformatics **1** (2004) 159–170
7. Inenaga, S., Bannai, H., Hyyrö, H., Shinohara, A., Takeda, M., Nakai, K., Miyano, S.: Finding optimal pairs of cooperative and competing patterns with bounded distance. In: Proc. 7th International Conference on Discovery Science (DS'04). Volume 3245 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2004) 32–46
8. Baeza-Yates, R., Gonnet, G.H.: Efficient text searching of regular expressions. In: Proc. 16th International Colloquium on Automata, Languages and Programming (ICALP'89). Volume 372 of Lecture Notes in Computer Science., Springer-Verlag (1989) 46–62
9. Andersson, A., Larsson, N.J., Swanson, K.: Suffix trees on words. Algorithmica **23** (1999) 246–260
10. Inenaga, S., Takeda, M.: On-line linear-time construction of word suffix trees. In: Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM'06). Lecture Notes in Computer Science, Springer-Verlag (2006) To appear.
11. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science **40** (1985) 31–55
12. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260
13. Blumer, A., Blumer, J., Haussler, D., McConnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM **34** (1987) 578–595
14. Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line construction of compact directed acyclic word graphs. Discrete Applied Mathematics **146** (2005) 156–179

On-Line Repetition Detection

Jin-Ju Hong and Gen-Huey Chen*

Department of Computer Science and Information Engineering
National Taiwan University
ghchen@csie.ntu.edu.tw

Abstract. A q-repetition is the concatenation of q copies of a primitive string, where $q \geq 2$. Given a string S character by character, the on-line repetition detection problem is to determine whether S contains a q-repetition in an on-line manner. For $q = 2$, the problem can be solved in $O(m \log \beta)$ time, where m is the ending position of the first 2-repetition and β is the number of distinct characters in the m-th prefix of S. In this paper, we present an on-line algorithm that can detect a q-repetition for $q \geq 3$ with the same time complexity.

1 Introduction

A *repetition* is the concatenation of two or more copies of a string. Detecting repetitions in a string is a fundamental problem in many areas such as combinatorics [16, 14], automata and formal language theory [10, 18], data compression [7], bioinformatics [9], *etc.* Several algorithms that can detect repetitions in a string of length n in $O(n \log n)$ time can be found in the literature [1, 3, 4, 5, 15, 19].

A string is *primitive* if it is not a repetition. A *q-repetition* is the concatenation of q copies of a primitive string, where $q \geq 2$. The *on-line repetition detection problem*, which is the on-line version of the repetition detection problem, reads the input string S character by character and detects a q-repetition as soon as it is completely read.

Let $S[i]$ denote the i-th character of a string S and $S[i..j]$ denote the substring of S from the i-th character to the j-th character, where $i \leq j$. If $i > j$, $S[i..j]$ is considered empty. Let $|S|$ denote the length of S. Suppose that $S[i..m]$ is the first q-repetition in S ($m = |S|$, if there is no q-repetition contained in S). Leung, Peng and Ting [13] first solved the on-line repetition detection problem for $q = 2$ with $O(m \log^2 m)$ time. Later, Chen, Hong and Lu [2] improved the time complexity to $O(m \log \beta)$, where β is the number of distinct characters in $S[1..m]$. In this paper, we further solve the problem for $q \geq 3$ with the same time complexity.

2 Preliminaries

In this section, some definitions, notations and lemmas are introduced, which will be used in the proposed algorithm.

* Corresponding author.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 74–85, 2006.

2.1 Periods

A string S has a period of length p if $S[i] = S[j]$ for every pair of i and j satisfying $i \equiv j \pmod{p}$, where $1 \leq p \leq |S|$ and $S[1..p]$ is a *period* of S. Notice that S might have several periods, and S itself is a period of S. Let $\pi(S)$ denote the length of the smallest. For example, if $S = \texttt{abcabcab}$, then p can be 3 or 6 or 8 and so $\pi(S) = 3$. The following lemma is immediate.

Lemma 1. *p is a period length of S if and only if $S[1..(n-p)] = S[(1+p)..n]$, where $1 \leq p \leq n = |S|$.*

The following lemma was proved in [8].

Lemma 2. [8] *Suppose that p_1 and p_2 are two period lengths of S. If $|S| \geq p_1 + p_2 - \gcd(p_1, p_2)$, then $\gcd(p_1, p_2)$ is also a period length of S.*

Lemma 3. *Let S' be a substring of S. If $|S'| \geq 2 \times \pi(S)$, then $\pi(S') = \pi(S)$.*

Proof. Let $p = \pi(S)$ and $p' = \pi(S')$. Since S' is a substring of S, we have $p' \leq p$. If $p' < p$, then $|S'| \geq 2p > p + p' - \gcd(p, p')$. By Lemma 2, $\gcd(p, p')$ is also a period length of S'. It follows that $p' \mid p$ (i.e., p' divides p) and $p' = \gcd(p, p')$. Without loss of generality, assume $S' = S[x..y]$. For any $1 \leq i \leq j \leq |S|$ with $i \equiv j \pmod{p'}$, there are $x \leq i' \leq j' \leq y$ such that $i' \equiv i \pmod{p}$ and $j' \equiv j \pmod{p}$. Now that $p' \mid p$, we have $i' \equiv j' \pmod{p'}$, which implies $S[i] = S[i'] = S[j'] = S[j]$. Hence, p' is also a period length of S, which is a contradiction. □

According to the definition of q-repetition and Lemma 2, it is not difficult to see that S is a q-repetition if and only if $|S| = q \times \pi(S)$. The following lemma was proved in [6].

Lemma 4. [6] *If $\pi(S[1..2p_1]) = p_1$, $\pi(S[1..2p_2]) = p_2$ and $\pi(S[1..2p_3]) = p_3$, where $p_1 < p_2 < p_3$, then $p_1 + p_2 \leq p_3$.*

Lemma 5. *Suppose that $\pi(S[1..2p_i]) = p_i$ for $1 \leq i \leq t$, where $p_1 < p_2 < \cdots < p_t$. Then, $p_1 + p_2 + \cdots + p_t < \frac{3}{2} \times |S|$.*

Proof. Clearly, this lemma holds for $t \leq 2$. When $t \geq 3$,

$$\begin{aligned} 2 \times \sum_{1 \leq i \leq t} p_i &= p_1 + \left(\sum_{1 \leq i \leq t-2} (p_i + p_{i+1}) \right) + p_{t-1} + 2p_t \\ &\leq p_1 + \left(\sum_{1 \leq i \leq t-2} p_{i+2} \right) + p_{t-1} + 2p_t \qquad \text{(by Lemma 4)} \\ &= \left(\sum_{1 \leq i \leq t} p_i \right) - p_2 + p_{t-1} + 2p_t. \end{aligned}$$

Therefore, $\sum_{1 \leq i \leq t} p_i \leq 2p_t + p_{t-1} - p_2 < 3p_t \leq \frac{3}{2} \times |S|$. □

The following lemma can be proved similarly.

Lemma 6. *Suppose that* $\pi(S[(n-2p_i+1)..n]) = p_i$ *for* $1 \le i \le t$, *where* $p_1 < p_2 < \cdots < p_t$ *and* $n \ge 2p_t$. *Then,* $p_1 + p_2 + \cdots + p_t < \frac{3}{2} \times n$.

2.2 Longest Common Prefix and Longest Common Suffix

Let $\ell^*_{pre}(S : i_1, i_2, i_3)$ denote the length of the longest common prefix of $S[i_1..i_3]$ and $S[i_2..i_3]$, where $1 \le i_1 \le i_2 \le i_3 \le |S|$. For example, if $S =$ `ababbabab`, then $\ell^*_{pre}(S : 3, 6, 8) = 2$. $\ell^*_{pre}(S : i_1, i_2, |S|)$ was also called *longest common extension* in [11, 12]. Main and Lorentz [15] used the longest common prefix as a tool to find repetitions In the following, we show a linear-time processing that reads and processes $S[1], S[2], \ldots, S[i], \ldots$ in this sequence so that after $S[i]$ is processed, each $\ell^*_{pre}(S : 1, i_2, i_3)$ can be determined in constant time, where $1 \le i_2 \le i_3 \le i \le |S|$.

The processing for $S[i]$ computes $\pi(S[1..i])$ and $\ell^*_{pre}(S : 1, j, i)$ for all $\pi(S[1..(i-1)]) + 1 \le j \le \pi(S[1..i])$, and stores the computed values of $\pi(S[1..i])$ and $\ell^*_{pre}(S : 1, j, i)$ in $A[i]$ and $B[j]$, respectively, where A and B are two arrays. Clearly, $\pi(S[1..1]) = 1$ and $\pi(S[1..(i-1)]) \le \pi(S[1..i])$, for all $1 < i \le |S|$.

Lemma 7. *After* $S[i]$ *is processed, each* $\ell^*_{pre}(S : 1, i_2, i_3)$ *can be determined in constant time, where* $1 \le i_2 \le i_3 \le i \le |S|$.

Proof. Clearly, $\ell^*_{pre}(S : 1, i_2, i_3) = \min\{\ell^*_{pre}(S : 1, i_2, i), |S[i_2..i_3]|\}$. We show how to compute $\ell^*_{pre}(S : 1, i_2, i)$ in constant time as follows.

If $i_2 = 1$, then $\ell^*_{pre}(S : 1, i_2, i) = i$. If $1 < i_2 \le \pi(S[1..i])(= A[i])$, then $\ell^*_{pre}(S : 1, i_2, i) = B[i_2]$ as explained below. Note that $B[i_2]$ contains the value of $\ell^*_{pre}(S : 1, i_2, i')$ computed when $S[i']$ is processed, where $i' \le i$ and $\pi(S[1..(i'-1)])+1 \le i_2 \le \pi(S[1..i'])$. Since $i_2 \le \pi(S[1..i'])$, we have $\ell^*_{pre}(S : 1, i_2, i') < |S[i_2..i']|$. Hence, $\ell^*_{pre}(S : 1, i_2, i) = \ell^*_{pre}(S : 1, i_2, i') = B[i_2]$.

Then, consider the case $\pi(S[1..i]) < i_2 \le |S|$. Let $i'_2 \equiv i_2 \pmod{\pi(S[1..i])}$, where $1 \le i'_2 \le \pi(S[1..i])$. $\ell^*_{pre}(S : 1, i'_2, i)$ can be determined in constant time according to the discussion above. Moreover, $S[i_2..i]$ is a prefix of $S[i'_2..i]$. Hence, $\ell^*_{pre}(S : 1, i_2, i) = \min\{\ell^*_{pre}(S : 1, i'_2, i), |S[i_2..i]|\}$. □

The processing for $S[1]$ needs only to set $A[1] = 1$. For $i > 1$, we determine whether $p = \pi(S[1..i])$ for $p = A[i-1], A[i-1]+1, \ldots$ till $p = \pi(S[1..i])$. For each $p < i$, we compute $\ell^*_{pre}(S : 1, p+1, i)$ by Lemma 7. If $\ell^*_{pre}(S : 1, p+1, i) = |S[(p+1)..i]|$, then we have $\pi(S[1..i]) = p$ and the processing for $S[i]$ stops. Otherwise, set $B[p+1] = \ell^*_{pre}(S : 1, p+1, i)$. If $p = i$, then $\pi(S[1..i]) = i$.

It is not difficult to see that the processing for $S[i]$ takes $O\big(\pi(S[1..i]) - \pi(S[1..(i-1)])\big)$ time. Hence, the processing for $S[1], S[2], \ldots, S[i]$ takes a total of $O(i)$ time. The following lemma results.

Lemma 8. *Suppose that* $S[i_1], S[i_1+1], \ldots, S[j], \ldots$ *are read in this sequence. With an* $O(j - i_1)$*-time processing, each* $\pi(S[i_1..i_3])$ *and each* $\ell^*_{pre}(S : i_1, i_2, i_3)$, *where* $i_1 \le i_2 \le i_3 \le j$, *can be determined in constant time, after reading* $S[i_1], S[i_1+1], \ldots, S[j]$.

We use $\ell^*_{suf}(S : i_1, i_2, i_3)$ to denote the length of the longest common suffix of $S[i_1..i_2]$ and $S[i_1..i_3]$, where $1 \leq i_1 \leq i_2 \leq i_3 \leq |S|$. The following lemma can be proved similar to Lemma 8.

Lemma 9. *Suppose that $S[i_3], S[i_3 - 1], \ldots, S[j], \ldots$ are read in this sequence. With an $O(i_3 - j)$-time processing, each $\pi(S[i_1..i_3])$ and each $\ell^*_{suf}(S : i_1, i_2, i_3)$, where $j \leq i_1 \leq i_2 \leq i_3$, can be determined in constant time, after reading $S[i_3]$, $S[i_3 - 1], \ldots, S[j]$.*

2.3 *f*-Factorization

Let XY denote the concatenation of two strings X and Y. Suppose that $\{F_1, F_2, \ldots, F_t\}$ is a partition of S (i.e., $S = F_1F_2 \cdots F_t$), where each F_i $(1 \leq i \leq t)$ is a nonempty substring of S. Let $S[b_i]$ be the leading character of F_i. Then, $\{F_1, F_2, \ldots, F_t\}$ is the *f-factorization* [5] (or *s-factorization* [4]) of S if and only if $F_1 = S[1]$ and for $k \geq 2$, F_k is the longest prefix of $S[b_k..|S|]$ that is also a prefix of some $S[i..|S|]$, where $1 \leq i \leq b_k - 1$. We let $F_k = S[b_k]$ if no such prefix of $S[b_k..|S|]$ is found. For example, if $S = \texttt{aaabbabbab}$, then $F_1 = \texttt{a}$, $F_2 = \texttt{aa}, F_3 = \texttt{b}, F_4 = \texttt{b}$ and $F_5 = \texttt{abbab}$.

By the aid of Ukkonen's on-line suffix tree construction algorithm [20], it is easy to obtain all f-factorizations of $S[1..1], S[1..2], \ldots, S[1..i]$ on-line in total $O(i \log \alpha)$ time, where $1 \leq i \leq |S|$ and α is the number of distinct characters in $S[1..i]$.

3 The Algorithm

The algorithm reads $S[1], S[2], \ldots, S[i], \ldots$ in this sequence, where $1 \leq i \leq |S|$. When $S[i]$ is read, we determine whether $S[1..i]$ contains a q-repetition or not. If it exists, then the algorithm reports it. The algorithm halts when a q-repetition is detected or the entire S has been read. In the rest of this section, the execution of the i-th iteration (i.e., the execution for $S[i]$) of the algorithm is elaborated.

The execution for $S[1]$ is to set $F_1 = S[1]$ (i.e., $b_1 = 1$). When $i \geq 2$, the f-factorization of $S[1..i]$ is constructed first (by executing Ukkonen's algorithm). Suppose that $\{F_1, F_2, \ldots, F_k\}$ is the f-factorization of $S[1..i]$, where $k \geq 2$. Since $S[1..(i-1)]$ contains no q-repetition and $F_k (= S[b_k..i])$ is either a single character or a substring of $S[1..(i-1)]$, there is no q-repetition in F_k. Hence, we only need to determine whether or not there is a q-repetition in $S[1..i]$ whose leading character precedes $S[b_k]$ and whose ending character is $S[i]$.

Suppose that $S[r..i]$ is such a q-repetition in $S[1..i]$, where $1 \leq r \leq b_k - 1$. It is classified into one of the following four types.

- Type-A, if $b_{k-1} \leq r \leq b_k - 1$ and $|F_k| \leq \pi(S[r..i])$;
- Type-B, if $b_{k-1} \leq r \leq b_k - 1$ and $\pi(S[r..i]) < |F_k| < 2 \times \pi(S[r..i])$;
- Type-C, if $b_{k-1} \leq r \leq b_k - 1$ and $|F_k| \geq 2 \times \pi(S[r..i])$;
- Type-D, if $r < b_{k-1}$.

In the following, an $O(|F_{k-1}|)$-time preprocessing is first described. Then, the detection of a q-repetition is detailed, according to the four types.

3.1 Preprocessing

The preprocessing is performed when the leading character of F_k is read (i.e., when $i = b_k$). The purpose of the preprocessing is to compute sets P_j^A and P_j^B for all $b_k \le j < b_k + 2 \times |F_{k-1}|$. An integer $p \in P_j^A$ (P_j^B) if and only if there is a string W of length j such that $W[1..(b_k-1)] = F_1F_2 \cdots F_{k-1}$, $W[(j-p\times q+1)..j]$ is a type-A (type-B) q-repetition, but $W[(j-p\times q)..(j-1)]$ is not a type-A (type-B) q-repetition. Intuitively, $p \in P_j^A$ (P_j^B) means that there is a possible type-A (type-B) q-repetition of length $p \times q$ that ends at the j-th position, considering $S[1], S[2], \ldots, S[b_k - 1]$. The preprocessing consists of the following three steps.

Step P1: Compute $\pi(S[i_2..(b_k - 1)])$ and $\ell^*_{suf}(S : b_{k-1}, i_2, b_k - 1)$ for all $b_{k-1} \le i_2 \le b_k - 1$; store the value of $\ell^*_{suf}(S : b_{k-1}, i_2, b_k - 1)$ in $C[i_2]$, where C is an array.

Step P2: Set P_j^A and P_j^B to be empty, where $b_k \le j < b_k + 2 \times |F_{k-1}|$.

Step P3: For each integer $1 \le p < |F_{k-1}|$ satisfying that $S[(b_k - p)..(b_k - 1)]$ is primitive, if $p\times(q-1) \le \lambda \le p\times q-1$, then insert p into $P^A_{b_k-1+p\times q-\lambda}$; if $p \times (q-2) < \lambda < p \times (q-1)$, then insert p into $P^B_{b_k-1+p\times q-\lambda}$, where $\lambda = \ell^*_{suf}(S : b_{k-1}, b_k - 1 - p, b_k - 1) + p$.

In order to verify the correctness of Step P3, it suffices to show that $S\,[(b_k - p)..(b_k - 1)]$ is primitive and $p\times(q-1) \le \lambda \le p\times q-1$ $\big(p\times(q-2) < \lambda < p\times(q-1)\big)$ if and only if there is a string W of length j such that $W[1..(b_k-1)] = F_1F_2 \cdots F_{k-1}$, $W[(j - p \times q + 1)..j]$ is a type-A (type-B) q-repetition, but $W[(j - p \times q)..(j - 1)]$ is not a type-A (type-B) q-repetition, where $b_k \le j < b_k + 2 \times |F_{k-1}|$.

First, the case of $P^A_{b_k-1+p\times q-\lambda}$ is considered. Suppose that $S[(b_k-p)..(b_k-1)]$ is primitive and $p\times(q-1) \le \lambda \le p\times q-1$. Let $j = b_k-1+p\times q-\lambda$. The string W is determined to be the concatenation of $F_1F_2 \cdots F_{k-1}$ and $S[(b_k - p)..(j - p)]$, where $|W| = j$. Refer to Figure 1, where a type-A 4-repetition is shown and $p \in P_j^A$. By Lemma 1, p is a period length of $W[(b_k - \lambda)..(b_k - 1)]$, where $b_k - \lambda = j - p \times q + 1$. Since $W[b_k..j] = S[(b_k-p)..(j-p)] = W[(b_k-p)..(j-p)]$, p is also a period length of $W[(j - p \times q + 1)..j]$.

Assume for a contradiction that $\pi(W[(j - p \times q + 1)..j]) < p$. By Lemma 2, we have $\pi(W[(j - p \times q + 1)..j]) = \gcd(\pi(W[(j - p \times q + 1)..j]), p)$ (i.e., $\pi(W\,[(j - p \times q + 1)..j]) \mid p$). Further, by Lemma 3, $\pi(W[(b_k - p)..(b_k - 1)]) = \pi(W[(j-p\times q+1)..j])$, which contradicts to that $S[(b_k-p)..(b_k-1)]$ is primitive. Hence, $\pi(W[(j - p \times q + 1)..j]) = p$ and $W[(j - p \times q + 1)..j]$ is a q-repetition.

Since $b_{k-1} \le j-p\times q+1 \le b_k-1$ and $|W[b_k..j]| = j - b_k + 1 = p \times q - \lambda \le p$, $W[(j - p \times q + 1)..j]$ is a type-A q-repetition. Also, since either $W[j - p \times q] \ne W[j - p \times q + p]$ or $j - p \times q < b_{k-1}$, $W[(j - p \times q)..(j - 1)]$ is not a type-A q-repetition.

Conversely, suppose that there is a string W of length j, where $b_k \le j < b_k + 2 \times |F_{k-1}|$, such that $W[1..(b_k-1)] = F_1F_2 \cdots F_{k-1}$, $W[(j-p\times q+1)..j]$ is a type-A q-repetition, but $W[(j-p\times q)..(j-1)]$ is not a type-A q-repetition. Then, $b_{k-1} \le j-p\times q+1 \le b_k-1$, $\pi(W[(j-p\times q+1)..j]) = p$, and $|W[b_k..j]| \le p$. It is not

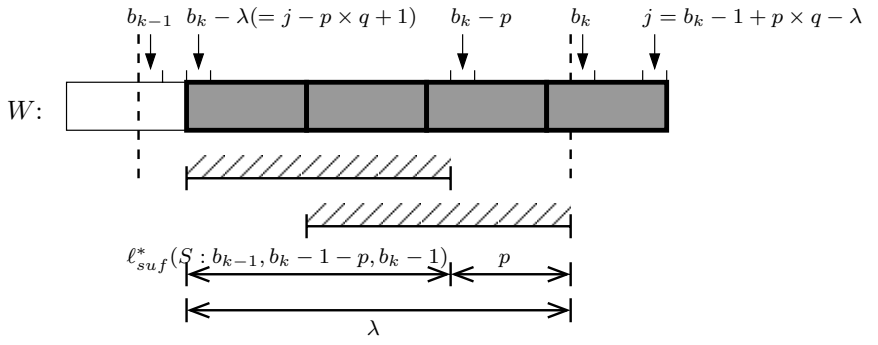


Fig. 1. A type-A 4-repetition and $p \in P_j^A$

difficult to see that $p\times(q-1) \leq \ell^*_{suf}(S : b_{k-1}, b_k-1-p, b_k-1)+p(=\lambda) \leq p\times q-1$ (refer to Figure 1). Since $\pi(W[(j-p\times q+1)..j]) = p$ and $W[(b_k-p)..(b_k-1)]$ is a substring of $W[(j-p\times q+1)..j]$ of length p, $W[(b_k-p)..(b_k-1)]$ is primitive.

Then, the case of $P^B_{b_k-1+p\times q-\lambda}$ is considered, in which $p \times (q-2) < \lambda < p\times(q-1)$. The string W is determined to be the concatenation of $F_1F_2\cdots F_{k-1}$, $S[(b_k-p)..(b_k-1)]$ and $S[(b_k-p)..(j-2p)]$. Refer to Figure 2 for an example of a type-B 4-repetition. The proof is very similar to the proof for $P^A_{b_k-1+p\times q-\lambda}$, and so omitted.

The time requirement of the preprocessing is analyzed below. Since $F_{k-1} = S[b_{k-1}..(b_k-1)]$ is available, Step P1 takes $O(b_k - b_{k-1}) = O(|F_{k-1}|)$ time, by the aid of Lemma 9. By the way, each λ and each $\pi(S[(b_k-p)..(b_k-1)])$, where $1 \leq p < |F_{k-1}|$, can be computed in constant time. Step P2 also takes $O(|F_{k-1}|)$ time. Also note that $S[(b_k-p)..(b_k-1)]$ is primitive if and only if $\pi(S[(b_k-p)..(b_k-1)]) = |S[(b_k-p)..(b_k-1)]|$ or $\pi(S[(b_k-p)..(b_k-1)])$ cannot divide $|S[(b_k-p)..(b_k-1)]|$. Therefore, Step P3 can be completed in $O(|F_{k-1}|)$ time. The following lemma results.

Lemma 10. *The preprocessing, which constructs P_j^A and P_j^B for all $b_k \leq j < b_k + 2 \times |F_{k-1}|$, takes $O(|F_{k-1}|)$ time.*

3.2 Detection of a Type-A q-repetition

It is not difficult to see the following lemma from the discussion above.

Lemma 11. *$S[(i-p\times q+1)..i]$ is a type-A q-repetition if and only if $p \in P_i^A$ and p is a period length of $S[(b_k-p)..i]$.*

According to Lemma 1 and Lemma 11, there is a type-A q-repetition ending at $S[i]$ if and only if $S[(b_k-p)..(i-p)] = S[b_k..i]$ for some $p \in P_i^A$. Consequently, detecting a type-A q-repetition can be accomplished as follows.

Step A1: Determine if $S[(b_k-p)..(i-p)] = S[b_k..i]$ for some $p \in P_i^A$.

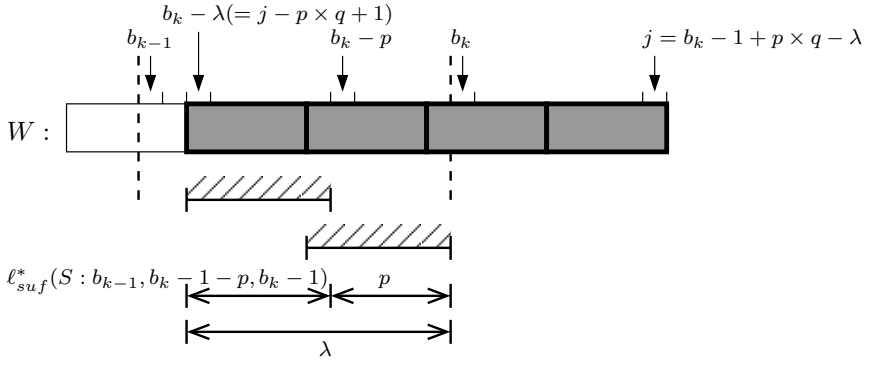


Fig. 2. A type-B 4-repetition and $p \in P_j^B$

If $S[(b_k - p)..(i - p)] = S[b_k..i]$ for some $p \in P_i^A$, then $S[(i - p \times q + 1)..i]$ is a type-A q-repetition. The time requirement is analyzed as follows. For each $p \in P_i^A$, Step A1 performs at most $|S[b_k..i]|$ $(\leq p)$ character comparisons. Since $\ell^*_{suf}(S : b_{k-1}, b_k - 1 - p, b_k - 1) = \lambda - p \geq p \times (q - 1) - p \geq p$, we have $S[(b_k - 2p)..(b_k - 1 - p)] = S[(b_k - p)..(b_k - 1)]$. Further, since $S[(b_k - p)..(b_k - 1)]$ is primitive, $S[(b_k - 2p)..(b_k - 1)]$ is a 2-repetition and $\pi(S[(b_k - 2p)..(b_k - 1)]) = p$.

Since $|S[b_{k-1}..(b_k - 1)]| \geq \lambda \geq p \times (q - 1) \geq 2p$, $S[(b_k - 2p)..(b_k - 1)]$ is a suffix of $F_{k-1} (= S[b_{k-1}..(b_k - 1)])$. According to Lemma 6, we have $\sum_{\substack{p \in P_j^A \\ b_k \leq j \leq i}} p < \frac{3}{2} \times |F_{k-1}|$. That is, detecting a type-A q-repetition whose ending character belongs to $F_k (= S[b_k..i])$ requires at most $\frac{3}{2} \times |F_{k-1}|$ comparisons.

Lemma 12. *Detecting a type-A q-repetition whose ending character belongs to F_k takes $O(|F_{k-1}|)$ time.*

3.3 Detection of a Type-B q-repetition

The following lemma can be proved similar to Lemma 11.

Lemma 13. *$S[(i - p \times q + 1)..i]$ is a type-B q-repetition if and only if $p \in P_i^B$ and p is a period length of $S[(b_k - p)..i]$.*

According to the preprocessing (Step P3), if $p \in P_i^B$, then $p < |S[b_k..i]| < 2p$. Similarly, by Lemma 1 and Lemma 13, there is a type-B q-repetition ending at $S[i]$ if and only if $S[(b_k - p)..(i - p)] = S[b_k..i]$ for some $p \in P_i^B$. It is not difficult to see that the latter holds if and only if the following two conditions hold for some $p \in P_i^B$:

(1) $S[(b_k - p)..(b_k - 1)] = S[b_k..(b_k + p - 1)]$;
(2) $\ell^*_{pre}(S : b_k, b_k + p, i) = |S[b_k..i]| - p$.

Condition (1) requires that $S[b_k..(b_k + p - 1)]$, which is a prefix of F_k, is a suffix of F_{k-1}. Let f_j be the ending position of the leftmost occurrence of $S[b_k..j]$

in F_{k-1}, where $b_k \le j \le i$, i.e., $f_j = \min\{x : S[(x + b_k - j)..x] = S[b_k..j]$ and $b_{k-1} + i - b_k \le x \le b_k - 1\}$. If $S[b_k..j]$ does not occur in F_{k-1}, set $f_j = b_k$. $S[b_k..(b_k + p - 1)]$ is a suffix of F_{k-1} if and only if $f_{b_k+p-1} < b_k$ and $\ell^*_{suf}(S : b_{k-1}, f_{b_k+p-1}, b_k - 1)(= C[f_{b_k+p-1}]) \ge |S[b_k..(b_k + p - 1)]| = p$.

Consequently, there is a type-B q-repetition ending at $S[i]$ if and only if $f_{b_k+p-1} < b_k$, $C[f_{b_k+p-1}] \ge p$ and condition (2) hold for some $p \in P_i^B$. Detecting a type-B q-repetition can be accomplished as follows.

Step B1: Compute f_i and store the value of f_i in $D[i]$, where D is an array.
Step B2: Determine if $f_{b_k+p-1} < b_k$, $C[f_{b_k+p-1}] \ge p$ and $\ell^*_{pre}(S : b_k, b_k + p, i) = |S[b_k..i]| - p$ for some $p \in P_i^B$.

It takes total $O(|F_{k-1}|+|F_k|)$ time for Step B1 to compute f_j for all $b_k \le j \le i$, as elaborated below. If $i = b_k$, then computing f_i needs at most $f_i - b_{k-1} + 1$ character comparisons. If $i > b_k$ and $f_{i-1} = b_k$ (i.e., $S[b_k..(i - 1)]$ does not occur in F_{k-1}), then set $f_i = b_k$ immediately. If $i > b_k$ and $f_{i-1} < b_k$ (i.e., $S[(f_{i-1} - i + b_k + 1)..f_{i-1}]$ is the leftmost occurrence of $S[b_k..(i - 1)]$ in F_{k-1}), then computing f_i is equivalent to finding the leftmost occurrence of $S[b_k..i]$ in $S[(f_{i-1}-i+b_k+1)..(b_k-1)]$. The latter needs at most $2\times(f_i-f_{i-1})-1$ character comparisons, by the aid of the Morris-Pratt algorithm [17], as explained below.

For simplicity, let $X = S[b_k..i]$ and $Y = S[(f_{i-1} - i + b_k + 1)..(b_k - 1)]$. The Morris-Pratt algorithm consists of two phases: a preprocessing phase and a searching phase. The preprocessing phase constructs a table M of $|X|$ entries, where $M[j]$ $(2 \le j \le |X|)$ indicates the length of the longest proper suffix of $X[1..(j - 1)]$ that is also a prefix of $X[1..(j - 1)]$. Then, the searching phase determines the leftmost occurrence of X in Y with at most $2d - |X|$ character comparisons, where $d = |S[(f_{i-1} - i + b_k + 1)..f_i]|$ if X occurs in Y and $d = |S[(f_{i-1} - i + b_k + 1)..f_i]| - 1$ if X does not occur in Y.

In fact, it is not necessary for us to perform the preprocessing phase, because by Lemma 1, each entry of M can be determined in constant time provided $\pi(S[b_k..b_k]), \pi(S[b_k..(b_k+1)]), \ldots, \pi(S[b_k..i])$ are known. According to Lemma 8, each of them can be determined in constant time, if an $O(|S[b_k..i]|)$-time processing was made for $S[b_k], S[b_k + 1], \ldots, S[i]$. Moreover, since $X\,[1..(|X| - 1)] = S[b_k..(i - 1)] = S[(f_{i-1} - i + b_k + 1)..f_{i-1}]$, $|X| - 1$ character comparisons can be saved in the searching phase. The number of character comparisons needed for computing f_i is at most $2d - 2 \times |X| + 1 \le 2 \times |S[(f_{i-1} - i + b_k + 1)..f_i]| - 2 \times |S[b_k..i]| + 1 = 2 \times (f_i - f_{i-1}) - 1$.

The time complexity for Step B1 to compute f_j for all $b_k \le j \le i$ is bounded by

$$
\begin{aligned}
&O(|S[b_k..i]|) + O(f_{b_k} - b_{k-1} + 1) + O\Big(\sum_{b_k<j\le i} (2f_j - 2f_{j-1} - 1)\Big) \\
&\le O(|F_k|) + O(f_i - b_{k-1}) \\
&\le O(|F_k|) + O(|F_{k-1}|).
\end{aligned}
$$

On the other hand, it takes total $O(|F_{k-1}|)$ time for Step B2 to process sets P_j^B for all $b_k \le j \le i$, as explained below. The value of f_{b_k+p-1} was computed

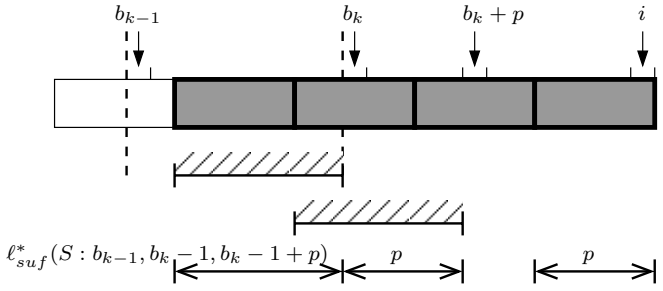


Fig. 3. A type-C 4-repetition

and stored in $D[b_k + p - 1]$ when $S[b_k + p - 1]$ was read. According to Lemma 8, $\ell^*_{pre}(S : b_k, b_k + p, i)$ can be determined in constant time, if an $O(|S[b_k..i]|)$-time processing was made for $S[b_k], S[b_k + 1], \ldots, S[i]$. Therefore, Step B2 takes constant time for each $p \in P_i^B$. According to the preprocessing (Step P3), there are at most $|F_{k-1}| - 1$ integers p contained in all sets P_j^B. Hence, Step B2 takes total $O(|F_{k-1}|)$ time for all sets P_j^B.

Lemma 14. *Detecting a type-B q-repetition whose ending character belongs to* F_k *takes* $O(|F_{k-1}| + |F_k|)$ *time.*

3.4 Detection of a type-C *q*-repetition

Refer to Figure 3, where a type-C 4-repetition is shown, and it is not difficult to see the following lemma.

Lemma 15. $S[(i - p \times q + 1)..i]$ *is a type-C q-repetition if and only if* $p = \pi(F_k) \leq \frac{1}{2} \times |F_k|$ *and* $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = p \times q - |F_k|$.

According to Lemma 15, detecting a type-C q-repetition can be accomplished as follows.

Step C1: If $|S[b_k..i]| = 2p$, then compute $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p)$ and store the value of $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p)$ in $E[i]$, where $p = \pi(S[b_k..i])\,(= \pi(F_k))$ and E is an array.

Step C2: Determine if $|S[b_k..i]| \geq 2p$ and $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = p \times q - |S[b_k..i]|$.

Step C1 is executed only when $|S[b_k..i]| = 2p$. According to Lemma 8, the value of p can be determined in constant time, if an $O(|S[b_k..i]|)$-time processing was made for $S[b_k], S[b_k + 1], \ldots, S[i]$. Then we show that it takes total $O(|S[b_k..i]|)$ time for Step C1 to compute $\ell^*_{suf}\big(S : b_{k-1}, b_k - 1, b_k - 1 + \pi(S[b_k..j])\big)$ (when $S[j]$ is read and $|S[b_k..j]| = 2 \times \pi(S[b_k..j])$) for all $b_k \leq j \leq i$.

Let $\mu = \ell^*_{suf}(S : b_k - p, b_k - 1, b_k - 1 + p) \leq p$, which can be determined with at most p character comparisons. There is a method to calculate $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p)$ in constant time, as described below.

If $\mu \geq |F_{k-1}|$, then
$\quad \ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = |F_{k-1}|$
else if $\mu < p$, then
$\quad \ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = \mu$
else
$\quad \ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = C[b_k - 1 - p] + \mu.$

Notice that $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) \leq |F_{k-1}|$. So, if $\mu \geq |F_{k-1}|$, then $b_k - p \leq b_{k-1}$ and $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = |F_{k-1}|$. Otherwise, $\mu < |F_{k-1}|$ and $b_{k-1} < b_k - \mu$. If $\mu < p$, then $S[b_k - 1 - \mu] \neq S[b_k - 1 + p - \mu]$, which implies $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = \mu$. If $\mu = p (< |F_{k-1}|)$, then $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = \ell^*_{suf}(S : b_{k-1}, b_k - 1 - p, b_k - 1) + \mu$, where the value of $\ell^*_{suf}(S : b_{k-1}, b_k - 1 - p, b_k - 1)$ can be found in $C[b_k - 1 - p]$.

At most $\pi(S[b_k..j])$ character comparisons are needed for Step C1 to compute each $\ell^*_{suf}\big(b_{k-1}, b_k - 1, b_k - 1 + \pi(S[b_k..j])\big)$, where $b_k \leq j \leq i$. By Lemma 5, the total number of character comparisons needed for Step C1 to calculate $\ell^*_{suf}\big(b_{k-1}, b_k - 1, b_k - 1 + \pi(S[b_k..j])\big)$ for all $b_k \leq j \leq i$ is bounded by $\frac{3}{2} \times |S[b_k..i]|$.

For the execution of Step C2, we only need to explain how $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p)$ can be determined in constant time, where $p = \pi(S[b_k..i])$. Let $i' = b_k + 2p - 1$ and $p' = \pi(S[b_k..i'])$. By Lemma 3, $p' = p$, and hence, $\ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p) = \ell^*_{suf}(S : b_{k-1}, b_k - 1, b_k - 1 + p')$. The latter was calculated when $S[i']$ was read, and its value can be found in $E[i']$. The execution of Step C2 for $S[b_k], S[b_k + 1], \ldots, S[i]$ (i.e., $|S[b_k..j]|$ and $p = \pi(S[b_k..j])$ for all $b_k \leq j \leq i$) takes total $O(S[b_k..i])$ time.

Lemma 16. *Detecting a type-C q-repetition whose ending character belongs to F_k takes $O(|F_k|)$ time.*

3.5 Detection of a Type-D *q*-repetition

Since no type-D q-repetition ends in F_1 and F_2, we assume $k \geq 3$. Throughout this section, we consider $\ell^*_{suf}(S : i_1, i_2, i_3) = \ell^*_{suf}(S : 1, i_2, i_3)$ if $i_1 < 1$.

Lemma 17. *$S[(i - p \times q + 1)..i]$ is a type-D q-repetition if and only if $p = \pi(S[b_{k-1}..i])$, $p \times (q - 1) < |S[b_{k-1}..i]| < p \times q$, and $\ell^*_{suf}(S : b_{k-1} - p, b_{k-1} - 1, b_{k-1} - 1 + p) = p \times q - |S[b_{k-1}..i]|$.*

The proof of Lemma 17 is omitted for brevity. Let $g_j = \ell^*_{suf}\big(b_{k-1} - \pi(S[b_{k-1}..j]), b_{k-1} - 1, b_{k-1} - 1 + \pi(S[b_{k-1}..j])\big)$, where $j \geq b_{k-1}$. According to Lemma 17, there is a type-D q-repetition ending at $S[i]$ if and only if $p \times (q-1) < |S[b_{k-1}..i]| < p \times q$ and $g_i = p \times q - |S[b_{k-1}..i]|$, where $p = \pi(S[b_{k-1}..i])$. Consequently, detecting a type-D q-repetition can be accomplished as follows.

Step D1: If $i = b_k$, then compute g_j for all $b_{k-1} \leq j < b_k$ with $|S[b_{k-1}..j]| = 2 \times \pi(S[b_{k-1}..j])$, and store the value of g_j in $G[j]$, where G is an array.

Step D2: If $|S[b_{k-1}..i]| = 2 \times p$, then compute g_i and store the value of g_i in $G[i]$.
Step D3: Determine if $p \times (q-1) < |S[b_{k-1}..i]| < p \times q$ and $g_i = p \times q - |S[b_{k-1}..i]|$.

According to Lemma 8, each $\pi(S[b_{k-1}..j])$, where $b_{k-1} \leq j \leq i$, can be determined in constant time, if an $O(|S[b_{k-1}..i]|)$-time processing was made for $S[b_{k-1}], S[b_{k-1}+1], \ldots, S[i]$. Step D1 is executed only when $i = b_k$. To compute each g_j, Step D1 performs at most $\pi(S[b_{k-1}..j])$ character comparisons. Similarly, Step D2 performs at most $\pi(S[b_{k-1}..i])$ character comparisons. By Lemma 5, the total number of character comparisons needed for Step D1 and Step D2 to compute g_j for all $b_{k-1} \leq j \leq i$ with $|S[b_{k-1}..j]| = 2 \times \pi(S[b_{k-1}..j])$ is bounded by $\frac{3}{2} \times |S[b_{k-1}..i]|$.

In Step D3, the condition $p \times (q-1) < |S[b_{k-1}..i]| < p \times q$ can be determined in constant time. When the condition holds, $|S[b_{k-1}..i]| > 2p$. By Lemma 3, we have $\pi(S[b_{k-1}..(b_{k-1}+2p-1)]) = \pi(S[b_{k-1}..i]) = p$. It follows that $g_i = g_{b_{k-1}+2p-1}$. The value of $g_{b_{k-1}+2p-1}$, which was calculated when $S[b_{k-1}+2p-1]$ was read, can be found in $G[b_{k-1}+2p-1]$. Therefore, the execution of Step D3 for $S[b_k], S[b_k+1], \ldots, S[i]$ takes total $O(|S[b_k..i]|)$ time.

Lemma 18. *Detecting a type-D q-repetition whose ending character belongs to F_k takes $O(|F_{k-1}| + |F_k|)$ time.*

3.6 Time Complexity

Suppose that there is a q-repetition whose ending character is $S[m]$. If no q-repetition occurs, then let $m = |S|$. We further assume that $S[m]$ belongs to F_k, where $k \geq 2$. According to Lemma 10, the preprocessing takes total $\sum_{2 \leq j \leq k} O(|F_{j-1}|) = O(m)$ time. According to Lemmas 12, 14, 16 and 18, detecting a q-repetition in S takes total $\sum_{2 \leq j \leq k} O(|F_{j-1}| + |F_j|) = O(m)$ time. Recall that computing the f-factorization of $S[1..m]$ on-line takes total $O(m \log \beta)$ time, where β is the number of distinct characters in $S[1..m]$. Therefore, the time complexity of the proposed algorithm is $O(m \log \beta)$.

4 Concluding Remarks

There were several off-line algorithms that could detect repetitions in S in $O(|S| \log \alpha)$ time, where α is the number of distinct characters in S. In this paper, we investigated some properties of repetitions and then presented an on-line $O(m \log \beta)$-time algorithm for detecting a q-repetition, where $q \geq 3$. The detection of a 2-repetition, which also takes $O(m \log \beta)$ time, can be found in [2].

One natural problem is how to further reduce the time complexity of detecting a q-repetition. Another interesting problem is to find all repetitions in an on-line manner.

References

1. A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.
2. G.-H. Chen, J.-J. Hong, and H.-I. Lu. An optimal algorithm for online square detection. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 280–287, 2005.
3. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.
4. M. Crochemore. Recherche linéaire d'un carré dans un mot. *Comptes Rendus des Séances de l'Académie des Sciences. Série I. Mathématique*, 296(18):781–784, 1983.
5. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.
6. M. Crochemore and W. Rytter. Squares, cubes, and time—space efficient string searching. *Algorithmica*, 13(5):405–425, 1995.
7. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.
8. N. J. Fine and H. S. Wilf. Uniqueness theorem for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.
9. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.
10. M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.
11. R. M. Kolpakov and G. Kucherov. Finding repeats with fixed gap. In *Proceedings of the 7th International Symposium on String Processing Information Retrieval (SPIRE)*, pages 162–168, 2000.
12. R. M. Kolpakov and G. Kucherov. Finding approximate repetitions under Hamming distance. *Theoretical Computer Science*, 303(1):135–156, 2003.
13. H.-F. Leung, Z. Peng, and H.-F. Ting. An efficient online algorithm for square detection. In *Proceeings of the 10th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 432–439, 2004.
14. M. Lothaire. *Applied Combinatorics on Words.* Cambridge University Press, 2005.
15. M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
16. M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 271–278. Springer-Verlag, 1985.
17. J. H. Morris, Jr. and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.
18. R. J. Ross and K. Winklmann. Repetitive strings are not context-free. *Informatique Théorique et Applications*, 16(3):191–199, 1982.
19. J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002.
20. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

Analyzing User Behavior to Rank Desktop Items

Paul-Alexandru Chirita and Wolfgang Nejdl

L3S Research Center / University of Hanover
Deutscher Pavillon, Expo Plaza 1
30539 Hanover, Germany
{chirita, nejdl}@l3s.de

Abstract. Existing desktop search applications, trying to keep up with the rapidly increasing storage capacities of our hard disks, are an important step towards more efficient personal information management, yet they offer an incomplete solution. While their indexing functionalities in terms of different file types they are able to cope with are impressive, their ranking capabilities are basic, and rely only on textual retrieval measures, comparable to the first generation of web search engines. In this paper we propose to connect semantically related desktop items by exploiting usage analysis information about sequences of accesses to local resources, as well as about each user's local resource organization structures. We investigate and evaluate in detail the possibilities to translate this information into a desktop linkage structure, and we propose several algorithms that exploit these newly created links in order to efficiently rank desktop items. Finally, we empirically show that the access based links lead to ranking results comparable with TFxIDF ranking, and significantly surpass TFxIDF when used in combination with it, making them a very valuable source of input to desktop search ranking algorithms.

1 Introduction

The capacity of our hard-disk drives has increased tremendously over the past decade, and so has the number of files we usually store on our computer. Using this space, it is quite common to have over 100,000 indexable items on the desktop. It is no wonder that sometimes we cannot find a document anymore, even when we know we saved it somewhere. Ironically, in some of these cases nowadays, the document we are looking for can be found faster on the World Wide Web than on our personal computer. In view of these trends, resource organization in personal repositories has received more and more attention during the past years. Thus, several projects have started to explore search and personal information management on the desktop, including Stuff I've Seen [6], Haystack [13], or our Beagle^{++} [4].

Web search has become more efficient than PC search due to the powerful link based ranking solutions like PageRank [12]. The recent arrival of desktop search applications, which index all data on a PC, promises to increase search efficiency on the desktop. However, even with these tools, searching through our (relatively small set of) personal documents is currently inferior to searching the (rather vast set of) documents on the web. Indeed, desktop search engines are now comparable to first generation web search engines, which provided full-text indexing, but only relied on textual information retrieval algorithms to rank their results.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 86–97, 2006.

Desktop ranking is hindered by the lack of links between documents, an important source of evidence for current web ranking algorithms. In this paper we propose to alleviate this deficiency by analyzing user's activity patterns, as well as her local resource organization structures. We investigate and evaluate in detail the possibilities to translate this information into a desktop linkage structure, and we propose several algorithms that exploit these newly created links in order to efficiently rank desktop items. Finally, we empirically show that the access based links lead to ranking results comparable with TFxIDF ranking, and significantly surpass TFxIDF when used in combination with it, making them a very valuable source of input to desktop search ranking algorithms.

The paper is organized as follows: We start with a discussion of the relevant background in Section 2. Then, in Section 3 we present the desktop ranking algorithms we propose and in Section 4 we show our experimental results. Finally, we conclude and discuss further work in Section 5.

2 Relevant Background

Though *ranking* plays an important role on the Web, there is almost no approach specifically aiming at *ranking* desktop search results. More, even though there exist quite a few systems organizing personal information sources and improving information access in these environments, few of the papers describing them concentrate on search algorithms. This section will describe several such systems and discuss their approaches to desktop search.

Several systems have been constructed in order to facilitate re-finding of various stored resources on the desktop. Stuff I've Seen [6] for example provides a unified index of the data that a person has seen on her computer, regardless of its type. Contextual cues such as time, author, thumbnails and previews can be used to search for and present information, but no desktop specific ranking scheme is investigated. Similarly, MyLifeBits [7] targets storing locally all digital media of each person, including documents, images, sounds and videos. They organize these data into collections and, like us, connect related resources with links. However, they do not investigate building desktop ranking algorithms that exploit these links, but rather use them to provide contextual information.

Haystack [1, 9] emphasizes the relationship between a particular individual and her corpus. It is quite similar to our approach in the sense that it automatically creates connections between documents with similar content and it exploits usage analysis to extend the desktop search results set. However, just like the previous articles, it does not investigate the possibilities to *rank* these results, once they have been obtained.

Connections [14] is a very recent system also targeted at enhancing desktop search quality. Similar to us and to Haystack, they also attempt to connect related desktop items, yet they exploit these links using rather complex measures combining BFS and link analysis techniques, which results in rather large search response delays. Nevertheless, while our algorithms are clearly faster, we intend to compare the two approaches in terms of output quality in future work.

Finally, Chirita et al. [3, 4] proposed various activity specific heuristics to generate links between resources. There, our approach was limited to specific desktop contexts (e.g., publications, or web pages), whereas in this paper we explore much more general

sources of linkage information such as file access patterns, which are applicable to *any* desktop resource.

3 Ranking Desktop Resources

Introduction. As the number of indexable items on our desktops (i.e., files that contain any kind of textual information, emails, etc.) can easily exceed 100,000, we can no longer manage them manually just by defining "good" file and directory names and structures. More, the currently employed textual information retrieval measures are no longer sufficient to order the usually several hundreds of results returned for our desktop search queries. We therefore need to investigate more advanced desktop organization paradigms and ranking algorithms. In this section we address the latter issue and propose several algorithms that exploit file access information in order to efficiently rank desktop search results.

Exploiting Usage Analysis to Generate Ranks. Current personal information systems create links between desktop resources only when a very specific desktop usage activity is encountered (e.g., the attachment of an email is saved as a file, or a web page is stored locally, etc.). We argue that in fact in almost all cases when two items are touched in a sequence several times, there will also be a relation between them, irrespective of the underlying user activity. Thus, we propose to add a link between such two items a and b whenever item b is touched after a for the T^{th} time, with T being a threshold set by the user. Higher values for T mean an increased accuracy of the ranking algorithm, at the cost of having a score associated to less resources. Theoretically, there is only a very low probability to have any two items a and b touched in a sequence even once. However, since *context switching* occurs quite often nowadays, we also investigated higher values for T, but experimental results showed them to perform worse than $T = 1$. This is in fact correct, since two files are accessed consequently more often because they are indeed related, than due to a switch of context.

After a short period of time a reputation metric can be computed over the graph resulted from this usage analysis process. There exist several applicable metrics. The most common one is PageRank [12]. On the one hand, it has the advantage of propagating the inferred semantic similarities (connections), i.e., if there is a link between resources a and b, as well as an additional link between resources b and c, then with a relatively high probability we should also have a connection between a and c. On the other hand, PageRank also implies a small additional computational overhead, which is not necessary for a simpler, yet more naïve metric, *in-link count*. According to this latter approach, the files accessed more often get a higher ranking. However, our experiments from Section 4 will show that although it does indeed yield a clear improvement over simple TFxIDF, file access counting is also significantly less effective than PageRank.

Another aspect that needs to be analyzed is the type of links residing on the PC desktop. We use directed links for each sequence $a \rightarrow b$, as when file b is relevant for file a, it does not necessarily mean that the reversed is true as well. Imagine for example that b is a report we are regularly appending, whereas a is the article we are writing. Clearly b is more relevant for a, than a is for b. This yields the following algorithm:

Algorithm 3.1. Ranking Desktop Items.

Pre-processing:
1: Let A be an empty link structure
2: Repeat for ever
3: **If** (File a is accessed at time t_a, File b is accessed at time t_b) AND ($t_a - t_b < \epsilon$),
4: **Then** Add the link $a \rightarrow b$ to A

Ranking:
1: Let A' be an additional, empty link structure
2: For each resource i
3: **For** each resource j linked to i
4: **If** ($\#Links(i \rightarrow j) > T$) in A
5: **Then** Add one link $i \rightarrow j$ to A'
6: Run PageRank using A' as underlying link structure

As it was not clear how many times two resources should be accessed in a sequence in order to infer a "semantic" connection between them, we studied several values for the T threshold, namely one, two and three. Additionally, we also explored the possibilities to directly use the original matrix A with PageRank, thus implicitly giving more weight to links that occurred more frequently (recall that in A each link is repeated as many times as it occurred during regular desktop activity). Finally, in order to address a broad scope of possible ranking algorithms, we also experimented with more trivial reputation measures, namely (1) frequency of accesses and (2) total access time.

Other Heuristics to Generate Desktop Links. There exists a plethora of other cues for inferring desktop links, most of them being currently unexplored by previous work. For example the *files stored within the same directory* have to some extent something in common, especially for filers, i.e., users that organize their personal data into carefully selected hierarchies. Similarly, *files having the same file name* (ignoring the path) are in many times semantically related. In this case however, each name should not consist exclusively of stopwords. More, for this second additional heuristic we had to utilize an extended stopword list, which also includes several very common file name words, such as "index", or "readme". In total, we appended 48 such words to the original list. Finally, we note that both these above mentioned approaches favor lower sets: If all files within such a set (e.g., all files residing in the same directory) are linked to each other, then the stationary probability of the Markov chain associated to this desktop linkage graph is higher for the files residing in a smaller set. This is in fact correct, since for example a directory storing 10 items has most probably been created manually, thus containing files that are to some extent related, whereas a directory storing 1,000 items has in most of the situations been generated automatically. Also, since these sub-graphs of the main desktop graph are cliques, several computational optimizations are possible; however, in order to keep our algorithms clear we will not discuss them here.

A third source of linkage information is *file type*. There is clearly a connection between the resources sharing the same type, even though it is a very small one.

Unfortunately, each such category will nowadays be filled with up to several thousands of items (e.g., JPG images), thus making this heuristic difficult to integrate into the ranking scheme. A more reliable approach is to *use text similarity to generate links between very similar desktop resources*. Likewise, if *the same entity appears in several desktop resources* (e.g., Hannover appears both as the name of a folder with pictures and as the subject of an email), then we argue that some kind of a semantic connection exists between the two resources. Finally, we note that users should be allowed to manually create links as well, possibly having a much higher weight associated to these special links.

Practical Issues. Several special cases might arise when applying usage analysis for desktop search. First, the textual log file capturing usage history should persist over system updates in order to preserve the rich linkage information. In our experiments, we collected only about 80 KB of log data over two months. Second and more important, what if the user looks for a file she stored five years ago, when she had no desktop search application installed? We propose several solutions to this:

1. The naïve approach is to simply enable ranking based exclusively on TFxIDF. However, much better results can be obtained by incorporating contextual information within the ranking scheme.
2. We therefore propose a more complex query term weighting scheme, such as BM25 [8]. Teevan et al. [15] have recently proposed an application of this metric to personalize web search based on desktop content. In our approach, their method must be adapted to personalize *desktop* search based on a specific *activity context*, represented for example by the files with a specific path or date range.
3. If the user remembers the approximate moment in time when she accessed the sought item, then this date represents a useful additional context based vertical ranking measure. For example, if the user remembers having used the target file around year 1998, the additional importance measure is represented by the normalized positive time difference between mid-1998 and the date of each output result.
4. If no contextual information is available, we propose to infer it through a relevance feedback process, in which the user first searches the desktop using TFxIDF exclusively, and then selects one or several (relatively) relevant results, which are then used to extract a context (e.g., date) or to propose expansions to the user query.

Comparison to the Web Model. Clearly, unlike in the web, most of the desktop search queries are navigational: users just want to locate something they know their stored before. So, are some desktop files more important than others, or are they all approximately equally important? We argue that, *as in the web*, some desktop resources are much more important than others, and thus users will most of the time be seeking only for these highly important items. For example, one year after some project was closed, a log file inspected by the researcher 400 times during an experiment will definitely be less important than the project report which was probably accessed only 100 times. Therefore, contextual information, though very important, is not sufficient in effectively locating desktop items, and more complex importance measures are needed in order to exploit user's activity patterns, her local desktop organization, etc. We thus propose to link together the resources matching these heuristics (i.e., having similar access

patterns, etc.), and then to utilize the resulting linkage structure to infer a global ranking over the PC Desktop.

4 Experimental Results

Experimental Setup. We evaluated the utility of our algorithms within three different environments: our laboratory (with researchers in different computer science areas and education), a partner laboratory with slightly different computer science interests, and the architecture department of our university. The last location was especially chosen to give us an insight from persons with very different activities and requirements. In total, 11 persons installed our logging tool and worked normally on their desktops for 2 months[1]. Then, during the subsequent 3 weeks, they performed several desktop searches related to their regular activities[2], and graded each top 10 result of each algorithm with a score ranging from 1 to 5, 1 defining a very poor result with respect to their desktop data and expectations, and 5 a very good one. This is in fact a Weighted P@10 [2]. For every query, we shuffled the top ten URIs output by each of our algorithms, such that the users were neither aware of their actual place in the rank list, nor of the algorithm(s) that produced them. On average, for every issued query the subjects had to evaluate about 30 desktop documents (i.e., the reunion of the outputs of all approaches we investigated). In total, 84 queries had been issued and about 2,500 documents were evaluated.

For the link based ranking algorithms (recall that for the sake of completeness we have also evaluated some access time ranking heuristics) we set the parameter ϵ to four times the average break time of the user. We have also attempted to set it to one hour, and eight times the average break time of the user, but manual inspection showed these values to yield less accurate usage sessions. Although much more complex techniques for computing usage session times do exist (e.g., exploiting mouse clicks or movements, scrollbar activities, keyboard activities, document printing, etc. [5, 11]), we think this heuristic suffices for proving our hypothesis, i.e., usage analysis based ranking improves over simple textual retrieval approaches.

In the following, we will first present an analysis of this experiment focused on the ranking algorithms, and then another one, focused on the quality of the search output they produced.

Ranking analysis. We first analyzed how our algorithms perform, in order to tune the parameters discussed before and to investigate whether the non-usage analysis heuristics do indeed make a difference in the overall rankings. We thus defined and analyzed the following 17 algorithms:

- **T1:** Algorithm 3.1 with $T = 1$.
- **T1Dir:** "T1" enriched with additional links created as complete subgraphs with the files residing in every desktop directory (i.e., all the files in a directory point to each other).

[1] The logger was implemented using a hook that catched all manual file open / create / save system calls.

[2] The only requirement we made here was to perform at least 5 queries, but almost every subject provided more. In all cases, we collected the *average* rating per algorithm for each person.

- **T1DirFnames:** "T1Dir" with further additional links created as complete subgraphs with the resources having the same file name (i.e., all items with the same file name point to each other, provided that the file name does not consist exclusively of stopwords).
- **T1Fnames:** "T1" enriched with the links between resources with identical file names as in the previous algorithm[3]. This was necessary to inspect the specific contribution of directories and file names respectively to the overall ranking scheme.
- **T1x3Dir:** Same as "T1Dir", but with the links inferred from usage analysis being three times more important than those inferred from the directory structure.
- **T1x3DirFnames:** Same as above, but also including the links provided by identical file names.
- **T1x3Fnames:** Same as "T1x3Dir", but using the file name heuristic instead of the directory one.
- **T2:** Algorithm 3.1 with $T = 2$.
- **T3:** Algorithm 3.1 with $T = 3$.
- **VisitFreq:** Ranking by access frequency.
- **1HourGap:** Ranking by total amount of time spent on accessing each resource, with sessions delimited by one hour of inactivity.
- **4xAvgGap:** Ranking by total access time, with sessions delimited by a period of inactivity longer than four times the average break time of the user.
- **8xAvgGap:** Same as above, but with sessions bounded by a period of inactivity longer than eight times the average average break time of the user.
- **Weighted:** Algorithm 3.1 directly using the matrix A, instead of A', i.e., with links weighted by the number of times they occurred.
- **WeightedDir:** Algorithm "Weighted" enriched with links between the files stored within the same directory.
- **WeightedDirFnames:** The previous algorithm with a link structure extended with connections between files with identical names.
- **WeightedFnames:** Same as above, but without the links generated by exploiting the desktop directory structure.

Since in-link count is almost identical to file access count (frequency), we only experimented with the latter measure. The only difference between these two measures is that in-link count will result in lower page scores when a threshold higher than one is used to filter-out the links (see also Algorithm 3.1).

We analyzed two aspects at this stage: First, it was important to inspect the final distribution of rankings, as this indicates how desktop search output looks like when using these algorithms. In *all* cases the resource rankings exhibits a distribution very well shaped by a power law: The left side of Figure 1 plots the output rankings for algorithm "T1", and its right side depicts the output when both directory and file name heuristics were added (in this latter case we notice a strong exponential cut-off towards the end, for the files that benefited less from the link enhancement techniques).

[3] For emails, this corresponded to having the same subject, eventually with "Re:" or "Fwd:" inserted in the beginning.

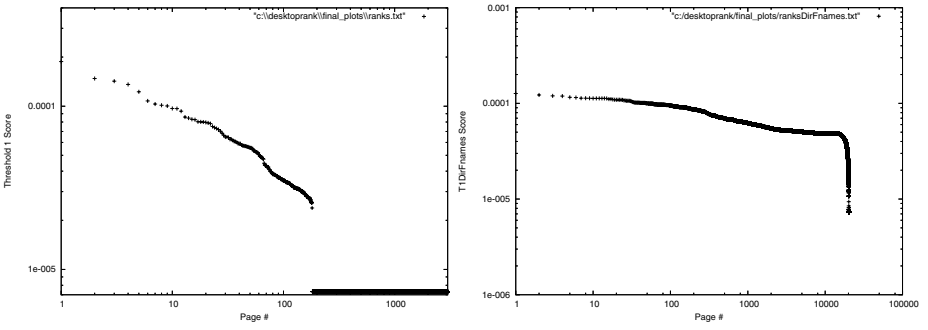


Fig. 1. Distribution of scores for the "T1" (left) and "T1DirFnames" (right) algorithms

The second aspect to analyze was whether there is a difference between these heuristics. For this purpose we used a variant of Kendall's τ measure of similarity between two ranking vectors [10], which resulted in a similarity score falling within [-1,1].

Three of our testers (one from each location) were specifically asked to extensively use our tool. When they reached 40 queries each, we applied the Kendall measure on their complete output, as returned by each algorithm. The results are illustrated in Table 1. After analyzing them, we drew the following conclusions:

- The heuristics to link the resources residing within the same directory, or the resources with identical file names did result in a rather different query output.
- The approaches "T1x3Dir", "T1x3DirFnames" and "T1x3Fnames" did not yield a significant difference in the results.
- The output of "T2" and "T3" was very similar, indicating that a threshold higher than 2 is not necessary for Algorithm 3.1.
- "4xAvgGap" and "8xAvgGap" performed very similar to each other.
- "Weighted" output was very close to "T1".
- Finally, when "Weighted" was combined with directory or file name information, we obtained almost identical outcomes as when we used "T1" with these heuristics.

As a rule of thumb, we considered similar all algorithm pairs with a Kendall τ score above 0.5, and therefore removed one of them from the search quality evaluation. Exceptions were "Weighted" and "VisitFreq" (both very similar to "T1") in order to have at least one representative of their underlying heuristics as well.

Finally, inspecting the rank distributions generated by these heuristics also helped us obtain an additional interesting result, namely that only about 2% of the desktop indexable items are actually manually accessed by the user. This further supports the idea of exploiting usage information in ranking desktop search results, as current textual measures many times output high scores for documents that have never been touched by the user (e.g., HTML program documentation files).

Search quality analysis. After the previous analysis, we kept 8 algorithms for precision evaluation: "T1", "T1Dir", "T1DirFnames", "T1Fnames", "T2", "VisitFreq", "4xAvgGap" and "Weighted". Even though they do not incorporate any textual information,

Table 1. Kendall similarity for the desktop ranking algorithms (average over 120 queries from 3 users)

Algorithm	T1	T1Dir	T1DirFnames	T1Fnames	T1x3Dir	T1x3DirFnames	T1x3Fnames	T2	T3	VisitFreq	1HourGap	4xAvgGap	8xAvgGap	Weighted	WeightedDir	WeightedDirFnames	WeightedFnames
Threshold 1	1																
T1Dir	0.22	1															
T1DirFnames	0.22	0.47	1														
T1Fnames	0.23	0.22	0.35	1													
T1x3Dir	0.28	0.86	0.46	0.23	1												
T1x3Dir-Fnames	0.24	0.48	0.75	0.40	0.48	1											
T1x3Fnames	0.22	0.24	0.36	0.88	0.24	0.41	1										
Threshold 2	0.20	0	-0.2	0	0.02	-0.2	0	1									
Threshold 3	0.01	-0.1	-0.3	-0.1	-0.1	-0.3	-0.1	0.60	1								
VisitFreq	0.66	0.24	0.15	0.27	0.26	0.20	0.28	0.26	0.05	1							
1HourGap	0.48	0.15	0.14	0.20	0.12	0.11	0.20	0.17	0.02	0.41	1						
4xAvgGap	0.43	0.25	0.18	0.23	0.26	0.19	0.24	0.20	0.04	0.43	0.34	1					
8xAvgGap	0.48	0.26	0.16	0.21	0.27	0.18	0.22	0.16	0.04	0.50	0.47	0.70	1				
Weighted	0.75	0.20	0.21	0.20	0.25	0.24	0.20	0.24	0.01	0.64	0.52	0.47	0.47	1			
WeightedDir	0.22	0.89	0.47	0.22	0.85	0.48	0.24	0	-0.1	0.21	0.11	0.26	0.27	0.22	1		
Weighted-DirFnames	0.21	0.47	0.89	0.34	0.46	0.75	0.36	-0.2	-0.3	0.15	0.14	0.18	0.16	0.21	0.47	1	
Weighted-Fnames	0.26	0.24	0.37	0.83	0.25	0.43	0.81	0	-0.1	0.31	0.28	0.28	0.26	0.25	0.24	0.36	1

we still started with ranking desktop search results only according to these measures, in order to see the impact of usage analysis on desktop ranking. The average results are summarized in the second column of Table 2. As we can see, all measures performed worse than TFxIDF (we used Lucene[4] together with an implementation of Porter's stemmer to select the query hits, as well as to compute the TFxIDF values), but only at a small difference. This indicates that users do issue a good amount of their desktop queries on aspects related to their relatively recent, or even current work. Also, as the "T2" algorithm does not improve over "T1", it is therefore sufficient to use Algorithm 3.1 with a threshold $T = 1$ in order to effectively catch the important desktop documents. This is explainable, since a threshold $T = 2$ would only downgrade files that were accessed only once, which have a relatively low score anyway compared to the other more frequently touched resources.

[4] http://lucene.apache.org

Table 2. Average grading for the usage analysis algorithms with and without a combination with TFxIDF, together with tests on the statistical significance of the improvement the latter ones bring over regular TFxIDF

Algorithm	Weighted P@10 (Usg. An.)	Weighted P@10 (Combined)	Signif. for Combined versus TFxIDF
T1 * TFxIDF	3.04	3.34	$p = 0.003$
T1Dir * TFxIDF	3.02	3.36	$p < 0.001$
T1DirFnames * TFxIDF	**2.99**	**3.42**	$\mathbf{p \ll 0.001}$
T1Fnames * TFxIDF	2.97	3.26	$p = 0.064$
T2 * TFxIDF	2.85	3.13	$p = 0.311$
VisitFreq * TFxIDF	2.98	3.23	$p = 0.141$
4xAvgGap * TFxIDF	2.94	3.09	$p = 0.494$
Weighted * TFxIDF	3.07	3.30	$p = 0.012$
TFxIDF	3.09	3.09	

Finally we investigated how our algorithms perform within a realistic desktop search scenario, i.e., combined with term frequency information. We used the following formula:

$$Score(file) = NormalizedScore(file) * NormalizedVSMScore(file, query)$$

The VSM score is computed using the Vector Space Model and both scores are normalized to fall within [0,1] for a given query[5]. The resulted average gradings are presented in the third column of Table 2. We notice that in this approach, *all* measures outperform TFxIDF in terms of weighted precision at the top 10 results, and most of them do that at a statistically significant difference (see column 4 of Table 2 for the p values with respect to each metric).

The usage analysis based PageRank ("T1") is clearly improving over regular TFxIDF ranking. As for the additional heuristics evaluated, connecting items with similar file name or residing in the same directory, they yielded a significant improvement only when both of them have been used. This is because when used by themselves, these heuristics tend to bias the results away from the usage analysis information, which is the most important by far. When used together, they add links in a more uniform manner, thus including the information delivered by each additional heuristic, while also keeping the main bias on usage analysis. Finally, the simpler usage analysis metrics we investigated (e.g., ranking by frequency or by total access time) did indeed improve over TFxIDF as well, but with a lower impact than the Algorithm 3.1 enriched with directory and file name information. We conclude that with TFxIDF in place, usage analysis significantly improves desktop search output rankings and it can be further enhanced by linking resources from the same directory and with identical file names.

The final results are also illustrated in Figure 2, in order to make the improvement provided by our algorithms also visible at a graphical level. The horizontal line residing at level 3.09 represents the performance of TFxIDF; the red bars (right hand side) depict the average grading of the algorithms combining TFxIDF with our approaches, and the blue ones (left hand side) depict the average grading obtained when using only our usage analysis algorithms to order desktop search output.

[5] In order to avoid obtaining many null scores when using access frequency or total access time (recall that many items have never been touched by the user), in these scenarios we also added a $1/N$ score to all items before normalizing, with N being the total amount of desktop items.

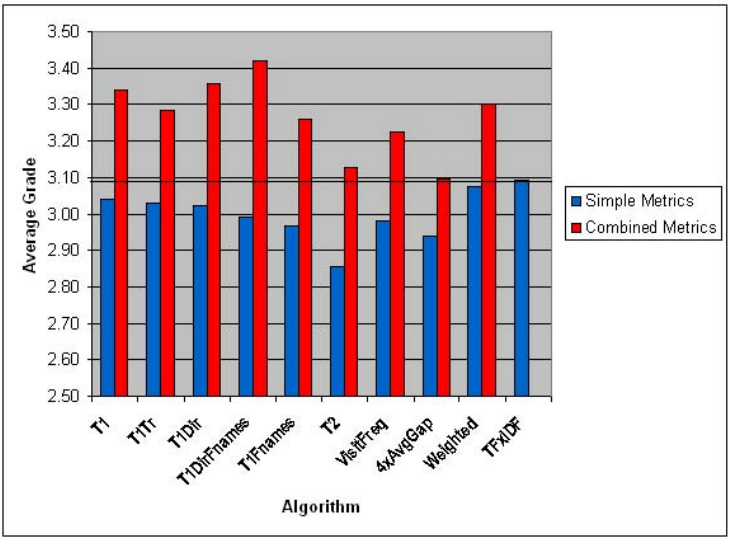


Fig. 2. Average grading for the usage analysis algorithms

5 Conclusions and Future Work

Currently there are quite several personal information systems managing PC desktop resources. However, all of them have focused on seeking solutions to find previously stored items in a faster way. In this paper we argued that in many cases these existing approaches already yield several hundreds of query results, which cannot be successfully ordered by using textual retrieval measures exclusively. To solve this problem, we proposed to introduce *ranking* for desktop items and we investigated in detail several approaches to achieve this goal, ranging from usage analysis to exploiting contextual information. Our extensive experiments showed that such techniques do indeed significantly increase desktop search quality with up to 10.67% in terms of Average (Weighted) Precision.

In future work we intend to explore content based heuristics to provide us with additional links between similar desktop documents, as well as to combine our techniques with "recency" information about file accesses, which was previously proved to be quite important in locating desktop resources [6]. Also, we would like to analyze the necessity and benefits of enabling desktop search restrictions to only some specific sub-tree of the local hierarchy, as well as of clustering near-duplicate desktop resources (which is a phenomenon more common than in the web).

Acknowledgements

First and foremost we thank Leo Sauermann from DFKI for his valuable suggestions and for implementing the module that logs user desktop activity. We also thank our colleagues Jörg Diederich and Uwe Thaden for pre-reviewing our paper, as well as to

all those who kindly agreed to participate in our experiments. Finally, we thank all colleagues involved in the Beagle^{++} project for the interesting discussions we had within this context.

References

1. E. Adar, D. Kargar, and L. A. Stein. Haystack: per-user information environments. In *Proc. of the 8th Intl. CIKM Conf. on Information and Knowledge Management*, 1999.
2. R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.
3. P. A. Chirita, R. Gavriloaie, S. Ghita, W. Nejdl, and R. Paiu. Activity based metadata for semantic desktop search. In *Proc. of the 2nd European Semantic Web Conference*, 2005.
4. P. A. Chirita, S. Ghita, W. Nejdl, and R. Paiu. Beagle++: Semantically enhanced searching and ranking on the desktop. In *Proc. of the 3rd European Semantic Web Conference*, 2006.
5. M. Claypool, D. Brown, P. Le, and M. Waseda. Inferring user interest. *IEEE Internet Computing*, 5(6), 2001.
6. S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. Robbins. Stuff i've seen: a system for personal information retrieval and re-use. In *Proc. of the 26th Intl. ACM SIGIR Conf. on Research and Development in Informaion Retrieval*, pages 72–79, 2003.
7. J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong. Mylifebits: fulfilling the memex vision. In *Proc. of the ACM Conference on Multimedia*, 2002.
8. K. S. Jones, S. Walker, and S. Robertson. Probabilistic model of information retrieval: Development and status. Technical report, Cambridge University, 1998.
9. D. R. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A customizable general-purpose information management tool for end users of semistructured data. In *Proc. of the 1st Intl. Conf. on Innovative Data Syst.*, 2003.
10. M. Kendall. *Rank Correlation Methods*. Hafner Publishing, 1955.
11. D. Oard and J. Kim. Modeling information content using observable behavior. In *Proceedings of the 64th Annual Meeting of the American Society for Information Science and Technology*, 2001.
12. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.
13. D. Quan and D. Karger. How to make a semantic web browser. In *Proc. of the 13th Intl. WWW Conf.*, 2004.
14. C. Soules and G. Ganger. Connections: using context to enhance file search. In *SOSP*, 2005.
15. J. Teevan, S. T. Dumais, and E. Horvitz. Personalizing search via automated analysis of interests and activities. In *Proc. of the 28th Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2005.

The Intention Behind Web Queries

Ricardo Baeza-Yates[1], Liliana Calderón-Benavides[2], and Cristina González-Caro[2]

[1] Yahoo! Research Barcelona
Ocata 1, 08003 Barcelona, Spain
ricado@baeza.cl
[2] Web Research Group
Universitat Pompeu Fabra, Passeig de Circumval-lació 8
08003 Barcelona, Spain
{liliana.calderon, cristina.gonzalez}@upf.edu

Abstract. The identification of the user's intention or interest through queries that they submit to a search engine can be very useful to offer them more adequate results. In this work we present a framework for the identification of user's interest in an automatic way, based on the analysis of query logs. This identification is made from two perspectives, the objectives or goals of a user and the categories in which these aims are situated. A manual classification of the queries was made in order to have a reference point and then we applied supervised and unsupervised learning techniques. The results obtained show that for a considerable amount of cases supervised learning is a good option, however through unsupervised learning we found relationships between users and behaviors that are not easy to detect just taking the query words. Also, through unsupervised learning we established that there are categories that we are not able to determine in contrast with other classes that were not considered but naturally appear after the clustering process. This allowed us to establish that the combination of supervised and unsupervised learning is a good alternative to find user's goals. From supervised learning we can identify the user interest given certain established goals and categories; on the other hand, with unsupervised learning we can validate the goals and categories used, refine them and select the most appropriate to the user's needs.

1 Introduction

Current Web search engines have been designed to offer resources to their users, but with the limitation that the goals or characteristics behind the queries made by them are not generally considered. Given that a query is the representation of a need, a set of factors, in most cases, are implicit within this representation. If we can discover these factors, they can be crucial in the information recommendation process. Techniques such as Web Usage Mining [1] cover the problem to improve the quality of information to users by analyzing Web log data. Particularly, Web Query Mining [2, 3], deals with the study of query logs from data registered in

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 98–109, 2006.

a search engine, with the purpose of discovering hidden information about the behavior of users of these kind of systems. Some work has been done to categorize the needs of users; for example, the categorization proposed by Broder [4] in which, according with the goal of the user, three classes are considered: Navigational, Informational and Transactional. Broder made a classification of queries through an user survey and manual classification of a query log. This work was later taken up by Rose and Levinson [5], who developed a framework for manual classification of search goals by extending the classes proposed by Broder. In their studies Broder, and Rose and Levinson showed that goals of queries can be identified manually. In Lee *et al.* [6] the work was focused on automatic identification of goals (navigational and informational) through the application of heuristics over clicks made by the users on the results offered by the search engine; in order to do it, they proposed two related features, the past user-click behavior and the anchor-link distribution. In the same context, works like Spereta *et al.* [7] tries to establish user profiles by using their search histories; Baeza-Yates *et al.* [2] discovered groups of related queries, through text clustering of documents clicked for the queries, allowing an improvement of the search process.

In general, the approaches try to make an approximation to the user from different perspectives. However, a model in which the user can be identified by using his/her goals hasn't been completely developed. We could then use this kind of information to understand and improve his/her information needs. Taking this into consideration, the main goal of this work is to develop a model for identification of the user's interests for a Web search engine, using the user interactions stored in the query log files of the system. The identification process is made from two perspectives, the first one is from the objectives or goals of each one of the users and the second is from the categories in which each of the objectives can be situated. To be able to measure precision and recall we manually classified more than 6,000 real queries, a reference set two orders of magnitude larger compared to the 50 CS related queries used by Lee *et al.* [6].

This paper is organized as follows. In Section 2 we describe user's goals and categories to which the user's queries can belong. A brief description of the used techniques to find user interest are presented in Section 3. In Section 4 we present the experimental design of this work. Finally, we present an analysis of the obtained results in Section 5 and conclude the paper in Section 6.

2 User's Goals and Categories

As a way to determine the motivations during an information search, we propose firstly, to find the user goals and secondly mapping these queries into categories. This information allow us to determine the path that a user follows when is searching for information on a web search engine.

2.1 User Goals

From the content of the queries we established three categories for the reasons or goals which motivate the user to make a search: *Informational, Not informational*

and *Ambiguous*. An informational query is one in which the user exhibits an interest to obtain information available in the Web, independently of the knowledge area of the resource retrieved. As not informational we categorize queries that find other resources or target an specific transaction (e.g. buy, download, reserve, etc.). Finally, ambiguous queries are those that their goal cannot be inferred directly from the query (in some cases because the user has an ambiguous interest). For Informational queries we could use a ranking biased towards text content while for Not Informational queries a better answer could be a few Web sites, good hub pages (many good links) or a price comparison portal.

2.2 Query Categories

A key point in the process of user interests identification is establishing the topic to which each submitted query belongs. The discovery of the kind of information requested allows us to identify it in a particular area of interest and relate it to specific characteristics of the area in which it is related (or in which he/she wants to be related).

Topics used to classify the queries are based on the general categories of the Open Directory Project, ODP[1] *(Arts, Games, Kids and Teens, Reference, Shopping, World, Business, Health, News, Society, Computers, Home, Recreation, Science, Sports)*. Apart of these general categories we considered three more which are: *Various* for those queries that from their content seems as belong to more than one category, *Other* for queries which can't be classified into one of the selected categories and *Sex* taking in account that a considerable amount of queries are related with this topic.

3 Selected Techniques

As a way to reach our purpose we selected two quite different models which, from the literature are available to categorize data and find hidden relationships among data. The selected models were Support Vector Machines (SVM) [8] and Probabilistic Latent Semantic Analysis (PLSA) [9].

3.1 Support Vector Machines Model

In this work, Support Vector Machines [8] have been used to build classification models for queries. We chose this classifier, given their proven effectiveness in different scenarios with a high feature dimensionality, including text classification [10]; considering that the queries were represented by the words of the pages selected by the users, this characteristic is quite useful. To solving the multiclass problem, we combine SVM with Error-Correcting Output Coding (ECOC) [11], which reduces the multiclass problem to a group of binary classification tasks and combine the binary classification results to predict multiclass labels. The RBF (Radial Basis Function) kernel was used to the SVM's setup, and we choose the kernel's parameters through a standard cross-validation process.

[1] Open Directory Project. http://dmoz.org

3.2 Probabilistic Latent Semantic Analysis

As we have commented, one of the main ideas that justifies the development of this work is to find the reasons which motivate the user to make a search in the Web. Considering this, and in accordance with different works such as Jin [12] and Lin [13], Probabilistic Latent Semantic Analysis (PLSA) [9] appear to be an efficient method of analyzing user interests.

Given that the starting point for PLSA is a statistical model which has been called Aspect Model [9], the implementation of this model used in this work was taken from PennAspect [14], a well tested software for information filtering and retrieval.

4 Experimental Design

Data Set. For this work we processed a log sample from the Chilean Web search engine TodoCL[2]. Thesample contains 6,042 queries having clicks in their answers. There are 22,190 clicks registered in the log, and these clicks are over 18,527 different URLs. Thus, in average users clicked 3.67 URLs per query.

Data Preprocessing. One of the most important ideas to exploit here is finding existing relationships in the data. In order to achieve this, each query was represented as a vector of terms that appeared in the documents giving an answer to the query (stop words were removed), $Q_i = w(t_1), w(t_2) \dots w(t_n)$, where w($tj$) is the associated weight of term j inside query Q_i. The classical TF-IDF weighting scheme was used to assign the weight to each query term and clicked page, replacing IDF by the number of clicks on each page (see [2]).

After that, a clustering process was applied over the data. We obtained query groups with similar characteristics, i.e. they belong to the same subject, are related with specific topics or describe the same situation using different terms. To do this, we used the simple K-means clustering method.

Manually Classified Data. To be able to evaluate the results of our automatic classification, we built a test set based on a team of people who performed a manual classification of the queries.

An important characteristic about the structure of these queries, as we mentioned before, is that they can be organized in clusters. This structure facilitated the manual classification process, by providing our team with information about the context of the query and at the same time, giving a global idea about the class to which each one of them belongs. This information is used to facilitate the human classifiers in the case that a query did not suggest a complete idea by itself. In any case, the task of a human classifier is to select the type of goal for a user and the category in which this goal can be situated. Considering the amount of queries and the different categories in which they can belong, the manual classification process is hard to do and subject to some subjectivity

[2] TodoCL. http://www.todocl.com

(and hence, errors). As a way to facilitate this process we created a software tool which offers to users the possibility to select the goals and the categories and save them in an organized and fast way.

Table 1. Manual classification of queries into goals and categories

Category	Inf	N-Inf	Amb	Total	Category	Inf	N-Inf	Amb	Total
Arts	102	23	29	154	Society	501	12	60	573
Games	11	26	8	45	Home	50	35	41	126
Education	232	29	23	284	Recreation	789	489	142	1,420
Reference	107	85	26	218	Science	129	7	9	145
Shopping	55	29	39	123	Sports	31	11	5	47
World	46	6	15	67	Computers	174	208	86	468
News	78	5	1	84	Sex	37	178	33	248
Business	960	107	93	1,160	Others	16	9	33	58
Health	171	21	40	232	Various	224	27	339	590

As we previously described, we established three categories to which the goals can belong: Informational (Inf), Not Informational (N-Inf) and Ambiguous (Amb). On the other hand, we established eighteen topics to classify the same queries. After the manual classification process of queries into goals and categories, the obtained amount are presented in the table 1.

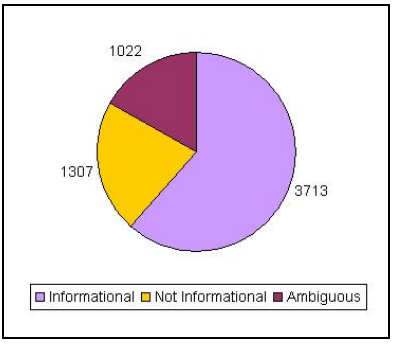


Fig. 1. Distribution of Queries into Goals

Manual classification of queries into goals. The figure 1 presents the amount of queries which were labeled by our team as Informational, Not Informational and Ambiguous. The goal with the higher number of queries was Informational, this happens because we considered as the kind of queries which not talking directly about an object (such as mp3 file, photo, among others), the name of an artist or the purchase or sale of a product or service. However the goals categories considered to label a query are different between this work and the work realized by Rose and Levinson [5], we agreed on the proportion of Informational queries, being this higher than the others.

Manual classification of queries into categories. A graphical representation of the manual classification of queries into categories is presented in the figure 2. The categories with higher amount of queries are Entertainment and Business, which confirm the search behavior of people that have been well described by Spink and Jansen in their works [15, 16].

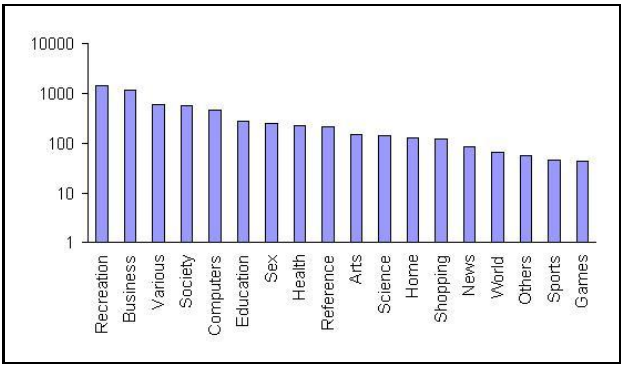


Fig. 2. Distribution of Queries into Categories

The figure 3 present the distribution of the percentage of categories into the different goals. The queries grouped as Informational goal belong to categories such as Business, Education, Science or News in which people are searching for resources answering in many of the cases to a specific information need. On the other hand, queries grouped as Not Informational belong to categories such as Recreation, Sex or Games in which the intention is, in most of the cases to visit a place to find one of this kind of sources. Finally, the queries grouped as Ambiguous are more present in the Various and Other categories due that it is not clear what the user wants and hence are quite difficult to classify in one of the other.

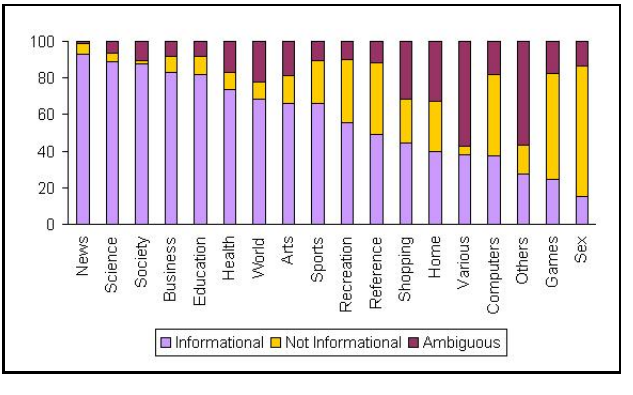


Fig. 3. Percentage distribution of queries into Goals and Categories

Performance. For the training phase, PLSA took, on average, four hours to build a model and calculate the different probabilities of each query and the words belonging to each latent class. As we mentioned before, we determined three goals and selected eighteen categories to which the user interest can be situated and their queries can be classified. These quantities were used to generate the query groups (in section 5.2 we will comment about this fact).

To build the models and make the predictions for categories, SVM spent about two hours. For the case of goals, considering the low amount of labels involved, this model took about fifty minutes.

The different algorithms were run on a Pentium III computer with 1.28 GBs of RAM, under a Linux OS.

5 Analysis of Results

5.1 Supervised Learning

After the manual classification of the queries was made, part of these labeled data was used like input to train an automatic classifier.

The obtained results in the classification process with supervised learning were good. From the labeled examples by our team of editors, quite suitable models for each goal and category were constructed. Although, not in all the cases the predictions agree with the human judgments, the prediction in most cases is related with the subject of the query, showing therefore the ambiguous nature of some queries that can be located in different goals or categories (our Ambiguous class). Nevertheless, in the case of the categories, the idea was, as far as possible, to assign a determined category to each query, the category "Various" was used as minimum as possible in the manual classification.

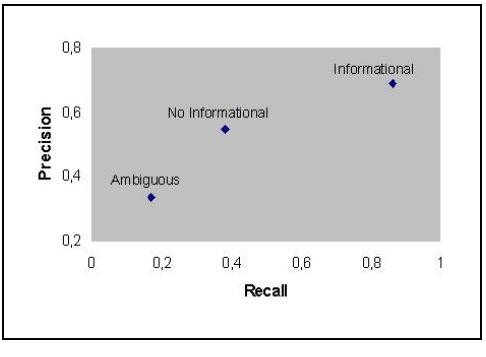


Fig. 4. Recall-Precision Graph of Goal's Automatic Classification

With respect to the user's information goals the results are good, the precision is over 50% for two of the goals. The best results are obtained with the Informational goal, its precision is high and its recall is almost perfect, see figure 4. This is due that the pages selected for the queries that belong to this goal are more

homogeneous, unlike those that belong to "Not Informational" or "Ambiguous", where the nature of the pages selected by the users is heterogeneous, since the users do not have a very precise idea of what they wish to find.

For the categories, the results are good in general; nevertheless, for some categories the results are better than for others. In the figure 5 we can see a sample of the most representative categories:

The categories that show better precision are those that have greater popularity, that is to say, those related to subjects that the people consult most frequently, like for example: Recreation. Since most of pages of this type of subjects handle a moderately similar vocabulary, the queries are more identifiable. Categories as "Technology and Computers" have, relatively, a specialized vocabulary, which allows to identify more accurately the related queries. Another particular case is the category "Sex", where the users do not change the words used to make their queries, most of these are built using the same words, without counting that, even though the users use different words to describe their queries, the pages which they choose as answer does not change, so the queries are repeated from the same words again.

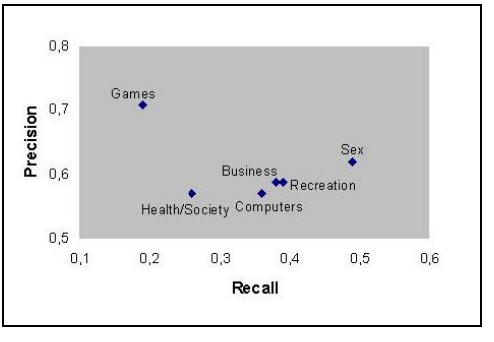


Fig. 5. Recall-Precision Graph of Categories's Automatic Classification

When we analyzed the relationship between goals and categories, we can observe the coherence that exists between the informational objectives of the users and the categories in which their queries are located. For the Informational goal, the greater distribution of queries are in the categories: Business, Recreation, Society, Education and News. Whereas the categories with smaller concentration of queries in this goal are: Games, Shopping, Home, Sex and Others, these last categories, suggest different motivations for the user that are not related to obtain information, for example Games, where the users are more interested in downloading software and resources. A particular case is the category "News", that it concentrates all its queries in this goal, these queries don't belong to other goals. Similarly, the goal "No Informational" shows enough coherence with the categories of the queries that belong to her. In this case, the categories with greater concentration of queries are Sex, Recreation and Technology and Computers, and those of smaller concentration of queries are Society, Health, Science

and News (as they do not have queries in this goal). Finally, "Ambiguous", that it's a goal that as its name indicates it, is not related to some category in individual, the greater concentration of queries is in the category "Various", which is logical, given the nature of the category and the goal.

5.2 Unsupervised Learning

Categories. To determine the relationships, at the categories level, between different queries used in this work, we considered topics (described in section 2.2), to which the queries can belong, as the variables that make that a user submit a specific query.

Before analyzing the obtained results with PLSA, and having in mind that this work is focussed on user interest, is important to highlight that in a common process of information search we are exposed, first, to the lack of precision between the transformation of a mental information need to a set of key terms that correctly describes this need, and second, to the lack of accuracy of search engines to provide an answer including aspects such as subjectivity or the context of the searching task. However, taking advantage of the results offered by PLSA model, and it's capability to provide a probabilistic information about the degree of membership of each query to each generated cluster, we can make an analysis of the composition of each cluster content. This information offer us the possibility to discover direct and not direct relationships between queries and topics (i. e. to which categories a query can belong), and from this information we can determine what is the user interest.

One of the most important aspects to highlight here is that although the amount of clusters, used to make the clustering process for categories identification of user interests, was taken from the ODP categorization, the obtained results from PLSA shows a hard grouping of queries around some of the categories such as Sex, Entertainment, Business, References or Health. However, the model could not create significant groups for categories like Arts, Sports, Science or Games. This happens not only because the amount of queries is very low, but also because they are mixed with other unrelated queries. This information was used from two point of views:

- Ratify that most of the selected categories used in the manual classification are clearly defined. However, there are other categories that have overlapped content and are difficult to determine. In contrast to these facts, other possible categories that we did not consider appeared, such as cars and law.
- From this information we can identify existent relationships between queries. The table 2 shows an example of these relationships. In this table we have a sample of queries grouped in cluster 6, which was labeled as Recreation or Entertainment. By observing the probabilities values (Prob1, Prob2, Prob3) of each one of these queries belonging to each cluster, the highest values are for clusters labeled as Business (cluster 7) and Sex (cluster 11).

In general terms, we can say that queries that were grouped in the Sex category, have a high probability to belong to entertainment, which is absolutely

Table 2. Queries with three highest probabilities in the Recreation cluster

QId	Query	Prob1	CId	Prob2	CId	Prob3	CId
4197	los jaivas main works	1.76E-03	6	1.99E-09	0	4.30E-12	7
243	ricardo arjona spanish songs	1.51E-03	6	2.01E-08	7	2.12E-42	11
5759	madonna erotic	1.50E-03	6	1.83E-08	7	2.20E-42	11
1917	porto seguro cd	1.50E-03	6	2.68E-08	7	1.29E-43	11
5378	rata blanca songs	1.50E-03	6	2.69E-08	7	8.84E-43	11

coherent; on the other hand, these same queries can be considered as belonging to business category due to that the content of pages answering sex or entertainment queries have terms related with payments or selling this kind of services.

A particular case was presented by the cluster which grouped queries related to health. About 70% of queries belonging to this cluster made reference to drugs, diseases or treatment of diseases. The reason for this case is that the medical vocabulary and the terms used to make this kind of queries are too specific, and is quite rare to find problems of synonimy or polysemy. The table 3 shows the five queries with highest probability in this cluster.

Table 3. Queries with highest probabilities in the Health cluster

IdQuery	Query	Probability
1831	electroconvulsive therapy	1.75E-03
2215	nasal polyps	1.53E-03
3507	dental hygienist	1.51E-03
3156	hepatitis	1.41E-03
5023	viagra	1.03E-03

Goals. Through PLSA we found that approximately 73% of the 2,168 queries grouped as ambiguous, belong to categories such as Sex or Entertainment. It is important to note that none of the queries, labeled in the manual classification as Health, is part of an ambiguous goal, as a person usually has in mind the name of an specific illness or drug.

From the 2,719 queries grouped as Informational, about 76% are related with References, Education, Health, Computers, Society and Home. Finally, from the 1,155 queries grouped as not informational, near to 70% were labeled (in the manual classification) as Computers, Entertainment, Society and Sex. The main difference between queries that belong to the ambiguous cluster and the not informational queries is that the second makes direct reference to a photos of famous artists or models, parts of computers and software downloads, and songs, among others.

6 Conclusions

In this work we have presented a first step to identify user's interests in a Web search engine based in a query log. An analysis was made from two perspectives:

the user's informational objectives and the categories in which the queries within these objectives can be located. In order to identify these interests, different techniques were used, initially a manual classification, whose objective was to make a recognition from the human judgments of the distribution of goals and categories that could have the queries to classify. Later, we carried out an automatic identification of these interests using supervised and unsupervised learning.

The supervised analysis allows us to establish that the user interests are identifiable using a particular representation of queries, together with the automatic classifier. This was a good combination, since representing the queries by the terms of the documents that gave good answers to them, reduces the problem of the low number of words that the users use to make their queries (and hence the sparsity of the query space), and additionally because the pages that belong to a same category share a similar vocabulary that allowed us to make a better classification.

From the unsupervised perspective, user needs related with entertainment, sex or business were very well detected and important relationships between these categories were reflected. Most of the queries that were grouped in one of these three categories can belong to another categories. On the other hand, not for all the proposed categories exist a strong way to determine users' needs. This happen because the terms used to summit a query and the content of the pages in an answer to this query can be used to describe different topics. From the eighteen proposed categories, just eleven of them were completely recognized. In the opposite, two new and well defined categories appeared in the clustering process, they were cars and law. This suggests to make a revision of the selected ODP categories, avoiding overlapping of information.

The bottom line is that for the Informational class and some categories we got over 70% precision and very good recall. This can be easily improved by trying other query representations, other classification techniques, etc.

Acknowledgements

The authors wish to thank Mari-Carmen Marcos for helpful comments and suggestions in the classification of queries. The authors are grateful to the Information Technologies Research Group from the University Autónoma of Bucaramanga for help in the manual classification process of queries. This work was partially supported by the Alpha Project AML/B7-311/97/0666/II-0291-FA.

References

1. Mobasher, B. In: Practical Handbook of Internet Computing. CRC Press (2005)
2. Baeza-Yates, R., Hurtado, C., Mendoza, M.: Query recommendation using query logs in search engines. In: Current Trends in Database Technology - EDBT, Springer-Verlag GmbH (2004) 588–596
3. Baeza-Yates, R.: Applications of web query mining. In: ECIR 2005. Volume 3408., Lecture Notes in Computer Science (2005)

4. Broder, A.: A taxonomy of web search. SIGIR Forum **36** (2002) 3–10
5. Rose, D.E., Levinson, D.: Understanding user goals in web search. In: International conference on WWW, ACM Press (2004) 13–19
6. Lee, U., Liu, Z., Cho, J.: Automatic identification of user goals in web search. In: International conference on WWW, ACM Press (2005) 391–400
7. Speretta, M., Gauch, S.: Personalizing search based on user search history (2004)
8. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Min. Knowl. Discov. **2** (1998) 121–167
9. Hofmann, T.: Probabilistic latent semantic analysis. In: Proc. of Uncertainty in Artificial Intelligence, Stockholm (1999)
10. Basu, A., Watters, C., Shepherd, M.: Support vector machines for text categorization. In: International Conference on System Sciences, Washington, DC, USA, IEEE Computer Society (2003) 103.3
11. Dietterich, T.G., Bakiri, G.: Solving multiclass learning problems via error-correcting output codes. Journal of AI Research **2** (1995) 263–286
12. Jin, X., Zhou, Y., Mobasher, B.: Web usage mining based on probabilistic latent semantic analysis. In: Knowledge discovery and data mining, New York, USA, ACM Press (2004) 197–205
13. Lin, C., Xue, G.R., Zeng, H.J., Yu, Y.: Using probabilistic latent semantic analysis for personalized web search. In: Web Technologies Research and Development, Berlin Heidelberg, Springer-Verlag GmbH (2005) 707–717
14. Schein, A., Popescul, A., Ungar, L.: Pennaspect: A two-way aspect model implementation. Technical report, (Department of Computer and Information Science, The University of Pennsylvania)
15. Spink, A., Wolfram, D., Jansen, M.B.J., Saracevic, T.: Searching the web: the public and their queries. Journal of the American Society for Information Science and Technology **52** (2001) 226–234
16. Jansen, B.J., Spink, A.: An analysis of web searching by european alltheweb.com users. Information Processing and Management: an International Journal **41** (2005) 361–381

Compact Features for Detection of Near-Duplicates in Distributed Retrieval

Yaniv Bernstein, Milad Shokouhi, and Justin Zobel

School of Computer Science and Information Technology
RMIT University, Melbourne, Australia

Abstract. In distributed information retrieval, answers from separate collections are combined into a single result set. However, the collections may overlap. The fact that the collections are distributed means that it is not in general feasible to prune duplicate and near-duplicate documents at index time. In this paper we introduce and analyze the *grainy hash vector*, a compact document representation that can be used to efficiently prune duplicate and near-duplicate documents from result lists. We demonstrate that, for a modest bandwidth and computational cost, many near-duplicates can be accurately removed from result lists produced by a cooperative distributed information retrieval system.

1 Introduction

In standard information retrieval (IR) systems, all documents are centrally managed and indexed on a single server. An alternative that has advantages when the data is physically dispersed is to use distributed information retrieval (DIR). In DIR, multiple separate servers, potentially at separate geographic locations, each provide a search service to a subset of the overall collection; the user interacts with a single interface known as the *broker*, which sends the query to the servers and aggregates the results.

Distributed information retrieval systems are generally classified as *cooperative* or *uncooperative*. In cooperative environments, distributed servers provide the broker with information about their contents that the broker can then use to select servers and interpret results [Gravano et al., 1997, Powell and French, 2003]. In uncooperative systems, the broker must obtain information from the servers by searching for and downloading answers [Callan and Connell, 2001]. In this paper, we are concerned with cooperative environments where the broker is able to request pertinent information from the distributed servers.

In most previous research in DIR, it is generally assumed that the overall collection is partitioned into disjoint subcollections; that is, it is assumed there is no overlap between the individually indexed sites [Callan and Connell, 2001, Nottelmann and Fuhr, 2003, Si and Callan, 2003, 2004]. This is not, in general, a valid assumption; duplication and near-duplication of documents between servers is a significant problem and is one of the current challenges in DIR [Allan et al, 2003]. Some research in web-based meta-search engines is concerned with elimination of exact duplicates from the list of final results [Gauch et al.,

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 110–121, 2006.

1996, Meng et al., 2002, Selberg and Etzioni, 1997, Zamir and Etzioni, 1999]. However, these techniques are restricted to detecting matches based on URL; this limits the applicability of the techniques to domains in which there is a unique identifier for each document. Furthermore, the issue of near-duplicates is not addressed by such techniques. To the best of our knowledge, more sophisticated measures for filtering duplicate and near-duplicate documents in a DIR context have not previously been explored.

In this paper we address the problem of robust duplicate and near-duplicate detection in DIR. We introduce a new type of feature vector, the *grainy hash vector* (GHV), which acts a compact surrogate of the document for near-duplicate detection. We analyze the properties of GHVs, and show empirically that they can be used for efficient and accurate merge-time detection of duplicate and near-duplicate documents.

2 Duplicates in Distributed Information Retrieval

The broker in a DIR system must perform two major tasks. The first is *server selection*: since it is costly to search all servers for every query, brokers need to select a limited number of servers to search. A server selection algorithm is used to identify the servers that are most likely to contain relevant documents. Many different server selection methods have been proposed; see for example Nottelmann and Fuhr [2003] and Si and Callan [2004]. Once the broker has submitted the query to each of the selected subset of servers and collected the returned results, it must perform *result merging*: based on returned information about the servers and each returned document, the broker must combine the results into a single list for presentation to the user.

Management of duplication across collections can be managed at either or both of the server-selection and result-merger stages. At the server selection stage, the broker can avoid selecting servers that have a high degree of collection overlap with a server that has already been selected. For such an approach to be effective, the rate of overlap between the underlying pairs of servers must be accurately estimated in advance; small estimation errors may lead to the loss of many relevant documents located in the ignored servers.

Hernandez and Kambhampati [2005] introduced COSCO for management of duplicates at selection time. The system estimates the overlap between different bibliographic servers and avoids selecting pairs of servers that appear to have high overlap for a query. They use CORI [Callan et al., 1995] as a benchmark and show that COSCO finds more relevant documents for a given number of servers than does CORI. However, COSCO only considers exact duplicates.

For the management of duplication at the result-merger stage, pairs of such documents are identified within the merged result list, and are purged before the results are returned to the user. It is worthwhile to perform duplicate management at the result-merger stage even if steps to manage it are taken at selection time, because this allows duplicate and near-duplicate documents to be identified even if their corresponding servers do not have a significant rate of overlap.

Although distributed search over servers with overlapping collections has been acknowledged as a problem [Allan et al, 2003, Meng et al., 2002], no serious attempt has been made to resolve the issue. ProFusion [Gauch et al., 1996], MetaCrawler [Selberg and Etzioni, 1997], and Grouper [Zamir and Etzioni, 1999] attempt to eliminate duplicate documents from the final results, by aggregating results that point to the same location according to their URL. This method has several deficiencies: it is ineffective for identical documents with different URLs (such as mirrored documents), for near-identical documents, and for domains in which a unique document identifier such as a URL is not available. Clearly, more sophisticated techniques for duplicate management are desirable.

3 Merge-Time Duplicate Management for DIR

Management of within-collection redundancy has been a subject of active research, with a range of techniques having been proposed [Manber, 1994, Brin et al., 1995, Broder et al., 1997, Fetterly et al., 2003, Bernstein and Zobel, 2004] However, management of redundancy *between* collections as in the case of DIR is subject to additional constraints. In particular, since collections are not centrally managed, it may not be practical to use a preprocessing approach to redundancy management; rather, it must occur at query time based on additional document information transmitted to the broker. Thus, management of near-duplicate documents is highly sensitive to both time (because it must be done on the fly) and bandwidth (because transmission of additional information in a distributed environment may be neither cheap nor fast).

We now describe an existing technique for near-duplicate detection that could be reasonably applied to the problem scenario we have described, though to our knowledge this has not previously been attempted. Deterministic term extraction techniques [Chowdhury et al., 2002, Ilyinski et al., 2002, Cooper et al., 2002, Conrad et al., 2003, Kolcz et al., 2004] extract a subset of terms from a document, from which a hash is produced. It is claimed that, if the terms are carefully chosen, near-duplicate documents are likely to have the same hash, whereas it is extremely unlikely that two dissimilar documents will hash to the same value. Thus, two documents with the same hash will be considered as near-identical, whereas documents with different hashes will be considered to be different. A critical consideration for the success of deterministic term extraction systems is the way in which the terms to be extracted from a document are selected. For example, in the I-Match system [Chowdhury et al., 2002], only those terms with the highest inverse document frequency are selected for the hash. Other approaches vary in detail, but have the same basic principle of selecting words of high significance.

Deterministic term extraction systems have several properties that make them appealing from the perspective of near-duplicate management in DIR. The fact that each document only has a single representative hash means that the additional bandwidth requirement in using deterministic term extraction is minimal; it also means that comparisons between documents at run-time are fast, as all

that is required is an equality test between the two representative hashes. The hashes themselves are relatively cheap to compute at index time. It is plausible that such features could be deployed for near-duplicate detection in a distributed context, and that they would be reasonably effective.

While empirical tests seem to show that deterministic term extraction techniques are effective in practice, some doubts remain. There has been no theoretical analysis that would verify claims of robustness; it always remains the case that a single difference between a pair of documents, in a critical term, can cause the system to fail to identify them as near-duplicates. Pugh and Henzinger [2003] and Kolcz et al. [2004] fortify their technique by extracting several hashes based on different term subsets. This has the effect of making the technique less appealing for DIR, as it increases the bandwidth cost and the comparison cost at run-time.

There is a further problem that arises in the domain of DIR. As noted above, most of the deterministic term extraction systems described in the literature rely on inverse document frequency or some similar collection statistic. In a distributed environment, however, gathering global collection statistics presents a significant challenge. Without such statistics, identical documents on different servers may produce different hashes. Thus, it is probably necessary to use a term extraction heuristic that does not use collection statistics. It is not apparent whether the technique can remain effective with such a heuristic.

An alternative approach is to use *chunks*. Chunk-based document fingerprinting is a technique for detecting near-duplicate documents that has been successfully used for applications such as filesystem-level duplicate detection [Manber, 1994], plagiarism detection [Lyon et al., 2001], copyright enforcement [Brin et al., 1995], enterprise version management [Conrad et al., 2003], and optimisation of indexing [Broder et al., 1997] and search [Bernstein and Zobel, 2005] on the web. For a detailed description of chunk-based fingerprinting, see Hoad and Zobel [2003] or Bernstein and Zobel [2004]. In brief, it operates by parsing documents into strings of contiguous text, known as *chunks*, and comparing the number of identical chunks shared by a pair of documents.

When a sliding window is passed over a document to extract a set of fixed-length overlapping chunks, these chunks are known as *shingles*. Broder et al. [1997] define a measurement known as *resemblance* for quantifying the level of identity between a pair of documents based on their shingle-sets. Resemblance is a symmetric measure that is maximized only when two documents are identical, defined as follows:

$$R(S,T) = \frac{|\hat{S} \cap \hat{T}|}{|\hat{S} \cup \hat{T}|}$$

where $\hat{S}$ is the set of shingles extracted from a document S. Resemblance is a useful and robust measure, and has been used in a number of applications. Because resemblance ranges smoothly between 0 and 1, it is sometimes interpreted as 'percentage similarity'. However, a single edit to a document will disrupt multiple shingles, potentially causing the reported resemblance values for a pair of

documents to be significantly lower than a human would intuitively expect of a percentage similarity. Caveat emptor.

Fetterly et al. [2003] describe an unbiased technique for estimating resemblance between a pair of documents using a feature of constant size; we refer to this technique as *minimal-chunk sampling*. Minimal-chunk sampling relies on the availability of a class of hash functions that are *min-wise independent* [Broder et al., 1998]. Min-wise independence states that the class of permutations is unbiased with respect to the identity of the first element in the permutation. In the context of hashing, it means that any value in an arbitrary set has an equal probability of hashing to a value that is the lowest in the set.

The minimal-chunk sampling heuristic with resolution ρ (a positive integer) must have access to ρ hash functions from a min-wise independent family. In practice, a family of hash functions is most commonly defined by a single algorithm parameterised by some value (frequently a seed). A particular function in the family is thus defined by the parameter passed to the general algorithm. Thus, the set of ρ min-wise independent hash functions is represented by a single hash function and an array of ρ independently chosen random seeds.

The set of shingles in a document is passed through each of the ρ hash functions, and the minimal hash under each permutation is stored. The result of the process is a vector of ρ hash values. The min-wise independence property of the hash functions means that, for two documents with resemblance r, the probability of the hash value in any given position on their corresponding vectors is (excluding collisions) independently r. This property of the minimal-chunk sampling heuristic allows us to easily analyze its performance, as the distribution of the number of matching hashes between a pair of documents with resemblance r is binomial parameterised by ρ and r.

Fetterly et al. [2003] use $\rho = 84$ to generate a vector of 84 hashes of 32 bits each. While such a vector would undoubtedly be a high-quality feature for detecting near-duplicate documents, at 336 bytes it is rather large. Even if ρ were reduced substantially, the size of the vector would still need to be several dozen bytes to be reliable for identification of near-duplicate documents. In the context of DIR, an overhead of this size for each result in the ranked list could prove burdensome; a more compact representation is desirable.

4 Grainy Hash Vectors

In this section, we describe our novel document-digest feature for near-duplicate document identification, the *grainy hash vector* (GHV). Whilst the GHV is a derivation of the minimal-chunk sampling technique described above, its innovative implementation combines the benefits of regular minimal-chunk sampling with those of deterministic term extraction. In particular, grainy hash vectors:

- Fit into a single machine word of either 32 or 64 bits;
- Have analyzable theoretical properties;
- Use bit-parallelism to allow fast comparison between vectors;
- Are robust in the presence of small differences between a pair of documents.

This combination of attributes makes the GHV an attractive feature in a DIR domain. A GHV parameterised by n and w is a n-bit vector consisting of ρ w-bit hashes, where

$$\rho(n, w) = \left\lfloor \frac{n}{w} \right\rfloor \tag{1}$$

In general, we want w to be a factor of n in order to avoid wasting space in the vector. Each of the hashes in the GHV is produced using the minimal-chunk sampling technique. For example, a GHV with $n = 32$ and $w = 2$ would consist of 16 2-bit hashes, each produced using the minimal-chunk sampling technique. By using small w, it is possible to pack a large number of hashes into a small space. The tradeoff is that the probability of collision at each point in the vector becomes non-negligible. However, we demonstrate that with an appropriate match threshold GHVs can provide powerful discrimination of near-duplicate documents even with $n = 32$.

When w is small, there will be an overwhelming bias in the value of the minimal-chunk hash; with $w = 1$, for instance, one would expect the minimal hash to have value 0 in most cases. In order to remove this bias, a much larger w — for example 32 — is used for the initial ordering. Once the minimal hash value has been identified, the w least-significant bits are stored in the GHV.

For two documents with resemblance r, the probability ϕ of a match between the hashes at a given position in the GHV with vector-width w is given by:

$$\phi(r; w) = r + (1 - r)(2^{-w}) \tag{2}$$

The first component of this function follows directly from the property of min-wise independent hash functions that states that the probability of a hash match between two documents with resemblance r is r; the second component is the probability that, in the case that the two source chunks are not the same, a collision renders them identical in the hash vector.

Given that these probabilities are independent for each field in the GHV, we note that the number of matching fields between a pair of documents with resemblance r is governed by a binomial distribution, parameterised as follows:

$$Bi(\rho(n, w), \phi(r; w)) \tag{3}$$

The following properties are thus directly derivable from the distribution:

$$P(X = k) = \binom{\rho}{k} \phi^k (1 - \phi)^{\rho - k} \tag{4}$$

$$E(X) = \rho\phi \tag{5}$$

$$\sigma^2 = \rho\phi(1 - \phi) \tag{6}$$

In practice, GHVs would be used with a threshold to determine whether a given pair of documents should be considered near-duplicates. For example, we may have $n = 32$, $w = 2$ and a threshold of 14, meaning that at least 14 of

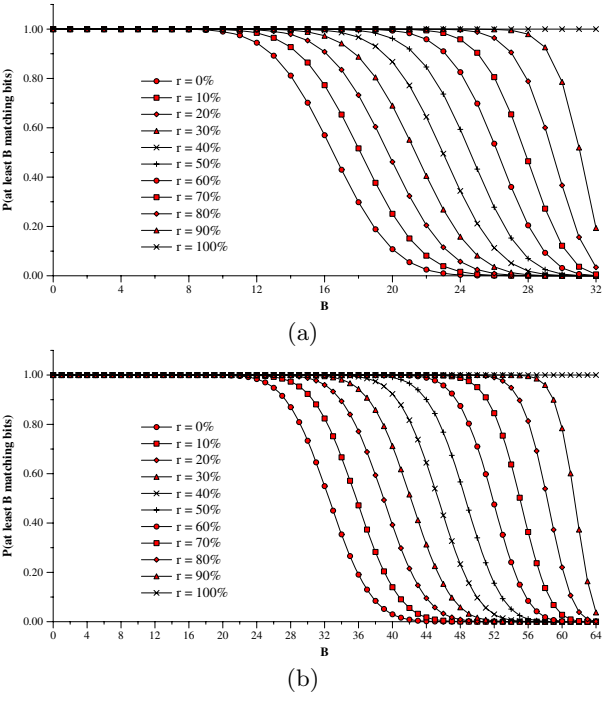


Fig. 1. Probability that the number of matching fields exceeds a value B for (a) 32-bit and (b) 64-bit hash vectors, $w = 1$ and various r

the $\rho(n, r) = 16$ hashes in the corresponding vectors of a pair of documents must match for them to be considered near-duplicates. Thus, we are interested in finding a set of parameters for our GHV that minimises the error level.

The issue of whether a pair of documents should be considered near-duplicates is a difficult one; circular and unsatisfactory definitions of what constitutes a duplicate or near-duplicate has weakened several previous works in the area [Zobel and Bernstein, 2006]. Addressing the issue of what constitutes a near-duplicate is beyond the scope of this paper; we assume only that the classification of document-pairs is based on a resemblance threshold.

Figure 1 displays curves, derived from the binomial distribution, showing the probability that the number of matching fields exceeds a given value for various document resemblance values. While these graphs use $w = 1$, similar curves obtain for different values of w. The steeper the gradient of the curves, the more precise the separation between documents with high levels of resemblance and those with low resemblance; as expected, the 64-bit GHVs are superior in this respect.

The single-word nature of grainy hash vectors allows us to compute the number of matches between a pair of vectors quickly by taking advantage of bit parallelism in current processors. We can use the bitwise exclusive-or (XOR)

operator to identify points in a pair of vectors that do not match. For a GHV of one machine word, this operation is completed in a single instruction.

In the case of $w = 1$ we need only count the number of 1-bits in the XOR vector — the *population function* — to determine the number of mismatches between the two vectors. While current hardware does not generally implement this function directly, it can still be computed efficiently either with lookup tables or using a 'divide-and-conquer' approach such that in Warren [2002]. If w is greater than one then the situation becomes more complicated. We must count the number of fields in which there is at least one mismatched bit. If w is a power of two then we can modify the XOR vector so that the number of 1-bits equals the number of mismatches in $O(\log w)$ time by using shift operators to collapse all bits in each field onto a single bit.

5 Experimental Evaluation

To validate the speed and effectiveness of GHVs in removing duplicates and near-duplicates from result lists, we ran experiments using the Associated Press (AP) newswire data, a subset of the TIPSTER collection used for the TREC ad hoc retrieval track [Harman, 1993]. The AP collection consists of 237,569 documents totaling 729 MB. For simplicity, we used results from a centralised IR system; however, the results are applicable to a DIR context.

For each document in AP we created 32- and 64-bit GHVs with w of 1, 2, 4, and 8. We then used DECO [Bernstein and Zobel, 2004] to calculate the resemblance between all pairs of documents in the AP collection, and the Zettair software[1] to create a ranked result list of depth 1000 on the AP documents for each of the TREC topics 51-100. For each topic we loaded the GHVs for the 1000 documents appearing in the ranked result list. Each pair of GHVs in this list was then compared using the algorithm described earlier, resulting in 499,500 comparisons. Pairs where the number of mismatching fields was below a threshold t were returned; the process was repeated for all possible values of t.

GHV efficiency. It took approximately 0.1 seconds to perform the 499,500 GHV comparisons required for full pruning of 1000 results on a standard Pentium 4 desktop computer. This figure did not vary significantly depending on whether the GHVs were 32 or 64 bits in length, or between values of w. In comparison to the significant other costs present in a DIR system — in particular, network latency — we argue that this is a reasonable cost to bear in order to improve the user experience.

Note that the timings presented above are pessimistic, based on an assumption that the GHVs for the full ranked list of 1000 documents need all be compared to each other. This is not generally necessary if the comparisons are undertaken in a 'lazy' fashion. Many search systems present results incrementally, often ten at a time; we can take advantage of this by only undertaking the GHV comparisons necessary to present the next page of results. Much of the time — depending

[1] `http://www.seg.rmit.edu.au/zettair/`

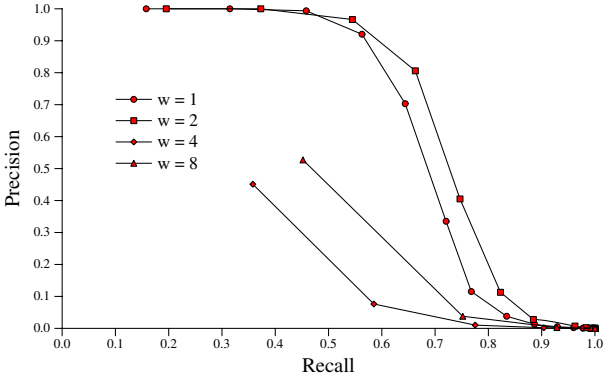


Fig. 2. Recall-precision curves at $r = 0.58$ for various w and GHVs of size 32 bits

upon the level of duplication in the ranked list and the persistence of the user — this can reduce the number of comparisons required from hundreds of thousands to just hundreds. Thus, we assert that the typical cost of merge-time GHV comparison is negligible in comparison to the cost of other stages of the DIR process.

GHV accuracy. Based on a user study, Bernstein and Zobel [2005] suggest 0.58 as a good resemblance threshold for *conditional content equivalence*: a relation between a pair of documents such that a user perceives them as being redundant with reference to the query. We adopt this value as our threshold: we consider all documents above this value to fulfil the condition of near-duplication, and all documents below to not fulfil the condition.

Figure 2 shows mean recall-precision curves for the 50 TREC topics using 32-bit GHVs and various values of w, where all document-pairs with resemblance above 0.58 are considered 'relevant', and all other pairs non-relevant. As can be seen, $w = 2$ clearly dominates, followed by w values of 1, 8, and 4. For $w = 4$ and $w = 8$, the precision level is inadequate for use even at the lowest threshold levels. For $w = 1$ and $w = 2$, precision is adequate up to a recall level of approximately 0.5, meaning that we can use 32-bit GHVs to detect about half the near-duplicate document pairs in the result list without introducing significant numbers of false positives. Based on these results, we recommend $w = 2$ and a threshold of 14 out of the 16 features matching.

Figure 3 shows the same curves when 64-bit GHVs are used. As expected, these curves are significantly better than the corresponding curves for 32-bit GHVs. Interestingly, the field widths dominate each other in the same order, with $w = 2$ performing best, followed by 1, 8, and 4. With $w = 2$ it is possible to achieve recall values of approximately 0.7 before precision moves below 1. Based on these results, we recommend $w = 2$ and a threshold of 24 out of the 32 features matching. It remains unclear why the $w = 2$ value performs best.

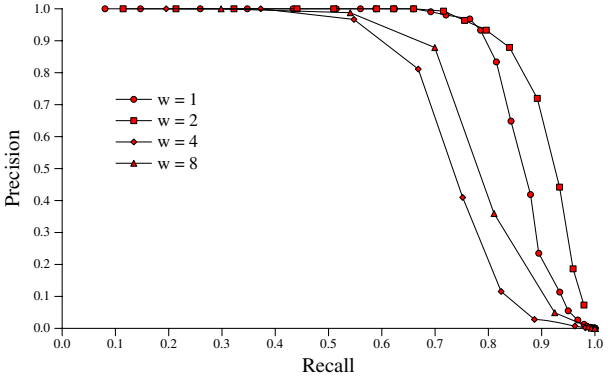


Fig. 3. Recall-precision curves at $r = 0.58$ for various w and GHVs of size 64 bits

The values represented in Figures 2 and 3 are probably underestimates of the true effectiveness one could expect of GHVs in a distributed environment. The main reason is that we performed our analysis on the AP collection of newswire data. While this collection contains a moderate degree of duplication, it is less redundant than many data sources, in particular the web. Furthermore, we have not explicitly examined the issue of overlapping collections, which would result in further increase in duplication in a typical result pool. As the rate of duplication in the lists rises, the measured effectiveness of GHVs improves, as more true positives will be identified but the false positive rate will not increase. However, it is apparent even from these experimental results that GHVs can effectively screen out all exact duplicate documents and many near-duplicate documents in a cooperative DIR system.

6 Conclusions

Management of document duplication has been cited as one of the major challenges facing the field of DIR [Allan et al, 2003]. Duplication of documents occurs in DIR due to the same phenomena that cause it to be a problem in centralised IR [Bernstein and Zobel, 2004], but is compounded by the effect of collection overlap. Despite the seriousness of the issue, it has seen little in-depth investigation, and techniques typically used for duplicate management in centralised IR systems do not translate well to the DIR domain.

We have introduced a new representation, the *grainy hash vector* (GHV), that can be deployed in cooperative DIR systems for efficient and accurate merge-time duplicate detection. GHVs are able to detect near-duplicates as well as exact duplicates, have well-defined mathematical properties, and can be independently constructed at index time at each site in the DIR system for transmission to the broker at query time. We demonstrate empirically on the TREC AP collection that GHVs can be used to efficiently and effectively identify duplicate and

near-duplicate document pairs at merge time. GHVs are an excellent mechanism for management of duplication in cooperative DIR.

Acknowledgements. This research was supported by the Australian Research Council.

References

J. Allan et al. Challenges in information retrieval and language modeling: report of a workshop held at the center for intelligent information retrieval, University of Massachusetts Amherst, september 2002. *SIGIR Forum*, 37(1):31–47, 2003.

Y. Bernstein and J. Zobel. A scalable system for identifying co-derivative documents. In *Proc. String Processing and Information Retrieval Symposium*, pages 55–67, Padova, Italy, 2004.

Y. Bernstein and J. Zobel. Redundant documents and search effectiveness. In *Proc. ACM CIKM Conf.*, pages 736–743, Bremen, Germany, 2005.

S. Brin, J. Davis, and H. García-Molina. Copy detection mechanisms for digital documents. In *Proc. ACM SIGMOD international conference on Management of Data*, pages 398–409, San Jose, California, 1995.

A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proc. ACM symposium on Theory of computing (STOC)*, pages 327–336, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-962-9.

J. Callan and M. Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems*, 19(2):97–130, 2001.

J. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *Proc. Int. ACM-SIGIR Conf.*, pages 21–28, Seattle, Washington, 1995.

A. Chowdhury, O. Frieder, D. Grossman, and M. C. McCabe. Collection statistics for fast duplicate document detection. *ACM Transactions on Information Systems*, 20 (2):171–191, 2002.

J. G. Conrad, X. S. Guo, and C. P. Schriber. Online duplicate document detection: Signature reliability in a dynamic retrieval environment. In *Proc. ACM-CIKM Conf.*, pages 443–452, New Orleans, Louisiana, 2003.

J. W. Cooper, A. R. Coden, and E. W. Brown. Detecting similar documents using salient terms. In *Proc. ACM-CIKM Conf.*, pages 245–251, McLean, Virginia, 2002.

D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In *Proc. first Latin American Web Congress*, pages 37–45. IEEE, 2003.

S. Gauch, G. Wang, and M. Gomez. ProFusion: Intelligent fusion from multiple, distributed search engines. *J. Universal Computer Science*, 2(9):637–649, 1996.

L. Gravano, C. K. Chang, H. Garcia-Molina, and A. Paepcke. STARTS: Stanford proposal for Internet meta-searching. In *Proc. ACM SIGMOD international conference on Management of Data*, pages 207–218, Tucson, Arizona, 1997.

D. Harman. Overview of the first TREC conference. In *Proc. ACM-SIGIR Conf.*, pages 36–47, Pittsburgh, Pennsylvania, 1993.

T. Hernandez and S. Kambhampati. Improving text collection selection with coverage and overlap statistics. In *Proc. Int. Conf. on World Wide Web*, pages 1128–1129, Chiba, Japan, 2005.

T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarised documents. *J. the American Society for Information Science and Technology*, 54(3):203–215, 2003.

S. Ilyinski, M. Kuzmin, A. Melkov, and I. Segalovich. An efficient method to detect duplicates of web documents with the use of inverted index. In *Proc. Int. Conf. on World Wide Web*, Honolulu, Hawaii, 2002.

A. Kolcz, A. Chowdhury, and J. Alspector. Improved robustness of signature-based near-replica detection via lexicon randomization. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 605–610, Seattle, WA, 2004.

C. Lyon, J. Malcolm, and B. Dickerson. Detecting short passages of similar text in large document collections. In *Proc. Conf. on Empirical Methods in Natural Language Processing*, Philadelphia, Pennsylvania, 2001.

U. Manber. Finding similar files in a large file system. In *Proc. USENIX Winter Technical Conf.*, pages 1–10, San Fransisco, CA, 17–21 1994.

W. Meng, C. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Computing Surveys*, 34(1):48–89, 2002.

H. Nottelmann and N. Fuhr. Evaluating different methods of estimating retrieval quality for resource selection. In *Proc. Int. ACM-SIGIR Conf.*, pages 290–297, Toronto, Canada, 2003.

A. L. Powell and J. French. Comparing the performance of collection selection algorithms. *ACM Transactions on Information Systems*, 21(4):412–456, 2003.

W. Pugh and M. H. Henzinger. Detecting duplicate and near-duplicate files (United States Patent 6,658,423), 2003.

E. Selberg and O. Etzioni. The MetaCrawler architecture for resource aggregation on the Web. *IEEE Expert*, (January–February):11–14, 1997.

L. Si and J. Callan. Unified utility maximization framework for resource selection. In *Proc. ACM-CIKM Conf.*, pages 32–41, Washington, D.C., 2004.

L. Si and J. Callan. Relevant document distribution estimation method for resource selection. In *Proc. ACM-SIGIR Conf.*, pages 298–305, Toronto, Canada, 2003.

H. S. Warren, Jr. *Hacker's Delight*. Addison Wesley, 2002.

O. Zamir and O. Etzioni. Grouper: a dynamic clustering interface to web search results. In *Proc. Int. Conf. on World Wide Web*, pages 1361–1374, Toronto, Canada, 1999.

J. Zobel and Y. Bernstein. The case of the duplicate documents: Measurement, search, and science. In *Proc. Asia-Pacific Web Conf.*, pages 26–39, Harbin, China, 2006.

Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory

Simon J. Puglisi[1], W. F. Smyth[1,2], and Andrew Turpin[3]

[1] Curtin University of Technology, Perth, Australia
puglissj@cs.curtin.edu.au
http://www.computing.edu.au/~puglissj
[2] McMaster University, Hamilton, Canada
smyth@mcmaster.edu
[3] RMIT University, Melbourne, Australia
aht@cs.rmit.edu.au

Abstract. Recent advances in the asymptotic resource costs of pattern matching with compressed suffix arrays are attractive, but a key rival structure, the compressed inverted file, has been dismissed or ignored in papers presenting the new structures. In this paper we examine the resource requirements of compressed suffix array algorithms against compressed inverted file data structures for general pattern matching in genomic and English texts. In both cases, the inverted file indexes q-grams, thus allowing full pattern matching capabilities, rather than simple word based search, making their functionality equivalent to the compressed suffix array structures. When using equivalent memory for the two structures, inverted files are faster at reporting the location of patterns when the number of occurrences of the patterns is high.

1 Introduction

The problem of finding an m character pattern $P[1 \ldots m]$ in an n character text $T[1 \ldots n]$ recurs frequently as a component in larger algorithms, and as a stand alone problem. When n and m are sufficiently small that a $O(n+m)$ time search is tolerable, then one of the gamut of online string matching algorithms such as KMP or Boyer-Moore solve the problem effectively [30]. However, when n is very large compared with m, or when the text is to be searched many times, then algorithms with running times linear in the text size become less attractive, and it is worthwhile preprocessing the text to build an index to aid search.

If a suffix tree data structure is built from T, then patterns can be located in the text in time $O(m + occ)$, where occ is the number of occurrences of the pattern P in T [23,31]. The problem with a suffix tree, however, is that it typically requires upwards of $12n$ bytes of memory [16], assuming that characters in T and P are drawn from a small alphabet like the ASCII alphabet of 256 characters, and even more space for larger alphabets. Using only $5n$ bytes of memory, Manber and Myers introduced the suffix array, which allows a search for P in T in $O(m \log n + occ)$ time, or $O(m + \log n + occ)$ if an extra $4n$

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 122–133, 2006.

bytes of memory are used [20]. The search time for P in T using a suffix array was improved to $O(m + occ)$ time by Sim et al., by the use of auxiliary data structures that require 1 to $4n$ bytes extra depending on alphabet size [8,29]. The time to construct suffix arrays has also been reduced in recent years to that of suffix trees, $O(n)$, making them very competitive with the suffix tree structure [26].

Recently the memory requirements of suffix arrays have been reduced through the use of compression techniques, intuitively similar to the Burrows-Wheeler Transform [6], but in practice implemented quite differently. Using these structures, the suffix array can typically be stored in less than $2n$ bytes [11,13,17,27]. What is even more remarkable is that some of these algorithms produce a *self-index*; that is, an index that can also be used to recover the original text. Hence there is no need to store the original text in addition to the compressed suffix array. In general these structures take $O(m + C + Locc)$ time to find all the positions where P occurs in T, where C refers to the time taken to count the number of times P occurs and L is the time required to "decode" one position of occurrence from the compressed structure. The plethora of different approaches for compressing the suffix array offer tradeoffs for C, L and final index size.

While these new compressed self-indexes are indeed remarkable achievements, one of their key rivals for solving the pattern matching problem has been overlooked or dismissed by some authors: the explicit index, or the inverted file. The inverted file data structure has long been used as the underlying data structure of choice for the *restricted* pattern matching problems where P is a unit of text, such as a word or phrase, and T is a natural language [34]. It is a fundamental technology underlying all Internet search engines such as Google, Yahoo and MSN Search. In the restricted pattern matching problem, a vocabulary of possible P's must be known in advance, hence allowing an inverted file to be constructed. While this may seem to prevent inverted files from being used to solve the unrestricted pattern matching problem—that is, where P can be of any form—this is not the case. By assuming a vocabulary of all possible q-grams, where a q-gram is a string of characters of length q, inverted files can be used to solve the general pattern matching problem. Indeed, they have been successfully used to index P in one of the most popular and important applications of unrestricted pattern matching of our time: the BLAST application for searching genomic data [2,25].

Although inverted files may require $O(m + n)$ time to locate P in T, this is usually only realised on pathological data. In practice, the running times for restricted pattern search using inverted files are extremely fast. It is unknown how the speed of inverted files on the unrestricted pattern matching problem compare to that of the new compressed self-indexes described above. In this paper we present the first empirical comparison between a compressed inverted file on q-grams and a compressed suffix array structure for the in-memory, unrestricted pattern matching problem. We show that the compressed inverted file is indeed faster than its counterpart when the number of occurrences of P is high, but that the compressed suffix array is superior for searching when occ is low.

```
          1111111111222222222233333333334444444444555555555566666666
12345678901234567890123456789012345678901234567890123456789012345678
of all the saws I ever saw, I never saw a saw saw like that saw saws
```

Fig. 1. An example text used for illustration throughout the paper. Numbers indicate the position of each character, for example `a` occurs at position 44.

An alternate technology, which we do not consider in this paper, is the Ziv-Lempel index on q-grams published by Kärkkäinen [15].

2 Background

In this section we describe inverted files and compressed suffix arrays in turn.

2.1 Inverted Files

An inverted file is a data structure similar to a book index. Textual units of a pre-determined form are extracted from T, for example words or q-grams, and then the position of each occurrence of each unit is stored in an *inverted list* for that pattern. For example, for the text in Figure 1, assuming a textual unit of a 3-gram, the inverted list for `saw` would be $\{12, 24, 37, 43, 47, 61, 65\}$; and for `ver` $\{20, 33\}$. In order to locate the lists quickly, all of the textual units are stored in a hash table with a pointer to the inverted list for that unit.

Without data compression, the size of the inverted index is obviously several times larger than the text; for our q-gram units there is a 4 byte pointer per text position, and some entries in a hash table, so a total space usage of at least $4n$ bytes. When the lists are compressed, however, space savings accrue that make inverted files the data structure of choice when searching for words in English text [34]. The preferred method for compressing inverted lists is to take differences between entries in the lists (so the list for `saw` would become $\{12, 12, 13, 6, 4, 14, 4\}$) and then code these small integers with an appropriate coding scheme [34]. In our experiments below we use the Simple-9 coding scheme which, while not the best available, gives a good tradeoff between compression levels and decoding speed [3].

The steps outlined in Figure 2 are followed to search for pattern P using an inverted file. There are several nuances hidden by the pseudo code. Firstly, there are various methods for computing a cover of P in Step 4. In the experiments reported below we compute the set containing all the distinct substrings of P of length q. This means we make $m - q + 1$ hits on the hash table but we are guarantee to have the shortest inverted list at Step 8. As an alternative one could use a simple greedy left-to-right match, taking the q-grams $P[1 \ldots q]$, $P[q + 1 \ldots 2q]$, and so on, with the final q-gram being $P[m - q + 1 \ldots m]$.

The second nuance is the early termination of the loop that intersects the inverted lists for each q-gram to generate the list of candidate matches C (Step 9). It was observed by Zobel et al. [35] that it is often beneficial to stop intersecting lists and go straight to the text when the length of candidates fell below some

1) If $m < q$ then
2) Set $S \leftarrow \{Ps | s =$ any substring drawn from Σ of length $q - m\}$.
3) else
4) Set S to be a set of strings that cover P.
5) Let p_i be the starting position of $s_i \in S$ in P.
6) For each $s_i \in S$, set ℓ_i to the length of inverted list for s_i.
7) Sort S into ascending order by ℓ_i's.
8) Initialise C to the inverted list for s_1 and remove s_1 from S.
9) While S is not empty and $|C| > k$ do
10) Set $C' \leftarrow$ elements of s_1 less p_1.
11) Set $C \leftarrow C \cap C'$.
12) Remove s_1 from S.
13) Check for an occurrence of P beginning at each position $T[c \in C]$.

Fig. 2. Algorithm for searching for a pattern $P[1 \ldots m]$ using an inverted file constructed on q-grams from a text T

parameter k. Note that if $k = 0$, then the list of candidates C contains all the locations of P, and there is no need to check the text. If the text is not required for some other reason (for example, returning to the user) then the text need not be stored which is a substantial memory saving. In our experiments below we use $k = 2000$ for DNA data and $k = 10000$ for protein data and English text.

The time required for the search algorithm outlined in Figure 2 depends markedly on the input to the algorithm. If $m > q$, then a maximum of m/q lists will be examined, allowing Steps 1 to 7 to complete in $O(m/q \log(m/q))$ time. On the other hand, if the pattern is shorter than q, the number of q-grams is up to $|\Sigma|^{q-m}$, and so this situation should be avoided for large alphabets by keeping q small. Each text position can only occur in one inverted list, so the total number of inverted list entries processed in Steps 9 to 12 is at most n, hence the loop completes in $O(n)$ time. In practice, however, it is hoped that this will be much closer to $O(occ)$ time. Step 13 requires at least $O(\max(occ, k))$ time.

An important variation on the classical inverted file described above is the *block-addressing* inverted file [21,24]. The idea is instead of storing a pointer i to every occurrence of a q-gram v, one chooses a block size b, and stores only unique values of $\lfloor i/b \rfloor$ in each inverted list – in other words only pointers to the blocks of the text in which v occurs are stored. For example, if we chose $b = 10$ then the list for `saw` would become $\{1, 2, 3, 4, 6\}$; and for `ver` $\{2, 3\}$.

The benefits of block addressing are twofold: the inverted lists are shorter (because only one pointer per occurrence per block is stored) and the gaps between stored blocks are likely to be smaller, making the gapped lists more compressible. The cost is that some sequential searching of the text is always required to find the exact positions of occurrences inside matching blocks, and for this reason we must always store the text explicitly, or in some accessible way.

One way to reduce the burden of storing the text is to store it in a compressed, but searchable form. This idea has been shown to be very effective for word

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	50	51	52	53	54	55	56	57	58	...	68	69
$S[i]$	69	16	28	40	3	18	50	30	36	46	42	60	23	64	11	55	7	27	...	15	37	47	43	61	24	65	12	59	...	67	14

Fig. 3. Suffix array S for the example string in Figure 1. The underlined regions are the same run of values but offset by one.

indexing inverted files [9,24]. It may be possible that those ideas could be adapted for the q-gram case, though we do not compress the text for the inverted file experiments reported in this work.

2.2 Compressed Suffix Arrays

The suffix array of text T is defined as an array S containing a permutation of the integers $1 \dots n$ such that the suffix $T[S[i] \dots n]$ is lexicographically not larger than the suffix $T[S[i+1] \dots n]$. In effect, S is a sorted list of pointers to every suffix of T. Figure 3 shows portions of S built for the example text in Figure 1. Note that it is convenient for suffix array algorithms to have a special terminating character \$ which is smaller than all other characters added to the end of T, hence the first entry in S in the figure is 69, the position of \$ in T. The next 16 entries all point to suffixes beginning with a space, with $T[S[18]]$ being the comma. The middle section shown, $S[51 \dots 57]$ shows the pointers to suffixes beginning with `saw`.

In order to use S to search for P we can employ the algorithm of Manber and Myers [20], which is a simple binary search for the suffix beginning with the string lexicographically less than P, and the suffix beginning with the string lexicographically greater then P. For example, in Figure 3, if the pattern was $P =$ `saw`, we would binary search to positions 51 and 57 in S which are the boundaries of the region of pointers to suffixes beginning with `saw`. For each step of the binary search we may have to do m comparisons, so the total time is $O(m \log n)$. If an extra array of size $4n$ bytes, called the LCP array (longest common prefix), is stored, then search times can be reduced to $O(m + \log n)$. By enhancing the suffix array with various other auxiliary arrays Abouelhoda et al., [1] and Sim et al.,[29] show how to improve search time to $O(m)$ for texts with constant sized alphabets.

The $5n$ byte requirement of text plus suffix array (assuming an alphabet of 256 characters) can be significantly reduced by compressing the "self repetitions" in the suffix array. For example, in Figure 3 the entries in $S[9 \dots 15]$ are all one less than the entries in $S[51 \dots 57]$ (underlined). There is a myriad of techniques for achieving space savings which are elegantly surveyed by Navarro and Mäkinen [18]. It also turns out that the connection between the BWT and S can be exploited so that you get a *self-index*. A self-index is capable of reproducing any substring of the text, and so can be stored in place of the text.

For the experiments in this paper we use an implementation variant of the Sadakane's "succinct suffix array" (SSA) [28] described and implemented by Makinen and Navarro [17, Section 5]. This index was selected because previous experiments indicate it is representative of the state of the art in compressed

suffix structures [17]. It was also the fastest index for which we could find publicly available code [12]. Asymptotically the SSA counts occurrences in $O(H_0 m)$ time and then retrieves each position in $O(H_0 \log^{1+\varepsilon} n)$ time, where H_0 refers to the zero-order empirical entropy of the text [17,22]. The total search time is thus $O(H_0 m + occ H_0 \log^{1+\varepsilon} n)$. Finally we remark that to reduce space further, instead of indexing every suffix, we can sample (index) only every srth suffix – sr is called the *sample rate*. This is the main way to trade index size for search time in many compressed suffix structures, including the SSA.

3 Method

Our investigations are carried out using three datasets as summarised in Table 1, where the "H_0" column is the zero-order entropy of the collection, and the "Occurrences" column indicates the average number of occurrences of each pattern in the collection. Each of these three datasets is now discussed in turn.

DNA A common, current string processing task employed by molecular biologists is sequence alignment, where a query sequence is matched against a database of possible sequences and the best matches reported. While not an exact string matching problem, the first phase of algorithms such as BLAST [2] and its recent variants [7] perform an exact matching task on small subsequences of the query sequence. These results are then post-processed to find the true alignment. In BLAST, currently the most popular tool for the task, 80% of the time in performing an alignment is spent in exact matching of all 11 character subsequences of the query sequence [7].

Accordingly, this DNA dataset contained chormosones 18 and 19 from the human genome [10], and 31,165 patterns of length 11, derived from a set of real sequences issued to BLAST in a laboratory at the Royal Womens Hospital, Melbourne. The patterns are all 11 character substrings taken from the real sequences, as would be used in the BLAST algorithm.

PROT Another common pattern matching task studied in bioinformatics is protein alignment. The process is similar to the DNA task described above, but the alphabet of possible symbols is 20, rather than 4. As BLAST finds exact matches of 2-grams in the first step of its protein alignment algorithm, we tested

Table 1. Collections and pattern sets used

Dataset	Collection			
	Source	Size (Mb)	H_0	
PROT	Genbank non-redundant database	150	4.21	
WSJ	TREC newswire	150	4.53	
DNA	DNA Human chromosone 18 and 19	134	2.21	
	Pattern Set			
	Source	m	Occurrences	Number
PROT	Random	2.0	523,296	100
WSJ	Words in titles of TREC topics 200-450	8.2	4,861	2651
DNA	Royal Womens Hospital, Melbourne	11.0	185	31,165

1000 random patterns of length 2 on a 150Mb subset of the Genbank non-redundant protein set [4,5]. Note that in the figures reported below on the whole pattern set, this pattern set is broken into 10 blocks of 100 patterns each, so in fact the timing is the average for 100 patterns, not the entire 1000.

WSJ One of the most common uses of pattern matching is in searching text. Accordingly, we include a data file of newspaper articles from the Wall Street Journal subcollection of the TREC collection [14], and 2651 English words as patterns. The words were extracted from the Title field of the TREC topics number 200 through 450. Only words with six or more characters were included in the pattern set, some did include punctuation characters, and some words appeared in the pattern set more than once.

Experimental Setup

All experiments were conducted on an otherwise unoccupied 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O6 option. Times were recorded with the standard Unix `getrusage` function. All running times given are the average of at least five runs and do not include time spent loading the index from disk. Index sizes for the inverted file include the size of the original, uncompressed text.

4 Results

Figure 4 shows the running times and index sizes for the three real pattern sets on the three data sets. In order to equate IF sizes with SSA sizes as much as possible, the q-gram size differs for each IF: for PROT, $q = 2$; for WSJ, $q = 3$; and for DNA, $q = 7$. As can be seen, on the PROT and WSJ sets, the inverted file outperforms the compressed suffix array. For DNA, however, the SSA is a clear winner. This raises the question, under what circumstances is the SSA superior to the IF? From Figure 4 it seems that increasing m, the pattern length, favours the SSA (m increases from PROT (top) to DNA (bottom)). A consequence of increasing m is (typically) a reduction in the number occurrences of the pattern. This is readily seen in Table 1, where m decreases and the average number of occurrences increases.

As the number of occurrences increases, it is possible for the time taken by the inverted file to grow slowly, as the effort invested in intersecting inverted lists for q-grams will more likely lead to matches. The suffix array, however, must trace back from the compressed suffix array to the actual string position, so increasing the number of occurrences will reduce run-times, more so if the sample rate is high. This is shown convincingly in Figure 5, where there is a point on each graph for each pattern that is the target of a search. Lines are drawn in panel (b) to aid clarity and do not represent fitted curves. Both panels clearly show that as *occ* increases, the SSA runtime increases at a greater rate than the IF runtime. Depending on the index size and data set, the minimum number of

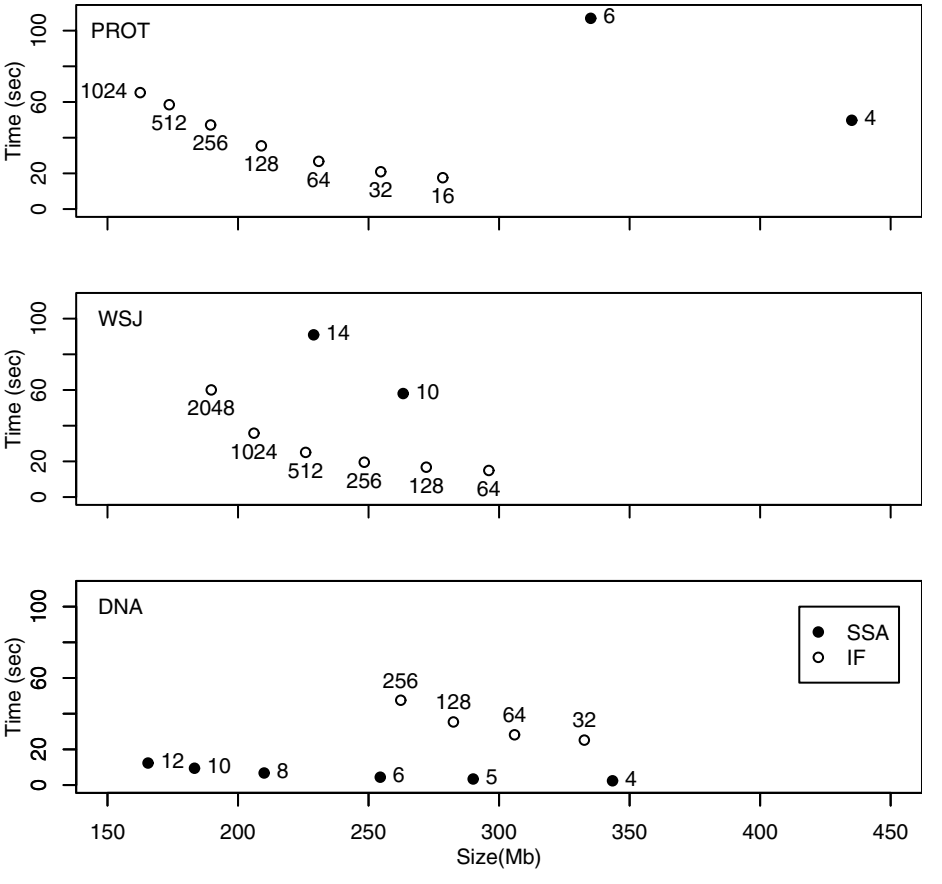


Fig. 4. Resources required for the three data sets. Numbers adjacent to the IF (open) dots indicate the blocksize used, while the numbers adjacent to the SA (filled) dots indicate the sample rate used.

occurrences of a pattern required so that the SSA is slower than the IF ranges from several thousand up to 70,000.

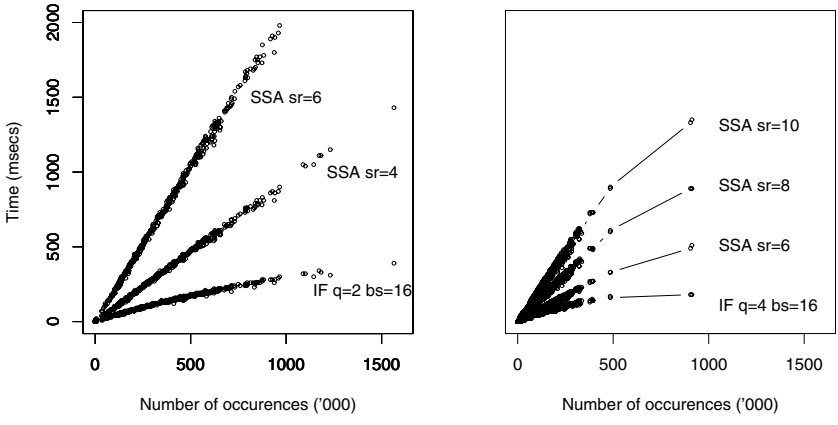
Once the area of the SSA that contains a pattern has been located, the time required to retrieve each pattern location from the SSA is $O(H_0 \log^{1+\varepsilon} n)$. If we assume that the constant of proportionality is c in this bound, then for PROT the time to fetch an occurrence is

$$t_p = 4.21c \log^{1+\varepsilon}(150),$$

and the time to fetch an occurrence from a located region of the DNA SSA is

$$t_d = 2.21c \log^{1+\varepsilon}(134).$$

These two quantities give the slope of the lines for the SSA in Figure 5, and seeing as $t_d/t_p \approx 0.5$ (assuming a small ε), the line for an SSA in panel (a)



(a) PROT $m = 2$ (b) DNA $m = 5$

Fig. 5. Time to search for each (a) two-character pattern in PROT and (b) each five-character pattern in DNA. Each dot represents a single pattern search.

should be twice as steep as the SSA with the same sample rate in panel (b). (The sample rate is buried in the c of t_p and t_d, so SSAs with different sample rates will have different constants of proportionality in the bound.) Examining Figure 5, we can see that this analysis is supported by the data. For the low entropy dataset, DNA in panel (b), the slope of the SSA with a sample rate of 6 ($sr = 6$) is about half that of the SSA with $sr = 6$ on the high entropy data set in panel (a).

The connection between the inverted file performance and entropy is less obvious. In the two graphs the IF curves have the same shape, but one uses $q = 2$ (panel (a)), while the other uses $q = 4$ (panel (b)). By altering the size of the q-grams employed, there is probably some compensation for the differing entropies of the files. This warrants further investigation.

5 Conclusions

We have demonstrated that on several practical, general pattern matching problems, compressed, blocked inverted files built on q-grams outperform compressed suffix arrays. For high zero-order entropy files, such as English text or protein data, when the number of occurrences of the target pattern is lower than about 5,000, the compressed suffix array data structure is superior to the inverted file. For more frequent patterns, inverted files provide faster search times. As zero-order entropy decreases, however, the compressed suffix array handles an increase in the number of occurrences more readily. For example, on DNA data, the suffix array is faster when the pattern occurs less than 70,000 times.

In this study we have addressed the pattern matching problem where it is required that each location of P in T be reported. This formulation of the pattern

matching problem is particularly suited to the inverted file data structure. An alternate version of the problem is the *counting* problem, where the number of occurrences of P in T is reported, but not the locations of each occurrence. Suffix arrays (compressed or otherwise) will perform much faster on the counting variant of the problem than on the location variant presented in this paper. Inverted files, on the other hand, will require the same amount of time for either problem. Inverted files should not be considered as suitable data structure for counting patterns in text.

This study has also enforced the restriction that all indexes must fit in primary memory (RAM). However, for large texts, this is unrealistic and secondary memory must be employed. The virtues of inverted files on disk for word queries are well known, and they have also been successfully employed for genomic applications [32]. Recent work shows implementations of compressed suffix arrays on disk are possible [19] but the question of their efficiency relative to the inverted files remains open.

Finally we observe that we did not compress the text that is used for false match checking on our inverted file system. Even simple byte-packing for the DNA data [33] would reduce the text size to about 34Mb, down from its original 134Mb, a saving of 100Mb. In effect, the points for the IF in the bottom panel of Figure 4 would shift left by 100Mb and, even after compensating for the overhead of the compressed DNA matching, it seems that this change would make the inverted file very competitive with the SSA on the DNA data.

Acknowledgments

This research is supported by by grants from the Australian Research Council (Turpin) and the Natural Sciences and Research Council of Canada (Smyth). Thanks to Libby Fitzpatrick at the Royal Womens Hospital, Melbourne for supplying the DNA queries.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Optimal exact string matching based on suffix arrays. In *SPIRE 2002*, number 2476 in LNCS, pages 31–43. Springer-Verlag, Berlin, 2002.
2. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
3. V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8:151–166, 2005.
4. D. Benson, D.J. Lipman, and J. Ostell. GenBank. *Nucleic Acids Research*, 21(13):2963–2965, 1993.
5. D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, and D.L. Wheeler. Genbank. *Nucleic Acids Research*, 33:D34–D38, 2005.
6. M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

7. M. Cameron, H.E. Williams, and A. Cannane. Improved gapped alignment in blast. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(3):116–129, 2004.
8. Y. Choi and K. Park. Time and space efficient search with suffix arrays. In S. Hong, editor, *Proceedings of AWOCA'04*, pages 230–238, Ballina, Australia, 2004.
9. E. S. De Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.
10. Ensembl. Ensembl Genome Browser, 2006. http://www.ensembl.org.
11. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Found. of Comp. Sci.*, pages 390–398, Redondo Beach, CA, 2000. IEEE Computer Society.
12. P. Ferragina and G. Navarro. Pizza& Chili Corpus – Compressed Indexes and their Testbeds, 2005. http://pizzachili.dcc.uchile.cl.
13. R. Grossi, J. S. Vitter, and A. Gupta. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 636–645, 2004.
14. D. K. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing and Management*, 31(3):271–289, 1995.
15. J. Kärkkäinen. Ziv-Lempel index for q-grams. *Algorithmica*, 21(1):137–154, 1998.
16. S. Kurtz. Reducing the space requirement of suffix trees. *Software, Practice and Experience*, 29(13):1149–1171, 1999.
17. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(2):40–66, 2005.
18. V. Mäkinen and G. Navarro. Compressed full text indexes. Technical Report TR/DCC-2005-7, Department of Computer Science, University of Chile, June 2006.
19. V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching - efficient secondary memory and distributed implementation of compressed suffix arrays. In *Algorithms and Computation: 15th International Symposium, ISAAC 2004*, number 3341 in LNCS, pages 681–692. Springer-Verlag, Berlin, 2004.
20. U. Manber and G. W. Myers. Suffix arrays: a new model for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
21. U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the USENIX Technical Conference*, pages 23–32, Berkeley, CA, 1994. USENIX Association.
22. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
23. E. M. McCreight. A space-economical suffix tree construction algroithm. *Journal of the ACM*, 23(2):262–272, 1976.
24. G. Navarro, E. S. De Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3:49–77, 2000.
25. NCBI. NCBI Blast, 2006. http://www.ncbi.nlm.nih.gov/BLAST/.
26. Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. In *Proceedings of the Prague Stringology Conference*, pages 1–30, Prague, August 2005. Czech Technical University.
27. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation: 11th International Conference, ISAAC 2000*, number 1969 in LNCS, pages 410–421. Springer-Verlag, Berlin, 2000.

28. K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, San Francisco, CA, 2002.
29. J. S. Sim, D. K. Kim, H. Park, and K. Park. Linear-time search in suffix arrays. In M. Miller and K. Park, editors, *Proceedings of AWOCA'03*, pages 139–146, Seoul, Korea, 2003.
30. W. F. Smyth. *Computing Patterns in Strings.* Addison-Wesley-Pearson Education Limited, Essex, England, 2003.
31. P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th annual Symposium on Foundations of Computer Science*, pages 1–11, 1973.
32. H. E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):63–78, 2002.
33. Hugh Williams and Justin Zobel. Compression of nucleotide databases for fast searching. *CABIOS Computer Applications in the Biological Sciences*, 13(5):549–554, October 1997.
34. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images - 2nd Edition.* Morgan Kaufmann Publishing, San Francisco, 1999.
35. J. Zobel, A. Moffat, and R. Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the International Conference on Very Large Data Bases*, pages 290–301, Dublin, Ireland, August 1993.

Efficient Lazy Algorithms for Minimal-Interval Semantics

Paolo Boldi and Sebastiano Vigna

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano

Abstract. Minimal-interval semantics [3] associates with each query over a document a set of intervals, called *witnesses*, that are incomparable with respect to inclusion (i.e., they form an antichain): witnesses define the minimal regions of the document satisfying the query. Minimal-interval semantics makes it easy to define and compute several sophisticated proximity operators, provides snippets for user presentation, and can be used to rank documents: thus, computing efficiently the antichains obtained by operations such as logic conjunction and disjunction is a basic issue. In this paper we provide the first algorithms for computing such operators that are linear in the number of intervals and logarithmic in the number of input antichains. The space used is linear in the number of antichains. Moreover, the algorithms are *lazy* — they do not assume random access to the input antichains. These properties make the usage of our algorithms feasible in large-scale web search engines.

1 Introduction

Search engines are a popular way to retrieve information in the web. However, the classical problem studied by the theory of information retrieval, that of answering a query by returning the set of documents that match the information provided by the user, is complicated by the huge number of documents to be taken into consideration. On the web retrieving many relevant documents is usually not a problem — the documents are simply too many already. Precision, rather than recall (in particular, precision in the first 10−20 results) is the main issue.

A first possibility for extending the user capabilities is *query expansion*, an automatic or semi-automatic mechanism that that tries to enrich a given query, by using for example some semantics extracted from the context, or by asking directly the user what is the intended meaning of his/her query.

A different approach is that of departing from the Boolean model, and provide the user with more powerful (but understandable) operators. In this paper we concentrate on *minimal-interval semantics*, a semantic model that uses *antichains of intervals of natural numbers* to represent the semantics of a query. Each interval is a *witness* of the satisfiability of the query, and defines a region of the document that the query satisfies (words in the document are numbered starting from 0, so regions of text are identified with intervals of integers). For instance, a query formed by the conjunction of two terms is satisfied by the minimal intervals of the document containing both terms: minimality is guaranteed

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 134–149, 2006.

by the fact that in an antichain every pair of elements is incomparable with respect to inclusion.

This approach has been defined and studied in full extent by Clarke, Cormack and Burkowski in their seminal paper [3]. They showed that antichains have a natural lattice structure that can be used to interpret conjunctions and disjunctions in queries. Moreover, it is possible to define several additional operators (proximity, followed-by, and so on) directly on the antichains. The authors have also described families of successful ranking schemes based on the number and length of the intervals involved.

The main feature of minimal-interval semantics is that, by its very definition, an antichain of intervals cannot contain more than n intervals, where n is the number of words in the document. Thus, it is in principle possible to compute all minimal-interval operators in time linear in the document size. This is not true, for instance, if we consider different interval-semantics approaches in which *all* intervals are retained and indexed (e.g., the PAT system [5] or the `sgrep` tool [6]), as the overall number of regions is quadratic in the document size.

In this paper, we attack the problem of providing efficient algorithms for the computation of such operators. As a subproblem, we can compute the proximity of a set of terms, and indeed we are partly inspired by previous work on proximity [11,9]. Our algorithms are linear in the number of input intervals. For conjunction and disjunction, there is also a multiplicative logarithmic factor in the number of input antichains, which however can be shown to be essentially unavoidable in the disjunctive case. The space used by all algorithms is linear in the number of input antichains (in fact, we need to store just one interval per antichain). Moreover, our algorithms are *lazy*, that is, while building their results they do not advance the input lists more than necessary.

Previously, the only attempt at linear lazy algorithms for minimal-interval region algebras we are aware of is the work of Young–Lai and Tompa on *structure selection queries* [13], a special type of expressions built on the primitives "contained-in", "overlaps", and so on, that can be evaluated lazily in linear time. Their motivations are similar to ours — application of region algebras to very large text collections. Similarly, Navarro and Baeza–Yates [8] propose a class of algorithms that using tree-traversals are able to compute efficiently several operations on overlapping regions. Their motivations are efficient implementation of structured query languages that permit such regions. Indeed, some of the techniques used therein (e.g., double stacks) are similar to our double indirect priority queues. Albeit similar in spirit, they do not provide algorithms for any of the operators we consider.

We believe that the existence of (almost) linear lazy algorithms for minimal-interval semantics makes it the natural candidate for advancing web search engines beyond a purely Boolean model: in particular, the possibility of limiting the interval width has a very natural interpretation for the user in terms of proximity, and ordered conjunction has obvious applications (e.g., searching for a verse in a song when some word is missing).

Minimal intervals can also be used together with other standard information-retrieval techniques. For instance, the Indri search engine [10] expands a query into a number of subqueries, many of which are interval-based, and combines the results.

In Section 2 we briefly introduce minimal-interval semantics, referring to the original paper for examples and motivations. The presentation is rather algebraic, and uses standard terms from mathematics and order theory (e.g., "interval" instead of "extent" as in [3]). The resulting structure is essentially identical to that described in the original paper, but our systematic approach makes good use of well-known results from order theory (for instance, we do not need to prove that antichains form a lattice, as they are a well-known representation of the ideal completion of a partial order), making the introduction self-contained. For some mathematical background, see, for instance, Birkhoff's classic [1].

Another advantage of our presentation is that by representing abstractly regions of text as intervals of natural numbers we can easily highlight connections with other areas of computer science: antichains of intervals have been used for role-based access control [4], and for testing distributed computations [7]. The problem of computing operators on antichains has thus an intrinsic interest that goes beyond the problems of information retrieval.

Finally, we present our algorithms. An implementation is available as a part of MG4J (`http://mg4j.dsi.unimi.it/`).

2 Minimal-Interval Semantics

Let us denote with $\mathscr{I}_n$ the set of intervals of $n = \{0, 1, \ldots, n-1\}$ (a subset X of n is an *interval* if $x, y \in X$ and $x < z < y$ then $z \in X$; note that $\varnothing \in \mathscr{I}_n$) ordered by inclusion. By numbering words in the document starting from 0 (see Figure 1), elements of $\mathscr{I}_n$ can be thought of as regions of text.

Given intervals I and J, the interval *spanned* by I and J is the least interval containing I and J (in fact, the supremum in $\mathscr{I}_n$). Nonempty intervals will be denoted by $[\ell \,..\, r]$, where ℓ is the left extreme and r is the right extreme (i.e, the smallest and largest element in the interval). Intervals are ordered by containment: when we want to order them by *reverse* containment instead, we shall write $\mathscr{I}_n^{\mathrm{op}}$ ("op" stands for "opposite").

The idea behind minimal-interval semantics [3] is that every interval in $\mathscr{I}_n$ is a *witness* that a given query is satisfied by a document made of n words. *Smaller witnesses imply a better match, or more information*; in particular, if an interval is a witness any containing interval is a witness. We also expect that *more witnesses imply more information*. Thus, when expressing the semantics of a query, we discard non-minimal intervals, as there are intervals that provide more relevant information. As a result, minimal-interval semantics associates with each query an *antichain*[1] of intervals. For instance, in Figure 1 we see a short passage of text, and the antichain of intervals corresponding to a query. Note that, for instance, the interval $[0 \,..\, 3]$ is not included as it is not minimal.

[1] An *antichain* of a partial order is a subset of elements pairwise incomparable.

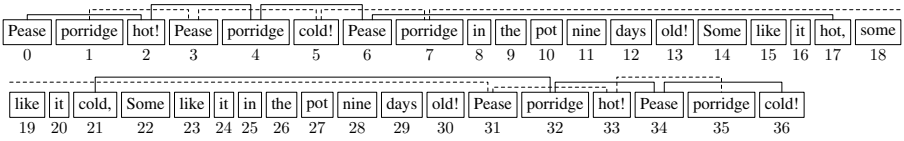


Fig. 1. A sample text; the intervals corresponding to the semantics of the query "(hot OR cold) AND porridge AND pease" are shown. For easier reading, every other interval is dashed.

It is however more convenient to start from an algebraic viewpoint. An *order ideal* X (henceforth called just an *ideal*) is a subset of a partial order that is closed downwards: if $y \leq x$ and $x \in X$, then $y \in X$. The *ideal completion* of an order P is a distributive lattice whose elements are the ideal of P ordered by inclusion. The ideal completion of $\mathscr{I}_n^{\mathrm{op}}$ will be the base of our semantics:

$$\mathscr{E}_n = \left\{ X \subseteq \mathscr{I}_n^{\mathrm{op}} \mid \ X \text{ is an ideal} \right\}.$$

It is known that (at least in the finite case) an ideal over a finite partial order is uniquely represented by the antichain of its maximal elements. Intuitively, the antichain of maximal elements is the "upper border" of the ideal. Because of this bijection, antichains of intervals are endowed with a partial order, and with the algebraic structure of a distributive lattice.

The lattice of antichains thus defined is essentially the classic Clarke–Cormack–Burkowski minimal-interval lattice, with the important difference that since we allow the empty interval, we have a top element that has the empty interval only as a witness. For the purposes of this paper, the difference is immaterial, though.

To make the reader grasp more easily the meaning of $\mathscr{E}_n$, we now describe in an elementary way its order and its lattice operations (note that we are not giving a definition: the operations are simply the reflection on the set of antichains of those of $\mathscr{E}_n$). Given antichains A and B, we have

$$A \leq B \iff \forall I \in A \quad \exists J \in B \quad J \subseteq I.$$

Intuitively, $A \leq B$ if every witness I in A (an interval) can be substituted by a better (or equal) witness J in B, where "better" means that the new witness J is contained in I.

Correspondingly, the $\vee$ of two antichains A and B is given by the union of the intervals in A and B from which non-minimal intervals have been eliminated. Finally, the $\wedge$ of A and B is given by the set of all intervals spanned by a pair of intervals $I \in A$ and $J \in B$, from which non-minimal intervals have been eliminated. It is this very natural algebraic structure that has led to the definition of the Clarke–Cormack–Burkowski lattice.

For instance, if we consider Figure 1 the lists for "porridge" ($\{ 1, 4, 7, 32, 35 \}$), "pease" ($\{ 0, 3, 6, 31, 34 \}$) and "hot" or "cold" ($\{ 2, 5, 17, 21, 33, 36 \}$) give us a large number of spanned intervals, from which we keep the antichain

$$\{\,[0\,..\,2], [1\,..\,3], [2\,..\,4], [3\,..\,5], [4\,..\,6], [5\,..\,7], [6\,..\,17], [7\,..\,31],\\ [21\,..\,32], [31\,..\,33], [32\,..\,34], [33\,..\,35], [34\,..\,36]\,\}.$$

A simple snippet extraction algorithm would compute greedily the first k smallest nonoverlapping intervals of the antichain, which would yield, for $k = 3$, the intervals $[0\,..\,2]$, $[3\,..\,5]$, $[31\,..\,33]$, that is, "Pease porridge hot!", "Pease porridge cold!", and, again, "Pease porridge hot!". A ranking scheme such as those proposed in [2] would use the number and the length of the intervals to assign a score to the document with respect to the query.

Finally, we remark that the intervals in an antichain can be ordered in principle either by left or by right extreme, but these orders can be easily shown to be the same, so the intervals in an antichain are naturally linearly ordered by their extremes.

3 Operators

We shall not give a formal definition of query: the syntax is implied by our choice of operators. As a guide, the reader must consider that the semantics of a query containing a single term is the antichain of singleton intervals corresponding to the positions in which the term appears.

For completeness, we define explicitly the operators AND and OR, which are applied to a list of input antichains $A_0, A_1, \ldots, A_{n-1}$, resulting in the $\wedge$ and $\vee$, respectively, of the antichains $A_0, A_1, \ldots, A_{n-1}$. There are other useful operators that can be defined directly on the antichain representation [3]:

1. BLOCK, given input antichains $A_0, A_1, \ldots, A_{n-1}$, returns the set of intervals of the form $[\ell_0\,..\,r_0] \cup [\ell_1\,..\,r_1] \cup \cdots \cup [\ell_{n-1}\,..\,r_{n-1}]$ for which $[\ell_i\,..\,r_i] \in A_i$ $(0 \le i < n)$ and $r_{i-1} + 1 = l_i$ $(0 < i < n)$.
2. $\text{AND}_{\le}$, given input antichains $A_0, A_1, \ldots, A_{n-1}$, returns the set of minimal intervals among those spanned by a set of intervals $[\ell_i\,..\,r_i] \in A_i$ $(0 \le i < n)$ satisfying $l_{i-1} \le l_i$ $(0 < i < n)$.
3. $\text{AND}_{<}$, given input antichains $A_0, A_1, \ldots, A_{n-1}$, returns the set of minimal intervals among those spanned by a set of intervals $[\ell_i\,..\,r_i] \in A_i$ $(0 \le i < n)$ satisfying $r_{i-1} < l_i$ $(0 < i < n)$.
4. LOWPASS_k, given an input antichain A, returns the set of intervals from A not longer than k.

More informally, given input antichains $A_0, A_1, \ldots, A_{n-1}$, the operator BLOCK builds sequences of consecutive intervals, each of which is taken from a different antichain, in the given order. It can be used, for instance, to implement a phrase operator. The $\text{AND}_{\le}$ and $\text{AND}_{<}$ operators are ordered-AND operators which return intervals containing intervals from all of the A_i, much like the AND operator. However, in the case of $\text{AND}_{\le}$ and $\text{AND}_{<}$ the left extremes of the intervals must be nondecreasing, and in the case of $\text{AND}_{<}$ the intervals must be nonoverlapping. These operators (in particular $\text{AND}_{\le}$) can be used, for instance, to search for terms that must appear in a specified order. Finally,

LOWPASS restricts the result to intervals shorter than a given threshold, and be easily combined with AND or $\text{AND}_{\leq}$ to implement searches for terms that must not be too far apart, and possibly appear in a given order.

Note that the natural lattice operators AND and OR cannot return the empty antichain when all their inputs are nonempty. This is not true of the above operators: for instance, BLOCK might fail to find a sequence of consecutive intervals even if all its inputs are nonempty.

Finally, we remark that all intervals satisfying the definition of the BLOCK operator are minimal. Indeed, assume by contradiction that for two concatenations of minimal intervals we have $[\ell \,..\, r] \subset [\ell' \,..\, r']$ (which implies either $\ell' < \ell$ or $r < r'$). Assume that $\ell' < \ell'$ (the case $r < r'$ is similar), and note that removing the first component interval from both concatenations we still get intervals strictly containing one another. We iterate the process, obtaining two intervals of A_{n-1} (A_0, respectively) strictly containing one another.

4 Lazy Evaluation of Query Operators

Most search engines use inverted files to index their document collections [12]. Usually, inverted indices are to be scanned in a sequential, left-to-right manner. Thus, given a document containing a term t, we assume that it is possible to obtain a *list* L_t containing the positions of t in the document *in increasing order*. Each call to next(L_t) returns a new position, and, when no more positions are available, **null** is returned. We identify position k with the singleton interval $[k \,..\, k]$, so that L_t can be viewed as an antichain of intervals. More generally, the list L_i will return the intervals of the input antichain A_i.

The main point of this paper is that algorithms for computing operators on antichain of intervals should be always *lazy* and *linear in the input intervals*: if an algorithm is lazy, when only a small number of intervals is needed (e.g., for presenting snippets) the computational cost is significantly reduced. Linearity in the input intervals is the best possible result for a lazy algorithm, as input must be read at some point. All algorithms described in this paper satisfy this property, albeit in the case of AND and OR there is also a logarithmic factor in the number of input antichains.

The logarithmic factor in the number of antichains can be easily proved to be unavoidable for the OR operator in a model in which intervals can be handled just by comparing their extremes:

Theorem 1. *Every algorithm to compute* OR *that is only allowed to compare interval extremes requires* $\Omega(n \log n)$ *comparisons.*

Proof. It is possible to sort n distinct integers by computing the OR of n antichains, each containing a single singleton interval containing one of the integers to be sorted. The resulting antichain is exactly the list of sorted integers. By an application of the $\Omega(n \log n)$ lower bound for sorting in this model, we get to the result.

5 Algorithms Based on Double Indirect Queues

The algorithms we provide for AND and OR are inspired by the plane-sweeping technique used in [11] for their proximity algorithm, which is on its own right a variant of the standard sorted-list merge. The algorithms are implemented using a *double indirect priority queue*.

A double indirect priority queue Q is a data structure based on an array (called the *reference array*), which is managed outside the queue itself, and two priority orders that compare items from the reference array: these two orders are called *primary* and *secondary*. At each time, the queue contains a set of indices into the reference array (initially, a specified set, possibly empty). An array index x can be added to the queue calling the function enqueue(Q,x).

The *index* of the least item in the reference array with respect to the primary (secondary) priority order can be accessed by invoking the function topIndex(Q) (secondaryTopIndex(Q), resp.). The index of the least item with respect to the primary priority order is also returned by dequeue(Q), which also removes the index from Q. Analogously, top(Q) (secondaryTop(Q), resp.) return the least *item* in the reference array with respect to the primary (secondary) priority order.

The data structure assumes that the only item of the reference array that might change its value is the top item. Such a change must be communicated immediately to the queue by calling the function change(Q). Table 1 summarises the operations available on a double indirect priority queue.

A double indirect priority queue can be easily and efficiently implemented using two priority queues: a *primary* semi-indirect queue and a *secondary* indirect queue[2]. Note that we need the secondary queue to be fully indirect, as the primary queue must be able to adapt just to changes of its top item, but the secondary queue must be able to adapt to changes of *any* item (as it must be able to reflect changes in the top of the *primary* queue).

A trivial array-based implementation requires linear space and has constant cost for all operations modifying the queue, whereas retrieving the (secondary) top requires $O(n)$ time. A better implementation uses a priority queue (e.g., based on a heap) with linear space and logarithmic time complexity for all operations modifying the queue, and constant-time access to the (secondary) top. Sophisticated heaps with linear costs for several operations do not modify significantly the overall behaviour, as each time the queue is advanced the interval corresponding to the top index becomes greater: there are data structures that make it possible to *decrease* in constant time the top, but not in *increase it* (or we could sort in linear time by comparison).

All algorithms based on double priority queues have complexity $O(m \log n)$ if the input is formed by n antichains containing m intervals overall. This is immediate, as all loops contain exactly one queue advancement.

[2] A *semi-indirect* queue has a change operation that allows to restore the correct state after a change in the value associated to the top item. An *indirect* queue has a change operation that restores the correct state after a change in the value associated to any index.

Table 1. The operations available for a double indirect priority queue

enqueue(Q,x)	insert item with index x in the queue
topIndex(Q)	returns the index of the top item w.r.t. the primary order
secondaryTopIndex(Q)	returns the index of the top item w.r.t. the secondary order
top(Q)	returns the top item w.r.t. the primary order
secondaryTop(Q)	returns the top item w.r.t. the secondary order
dequeue(Q)	returns the top item w.r.t. the primary order, and deletes it from the queue
change(Q)	signals that the top item has changed
size(Q)	returns the number of indices currently in the queue

5.1 Basic Comparators

To describe our algorithms we will use two main priority orders. The first one, denoted by $\preceq$, is defined by

$$[\ell \,..\, r] \preceq [\ell' \,..\, r'] \iff \ell < \ell' \text{ or } \ell = \ell' \text{ and } r \geq r'.$$

In other words, $[\ell \,..\, r] \preceq [\ell' \,..\, r']$ if $[\ell \,..\, r]$ starts before or prolongs $[\ell' \,..\, r']$. Note in particular that (somewhat counterintuitively) $[\ell \,..\, r] \preceq [\ell \,..\, r']$ iff $r \geq r'$. This order will always be used as a primary order in a queue.

The second order, denoted by $\trianglelefteq$, is easier: it compares the right extremes according to their natural order:

$$[\ell \,..\, r] \trianglelefteq [\ell' \,..\, r'] \iff r \leq r'.$$

We remark that in a queue using $\preceq$ as primary order the left extreme of the top interval is nondecreasing.

The algorithms for AND/OR use a double indirect priority queue with primary priority order $\preceq$. The reference array underlying the queue contains one interval per input antichain, which we assume without loss of generality nonempty (in the case of AND, an empty list implies an empty result, and in the case of OR empty lists can be simply dropped). In the initialisation phase, the reference array is filled with the first interval from each antichain, and the queue contains all indices.

To simplify the description, we define a function advance(Q) that stores temporarily the current top interval, updates with the next interval the list associated with the top index, notifies the queue of the change, and finally returns the stored interval. If the update cannot be performed because the list is empty, the top index is dequeued. The function is described in pseudocode in Algorithm 1, where we assume that $[\ell_i \,..\, r_i]$ is the interval in the reference array for list i.

Algorithm 1. The advance function.

```
0  function advance(Q) begin
1      i ← topIndex(Q);
2      c ← [ℓ_i .. r_i];
3      if the input list i is not empty then
4          [ℓ_i .. r_i] ← next(L_i);
5          change(Q)
6      else
7          dequeue(Q)
8      end;
9      return c
10 end;
```

5.2 The OR Operator

We start with the simplest nontrivial operator. To compute the interval antichain corresponding to the OR of the antichains $A_0, A_1, \ldots, A_{n-1}$ we create a double indirect priority queue Q with primary priority order $\preceq$ and secondary priority order $\trianglelefteq$. As a consequence, the right extreme of the secondary top interval is nondecreasing, because every time we advance the queue we either eliminate an interval or substitute it with one that has a larger right extreme.

When we want to compute the next interval, we advance Q and store the returned interval $[\ell \,..\, r]$ (which is, essentially, the leftmost largest remaining interval) as a candidate. We repeat the process until Q is empty or $[\ell \,..\, r]$ does not contain the secondary-top interval. The algorithm is described in pseudocode in Algorithm 2.

Theorem 2. *The algorithm for* OR *is correct.*

Proof. First of all, note that all intervals in $A_0, A_1, \ldots, A_{n-1}$ are assigned to c at some point, and if c contains a minimal interval, we certainly exit the loop (more precisely, we exit when c is the last instance of a given minimal interval to appear in the queue top). Thus, we only have to prove is that only minimal intervals are returned.

Assume that at the start of the while loop $[\ell \,..\, r]$ is the primary-top interval, and, after advancing the queue, let $[\ell' \,..\, r']$ be the primary-top interval and $[\ell'' \,..\, r'']$ the secondary-top interval. If $[\ell \,..\, r]$ is not minimal, then it must contain some smaller interval, say $[\bar{\ell} \,..\, \bar{r}] \subset [\ell \,..\, r]$ coming from the i-th list. We can assume without loss of generality that $[\bar{\ell} \,..\, \bar{r}]$ is actually one of the intervals currently in Q, as if this is not true the interval in the reference array at index i has smaller extremes but left extreme larger than or equal to l, so it is *a fortiori* included in $[\ell \,..\, r]$ (note that this fact strictly depends on the definition of $\preceq$).

Since $[\bar{\ell} \,..\, \bar{r}]$ is in the queue, we have $[\ell'' \,..\, r''] \trianglelefteq [\bar{\ell} \,..\, \bar{r}] \trianglelefteq [\ell \,..\, r]$ hence $r'' \leq r$, so $[\ell \,..\, r] \supseteq [\ell'' \,..\, r'']$ (by monotonicity of the top-interval left extreme $\ell \leq \ell' \leq \ell''$) and we repeat the loop. We conclude that only minimal intervals are returned.

To prove that all returned intervals are unique, we just have to note that if several copies of the interval I are present in the input antichains, then as soon as the first copy of I becomes the top, all other copies of I are in the reference array (or there would be intervals in the reference array with left extreme smaller than I). Thus, the while loop will be repeated until all copies are discarded. At that point, I will be returned only if it is minimal.

Algorithm 2. The algorithm for the OR operator.

```
0  function next begin
1     if Q is empty return null;
2     do
3        c ← advance(Q)
4     while ¬ (Q is empty) and secondaryTop(Q) ⊆ c;
5     return c
6  end;
```

5.3 The AND Operator

Then AND operator is much more subtle. The primary comparator of Q is $\preceq$, whereas the secondary comparator is $\trianglerighteq$ (note the inversion). At any time, the interval *spanned* by Q is the interval defined by the left extreme of the primary-top interval and the right extreme of the secondary-top interval; it will be denoted by span(Q). Clearly, it is the minimum interval containing all intervals currently in the queue. Note that the right extreme of the secondary top cannot decrease while Q is *full*, that is, the size of Q is n.

When we want to compute the next interval, we store the interval $[\ell\,..\,r]$ currently spanned by Q as a candidate and advance Q. If the new interval spanned by Q is included in $[\ell\,..\,r]$ we repeat the operation, updating the candidate. Then, before returning $[\ell\,..\,r]$ we advance Q until the spanned interval does not contain $[\ell\,..\,r]$. If at any time Q is no longer full, we just return the candidate. The algorithm is described in pseudocode in Algorithm 3.

Theorem 3. *The algorithm for* AND *is correct.*

Proof. We say that a queue configuration is *complete* if it contains all copies of the primary top interval from all lists that contain it. Now observe that *every complete configuration of a double indirect priority queue is entirely defined by its primary top interval.* More precisely, if the top is an interval I from list i, then for every other list j the corresponding interval J in the queue is the minimum interval in A_j larger than or equal to I (following $\preceq$). Indeed, suppose by contradiction that there is another interval K from A_j satisfying $I \preceq K \prec J$.

Algorithm 3. The algorithm for the AND operator.

```
0   function next begin
1     if ¬ (Q is full) then return null;
2     do
3       c ← span(Q);
4       advance(Q)
5     while Q is full and span(Q) ⊆ c;
6
7     while Q is full and c ⊆ span(Q) do
8       advance(Q)
9     end;
10    return c
11  end;
```

Then, at some point K must have entered the queue, and must have been dequeued when the top was some interval $I' \preceq I$, so we get $K \preceq I' \preceq I \preceq K$, which yields $K = I$: a contradiction, as we assumed the state of the queue to be complete.

We now show that for every minimal interval $[\ell\,..\,r]$ in the AND of A_0, A_1, ..., A_{n-1} there is a complete state of Q spanning $[\ell\,..\,r]$. Consider for each i the set C_i of intervals of A_i contained in $[\ell\,..\,r]$. At least one of these sets must contain a (necessarily unique) *right delimiter*, that is, an interval of the form $[\ell'\,..\,r]$. Moreover, at least one of the sets containing a delimiter must be a singleton. Indeed, if every C_i containing a right delimiter would also contain some other interval, the right extreme of that interval would clearly be smaller than r: removing all right delimiters from the C_i's, we would span a strictly smaller new interval showing that $[\ell\,..\,r]$ was not minimal. We conclude that at least one C_i, say $C_{\bar{\imath}}$, is a singleton containing a right delimiter.

Consider now for each C_i the leftmost (in the sense of $\preceq$) interval I_i. The resulting set of intervals defines a complete configuration of Q: if i is such that $I_i = [\ell\,..\,r']$ and if $I_i \in A_j$ necessarily $I_i = I_j$, because A_j cannot contain two intervals with the same left extreme. The set of intervals also spans $[\ell\,..\,r]$ (because the right extreme of $I_{\bar{\imath}}$ is r, and the left extreme of the $\preceq$-least interval I_i is ℓ). We conclude that all minimal intervals are sooner or later spanned by Q.

However, no minimal interval can be spanned during the second while loop. All intervals spanned in that loop contain the candidate interval, which makes them nonminimal (independently of the minimality of the candidate) or copies of the minimal candidate we are going to return. Finally, if an interval is spanned in the first while loop and we do not get out of the loop, the next candidate interval will be smaller or equal. We conclude that sooner or later all minimal intervals cause an interruption of the first while loop, and are thus returned.

We are left to prove that if an interval is returned, it is necessary minimal. We prove at the same time the following invariant: no interval containing a previously returned interval will be ever spanned by Q (this is trivially true at the first call). Assume now that the interval $[\ell\,..\,r]$ spanned by Q at the start of the first while loop is not minimal, so $[\bar{\ell}\,..\,\bar{r}] \subset [\ell\,..\,r]$, for some minimal interval $[\bar{\ell}\,..\,\bar{r}]$ that will be necessarily spanned later (because of the invariant, as we already proved that all minimal intervals are returned). Then, letting $[\ell''\,..\,r'']$ be the secondary-top interval after we advanced Q, we have $r'' \leq \bar{r}$ by monotonicity of the secondary-top right extreme. On the other hand, always by monotonicity of the secondary-top right extreme, $r \leq r''$, and since $[\bar{\ell}\,..\,\bar{r}] \subset [\ell\,..\,r]$, $\bar{r} \leq r$. We conclude $r = r'' = \bar{r}$. By monotonicity of the primary-top left extreme, the interval spanned by Q is contained in $[\ell\,..\,r]$, so we will not exit the first while loop.

We must prove that the invariant is true at the end of the call. However, this is trivial, as the second while loop advances Q, eliminating all intervals that could contain c. By monotonicity of the primary-top interval left extreme, after the second while loop the left extreme of the interval spanned by Q will be larger than that of c: thus, no following intervals spanned by Q will be able to contain c.

Finally, we remark that the invariant yields immediately that all returned intervals are unique.

6 Greedy Algorithms

The algorithms for BLOCK, $\text{AND}_{\leq}$ and $\text{AND}_{<}$ are much simpler: they are just a greedy enumeration procedure with backtracking (in the latter two cases, borrowing also some ideas from queue-based algorithms). In some case they are part of the folklore, at least when applied to list of term positions. Nonetheless, a thorough correctness proof for the case of interval antichains is not completely obvious. All algorithms have trivially complexity $O(m)$, where m is the number of intervals in the input antichains, as all loop bodies advance at least one of the input lists.

6.1 The BLOCK Operator

We keep track of a current interval for all lists L_0, L_1, ..., L_{n-1}; initially, these intervals are set to $[-1\,..\,-1]$. When we want to compute the next interval, we update the interval associated to the first list. Then, we try to fix index i (initially, $i = 1$). To do so, we advance the list L_i until the returned interval has left extreme larger than the right extreme of the current interval for L_{i-1}. If we go too far, we just advance the first list, reset i to 1 and restart the process, otherwise we increment i. When we find an interval for L_{n-1} we return the interval spanned by all current intervals. The algorithm is described in pseudocode in Algorithm 4.

Theorem 4. *The algorithm for* BLOCK *is correct.*

Algorithm 4. The algorithm for the BLOCK operator.

```
0   function next begin
1     if L_0 is empty then return null;
2     [ℓ_0 .. r_0] ← next(L_0);
3     i ← 1;
4     while i < n do
5       while ¬ (L_i is empty) and ℓ_i ≤ r_{i−1} do
6         [ℓ_i .. r_i] ← next(L_i)
7       end;
8       if ℓ_i ≤ r_{i−1} then return null
9       else if ℓ_i = r_{i−1} + 1 then i ← i + 1
10      else begin
11        if L_0 is empty then return null;
12        [ℓ_0 .. r_0] ← next(L_0);
13        i ← 1
14      end
15    end;
16    return [ℓ_0 .. r_{n−1}]
17  end;
```

Algorithm 5. The algorithm for the $\mathrm{AND}_{\leq}$ / $\mathrm{AND}_{<}$ operator. For the $\mathrm{AND}_{<}$ operator, the test $\ell_i < \ell_{i-1}$ must be substituted with $\ell_i \leq r_{i-1}$.

```
0   function next begin
1     if some L_i is empty then return null;
2     do
3       c ← [min_{j∈n} ℓ_j .. max_{j∈n} r_j];
4       if L_0 is empty then return c;
5       [ℓ_0 .. r_0] ← next(L_0);
6       i ← 1;
7       while i < n do
8         while ¬ (L_i is empty) and ℓ_i < ℓ_{i−1} do
9           [ℓ_i .. r_i] ← next(L_i)
10        end;
11        if ℓ_i < ℓ_{i−1} then return null;
12        i ← i + 1;
13      end
14    while [min_{j∈n} ℓ_j .. max_{j∈n} r_j] ⊆ c;
15    return c
16  end;
```

Proof. At the start of an iteration of the external while loop with a certain index i we clearly have $r_k + 1 = \ell_{k+1}$ for $k = 0, 1, \ldots, i-2$. Thus, if we complete the execution we certainly return a correct interval.

To complete the proof, we start by proving the following invariant property: at the start of the external while loop, for all $0 < j < n$ there are no intervals in A_j with left extreme in $[r_{j-1} + 1 \,..\, \ell_j - 1]$. In other words, the j-th current interval $[\ell_j \,..\, r_j]$ has either left extreme smaller than or equal to r_{j-1}, or it is the first interval in A_j whose left extreme is larger than r_{j-1}. The property is trivially true at the beginning, and advancing $[\ell_0 \,..\, r_0]$ cannot change this fact. We are left to prove that the execution of the internal while loop cannot either.

During the execution of the internal loop, only $[\ell_i \,..\, r_i]$ can change. This affects the invariant because it modifies the intervals $[r_{i-1}+1 \,..\, \ell_i-1]$ and $[r_i+1 \,..\, \ell_{i+1}-1]$, but in the second case the interval is made smaller, so the invariant is *a fortiori* true. In the first case, at the beginning of the execution of the internal while loop either $r_{i-1} + 1 \leq \ell_i - 1$, that is, $r_{i-1} < \ell_i$, so the loop is not executed at all and the invariant cannot change, or $r_{i-1} + 1 > \ell_i - 1$, which means that the interval $[r_{i-1} + 1 \,..\, \ell_i - 1]$ is empty, and the loop will advance $[\ell_i \,..\, r_i]$ up to the first interval in A_i with a left extreme larger than or r_{i-1}, making again the invariant true.

Suppose now that there are $[\bar{\ell}_0 \,..\, \bar{r}_0]$, $[\bar{\ell}_1 \,..\, \bar{r}_1]$, $\ldots$, $[\bar{\ell}_k \,..\, \bar{r}_k]$ satisfying $r_i + 1 = \ell_{i+1}$ for some $k > 0$ and $0 \leq i < k$. We prove by induction on k that at some point during the execution of the algorithm we will be at the start of the external while loop with $i = k$ and $[\ell_j \,..\, r_j] = [\bar{\ell}_j \,..\, \bar{r}_j]$ for $j = 0, 1, \ldots, k$. The thesis is trivially true for $k = 0$. Assume the thesis for $k - 1$, so we are at the start of the external while loop with $i = k - 1$ and $l_j = \bar{l}_j$, $r_j = \bar{r}_j$ for $j = 0, 1, \ldots k-1$. Because of the invariant, either $[l_k \,..\, r_k] = [\bar{l}_k \,..\, \bar{r}_k]$ or $[l_k \,..\, r_k]$ will be advanced by the execution of the internal while loop up to $[\bar{l}_k \,..\, \bar{r}_k]$. Thus, at the end of the external while loop the thesis will be true for k. We conclude that all concatenations of intervals from A_0, A_1, $\ldots$, A_{n-1} are returned.

We note that all intervals returned are unique (minimal has been already discussed in Section 3), as $[l_0 \,..\, r_0]$ is advanced at each call, so a duplicate returned interval would imply the existence of two comparable intervals in A_0.

6.2 The AND$_\leq$ and AND$_<$ Operators

The algorithms for computing these operators are a medley of the algorithms for AND and for BLOCK. As in the case of AND, we must check that future intervals are not smaller then our current candidate. As in the case of BLOCK, there is no queue and the lists $L_0, L_1, \ldots, L_{n-1}$ are advanced greedily. Again, we keep track of a current interval $[\ell_i \,..\, r_i]$ for all lists $L_0, L_1, \ldots, L_{n-1}$; initially, these intervals are $[-1 \,..\, -1]$. At any time, the *spanned interval* is $[\min_{j \in n} \ell_j \,..\, \max_{j \in n} r_j]$.

When we want to return a new interval, we store the interval currently spanned. Then, we update the interval associated to the first list and try to fix index i (initially, $i = 1$). To do so, in the case of AND$_\leq$ we advance L_i until the returned interval has left extreme larger than or equal to the left extreme of the current interval; in the case of AND$_<$ we advance L_i until the returned

interval has left extreme larger than the right extreme of the current interval. If we can find a suitable interval, we increment i and continue. When we find an interval for A_{n-1} we check whether it is contained in the candidate, in which case we choose it as a new candidate, and restart the process, otherwise we return the candidate. The algorithm is described in pseudocode in Algorithm 5.

Theorem 5. *The algorithm for* $\text{AND}_{\leq}/\text{AND}_{<}$ *is correct.*

Proof. We discuss the correctness of the algorithm for $\text{AND}_{\leq}$; the case of $\text{AND}_{<}$ is completely analogous.

The first part of the proof is very similar to that for the BLOCK operator. Similarly to that case, one proves that the following invariant property is true at the start of the internal while loop: for all $0 < j < n$ there are no intervals in A_j with left extreme in $[\ell_{j-1} + 1 \,..\, \ell_j - 1]$. In other words, the j-th current interval $[\ell_j \,..\, r_j]$ has either left extreme smaller than to ℓ_{j-1}, or it is the first interval in A_{j+1} whose left extreme is larger than or equal to ℓ_{j-1}. Let us say that a sequence $[\bar{\ell}_i \,..\, \bar{r}_i]$, $i = 0, 1, \ldots, n$ of intervals, one from each list, with nondecreasing left extremes is *leftmost* if for all $0 < j < n$ there are no intervals in A_j with left extreme in $[\bar{\ell}_{j-1} + 1 \,..\, \bar{\ell}_j - 1]$. Then, it is immediate to show that all sequences of leftmost intervals appear at some point at the start of the external loop.

We now note that every minimal interval $[\ell \,..\, r]$ spanned by minimal intervals from the A_i's has a unique leftmost representation. To obtain it from a generic set of interval, substitute iteratively the interval for list $i > 0$ with the interval with smallest left extreme satisfying $\ell_i \geq \ell_{i-1}$. Note that the interval for A_{n-1} cannot change, for $[\ell \,..\, r]$ was assumed to be minimal, so the resulting set of intervals still spans $[l \,..\, r]$. We conclude that sooner or later all minimal intervals of the result are spanned, and thus returned.

We are left to prove that only minimal intervals are returned. As in the proof for the AND operator, we prove at the same time the following invariant: no interval ever spanned in the future will contain a previously returned interval (the invariant is trivially true at the start). Suppose the current candidate $[\ell \,..\, r]$ is not minimal. This means that there is an interval $[\bar{\ell} \,..\, \bar{r}] \subset [\ell \,..\, r]$ that will be spanned later (because of the invariant). By monotonicity of the extremes of spanned intervals, this implies that after advancing the interval set the new spanned interval must be contained in $[\ell \,..\, r]$, so we will not get out of the external while loop.

We must show that the invariant holds at the end of a call. But if the candidate does not contain the currently spanned interval, this means that both extremes are larger than those of the candidate (the left extreme is increased each time the external while loop is executed, and the right extreme must be larger then that of the candidate, or the loop would repeat). We conclude that no spanned intervals will ever contain the candidate.

Finally, as in the proof of Theorem 3 we remark that the invariant yields immediately that all returned intervals are unique.

7 Conclusions

We have provided efficient algorithms for the computation of several operators based on minimal-interval semantics. In particular, the algorithm for OR has been proved to be optimal in a comparison-based model. Moreover, the algorithms are lazy and use space linear in the number of input antichains. This compares favourably with the previously known algorithms [3], which in particular required an eager computation of all component antichains (albeit it should be noted that the two bounds, $O(ns \log m)$ and $O(m \log n)$, are in general incomparable).

The algorithms for AND and OR can be slightly modified so to be even lazier (essentially, once the right candidate to be returned is found, one can delay the loop that advances the input lists), but the descriptions become clumsier, so we delay them to the full version of this paper.

An interesting open problem is that of providing a matching lower bound for the AND operator, at least for a comparison-based computational model.

References

1. Garrett Birkhoff. *Lattice Theory*, volume XXV of *AMS Colloquium Publications*. American Mathematical Society, third (new) edition, 1970.
2. Charles L. A. Clarke and Gordon V. Cormack. Shortest-substring retrieval and ranking. *ACM Trans. Inf. Syst*, 18(1):44–78, 2000.
3. Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *Comput. J.*, 38(1):43–56, 1995.
4. J. Crampton and G. Loizou. The completion of a poset in a lattice of antichains. *International Mathematical Journal*, 1(3):223–238, 2001.
5. G. H. Gonnet. PAT 3.1: An efficient text searching system. User's manual. Technical report, Center for the New Oxford English Dictionary. University of Waterloo, Waterloo, Canada, 1987.
6. Jani Jaakkola and Pekka Kilpeläinen. Nested text-region algebra. Technical Report C-1999-2, Department of Computer Science, University of Helsinki, 1999.
7. Guy-Vincent Jourdan, Jean-Xavier Rampon, and Claude Jard. Computing on-line the lattice of maximal antichains of posets. *Order*, 11(3):197–210, 1994.
8. Gonzalo Navarro and Ricardo Baeza-Yates. A class of linear algorithms to process sets of segments. In *Proc. CLEI'96*, volume 2, pages 671–682, 1996.
9. Jürg Nievergelt and Franco P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Comm. ACM*, 25(10):739–747, 1982.
10. The Lemur Project. Indri. `http://www.lemurproject.org/indri/`.
11. Kunihiko Sadakane and Hiroshi Imai. Fast algorithms for k-word proximity search. *IEICE Trans. Fundamentals*, E84-A(9), September 2001.
12. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.
13. Matthew Young-Lai and Frank Wm. Tompa. One-pass evaluation of region algebra expressions. *Inf. Syst.*, 28(3):159–168, 2003.

Output-Sensitive Autocompletion Search

Holger Bast[1], Christian W. Mortensen[2], and Ingmar Weber[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{bast, iweber}@mpi-inf.mpg.de
[2] IT University of Copenhagen, Denmark
cworm@itu.dk

Abstract. We consider the following autocompletion search scenario: imagine a user of a search engine typing a query; then with every keystroke display those completions of the last query word that would lead to the best hits, and also display the best such hits. The following problem is at the core of this feature: for a fixed document collection, given a set D of documents, and an alphabetical range W of words, compute the set of all word-in-document pairs (w, d) from the collection such that $w \in W$ and $d \in D$. We present a new data structure with the help of which such autocompletion queries can be processed, on the average, in time linear in the input plus output size, independent of the size of the underlying document collection. At the same time, our data structure uses no more space than an inverted index. Actual query processing times on a large test collection correlate almost perfectly with our theoretical bound.

1 Introduction

Autocompletion, in its most basic form, is the following mechanism: the user types the first few letters of some word, and either by pressing a dedicated key or automatically after each key stroke a procedure is invoked that displays all relevant words that are continuations of the typed sequence. The most prominent example of this feature is the tab-completion mechanism in a Unix shell. In the recently launched Google Suggest service frequent queries are completed. Algorithmically, this basic form of autocompletion merely requires two simple string searches to find the endpoints of the range of corresponding words.

1.1 Problem Definition

The problem we consider in this paper is derived from a more sophisticated form of autocompletion, which takes into account the *context* in which the to-be-completed word has been typed. Here, we would like an (instant) display of only those completions of the last query word which lead to hits, as well as a display of such hits. For example, if the user has typed `search autoc`, context-aware completions might be `autocomplete` and `autocompletion`, but not `autocratic`. The following definition formalizes the core problem in providing such a feature.

Definition 1. *An* autocompletion query *is a pair (D, W), where W is a range of words (all possible completions of the last word which the user has started typing), and D is a set of documents (the hits for the preceding part of the query). To process the query means to compute the set of all word-in-document pairs (w, d) with $w \in W$ and $d \in D$.*

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 150–162, 2006.

Given an algorithm for solving autocompletion queries according to this definition, we obtain the context-sensitive autocompletion feature as follows:

For the example query `search autoc`, W would be all words from the vocabulary starting with `autoc`, and D would be the set of all hits for the query `search`. The output would be all word-in-document pairs (w, d), where w starts with `autoc` and d contains w as well as a word starting with `search`. [1]

Now if the user continues with the last query word, e.g., `search autoco`, then we can just filter the sequence of word-in-document pairs from the previous queries, keeping only those pairs (w', d'), where w' starts with `autoc`. If, on the other hand, she starts a new query word, e.g., `search autoc pub`, then we have another autocompletion query according to Definition 1, where now W is the set of all words from the vocabulary starting with `pub`, and D is the set of all hits for `search autoc`. For the very first query word, D is the set of all documents.

In practice, we are actually interested in the *best* hits and completions for a query. This can be achieved by the following standard approach. Assume we have precomputed scores for each word-in-document pair. Given a sequence of pairs (w, d) according to Definition 1, we can then easily compute for each word w' occurring in that sequence an aggregate of the scores of all pairs (w', d) from that sequence, as well as for each document d' an aggregate of the scores of all pairs (w, d'). The precomputation of scores for word-in-document pairs such that these aggregations reflect user-perceived relevance to the given query is a much-researched area in information retrieval [1], and beyond the scope of this paper. It is for these reasons that the ranking issue is factored out of Definition 1.

To answer a series of autocompletion queries, we can obtain the new set of candidate documents D from the sequence of matching word-in-document pairs for the last query by sorting the matching (w, d) pairs. This sort takes time $O((\sum_{w \in W} |D \cap D_w|) \log(\sum_{w \in W} |D \cap D_w|))$ and would in practice be done together with the ranking of the completions and documents. The time for this sort is also included in the running times of our experiments in Section 6, but is dominated by the work to find all matching word-in-document pairs.

1.2 Main Result

Theorem 1. *Given a collection with $n \geq 16$ documents, m distinct words, and $N \geq 32m$ word-in-document pairs [2], there is a data structure* AUTOTREE *with the following properties:*

(a) AUTOTREE *can be constructed in $O(N)$ time.*
(b) AUTOTREE *uses at most $N\lceil \log_2 n \rceil$ bits of space (which is the space used by an ordinary uncompressed inverted index)*[3].

[1] We always assume an implicit prefix search, that is, we are actually interested in hits for all words *starting* with `search`, which is usually what one wants in practice. Whole-word-only matching can be enforced by introducing a special end of word symbol `$`.

[2] The conditions on n and N are technicalities and are satified for any realistic document collection.

[3] Strictly speaking, an uncompressed inverted index needs even more space, to store the list lengths.

(c) AUTOTREE *can process an autocompletion query* (D, W) *in time*

$$O\left((\alpha+\beta)|D| + \sum\nolimits_{w \in W} |D \cap D_w|\right),$$

where D_w *is the set of documents containing word* w*. Here* $\alpha = N|W|/(mn)$*, which is bounded above by* 1*, unless the word range is very large (e.g., when completing a single letter). If we assume that the words in a document with* L *words are a random size-*L *subset of all words,* β *is at most* 2 *in expectation. In our experiments,* β *is indeed around* 2 *on the average and about* 4 *in the (rare) worst case. Our analysis implies a general worst-case bound of* $\log(mn/N)$.

Note that for constant α and β the running time is asymptotically optimal, as it takes $\Omega(|D|)$ time to inspect all of D and it takes $\Omega(\sum_{w \in W} |D \cap D_w|)$ time to output the result.

We implemented AUTOTREE, and in Section 6 show that its processing time correlates almost perfectly with the bound from Theorem 1(c) above. In that Section, we also compare it to an inverted index, its presumably closest competitor (see Section 1.4), which AUTOTREE outperforms by a factor of 10 in worst-case processing time (which is key for an interactive feature), and by a factor of 4 in average-case processing time.

1.3 Related Work

To the best of our knowledge, the autocompletion problem, as we have defined it above, has not been explicitly studied in the literature. The problem is derived from a search engine, which we have devised and implemented, and which is described in [2]; for a live demo, see `http://search.mpi-inf.mpg.de/wikipedia`. The emphasis in [2] is on usability (of the autocompletion feature) and on compressibility (of the data), and not on designing an output-sensitive algorithm. The data structures and algorithms in [2] are completely different from those presented in this article.

The most straightforward way to process an autocompletion query (D, W) would be to explicitly search each document from D for occurrences of a word from W. However, this would give us a non-constant query processing time per element of D, completely independent of the respective $|W|$ or output size $\sum_{w \in W} |D \cap D_w|$. For these reasons, we do not consider this approach further in this paper. Instead, our baseline in this paper is based on an inverted index, the data structure underlying most (if not all) large-scale commercial search engines [1]; see Section 1.4.

Definition 1 looks reminiscent of multi-dimensional search problems, where the collections consists of tuples (of some fixed dimensionality), and queries are asking for all tuples contained in a tuple of given ranges [3,4,5,6]. Provided that we are willing to limit the number of query words, such data structures could indeed be used to process our autocompletion queries. If we want fast processing times, however, any of the known data structures uses space on the order of N^{1+d}, where N is the number of word-in-document pairs in the collection, and d grows (fast) with the dimensionality. In the description of our data structures we will point out some interesting analogies to the geometric range-search data structures from [7] and [8].

The large body of work on string searching concerned with data structures such as PAT/suffix tree/arrays [9,10] is not directly applicable to our problem. Instead, it can be seen as orthogonal to the problem we are discussing here. Namely, in the context of our autocompletion problem these data structures would serve to get from mere prefix search to full substring search. For example, our Theorem 1 could be enhanced to full substring search by first building a suffix data structure like that of [10], and then building our data structure on top of the sorted list of all suffixes (instead of the list of the distinct words).

There is a large body of more applied work on algorithms and mechanisms for *predicting* user input, for example, for typing messages with a mobile phone, for users with disabilities concerning typing, or for the composition of standard letters [11,12,13,14]. In [15], contextual information has been used to select promising extensions for a query; the emphasis of that paper is on the quality of the extensions, while our emphasis here is on efficiency. An interesting, somewhat related phrase-browsing feature has been presented in [16,17]; in that work, emphasis was on the identification of frequent phrases in a collection.

1.4 The BASIC Scheme and Outline of the Rest of the Paper

The following BASIC scheme is our baseline in this paper. It is based on the *inverted index* [1], for which we simply precompute for each word from the collection the list of documents containing that word. For an efficient query processing, these lists are typically sorted, and we assume a sorting by document number.

Having precomputed these lists, BASIC processes an autocompletion query (D, W) very simply as follows: For each word $w \in W$, fetch the list D_w of documents that contain w, compute the intersection $D \cap D_w$, and append it to the output.

Lemma 1. BASIC *uses time at least* $\Omega(\sum_{w \in W} \min\{|D|, |D_w|\})$ *to process an autocompletion query* (D, W). *The inverted lists can be stored using a total of at most* $N \cdot \lceil \log_2 n \rceil$ *bits, where* n *is the total number of documents, and* N *is the total number of word-in-document pairs in the collection.*

Lemma 1, whose proof can be found in [18], points out the inherent problem of BASIC: its query processing time depends on the size of both $|D|$ *and* $|W|$, and it can become $|D| \cdot |W|$ in the worst case.

In the following sections, we develop a new indexing scheme AUTOTREE, with the properties given in Theorem 1. A combination of four main ideas will lead us to this new scheme: a tree over the words (Section 2), relative bit vectors (Section 3), pushing up the words (Section 4), and dividing into blocks (Section 5). In Section 6, we will complement our theoretical findings with experiments on a large test collection.

All space and time bounds are concisely stated in formal lemmas, the proofs of which can be found in [18].

2 Building a Tree Over the Words (TREE)

The idea behind our first scheme on the way to Theorem 1 is to *increase the amount of preprocessing by precomputing inverted lists not only for words but also for their*

prefixes. More precisely, we construct a complete binary tree with m leaves, where m is the number of distinct words in the collection. We assume here and throughout the paper that m is a power of two. For each node v of the tree, we then precompute the sorted list D_v of documents which contain at least one word from the subtree of that node. The lists of the leaves are then exactly the lists of an ordinary inverted index, and the list of an inner node is exactly the union of the lists of its two children. The list of the root node is exactly the set of all non-empty documents. A simple example is given in Figure 1.

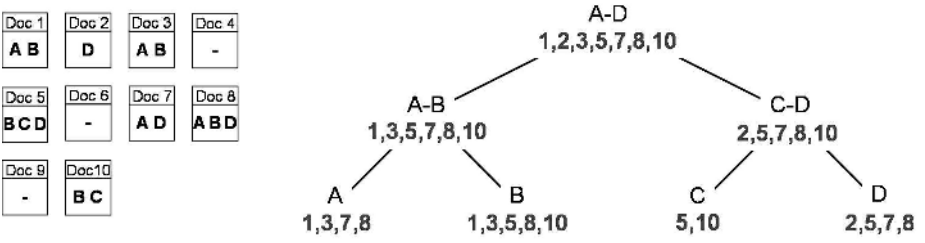


Fig. 1. Toy example for the data structure of scheme TREE with 10 documents and 4 different words

Given this tree data structure, an autocompletion query given by a word range W and a set of documents D is then processed as follows.

1. Compute the unique minimal sequence $v_1, \ldots, v_\ell$ of nodes with the property that their subtrees cover exactly the range of words W. Process these ℓ nodes from left to right, and for each node v invoke the following procedure.
2. Fetch the list D_v of v and compute the intersection $D \cap D_v$. If the intersection is empty, do nothing. If the intersection is non-empty, then if v is a leaf corresponding to word w, report for each $d \in D \cap D_v$ the pair (w, d). If v is not a leaf, invoke this procedure (step 2) recursively for each of the two children of v.

Scheme TREE can potentially save us time: If the intersection computed at an inner node v in step 2 is empty, we know that none of the words in the whole subtree of v is a completion leading to a hit, that is, *with a single intersection we are able to rule out a large number of potential completions*. However, if the intersection at v is non-empty, we know nothing more than that there is *at least one word* in the subtree which will lead to a hit, and we will have to examine both children recursively. The following lemma shows the potential of TREE to make the query processing time depend on the output size instead of on W as for BASIC. Since TREE is just a step on the way to our final scheme AUTOTREE, we do not give the exact query processing time here but just the number of nodes visited, because we need exactly this information in the next section.

Lemma 2. *When processing an autocompletion query (D, W) with TREE, at most $2(|W'| + 1) \log_2 |W|$ nodes are visited, where W' is the set of all words from W that occur in at least one document from D.*

The price TREE pays in terms of space is large. In the worst case, each level of the tree would use just as much space as the inverted index stored at the leaf level, which would give a blow-up factor of $\log_2 m$.

3 Relative Bitvectors (TREE+BITVEC)

In this section, we describe and analyze TREE+BITVEC, which reduces the space usage from the last section, while maintaining as much as possible of its potential for a query processing time depending on W', the set of matching completions, instead of on W. *The basic trick will be to store the inverted lists via relative bit vectors*. The resulting data structure turns out to have similarities with the static 2-dimensional orthogonal range counting structure of Chazelle [7].

In the root node, the list of all non-empty documents is stored as a bit vector: when N is the number of documents, there are N consecutive bits, and the ith bit corresponds to document number i, and the bit is set to 1 if and only if that document contains at least one word from the subtree of the node. In the case of the root node this means that the ith bit is 1 if and only if document number i contains any word at all.

Now consider any one child v of the root node, and with it store a vector of N' bits, were N' is the number of 1-bits in the parent's bit vector. To make it interesting already at this point in the tree, assume that indeed some documents are empty, so that not all bits of the parent's bit vector are set to one, and $N' < N$. Now the jth bit of v corresponds to the jth 1-bit of its parent, which in turn corresponds to a document number i_j. We then set the jth bit of v to 1 if and only if document number i_j contains a word in the subtree of v.

The same principle is now used for every node v that is not the root. Constructing these bit vectors is relatively straightforward; it is part of the construction given in Appendix A.

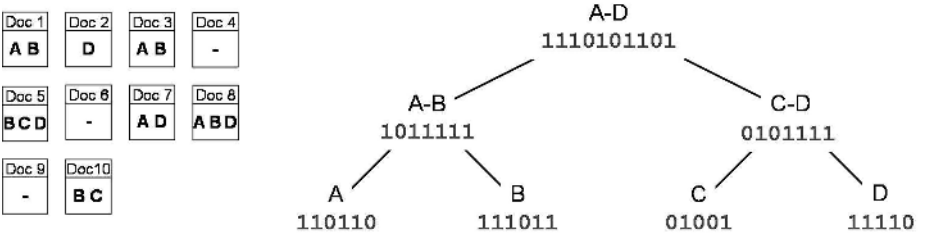


Fig. 2. The data structure of TREE+BITVEC for the toy collection from Figure 1

Lemma 3. *Let s_{tree} denote the total lengths of the inverted lists of algorithm TREE. The total number of bits used in the bit vectors of algorithm TREE+BITVEC is then at most $2s_{tree}$ plus the number of empty documents (which cost a 0-bit in the root each).*

The procedure for processing a query with TREE+BITVEC is, in principle, the same as for TREE. The only difference comes from the fact that the bit vectors, except that of the root, can only be interpreted relative to their respective parents.

To deal with this, we ensure that whenever we visit a node v, we have the set $\mathcal{I}_v$ of those positions of the bit vector stored at v that correspond to documents from the given set D, as well as the $|\mathcal{I}_v|$ numbers of those documents. For the root node, this is trivial to compute. For any other node v, $\mathcal{I}_v$ can be computed from its parent u: for each $i \in \mathcal{I}_u$, check if the ith bit of u is set to 1, if so compute the number of 1-bits at positions less than or equal to i, and add this number to the set $\mathcal{I}_v$ and store by it the number of the document from D that was stored by i. With this enhancement, we can follow the same steps as before, except that we have to ensure now that whenever we visit a node that is not the root, we have visited its parent before. The lemma below shows that we have to visit an additional number of up to $2\log_2 m$ nodes because of this.

Lemma 4. *When processing an autocompletion query (D, W) with TREE+BITVEC, at most $2(|W'| + 1)\log_2 |W| + 2\log_2 m$ nodes are visited, with W' defined as in Lemma 2.*

4 Pushing Up the Words (TREE+BITVEC+PUSHUP)

The scheme TREE+BITVEC+PUSHUP presented in this section gets rid of the $\log_2 |W|$ factor in the query processing time from Lemma 4. *The idea is to modify the TREE+BITVEC data structure such that for each element of a non-empty intersection, we find a new word-in-document pair (w, d) that is part of the output.* For that we store with each single 1-bit, which indicates that a particular document contains a word from a particular range, one word from that document and that range. We do this in such a way that each word is stored only in one place for each document in which it occurs. When there is only one document, this leads to a data structure that is similar to the priority search tree of McCreight, which was designed to solve the so-called 3-sided dynamic orthogonal range-reporting problem in two dimensions [8].

Let us start with the root node. Each 1-bit of the bit vector of the root node corresponds to a non-empty document, and we store by that 1-bit the *lexicographically smallest* word occurring in that document. Actually, we will not store the word but rather its number, where we assume that we have numbered the words from $0, \ldots, m-1$.

More than that, for all nodes at depth i (i.e., i edges away from the root), we omit the leading i bits of its word number, because for a fixed node these are all identical and can be computed from the position of the node in the tree. However, asympotically this saving is not required for the space bounds in Theorem 1 as dividing the words into blocks will already give a sufficient reduction of the space needed for the word numbers.

Now consider anyone child v of the root node, which has exactly one half H of all words in its subtree. The bit vector of v will still have one bit for each 1-bit of its parent node, but the definition of a 1-bit of v is slightly different now from that for TREE+BITVEC. Consider the jth bit of the bit vector of v, which corresponds to the jth set bit of the root node, which corresponds to some document number i_j. Then this document contains at least one word — otherwise the jth bit in the root node would not have been set — and the number of the lexicographically smallest word contained is stored by that jth bit. Now, if document i_j contains other words, and at least one of these *other* words is contained in H, only then the jth bit of the bit vector of v is set

to 1, and we store by that 1-bit *the lexicographically smallest word contained in that document that has not already been stored in one of its ancestors* (here only the root node).

Figure 3 explains this data structure by a simple example. The construction of the data structure is relatively straightforward and can be done in time $O(N)$. Details are given in Appendix A.

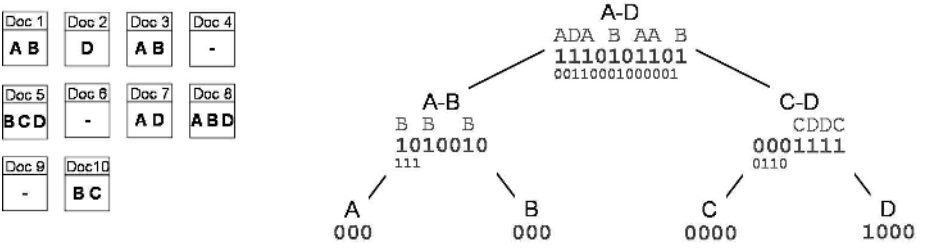


Fig. 3. The data structure of TREE+BITVEC+PUSHUP for the example collection from Figure 1. The large bitvector in each node encodes the inverted list. The words stored by the 1-bits of that vector are shown in gray on top of the vector. The word list actually stored is shown below the vector, where A=00, B=01, C=10, D=11, and for each node the common prefix is removed, e.g., for the node marked C-D, C is encoded by 0 and D is encoded by 1. A total of 49 bits is used, not counting the redundant 000 vectors and bookkeeping information like list lengths etc.

To process a query we start at the root. Then, we visit nodes in such an order that whenever we visit a node v, we have the set $\mathcal{I}_v$ of exactly those positions in the bit vector of v that correspond to elements from D (and for each $i \in \mathcal{I}_v$ we know its corresponding element d_i in D). For each such position with a 1-bit, we now check whether the word w stored by that 1-bit is in W, and if so output (w, d_i). This can be implemented by random lookups into the bit vector in time $O(|\mathcal{I}_v|)$ as follows. First, it is easy to intersect D with the documents in the root node, because we can simply lookup the document numbers in the bitvector at the root. Consider then a child v of the root. What we want to do is to compute a new set I_v of document indices, which gives the numbering of the document indices of D in terms of the numbering used in v. This amounts to counting the number of 1-bits in the bitvector of v up to a given sequence of indices. Each of these so-called *rank* computations can be performed in constant time with an auxiliary data structure that uses space sublinear in the size of the bitvector [19].

Consider again the check whether a word w stored by a 1-bit corresponding to a document from D is actually in W. This check can only fail for relatively few nodes, namely those with a least one leaf not from W in their subtree. These checks do not contribute an element to the output set, and are accounted for by the factor β mentioned in Theorem 1, and Lemmas 5 and 7 below.

Lemma 5. *With TREE+BITVEC+PUSHUP, an autocompletion query (D, W) can be processed in time $O\big(|D| \cdot \beta + \sum_{w \in W} |D \cap D_w|\big)$, where β is bounded by $\log_2 m$ as well as by the average number of distinct words in a document from D. For the special case, where W is the range of* all *words, the bound holds with $\beta = 1$.*

Lemma 6. *The bit vectors of TREE+BITVEC+PUSHUP require a total of at most $2N + n$ bits. The auxiliary data structure (for the constant-time rank computation) requires at most N bits.*

5 Divide into Blocks (TREE+BITVEC+PUSHUP+BLOCKS)

This section is our last station on the way to our main result, Theorem 1.

For a given B, with $1 \leq B \leq m$, we divide the set of all words in blocks of equal size B. We then construct the data structure according to TREE+BITVEC+PUSHUP for each block separately. As we only have to consider those blocks, which contain any words from W, this gives a further speedup in query processing time. An autocompletion query given by a word range W and a set of documents D is then processed in the following three steps.

1. Determine the set of ℓ (consecutive) blocks, which contain at least one word from W, and for $i = 1, \ldots, \ell$, compute the subrange W_i of W that falls into block i. Note that $W = W_1 \dot\cup \cdots \dot\cup W_\ell$.
2. For $i = 1, \ldots, \ell$, process the query given by W_i and D according to TREE+ BITVEC+ PUSHUP, resulting in a set of matches $M_i := \{(w, d) \in C : w \in W_i, d \in D\}$, where C is the set of of word-in-document pairs.
3. Compute the union of the sets of matching word-in-document pairs $\cup_{i=1}^{\ell} M_i$ (a simple concatenation).

Lemma 7. *With TREE+BITVEC+PUSHUP+BLOCKS and block size B, an autocompletion query (D, W) can be processed in time $O\big(|D| \cdot (\alpha + \beta) + \sum_{w \in W} |D \cap D_w|\big)$, where $\alpha = |W|/B$ and β is bounded by $\log_2 B$ as well as by the average number of distinct words from $W_1 \cup W_\ell$ (the first and the last subrange from above) in a document from D.*

Lemma 8. *TREE+BITVEC+PUSHUP+BLOCKS with block size B requires at most $3N + n \cdot \lceil m/B \rceil$ bits for its bit vectors and at most $N \lceil \log_2 B \rceil$ bits for the word numbers stored by the 1-bits. For $B \geq mn/N$, this adds up to at most $N(4 + \lceil \log_2 B \rceil)$ bits.*

Part (a) of Theorem 1 is established by the construction given in Appendix A. Part (b) of Theorem 1 follows from Lemma 8 by choosing $B = \lceil nm/N \rceil$. This choice of B minimizes the space bound of Lemma 8, and we call the corresponding data structure AUTOTREE. Part (c) of Theorem 1 follows from Lemma 7 and the following remarks. If the words in a document with L words are a random size-L subset of all words, then the average number of words per document that fall into a fixed block is at most 1. In our experiments, the average value for β was 2.2. For the exact definition of β, see [18].

6 Experiments

We tested both AUTOTREE and our baseline BASIC on the corpus of the TREC 2004 Robust Track (ROBUST '04), which consists of the documents on TREC disks 4 and

5, minus the Congressional Record [20]. We implemented AUTOTREE with a block size of 4096, which is the optimal block size according to Section 5, rounded to the nearest power of two. The following table gives details on the collection and on the space consumption of the two schemes; as we can see, AUTOTREE does indeed use no more space (and for this collection, in fact, significantly less) than BASIC, as guaranteed by Theorem 1.

Table 1. The characteristics of our test collection: n = number of documents, m = number of distinct words, N/n = average number of distinct words in a document, B^* = space-optimal choice for the block size. The last two columns give the space usage of BASIC and AUTOTREE in bits per word-in-document pair.

						bits per word-in-doc pair	
Collection	raw size	n	m	N/n	B^*	BASIC	AUTOTREE
ROBUST '04	1,904 MB	528,025	771,189	219.2	4,096	19.0	13.9

Queries are derived from the 200 "old" [4] queries (topics 301-450 and 601-650) of the TREC Robust Track in 2004 [20], by "typing" these queries from left to right, taking a minimum word length of 4 for the first query word, and 2 for any query word after the first. From these autocompletion queries we further omitted those, which would be obtained by simple filtering from a prefix according to the explanation following Definition 1. This filtering procedure is identical for AUTOTREE and BASIC and takes only a small fraction of the time for the autocompletion queries processed according to Definition 1, which is why we omitted it from consideration in our experiments. To give an example, for the ad hoc query `world bank criticism`, we considered the autocompletion queries `worl`, `world ba`, and `world bank cr`. We considered a total number of 512 such autocompletion queries.

We implemented BASIC and AUTOTREE in C++ and measured query processing times on a Dual Opteron machine, with 2 Intel Xeon 3 GHz processors, 8 GB of main memory, running Linux. We measured the time for producing the output according to Definition 1. The time for scoring and ranking would be identical for AUTOTREE and BASIC, and would, according to a number of tests, take only a small fraction of the aforementioned processing time. We therefore excluded it from our measurements. For BASIC, we implemented a fast linear-time intersect, which, on average, turned out to be faster than its asymptotically optimal relatives [21].

The results from Table 2 conform nicely to our theoretical analysis. Four main observations can be made: (i) with respect to maximal query processing time, which is key for an interactive application, AUTOTREE improves over BASIC by a factor of 10; (ii) in average processing time, which is significant for throughput in a high-load scenario, the improvement is still a factor of 4; (iii) processing times of AUTOTREE are sharply concentrated around their mean, while for BASIC they vary widely (in both directions as we checked); (iv) the almost perfect correlation between query processing times and our analytical bounds (explained in the caption of Figure 2) demonstrates

[4] They are "old" as they had been used in previous years for TREC.

Table 2. Processing times statistics of BASIC and AUTOTREE for all 512 autocompletion queries. The 6th and 7th column show the kth worst processing time, where k is 10% and 5%, respectively, of the number of queries. The last column gives the correlation factor between query processing times and total list volume $\sum_{w \in W}(|D| + |D_w|)$ for BASIC, and input size plus total output volume $|D| + 5\sum_{w \in W} |D \cap D_w|$ for AUTOTREE.

Scheme	Max	Mean	StdDev	Median	90%-ile	95%-ile	Correlation
BASIC	6.32secs	0.19secs	0.55secs	0.034secs	0.41secs	0.93secs	0.996
AUTOTREE	0.63secs	0.05secs	0.06secs	0.028secs	0.11secs	0.14secs	0.973

Table 3. Breakdown of query processing for BASIC and AUTOTREE by number of query words

	1-word		multi-word	
Scheme	Max	Mean	Max	Mean
BASIC	0.10secs	0.01secs	6.32secs	0.30secs
AUTOTREE	0.37secs	0.09secs	0.63secs	0.02secs

both the soundness of our theoretical modelling and analysis as well as the accuracy of our implementation.

Table 3, finally, breaks down query processing times by the number of query words. As we can see, BASIC is significantly faster than AUTOTREE for the 1-word queries, however, not because AUTOTREE is slow, but because BASIC is extremely fast on these queries. This is so, because BASIC does not have to compute any intersections for 1-query but merely has to copy all relevant lists D_w to the output, whereas AUTOTREE has to extract, for each output element, bits from its (packed) document id and word id vectors. On multi-word queries, BASIC has to process a much larger volume than AUTOTREE, and we see essentially the situation discussed above for the overall figures.

References

1. Witten, I.H., Bell, T.C., Moffat, A.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edition. Morgan Kaufmann (1999)
2. Bast, H., Weber, I.: Type less, find more: Fast autocompletion search with a succinct index. In: 29th Conference on Research and Development in Information Retrieval (SIGIR'06). (2006)
3. Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys **30**(2) (1998) 170–231
4. Arge, L., Samoladas, V., Vitter, J.S.: On two-dimensional indexability and optimal range search indexing. In: 18th Symposium on Principles of database systems (PODS'99). (1999) 346–357
5. Ferragina, P., Koudas, N., Muthukrishnan, S., Srivastava, D.: Two-dimensional substring indexing. Journal of Computer and System Science **66**(4) (2003) 763–774

6. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: 41st Symposium on Foundations of Computer Science (FOCS'00). (2000) 198–207
7. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing **17**(3) (1988) 427–462
8. McCreight, E.M.: Priority search trees. SIAM Journal on Computing **14**(2) (1985) 257–276
9. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In: 32nd Symposium on the Theory of Computing (STOC'00). (2000) 397–406
10. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. Journal of the ACM **46**(2) (1999) 236–280
11. Jakobsson, M.: Autocompletion in full text transaction entry: a method for humanized input. In: Conference on Human Factors in Computing Systems (CHI'86). (1986) 327–323
12. Darragh, J.J., Witten, I.H., James, M.L.: The reactive keyboard: A predictive typing aid. IEEE Computer (1990) 41–49
13. Stocky, T., Faaborg, A., Lieberman, H.: A commonsense approach to predictive text entry. In: Conference on Human Factors in Computing Systems (CHI'04). (2004) 1163–1166
14. Bickel, S., Haider, P., Scheffer, T.: Learning to complete sentences. In: 16th European Conference on Machine Learning (ECML'05). (2005) 497–504
15. Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., Ruppin, E.: Placing search in context: The concept revisited. In: 10th World Wide Web Conference (WWW'10). (2001) 406–414
16. Paynter, G.W., Witten, I.H., Cunningham, S.J., G., G.B.: Scalable browsing for large collections: A case study. In: 5th Conference on Digital Libraries (DL'00). (2000) 215–223
17. Nevill-Manning, C.G., Witten, I., Paynter, G.W.: Lexically-generated subject hierarchies for browsing large collections. International Journal of Digital Libraries **2**(2/3) (1999) 111–123
18. Bast, H., Mortensen, C.W., Weber, I.: Output-sensitive autocompletion search. Technical Report **1–007** (2006) See first author's website `http://www.mpi-inf.mpg.de/~bast/publications.html`.
19. Munro, J.I.: Tables. In: 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96). (1996) 37–42
20. Voorhees, E.: Overview of the trec 2004 robust retrieval track. In: 13th Text Retrieval Conference (TREC'04). (2004) http://trec.nist.gov/pubs/trec13/papers/ROBUST.OVERVIEW.pdf.
21. Demaine, E.D., Lopez-Ortiz, A., Munro, J.I.: Adaptive set intersections, unions, and differences. In: 11th Symposium on Discrete Algorithms (SODA'00). (2000) 743–752

A The Index Construction for TREE+BITVEC+PUSHUP

In this appendix we describe the construction of the index for TREE+BITVEC+PUSHUP. Full proofs of Lemmas 2, 3, 4, 5, 6, 7, and 8 can be found in [18].

The construction of the tree for algorithm TREE+BITVEC+PUSHUP is relatively straightforward and takes *constant amortized time* per word-in-document occurrence (assuming each document contains its word sorted in ascending order).

1. Process the documents in order of ascending document numbers, and for each document d do the following.
2. Process the distinct words in document d in order of ascending word number, and for each word w do the following. Maintain a *current node*, which we initialize as an artificial parent of the root node.

3. If the current node does not contain w in its subtree, then set the current node to its parent, until it does contain w in its subtree. For each node left behind in this process, append a 0-bit to the bit vector of those of its children which have not been visited.
   Note: for a particular word, this operation may take non-constant time, but once we go from a node to its parent in this step, the old node will never be visited again. Since we only visit nodes, by which a word will be stored and such nodes are visited at most three times, this gives constant amortized time for this step.
4. Set the current node to that one child which contains w in its subtree. Store the word w by this node. Add a 1-bit to the bit vector of that node.

A Compressed Self-index Using a Ziv-Lempel Dictionary

Luís M.S. Russo* and Arlindo L. Oliveira

INESC-ID/IST
{lsr, aml}@algos.inesc-id.pt

Abstract. A compressed full-text self-index for a text T, of size u, is a data structure used to search patterns P, of size m, in T that requires reduced space, i.e. that depends on the empirical entropy (H_k, H_0) of T, and is, furthermore, able to reproduce any substring of T. In this paper we present a new compressed self-index able to locate the occurrences of P in $O((m + occ) \log n)$ time, where occ is the number of occurrences and σ the size of the alphabet of T. The fundamental improvement over previous LZ78 based indexes is the reduction of the search time dependency on m from $O(m^2)$ to $O(m)$. To achieve this result we point out the main obstacle to linear time algorithms based on LZ78 data compression and expose and explore the nature of a recurrent structure in LZ-indexes, the $\mathcal{T}_{78}$ suffix tree. We show that our method is very competitive in practice by comparing it against the LZ-Index, the FM-index and a compressed suffix array.

1 Overview

The exact matching problem consists in searching for a short (pattern) sequence P in a longer (text) sequence T. Naive and linear solutions for this problem can be found in undergraduate computer science textbooks [1]. This problem has outgrown its initial motivation, text editing subroutines. Text databases storing large amounts of information such as pitch sequences, DNA or protein sequences, large natural texts, program code, etc. need fast pattern matching algorithms. With the increasing amount of digital information available, on-line approaches to the problem stopped being viable. The study of index data structures, that are able to reduce the time it takes to locate the occurrences of P, has been the focus of the string processing community for several years. Classical indexes however have a tendency to be space greedy. This constitutes a severe problem, since not being able to store indexes in main memory limits their usage.

In recent years a new and extremely successful approach to this problem has emerged. *Compressed full-text indexes*, which use data compression techniques to produce less space demanding data structures have been proposed by several

* Supported by the Portuguese Science and Technology Foundation by grant SFRH/BD/12101/2003 in project POCI 2010 and Project BIOGRID POSI/SRI/47778/2002.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 163–180, 2006.

researchers [2, 3, 4, 5, 6]. Usually a text stored in compress format requires less space than its uncompressed version. The idea is that an index based on the compressed format may also require less space. In fact, it turns out that data compression algorithms explore the internal structure of a string much in the same way that indexes do. An important tool to describe the space of compressed indexes is the k-th order empirical entropy of T defined by Manzini [7], denoted simply by H_k. The empirical entropy provides a measure of the complexity of T taken as a finite object. This is opposed to the classical notion of entropy by Shannon. State of the art compressed indexes consider T as finite and organise it globally. In a way our contribution is to organise globally Ziv-Lempel compressed indexes that were only locally organised. The empirical entropy provides a lower bound to the number of bits needed to compress T using a compressor that encodes each character considering only the context of k characters that follow it in T. Makinen and Navarro presented a comprehensive survey on compressed full-text indexes [8].

A surprising way to reduce the space requirements of a full-text index, discovered in this line of research, is to turn it into a self-index. Basically it turned out that with a negligible amount of information, it is possible to make full-text indexes reproduce any substring of T without storing T explicitly.

Compressed suffix arrays [6, 2] and the FM-index [3] are the main trends of compressed indexes. This is partially due to the fact that LZ-indexes [3, 4, 5] require a considerable amount of time to determine the number of occurrences of P in T, denoted by *occ*. In fact, the index of Kärkkäinen et al. [5], which was not a self-index, required $O(m^2 + (m + occ) \log u)$ time and Navarro's [4] index required $O((m^3 \log \sigma) + (m + occ) \log u)$ which was recently improved to $O((m^2 \log m) + (m + occ) \log u)$ by Arroyuelo et al. [9]. It can be seen that in all these approaches the dependency on m is at least $O(m^2)$. The only LZ based index that was able to achieve $O(m)$ time was presented by Ferragina et al. [3]. However this index requires a considerable amount of space, $O(uH_k(T) \log^{\epsilon} u) + o(u)$ bits. In fact the index presented by Ferragina et al. is not used in practice. Instead they simple add an FM-Index to their structure. Using an FM-Index may lead to alphabet related problems, i.e. large hidden σ dependencies. Some solutions have been presented to address this problem [10, 11]. However our approach is simpler and alphabet independent.

The Ziv-Lempel algorithm is a dictionary based compression method. In essence, the idea is that, given T, the algorithm infers a suitable dictionary and encodes T accordingly. The problem with compressed indexes based on this approach is that the encoding of T is not suitable for pattern matching. In fact the dictionary generated by the Ziv-Lempel algorithm is dynamically updated at the same time that T is processed. This means that the same string may be encoded in several different ways, since the dictionary changes from one occurrence, of the string, to another. This results in an undesirable encoding. The solution to this problem forces us to destroy the on-line property of the Ziv-Lempel algorithm. Our algorithm runs in two phases: in the first one we use the LZ78 algorithm to infer a dictionary; in the second one we organise T in an off-line way.

2 Basic Concepts and Notation

For basic concepts related to strings and suffix trees we refer the reader to Gusfield [12]. We use the following conventions: strings start at index position 0; prefixes, substrings and suffixes are denoted respectively as $S[..i]$, $S[i..j]$, $S[j..]$; m is the size of the pattern string P, u is the size of the text string T and occ is the number of occurrences of P in T. By suffix tree we refer to a generalised suffix tree. The terminator symbols are not considered as part of the edge-labels. A point is a node in the suffix trie. We refer indifferently to points in a suffix tree and to their path-labels. SDEP(p) is the string depth of point p. FATHER(v) is

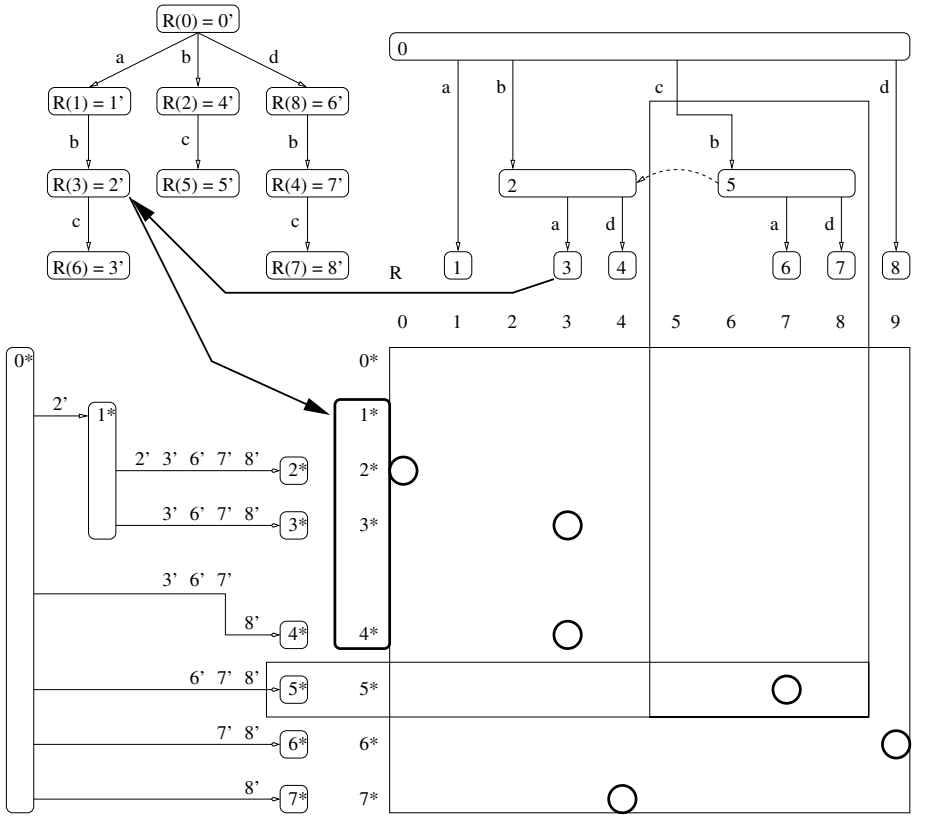


Fig. 1. (top-right) Suffix tree for strings $\{a, b, ba, bd, cba, cbd, d\}$. Suffix link from cb to b shown by a dashed arrow. Nodes show their DFS value in $\mathcal{T}$. (top-left) Reverse tree of the suffix tree on the right. Nodes show their DFS value in $\mathcal{T}^R$. The R mapping is shown and $R(3)$ is indicated by a bold arrow. (bottom-left) Sparse suffix tree of T, nodes show their $\text{DFS}_{\mathcal{ST}}$ values. Weak descent $W(\text{ROOT}_{\mathcal{ST}}, 2')$ shown in bold rectangle. (bottom-right) Linking points over spaces supported by DFS' and $\text{DFS}_{\mathcal{ST}}$ values. Orthogonal range query [5*,5*]:[5,8].

the father node of node v. SUFFIXLINK(v) is v's suffix link. LETTER(v, i) equals $v[i]$, i.e. the i-th letter of the path-label of node v. DESCEND?(p, c) is true iff it is possible to descend from point p with c and DESCEND(p, c) returns the resulting point. By DFS(v) we refer to the depth-first time-stamp [1] of a node v in a suffix tree and by DFS'(p) to the depth-first time-stamp of a point p in a suffix trie. As a running example consider $T = cbdbddcbababa$ and $\mathcal{T}$ as the suffix tree in figure 1 (top-right).

Definition 1. *The* ***range*** $I(p)$ *of a point* p *of a suffix tree* $\mathcal{T}$ *is the interval of the* DFS' *values of the points that are descendants of* p.

In our example DFS(c) is undefined, DFS(cb) = 5, DFS'(c) = 5, DFS'(cb) = 6, $I(c) = [5, 8]$.

Definition 2. *The* ***reverse tree*** $\mathcal{T}^R$ *of a suffix tree* $\mathcal{T}$ *is the minimal labelled tree that, for every node* v *of* $\mathcal{T}$*, contains a node* v^R*, where* v^R *denotes the reverse string of* v.

The tree $\mathcal{T}^R$ is shown in figure 1 (top-left). Observe for example that, since cbd is a node of $\mathcal{T}$, there is a node $cbd^R = dbc$ in $\mathcal{T}^R$. We define a canonical mapping R that, for every node v in $\mathcal{T}$, maps DFS(v) to DFS(v^R) (see figure 1). We will use $R(v)$ to denote R(DFS(v)). Note that since the nodes of $\mathcal{T}$ form a suffix closed set, the nodes of $\mathcal{T}^R$ form a prefix closed set.

2.1 Succinct Suffix Trees

Our approach is based on suffix trees. We start by presenting a representation of suffix trees that is adequate for our goals and analyse its space requirements.

By bitmap B we refer to a string over $\{0, 1\}$. Fundamental tools to produce succinct data structures are the RANK and SELECT operations over bitmaps. The operation RANK(B, i) counts the number of 1's in $B[..i-1]$ and SELECT(B, i) returns the smallest j such that RANK($B, j+1$) = i. Munro [13] showed how to support these operations in $O(1)$ time and $|B| + o(|B|)$ bits.

Geary et al. [14] presented a succinct representation of ordinal d-node trees in $2d + o(d)$ bits, supporting, among others, the following operations in constant time: ANC(v, j) returns the j-th ancestor of node v (for example ANC(v, 1) is FATHER(v)); LEFTRANK(v) returns DFS(v); RIGHTRANK(v) returns the largest DFS value among the descendants of v; SELECT(j) returns the node with DFS time j; CHILD(v, j) returns the j-th child of node v; DEG(v) returns the number of children of node v; DEPTH(v) returns the tree depth of node v.

We assume that the tree structure of $\mathcal{T}$ and $\mathcal{T}^R$ are stored using the previous representation. Arroyuelo et al. [9] proposed a way to represent the R mapping. Since R is a permutation, R and R^{-1} can be stored using the representation of Munro et al. [15] in $(1+\epsilon)d \log d + o(d)$ bits, where ϵ is fixed and $0 < \epsilon \leq 1$. This way R and R^{-1} can be computed in $O(1)$ and $O(1/\epsilon)$ time respectively.

Lemma 1. *A suffix tree* $\mathcal{T}$ *with* d *nodes can be stored in* $(1+\epsilon)d(\log d) + 5d + o(d)$ *bits. Let* p *be a point,* c *a letter and* v *a node of* $\mathcal{T}$*. This representation*

provides the operations given by Geary et al. in $O(1)$ time. Moreover it provides $\textsc{Sdep}(v)$ *in $O(1)$ time,* $\textsc{Suffix\_Link}(v)$, $\textsc{Letter}(v,i)$, *in $O(1/\epsilon)$ time and* $\textsc{Descend?}(p,c)$, $\textsc{Descend}(p,c)$ *in $O((\log\sigma)/\epsilon)$ time.*

Proof. According to our notation $R(v)$ represents $\textsc{Select}_{\mathcal{T}^R}(R(\textsc{LeftRank}(v)))$. $\textsc{Sdep}(v)$ can be computed as $\textsc{Depth}_{\mathcal{T}^R}(\textsc{Select}_{\mathcal{T}^R}(R(\textsc{LeftRank}(v))))$ which can be represented as $\textsc{Depth}_{\mathcal{T}^R}(R(v))$. The operation $\textsc{Suffix\_Link}(v)$ is computed as $R^{-1}(\textsc{Father}_{\mathcal{T}^R}(R(v)))$. Observe that $v[0]$ represents the letter just below the root. For example $cbd[0] = c$. We define a bitmap D to compute $v[0]$, in a way similar to Sadakane [2]. We have that $D[0] = 1$ and, for $i > 0$, $D[i] = 0$ iff $\textsc{Dfs}(v) = i$, $\textsc{Dfs}(v') = i + 1$ and $v[0] = v'[0]$. In our example $D = 11001001$. We can compute $v[0]$, when v is not the $\textsc{Root}$, in $O(1)$ as the letter in position $\textsc{Rank}_1(D, \textsc{Dfs}(v))$ of Σ. This requires $d + o(d)$ bits. The operation $\textsc{Letter}(v,i)$ can be computed from $R^{-1}(\textsc{Anc}_{\mathcal{T}^R}(R(v),i))$. This expression represents following enough suffix links to make the letter we want appear just bellow the root, i.e. $\textsc{Letter}(v,i) = R^{-1}(\textsc{Anc}_{\mathcal{T}^R}(R(v),i)[0]$. When p is not a node, $\textsc{Descend?}(p,c)$ can be computed in $O(1/\epsilon)$ time by consulting $\textsc{Letter}$ for the point below p. If p is a node, we do a binary search among the children of p. If we find a child that starts with c, we return true. Procedure $\textsc{Descend}(p,c)$ updates the value of p. When p is a point, this is done in $O(1)$ time. When p is a node, we first proceed as *Descend?*. □

Finally observe that with this representation we cannot compute $\textsc{Dfs'}(v)$. The $\textsc{Dfs'}$ values are essential to our algorithm because they serve as a supporting space for range queries.

Lemma 2. *For a suffix tree $\mathcal{T}$ with d nodes and t points, operations* $\textsc{Dfs'}(p)$ *and $I(p)$ can be computed in $O(1)$ time using $(2 + \lceil \log t \rceil - \lfloor \log d \rfloor)d + o(d)$ extra bits.*

Proof. Observe that the $\textsc{Dfs'}(v)$ values appear sorted in $\textsc{Dfs}(v)$ order. Therefore we can store the $\textsc{Dfs'}(v)$ values, for the nodes of $\mathcal{T}$, with the representation of Grossi et al. [6, Lemma 2]. For a point p, $\textsc{Dfs'}(p)$ is computed as $\textsc{Dfs'}(v) - \textsc{Sdep}(v) + \textsc{Sdep}(p)$, where v is the highest node that is a descendant of p. Also $I(p) = [\textsc{Dfs'}(p), \textsc{Dfs'}(\textsc{Select}(\textsc{RightRank}(v))]$. □

2.2 Descend and Suffix Walks

Given a string P we can traverse a suffix tree $\mathcal{T}$ in greedy way, i.e. start at $\textsc{Root}$ and descend as much as possible. When it is impossible to descend any further, follow suffix-links until descending becomes possible again, as in Algorithm 1.

Definition 3. *The* ***descend and suffix walk*** *of a string P over a suffix tree $\mathcal{T}$ is the sequence $p_0 \dots p_{2m}$ of points of $\mathcal{T}$ computed by Algorithm 1.*

Definition 4. *The* ***right, left traces*** *of a string P over a suffix tree $\mathcal{T}$ are the sub-sequences of the descend and suffix walk, given respectively by lines 6 and 8 of Algorithm 1.*

Algorithm 1. Descend and Suffix Walk Algorithm

```
 1: procedure DESCEND&SUFFIX(P)
 2:     P ← P.$'
 3:     j ← 0
 4:     point ← ROOT
 5:     for i ← 0, i < |P| do
 6:         trace_left[i] ← point
 7:         while NOT DESCEND?(point, P[i]) do
 8:             trace_right[j] ← point
 9:             j++
10:             point ← SUFFIXLINK(point)
11:         end while
12:         point ← DESCEND(point, P[i])
13:     end for
14: end procedure
```

Table 1. (Top) Descend and suffix walk of *cbdbddc* in $\mathcal{T}$. (Bottom) Values for locating type > 1 occurrences.

i	0	1	2	3	4	5	6	7
P[i]	c	b	d	b	d	d	c	$'
trace_left[i]	ϵ	c	cb	cbd	b	bd	d	c
DFS'(father_left[i])	0	0	6	8	2	4	9	0
DFS'(trace_left[i])	0	5	6	8	2	4	9	5
DFS'(child_left[i])	0	6	6	8	2	4	9	6
trace_right[i]	cbd	bd	d	bd	d	d	c	ϵ
DFS'(father_right[i])	8	4	9	4	9	9	0	0
DFS'(trace_right[i])	8	4	9	4	9	9	5	0
I(trace_right[i])	[8,8]	[4,4]	[9,9]	[4,4]	[9,9]	[9,9]	[5,8]	[0,9]
DFS'(child_right[i])	8	4	9	4	9	9	6	0
P[i..]	cbd.bd.d.c	bd.bd.d.c	d.bd.d.c	bd.d.c	d.d.c	d.c	c	ϵ
tail(P[i..])	c	c	c	c	c	c	c	ϵ
H(P[i..])	748	448	848	48	88	8	ϵ	ϵ
R(H(P[i..]))	6'7'8'	udef	udef	6'7'	6'6'	6'	ϵ	ϵ
|father_left[i]| == i		FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
W(R(H(P[i..])), R(father_left[i]))			$\emptyset$	[5*,5*]	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
I(tail(P[i..]))		[5,8]	[5,8]	[5,8]	[5,8]	[5,8]	[5,8]	[0,9]
occ'			0	1	0	0	0	0

By **father_right**$[i]$ (resp. **father_left**$[i]$), we refer to the lowest ancestor of $trace\_right[i]$ (resp. $trace\_left[i]$) that is node of $\mathcal{T}$ and by **child_right**$[i]$ (resp. **child_left**$[i]$), to the highest descendant of $trace\_right[i]$ (resp. $trace\_left[i]$) that is node of $\mathcal{T}$.

Note that we define in an artificial way SUFFIXLINK(ROOT) as a node that descends to the root by every letter including terminator symbols. It is important to notice that Algorithm 1 starts by appending to P a terminator character $\$'$ that fails to match with any other character. Observe that in Algorithm 1 the operation SUFFIXLINK is computed for points, not just nodes. This is done in the

classical way. The operation SuffixLink over points doesn't have $O(1/\epsilon)$ guaranteed time. However the total time of Algorithm 1 amortises to $O((m/\epsilon)\log\sigma)$ (see Gusfield [12] for details). Table 1 (top) shows the descend and suffix walk of *cbdbddc* in $\mathcal{T}$.

3 A Full-Text Index Using Suffix Tree Dictionaries

In this section we explain the main contribution of this paper. Our data structure is very similar to an inverted file. We will use this similarity to provide insight into the algorithm.

3.1 Generic Inverted Index

Throughout section 3 we assume that we are given an arbitrary suffix tree $\mathcal{T}$ with d nodes, that we will use as a dictionary. We consider as dictionary *words* the path-labels of the nodes of $\mathcal{T}$. The first thing we should do is to organise T according to our dictionary $\mathcal{T}$, much like what is done in inverted files when given a lexicon.

Definition 5. *The* $\mathcal{T}$***-maximal parsing*** *of string* T *is the sequence of nodes* $v_1, \ldots, v_f$ *such that* $T = v_1 \ldots v_f$ *and, for every* j, v_j *is the largest prefix of* $v_j \ldots v_f$ *that is a node of* $\mathcal{T}$.

We assume that $\mathcal{T}$ is appropriate for T, i.e. that it is possible to parse T in a maximal way. In our example, the $\mathcal{T}$-maximal parsing of a string T is the sequence cbd, bd, d, cba, ba, ba. We refer to the elements of the $\mathcal{T}$-maximal parsing of T as *blocks*. We will store the $\mathcal{T}$-maximal parsing of T in compact form as a string of numbered blocks.

Definition 6. *The* ***translation*** $V(v_1 \ldots v_f)$ *of a sequence* $v_1 \ldots v_f$ *of nodes is a string such that* $V(v_1 \ldots v_f)[i] = \textsc{Dfs}(v_i)$.

We denote by $\mathcal{T}(T)$ the translation of the $\mathcal{T}$-maximal parsing of T. Since the $\mathcal{T}$-maximal parsing of T is the sequence cbd, bd, d, cba, ba, ba, its translation is the string $\mathcal{T}(T) = 748633$. Note that word ba is associated with two blocks, v_5 and v_6.

Inverted files usually store a list of occurrences for every word of the dictionary. To play this role we will use a stronger indexing structure, a sparse suffix tree. For technical reasons we must reverse the string $\mathcal{T}(T)$. This is achieved by extending the canonical mapping R to sequences in the following way: $R(v_1 \ldots v_f) = R(v_f) \ldots R(v_1)$. In our example $R(\mathcal{T}(T)) = R(748633) = R(3)R(3)R(6)R(8)R(4)R(7) = 2'2'3'6'7'8'$. This corresponds to the notion of reverse string, because the concatenation of the path-labels of $R(\mathcal{T}(T))$ in $\mathcal{T}^R$ is $ab.ab.abc.d.db.dbc = T^R$.

Definition 7. *The* ***sparse suffix tree***[1] $\mathcal{ST}$ *of a string* T *and a suffix tree* $\mathcal{T}$ *is the suffix tree of* $R(\mathcal{T}(T))$.

[1] Similar to a concept defined by Kärkkäinen et al. [16]

The sparse suffix tree of our example is shown in figure 1 (bottom-left). We can descend in the sparse suffix tree in the usual way with $\text{DESCEND}_{\mathcal{ST}}$. However, since $\mathcal{T}^{\mathcal{R}}$ provides the alphabet for $\mathcal{ST}$, we can also take that into consideration when descending.

Definition 8. *The* ***weak descent*** $W(p, v^R)$ *for a point* p *in* $\mathcal{ST}$ *and a node* v^R *in* $\mathcal{T}^R$ *is the interval of* $\text{DFS}_{\mathcal{ST}}$ *values of the nodes below the following points:*
$\{p.\text{DFS}_{\mathcal{T}^R}(v') \mid v'$ *is a descendant of* v^R *in* $\mathcal{T}^R\}$

For example, $W(\text{ROOT}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, since this contains the $\text{DFS}_{\mathcal{ST}}$ values for the nodes below $2', 3'$ in $\mathcal{ST}$, see figure 1. This can be computed in $O((\log d)/\epsilon)$ time. We do two binary searches in the children of p, searching for $\text{LEFTRANK}_{\mathcal{T}^R}(v)$ and $\text{RIGHTRANK}_{\mathcal{T}^R}(v)$. Then $W(p, v^R) = [\text{LEFTRANK}_{\mathcal{ST}}(v''), \text{RIGHTRANK}_{\mathcal{ST}}(v''')]$, where v'' and v''' are the nodes found by the binary searches.

In order to find occurrences of strings across more than one block, we will need to store the relations across contiguous blocks. This motivates the following two definitions.

Definition 9. *The* ***head***, ***tail*** *of the* $\mathcal{T}$*-maximal parsing are respectively sequence* $v_1, \ldots, v_i$ *and string* $v_{i+1} \ldots v_f$ *such that* $v_1, \ldots, v_i$ *is the smallest sequence for which* $v_{i+1} \ldots v_f$ *is a point in* $\mathcal{T}$*.*

We denote by $H(T)$ the translation of the head of the $\mathcal{T}$-maximal parsing of T. The head of the $\mathcal{T}$-maximal parsing of T is cbd, bd, d, cba, ba and the tail is the string ba. Hence $H(T)$ equals 74863.

Next we define a set of points relating the leaves of $\mathcal{ST}$ with the points in $\mathcal{T}$.

Definition 10. *The* ***linking points set*** *of the* $\mathcal{T}$*-maximal parsing* $v_1 \ldots v_f$ *of* T *is the following set:*

$$\mathcal{L} = \left\{ \langle \text{DFS}(R(V(v_1 \ldots v_i))), \text{DFS'}(p_i) \rangle \;\middle|\; \begin{array}{l} p_i \text{ is the largest prefix of } v_{i+1} \ldots v_f \\ \text{that is a point in } \mathcal{T}, \text{ for } 0 < i \leq f \end{array} \right\}$$

The set $\mathcal{L}$ is shown in figure 1 (bottom-right) and consists of the following points:

- $\langle \text{DFS}(R(V(cbd, bd, d, cba, ba, ba))), \text{DFS'}(\epsilon) \rangle = \langle \text{DFS}(2'2'3'6'7'8'), 0 \rangle = \langle 2^*, 0 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd, d, cba, ba))), \text{DFS'}(ba) \rangle = \langle \text{DFS}(2'3'6'7'8'), 3 \rangle = \langle 3^*, 3 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd, d, cba))), \text{DFS'}(ba) \rangle = \langle \text{DFS}(3'6'7'8'), 3 \rangle = \langle 4^*, 3 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd, d))), \text{DFS'}(cba) \rangle = \langle \text{DFS}(6'7'8'), 7 \rangle = \langle 5^*, 7 \rangle$
- $\langle \text{DFS}(R(V(cbd, bd))), \text{DFS'}(d) \rangle = \langle \text{DFS}(7'8'), 9 \rangle = \langle 6^*, 9 \rangle$
- $\langle \text{DFS}(R(V(cbd))), \text{DFS'}(bd) \rangle = \langle \text{DFS}(8'), 4 \rangle = \langle 7^*, 4 \rangle$

We need to process the linking points to be able to compute orthogonal range queries. Chazelle [17] presented a minimal space structure for computing range queries in a $[1, f] \times [1, f]$ grid, that uses $f \log f(1 + o(1))$ bits and $O(f \log f)$ time to be built. It reports points in $O((1 + occ') \log f)$ time, where occ' is the number of points reported. We want to use this data structure for the $[0, d' - 1] \times [0, t - 1]$ space, where d' is the number of nodes of $\mathcal{ST}$. However we only need to store f points. Therefore we must reduce the support spaces to rank spaces. The space

$[0, d'-1]$ can be reduced to $[1, f]$ in $O(1)$ time, with RANK over a bitmap of $d' + o(d')$ bits. The space $[0, t-1]$ requires more bits to be reduced. We store an array containing the DFS' values of the linking points. This array requires $(2 + \lceil \log t \rceil - \lfloor \log f \rfloor)f + o(f)$ extra bits using the representation of Grossi et al. [6]. The reduction is obtained in $O(\log f)$ time with a binary search over this array.

We propose an index data structure composed of the dictionary $\mathcal{T}$, the sparse suffix tree $\mathcal{ST}$ and the linking points $\mathcal{L}$. We will now explain how to use this index to solve the exact matching problem. Our search algorithm proceeds differently depending on whether the pattern is completely contained inside a block or spans more than one block. We refer to this as **type** 1 and **type** > 1 occurrences.

3.2 Occurrences Lying Inside a Single Block

The algorithm for finding occurrences inside a single block starts by identifying all the words in the dictionary $\mathcal{T}$ that contain P as a substring. Since $\mathcal{T}$ is a suffix tree, it is possible to achieve this in a simple way.

- Descend by P in $\mathcal{T}$. If this is impossible then there are no type 1 occurrences of P.
- Start a depth-first traversal of the sub-tree below P.
- For each node v reached compute the range query $W(\text{ROOT}_{\mathcal{ST}}, p^R) : [0, t]$.

The search in $\mathcal{T}$ consists in considering words that start with P and appending some letters. The weak descend and the range query consist in prepending some letters to the words found on the search in $\mathcal{T}$. For example, consider $P = b$. By reading b, we reach node 2 of $\mathcal{T}$, see figure 1. The search on $\mathcal{T}$ returns nodes $2, 3, 4$, i.e. leads us to consider words b, ba, bd. This originates the following weak descends: $W(\text{ROOT}_{\mathcal{ST}}, 4') = \emptyset$, $W(\text{ROOT}_{\mathcal{ST}}, 2') = [1^*, 4^*]$, $W(\text{ROOT}_{\mathcal{ST}}, 7') = [6^*, 7^*]$. We don't need to consider words that start with b, since they don't correspond to blocks; there may be occurrences of ba or cba because of ba; there may be occurrences of bd and cbd because of bd. The range queries return no occurrences for b, occurrences 2^*, 3^* and 4^* for ba and occurrences 6^* and 7^* for bd. This corresponds to occurrences *cbd.bd.d.cba.ba.*$\underline{b}$*a*, *cbd.bd.d.cba.*$\underline{b}$*a.ba*, *cbd.bd.d.c*$\underline{b}$*a.ba.ba* for ba and occurrences *cbd.*$\underline{b}$*d.d.cba.ba.ba*, *c*$\underline{b}$*d.bd.d.cba.ba.ba*, for bd.

Theorem 1. *The above procedure is correct and complete.*

Proof. (Correct) Clearly every reported block is $\alpha.P.\beta$ for some α,β and hence it contains an occurrence of P. (Complete) Suppose block $v_i = \alpha.P.\beta$, hence $\alpha.P.\beta$ is a node in $\mathcal{T}$. Since $\mathcal{T}$ is a suffix tree, $P.\beta$ is also a node in $\mathcal{T}$. Node $P.\beta$ is reached by the search in $\mathcal{T}$, since it starts by P. Every node v of $\mathcal{ST}$ for which $v[0] = \text{DFS}((\alpha.P.\beta)^R)$ has its $\text{DFS}_{\mathcal{ST}}$ time in $W(\text{ROOT}_{\mathcal{ST}}, (P.\beta)^R)$, hence block v_i is found in the range query. □

This algorithm was essentially presented by Navarro [4], except that the range queries were computed as depth-first searches in a trie similar to $\mathcal{T}^R$. In Navarro's

algorithm each node of that trie stored one block. Therefore the time of theses searches was bounded by the number of type 1 occurrences of p, denoted by occ_1. We do not have a direct correspondence between the nodes of $\mathcal{T}^R$ and the blocks of $\mathcal{T}$-maximal parsing, which means that this approach has no worst case guarantees. In essence the problem is that we may be executing more range queries than the number of occurrences found.

Definition 11. *A **spurious** entry for string T in the suffix tree $\mathcal{T}$ is a leaf v of $\mathcal{T}$ such that v^R is a leaf of $\mathcal{T}^R$ and v is not a block in the $\mathcal{T}$-maximal parsing of T.*

For a dictionary $\mathcal{T}$ without spurious entries, we can guarantee that some orthogonal range queries must return occurrences.

Lemma 3. *Assuming $\mathcal{T}$ has no spurious entries for T and v is a leaf of $\mathcal{T}$, then the query $W(\text{ROOT}_{\mathcal{ST}}, v^R) : [0, t]$ returns at least one linking point.*

Proof. There is some α such that $(\alpha.v)^R$ is a leaf in $\mathcal{T}^R$. Since $\mathcal{T}$ is a suffix tree and v is a leaf of $\mathcal{T}$, then $\alpha.v$ is also a leaf of $\mathcal{T}$. Hence, at least one linking point will be found by $W(\text{ROOT}_{\mathcal{ST}}, v^R) : [0, t]$, since $\text{DFS}_{\mathcal{ST}}((\alpha.v)^R) \in W(\text{ROOT}_{\mathcal{ST}}, v^R)$. □

Spurious entries may be safely removed from the dictionary. Removing spurious entries can be done by considering $\mathcal{T}$ and $\mathcal{T}^R$ as a DAG, i.e. a node w in the DAG represents simultaneously v and v^R; there is an edge from w to w' if that edge exists in $\mathcal{T}$ or in $\mathcal{T}^R$. To remove spurious entries we perform a DFS over this DAG. We remove nodes that do not have blocks and are sinks or unary and the edge comes from $\mathcal{T}$. The nodes are checked and removed in their finishing time (see Cormen et al. [1] for definitions). This procedure runs in $O(d)$ time. Note that the resulting structure remains a suffix tree.

3.3 Occurrences Spanning More Than a Single Block

In this section we focus on finding occurrences that span two or more consecutive blocks, i.e. type > 1. The ideas presented in this section are similar to those of Kärkkäinen et al. [16] and related with the approach proposed by Ferragina et al. [3].

We are now faced with the problem of retrieving the words in our dictionary that appear concatenated in $\mathcal{T}(T)$ and have P as a substring. Suppose that $P = cbdbddc$ and that we split P in two as $cbdbdd$ and c. We will now search for c in $\mathcal{T}$ and for $cbdbdd$ in $\mathcal{ST}$. The point c in $\mathcal{T}$ induces the range $I(c) = [5, 8]$; on the other hand string $cbdbdd$ is parsed into cbd, bd, b and hence will be translated into 748. To search on the sparse suffix tree, we need $R(748) = 6'7'8'$. This will induce the range $[5^*, 5^*]$. Finally, to solve our problem we perform the orthogonal range query $[5^*, 5^*] : [5, 8]$ over the linking points $\mathcal{L}$. This corresponds to the question: is the string $cbdbdd$, parsed as $cbd.bd.d$, ever followed by a block that starts by c? The answer is yes, since there is a linking point in $[5^*, 5^*] : [5, 8]$. This point corresponds to $\underline{cbd.bd.d.c}ba.ba.ba.$

We will now explain how to determine in which points to break P. The pattern should be separated in the head and tail of $P[i..]$, for every $0 < i < m$, to account for every possible translation that can occur. These points can be determined using the following dynamic programming equations:

$$tail(P[i..]) = \begin{cases} trace\_right[i] & \text{, if } |trace\_right[i]| = m - i \\ tail(P[i + |father\_right[i]|..]) & \text{, otherwise} \end{cases}$$

$$H(P[i..]) = \begin{cases} \epsilon & \text{, if } |trace\_right[i]| = m - i \\ father\_right[i].H(P[i + |father\_right[i]|..]) & \text{, otherwise} \end{cases}$$

We use Algorithm 3.3 to locate points $R(H(P[i..]))$ in $\mathcal{ST}$. Whenever it is not possible to descend by a letter, the $\text{DESCEND}_{\mathcal{ST}}$ procedure returns the $udef$ state. See table 1 (bottom) for an example of this computation. Assume that the descend and suffix walk of P is already computed. Hence the arguments of $\text{DESCEND}_{\mathcal{ST}}$ are available when $\text{DESCEND}_{\mathcal{ST}}$ is executed. Therefore Algorithm 3.3 runs in $O((m/\epsilon)\log d)$ time, since it runs m times the $\text{DESCEND}_{\mathcal{ST}}$ operation, which requires $O((\log d)/\epsilon)$ time.

Algorithm 2. Locate $R(H(P[i..]))$ Algorithm

```
1: procedure Locate_HPI
2:     for i ← m − 1, 0 < i do
3:         R(H(P[i..])) ← ROOT_ST
4:         if |trace_right[i]| < m − i then
5:             R(H(P[i..])) ← DESCEND_ST(R(H(P[i + |father_right[i]|..])), father_right[i])
6:         end if
7:     end for
8: end procedure
```

Having located $tail(P[i..])$ in $\mathcal{T}$ and $R(H(P[i..]))$ in $\mathcal{ST}$, we know where to break the pattern. Now all that we need are the ranges for the range query. The range for $\mathcal{T}$ is simply $I(tail(P[i..]))$. Whenever $P[..i-1]^R$ is a node of $\mathcal{T}^{\mathcal{R}}$ the range for $\mathcal{ST}$ is $W(R(H(P[i..])), P[..i-1]^R)$.

Let us consider for example the case of $i = 3$. We have that $H(P[3..]) = 48$ and $R(H(P[3..])) = 6'7'$. Hence $W(6'7', (cbd)^R) = [5^*, 5^*]$, since $8'$ is the only descendant of itself in $\mathcal{T}^R$. This means that, when we are extending $bd.d$ to the left by prepending a word from our dictionary that terminates in cbd, the only such word is cbd. Therefore we end up considering only the node $cbd.bd.d$.

Our algorithm for finding type > 1 occurrences of P proceeds as follows:

- Compute the descend and suffix walk of P in $\mathcal{T}$.
- Compute $tail(P[i..])$ from the descend and suffix walk of P.
- Locate the $R(H(P[i..]))$ points in $\mathcal{ST}$.
- If $|father\_left[i]| = i$ then $P[..i-1]^R = R(father\_left[i])$, compute $W(R(H(P[i..])), R(father\_left[i]))$.
- Compute $I(tail(P[i..]))$ from $tail(P[i..])$.
- Compute the orthogonal range queries $W(R(H(P[i..])), R(father\_left[i])) : I(tail(P[i..]))$.

An example of our algorithm is shown in Table 1 (bottom). The only range query that finds occurrences (occ') is the $[5^*, 5^*] : [5, 8]$ query, as we have explained in this Section.

4 A Compressed Self-Index Based on LZ78 Dictionaries

We found it interesting to present this work in a general form, since it seems relevant to explore other techniques for inferring dictionaries, given a text T. We will now give a concrete instantiation of the above algorithm, using the Ziv-Lempel 78 Algorithm [18].

Definition 12. *The **LZ78 parsing** of a string T is the sequence $Z_1, \ldots, Z_n$ of strings such that $T = Z_1 \ldots Z_n$ and for every i, $Z_i = Z_j c$ where Z_j is the largest prefix of $Z_i \ldots Z_n$ among the $Z_1, \ldots, Z_{i-1}$.*

Given a string T, we proceed as follows: compute the LZ78 parsing of $T^R = Z_1 \ldots Z_n$, then consider the suffix tree for strings $\{Z_1^R, \ldots, Z_n^R\}$ as our dictionary, denoted by $\mathcal{T}_{78}$. In our example T^R is parsed into $a, b, ab, abc, d, db, dbc$ and the resulting dictionary can be seen in figure 1 (top-right). The following lemmas expose why the dictionary we propose is adequate in terms of space.

Lemma 4. *If the number of blocks of the LZ78 parsing of T is n then the $\mathcal{T}_{78}$ has at most $2n$ nodes, i.e. $d \leq 2n$.*

Proof. Observe that every suffix of a Z_i^R is a Z_j^R for some j. Therefore the set $\{Z_1^R, \ldots, Z_n^R\}$ is suffix closed. Hence a suffix tree based on $\{Z_1^R, \ldots, Z_n^R\}$ will have at most $2n$ nodes. □

Lemma 5. *If the number of blocks of the LZ78 parsing of T is n then the $\mathcal{T}_{78}$-maximal parsing of T has at most n blocks, i.e. $f \leq n$.*

Proof. The idea is to show that if a block v_i of the $\mathcal{T}_{78}$-maximal parsing is a substring of some Z_j^R then it is a suffix. Suppose that v_i is a substring of Z_j^R. We have that $Z_j^R = \alpha.v_i.\beta$. Since the dictionary is a suffix tree and Z_j^R is a node, $v_i\beta$ is also a node and hence a dictionary word. Since the parsing is maximal, we have that $v_i.\beta = v_i$, i.e. that v_i is a suffix of Z_j^R. □

4.1 Space and Time Complexity

We will refer to the index that uses LZ78 dictionaries as the Inverted-LZ-Index. The next theorem gives an overview of the space/time complexity of this structure.

Theorem 2. *Let d and d' be the number of nodes of $\mathcal{T}_{78}$ and $\mathcal{ST}_{78}$ respectively. Let t be the number of points of $\mathcal{T}_{78}$. Let f be the size of the $\mathcal{T}_{78}$-maximal parsing of T. The space/time trade-off of the Inverted-LZ-Index can be summarised as follows:*

Space in bits	$[\frac{d}{n}(\frac{\lceil \log t \rceil - \lfloor \log d \rfloor}{\log u} + 1 + \epsilon) + \frac{d'}{n}(1+\epsilon) + \frac{f}{n}(\frac{\lceil \log t \rceil - \lfloor \log f \rfloor}{\log u} + 1)]uH_k + o(u \log \sigma)$
Time to count	$O((occ + m/\epsilon) \log n)$
Time to locate	*free after counting*
Time to display l chars	$O(l/(\epsilon \log_\sigma u))$ *with* $3u$ *extra bits*
Conditions	$k = o(\log_\sigma u)$, $\sigma = O(n)$, $0 < \epsilon \leq 1$, ϵ *is constant*

Proof. (Space) The space requirements come from adding up the space of $\mathcal{T}_{78}$, $\mathcal{ST}_{78}$ and the range data structure. Ziv et al. [18] showed that $\sqrt{u} \leq n \leq u/\log_\sigma u$, and, therefore $n = o(u \log \sigma)$. The relation between n and H_k was established by Kosaraju et al. [19] who showed that $n \log u = uH_k + o(u \log \sigma)$ for $k = o(\log_\sigma u)$.

(Count/Locate) We have already seen that Algorithm 1 runs in $O((m/\epsilon) \log \sigma)$ time. The time to find occurrences of type 1 is $O((1 + occ_1) \log n)$. Observe that the number of queries computed is less than or equal to twice the number of leaves below P. By lemma 3 we know that the queries at the leaves must return occurrences. Therefore the total time amortises to $O((1+occ_1) \log n)$. The time to find occurrences of type > 1 is the time of Algorithm 3.3, plus m weak descents and m range queries. Therefore the total time for occurrences of type > 1 is $O((occ_{>1} + m/\epsilon) \log n)$, where $occ_{>1}$ is the number of type > 1 occurrences.

(Display) Observe that even though we don't store $R(\mathcal{T}_{78}(T))$ explicitly, we have $O(1/\epsilon)$ access time to it. The idea is to store a pointer to the leaf of $\mathcal{ST}_{78}$ with path-label $R(\mathcal{T}_{78}(T))$, denoted by $\textsc{FirstLeaf}_{\mathcal{ST}}$. Therefore $R(\mathcal{T}_{78}(T))[i] = \textsc{Letter}_{ST}(\textsc{FirstLeaf}_{\mathcal{ST}}, i)$. Hence we can compute the j-th letter of $R(\mathcal{T}_{78}(T))[i]$ in as $\textsc{Letter}(\textsc{Letter}_{ST}(\textsc{FirstLeaf}_{\mathcal{ST}}, i), j)$, in $O(1/\epsilon)$ time. To achieve optimal $O(l/(\epsilon \log_\sigma u))$ time we use an approach based on the work of Sadakane [20], similar to Arroyuelo et al. [9]. We define a new bitmap D' similar to bitmap D used to retrieve the first $\log u$ bits of a node v instead of the first letter. This requires $d + o(d)$ bits. We also need a bitmap Q that indicates which sequences of $\log u$ bits do appear as the first bits of some v. By $(i)_2$ we denote the binary representation of i, with $\log u$ bits. The Q bitmap is defined as $Q[i] = 1$ iff $(i)_2$ is the prefix of some $(v)_2$ padded with zeros. Bitmap Q contains $2^{\log u} = u$ bits and can therefore be stored in $u + o(u)$ bits. With these bitmaps we are able to retrieve $\log u$ bits from a block in $O(1)$ time, i.e. $\log_\sigma u$ letters. We repeat these bitmaps for $\mathcal{ST}_{78}$ and hence are able to retrieve $\log u$ bits from consecutive blocks. Finally we need another bitmap to be able to skip blocks. We use a bitmap V that marks the beginnings of the blocks in $R(\mathcal{T}_{78}(T))$. This requires another $u + o(u)$ bits. As pointed out by Arroyuelo et al. [9], this bitmap can be used to report the occurrences of P as positions in T instead of as a block and an offset. □

The worst case of the space expression is $(6.5 + 4\epsilon)H_k + o(u \log \sigma)$. However the worst example we were able to find, based on De Bruijn cycles, yielded $(5.5 + 3\epsilon)H_k + o(u \log \sigma)$ bits. In the next section we show concrete values for the space expression.

5 Practical Issues and Testing

We implemented a prototype for testing these ideas. It was pointed out by Navarro [4] that the range data structure was space consuming and actually slower in practice than to do a complete scan choosing the range that required less work. Therefore we did not implement the range data structure. Observe that this way we have no worst case guarantees for the search time.

The sparse suffix tree $\mathcal{ST}_{78}$ is stored in a suffix array fashion. The nodes of the $\mathcal{T}_{78}^R$ are stored as ranges over $\mathcal{ST}_{78}$, that correspond to the elements of $\mathcal{ST}_{78}$ that are traversed by the type 1 searches (see figure 1 for the range of node $2'$). The $\mathcal{T}_{78}$ tree is implemented in a pointer like fashion. Every node is stored in a memory cell indexed by its breath-first time-stamp. For example, node 6 will be stored in cell 3. The LETTER operation is replaced by a HEAD pointer, that, for every node v with father node $v[..i-1]$, points to node $v[i..]$. This information suffices to be able to read of edge-labels, by using suffix links. Every node stores its DFS time, a suffix link, the string depth, the HEAD pointer and the range of its corresponding R node.

We compared our implementation, Inverted-Lempel-Ziv-Index (ILZI), against Navarro's implementation of the FM-index (FMI), Sadakane's CSArray (CSAx1,CSAx8) and Navarro's LZ-Index (LZI), all of which are publicly available [21], using the files from the Pizza&Chili corpus [22][2].

We show the size of different indexes along with experimental values for the terms of the theoretical space requirements of our index, table 2. The FM-Index and the compressed suffix array needed to be parametrised. The parameters we used are also shown in table 2 in the *par* line. The parameter of the FM-Index was chosen with minimum value of 5 so that its size is close to the size of ILZI. The parameter of CSAx1 (resp. CSAx8) was chosen so that its size is close to the size of ILZI with $L = D$ (resp. $L = 8 \times D$). We used all the indexes to determine *occ* and reported this time divided by m as the counting time per character (c). We used all indexes to report occurrences, subtracted the counting time and divided by the number of occurrences found. We report this time as the reporting time per occurrence (r). Finally we used the indexes to display part of the text around the occurrences, subtracted the counting and reporting times, divided by the number of occurrences and letters. We report this time as the displaying time per character (o), also in table 2. The reporting time per occurrence is shown for different values of m, since for the LZ-based indexes this value is not constant. The time per occurrence and displaying time per character are relatively constant for different values of m and therefore we only present their values for $m = 20$.

In the space column of table 2 we present the ratios of the space size in bits with $u8$ and uH_k. In this, way for the raw string, we obtain the numbers of letters that should fit into a *byte*. Observe that our index has acceptable space requirements both in theory and in practice. For example for the xml file the practical

[2] Tested on Pentium 4, 3.2 GHz, 1 MB of L2, 1Gb of RAM, with Fedora Core 3, compiled with `gcc-3.4 -O9`.

Table 2. Results for test files. On the left we show the space values and on the right the time values in seconds (s). In the space column variable i represents the size of the different indexes and of the original string (Raw) in bits. Therefore $i/2^{23}$ gives the size in Megabytes (MB), $i/u8$ gives the ratio with the original string, i/uH_k gives the ratio with a compressed string, where H_k is estimated as $(n \log u)/n$. The bottom part of the space column shows empirical values for the space terms of our index, d/n, d'/n, f/n, $((\lceil \log t \rceil - \lfloor \log d \rfloor)/\log u)$ in column DFS', $((\lceil \log t \rceil - \lfloor \log f \rfloor)/\log u)$ in column $\mathcal{L}$ and the empirical value of the space expression in total. In the time column the best values among different indexes are displayed in bold and the second best are underlined.

File	Space							Time					
english		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	54.3	81.1	66.8	55.6	56.3	c 5	1.77e-3	6.78e-4	**1.30e-6**	3.41e-6	3.85e-6
	$i/u8$	1.00	1.09	1.62	1.34	1.11	1.13	c 10	4.33e-5	4.08e-5	**1.36e-6**	3.30e-6	3.80e-6
	i/uH_k	2.76	2.99	4.47	3.69	3.07	3.11	c 20	3.35e-6	3.01e-5	**1.19e-6**	2.92e-6	3.48e-6
	par				5	17	7	c 40	1.98e-6	3.17e-5	**1.06e-6**	2.43e-6	3.18e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	3.32e-7	**1.48e-7**	3.21e-5	7.28e-6	3.23e-6
	0.64	1.33	0.94	0.08	0.04	2.99 + 1.96 ϵ		o 20	3.09e-7	2.85e-7	**2.04e-7**	1.28e-6	9.16e-7
xml		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	26.1	44.5	64.9	26.2	25.8	c 5	4.37e-4	5.11e-4	**1.23e-6**	1.90e-5	5.02e-6
	$i/u8$	1.00	0.52	0.89	1.30	0.52	0.52	c 10	1.45e-4	1.69e-4	**1.30e-6**	1.41e-5	4.93e-6
	i/uH_k	5.08	2.65	4.52	6.60	2.67	2.62	c 20	3.25e-5	4.49e-5	**1.31e-6**	1.14e-5	4.86e-6
	par				5	44	19	c 40	6.18e-6	2.84e-5	**1.23e-6**	8.64e-6	4.68e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.40e-7**	4.67e-7	3.25e-5	2.00e-5	8.35e-6
	0.54	1.08	0.87	0.12	0.08	2.62 + 1.62 ϵ		o 20	2.84e-7	2.05e-7	**1.24e-7**	2.99e-6	1.97e-6
dna		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	44.0	60.9	63.4	45.1	37.0	c 5	1.93e-2	7.44e-3	**1.17e-6**	2.85e-6	4.72e-6
	$i/u8$	1.00	0.88	1.22	1.27	0.90	0.74	c 10	4.44e-4	1.76e-4	**1.42e-6**	3.57e-6	5.32e-6
	i/uH_k	3.63	3.19	4.42	4.60	3.27	2.69	c 20	3.51e-6	1.09e-5	**1.26e-6**	3.46e-6	5.16e-6
	par				5	26	11	c 40	1.66e-6	1.14e-5	**1.10e-6**	2.94e-6	4.92e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.61e-7**	3.98e-7	3.76e-5	1.37e-5	1.05e-5
	0.92	1.20	0.97	0.08	0.04	3.20 + 2.12 ϵ		o 20	2.93e-7	**2.62e-7**	7.78e-7	2.44e-6	2.66e-6
proteins		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	63.7	102.8	152.9	100.9	104.8	100.1	c 5	4.71e-4	1.88e-4	**1.27e-6**	3.12e-6	3.43e-6
	$i/u8$	1.00	1.61	2.40	1.58	1.64	1.57	c 10	3.77e-6	1.92e-5	**1.15e-6**	2.98e-6	3.37e-6
	i/uH_k	1.88	3.04	4.52	2.98	3.10	2.96	c 20	2.43e-6	2.16e-5	**1.03e-6**	2.51e-6	3.10e-6
	par				10	13	6	c 40	1.80e-6	2.30e-5	**9.53e-7**	1.88e-6	2.80e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**4.15e-7**	6.00e-7	1.69e-5	8.20e-6	6.47e-6
	0.85	1.22	0.98	0.04	0.04	3.11 + 2.07 ϵ		o 20	**3.12e-7**	4.27e-7	5.86e-7	1.11e-6	1.16e-6
pitches		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	53.2	84.7	124.8	86.8	85.6	86.1	c 5	2.58e-4	1.19e-4	**1.47e-6**	2.87e-6	3.06e-6
	$i/u8$	1.00	1.59	2.34	1.63	1.61	1.62	c 10	2.78e-5	3.78e-5	**1.34e-6**	2.68e-6	2.94e-6
	i/uH_k	1.99	3.16	4.66	3.24	3.19	3.21	c 20	1.15e-5	3.34e-5	**1.18e-6**	2.21e-6	2.60e-6
	par				9	12	5	c 40	6.78e-6	3.23e-5	**1.05e-6**	1.60e-6	2.21e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.39e-7**	4.85e-7	1.66e-5	5.60e-6	2.22e-6
	0.76	1.25	0.94	0.08	0.08	3.08 + 2.01 ϵ		o 20	2.66e-7	**1.45e-7**	6.33e-7	5.30e-7	4.06e-7
sources		Raw	ILZI	LZI	FMI	CSAx1	CSAx8	m	ILZI	LZI	FMI	CSAx1	CSAx8
	$i/2^{23}$	50.0	53.5	80.9	68.1	53.3	54.6	c 5	8.91e-4	3.66e-4	**1.40e-6**	3.16e-6	3.48e-6
	$i/u8$	1.00	1.07	1.62	1.36	1.07	1.09	c 10	1.22e-4	6.43e-5	**1.42e-6**	3.00e-6	3.45e-6
	i/uH_k	2.80	3.00	4.53	3.81	2.99	3.06	c 20	1.58e-5	3.19e-5	**1.27e-6**	2.68e-6	3.21e-6
	par				5	17	7	c 40	3.60e-6	2.95e-5	**1.13e-6**	2.30e-6	2.89e-6
	d/n	d'/n	f/n	DFS'	$\mathcal{L}$	total		r 20	**3.33e-7**	4.56e-7	3.67e-5	7.17e-6	3.01e-6
	0.60	1.19	0.90	0.08	0.04	2.78 + 1.79 ϵ		o 20	2.83e-7	2.70e-7	**2.36e-7**	1.15e-6	7.92e-7

value is $2.65uH_k$ bits and the theoretical value is $(2.62 + 1.62\epsilon)uH_k + o(u \log \sigma)$ bits.

The counting time per character of LZ-based indexes is affected by *occ*, whereas the FM-index and CSArray have a fairly constant value. This can be seen by the fact that the counting time per character decreases for larger values of m, where *occ* is smaller. By looking at the c lines of table 2 it can be seen that our reduction of the dependency on m from $O(m^2)$ to $O(m)$ had significant impact in the query time. This makes our index up to an order of magnitude faster than LZI for counting when m is large. Also, for a large m, our index sometimes qualifies second, being faster than the CSArray. For $m = 40$ it is very close to the best counting time, expect for the xml and the pitches file where it is respectively around 5 times and 6.5 times slower than the FM-Index. Contrarily, for small patterns, $m = 5$, it is up to 2.6 times slower than LZI and up to four orders of magnitude slower than the FM-Index and the CSArray.

On the other hand LZ-based indexes are extremely fast at reporting occurrences. In fact they are the only self-indexes using $O(uH_k)$ bits able to spend $O(\log n)$ time per occurrence. This is also visible in table 2 as our index and LZI rank first and second and are one to two orders of magnitude faster than the alternatives.

The displaying time per character is not a very decisive factor to tell indexes apart since all of them are very fast. The FM-index performed extremely well on natural language based files. The LZ-based indexes had more stable performance and are among the fastest for all samples.

6 Conclusions

This paper presents two fundamental observations on LZ78 based compressed indexes. The first one is that our dictionary $\mathcal{T}_{78}$ is a suffix tree. This structure was first presented by Kärkkäinen [5] but this version required T to be present and since it was based in LZ77, it was not necessarily a suffix tree. In the work presented by Navarro [4] the structure is called RevTrie but its suffix tree nature is not explored and, in fact, reading an edge-label requires $O(m^2)$. In the work presented by Ferragina and Manzini [3] it appears as an FM-Index of $T^R_\$$. They present an argument to prove that its space requirements can be related to the entropy of the text T. However its suffix tree structure is also not explored. The second observation is about the way the same string appears in the LZ78 parsing. A string S may appear in $O(m)$ different ways as the concatenation of LZ78 blocks. This, in turn, forces algorithms based on the LZ78 parsing to have quadratic behaviour. We solve this problem by discarding the original parsing and using a maximal parsing. In the maximal parsing, a string S appears in at most one way as the concatenation of blocks. Navarro uses the original LZ78 parsing. Ferragina and Manzini discard the parsing and solve the problem by using an FM-index, i.e. resorting to the Burrows-Wheeler transformation.

Our index is a significant contribution to LZ-based compressed indexes. We improved the counting time performance of LZ-based indexes to linear time. At

the same time, the structure we propose is smaller than LZI, for all the files we tested. In theory, with the terms we obtained in table 2, we can choose an ϵ to make the index smaller than $4uH_k+o(u\log\sigma)$. In practice it can be seen in table 2 that ILZI is always smaller than LZI. However a new version of the LZ-index proposed by Arroyuelo et al. [9] requires only $(2+\epsilon)uH_k+o(u\log\sigma)$ with worst case guarantees. Without worst case guarantees it requires $(1+\epsilon)uH_k+o(u\log\sigma)$ bits and it has $O(m^2)$ average search time for $m \geq 2\log_\sigma u$. It is interesting to notice that Arroyuelo et al. independently explored the suffix tree structure of $\mathcal{T}_{78}$ to reduce the time to read an edge-label to $O(m)$. We cannot achieve the reduced space requirements of Arroyuelo et al. essentially because we are storing more structures. In fact, as a second contribution of this paper, we pointed out a possible representation of suffix trees (lemma 1). This representation is not very competitive when compared to the compressed suffix trees presented by Sadakane [23]. Nevertheless it is adequate for our goals. For suffix trees, in general, it requires more space than the representation of Sadakane. In fact, the problem is the space required to store R and R^{-1}, $(1+\epsilon)n\log n$ bits. Arroyuelo et al. [9] showed how to reduce the space requirements of R. However even with such an improvement it is still not comparable to Sadakane's approach in terms of space. We expect further work based on this approach to produce a competitive representation.

Acknowledgements

We are deeply grateful to Gonzalo Navarro for several reasons: organising the Workshop on Compression, Text, and Algorithms at DCC in November of 2005 that motivated stimulating discussions on compressed indexes; providing prototypes together with Sadakane; creating the Pizza&Chili Corpus together with Ferragina; for suggestions and corrections along with Arroyuelo and several anonymous reviewers. We would like to thank Luis Coelho for countless discussions about our index.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. Second edn. McGraw (2001)
2. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms **48**(2) (2003) 294–313
3. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4) (2005) 552–581
4. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms **2**(1) (2004) 87–114
5. Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proceedings of the 3rd South American Workshop on String Processing, Carleton University Press (1996) 141–155
6. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2) (2005) 378–407

7. Manzini, G.: An analysis of the burrows-wheeler transform. J. ACM **48**(3) (2001) 407–430
8. Makinen, V., Navarro, G.: Compressed full text indexes. Technical Report TR/DCC-2006-6, Dept. of Computer Science, University of Chile (2006) 2nd version.
9. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index. In: Proceedings of CPM 2006. LNCS 4009 (2006) 319–330
10. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An alphabet-friendly FM-index. In: Proceedings of SPIRE 2004. LNCS 3246, Springer (2004) 150–160 Extended version to appear in ACM TALG.
11. Grabowski, S., Mäkinen, V., Navarro, G.: First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In: Proceedings of SPIRE 2004. LNCS 3246, Springer (2004) 210–211
12. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1999)
13. Munro, J.I.: Tables. In Chandru, V., Vinay, V., eds.: Proceedings of FSTTCS 1996. Volume 1180 of LNCS., Springer (1996) 37–42
14. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. In: SODA, SIAM (2004) 1–10
15. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations. In: ICALP. Volume 2719 of LNCS., Springer (2003) 345–356
16. Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: COCOON. Volume 1090 of LNCS., Springer (1996) 219–230
17. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. **17**(3) (1988) 427–462
18. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24**(5) (1978) 530–536
19. Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with lempel-ziv algorithms. SIAM J. Comput. **29**(3) (1999) 893–911
20. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: SODA, ACM Press (2006) 1230–1239
21. (http://www.dcc.uchile.cl/~gnavarro/eindex.html)
22. (http://pizzachili.dcc.uchile.cl/)
23. Sadakane, K.: Compressed suffix trees with full functionality. (to appear in Theory of Computing Systems)

Mapping Words into Codewords on PPM*

Joaquín Adiego and Pablo de la Fuente

Depto. de Informática, Universidad de Valladolid, Valladolid, Spain
{jadiego, pfuente}@infor.uva.es

Abstract. We describe a simple and efficient scheme which allows words to be managed in PPM modelling when a natural language text file is being compressed. The main idea for managing words is to assign them codes to make them easier to manipulate. A general technique is used to obtain this objective: a dictionary mapping on PPM modelling. In order to test our idea, we are implementing three prototypes: one implements the basic dictionary mapping on PPM, another implements the dictionary mapping with the separate alphabets model and the last one implements the dictionary with the spaceless words model. This technique can be applied directly or it can be combined with some word compression model. The results for files of 1 Mb. and over are better than those achieved by the character PPM which was taken as a base. The comparison between different prototypes shows that the best option is to use a word based PPM in conjunction with the spaceless word concept.

Keywords: Text Compression, PPM, Dictionary Algorithms, Natural Language Processing.

1 Introduction

In modern computational environments, processing times and storage costs have been reduced. On the other hand, the amount of data stored and transmitted has increased dramatically. Although most data is multimedia, the amount of textual data, predominant a few years ago, is not negligible. Information Retrieval Systems and Digital Libraries are systems where textual information, with and without format, is still predominant. Besides, these systems are used in several environments such as networks, optical and magnetical media. In these cases, the use of compression techniques is the best choice to solve storage problems and improve access time in storing and processing. Improvements in processing times are achieved thanks to the reduced disk transfer times necessary to access the text in compressed form. Since processor speeds in the last few decades have increased much faster than disk transfer speeds, trading disk transfer times for processor decompression times has become a much better choice [21]. On the other hand, the use of compression techniques reduces transmission times

* This work was partially supported by the TIC2003-09268 project from MCyT, Spain.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 181–192, 2006.

and increases the efficiency using communication channels. These compression propierties allow us to keep costs down.

Classical text compression algorithms perform compression at the character level. When an algorithm is adaptive then the algorithm slowly learns correlations between sequences of characters. However, the algorithm usually has a chance to take advantage of longer sequences before either the end of input is reached or the tables maintained by the algorithm reach their capacity. If text compression algorithms were to use larger units than single characters as the basic storage element, they would be able to take advantage of the longer range sequences and, perhaps, achieve better compression performance. Faster compression may also be possible by working with larger units [13].

In this paper, we explore the use of a word representation as the basic unit in PPM, one of the most promising lossless discrete-data compression algorithms at the character level, which uses Markov models of order k.

When the source file is a natural language document, we have no difficulty in recognizing a word as consisting of a sequence of consecutive letters. Each word is separated from the next by space and/or punctuation characters. Following the same approach as Bentley et al. [6], we generalize slightly by considering a natural language text file to consist of alternating alphanumeric-strings and punctuation-strings, where a word-string is a maximal sequence of alphanumeric characters and a punctuation-string is a maximal sequence of non-alphanumeric characters. We use the generic name *word* to refer to either an alphanumeric string or a punctuation string. The generalization allows us to decompose all kinds of text files into sequences of words. We should be able to take advantage of the fact the the alphanumeric and non-alphanumeric words strictly alternate.

The following sections of this paper will consider the problem of generalizing a PPM based compression algorithm to be word-based, then particular PPM word-based algorithms will be described, and finally some experimental results will be reported. Finally, our conclusions and future work are presented.

2 Natural Language Text Compression

With regard to compressing natural language texts the most successful techniques are based on models where the text words are taken as the source symbols [15], as opposed to the traditional models where the characters are the source symbols.

In an English text, for example, words follow a Zipf law, that is, the relative frequency of the i-th most frequent word is $1/i^{\theta}$, for some $1 < \theta < 2$ [20,3]. On the other hand, the model size (assigning a codeword to each different text word) is not significant in large text collections. Heaps law establishes that the number of different words in a text of n words is $O(n^{\beta})$ for some β between 0.4 and 0.6 [12,3]. Thus, the model size grows sublinearly with the collection size.

Natural language is not only made up of words. There are also punctuation, separator, and other special characters. The sequence of characters between every pair of consecutive words is called a *separator*. Separators must also be considered

to be symbols of the source alphabet. There are even fewer different separators than different words, and their distribution is even more skewed. Note that, since words and separators strictly alternate in the text, we can have two separate source alphabets, usually leading to better compression. As explained in the Introduction, we will use the generic name *words* to refer to both text words and separators in this paper.

Words reflect much better than characters the true entropy of the text [4]. For example, a semiadaptive Huffman coder over the model that considers characters as symbols typically obtains a compressed file whose size is around 60% of the original size, in natural language. A Huffman coder, when words are the symbols, obtains 25% [21]. Existing compression algorithms that consider the input as a sequence of words are ad hoc in nature. The scheme described by Bentley et al. [6] maintains a list of words sorted into least-recently used order. A word is encoded by its position in this dynamically changing list. Words near the front of the list tend to have shorter codes than those near the end and, assuming words in frequent use stay near the front of the list, compression is achieved. Another example is the WLZW algorithm, which uses Ziv-Lempel on words [10].

Since the text is not only composed of words but also separators, a model must also be chosen for them. An obvious possibility is to consider the different inter-word separators as symbols too, and make a unique alphabet for words and separators. However, this idea is not using a fundamental *alternation* property: words and separators always follow one another. In [15,5] two different alphabets are used: one for words and one for separators. Once it is known that the text starts with word or separator, there is no confusion on which alphabet to use. This model is called **separate alphabets**.

In [17,21] a new idea to use the two alphabets is proposed, called **spaceless words**. An important fact that is not used in the method of separate alphabets is that a word is followed by a single space in most cases. In general, it is possible to be emphasized that at least 70% of separators in text are single space [15]. Then, the spaceless words model take a single space as a *default*. That is, if a word is followed by a single space, we just encode the word. If not, we encode the word and then the separator. At decoding time, we decode a word and assume that a space follows, except if the next symbol corresponds to a separator. Of course the alternation property does not hold anymore, so we have a single alphabet for words and separators (single space excluded). This variation achieves slightly better compression ratios in reported experiments.

3 *k*-th Order Models

These models assign a probability to each source symbol as a function of the *context* of k source symbols that precede it. They are used to build very effective compressors such as Prediction by Partial Matching (PPM) and those based on the Burrows-Wheeler Transform (BWT).

PPM [9,18] is a statistical compressor that models the character frequencies according to the *context* given by the k characters preceding it in the text (this

is called a k-th order model), as opposed to Huffman that does not consider the preceding characters. Moreover, PPM is adaptive, so the statistics are updated as the compression progresses. The larger k is, the more accurate the statistical model and the better the compression are, but more memory and time is necessary to compress and uncompress.

The PPM technique can be viewed as blending together several fixed-order models to predict the next character in the input sequence. More exactly, PPM uses $k+1$ models, of order 0 to k, in parallel. It usually compresses using the k-th order model, unless the character to compress has never been seen in that model. In this case, it switches to a lower-order model until the character is found. The coding of each character is done with an arithmetic compressor, according to the computed statistics at that point. Well known representatives of this family are Shkarin/Cheney's *ppmdi* and Bloom/Tarhio's *ppmz*.

The BWT [7] is a reversible permutation of the text that puts together characters having the same k-th order context (for any k). The BWT is a composite of three different algorithms: (i) the block sorting main engine, a lossless and very slightly expansive preprocessor; (ii) the move-to-front coder (MTF), a byte-for-byte simple, fast, locally adaptive noncompressive coder; and (iii) a simple statistical compressor, like a first order Huffman or arithmetic coding, doing the compression. Steps (ii) and (iii) work like a local optimization over the permuted text obtaining results similar to k-th order compression.

In [16] the block-sorting algorithm of the BWT is extended to word-based models, including other transformations, like *spaceless words* mentioned above, in order to improve the compression. Experimental results shows that the combination of word-based modeling, BWT and MFT-like transformations allows to obtain good compression effectiveness to be attained within reasonable resource costs.

4 Dictionary Mapping

In this section we propose a word-based scheme on PPM. Our objective has been carried out plugging an additional layer to precede PPM that replaces words by two byte codewords, and then these codewords will be codified with a conventional PPM.

According to Skibinski et al. [19] replacing words by codewords has advantages and drawbacks. First, the concept of replacing words with shorter codewords from a given static dictionary has at least two shortcomings:

1. The dictionary must be quite large (at least tens of thousands of words) and it is appropriate for natural language only.
2. No "high level" correlations, e.g. related to grammar, are implicitly taken into account.

In spite of these drawbacks, such an approach to text compression turns out to be an attractive one, and it has not been given as much attention as it deserves. On the other hand, the benefits of dictionary-based text compression schemes

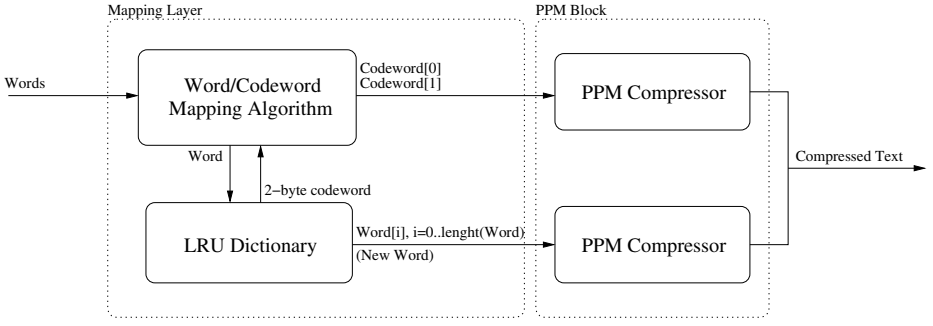


Fig. 1. Dictionary mapping

are: the ease of dictionary generation (assuming enough training text in a given language), clarity of ideas, high processing speed and acceptable compression ratios. Our proposal tries to solve both shortcomings.

Figure 1 shows a graphical representation of the dictionary mapping scheme. The proposal is made up of two different blocks: (1) Mapping Layer, which manages the vocabulary and maps words into codewords. To limit the number of different codes, a dictionary with a capacity of 2^{16} is used, when this dictionary is full an LRU policy is applied. (2) PPM Block, with two PPM compressors, which can be the same or not, one in charge of coding codewords and the other in charge of coding new words when they appear for the first time in the text.

This compression scheme can be seen as a PPC[1] compression scheme. This is a relatively new concept to compress data streams, based on the idea of preprocessing the data stream (through permutations and/or partitions) before compressing it. A successful PPC example is *bzip2*.

Algorithm 1 (Compression with dictionary mapping on PPM)

map.add(*ESCAPE_WORD*)
while (*there are more words*) **do**
  word ← *get_word*()
  if *map.find*(*word*) = *true*
    then
      CodePPM_ENCODE(*map.codeword*(*word*))
    else
      CodePPM_ENCODE(*map.codeword*(*ESCAPE_WORD*))
      for $0 \leq i <$ *word.length*() **do**
        WordPPM_ENCODE(*word*[*i*])
      od
      map.add(*word*)
  fi
od

[1] Permutation–Partition–Compression

Algorithm 1 shows a generic scheme for compressing the text using dictionary mapping on PPM. This scheme uses two independent PPM encoders, one codifies the codewords and the other codifies the new words character by character. When a new word is reached, a reserved codeword (the general escape mechanism) is emitted, the new word is codified using another PPM encoder and it is added to the dictionary.

We handle a limited length size dictionary in order to always obtain two byte codewords, therefore the dictionary capacity is at most 2^{16} words. When the dictionary is full and it is necessary to insert a new word, the least-recently used (LRU) word is removed from the dictionary and its place is occupied by the new word. The idea behind this decision is to remove from the dictionary the words that have been used just once and probably they will never be used again. Since we are codifing natural language texts which obey the Zipf law [20] it is quite unlikely that a word will constantly be coming in and out the dictionary. Using this technique, we can represent any word of the vocabulary with two bytes and, consequently, an order-k PPM modelling codewords can better predict word sequences using the same amount of memory as another order-k PPM modelling the untransformed text.

The decompression algorithm is similar.

5 Evaluation

Tests were carried out on the SuSE Linux 9.3 operating system, running on a computer with a Pentium IV processor at 1.5 GHz and 384 megabytes of RAM. We used a **g++** compiler with full optimization. For the experiments we selected all the text files from Canterbury and Large corpora of the Canterbury Corpus[2] [2]. We also selected different size collections of WSJ, ZIFF and AP from TREC-3[3] [11]. In this case we concatenated files so as to obtain approximately similar subcollection sizes from the three collections, so the size in MB is approximate.

In order to test the dictionary mapping itself, and in conjunction with the two word-based techniques described in Section 2, we implemented several prototypes for basic dictionary mapping on PPM (denoted by *mppm*), dictionary mapping with separate alphabets model (denoted by $mppm_{sa}$) and dictionary mapping with spaceless words model (denoted by $mppm_{sw}$). In all the versions we used the Shkarin/Cheney's *ppmdi* [18] to obtain comparable results with the compressors mentioned below.

First we compressed the text files from Canterbury and Large corpora of the Canterbury Corpus. Table 1 shows the compression (in bits per character) obtained with our prototypes. The two first columns show, respectively, the file denomination and its size in bytes, whereas the third column shows the best results reported on the Canterbury Corpus site[4]. Column "ppmz" shows the

[2] `http://corpus.canterbury.ac.nz/`
[3] `http://trec.nist.gov/`
[4] `http://corpus.canterbury.ac.nz/details/`

compression obtained by Bloom/Tarhio's *ppmz v.9.1* for Linux[5], one of the better PPM variations but with high resources demand. Column "ppmdi" shows the compression obtained by the character based Shkarin/Cheney's *ppmdi*[6] (extracted in turn from James Cheney's *xmlppm v.0.98.2*). This *ppmdi* version uses the same memory requirements as *ppmdi* used to codify codewords in the *mppm* prototypes. In order to codify new words in *mppm* prototypes, another *ppmdi* compressor is needed but, in this case, with minor memory requirements. Word-based BWT compression was excluded because we could not find the software, yet results reported in [16] indicate that the compression ratios achieved for Canterbury Corpus are slightly worse to those of $mppm_{sw}$. Although, in order to be able to compare, it is necessary to make more tests, mainly with files of great size.

Table 1. Compression (bpc) for each method and collection, for Canterbury and Large corpora of the Canterbury Corpus

File	Size(bytes)	Best	*ppmz*	*ppmdi*	*mppm*	$mppm_{sa}$	$mppm_{sw}$
LIST	3,721	2.40	**2.253**	2.281	2.736	2.764	2.668
MAN	4,227	2.98	2.865	**2.852**	3.418	3.397	3.283
CSRC	11,150	2.08	1.867	**1.849**	2.293	2.329	2.244
HTML	24,603	2.32	2.192	**2.134**	2.382	2.473	2.355
PLAY	125,179	2.49	2.335	**2.307**	2.411	2.488	2.371
TEXT	152,089	2.20	**2.081**	2.033	2.145	2.189	2.090
TECH	426,754	1.95	1.827	**1.794**	1.861	1.873	1.834
POEM	481,861	2.36	**2.216**	2.253	2.267	2.344	2.266
WORLD	2,473,400	1.40	**1.295**	1.436	1.391	1.426	1.346
BIBLE	4,047,392	1.53	1.473	1.516	1.464	1.547	**1.436**

We can observe that the *ppmdi* compressor is better than the *mppm* prototypes for small sizes but worse for greater files (over 1 Mb), this is due to the overload when vocabulary is coded. On the other hand, $mppm_{sw}$ is the best of the *mppm* family and it is also the best choice for medium and large files, even improving on *ppmz* by 2.5% (as it uses memory without limitation). Comparing $mppm_{sw}$ with the best results reported in the site, it improves the compression from all files greater than 100 Kb. Also, all the prototypes of the *mppm* family fulfill this affirmation. In this collection, $mppm_{sw}$ improves *ppmdi* compression by up to 7%, *mppm* compression by up to 3.5% and $mppm_{sa}$ compression by up to 8%. $mppm_{sa}$ was expected to be superior to basic *mppm* for all files since it takes advantage of the alternation property. But it does not happen in most files in Table 1. This surprising behavior can be due to the fact that the ppm used in the basic *mppm* is able to predict longer sequences (including both words and separators) and therefore, it uses less bits in their codification than the $mppm_{sa}$,

[5] `http://www.cs.hut.fi/~tarhio/ppmz/`

[6] `http://pizzachili.dcc.uchile.cl/initiative.html`

which is composes of two PPM encoders, one for modelling words and the other for separators.

Next, we compressed different size collections of WSJ, ZIFF and AP from TREC-3 in order to verify the behavior of the algorithms when managing medium and large collections. TREC-3 collections are formed by semistructured documents, this can harm *mppm* compressors but allows us to compress documents with structure-aware compressors that obtain better compression than classical compressors. Therefore, we compressed the collections with several classic compressor systems: (1)*GNU gzip v.1.3.5*[7], which use LZ77 plus a variant of the Huffman algorithm (we also tried *zip* with almost identical results but slower processing); (2)*bzip2 v.1.0.2*[8], which uses the Burrows-Wheeler block sorting text compression algorithm, plus Huffman coding (where maximum compression is the default); (3)*ppmdi* (extracted from *xmlppm v.0.98.2*), the same PPM compressor used in *mppm* family and with the same parameters. This time, *ppmz* has been excluded due to its high memory and time requirements. However, to be able to have an idea of the *ppmz* behavior with TREC-3 collections, we have compressed the smallest collections obtaining a compression of 1.936 bpc for AP, 1.917 bpc for WSJ and 1.661 bpc for ZIFF. This compression ratio is just slightly better than the obtained by $mppm_{sw}$, but $mppm_{sw}$ demands much less resources. For longer texts, *ppmz* is simply not a choice.

On the other hand, we compressed the collections with other compression systems that exploit text structure: (1)*xmill v.0.8*[9] [14], an XML-specific compressor designed to exchange and store XML documents. Its compression approach is not intended to directly support querying or updating of the compressed documents. *xmill* is based on the *zlib* library, which combines Lempel-Ziv compression with a variant of Huffman. Its main idea is to split the file into three components: elements and attributes, text, and structure. Each component is compressed separately. (2)*xmlppm v.0.98.2*[10] [8], a PPM-like coder, where the context is given by the path from the root to the tree node that contains the current text. This is an adaptive compressor that does not permit random access to individual documents. The idea is an evolution over *xmill*, as different compressors are used for each component, and the XML hierarchy information is used to improve compression. (3)*scmppm v.0.93.3*[11] [1], that implements SCM, a generic model used to compress semistructured documents, which takes advantage of the context information usually implicit in the structure of the text. The idea is to use a separate model to compress the text that lies inside each different structure type. Like *xmlppm*, *scmppm* uses Shkarin/Cheney's *ppmdi* [18] compressors.

Table 2 shows the compression obtained with our prototypes for TREC-3 collections. We can observe that $mppm_{sw}$ is the best choice for the *mppm* family, improving $mppm_{sa}$ by up to 4% and mppm basic by up to 4.5%. Let us focus on

[7] `http://www.gnu.org`
[8] `http://www.bzip.org`
[9] `http://sourceforge.net/projects/xmill`
[10] `http://sourceforge.net/projects/xmlppm`
[11] `http://www.infor.uva.es/~jadiego`

Table 2. Compression (bpc) for each dictionary mapping prototype for each TREC-3 collection

	TREC3-AP			TREC3-WSJ			TREC3-ZIFF		
Mb	*mppm*	$mppm_{sa}$	$mppm_{sw}$	*mppm*	$mppm_{sa}$	$mppm_{sw}$	*mppm*	$mppm_{sa}$	$mppm_{sw}$
1	2.035	2.030	1.955	2.002	2.022	1.932	1.692	1.756	1.652
5	1.918	1.888	1.848	1.914	1.910	1.857	1.729	1.772	1.691
10	1.896	1.856	1.823	1.901	1.879	1.832	1.748	1.782	1.708
20	1.878	1.827	1.801	1.888	1.860	1.820	1.752	1.782	1.710
40	1.874	1.818	1.796	1.886	1.854	1.814	1.749	1.776	1.706
60	1.876	1.817	1.795	1.887	1.852	1.814	1.745	1.771	1.701
100	1.879	1.819	1.797	1.879	1.840	1.801	1.750	1.772	1.706

Table 3. Compression (bpc) for classical compressors for each TREC-3 collection

	TREC3-AP			TREC3-WSJ			TREC3-ZIFF		
Mb	*gzip*	*bzip2*	*ppmdi*	*gzip*	*bzip2*	*ppmdi*	*gzip*	*bzip2*	*ppmdi*
1	3.010	2.264	2.114	2.965	2.195	2.048	2.488	1.863	1.686
5	3.006	2.193	2.057	2.970	2.148	2.034	2.604	1.965	1.803
10	2.984	2.175	2.047	2.970	2.154	2.033	2.640	2.000	1.837
20	2.970	2.168	2.041	2.973	2.153	2.035	2.647	2.012	1.850
40	2.978	2.172	2.045	2.977	2.158	2.040	2.649	2.013	1.851
60	2.983	2.174	2.046	2.983	2.160	2.043	2.648	2.010	1.849
100	2.987	2.178	2.050	2.979	2.148	2.032	2.654	2.016	1.853

Table 4. Compression (bpc) for structure-aware methods for each TREC-3 collection

	TREC3-AP			TREC3-WSJ			TREC3-ZIFF		
Mb	*xmill*	*xmlppm*	*scmppm*	*xmill*	*xmlppm*	*scmppm*	*xmill*	*xmlppm*	*scmppm*
1	2.944	2.110	2.083	2.898	2.044	2.030	2.489	1.682	1.743
5	2.910	2.052	2.000	2.878	2.029	1.984	2.596	1.799	1.782
10	2.893	2.040	1.977	2.881	2.028	1.972	2.634	1.834	1.803
20	2.877	2.036	1.963	2.882	2.030	1.971	2.640	1.846	1.812
40	2.883	2.040	1.964	2.888	2.035	1.974	2.639	1.847	1.808
60	2.888	2.044	1.964	2.891	2.038	1.975	2.635	1.846	1.803
100	2.891	2.048	1.968	2.872	2.027	1.958	2.640	1.849	1.807

the $mppm_{sw}$ prototype in order to compare it with other systems. Compression for standard systems is shown in Table 3. The *gzip* obtained the worst compression ratios, not competitive in this experiment. It is followed by *bzip2* with the best compression as default and a great difference between it and *gzip*. The best standard compressor is *ppmdi*, the base for the *mppm* family, and with compression ratios near to *bzip2*. Our $mppm_{sw}$ prototype compressed significantly better than standard compressors. It improves *gzip* by up to 66%, *bzip* by up to 21% and *ppmdi* by up to 14%. Finally, in Table 4, we can see the compression obtained with structure-aware compressors for the same collections. *xmill* obtains

an average compression roughly constant in all cases because it uses *zlib* as its main compression machinery, like *gzip*, its compression is not competitive in this experiment. On the other hand, *xmlppm* and *scmppm* obtain a good compression both surpassing standard compressors. However, in this case, our $mppm_{sw}$ prototype also still obtains the best compression, reaching an improvement on *xmill* of up to 54%, on *xmlppm* of up to 13.5% and on *scmppm* of up to 9.5%. In addition, $mppm_{sw}$ uses less memory than *xmlppm* and *scmppm*. In view of these results, we can conclude that $mppm_{sw}$ is an excellent alternative to compress natural language documents. A graphical representation of average compression is shown in Figure 2. In this graph we can observe that all the prototypes based on dictionary mapping are better (over 1 Mb in size) than all compressor systems against which they have been compared.

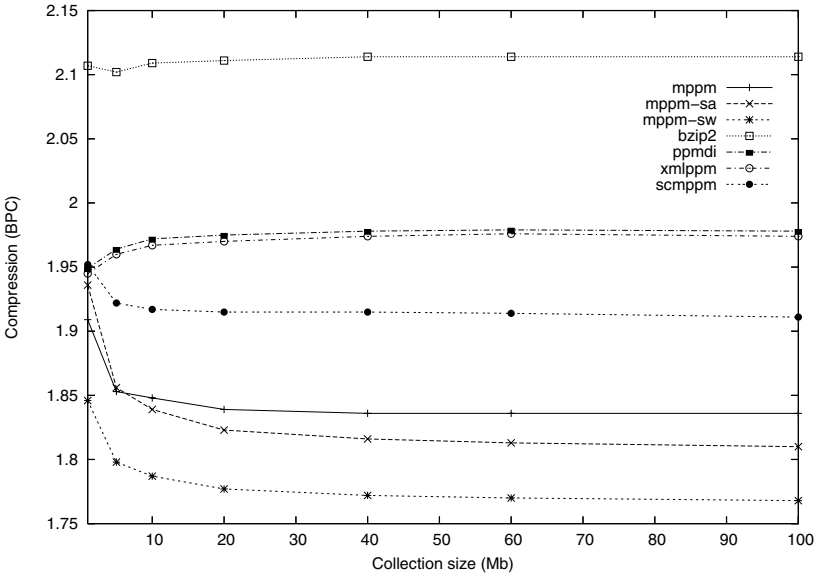


Fig. 2. Average compression for each TREC-3 collection size

The increase in space of the *mppm* prototypes with respect to *ppmdi* varies from 40% for $mppm_{sw}$ to 114% for $mppm_{sa}$. This increase is due to the necessity to store the vocabulary (dictionary) and to have an additional model to codify the new words. The $mppm_{sw}$ prototype is about 15% faster than *mppm* basic and both use approximately the same amount of memory. Besides for files up to 1 Mb, $mppm_{sw}$ is about 5000% faster than *ppmz* and uses 92% less memory than *ppmz*. That is why $mppm_{sw}$ is a very efficient alternative to *ppmz* for medium and large text files. On the other hand, $mppm_{sw}$ uses about 40% more memory and is also 40% slower than *ppmdi*. This increase in time is due to the time needed to locate a word in the used data structure (in this case a balanced search tree) this

being $O(\log_2 n)$. It is possible to turn it into $O(1)$ if a hash table is used (where size can be estimated previously by using Heaps law[12]). The *ppmdi* obtains a constant average memory usage in all cases because it does not have to store the vocabulary.

6 Conclusions and Future Work

We have proposed a new, simple and efficient general scheme for compressing natural language text documents by extending the PPM to allow easy word handling using an additional layer. When file size grows, our proposal improves compression up to 14% with respect to the character based PPM. Our proposal uses just a little bit more memory and is a little slower, but these drawbacks are clearly compensated for by the gain in compression.

We have shown that the idea significantly improves compression and we have compared our prototype with standard and specific compressor systems, showing that our prototypes obtain the best compression for files over 1 Mb, improving the compression when file size grows. In addition, $mppm_{sw}$ is an interesting alternative for *ppmz* for natural language text files.

In this paper we have considered the compression of natural language text documents, and we will have to investigate the possibility of applying word mapping to binary files. We will have to generalize the mapping algorithm and we will have to avoid the generation of dynamic codes, as they prevent PPM from making good predictions. On the other hand, current mppm prototypes are a basic implementation and we are working on several improvements, which will make them even more competitive in terms of time and space.

References

1. J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Proceedings of 14th Data Compression Conference (DCC 2004)*, page 522, 2004.
2. R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of 7th Data Compression Conference (DCC 1997)*, pages 201–210, 1997.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley-Longman, may 1999.
4. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, N.J., 1990.
5. T. C. Bell, A. Moffat, C. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44:508–531, 1993.
6. J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.
7. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

8. J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proceedings of 11th Data Compression Conference (DCC 2001)*, pages 163–172, 2001.
9. J. G. Clearly and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.
10. J. Dvorský, J. Pokorný, and V. Snásel. Word-based compression methods and indexing for text retrieval systems. In *Proc. ADBIS'99*, LNCS 1691, pages 75–84. Springer, 1999.
11. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.
12. H. S. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, 1978.
13. R. N. Horspool and G. V. Cormack. Constructing word-based text compression algorithms. In *Proceedings of 2nd Data Compression Conference (DCC 1992)*, pages 62–71, 1992.
14. H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proc. ACM SIGMOD 2000*, pages 153–164, 2000.
15. A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.
16. A. Moffat and R. Yugo Kartono Isal. Word-based text compression using the Burrows–Wheeler transform. *Information Processing & Management*, 41(5):1175–1192, 2005.
17. E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proceedings of the Fourth South American Workshop on String Processing*, pages 95–111, 1997.
18. D. Shkarin. PPM: One step to practicality. In *Proceedings of 12th Data Compression Conference (DCC 2002)*, pages 202–211, 2002.
19. P. Skibinski, Sz. Grabowski, and S. Deorowicz. Revisiting dictionary-based compression. *Software–Practice and Experience*, 35(15):1455–1476, 2005.
20. G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison–Wesley, 1949.
21. N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.

Improving Usability Through Password-Corrective Hashing

Andrew Mehler and Steven Skiena

Dept. of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794
{mehler, skiena}@cs.sunysb.edu

Abstract. We propose a way to increase the usability of password authentication systems by compensating for transposition and substitution errors. We show how to correct for these errors with low false positive rates (i.e., low probability that an arbitrary string will be accepted as the password for authentication). Thus our techniques increase usability with provably little loss of security.

In particular, we propose applying a single *password-corrective* hash function to each entered password attempt. The key property of the hash function is that two strings differing by a single data entry error be likely to be hashed to the same key, while more substantially differing strings are hashed to different keys.

We develop precise analytical formulae for the precision/recall tradeoffs for a variety of corrective hash functions. We evaluate these methods at parameter values reflecting common classes of keys/passwords. Finally, we evaluate these schemes using a popular crack-list (dictionary) of 680,000 common words. We show that we can correct for *all* user transposition errors while reducing the computational cost of a crack attack by only 13%.

1 Introduction

The design of any password authentication system requires a tradeoff between security and usability. For example, mandating longer passwords in a system improves security while complicating the user's ability to remember passwords and enter them correctly.

The data entry problem is by no means trivial. Empirical studies of typing accuracy [1,2,3] suggest that typists make data-entry errors roughly once every 30 keystrokes on typical English text. Assuming ten-character passwords, this implies that roughly one out of every three login attempts by legitimate users fail due to data entry errors. Indeed, typing error rates are presumably even higher on the cryptic, case-sensitive, punctuation-intensive strings recommended for high-security passwords. An inspiration for this paper was the painful memory of repeatedly typing a 128-bit wireless encryption WEP key (consisting of 26 hexadecimal-characters) until achieving the required perfect fidelity. Finally,

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 193–204, 2006.

users juggling passwords for several different systems can easily confuse typing errors with recalling the wrong password [4]. Subsequent cycling through passwords on other systems may may result in users getting locked out, with a subsequent need for a password reset.

In this paper, we propose a way to increase the usability of password authentication systems by correcting for two common classes of data entry errors, namely transposition and substitution errors. Transpositions and substitutions can arise from physical input errors or from partial password recall. We show how to identify and correct for these errors with low false positive rates (i.e., low probability that an arbitrary string will be accepted as the password for authentication). Thus our techniques increase usability with provably little loss of security. Indeed, they may arguably even *increase* security in practice, because users benefiting from our correcting schemes will be more inclined to choose strong passwords, and not resort to insecure practices such as writing down a password.

Some naive approaches to this problem suggest themselves. The first would involve explicitly comparing an entered string to the password on file to check for equivalence modulo single transpositions or substitutions. However, this requires that the password file be stored as plain-text instead of being encrypted, which is clearly a bad idea for security. The second approach involves automatic repeated login attempts using explicit transformations of the entered string. Indeed, SAMBA appears to employ such a method to relax sensitivity to password case and character order [5]. However, such methods quickly get expensive, as there are $n - 1$ possible transpositions and nm possible substitutions on an n-character password defined on an alphabet of size m. Finally, one might generate all variants of a password, and then store these encrypted. A login would then check each possibility. Not only would this increase the size of the password file, but it may also make malicious decrypting easier if it is known that a set of encrypted keys differ by only a transposition.

Instead, we propose a simple technique of applying a single *password-corrective* hash function to each entered password attempt. That is, this hash function is applied to the entered password, and the resulting key is then encrypted and stored. The important property required of the hash function is that two strings differing by a single data entry error (i.e. one transposition or substitution) be likely to be hashed to the same key, while more substantially differing strings are hashed to different keys.

In this paper, we study the efficiency of a variety of hash functions in correcting single transposition and substitution errors. We rigorously analyze the recall / false positive rate tradeoffs for each class of hash functions to determine the most appropriate choice for common password applications. In particular:

- We develop precise analytical formulae for the precision/recall tradeoffs for correcting transposition errors using sorting-network and block-sorting hash functions. These functions contain parameters trading off security for usability; tradeoffs which are made explicit through our analysis.

- We do the same for two classes of alphabet-weakening hash functions, which correct for substitution errors. These alphabet-weakening schemes can be composed with the permutation-based functions described above, yielding a function which can simultaneously correct for transposition and substitution errors.
- We prove the curious property that the limiting case of both of our permutation-based methods (character sorting) has the highest precision among all perfect-recall methods for correcting transposition errors.
- The explicit precision / recall performance of these methods is very sensitive to the length and alphabet size of the associated keys. Therefore, we evaluate these tradeoffs at parameter values reflecting common classes of keys/passwords (including system passwords, social security numbers, WEP passwords, and credit card numbers) to identify the most desirable hash functions and parameters for each.
- Finally, we evaluate these schemes using a popular crack-list (dictionary) of 680,000 common words. We show that we can correct for *all* user transposition errors while reducing the computational cost of a crack attack by only 13%.

This paper is organized as follows. Previous work on password system usability and corrective hashing techniques is reviewed in Section 1.1. We introduce the notion of password-corrective hashing in Section 2. The next two sections present our analysis of hashing techniques against transposition and substitution errors, respectively. Finally, Section 5 details our experiments using standard crack-lists.

1.1 Previous Work

The importance of user interaction in password authentication is well known. Basic facts about human memory are in conflict with most password policies. Sasse and Adams [6] stress human factors in developing security. Sasse, Brostoff and Weirich [4] note usability problems with password authentication, such as the number of passwords a user must remember, strict password policies, varying systems, and memory demands. Their studies found that users rarely completely forget a password. Instead, users often partially recall a password or recall the wrong password (typically from another system the user is enrolled in). They note that the user cannot know which of these two reasons apply after a failed authentication attempt.

There have been many human factor studies of data entry methods. Grudin [1] investigated error rates by typists, discovering that novice typists (20 wpm) had per-character error rates of about 3.2%, while experts (60-90 wpm) had error rates of $0.4\% - 1.9\%$. Mackenzie [2] sought to partition these errors by type, identifying per-character substitution error rates of 0.962%, insertion error rates of 0.218% and deletion rates of 1.045%. Peterson [3] found that transpositions represented between 2.6% and 13.1% of all data-entry errors, while substitutions accounted for between 26.9% and 40.0% of all errors.

There has been some previous work in developing password recovery schemes that tolerates errors. Frykholm and Juels [7] require users to supplies answers to personal questions for authentication, but the answers are not required to be entirely correct. Spector and Ginzberg [8] propose a pass-phrase scheme that matches phrase semantics, and is flexible on syntax and actual words used.

Cranor and Garfinkel [9] suggest system require more than 10^{12} different potential passwords effective security. While most systems in theory allow this many, users restricting themselves to dictionary words use only about 10^6 different password keys. Our methods do reduce the theoretical number of potential passwords for a system; however for added security, password length can be made longer. The convenience offered by our system should make longer passwords less of a burden.

2 Password Correction Hashing

In general, it is logical to assume that a minor difference between an entered password and the version on file still represents an authorized attempt to access an account. We will evaluate different hashing schemes to withstand substitution and transposition errors. These schemes transform an input string into a generalized representation; typically similar size to the original string. We will evaluate how these generalized representations correctly fix transposition and substitution errors (recall) vs. how often they induce random strings to collide (false positive rate).

2.1 Preliminaries and Notation

A transposition error is one in which two consecutive characters of a password are switched. If the password is $c_1c_2 \ldots c_n$, then a transposition is $c_1c_2 \ldots c_{i+1}c_i \ldots c_n$. A substitution is when any single character is replaced by another. Thus for any $b \in \Sigma$, $c_1c_2 \ldots c_{i-1}bc_{i+1} \ldots c_n$.

In dealing with the password correcting systems, it is necessary to distinguish the types of errors the system makes. A system that makes the error of not allowing authorized access is preferable to one that allows unauthorized access.

We denote a pair of different strings which are considered equivalent to be *real positive*. For equivalence under transposition, the pair "12345" and "12435" represent a real positive. Similarly, "12345" and "67890" are a real negative, since they are not equivalent. The *true positives* are the real positives that the hashing scheme correctly hashes to the same representative string. The false positives are the real negatives that the scheme incorrectly hashes to the same representative string. The definition is symmetric for true negatives and false positives.

We have the following relationship

$$\text{true positives} + \text{false negatives} = \text{real positives} \tag{1}$$

$$\text{true negatives} + \text{false positives} = \text{real negatives} \tag{2}$$

We will survey hash schemes on strings of length n over an alphabet Σ of size m. *Recall* is defined as

$$\text{recall} = \text{true positives}/(\text{real positives})$$

that is it is the fraction of positives the scheme correctly identifies as positive. Higher recall means easier access to the system, whereas lower recall is less flexible on the errors in the password. The *False Positive Rate* is defined as,

$$\text{False positive rate} = \text{false positives}/(\text{real negatives})$$

i.e. the frequency an unauthorized access (negative) is incorrectly called a positive. The lower this value, the more secure a system is.

3 Correcting Transpositions

In analyzing transposition errors, we note that the number of different positives and negatives depends on n and m. Let $P_{trans}[n, m]$ be the number of transposition positives, and $N_{trans}[n, m]$ be the number of negatives. The positives are counted as follows. Choose among the $n - 1$ possible spots for a transposition. Then choose among the m characters for the $n-2$ spots not in the transposition. Finally we must choose from the m characters, 2 different characters that are in the transposition spot. We must choose different characters, since choosing the same character results in a transposition that gives back the original string. Thus

$$P_{trans}[n, m] = (n-1)m^{n-2}\binom{m}{2}$$

Since there are m^n different strings,

$$P_{trans}[n, m] + N_{trans}[n, m] = \binom{m^n}{2}$$

and thus

$$\begin{aligned} N_{trans}[n, m] &= \binom{m^n}{2} - P_{trans}[n, m] \\ &= (m^n/2)((m^n - 1) - (n-1)\frac{m-1}{m}) \end{aligned}$$

3.1 Character Sorting

Sorting is a natural choice for trying to eliminate transposition errors, since sorting will tend to impose its own order on a string. Sorting the input sequence renders the original order inconsequential, so character distribution is the only

distinguishing feature of a sequence. Thus all transpositions will be caught and hence

$$\text{recall}_{sort} = 1$$

To count the false positives associated with character sorting, we first count the true positives plus false positives. Any pair of strings with the same character distribution will be hashed together to a true or false positive. So we count pairs of strings with the same character distribution:

$$\text{tp}_{sort} + \text{fp}_{sort} = \sum_{k_i \geq 0} \binom{\binom{n}{k_1 \ldots k_m}}{2} = \frac{1}{2}\left(\sum_{k_i \geq 0} \binom{n}{k_1 \ldots k_m}^2 - m^n\right) \tag{3}$$

Since we know the recall and total positives (from the previous section), we can use the formula for recall to solve for true positives. We then subtract this from the above result to get false positives, and divide by negatives to get

$$\text{fp-rate}_{sort} = \frac{X/m^n - 1 - n(m-1)}{m^n - 1 - n(m-1)}; X = \sum_{k_i \geq 0} \binom{n}{k_1 \ldots k_m}^2 \tag{4}$$

In fact, character sorting offers the highest precision way of correcting all single transposition errors:

Theorem 1. *Character sorting has perfect recall for single transpositions, and has the lowest false positives of any method that does so.*

Proof. Character sorting must have perfect recall, since any two strings differing by a single transposition must have the same character set. Now consider another method M which also has perfect recall but fewer false positives. There must be two strings S and T that are a false positive under character sorting, but not in the new method. S and T are hashed together under character sorting, so they have the same character set. Thus there is a sequence of strings $S, s_1, s_2, \ldots s_j, T$ where each consecutive strings differ by a single transposition. Since S and T are not hashed together under M, there must exist consecutive strings s_i, s_{i+1} in the sequence that are not hashed together under M. Since s_i and s_{i+1} differ by a single transposition, this contradicts the assumption that M had perfect recall. Therefore character sorting has the best performance of any perfect recall method.

3.2 Even-Odd Transposition Sorting Networks: Single Stage

We now consider weakening (hashing) a string by sending it through k stages of an even-odd sorting network [10]. A sorting network is a computation graph. In an odd/even sorting network, at each stage adjacent entries can be swapped or left alone. Even pairs may be swapped in the even stages, and odd pairs in the

odd stages. We assume the first stage is an even stage. The following example illustrates a string as it is transformed through each stage of the network:

$$14572463 \rightarrow 14572436 \rightarrow 14527346 \rightarrow 14253746 \rightarrow \cdots \rightarrow 12344567$$

We first consider the case of a single stage sorting network. After the first stage of an odd-even network, all even transpositions will be corrected, but odd transpositions will remain, so

$$\text{recall}_{1-stage} = \frac{\text{even transpositions}}{\text{total transpositions}} = \frac{\lfloor n/2 \rfloor}{n-1} \approx 1/2$$

To calculate the false positives (fp) and the fp-rate, we first calculate the sum of false positives and true positives and then subtract the true positives (tp = recall $*$ positives). To determine tp + fp, we consider a string with k possible even transposition locations (i.e. $n-k$ characters are repeats, so no real transposition is possible). There are $\binom{\lfloor n/2 \rfloor}{k}$ ways to choose the k transposition locations; $m^{\lfloor n/2 \rfloor - k}$ ways to choose the characters for the repeated character transposition locations; $\binom{m}{2}^k$ ways to choose the characters for the k transposition locations; and finally 2^k ways to order the characters involved in the transposition locations. Each of these 2^k strings differs only in even transpositions, so all will get hashed together giving $\binom{2^k}{2}$ colliding pairs. Summing over k gives

$$\begin{aligned} \text{tp}_{1-stage} + \text{fp}_{1-stage} &= \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{\lfloor n/2 \rfloor}{k} \binom{m}{2}^k m^{\lfloor n/2 \rfloor - k} \binom{2^k}{2} \\ &= \frac{1}{2} m^{\lceil n/2 \rceil} ((2m-1)^{\lfloor n/2 \rfloor} - m^{\lfloor n/2 \rfloor}) \end{aligned}$$

Then we solve for false positives.

$$\begin{aligned} \text{fp}_{1-stage} &= (\text{tp}_{1-stage} + \text{fp}_{1-stage}) - \text{tp}_{1-stage} \\ &= \frac{1}{2} m^{\lceil n/2 \rceil} ((2m-1)^{\lfloor n/2 \rfloor} - m^{\lfloor n/2 \rfloor} - \lfloor n/2 \rfloor (\frac{m-1}{m}) m^{\lfloor n/2 \rfloor} \end{aligned}$$

Finally,

$$\begin{aligned} \text{fp-rate}_{1-stage} = \text{fp}_{1-stage}/\text{N[n,m]} &= \frac{(2-1/m)^{\lfloor n/2 \rfloor} - 1 - \lfloor n/2 \rfloor (\frac{m-1}{m})}{m^n - 1 - (n-1)(\frac{m-1}{m})} \\ &\approx m^{-n}(2^{\lfloor n/2 \rfloor} - 1 - n) \end{aligned}$$

3.3 2-Stage Sorting Networks

By adding an extra stage of sorting some odd transposition errors will now be caught, depending on whether the first stage moved the characters involved in the

transposition. Consider the string fragment *abcd*; The odd transposition (*acbd*) will be corrected in the second step when $a \leq b, c \leq d$. This gives $\binom{m+2}{4}$ corrected transposition errors from $\frac{1}{2}m^3(m-1)$ possible transposition errors. Odd length strings have an extra odd transposition, not surrounded by 4 characters, only 3, as in *abc*. In this case, there are $\binom{m+1}{3}$ transpositions that get corrected from $\frac{1}{2}m^2(m-1)$ possible errors.

$$\begin{aligned}\text{recall}_{2stage} &= (\lfloor n/2 \rfloor + \frac{(\lceil n/2 \rceil - 1) * \binom{m+2}{4}}{(\frac{1}{2}m^3(m-1))} + [nodd]\frac{\binom{m+1}{3}}{(\frac{1}{2}m^2(m-1))}) \times (\frac{1}{n-1}) \\ &\approx (1/12) + (11/12)(\frac{\lfloor n/2 \rfloor}{n-1})\end{aligned}$$

where $[nodd]$ evaluates to 1 if n is odd, and 0 otherwise. We do not have analytical results for the false positive rate of 2-stage sorting, so we instead ran simulations to get results. See Section 3.5 for these results.

3.4 Block Sorting Methods

With block sorting, we divide the string into fixed-size blocks, and completely sort each block. The following example illustrates the transformation for blocks of size 4:

$$1738|5901|9874|3509|1237 \rightarrow 1378|0159|4789|0359|1237$$

The only transposition errors not matched by such a scheme are those that occur across block boundaries.

We first consider the case of 2 blocks. The string is broken up into a block of size n_1 and one of size $n - n_1$, typically $n_1 = n/2$. Only a transposition that crosses over the block boundary will not be caught, so

$$\text{recall}_{2-block} = (n-2)/(n-1)$$

regardless of the block sizes. Now consider true positives. Consider the true positives that result from a single transposition spot. We can choose among all m characters for every digit, except the transposition digits must be different. Thus the $m^{n-1}(m-1)$ term. There are $(n-2)$ transposition spots (since we cant match across the blocks). Finally, each match is counted twice, so we divide by 2.

$$\text{tp}_{2-block} = (1/2)m^{n-1}(m-1)(n-2)$$

Let $\text{fptp}_{\text{CS}}(k)$ be the true positives plus false positives for complete sorting a string of length k. We get the fp-rate by using the results from complete sorting. Since within a block, the contents are completely sorted, we have

$$\text{fp}_{2-block} = \text{fptp}_{\text{CS}}(n_1) \times \text{fptp}_{\text{CS}}(n - n_1)$$

We can generalize for an arbitrary number of k blocks. The only transpositions not found are still ones occurring across block boundaries, so

$$\text{recall}_{k-blocks} = (n-k)/(n-1)$$

We again get the fp-rate by using results from complete sorting.

$$\mathrm{fp}_{k-blocks} = \prod_{j=0}^{k} \mathrm{fptp}_{\mathrm{CS}}(n_j) \tag{5}$$

3.5 Evaluation

In this section, we evaluate these transposition correction methods on a variety of alphabet sizes and string length pairs corresponding to important classes of passwords/keys. In particular, we consider:

- *System Passwords* – Typical online account passwords. We consider three cases: the full English alphabet with case, digits, underscore and period ($m = 64$), a smaller case-independent alphabet of size 32, and binary passwords on typical lengths.
- *WEP Keys* – Wireless encryption (WEP) keys. We consider hexadecimal WEP keys for 64 and 128-bit WEP ($n = 10, 26$, $m = 16$).
- *Social Security Numbers* – Nine digit identification numbers, ($n = 9$, $m = 10$).
- *Credit Cards* – Credit cards numbers comprise 16 digit numbers, so ($n = 16$, $m = 10$).
- *Proper Names* – The first/last names of people average about seven characters on the case-insensitive English alphabet, so ($n = 7$, $m = 26$).

Table 1 compares the performance of k-stage sorting networks, and k block sorting for correcting transposition errors. We see for most schemes, good recall is achieved at reasonably low false positive rates. High recall and low false positive rate will ensure that the added convenience of a system does not come with a loss of security. *The 2-Block scheme offers the best balance between high recall and low false positive rates, and is recommended.* It should be noted that these schemes do become risky on small alphabets, as the row for $m = 2$ indicates. Fortunately secure systems use large alphabet sizes, so this will not be a problem.

4 Correcting Substitution Errors

Another common class of entry errors is substitution errors, where one character gets replaced by another character. We now consider two classes of hash functions that weaken the alphabet by making distinct characters the same. Such schemes can overcome substitution errors, i.e. two strings should be hashed together if they differ by only a single substitution. For substitutions, we have

$$P[n,m]_{subs} = \frac{1}{2} m^n n(m-1)$$

$$N[n,m]_{subs} = \binom{m^n}{2} - P[n,m]_{subs} = \frac{1}{2} m^n (m^n - 1 - n(m-1))$$

Table 1. Recall and False Positive Rate for correcting transposition errors for common password/key lengths

		Algorithm									
Application		1-Stage		2-Stage		Complete		2-Block		3-Block	
n	m	Rec	fp-rate	Rec	fp-rate	Rec	fp-rate	Rec	fp-rate	Rec	fp-rate
Passwords											
8	64	0.571	3.75e-14	0.609	3.92e-13	1	2.17e-10	0.857	6.37e-13	0.714	2.07e-14
10	32	0.556	2.11e-14	0.596	3.64e-13	1	8.20e-11	0.889	4.23e-12	0.778	1.03e-13
20	2	0.526	4.93e-5	0.645	0.0110	1	0.125	0.947	0.0146	0.894	0.002135
WEP Key											
10	16	0.556	1.97e-11	0.600	3.30e-10	1	7.67e-7	0.889	3.08e-9	0.778	8.48e-11
26	16	0.520	2.67e-28	0.568	3.57e-28	1	2.39e-5	0.960	5.96e-15	0.920	2.58e-18
SSNs											
9	10	0.500	8.43e-10	0.587	1.60e-7	1	5.48e-5	0.875	5.93e-7	0.750	1.77e-8
Credit Cards											
16	10	0.533	1.62e-14	0.585	1.45e-12	1	4.12e-6	0.933	4.30e-9	0.867	4.06e-11
Names											
7	26	0.500	1.75e-11	0.598	5.96e-9	1	4.15e-7	0.833	6.43e-9	0.667	1.34e-10

4.1 High-Low Weakening

In this scheme, we partition characters in the alphabet Σ as being either high or low. This reduces the input key to a binary string. For example, considering the digits $0-4$ as low ('l') and $5-9$ as high ('h') transforms:

$$17385901987435091237 \rightarrow lhlhhhllhhhllhlhllllh$$

A substitution error is found whenever the substituted characters map to the same symbol. Let k be the size of the low set (and thus $m-k$ the size of the high set). The true positives follows since there are n character positions to perform a substitution, the other $n-1$ characters can be anything.

$$\mathrm{tp}_{high-low} = nm^{n-1}(\binom{k}{2} + \binom{m-k}{2})$$

We divide this by the number of positives to get

$$\mathrm{recall}_{high-low} = \frac{k(k-1)+(m-k)(m-k-1)}{m(m-1)}$$

To determine the false positives, we first calculate $\mathrm{tp}+\mathrm{fp}$. We sum over j where j is the number of characters in the string belonging to the low set.

$$\begin{aligned}\mathrm{tp}_{high-low} + \mathrm{fp}_{high-low} &= \sum_{j=0}^{n} \binom{k^j(m-k)^{n-j}}{2} \times \binom{n}{j} \\ &= \frac{1}{2}((2k^2+m^2-2mk)^n - m^n)\end{aligned}$$

We then subtract the true positives and divide by negatives to get the false positive rate.

4.2 Weak Set Methods

In this scheme, a set of k 'weak' characters get replaced by a single character, while the other characters remain the same. For example, defining the weak set as consisting of all non-alphabetic characters and replacing them with the weak symbol ('.') yields the transformation

$$L1saS!mps0n \rightarrow L.saS.mps.n$$

This leaves an alphabet of size $m - k + 1$. Only substitutions among these k characters are found, so

$$\text{recall}_{weak-set} = \frac{k(k-1)}{m(m-1)}$$

We get the false positives by first calculating $fp + tp$.

$$\text{tp}_{weak-set} + \text{fp}_{weak-set} = \sum_{j=0}^{n} \binom{k^j}{2} (m-k)^{n-j} \binom{n}{j} = \frac{1}{2}((m-k+k^2)^n - m^n$$

Solving for fp-rate gets

$$\text{fp-rate}_{weak-set} = \frac{(1-k/m+k^2/m)^n - 1 - n/m(k)(k-1)}{m^n - 1 - n(m-1)} \approx (\frac{1}{m} - \frac{k}{m^2} + \frac{k^2}{m^2})^n$$

4.3 Evaluation

Our experimental results for correcting single substitution errors have been omitted due to lack of space, but they show a clear recall / false positive rate tradeoff; and the false positive rates are more problematic than we obtained for transposition error correction in Table 1. Details appear in the full paper.

5 Resistance to Crack-List Attacks

Users usually choose passwords from a much smaller key space than that offered by the system. One failing of our analytical results is that we assumed a uniform distribution of passwords over the space of possible keys. Also, we assumed that all keys are the same length, which is not true in many domains.

To get a more complete sense of the performance of correction schemes, we tested on them a crack list of dictionary words and common names. We used the lists from ftp://ftp.cerias.purdue.edu/pub/dict/dictionaries, which includes dictionaries in English, German, Italian, Swedish, Norwegian, and Dutch; as well as lists of common names, organizations, abbreviations, popular movie and TV names, common slang, Internet words, famous people, and a few other popular terms that appear in passwords. Combined, these lists had 680,000 unique terms.

Table 2 shows the ability of block sorting and sorting networks to correct transpositions on the crack lists. We see that in the case of a complete sort, the

Table 2. Performance of Transposition Correcting Methods on Dictionary Data

Block Sorting				Sorting Network			
Blocks	Recall	Unique Codes	False Pos. Rate	Stages	Recall	Unique Codes	False Pos. Rate
1	1	593347	1.42e-06	Inf	1	593347	1.42e-06
2	0.89	656475	4.79e-07	9	0.93	596026	1.41e-06
3	0.79	670265	3.16e-07	8	0.89	600209	1.39e-06
4	0.68	676146	2.68e-07	7	0.85	607036	1.33e-06
5	0.57	678491	2.51e-07	6	0.80	618944	1.21e-06
6	0.47	679395	2.42e-07	5	0.75	632101	1.03e-06
7	0.38	679737	2.39e-07	4	0.70	648115	7.61e-07
8	0.30	679873	2.38e-07	3	0.66	658338	5.74e-07
9	0.24	679932	2.37e-07	2	0.60	668050	3.80e-07
10	0.18	679972	2.36e-07	1	0.55	672544	3.17e-07
Inf	0	680000	0	0	0	680000	0

number of unique keys is now only 593,347. That is, a cracker whose initial crack-list of 680,000 words could now get by with a list of 593,347 words; this is about 13% shorter. This is not much to pay for eliminating all transposition errors. For more extreme security, 5-block sorting still has over 50% recall, yet allows a reduction of only 1509 names off the crack list, or 0.22%. The performance of sorting networks is not quite as good, though still reasonably effective.

References

1. Grudin, J.: Non-hierarchic specification of components in transcription typewriting. Acta Psychologica **54** (1983) 249–262
2. MacKenzie, I., Soukoreff., R.: A character-level error analysis technique for evaluating text entry methods. proceedings of the second nordic conference on human-computer interaction. Nordic Conference on Human-Computer Interaction (2002) 241–244
3. Peterson, J.L.: A note on undetected typing errors. Communications of the ACM **29**(7) (July 1986)
4. Sasse, M.A., Brostoff, S., Weirich, D.: Transforming the weakest link ? a human/computer interaction approach to usable and effective security. British Telecom Technology Journal **19**(3) (2001) 122–131
5. T's, J., Eckstein, R., Collier-Brown, D.: Using Samba. O'Reilly Media (2003)
6. Adams, A., Sasse, M.: Users are not the enemy. Commun. ACM **42**(12) (1999) 40–46
7. Frykholm, N., Juels, A.: Error-tolerant password recovery. In: CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security, New York, NY, USA, ACM Press (2001) 1–9
8. Spector, Y., Ginzberg, J.: Pass-sentence? a new approach to computer code. Comput. Secur. **13**(2) (1994) 145–160
9. Cranor, L.F., Garfinkel, S.: Security and Usability. O'Reilly, Sebastopol, CA (2005)
10. Knuth, D.E.: The Art of Computer Programming. Volume 3. Addison-Wesley Publishing Company, Reading, MA (1973)

Word-Based Correction for Retrieval of Arabic OCR Degraded Documents

Walid Magdy and Kareem Darwish

IBM Technology Development Center
P.O. Box 166 El-Ahram, Giza, Egypt
{wmagdy, darwishk}@eg.ibm.com

Abstract. Arabic documents that are available only in print continue to be ubiquitous and they can be scanned and subsequently OCR'ed to ease their retrieval. This paper explores the effect of word-based OCR correction on the effectiveness of retrieving Arabic OCR documents using different index terms. The OCR correction uses an improved character segment based noisy channel model and is tested on real and synthetic OCR degradation. Results show that the effect of OCR correction depends on the length of the index term used and that indexing using short n-grams is perhaps superior to word-based error correction. The results are potentially applicable to other languages.

Keywords: OCR, Retrieval, and Error Correction.

1 Introduction

Since the advent of the printing press in the fifteenth century, the amount of printed text has grown overwhelmingly. Although a great deal of text is now generated in electronic character-coded formats (HTML, word processor files ... etc.), many documents available only in print remain important. This is due in part to the existence of large collections of legacy documents available only in print, and in part because printed text remains an important distribution channel that can effectively deliver information without the technical infrastructure that is required to deliver character-coded text. These factors are particularly important for Arabic, which is widely used in places where the installed computer infrastructure is often quite limited. Printed documents can be browsed and indexed for retrieval relatively easily in limited quantities, but effective access to the contents of large collections requires some form of automation.

One such form of automation is to scan the documents (to produce document images) and subsequently perform OCR on the document images to convert them into text. Typically, the OCR process introduces errors in the text representation of the document images. The introduced errors are more pronounced in Arabic OCR due to some of the orthographic and morphological features of Arabic.

The introduced errors adversely affect retrieval effectiveness of OCR'ed documents. This paper examines the effect of word-based post-OCR error correction on Arabic retrieval effectiveness. The paper examines the effect of correction in using different index terms on two collections of degraded Arabic documents. The correction uses a character segment based noisy channel model to correct OCR errors.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 205 – 216, 2006.

The paper will be organized as follows: Section 2 provides background information on Arabic OCR and retrieval along with OCR error correction; Section 3 presents the experimental setup; Section 4 reports and discusses experimental results; and Section 5 concludes the paper and provides possible future directions.

2 Background

This section reviews prior work on OCR for Arabic, OCR error correction, and information retrieval for Arabic text.

The goal of OCR is to transform a document image into character-coded text. The usual process is to automatically segment a document image into character images, apply an automatic classifier to determine the character codes that most likely correspond to each character image, and potential exploit sequential character contexts to select the most likely character in each position. The character error rate can be influenced by reproduction quality (e.g., original documents are typically better than photocopies), the resolution at which a document was scanned, and any mismatch between the instances on which the character image classifier was trained and the rendering of the characters in the printed document. Arabic OCR presents several challenges, including: a) Arabic's cursive script in which most characters are connected and their shape vary with position in the word; b) the optional use of word elongations and ligatures, which are special forms of certain letter sequences; c) the presence of dots in 15 of the 28 to distinguish between different letters and the optional use of diacritic which can be confused with dirt, dust, and speckle [9]; and d) the morphological complexity of Arabic, which results in an estimated 60 billion possible surface forms, complicates dictionary-based error correction. Arabic words are built from a closed set of about 10,000 root forms that typically contain 3 characters, although 4-character roots are not uncommon, and some 5-character roots do exist. Arabic stems are derived from these root forms by fitting the root letters into a small set of regular patterns, which sometimes includes addition of "infix" characters between two letters of the root [3]. There are a number of commercial Arabic OCR systems, with Sakhr's Automatic Reader and Shonut's Omni Page being perhaps the most widely used. Retrieval of OCR degraded text documents has been reported for many languages, including English [14], Chinese [28], and Arabic [9].

Concerning OCR error correction, much research has been done to correct recognition errors in OCR-degraded collections. There are two main categories of determining how to correct these errors. They are word-level and passage-level post-OCR processing. Some of the kinds of word level post-processing include the use of dictionary lookup, probabilistic relaxation, character and word n-gram frequency analysis [16], and morphological analysis. Passage-level post-processing techniques include the use of word n-grams, word collocations, grammar, conceptual closeness, passage level word clustering, linguistic context, and visual context. The following introduces some of the error correction techniques.

• Dictionary Lookup: Dictionary Lookup, which is the basis for the correction reported in this paper, is used to compare recognized words with words in a term list [16, 28]. Jurafsky and Martin illustrate the use of a noisy channel model to find the correct spelling of misspelled or misrecognized words in a dictionary [17]. The model

assumes that text errors are due to edit operations namely insertions, deletions, and substitutions, and the number of edit operations required to transform one words to another is called the Levenshtein edit distance [6]. The probabilities of edit operations are captured in confusion matrices, and the probability that a candidate correction would be observed is obtained using word frequency in text corpus [17, 19]. However, the dictionary lookup approach has the following problems [16]: a) A correctly recognized word might not be in the dictionary. This problem can be acute for morphologically complex such as Arabic, German, and Turkish. b) A word that is misrecognized is in the dictionary. This problem is pronounced in a language such as Arabic where a large fraction of three letters sequences are valid words.

- Character N-Grams: Character n-grams maybe used alone or in combination with dictionary lookup [19, 26]. The premise for using n-grams is that some letter sequences are more common than others and other letter sequences are rare or impossible.
- Using Morphology: Many morphologically complex languages, such as Arabic, Swedish, Finnish, Turkish, and German, have enormous numbers of possible words. Accounting for and listing all the possible words is not feasible for purposes of error correction. Domeij proposed a method to build a spell checker that de-compounds words into constituent stems and attempts to correct the resulting stems using dictionary lookup techniques [11]. Similar work was done for Turkish in which an error tolerant finite state recognizer was employed [24]. The finite state recognizer tolerated a maximum number of edit operations away from correctly spelled candidate words. These techniques can potentially be applied to Arabic.
- Word Clustering: Another approach tries to cluster different spellings of a word based on a weighted Levenshtein edit distance. The insight is that an important word, specially acronyms and named-entities, are likely to appear more than once in a passage. Taghva described an English recognizer that identifies acronyms and named-entities, clusters them, and then treats the words in each cluster as one word [26]. Applying this technique for Arabic requires accounting for morphology, because prefixes or suffixes might be affixed to instances of named entities. DeRoeck introduced a clustering technique tolerant of Arabic's complex morphology [10]. Perhaps the technique can be modified to make it tolerant of errors.
- Using Grammar: In this approach, a passage containing spelling errors is parsed based on a language specific grammar. In a system described by Agirre, an English grammar was used to parse sentences with spelling mistakes [2]. Parsing such sentences gives clues to the expected part of speech of the word that should replace the misspelled word. Thus candidates produced by the spell checker can be filtered. Applying this technique to Arabic might prove challenging because the work on Arabic parsing has been very limited [22].
- Word N-Grams (Language Modeling): The word n-gram technique is a flexible method that can be used to calculate the likelihood that a word sequence would appear [27]. Using this method, the candidate correction of a misspelled word might be successfully picked given the context in which it is mentioned.

As for Arabic IR, most early studies of character-coded Arabic text retrieval relied on relatively small test collections [1, 5]; more recent results are based on a single large collection (from TREC-2001/2002) [13, 23]. Several types of index terms have

been examined, including words, word clusters, terms obtained through morphological analysis (e.g., stems and roots), and character n-grams of various lengths. The effects of normalizing alternative characters, removal of diacritics and stop-word removal have also been explored [7, 12, 18, 20, 21]. Early studies conducted on small collections suggested that roots were the best Arabic index terms [1, 5]. More recent studies using the larger TREC-2001/2002 Arabic test collection indicate that lightly stemmed words and character 3 and 4-grams result in better retrieval effectiveness than roots [7, 12, 18, 20, 21]. Retrieval effectiveness is known to be affected by the size, genre, and document length in the test collection, and by many details of system processing (e.g., character normalization, stop-word removal, and morphological analysis). As for OCR degraded Arabic text, a previous study suggest that 3 and 4 character grams and their combinations with index terms obtained through morphological analysis, such light stems, outperform all other kinds of index terms [9].

3 Experimental Setup

In the setup, documents are scanned, OCR'ed, OCR errors are optionally corrected, indexed, and searched. For evaluation, two collections were used. The first is a small collection of OCR degraded text. As for the second, due to the lack of existence of a large collection of Arabic OCR text, a large existing character-coded Arabic collection was corrupted to simulate OCR errors in the documents. For both collections, a portion of the collection was used to train an m:n OCR error correction model. The effect of corrupting the collection and its subsequent correction on retrieval effectiveness was examined. The following will present the collections, the error model which was used to corrupt the large collection, the error model that was used to correct both collections, and the design of experiments that were intended to test the effect of error correction on retrieval using different index terms.

The Document Collection. The first document collection is the Zad collection which was built from *Zad Al-Me'ad*, a printed 14th century religious book, which was scanned at 300x300 dpi and OCR'ed, and for which an electronic copy is available. The collection consists of 2,730 separate documents, 25 topics, and relevance judgments which were built by exhaustively searching the collection. The number of relevant documents per topic ranges from 3 to 72, averaging 20. The average query length is 5.4 words [9].

As for the large collection, the best presently available Arabic test collection was created for the TREC-2002 "Cross-Language IR (CLIR) track;" for brevity, it is referred to here simply as the TREC collection. It contains 383,872 articles from the Agence France Press (AFP) Arabic newswire. NIST developed 50 topics in cooperation with the Linguistic Data Consortium (LDC), and relevance judgments were developed at the LDC by manually judging a pool of documents obtained from combining the top 100 documents from all the runs submitted by the participating teams in TREC 2002 CLIR track. The number of known relevant documents ranges from 10 to 523, with an average of 118 relevant documents per topic [23]. The topic descriptions include a title field that briefly names the topic, a description field that usually consists of a single sentence description, and a narrative field that is intended to contain any information that would be needed by a human judge to accurately

assess the relevance of a document [15]. As for the corruption of the collection, a unigram model was used. OCR degradation was modeled as a noisy channel in which the observed characters result from the application of some distortion function on the real characters. The model used here accounts for three character edit operations: insertion, deletion, and substitution. Formally, given a clean word $\#C_1..C_i..C_n\#$ and the resulting word after OCR degradation $\#D_1..D_j..D_m\#$, where D_j resulted from C_i, ε representing the null character, L representing the position of the letter in the word (beginning, middle, end, or isolated), and # marking word boundaries, the probability estimates for the three edit operations for the models, are:

$$P_{\text{substitution}}(C_i \rightarrow D_j) = \frac{count(C_i \rightarrow D_j \mid L_{C_i})}{count(C_i \mid L_{C_i})}$$

$$P_{\text{deletion}}(C_i \rightarrow \varepsilon) = \frac{count(C_i \rightarrow \varepsilon \mid L_{C_i})}{count(C_i \mid L_{C_i})}; \quad P_{\text{insertion}}(\varepsilon \rightarrow D_j) = \frac{count(\varepsilon \rightarrow D_j)}{count(C)}$$

The models was trained using 2,000 words obtained by automatically aligning the real OCR outputs from the 300x300 dpi version of the Zad collection with the associated clean text version.

The resulting character-level alignments were used to create a garbler that reads in a clean word $\#C_1..C_i..C_n\#$ and synthesizes OCR degradation to produce $\#D'_1..D'_j..D'_m\#$. For a given character C_i, the garbler chooses a single edit operation to perform by sampling the estimated probability distribution over the possible edit operations. If an insertion operation is chosen, the model picks a character to be inserted prior to C_i by sampling the estimated probability distribution for possible insertions. Insertions before the # (end-of-word) marker are also allowed. If a substitution operation is chosen, the substituted character is selected by sampling the probability distribution of possible substitutions. If a deletion operation is chosen, the selected character is simply deleted.

Error Correction Model. A noisy channel OCR correction model was trained from manually correcting 2,000 randomly picked words from the automatically corrupted TREC documents. Another model was trained from 4,000 randomly picked tokens from the Zad collection. The trained models were used to correct the respective collections.

As for OCR model training, the goal is to learn an effective model of OCR degradation to enable effective correction of the OCR errors. It is desirable to minimize the number of training examples, because the process of producing the examples is manual. Previously published papers indicate that training an error model with 2,000 examples produces a good model with as little as 5,000 examples producing nearly the best possible model [8]. For this work, 2,000 words were randomly picked from the corrupted TREC collection to train the error correction model and 4,000 words were used from the Zad collection[1]. 2,000 words amount to nearly 2-4 pages in an average size book and require 20 to 30 minutes of correction

[1] Extra training data was used for the real OCR output because error types were more variant than those for the automatically corrupted data.

time. The characters in the corrupted and manually corrected training examples may be aligned in two different ways, namely: 1:1 character alignment (as done in the synthetic degradation process), where each character is mapped to no more than one character; or using m:n alignment, where any number of characters are aligned to any other number of characters. The second method is more general and potentially more accurate especially for Arabic where a character can be confused with as many as three or four characters. The following example highlights the difference between the 1:1 and the m:n alignment approaches. Given the training pair (rnacle,made):

1:1 alignment : m:n alignment:

For alignment, Levenstein dynamic programming minimum edit distance algorithm was used to produce 1:1 alignments. The algorithm computes the minimum number of edit operations required to transform one string into another. Given the output alignments of the algorithm, properly aligned characters (such as a → a and e → e) are used as anchors, ε's (null characters) are combined to misaligned adjacent characters producing m:n alignments, and ε's between correctly aligned characters are counted as deletions or insertions.

To formalize the error model, given a clean word $\#C_1..C_k..\ C_l..C_n\#$ and the resulting word after OCR degradation $\#D_1..D_x..\ D_y..D_m\#$, where $D_x..\ D_y$ resulted from $C_k..\ C_l$, ε representing the null character, and # marking word boundaries, the probability estimates for the three edit operations for the models are:

$$P_{substitution}\ (C_k..C_l -> D_x..\ D_y) = \frac{count(C_k..C_l \rightarrow D_x..D_y)}{count(C_k..C_l)}$$

$$P_{deletion}\ (C_k..C_l -> \varepsilon) = \frac{count(C_k..C_l \rightarrow \varepsilon)}{count(C_k..C_l)}$$

$$P_{insertion}\ (\varepsilon ->\ D_x..\ D_y) = \frac{count(\varepsilon \rightarrow D_x..D_y)}{count(C)}$$

When decoding a corrupted string δ composed of the characters $D_1..D_x..\ D_y..D_m$, the goal is to find a string χ composed of the characters $C_1..C_k..\ C_l..C_n$ such that argmax_χ $P(\delta|\chi)\cdot P(\chi)$ is maximum. $P(\chi)$ is the prior probability of observing χ in text and $P(\delta|\chi)$ is the probability of producing δ from χ.

For the Zad collection, $P(\chi)$ was computed from a web-mined collection of religious text by *Ibn Taymiya*, who was the main teacher of the medieval author of the Zad book. The collection contained approximately 16 million words, with 279,000 unique surface forms. As for the TREC collection, $P(\chi)$ was computed from a web-mined collection of Arabic newswire documents from the BBC, Al-Ahram newspaper,

Al-Jazeera news site, Al-Wafd newspaper, and Al-Moheet news site. The collection contains 12 million words, with nearly 260,000 unique surface words.
$P(\delta|\chi)$ is calculated using the trained model, as follows:

$$P(\delta \mid \chi) = \prod_{all:D_x..D_y} P(D_x..D_y \mid C_k..C_l)$$

The segments $D_x..\ D_y$ are generated by finding all possible 2^{n-1} segmentations of the word δ. For example, given "macle" then all possible segmentations are (m,a,c,l,e), (ma,c,l,e), (m,ac,l,e), (mac,l,e), (m,a,cl,e), (ma,cl,e), (m,acl,e), (macl,e), (m,a,c,le), (ma,c,le), (m,ac,le), (mac,le), (m,a,cle), (ma,cle), (m,acle), (macle).

All segment sequences $C_k..\ C_l$ known to produce $D_x..\ D_y$ for each of the possible segmentations are produced. If a sequence of $C_k..\ C_l$ segments generates a valid word χ which exists in the web-mined collection, then $\text{argmax}_\chi\ P(\delta|\chi)\cdot P(\chi)$ is computed, otherwise the sequence is discarded. Possible corrections are subsequently ranked.

In testing, two types of tests were performed to measure the effect of error correction. The first type examined the change in Word Error Rate (WER) which was computed by examining a set of approximately 2,000 and 6,000 words for the Zad and TREC collections respectively. Although the model accounts for m:n character alignments, this would not produce significantly better results than a model that accounts for 1:1 character alignments for the TREC collection, because the automatic garbler used a character unigram model to generate the test examples and the TREC collection. Offline experiments that are not reported here confirm this. The second examined the effect of correction on retrieval effectiveness. The retrieval experiments were performed on the original (uncorrupted/clean), automatically corrupted, and corrected versions of the Zad and TREC collections described above. Multiple corrected versions of the collections were generated by replacing each of the words in the corrupted versions by the top 1, 2, 3, or 5 corrections. The collections were indexed and searched using words, character 3-grams, character 4-grams, and lightly stemmed words obtained using Al-Stem [23]. For all experiments, Indri was used with default parameters with no blind relevance feedback. The figure of merit for evaluating retrieval results was mean average precision (MAP). Statistical significance between different retrieval results was performed using a paired 2-tailed t-test and a p-value of less than 0.05 to assume statistical significance. The use of the t-test was reported in this paper instead of the Wilcoxon test for two reasons. The first is that there are some indications that the t-test is sufficiently reliable despite the fact that the normality condition might not be met [25] and the Wilcoxon test with continuity correction was used offline to compare results in this paper but yielded no significant change in the conclusions reached. Hence, the Wilcoxon test p-values were not reported here.

4 Results and Discussion

Table 1 and Figures 1 and 2 summarize the effect of correction on WER for the Zad and TREC collections. A set of 2,000 and 6,000 words was used to test the correction of the Zad and the TREC collections respectively. The evaluation involved

Table 1. Effect of correction on Word Error Rate for ZAD/TREC collections

	ZAD							TREC						
		Number of Corrections							Number of Corrections					
%	bad	1	2	3	5	10	all	bad	1	2	3	5	10	all
WER	39.0	22.2	16.9	15.0	13.2	11.5	8.1	30.8	16.7	11.9	10.2	9.2	8.1	6.8
Error Reduction	-	42.9	56.7	61.5	66.0	70.5	79.2	-	45.8	61.4	66.8	70.1	73.7	78.0

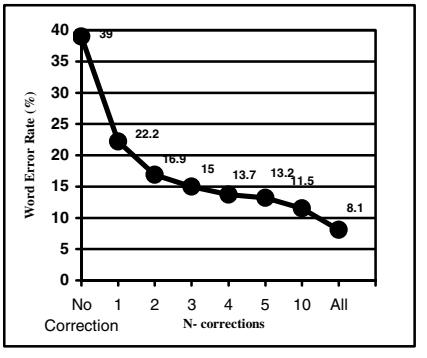


Fig. 1. Effect of correction on Word Error Rate for ZAD

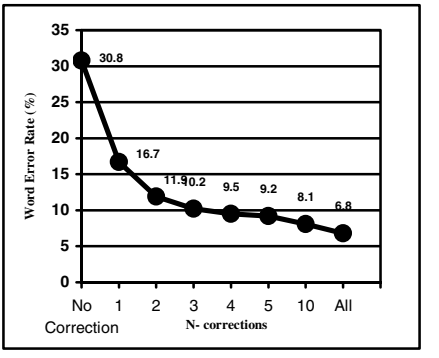


Fig. 2. Effect of correction on Word Error Rate for TREC

examining the top 1, 2, 3, 5, 10 generated corrections to determine if the proper correction exists within them.

The results show that in 8.1% and 6.8% of the cases none of the proposed corrections were actually correct for the Zad and TREC collections respectively. This is probably due to the absence of the proper corrections from the web-mined document collections. Perhaps, an increase in the size of the web-mined collections would improve word coverage and further reduce the word error rate. Nonetheless, the correction had a dramatic effect on the WER, dropping it by approximately 44% and 46%, for the Zad and TREC collections respectively and made the documents much easier to read (based on visual inspection of corrected documents). Another interesting observation is that although a word may be miscorrected, the miscorrection is often a morphological variant of the proper correction.

Figures 3 and 4 and Table 2 summarize the retrieval results of searching the original (clean), OCR'ed (corrupted/bad), and corrected (using 1, 2, 3, and 5 corrections) versions of the Zad and TREC collections respectively using words, character 3-grams, character 4-grams, and light stems. Table 3 provides the *p*-values of the paired 2-tailed t-test of comparing the results for the Zad and TREC collections. The results confirm that character 3 and 4-grams are indeed the best index terms with 3-grams on uncorrected text outperforming words and light stems even after correction. The results show that retrieval effectiveness statistically significantly generally improved or where statistically indistinguishable from the original uncorrupted versions of the collections when indexing using words. Same is true for the light stems for the TREC collection. However, the error correction did not

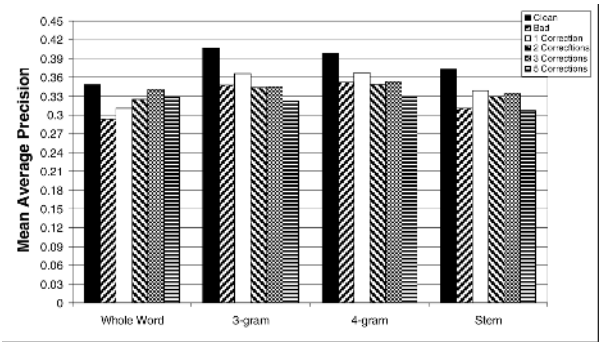


Fig. 3. Results in MAP searching the original, bad, and corrected versions of ZAD

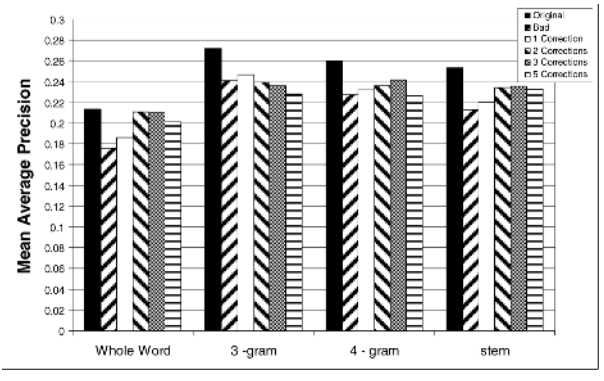


Fig. 4. Results in MAP searching the original, corrupted, and corrected versions of TREC

statistically significantly improve retrieval effectiveness over corrupted versions when indexing using character 3-grams and 4-grams and was generally statistically significantly worse than the corrected text (except for 4-grams for the Zad collection when using 3 and 5 corrections). The results seem to suggest that the effect of error correction on retrieval effectiveness is directly proportional to the length of the index term, with longer index terms benefiting more from correction. This would suggest that using sub-word correction schemes, as opposed to whole word correction, would be better suited for languages where the best index terms are short or the best character n-grams are short. Possibly better error correction could have had a statistically significant positive impact for shorter index term. Nonetheless, given a mildly or moderately degraded Arabic collection and barring the use of anything better than word based error correction, doing no correction and searching using character 3 or 4-grams does not seem to be a bad strategy. The results also suggest that indexing using short n-grams such as 3-grams is a better strategy than error correction. It is not clear whether this would be applicable to other languages, but the results indicate that this might be the case. Lastly, the use of more than one correction had little effect on retrieval effectiveness.

Table 2. Results in MAP of searching the original, bad, and corrected versions of ZAD and TREC collections

	Word	3 –gram	4 - gram	Light Stem	Word	3 –gram	4 - gram	Light Stem
Original (Clean)	0.35	0.41	0.40	0.37	0.21	0.27	0.26	0.25
Bad	0.29	0.35	0.35	0.31	0.18	0.24	0.23	0.21
1 Correction	0.31	0.37	0.37	0.34	0.19	0.25	0.23	0.22
2 Corrections	0.32	0.34	0.35	0.33	0.21	0.24	0.24	0.24
3 Corrections	0.34	0.35	0.35	0.34	0.21	0.24	0.24	0.24
5 Corrections	0.33	0.32	0.33	0.31	0.20	0.28	0.23	0.23

Table 3. *p*-value of the paired 2-tailed t-test comparison of retrieval results for the ZAD and TREC Collections. Black and Grey squares indicate that results are statistically significantly worse and better than corrected version respectively.

		Number of Corrections							
		ZAD Collection				TREC Collection			
		1	2	3	5	1	2	3	5
Words	Original	0.02	0.14	0.70	0.45	0.06	0.77	0.79	0.24
	Bad	0.12	0.05	0.03	0.06	0.46	0.00	0.01	0.05
Character 3-grams	Original	0.00	0.01	0.01	0.01	0.00	0.00	0.00	0.00
	Bad	0.06	0.84	0.91	0.32	0.29	0.76	0.53	0.17
Character 4-grams	Original	0.04	0.00	0.06	0.07	0.00	0.00	0.13	0.01
	Bad	0.28	0.70	0.94	0.44	0.59	0.21	0.23	0.95
Light Stems	Original	0.07	0.00	0.01	0.02	0.04	0.05	0.10	0.15
	Bad	0.02	0.25	0.04	0.82	0.61	0.01	0.02	0.10

5 Conclusion and Future Work

This paper examined the effect of OCR error correction on retrieval effectiveness of Arabic OCR degraded documents. Despite the fact that word error rate was nearly halved, the effect on retrieval effectiveness was less pronounced with statistically significant increases for longer index terms, namely words and light stems, and no statistically significant increases for shorter index terms. It would seem that perhaps using error correction for some languages with shorter optimal index terms might have less profound effect as opposed to languages with longer optimal index terms. This would suggest that an investigation of local correction versus whole word error correction is warranted and that indexing using short n-grams might be more beneficial than error correction.

For future work, there are a few clear directions to follow. They include the investigation of further improved error correction through the use of morphology and word n-gram (language modeling) techniques. Further, investigating sub-word error correction techniques may prove useful for languages where the best index terms are

short. Also, a serious exploration of the effect of correction on large real OCR document collections is warranted. Unfortunately, there are no reports in the literature of TREC size Arabic OCR document collections and much effort needs to be invested to create such collections. Also, the results reported in this paper need to be confirmed for other collection with different degradation levels.

References

1. Abu-Salem, H., M. Al-Omari, and M. Evens. Stemming Methodologies Over Individual Query Words for Arabic Information Retrieval. *JASIS, 50(6)* (1999) 524-529.
2. Agirre, E., K. Gojenola, K. Sarasola, and A. Voutilainen. Towards a Single Proposal in Spelling Correction. *In COLING-ACL'98* (1998).
3. Ahmed, M. A Large-Scale Computational Processor of Arabic Morphology and Applications. *MSc. Thesis, in Faculty of Engineering Cairo University: Cairo, Egypt.* (2000).
4. Aljlayl, M., S. Beitzel, E. Jensen, A. Chowdhury, D. Holmes, M. Lee, D. Grossman, and O. Frieder. IIT at TREC-10. *In TREC-2001, Gaithersbury, MD* (2001).
5. Al-Kharashi, I. and M Evens. Comparing Words, Stems, and Roots as Index Terms in an Arabic Information Retrieval System. *JASIS **45**(8)* (1994) 548-560.
6. Baeza-Yates, R. and G. Navarro. A Faster Algorithm for Approximate String Matching. *In Combinatorial Pattern Matching (CPM'96), Springer-Verlag LNCS* (1996).
7. Darwish, K. and D. Oard. CLIR Experiments at Maryland for TREC 2002: Evidence Combination for Arabic-English Retrieval. *In TREC-2002, Gaithersburg, MD* (2002).
8. Darwish, K. and D. Oard. Probabilistic Structured Query Methods. *In SIGIR-2003* (2003).
9. Darwish, K. and D. Oard. Term Selection for Searching Printed Arabic. *In SIGIR-2002* (2002).
10. De Roeck, A. and W. Al-Fares. A Morphologically Sensitive Clustering Algorithm for Identifying Arabic Roots. *In the 38th Annual Meeting of the ACL*, Hong Kong, (2000).
11. Domeij, R., J. Hollman, V. Kann. Detection of spelling errors in Swedish not using a word list en clair. *Journal of Quantitative Linguistics* (1994) 195-201.
12. Fraser, A., J. Xu, and R. Weischedel. TREC 2002 Cross-lingual Retrieval at BBN. In TREC-2002. Gaithersburg, MD (2002).
13. Gey, F. and D. Oard. The TREC-2001 Cross-Language Information Retrieval Track: Searching Arabic Using English, French or Arabic Queries. *In TREC-2001, Gaithersburg, MD* (2001).
14. Harding, S., W. Croft, and C. Weir. Probabilistic Retrieval of OCR-degraded Text Using N-Grams. *In European Conference on Digital Libraries* (1997).
15. Harman, D. Overview of the Fourth Text REtrieval Conference. *In TREC* (1995).
16. Hong, T. Degraded Text Recognition Using Visual and Linguistic Context. *Ph.D. Thesis, Computer Science Department, SUNY Buffalo: Buffalo* (1995).
17. Jurafsky, D. and J. Martin. Speech and Language Processing. *Prentice Hall* (2000).
18. Larkey, L., J. Allen, M. E. Connell, A. Bolivar, and C. Wade. UMass at TREC 2002: Cross Language and Novelty Tracks. In TREC-2002. Gaithersburg, MD (2002).
19. Lu, Z., I. Bazzi, A. Kornai, J. Makhoul, P. Natarajan, and R. Schwartz. A Robust, Language-Independent OCR System. *In the 27th AIPR Workshop: Advances in Computer Assisted Recognition, SPIE* (1999).

20. Mayfield, J., P. McNamee, C. Costello, C. Piatko, and A. Banerjee. JHU/APL at TREC 2001: Experiments in Filtering and in Arabic, Video, and Web Retrieval. *In TREC-2001. Gaithersburg, MD* (2001).
21. McNamee, P., C. Piatko, and J. Mayfield. JHU/APL at TREC 2002: Experiments in Filtering and Arabic Retrieval. *In TREC-2002, Gaithersburg, MD* (2002).
22. Moussa B., M. Maamouri, H. Jin, A. Bies, X. Ma. Arabic Treebank: Part 1 - 10Kword English Translation. *Linguistic Data Consortium*.
23. Oard, D. and F. Gey. The TREC 2002 Arabic/English CLIR Track. *In TREC-2002, Gaithersburg, MD* (2002).
24. Oflazer, K. Error-Tolerant Finite State Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics 22(1)* (1996) 73-90.
25. Sanderson, M. and J. Zobel. Information Retrieval System Evaluation: Effort, Sensitivity, and Reliability. *In SIGIR 2005, Sheffield* (2005).
26. Taghva, K., J. Borsack, and A. Condit. An Expert System for Automatically Correcting OCR Output. *In SPIE - Document Recognition* (1994).
27. Tillenius, M., Efficient generation and ranking of spelling error corrections. *NADA* (1996).
28. Tseng, Y. and D. Oard. Document Image Retrieval Techniques for Chinese. *In Symposium on Document Image Understanding Technology, Columbia, MD* (2001).

A Statistical Model of Query Log Generation

Georges Dupret[1], Benjamin Piwowarski[1], Carlos Hurtado[2] and Marcelo Mendoza[2]

[1] Yahoo! Research Latin America
[2] Departamento de Ciencias de la Computación, Universidad de Chile

Abstract. Query logs record past query sessions across a time span. A statistical model is proposed to explain the log generation process. Within a search engine list of results, the model explains the document selection – a user's click – by taking into account both a document position and its popularity. We show that it is possible to quantify this influence and consequently estimate document "un-biased" popularities. Among other applications, this allows to re-order the result list to match more closely user preferences and to use the logs as a feedback to improve search engines.

1 Introduction

The query log data of a search engine record the user queries, along with the URL and the position of selected documents in the list returned to the user. Documents can then be reordered according to their popularity for a given query. If a discrepancy is observed between this new order and the search engine ranking, some action can be taken to improve the engine. In Fig. 1 we plotted the number of selections of documents against their position in the ranking of a given query. It is intuitive that documents at positions 2 and 4 should be re-ordered because the latter has a higher popularity among users. Nevertheless, comparison of the documents appearing at positions 9 and 13 reveals a caveat of this method: The lower popularity of the latter document might be caused by a less favorable position rather than by a lower relevance to the query.

In this work, we propose to explain the selections observed in the logs as the result of a process that reflects 1) the attractiveness of a document surrogate for a particular query, 2) the ability of the engine to assess the document relative relevance accurately and 3) the influence of the position of the document on user selections.

In Section 2 we model these different aspects and gain insight into the log generation process. To have an intuitive understanding of this model, imagine for a moment that the search engine of Fig. 1 is stochastic and ranks a given document, say u, half of the time at rank 6, and half of the time at rank 15. (This situation may seem unrealistic, but in fact each time the document collection is updated, the ranking for a given query is altered and the engine, although deterministic, appears as stochastic in the logs.) If we observe 3 times more user selections when u is presented at position 6 than when presented at position 15,

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 217–228, 2006.

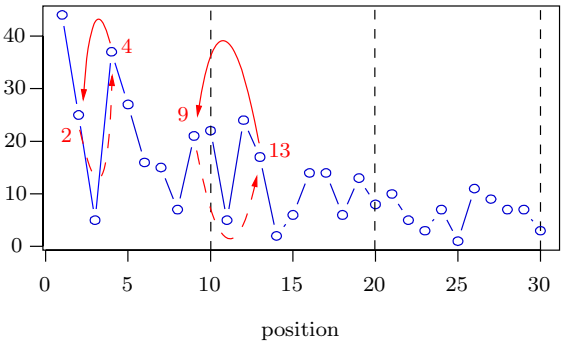


Fig. 1. Number of selection of documents per position for a typical query.

we can draw the conclusion that the position effect at position 6 is three times larger than at to position 15 (Section 2.2). Once the relative position effect is known, it is possible to compensate it on the observed popularity of document surrogates and re-order documents accordingly. In Section 3, we derive statistics that relate the model to aggregated values like the engine precision and the observed number of selections at a given position. Section 4 presents the results of numerical experiments based on the logs of a real search engine. In Section 5 we examine some re-ordering strategies found in the literature in terms of our model. Finally, in Section 6 we compare this work to the model proposed by Radlinski and Joachims [5] and report other potential applications.

2 Log Generation Model

In this section, we describe a probabilistic model of user selections which gives rise to the logs. It makes use of two hidden, latent variables, namely the influence of the position on user decisions and the attractiveness of the surrogate for a given query. The resulting Bayesian model makes a conditional independence assumption on these two variables.

2.1 Variables and Assumptions

We consider that an observation in the log is the realization of four random variables: A query (q) issued by the user, a document (u), the position of the document (p) and the information about whether the document was selected or not by the user (s). The probability of observing a selection is written $\mathrm{P}_{\mathcal{S}}(s, p, u, q)$, where the subscript $\mathcal{S}$ recalls the relation with the selections observed in the logs.

More specifically, a query q is defined as a sequence of terms given to the search engine. A significant number of queries are repeated by different users at different times, giving rise to various *query sessions* of q.

Table 1. Variables of the log-generation model

q	discrete	the **q**uery is q.
u	discrete	the document or URL is u.
a	binary	the document surrogate is **a**ttractive and justifies a selection.
p	discrete	the **p**osition of the document is p.
c	binary	the user **c**onsiders a given position.
s	binary	the user **s**elects the document.

The selection of a document u following a query q depends on the surrogate "attractiveness" a of the user who issued query q. When a is true, we say that the surrogate is attractive. The probability of a being true reflects the proportion of users that estimate that the surrogate is promising enough to justify a selection. If we make the assumption that the surrogate represents fairly the document, the probability of a can be interpreted as a relative measure of relevance. This is the approach taken by Radlinski and Joachims [5] among others. It is relative because only the relative relevance of selected documents can be compared this way [3].

While click-through data is typically noisy and clicks are not perfect relevance judgements, the user selections do convey information [8,4]. Occasional user selection mistakes will be cancelled out while averaging over a large number of selections. The question of how large should the logs be to counterbalance user variability and mistakes is an open question. The observed noise is large, which suggests that a large quantity of data is necessary. On the other hand, commercial search engines generate an enormous amount of data, and the comparatively small amount used in experiments reported in the literature [9,10,5] have led to satisfactory results.

The decision about the surrogate attractiveness can be regarded as a "positionless" process where users fetch all documents whose surrogate is pertinent to their query, wherever they are situated in the ranking. This is modelled by the upper part of the Bayesian network in Fig. 2: If the surrogate of document u is relevant to query q, the binary variable a is true. It is false otherwise.

Depending on the *position* p of document u in the ranking of query q, the user will consider the document or not, as reflected by the variable c ("consideration"). If this process was the only one involved in document selection, it would suppose "blind" users who select documents based solely on their position in the ranking or on any other interface artifact that appear at fixed positions like the division of results in pages. In the "blind" process, selecting a document or not is entirely determined by the document position p. The position p in turn is determined exogenously by the search engine based on the document u, the query q and the entire document collection (not represented because it is fixed), as shown by the lower branch of the Bayesian network. The existence of a bias due to the position of a document is well documented in the literature. Joachims [4] observes that despite differences in the quality of the ranking, users tend to click on average at the same positions.

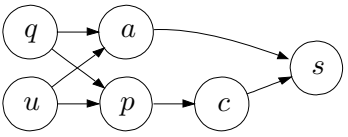


Fig. 2. Bayesian Network associated with the Log-Generation Model

2.2 Model and Estimation

According to the assumptions we just identified, we apply the chain rule to the probability of observing a selection (Fig. 2):

$$\mathrm{P}(s, c, a, p, u, q) = \mathrm{P}(s|a, c)\mathrm{P}(c|p)\mathrm{P}(p|u, q)\mathrm{P}(a|u, q)\mathrm{P}(u)\mathrm{P}(q)$$

where $\mathrm{P}(s|a, c)$ is deterministic because a user selects a document only if its surrogate is both attractive and considered. To eliminate the latent variables, we marginalize $\mathrm{P}(s, c, a, p, u, q)$ over a and c. Using the boldface ($\mathbf{s}$, $\mathbf{c}$ or $\mathbf{a}$) for binary random variables to denote the event "s, c or a is true", we have:

$$\mathrm{P}_{\mathcal{S}}(\mathbf{s}, p, u, q) = \mathrm{P}(\mathbf{c}|p)\mathrm{P}(p|u, q)\mathrm{P}(\mathbf{a}|u, q)\mathrm{P}(u)\mathrm{P}(q)$$

We introduce mnemonic indices. The term $\mathrm{P}(\mathbf{c}|p)$ is rewriten $\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p)$ to emphasize that it represents the position effect on user decision to make a selection. The term $\mathrm{P}(p|u, q)$ is the probability that a document is presented at position p. As this depends exclusively on the search engine, we add the subscript $\mathcal{E}$. The term $\mathrm{P}(\mathbf{a}|u, q)$ is the probability that the surrogate is attractive given the query. We use the subscript $\mathcal{A}$ to mark this fact. The log-generation model is:

Lemma 1 (Generation Process).
The process governing the log generation obeys to

$$\mathrm{P}_{\mathcal{S}}(\mathbf{s}, p, u, q) = \mathrm{P}_{\mathcal{A}}(\mathbf{a}|u, q)\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p)\mathrm{P}_{\mathcal{E}}(p|u, q)\mathrm{P}(u)\mathrm{P}(q)$$

The quantities $\mathrm{P}_{\mathcal{S}}(\mathbf{s}, p, u, q)$ and $\mathrm{P}_{\mathcal{E}}(p|u, q)$ can be estimated by simple counting from the logs: $\mathrm{P}_{\mathcal{S}}(\mathbf{s}, p, u, q)$ is estimated by the number of selection of document u at position p for all the sessions of query q, divided by the number of query sessions. $\mathrm{P}_{\mathcal{E}}(p|u, q)$ is estimated by the number of times u appears in position p of query q, divided by the number of sessions of q. $\mathrm{P}(u) = 1/U$ where U is the total number of documents. It is uniform and independent of $\mathrm{P}(q)$. The probability of observing a query is estimated by the proportion of sessions for that query.

The remaining two probabilities $\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p)$ and $\mathrm{P}_{\mathcal{A}}(\mathbf{a}|u, q)$ can be estimated using Lemma 1. We remark first that if $\mathrm{P}^*_{\mathcal{A}}(\mathbf{a}, u, q)$ and $\mathrm{P}^*_{\mathcal{P}}(\mathbf{c}|p)$ represent a solution to this system, then $\lambda\mathrm{P}^*_{\mathcal{A}}(\mathbf{a}, u, q)$ and $(1/\lambda)\mathrm{P}^*_{\mathcal{P}}(\mathbf{c}|p)$ is also a solution for any constant[1] $\lambda \neq 0$. This reflects that only the relative position effect can be

[1] Note that this solution may not be valid from a probabilistic point of view.

estimated. In practice, we either set $\sum_p \mathrm{P}_{\mathcal{P}}(\mathbf{c}|p) = 1$ or set the effect of a given position to a constant, say $\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p = 1) = 1$ and normalize $\mathrm{P}^*_{\mathcal{P}}(\mathbf{c}|p)$ afterward.

If the search engine is deterministic, i.e. if $\mathrm{P}_{\mathcal{E}}(p|u, q) = \{0, 1\}\ \forall u, q$, a simple analysis shows that there are more unknowns than equations in the system. Intuitively, this reflects that if each document appear always at the same positions in the ranking of a query, it is impossible to distinguish the effects of attractivity and position. The vast majority of search engines are designed to order the documents in a deterministic way, but in practice each time the document database is updated, the ranking of documents is altered, giving rise to a situation where the engine appears as stochastic and the system of equations can be solved.

If we restrict the system to the cases where $0 < \mathrm{P}_{\mathcal{E}}(p|u, q) < 1$ (inequalities are strict), we can transform the model into an overspecified system of linear equations by taking the logarithm:

$$\begin{cases} \log \mathrm{P}_{\mathcal{A}}(\mathbf{a}, u, q) + \log \mathrm{P}_{\mathcal{P}}(\mathbf{c}|p) = \log \frac{\mathrm{P}_{\mathcal{S}}(\mathbf{s},p,u,q)}{\mathrm{P}_{\mathcal{E}}(p|u,q)} \\ \log \mathrm{P}_{\mathcal{P}}(\mathbf{c}|p = 1) = \log(1) = 0 \end{cases} \tag{1}$$

where the unkowns are $\log \mathrm{P}_{\mathcal{A}}(\mathbf{a}, u, q)$ and $\log \mathrm{P}_{\mathcal{P}}(\mathbf{c}|p)$. The advantage is that the system can be solved now using standard software for sparse matrix algebra.

3 Aggregate Behavior

In this section we study the relation between the log generation process and aggregate values like the total number of selections at a position, the position effect and the engine precision.

The proportion of selections at a given position is simply the sum over all queries and documents of the selections at that position:

Definition 1 (Proportion of Selections at a position).
The proportion of selections at position p is defined as

$$\mathcal{S}_p = \sum_{u,q} \mathrm{P}_{\mathcal{S}}(\mathbf{s}, p, u, q)$$

Similarly, we also define $\mathcal{S} = \sum_p \mathcal{S}_p$.

To obtain the expected number of selections at a position, one need to aggregate $\mathcal{S}_p$ over all the sessions.

Precision is traditionally defined as the concentration of relevant documents in the result set. We can define by analogy a surrogate precision measure. To obtain the contribution of position p to this new measure, we observe that a document is placed at p by the engine with probability $\mathrm{P}_{\mathcal{E}}(p|u, q)$ and its surrogate is attractive with a probability $\mathrm{P}_{\mathcal{A}}(\mathbf{a}|u, q)$. By aggregating over documents and queries, we obtain:

Definition 2 (Attractiveness Gain).
For a given search engine, the attractiveness gain achieved at position p is the sum of the attractiveness of the documents surrogates appearing at that position:

$$\mathcal{G}_p = \sum_{u,q} \mathrm{P}_{\mathcal{A}}(\mathbf{a}|u,q)\mathrm{P}_{\mathcal{E}}(p|u,q)\mathrm{P}(u)\mathrm{P}(q)$$

If we make the assumption that the surrogates are fair representations of the documents, we can take the probability of attractiveness as an estimate of the document probability of pertinence. The sum of the gains up to rank k is then an empirical measure of the search engine precision at rank k. Note that this measure cannot be used to compare two different search engines in different settings [3].

We can now explain the selections observed in the logs as a consequence of Lemma 1 (Log Generation Process) and Definitions 1 and 2:

Lemma 2 (Aggregate Behavior).
The proportion of selections $\mathcal{S}_p$ at position p is the product of the gain and the position effect:

$$\mathcal{S}_p = \mathcal{G}_p \mathrm{P}_{\mathcal{P}}(\mathbf{c}|p).$$

A first consequence of this lemma concerns deterministic search engines where the probability $\mathrm{P}_{\mathcal{E}}(p|u,q)$ of finding document u at position p is 1 for a given position p_{uq} and 0 everywhere else: Rewriting the aggregate behavior in Lemma 2 with $p = p_{uq}$, we obtain

$$\mathrm{P}_{\mathcal{A}}(\mathbf{a}|u,q) = \frac{\mathcal{G}_p}{\mathcal{S}_p}\mathrm{P}_{\mathcal{S}}(\mathbf{s},p|u,q)$$

This matches intuition because the number of selections $\mathcal{S}_p$ generally decreases with the position and because selection at a later position is a better indicator of a surrogate attractiveness than a selection among the first positions. On the other hand, the term $\mathcal{G}_p$ counter-balances this effect and reflects the ability of the search engine to place attractive surrogates first.

4 Numerical Experiment

To illustrate the proposed method, we apply the former results to the logs of `todocl`, a search engine of the Chilean Web, for two periods of approximately 3 and 6 months separated by a new crawling. Various characteristics of the logs are shown in Table 2. The last column reports the numbers of distinct queries and documents that appear in both logs. We denote the three months log by L_3 and the six months log by L_6.

We first examine the impact of a strong and wrong simplifying assumption of our model, namely that the popularity of a surrogate does not depend on the surrogates that precede it in the results list and does not depend on the previous

Table 2. Characteristics of the three (L_3) and six months (L_6) logs and their intersection. The number of sessions in the last column is the sum of the number of sessions involving the queries and URL common to both logs.

	L_3	L_6	Common
Distinct Queries	65,282	127,642	2,159
Distinct selected URLs	122,184	238,457	9,747
Sessions	102,865	245,170	52,482

user selections during a given session, or at least that these effects cancel out. If this hypothesis is true, we should have that the popularity estimates in L_3 and L_6 are approximately equal:

$$\hat{\mathrm{P}}^{3}_{\mathcal{A}}(\mathbf{a}|p_3, u, q) \simeq \hat{\mathrm{P}}^{6}_{\mathcal{A}}(\mathbf{a}|p_6, u, q)$$

where p_3 and p_6 are the position of u in the results of q in L_3 and L_6 respectively. Because we cannot evaluate the probability of attractiveness directly, we restrict our attention to the cases where $p_3 = p_6$: We select the document query pairs whose document occupies the same position in the two logs. Formally, this set is defined as $I = \{(u, q)|(u, q) \in \mathsf{L}_3, (u, q) \in \mathsf{L}_6, p_3(u, q) = p_6(u, q)\}$. For these pairs, the position effect cancels out and the assumption holds if $\hat{\mathrm{P}}^{3}_{\mathcal{S}}(\mathbf{s}|p_3, u, q) \simeq \hat{\mathrm{P}}^{6}_{\mathcal{S}}(\mathbf{s}|p_6, u, q)$ where $\hat{\mathrm{P}}^{3}_{\mathcal{S}}(\mathbf{s}|p, u, q)$ denotes the estimate of $\mathrm{P}_{\mathcal{S}}(\mathbf{s}|p, u, q)$ derived from L_3 and $\hat{\mathrm{P}}^{6}_{\mathcal{S}}(\mathbf{s}|p, u, q)$ its equivalent from L_6.

The set I contains 2,159 distinct queries and 7,755 distinct documents for 8,481 pairs. Although for some pairs $(u, q) \in I$ the document u remains at the same position, the ranking of the other documents do change. This change in the *context* of the document is significant: Denoting Q_c the sets of queries q containing at least one document u that has not changed its position (i.e. $(u, q) \in I$), a query in Q_c contains an average of 14.1 and only 5.2 distinct documents in L_6 and L_3 respectively, revealing that the list of results contain different documents before and after the new crawl. The new position of documents that have been displaced by the new crawl are in average 7.8 ranks apart from the original one.

In Fig 3, we divide the $\hat{\mathrm{P}}^{3}_{\mathcal{S}}(\mathbf{s}|p, u, q)$ estimates for the (u, q) pairs of I in 10 bins of equal range and represent the corresponding six months estimates in 10 box-plots. Considering for example $\hat{\mathrm{P}}^{3}_{\mathcal{S}}(\mathbf{s}|p, u, q)$ between 0.4 and 0.5 (the $]0.4, .5]$ bin on the plot), we observe that the median of the corresponding $\hat{\mathrm{P}}^{6}_{\mathcal{S}}(\mathbf{s}|p, u, q)$ is slightly larger than 0.4. On the other hand, the lower and upper hinges (median of estimates smaller and larger than the median) are approximately 0.25 and 0.5, respectively.

The alignment of the box-plot medians with the diagonal shows that the surrogate estimate is stable under context variations, leading to the conclusion that the impact of previous selections tends to cancel out. If the median aligned on a horizontal line, this would mean that selections were governed by variables not included in the model. On the other hand, the large variation imply that

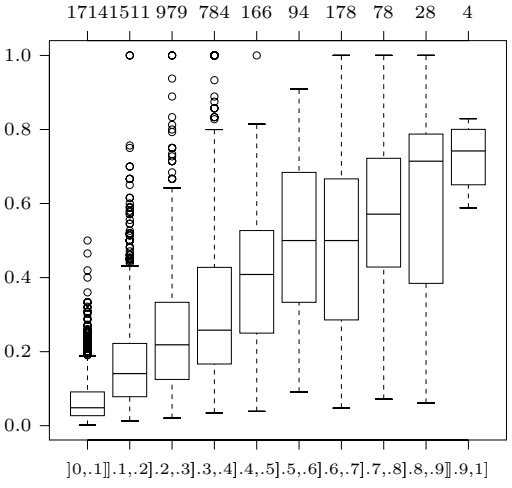


Fig. 3. On the lower axis, the estimates of the probability of selecting a document in a session of a query have been divided in 10 bins according to their values. For example, the second bin corresponds to the u, q pairs for which $10\% < \hat{\mathrm{P}}^3_{\mathcal{S}}(\mathbf{s}|p,u,q) \leq 20\%$. On the ordinate, we have for each of these u, q pairs the corresponding $\hat{\mathrm{P}}^6_{\mathcal{S}}(\mathbf{s}|p,u,q)$ estimate. On the upper axis, we report the number of u, q pairs that fall into the bin. The median is indicated by the black center line. The first and third quartiles are the edges (hinges) of the main box. The extreme values (more than 1.5 the lower/upper inter-quartile range) are plotted as points and are situated after the notches.

estimates will suffer from large variance. The inclusion of more variables into the model might reduce this problem, but this needs to be verified. The medians alignment to the diagonal deteriorates as the document popularity increases, suggesting that surrogates that enjoyed a high popularity in L_3 experience some decrease in L_6 (*Sic transit gloria mundi*[2]). This can be due partly to the documents getting outdated compared to documents that appear for the first time in L_6. Moreover, because L_6 contains twice as many documents, there is potentially a larger number of attractive surrogates for each query. This will partly divert selections from the documents already in L_3 and push down the relative number of selections. The larger distance of the median from the diagonal will induce a bias in the estimates of the model.

We compute the gain $\mathcal{S}_p$ from the solution of the linear system in Eq. 1 and plot it in Fig. 4 (triangle). While the position effect is expected to display discontinuities, in particular where pages change, it is reasonable to make the hypothesis that the gain decreases smoothly. We use a linear regression of second order on the experimental values and plot it as a continuous curve on Fig. 4. The position effect (circles) decreases from a 5% at position 1 and stabilizes around 3% around position 20, inducing that users are approximately 1.7 more likely to make a selection at the first than at the latter position in the blind

[2] Thus passes away the glory of the world

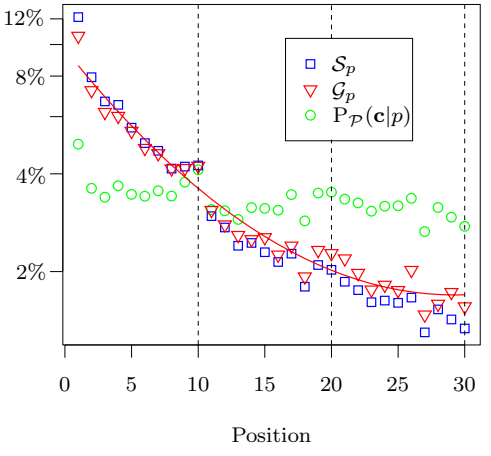


Fig. 4. All values are normalized by the sum of their values over positions 1 to 30. 1) The normalized number of selections $\mathcal{S}_p$ at position p (squares). 2) The position effect $\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p)$ (squares). 3) The normalized gain $\mathcal{G}_p$ (triangle).

process. There is a distinct increase of the position effect corresponding to the page change at position 10.

The gain presents a slope similar to the probability of selections, suggesting that users of the `todocl` engine are more influenced by the relevance of surrogates than by their position, although much larger samples of log data should be used to obtain reliable position effect estimates before definitive conclusions are drawn.

5 Re-ordering Strategies

We analyze in this section various strategies to take the logs into account in the document re-ranking and derive what Radlinski and Joachims [5] call a *osmosis* search engine, i.e. an engine that adapts itself to the users. These authors consider that the surrogates represent fairly the documents and that ordering according to the attractiveness respects the preference of the users for the documents themselves. We make the same hypothesis in this section and its results are valid only if it holds.

In order to compare two ranking strategies, we need to identify what the optimal ordering strategy should be. An optimal search engine estimates accurately the relevance of documents to queries, orders them accordingly and presents adequate surrogates to the users:

Definition 3 (Optimal Search Engine). *A search engine is optimal if, for all documents u and v and all query q, we have that*

$$p_{uq} < p_{vq} \Leftrightarrow \mathrm{P}_{\mathcal{A}}(\mathbf{a}|u,q) > \mathrm{P}_{\mathcal{A}}(\mathbf{a}|v,q).$$

where p_{uq} and p_{vq} are the positions of documents u and v in the result list of query q.

A consequence of this definition is that optimal search engines are almost always deterministic: Only when $\mathrm{P}_{\mathcal{A}}(\mathbf{a}|u,q) = \mathrm{P}_{\mathcal{A}}(\mathbf{a}|v,q)$ can a search engine be both stochastic and optimal.

The best strategy consists in ordering the documents according to the surrogates attractivity estimates, but it might be interesting to analyze the consequence of ordering according to the observed number of selections. Under this scheme, two documents are swapped if $\mathrm{P}_{\mathcal{S}}(\mathbf{s},p_2,u_2,q) > \mathrm{P}_{\mathcal{S}}(\mathbf{s},p_1,u_1,q)$ and $p_2 > p_1$. If the engine is deterministic, this can be rewritten $\mathrm{P}_{\mathcal{A}}(\mathbf{a},u_2,q)\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p_2) > \mathrm{P}_{\mathcal{A}}(\mathbf{a},u_1,q)\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p_1)$. This strategy implies that we have a permutation if

$$\mathrm{P}_{\mathcal{A}}(\mathbf{a},u_2,q) > \frac{\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p_1)}{\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p_2)}\mathrm{P}_{\mathcal{A}}(\mathbf{a},u_1,q)$$

This is adequate only if the position effects are equal. On the other hand, if $\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p_1)$ is sufficiently larger than $\mathrm{P}_{\mathcal{P}}(\mathbf{c}|p_2)$, document u_2 remains trapped at its original position.

Most applications that do not attempt to re-order selections, like query clustering [10], simply discard document popularity and consider only whether a document was selected or not. This is equivalent to setting the position effect and the popularity to a constant.

6 Related Work

In [5], Radlinski and Joachims pursue a goal similar to our. Based on eye tracking study and expert judgements, they identify six preference feedback relations based on query logs. For example, the selection of a document is interpreted as a preference feedback over all the documents preceding it in the list, but not selected. They observe that users often reformulate their query to improve search. The set of reformulation –called a query chain– is used to derive a relation of preference among documents. The selection of a document is interpreted as a preference feedback over all the preceding documents appearing but not selected in earlier formulations of the query. The preference relations are then set as constraints and used to train a Large Margin classifier.

It is interesting to compare the assumptions in Radlinski and Joachims [5] to ours. The preference feedback over preceding unselected documents parallels our assumption that documents should be ordered according to their (un-biased) popularity, and that the position of the selected document is of importance. They take popularity into account because a positive feedback is accounted for each document selection, and they take position into account because documents selected at latter positions have a preference relation over a larger number of documents than selected documents appearing at the top of the list. Popularity and position are thus two main variables in both their analysis and ours. On the other hand (provided we have enough data) we are able to derive a quantitative estimate of the position effect with no need of a priori hypothesis while Radlinski and Joachims fix arbitrarily some parameters to limit how quickly the original order is changed by the preference feedback data.

Our model does not take into account the influence of previous selections on user next selection. To a certain extent, Radlinski and Joachims include this information implicitly because no preference feedback is derived over a document that was previously selected. This is probably the most severe limitation of our model and the topic of future work. Future work should also attempt to take query chains into account.

Both methods fail to address the problem of ambiguous queries. An example of such query is "car" that refers to several topics like "renting a car", "buying a car", etc. The user has probably one and only one of these topics in mind and will select documents accordingly. The problem for both models is that they are attempting to order documents in answer to different information needs into a single list. A possible solution is to previously disambiguate queries [2].

In conclusion, Radlinski and Joachims model is more complete than ours essentially because previous selections are taken into account although implicitly. Our model on the other hand is less heuristic and makes explicit assumptions. It opens the doors to a more formal analysis. It also offers some primary theoretical results. The position effect is quantifiable, the difference between document preferences and surrogate attractiveness is made and dependence relations between variables is determined before hand.

7 Conclusions

We proposed a theoretical interpretation of data found in search engine logs. For a given query, it assumes that two factors influence a document selection: the position of the document in the result list and the attractiveness of the document surrogate. The main objective of this model is to estimate the effect of the document position in the ranking on users decisions to select it, thereby getting an un-biased estimate of the attractiveness of the document surrogate.

We foresee various applications to this work, but the most important one relates to the development of search engine that learns to rank documents from the users. Frequent queries rankings can be altered and cached to match users preferences rather than engine scores and consequently increase the engine precision[3]. Moreover, the score function of the search engine can be tuned based on user selections to improve the engine precision both on queries already in the logs and on future queries. Probabilistic retrieval models [7,6] rely on the probability of a document term to flag a relevant document given a query term. The estimation of this probability is based on user feedback and is unpractical to obtain explicitly, making the possibility to extract automatically the necessary information from the logs an important and novel method to improve significantly these engines.

It is also important to weight appropriately the documents in the relevance set when using feedback methods or when clustering queries. The knowledge of the influence of the position of a document on users selections decisions can be

[3] Pre-computation of frequent queries is also an effective way of improving engine speed due to the heavy skew of the query frequency distribution [1].

used to study the interface. Our approach was developed for results presented in list, but it is straightforward to extend it to tables of images or other multimedia content where the automatic estimation of a relevance score to a query is usually more problematic than for text documents.

The influence of previously seen documents on the user selection decision was neglected in the model. A user who finds a document that fulfil his information need is not likely to continue his search, thereby discarding other relevant documents. This is a topic for future work.

Acknowledgments

Carlos Hurtado was supported by Millennium Nucleus, Center for Web Research (P04-067-F), Mideplan, Chile.

References

1. R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE 2003, Manaus, Brazil, October 8-10, 2003. Proceedings*, Lecture Notes in Computer Science 2857, pages 56 – 65, 2003.
2. G. Dupret and M. Mendoza. Recommending better queries based on click-through data. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE 2005)*, LNCS 3246, pages 41–44. Springer, 2005.
3. T. Joachims. Evaluating search engines using clickthrough data. Department of Computer Science, Cornell University, 2002.
4. T. Joachims. Optimizing search engines using clickthrough data. In *KDD '02: Proceedings of the eighth ACM SIGKDD*, pages 133–142, New York, NY, USA, 2002. ACM Press.
5. F. Radlinski and T. Joachims. Query chains: learning to rank from implicit feedback. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 239–248, New York, NY, USA, 2005. ACM Press.
6. B. A. Ribeiro-Neto and R. Muntz. A belief network model for IR. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 253–260, New York, NY, USA, 1996. ACM Press.
7. S. E. Robertson and K. S. Jones. *Relevance weighting of search terms.* Taylor Graham Publishing, London, UK, UK, 1988.
8. T. Joachims. Unbiased evaluation of retrieval quality using clickthrough data. Technical report, Cornell University, Department of Computer Science, http://www.joachims.org, 2002.
9. J.-R. Wen, J.-Y. Nie, and H.-J. Zhang. Clustering user queries of a search engine. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 162–168, New York, NY, USA, 2001. ACM Press.
10. O. R. Zaïane and A. Strilets. Finding similar queries to satisfy searches based on query traces. In *Proceedings of the International Workshop on Efficient Web-Based Information Systems (EWIS)*, Montpellier, France, Sept. 2002.

Using String Comparison in Context for Improved Relevance Feedback in Different Text Media

Adenike M. Lam-Adesina and Gareth J. F. Jones

Centre for Digital Video Processing & School of Computing
Dublin City University, Dublin 9, Ireland
{adenike, gjones}@computing.dcu.ie

Abstract. Query expansion is a long standing relevance feedback technique for improving the effectiveness of information retrieval systems. Previous investigations have shown it to be generally effective for electronic text, to give proportionally better improvement for automatic transcriptions of spoken documents, and to be at best of questionable utility for optical character recognized scanned text documents. We introduce two corpus-based methods based on using a string-edit distance measure in context to automatically detect and correct transcription errors. One method operates at query-time and requires no modification of the document index file, and the other at index-time and operates using the standard query-time expansion process. Experimental investigations show these methods to produce improvements in relevance feedback for all three media types, but most significantly mean that relevance feedback can now successfully be applied to scanned text documents.

1 Introduction

Query expansion within relevance feedback (RF) has been shown to improve effectiveness for many information retrieval (IR) tasks, However, its performance varies for different text media for the same retrieval task. Performance differences arise from the indexing errors associated with the individual media. In this study we are concerned with improving the effectiveness of relevance feedback for a common retrieval task for the following text media: standard typed electronic text, transcriptions of spoken data created using automatic speech recognition (ASR), and transcriptions of scanned paper text documents generated using optical character recognition (OCR). While the accuracy of automatically generated digital document transcriptions continues to increase with advances in recognition technologies, the error levels are likely to remain sufficient to adversely affect relevance feedback effectiveness for the foreseeable future. Current transcription technologies achieve good performance on tasks such as read speech and recently printed texts, but still have very significant error levels for more challenging tasks such as conversational speech in noisy environments and nth generation photocopies or hand written texts. It can be observed that even the level of typographical errors found in published electronic texts can be sufficient to be detrimental to relevance feedback [1].

Relevance feedback using query expansion has previously been explored for all three media for both true relevance feedback using user-entered relevance judgements and

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 229–241, 2006.

pseudo relevance feedback (PRF) where the top ranked documents in the initial retrieval run are assumed to be relevant. The general conclusions for these studies are as follows: on average relevance feedback improves retrieval precision for typed text retrieval (TR), gives a proportionally greater improvement for spoken document retrieval (SDR) than text retrieval, but is not generally effective for document image retrieval (DIR). In fact relevance feedback can often reduce average performance for DIR.

In this paper we describe two string-based methods to improve relevance feedback performance for these text media. One operates at query-time and can be applied to existing document search collections without re-indexing. The other operates at indexing time and requires no modification of the standard query expansion process for relevance feedback at query-time. Both techniques apply a string-edit distance measure in context to identify likely misspellings or incorrectly transcribed valid words, and then seek to correct them from within the document collection. Both methods produce effective relevance feedback for DIR, and small improvements in relevance feedback for text retrieval, and the index-time method an improvement in SDR.

This paper focuses on the retrieval effectiveness of PRF in terms of standard precision and recall metrics. This is an experimental investigation and, as such, issues of computational efficiency of the implementation of the relevance feedback process are beyond the scope of this study.

The remainder of this paper is organised as follows: Section 2 gives a short review of relevant existing research, Section 3 outlines details of the Okapi BM25 information retrieval model used in this investigation, Section 4 summarizes the details of the test collection, Section 5 describes our extended relevance feedback methods and results for experiments using these techniques, and finally Section 6 concludes the paper.

2 Relevant Existing Research in Relevance Feedback

This section gives a brief summary of existing work in relevance feedback relevant to this paper. While relevance feedback has been studied for many years for different tasks, many recent investigations have taken place within tasks at the TREC workshops [2], including the main ad hoc search tasks and tasks focusing on other media. Another notable activity exploring relevance feedback was the Reliable Information Access (RIA) workshop in 2003 [3]. Within TREC, relevance feedback studies have mainly explored PRF, with results generally indicating that across a topic set PRF on average produces improvements in the standard TREC evaluation metrics.

Existing SDR studies of relevance feedback have again focused primarily on PRF within TREC SDR tasks [4]. Results here in general indicate that PRF is very effective for SDR with collections of automatically transcribed broadcast news [5]. These results are confirmed for a very different retrieval task of unstructured oral testimonies in the speech retrieval task introduced at CLEF 2005 [6].

The main results for DIR are again from TREC, this time in the confusion track [7]. Although participants explored a range of relevance feedback methods, the results were inconclusive since this was a single known-item search task. The absence of exhaustive document relevance information meant that it was not possible to study the effects of relevance feedback techniques thoroughly. Much more extensive evaluation of DIR has

been carried out at the University of Nevada at Las Vegas [8]. Results from these studies were again inconclusive, but suggested that relevance feedback is much less reliable for DIR than text retrieval and SDR.

In an earlier study of PRF for a parallel document collection for text retrieval, SDR and DIR we showed good performance for text retrieval, better relative performance for SDR, but a significant reduction in average precision and recall for DIR [9]. This result motivated us to investigate both the reasons for the ineffectiveness of PRF for DIR, and more generally to seek to understand the reasons for the variations in PRF effectiveness for different text media. In a previous study [1], we showed that the principal problem for PRF in DIR is the presence in the collection of high numbers of very rare index terms which are in fact character corrupted versions of standard words. While correctly spelled versions of these words have standard expected word frequencies within the collection as a whole.

In this paper we describe two techniques which address PRF problems for DIR and illustrate how these can also be effective for text retrieval and SDR.

3 Information Retrieval and Relevance Feedback Methods

The basis of our experimental setup is the City University research distribution version of the Okapi system [10]. The Okapi retrieval model has been shown to be very effective in many comparative evaluation exercises in recent years at TREC and elsewhere. The retrieval strategy adopted in this investigation follows standard practice for best-match ranked retrieval. The documents and search topics are first processed to remove common stop words from a list of around 260 words, suffix stripped using the Okapi implementation of Porter stemming to encourage matching of different word forms, and terms are further indexed using a small set of synonyms.

3.1 Term Weighting

Following preprocessing document terms are weighted using the Okapi BM25 weight [10]. The BM25 weight for a term is calculated as follows,

$$cw(i,j) = cfw(i) \times \frac{tf(i,j) \times (k_1 + 1)}{k_1 \times ((1-b) + (b \times ndl(j))) + tf(i,j)}$$

where $cw(i,j)$ represents the weight of term i in document j, $cfw(i) = log((N - n(i) + 0.5)/(n(i) + 0.5))$, $n(i)$ is the total number of documents containing term i, and N is the total number of documents in the collection, $tf(i,j)$ is the within document term frequency, and $ndl(j) = dl(j)/\text{Av}.dl$ is the normalized document length where $dl(j)$ is the length of j. k_1 and b are empirically selected tuning constants for a particular collection. The matching score for each document is computed by summing the weights of terms appearing in the query and the document.

3.2 Relevance Feedback

In the standard Okapi approach potential expansion terms are ranked using the Robertson's offer weight ($ow(i)$) [10], defined as,

$$ow(i) = r(i) \times rw(i) \tag{1}$$

where $r(i)$ is the number of relevant documents containing term i, and $rw(i)$ is the standard Robertson/Sparck Jones relevance weight [10] defined as,

$$rw(i) = \log \frac{(r(i) + 0.5)(N - n(i) - R + r(i) + 0.5)}{(n(i) - r(i) + 0.5)(R - r(i) + 0.5)}$$

where $n(i)$ and N have the same definitions as before and R is the total number of relevant documents for this query. The top ranking terms are then added to the original query. Term reweighting for relevance feedback is carried out by replacing $cfw(i)$ with $rw(i)$ in the BM25 weight. In this study we explore only query expansion since we generally observe this to be the dominant factor in relevance feedback.

Selection of expansion terms from whole documents can result in query drift if terms associated with non-relevant material are selected. In this study we adopt our sentence-based query-biased summary technique described in [11]. In this procedure potential expansion terms are selected from the query-biased summary of each potentially relevant document. This method has been shown to reduce the possibility of query drift in previous studies. Potential expansion terms are selected from the top R_1 documents assumed relevant, but the $ow(i)$ is calculated using a separate larger R value, since we find this to give more effective $ow(i)$'s.

4 Test Collections

The experimental investigation was carried out using a parallel research collection of text, spoken and image documents adapted from the TREC-8 SDR task [4]. The original SDR test collection consisted of the documents, search requests and relevant documents for each request. For our investigation we used a parallel document image collection consisting of scanned images generated from manual transcriptions of the audio data. The TREC-8 SDR collection is based on the English broadcast news portion of the TDT-2 News Corpus. The standard SDR collection of text and spoken document sets is augmented by forming a corresponding scanned document collection. The scanned document collection is based on the 21,759 "NEWS" stories in TDT-2 Version 3 (December 1999).

4.1 TDT-2 Document Set

The TREC-8 SDR portion of the TDT-2 News Corpus covers a period of 5 months from February to June 1998. The collection consists of 30 minute news broadcasts from CNN, ABC, PRI and VOA. Each broadcast is manually segmented into a number of news stories with unique identifers which form the basic document unit of the corpus. An individual news story was defined as containing two or more declarative statements about a single event. Other miscellaneous data items, e.g. commercials, were excluded from the data set. The collection contains a total of 21,759 stories with an average length of 180 words totalling about 385 hours of audio data.

Text Collection. There is no high-quality human reference transcription available for TDT-2 - only "closed-caption" quality transcriptions for the television sources and rough manual transcriptions for the radio sources made by commercial transcription services. A detailed manual transcription of a randomly selected 10 hour subset was carried out by the corpus developers to enable speech recognition accuracy to be evaluated. The television closed-caption sources (CNN, ABC) were found to have a Word Error Rate of approximately 14.5% and radio sources (PRI, VOA) to have a Word Error Rate of around 7.5%. The manual transcriptions are used as the document source for the scanned document collection used in this study.

Spoken Document Collection. The Spoken Document transcriptions used in our experiments are taken from the TDT-2 version 3 CD-ROMs. The transcription set used is designated `as1` on this release and was generated by NIST using the BBN BYBLOS Rough'N'Ready transcription system using a dynamically updated rolling language model. Full details of this recognition system are contained in [12]. This transcription was designated "B2" in the official NIST TREC-8 SDR documentation. The recognition Word Error Rate on a 10 hour subset of the data was reported by the developers to be 26.7%.

Scanned Document Collection. The printed version of the collection is formatted as hardcopy similar in style to newspaper clippings. To simulate the differences in formatting of stories from different newspaper sources, each story was printed in one of four fonts: *Times*, *Pandora*, *Computer Modern* and *San serif*. The stories were divided roughly equally between these font types with material from each source assigned to each one on a sequential basis. The stories were printed in one of three font sizes in single columns in one of six widths. Column width and font size were assigned sequentially from the beginning of each broadcast. The stories were printed using an Epson EPL-N4000 laser printer. In order to explore retrieval behaviour with a more errorful transcription than would naturally result from a printing of this quality, OCR transcription was performed with suboptimal system settings. All documents were scanned using an HPScanJet ADF at 200 dpi in Black & White at a threshold of 100. OCR was carried out using Page Keeper Standard Version 3.0 (OCR Engine Version 271) (SR3). Full details of the collection design are contained in [13].

4.2 TREC-8 SDR Test Collection

The TREC-8 SDR retrieval test collection contains a set of 50 search topics and corresponding relevance assessments. The goal in creating the topics was to devise topics with a few (but not too many) relevant documents in the collection to appropriately challenge test retrieval systems. Retrieval runs submitted by the TREC-8 SDR participants were used to form document pools for manual relevance assessment. The average topic length was 13.7 words and the mean number of relevant documents for each topic was 36.4 [4].

Table 1. Baseline and standard summary-based feedback results for TR, SDR and DIR

Media	TR				SDR				DIR			
	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet
Baseline	0.551	0.354	0.468	1608	0.496	0.321	0.406	1502	0.557	0.352	0.454	1581
Fbk 5	0.580	0.392	0.506	1639	0.500	0.346	0.423	1514	0.574	0.380	0.498	1578
chg bl. (%)	+5.3	+10.7	+8.1	+31	+0.8	+7.8	+4.2	+12	+3.1	+7.9	+9.7	-3
Fbk 20	0.598	0.396	0.514	1631	0.553	0.361	0.459	1532	0.539	0.352	0.440	1385
chg bl. (%)	+8.5	+11.9	+9.8	+23	+11.5	+12.5	+13.1	+30	-4.1	-0	-3.1	-196

5 Investigation of Relevance Feedback for Different Text Media

This section reports our investigation of query expansion for different text media. Results are shown for standard PRF methods, and our new techniques for enhancing the effectiveness of PRF. Retrieval metrics reported are precision at 10 and 30 document cutoff, standard TREC average precision (AvP) and the total number of relevant documents retrieved (RelRet). The total number of relevant documents retrieved for each run can be compared for Recall to the total number of relevant documents available across all topic statements in the TREC-8 SDR test set of 1818. Percentage change relative to a no PRF baseline is shown for precision measures and absolute change for the number of relevant documents retrieved.

The BM25 values were set empirically as $k_1 = 1.4$ and $b = 0.6$ using the baseline retrieval system with the text document collection without PRF to optimise AvP. The parameters for the summary-based PRF were set as follows. Summaries were based on the most significant 6 sentences, the top 5 ranked documents are the source of potential feedback terms (R_1), and the top 20 documents assumed relevant for computation of $ow(i)$ for term selection (R). The weight of the original query terms in each case was multiplied by 1.5 relative to the expansion terms, since the original terms have been chosen by the searcher themself. These values were again selected to optimise AvP on the text collection.

5.1 Baseline and Standard PRF Results

Table 1 shows baseline retrieval results without feedback (Baseline) and PRF results adding 5 (Fbk 5) and 20 (Fbk 20) expansion terms for text retrieval, SDR and DIR, with their changes from the baseline. From Table 1 it can be seen that there is a reduction in both baseline AvP and RelRet for SDR and a smaller one for DIR compared to TR. For PRF results, performance in terms of both precision and RelRet improves in all cases for TR and SDR. For DIR, PRF improves for 5 expansion terms, but AvP decreases by -3.1% and RelRet by -196 for 20 expansion terms.

In previous work [1] we demonstrated that the reduction in PRF performance for DIR is due to selection of some expansion terms with very low $n(i)$ values which are misrecognized versions of more common terms corrupted at the character level. We could of course try to correct these errors in post-processimg with a dictionary, however existing work [14] indicates that this can introduce more problems for information retrieval due to false substitutions than it solves. We thus do not explore dictionary-based substitution methods.

5.2 Improving PRF by String-Based Compensation for Transcription Errors

In previous work [1], we demonstrated that a simple filtering of terms with low $n(i)$ values partially addresses the problems with PRF for DIR associated with spelling mistakes illustrated in Table 1. However, the optimal value of $n(i)$ for filtering may be sensitive to the statistics of individual collections. Additionally, there are two notable problems with this very basic approach. First, correctly transcribed rare words that would actually be good expansion terms will be deleted along with the incorrectly transcribed ones, and thus not be available as potential expansion terms. Second, many incorrectly transcribed, apparently rare, words can be recognized manually as corrupted versions of correct terms appearing in assumed relevant documents. These variant forms are obvious to a human reader based on string similarity and the linguistic context in which they are found. Further, in such cases the $ow(i)$ values of the correctly transcribed terms will often be wrong since $r(i)$ will be underestimated when there is no other occurrence of i in a document within which it is incorrectly transcribed. Terms of this type will often be in high ranked documents for a query for which the terms are important. This is potentially a significant problem leading to distortion in the ranking of the $ow(i)$ ordered list compared to the one that would be formed without spelling errors in the documents, consequential reduction in the likelihood of choosing the best expansion terms, and thus potentially reduction in the possible effectiveness of relevance feedback. Spelling mistakes in text documents can also on occasion lead to similar problems for PRF in text retrieval which are not visible when looking across averaged results such as those shown in Table 1. While PRF works effectively for SDR, and the fixed vocabulary of automatic speech recognition systems used to generate automatic transcriptions means that misspellings of this type are not possible, the high overall Word Error Rate does affect retrieval effectiveness and means that there is scope to improve performance beyond that seen in Table 1.

Problems of eliminating good potential expansion terms and inaccurate estimates of $ow(i)$ can be overcome by identifying mistranscribed words within (assumed) relevant documents and combining them with correct words. In this section we introduce two methods for doing this. The first is a query-time technique that can be applied to existing indexed collections. While this method is found to be effective, it imposes an additional search time computational load. The second technique operates at index-time and imposes no additional search time cost.

Both procedures are based on a string comparison algorithm which computes an "edit distance" between two strings giving the minimum number of changes required to convert one string to the other [15]. These algorithms can make mistakes, sometimes merging words that are not related. However, within the constrained context of a small number of documents assumed to be relevant to a search query, often similar character strings really are the same word, leading to only a small number of false merges. This hypothesis is used as the basis of our correction techniques.

Query-Time Expansion Term Combination. In the query-time procedure the edit distance is computed between all terms within the top 5 ranked summaries used forPRF. Words within a preset edit distance are merged with the one with the larger $n(i)$ value assumed to be the correct. The $r(i)$ values of merged words are added, and the combined

Table 2. Results using string-comparison term merging at query-time

Media	TR				SDR				DIR			
MaxEd	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet
1	0.598	0.384	0.519	1614	0.551	0.362	0.459	1531	0.576	0.380	0.489	1593
chg bl. (%)	+8.5	+8.5	+10.9	+6	+11.1	+12.8	+13.1	+29	+3.4	+8.0	+7.7	+12
2	0.608	0.387	0.524	1614	0.553	0.362	0.456	1530	0.582	0.380	0.492	1581
chg.bl. (%)	+10.3	+9.3	+12.0	+6	+11.5	+12.8	+12.3	+28	+4.5	+8.0	+8.4	+0
3	0.610	0.399	0.528	1624	0.553	0.374	0.465	1541	0.596	0.385	0.505	1610
chg. bl. (%)	+10.7	+12.7	+12.8	+16	+11.5	+16.5	+14.5	+39	+7.0	+9.4	+11.2	+29
4	0.604	0.399	0.521	1639	0.549	0.363	0.454	1533	0.588	0.388	0.508	1616
chg. bl. (%)	+9.6	+12.7	+11.3	+31	+10.1	+13.1	+11.8	+31	+5.6	+10.2	+11.9	+35
5	0.598	0.393	0.523	1616	0.537	0.369	0.450	1552	0.592	0.386	0.507	1607
chg. bl. (%)	+8.5	+11.6	+11.8	+8	+8.3	+15.5	+10.8	+50	+6.3	+9.7	+11.7	+26

$n(i)$ value is taken as that of the larger value. The reduced set of potential expansion terms is then ranked by the $ow(i)$ computed using the merged $r(i)$ values.

Table 2 shows the result of using this merging approach with 20 expansion terms for maximum edit distance values of 1, 2, 3, 4 and 5, for TR, SDR and DIR. From Table 2 it can be seen that AvP is improved for both TR and DIR compared to the results shown in Table 1. For DIR performance clearly improves as the maximum distance is increased to 4 characters. There is little variation between results for maximum allowed edit distance values of 3, 4 and 5, suggesting that using a value of 4 will give good average stability across different queries for this collection. For TR the best maximum edit distance is 2 or 3, although any value above 1 gives very similar results. The technique does not appear to be effective for SDR; there is one AvP result above those in Table 1, but overall there is no trend indicating improvement. Analysis of $n(i)$ values in the speech documents shows that very few terms in the automatic transcription have low $n(i)$ values and occasional misspellings are not possible, and thus as observed the method has little scope for impact on PRF for SDR.

The success of this technique for TR and DIR can be attributed to the elimination of highly weighted rare misrecognized terms from the feedback terms for two reasons. First, the rank of non-relevant documents in the assumed relevant set which contain these terms is not now promoted by addition of these highly weighted terms to the search query. The rank of non-relevant documents may still be promoted due to the presence of other expansion terms, but this is a general drawback of query expansion in PRF for all media types. Second, in addition to their presence in the assumed relevant document set, although rare, if their $n(i)$ value > 1 these individual misrecognized terms can also occur in other documents, effectively with a random distribution. These other documents containing incorrect terms may include some or none of the original query terms, but when the query is expanded to include the highly weighted errorful terms, the matching score of the documents containing them can increase dramatically relative to other documents. While these documents may be relevant to the search request, it is most likely that they will often not be relevant. Overall then for TR and DIR the merging technique gives better estimation of $ow(i)$ due to more accurately calculating $r(i)$, and prevents problems of over promotion of documents containing

Table 3. Baseline and PRF results for indexing-time term combination ($e < 4$, $m > 4$, $R_1 = 5$)

Media	TR				SDR				DIR			
	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet
Baseline	0.451	0.365	0.476	1601	0.492	0.328	0.406	1487	0.559	0.359	0.464	1543
Fbk 5	0.596	0.386	0.508	1598	0.533	0.359	0.434	1520	0.598	0.378	0.499	1540
chg bl. (%)	+10.2	+5.8	+6.7	-3	+8.3	+9.5	+6.9	+33	+6.9	+5.3	+7.5	-3
Fbk 20	0.602	0.406	0.519	1617	0.533	0.365	0.458	1517	0.590	0.394	0.499	1600
chg. bl. (%)	+11.3	+11.2	+9.0	+16	+8.3	+11.3	+12.8	+30	+5.5	+9.7	+7.5	+57

Table 4. Baseline and PRF results for indexing-time term combination ($e < 4$, $m > 1$, $R_1 = 10$)

Media	TR				SDR				DIR			
	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet
Baseline	0.543	0.371	0.467	1568	0.498	0.335	0.414	1492	0.549	0.362	0.457	1539
Fbk 5	0.565	0.382	0.489	1573	0.535	0.366	0.451	1523	0.584	0.383	0.484	1558
chg bl. (%)	+4.1	+3.0	+4.7	1614	+7.4	+9.3	+8.9	+31	+6.4	+5.8	+5.9	+19
Fbk 20	0.588	0.412	0.525	1610	0.543	0.383	0.468	1549	0.590	0.399	0.503	1580
chg. bl. (%)	+8.3	+11.1	+12.4	+42	+9.0	+14.3	+13.0	+57	+7.5	+10.2	+10.1	+41

Table 5. Baseline and PRF results for indexing-time term combination ($e < 4$, $m > 4$, $R_1 = 10$)

Media	TR				SDR				DIR			
	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet
Baseline	0.539	0.369	0.472	1555	0.494	0.335	0.412	1487	0.539	0.378	0.464	1552
Fbk 5	0.600	0.391	0.510	1555	0.553	0.366	0.467	1493	0.569	0.378	0.495	1516
chg bl. (%)	+11.3	+6.0	+8.1	+0	+11.9	+8.5	+13.3	+6	+5.6	+0.0	+6.7	-36
Fbk 20	0.592	0.399	0.519	1539	0.561	0.389	0.483	1538	0.563	0.391	0.500	1591
chg. bl. (%)	+9.8	+8.1	+10.0	-16	+13.5	+16.1	+17.2	+51	+4.5	+3.4	+4.5	+39

errorful terms. While effective and not requiring re-indexing of the document collection, this method imposes a potentially significant computational load at query-time. In the next section we describe an index-time correction method using string-comparison in context which enables standard PRF methods to be used without modification.

Index-Time Combination for Term Correction. In addition to imposing a query-time computational load, having identified mistakes the search time technique cannot actually correct the mistakes in the documents. Thus incorrectly transcribed words in documents will still not match with the expanded query in the feedback retrieval run, there is no modification to term weights in the feedback retrieval run (for example based on correction of $tf(i, j)$ values), and the merging must be carried out each time a word appears in a new query.

In most cases when an incorrect word occurs in a document, we observe that it is often the case that the word appears correctly in other documents covering similar topics. We exploit this observation to correct mistranscribed words by using correctly transcribed ones in similar contexts. This procedure operates as follows.

All individual documents are converted into queries. Each document query is then used to query the complete original document collection. It is expected that the query (document) will retrieve itself in rank position 1 with the next ranked documents being closely related linguistically, and often topically. The procedure then seeks to correct mistranscriptions in the topmost ranked document using words within a preset edit distance contained in the next R_1 documents. The string-edit distance measure is used to compare each word in the top-ranked document to all words satisfying preset criteria in the documents ranked below them. These criteria are as follows. A candidate word must appear $\geq m$ times in the $R_1 - 1$ documents below rank 1 (since the item at rank 1 is the query itself), where $m = \sum_{k=2}^{R_1} tf(i,k)$ where k represents the documents containing the candidate combination terms. We also impose the constraint that only terms with identical first letter are allowed candidates; failure to do this was found to introduce too many incorrect candidates. Words satisfying these constraints and within an edit distance e are then added to the query document. The assumption being that if they are sufficiently frequent in the context of related documents and look similar to the term under consideration, then they are probably correct. We also explored the use of fixed values of $n(i)$ as the value of m, but found this to be not sufficiently discriminatory. Incorporating the $tf(i,j)$ rather than just binary presence/absence in m means that we capture multiple occurrences of a candidate word string closely related to the potential mistranscription, even if it only occurs in a very small number of documents matching the document query. The following is a short example snippet of a document with the inserted "corrections" shown in bold,

" ... look at out top stories - dosabled **disabled** gopfer **golfer** casey **case** martin won right drive cart **case** professional tnur. **tour** pga argued cart **case** gives martin unfair advamtage ..."

It can be seen that a number of accurate corrections are made, although some errors are made for short words, and no insertion is made for the term "advamtage" since candidates did not appear in the closely matching documents. "Advantage" is unlikely to be a topically specific term in this context and its appearance in related documents is thus likely to be a matter of chance.

This technique is similar to the document expansion technique described in [16] for SDR, but our method focuses on seeking to correct errors in individual elements of the identified content of the documents based on their character structure rather than using overall collection level statistics to select terms that are likely to have occurred in a document.

Tables 3, 4 and 5 show results for index-time combination with several settings of m and R_1, where $e < 4$ in all cases. These values were chosen after an extensive set of experiments with a subset of the test collection. Interestingly the value of $e < 4$ is the same as that which generally works best for the query-time technique. The tables show new baseline results which are needed since the features of the document collection have been changed. While the new AvP baseline figures here are only marginally higher than those in the original baseline in Table 1, results for 5 and 20 expansion terms show improvement in all retrieval measures. The change for the PRF runs is shown relative to the new baseline in each case. The method produces a marginal improvement for TR relative to Table 1, but PRF is now effective for DIR, although the absolute results are

Table 6. Index-time additions to 50 sample documents for Text, Speech and OCR collections

	Text	Speech	OCR
Identified Potential Errors	159	168	318
Correct Additions	54	57	95
False Positives	105	111	219

slightly lower than those in Table 2 using the query-time method. Using the indexing-time correction method there is now an improvement in retrieval performance for SDR compared to that in Table 1. While it may appear obvious that correction of the index file will improve retrieval effectiveness, the degree of change is not easily predictable. As we see here, while it only produces a small change in baseline retrieval performance, its effect on PRF is much more dramatic.

In order to explain these results more fully, we analyzed the behaviour of the correction method. We randomly selected 50 news story documents and extracted these for each of the document sets used to generate the results in Table 5. For each document we assessed each identified correction in the combined transcriptions. Results of this analysis are shown in Table 6 from which it can be seen that the number of corrections average around one per document for the Text and Speech data and two per document for the OCR data. A high number of False Positives appear in all cases. However, many of these are words strongly related to the correct word (e.g. "Yugoslav" appearing in place of "Yugoslavia", and similarly "Buddhism" for "Buddhist") which when stemmed will function as the correct search term. We also noted that on a number of occasions a word is added which, while not present in the original document, proves to be very useful for retrieval (e.g. "rifle" being combined with "right"). This last result indicates that there may be benefit in exploring document expansion methods further [16]. The number of corrections for the Text documents is perhaps unexpectedly high. However, it should be remembered that these transcriptions contain spelling mistakes and actual manual errors in transcriptions, as noted in Section 4.1.

A number of the false positives are short words unrelated to the contents of the document, and their presence in the document index may damage retrieval effectiveness. In order to reduce the number of false positives we imposed a further constraint on the index used to filter out combination words with < 6 characters. The length constraint was found to significantly reduce the number of false positives, but also the number correct additions. In retrieval experiments it was generally found to be more effective not to apply this length constraint, lack of space prevents us from reporting these results here.

Combining Index-Time and Query-Time Term Combination. Table 7 shows results of using the indexing-time combined collection from Table 5 with 20 expansion term PRF using query-time merging. The results follow similar trends with respect to the maximum edit distance to those in Table 2. In all cases any improvements over the results in Table 5 are very small, and absolute values are no better than those achieved for query-time only combination in Table 2. However, the improved result for SDR in Table 5 is preserved after the query-time combination. Overall though using both methods in

Table 7. Results using indexing-time term combination ($e < 4$, $m > 4$, $R_1 = 10$) with 20 expansion and query-time string-comparison term merging

Media	TR				SDR				DIR			
MaxEd	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet	P10	P30	AvP	RelRet
1	0.600	0.409	0.522	1538	0.563	0.391	0.483	1541	0.563	0.388	0.502	1598
chg bl. (%)	+11.3	+10.8	+10.6	17	+14.0	+16.7	+17.2	+54	+4.5	+8.4	+8.2	+46
2	0.598	0.409	0.523	1582	0.565	0.391	0.487	1549	0.576	0.390	0.499	1563
chg.bl. (%)	+10.9	+10.8	+10.8	+27	+14.4	+16.7	+18.2	+62	+6.9	+8.9	+7.5	+11
3	0.606	0.401	0.520	1604	0.561	0.388	0.482	1518	0.582	0.389	0.503	1580
chg. bl. (%)	+12.4	+8.7	+10.2	+49	+13.6	+15.8	+17.0	+31	+8.0	+8.7	+8.4	+28
4	0.586	0.398	0.507	1592	0.559	0.391	0.484	1548	0.569	0.391	0.508	1568
chg. bl. (%)	+8.7	+7.9	+7.4	+37	+13.2	+16.7	+17.5	+61	+5.6	+8.4	+9.5	+16
5	0.582	0.394	0.511	1575	0.559	0.393	0.488	1529	0.586	0.391	0.501	1590
chg. bl. (%)	+8.0	+6.8	+8.3	+20	+13.2	+17.3	+18.4	+42	+8.7	+8.4	+8.0	+38

combination is probably not justified computationally given the small variations from the results for the methods in isolation.

6 Conclusions and Further Work

Query-time and index-time methods have been described and evaluated using string-comparison in context to improve PRF for text retrieval, SDR and DIR. Positive results have been demonstrated on a parallel collection of text, speech and paper documents based on the TREC-8 SDR task. We are currently exploring the use of our document correction method for the CLEF speech retrieval task based on oral testimonies [6]. Following the promising results for text retrieval in this paper, we also intend to explore the application of these query and document combination techniques for term correction on larger text retrieval tasks. Preliminary results using the query-time method with the TREC-7 ad hoc search task indicate that it gives an improvement over results achieved using our standard information retrieval with PRF system. We believe that the results and methods described here easily extend to true relevance feedback, and we aim to demonstrate this in further work.

While our results so far are very encouraging, we can expect them to improve further if the correction methods are made more reliable. At present these make no formal use of linguistic context or data from the recognition process. A possible means to improve the correction methods accuracy could be to make use of statistical language models to give a quantitative measure of the likelihood of a potential correction term appearing in a particular place within a document, and the recognition likelihood data from speech recognition or OCR, or a statistical estimate of likely character string substitutions in combination with string-edit distance measures. Interesting methods using content correction techniques of this type have previously been reported in [17] [18].

Finally, we plan to explore the application of the results of this study for alternative information retrieval approaches such as query expansion when using language modelling methods.

References

[1] A. M. Lam-Adesina and G. J. F. Jones. Examining and Improving the Effectiveness of Relevance Feedback for Retrieval of Scanned Text Documents. *Information Processing and Management*, 43(3):633–649, 2006.

[2] `trec.nist.gov`

[3] `ir.nist.gov/ria/`

[4] J. S. Garafolo, C. G. P. Auzanne and E. M. Voorhees. The TREC Spoken Document Retrieval Track: A Success Story. In *Proceedings of the RIAO 2000 Conference: Content-Based Multimedia Information Access*, pages 1–20, Paris, 2000.

[5] S. E. Johnson, P. Jourlin, K. Sparck Jones and P. C. Woodland. Spoken Document Retrieval for TREC-8 at Cambridge University. In *Proceedings of the Eighth Text REtrieval Conference (TREC-9)*, pages 157–168, Gaithersburg, MD, 2000. NIST.

[6] R. W. White, D. W. Oard, G. J. F. Jones, D. Soergel and X. Huang. Overview of the CLEF-2005 Cross-Language Speech Retrieval Track. In *Proceedings of the CLEF 2005 Workshop*, Vienna, 2005.

[7] P. B. Kantor and E. M. Voorhees. The TREC-5 Confusion Track: Comparing Retrieval Methods for Scanned Text. *Information Retrieval*, 2:165–176, 2000.

[8] K. Taghva, J. Borsack, and A. Condit. Evaluation of Model-Based Retrieval Effectiveness with OCR Text. *ACM Transactions on Information Systems*, 14(1):64–93, 1996.

[9] G. J. F. Jones and A. M. Lam-Adesina. An Investigation of Mixed-Media Information Retrieval. In *Proceedings of the 6th European Conference on Research and Development for Digital Libraries*, Rome, pages 463–478, 2002, Springer.

[10] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu and M. Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference (TREC-3)*, pages 109–126. NIST, 1995.

[11] A. M. Lam-Adesina and G. J. F. Jones. Applying Summarization Techniques for Term Selection in Relevance Feedback. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1–9, New Orleans, 2001. ACM.

[12] C. Auzanne, J. S. Garafolo, J. G. Fiscus and W. M. Fisher. Automatic Language Model Adaptation for Spoken Document Retrieval. In *Proceedings of the RIAO 2000 Conference: Content-Based Multimedia Information Access*, pages 1–20, Paris, 2000.

[13] G. J. F. Jones and M. Han. Information Retrieval from Mixed-Media Collections: Report on Design and Indexing of a Scanned Document Collection. Technical Report 400, Department of Computer Science, University of Exeter, January 2001.

[14] E. Mittendorf and P. Schauble. Information Retrieval can Cope with Many Errors. *Information Retrieval*, 3:189–216, 2000.

[15] J. Zobel and P. Dart. Phonetic String Mathing: Lessons from Information Retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Zurich, pages 30–38, 1996, ACM.

[16] A. Singhal and F. C. N. Pereira. Document Expansion for Speech Retrieval. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Berkeley, pages 34–41, 1999, ACM.

[17] X. Tong and D. Evans. A Statistical Approach to Automatic OCR Error Correction in Context. In *Proceedings of the Fourth Workshop on Very Large Corpora*, Copenhagen, pages 88–100, 1996.

[18] K. Collins-Thompson, C. Schweizer and S. Dumais. Improved String Matching Under Noisy Channel Conditions. In *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM 2001)*, Atlanta, pages 357–364, 2001, ACM.

A Multiple Criteria Approach for Information Retrieval

Mohamed Farah and Daniel Vanderpooten

LAMSADE, Université Paris-Dauphine, France
{farah, vdp}@lamsade.dauphine.fr

Abstract. Research in Information Retrieval shows performance improvement when many sources of evidence are combined to produce a ranking of documents. Most current approaches assess document relevance by computing a single score which aggregates values of some attributes or criteria. We propose a multiple criteria framework using an aggregation mechanism based on decision rules identifying positive and negative reasons for judging whether a document should get a better ranking than another. The resulting procedure also handles imprecision in criteria design. Experimental results are reported.

Keywords: Information Retrieval, Relevance, Multiple Criteria.

1 Introduction

Information Retrieval (IR) is concerned with situations where a user, having information needs, performs queries on a collection of documents to find a limited subset of the most relevant ones. In the literature, a wide range of models have been proposed to rank documents according to their relevance to queries. They result in different rankings depending on the way they define relevance. In fact, relevance is reflected by the sources of evidence that are considered, as well as the way they are combined.

Most of the current approaches assess document relevance by computing a single score which aggregates values of elementary attributes related to the query terms, the document or the relationship between these two entities. For instance, in the Vector Space Model [1], the Okapi BM25 probabilistic model [2] as well as language models [3], term frequency (tf), document frequency (df) and document length (dl) are the main attributes which come into play. These attributes are combined in the term weighting formulation which corresponds to a first aggregation phase. The resulting scores are in turn considered to compute document relevance status value (rsv) to queries, as a second aggregation phase.

With the advent of hypertext collections, such as the Web, attributes characterizing the hyperlink structure are considered and led to link-based measures such as Kleinberg's HITS scores [4] and PageRank scores [5].

All these text- and link-based attributes can be combined to get better performance. A variety of aggregation operators have been used such as the min

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 242–254, 2006.

and max operators in [6] or the weighted linear operator in [7]. Other aggregation operators include similarity-based measures [8], P-norms [9], or fuzzy-logic conjunctive and disjunctive operators [10].

In some cases, aggregation is performed in two stages. In the first stage, text-based attributes are combined to get scores of documents. In the second stage, the resulting top ranked documents are re-ordered according to link information by using techniques such as spreading activation or probabilistic argumentation [11]. Thus, these approaches do not explicitly use link-attributes.

Each aggregation operator conveys a specific aggregation logic which reflects the degree of compensation we are ready to accept. In the IR literature, two main classes of operators are in use. The first class corresponds to a *totally compensatory logic*. It consists of building a single score using a more or less complex operator such as the weighted sum. For such operators, a very bad score on one criterion can be compensated by one or several good scores on other criteria. These operators require inter-criteria information such as weights, which are sometimes difficult to define and interpret. Indeed, these weights aim at capturing at the same time the relative importance of criteria but also a normalization factor when criteria are expressed on different scales.

The second class corresponds to a *non compensatory logic*. In this case, aggregation is mainly based on one criterion value such as the worst score or the score of the most important criterion. The remaining criteria are only used to discriminate documents with similar scores. This gives rise to min-based or lexicographic-based operators, variations of which are the *discrimin* and *leximin* operators [12]. A clear weakness of this class of operators is that a large part of the scores is ignored or plays a minor role.

In both classes, we do not consider *imprecision* underlying criteria design resulting from the fact that there are many formulations of the same criterion (see, e.g., [13] where four alternative formulations of the *tf* criterion are proposed). Therefore, it is important to give a limited interpretation to values, i.e., we should consider that slight differences in values are often not meaningful. This way, the resulting rankings are more *robust*.

In this paper, we propose a multiple criteria framework which combines any set of criteria while taking into consideration the imprecision underlying the criteria design process. We put emphasis on the importance of the design of good criteria families capturing *complementary* aspects of relevance and give clues to the design of such families. We describe ranking procedures based on natural decision rules.

The paper is organized as follows. We first introduce the multiple criteria framework where we describe the overall approach and its component phases (Section 2). Then, we highlight some specificities of the IR problem which are addressed in the proposed approach (Section 3). Section 4 deals with the modeling phase which consists in designing a set of relevance criteria. We present in Section 5, a filtering procedure whose purpose is to obtain a reduced set of potentially relevant documents. Section 6 shows how to aggregate such criteria

and build the final ranking. We report experimental results in Section 7 and provide conclusions in a final section.

2 A Multiple Criteria Framework for IR

Many studies argued that the reason why no consensus has been reached on the relevance concept is that there are many kinds of relevance, not just one (see e.g., [14]). Moreover, different sources of evidence are contributing to capture the relevance concept. Therefore, being able to make effective use of these sources of evidence can significantly improve retrieval effectiveness.

We propose a formal approach for IR where relevance is explicitly defined as multidimensional (by a set of criteria) and ranking is derived from pairwise comparisons of document performance vectors (*document profiles*) using decision rules identifying positive and negative reasons for judging whether or not a document should get a better ranking than another. The overall approach can be split into four phases:

- The *modeling phase* consists in identifying various attributes affecting relevance. These factors are used to develop a set of appropriate decision criteria which model different aspects of relevance. Each criterion will give rise to a *partial preference relation* (binary relation) modeling the way two documents are compared, according to that criterion.
- The *filtering phase* aims at identifying the set of *potentially relevant documents* with respect either to the query structure or to the criteria family. In the first case, a boolean filter selects documents that match query terms and query formula. In the second case, a profile-based filter selects documents that satisfy an *acceptance profile* defined by minimal required values on some or all criteria.
- The *aggregation phase* aggregates partial preference relations derived from pairwise comparisons of documents with respect to each criterion, into one or more *global preference relations*. A global preference relation indicates how two documents are compared with respect to all the considered criteria.
- The *exploitation phase* processes global preference relations resulting from the previous phase in order to derive the final ranking.

The last two phases correspond to the *ranking phase*.

It is worth noting that the proposed method is collection- and representation-independent to some extent. It can thus be used for any type of collection and combined with the best representation available. In fact, the context is mainly considered in the modeling phase in order to devise relevant criteria families.

3 Specificities of the IR Problem

The IR problem can be considered as a multiple criteria decision problem when we explicitly consider the multidimensional nature of relevance. Nevertheless, it has some particularities that have an impact on the modeling phase as well as on the aggregation and exploitation phases.

3.1 Specificities for the Modeling Phase

Specificity 1: Two kinds of criteria need to be considered to assess documents relevance: query-dependent and query-independent criteria.

Query-dependent criteria measure semantic proximity between documents and queries and are derived from factors about the form of occurrences of query terms in the document and the collection. Examples of such factors are term frequency (tf) and document frequency (df).

The evaluation of query-dependent criteria depends on the structure of the query. In fact, we should distinguish *one-term queries* from *multi-terms queries*. Some criteria are only relevant in the second case. Moreover, for multi-terms queries, two evaluation levels are required: (i) evaluation for each term of the query, and (ii) aggregation of these evaluations. Therefore, the design of such criteria deserves thorough analysis. This is addressed in Section 4.1.

Query-independent criteria mainly refer to characteristics of the document and the collection. They can be evaluated independently of the query. Examples of such criteria are document length (dl) and PageRank. We need such criteria to better help discriminating between documents. In fact, the query frequently consists of two or three terms in average, and this cannot be sufficient to rank thousands or millions of documents.

Specificity 2: Criteria can play different roles depending on which phase they are used in. In the filtering phase, they are primarily used to build acceptance profiles which help separating potentially relevant documents. In the ranking phase, they are used for pairwise comparisons.

3.2 Specificities for the Ranking Phase

Specificity 3: Criteria to be used to establish relevance are not specified by the user. They are rather based on factors evidenced to best capture relevance by the IR community. Consequently, it is difficult to get precise preference information regarding their relative importance. In this case, we assume that each criterion is neither prevailing nor negligible and use appropriate ranking procedures.

Specificity 4: The query is too poor to justify a precise ranking of documents. One can expect that many of the 'most relevant' documents should be present in the head of the ranking, but their exact ranking is meaningless. This can also be justified in terms of users behavior when interacting with the results pages of search engines. In fact, research in *eye-tracking* analysis of users behavior has shown that once users have started scrolling, rank becomes less of an influence for attention (see, e.g., [15]). Therefore, even if a ranking is a handy way to present results, its significance should not be overemphasized.

4 Modeling Phase

A criterion is the basis of relative relevance judgments as to whether a document is more or less relevant than some other document. It is modeled by a real-valued

function g defined on the set of documents which aims at comparing any pair of documents d and d', on a specific point of view, as follows:

$$g(d) \geq g(d') \Rightarrow d \text{ 'is at least as relevant as' } d'$$

Many formulations of each criterion are possible. Therefore, we should not overemphasize the criterion scores of documents. We briefly discuss two important issues in the modeling phase.

4.1 Evaluation of Query-Dependent Criteria

To build some query-dependent criteria, such as the *tf*-like criterion, we need to make a clear distinction between one-term and multi-terms queries. For one-term queries, criteria building has no specific difficulties, but to deal with multi-terms queries, i.e. conjunctive and/or disjunctive queries, we can proceed in two steps:

- build a *sub-criterion* corresponding to each term of the query. Each literal of the query formula can therefore be evaluated accordingly,
- select an aggregation operator corresponding to each query-type (conjunctive query, disjunctive query or a combination of both). This *sub-aggregation* step aggregates *homogeneous* partial measures derived from the previous step.

Since elements being aggregated in the sub-aggregation step are homogeneous, we can use analytic aggregation operators like conjunctive, disjunctive or compensatory operators [10], depending on the aggregation logic we wish to use and on the interpretation given to the juxtaposition of terms.

4.2 Modeling Imprecision

It is often inadequate to consider that slight differences in evaluation should give rise to clear-cut distinctions. This is particularly true when different formulations of criteria are acceptable. Imprecision underlying criteria design can be modeled using the following discrimination thresholds [17]:

- An *indifference threshold* allows for two close-valued documents to be judged as equivalent although they do not have exactly the same score on the criterion. The indifference threshold basically draws the boundaries between an indifference and a preference situation.
- A *preference threshold* is introduced when we want or need to be more precise when describing a preference situation. Therefore, it establishes the boundaries between a situation of a strict preference and an hesitation between an indifference and a preference situations, namely a weak preference.

A criterion g_j, having indifference and preference thresholds, q_j and p_j respectively ($p_j \geq q_j \geq 0$), is called a *pseudo-criterion*. Comparing two documents d

and d' according to a pseudo-criterion g_j leads to the following partial preference relations:

$$dI_j d' \Leftrightarrow -q_j \leq g_j(d) - g_j(d') \leq q_j$$
$$dQ_j d' \Leftrightarrow q_j < g_j(d) - g_j(d') \leq p_j$$
$$dP_j d' \Leftrightarrow g_j(d) - g_j(d') > p_j$$

where I_j, Q_j and P_j represent respectively *indifference*, *weak preference* and *strict preference relations* restricted to criterion g_j. These 3 relations could be grouped into an *outranking relation* $S_j = (I_j \cup Q_j \cup P_j)$ such that $dS_j d' \Leftrightarrow g_j(d) - g_j(d') \geq -q_j$ which corresponds to the assertion d *'is as least as relevant as'* d' with respect to the aspects covered by criterion g_j.

To model situations where a very low score of a document d' with respect to d, according to some criterion g_j, cannot be compensated by good score on one or several other criteria, we use a *veto threshold* v_j ($v_j \geq p_j$) and define the following *veto relation* V_j : $dV_j d' \Leftrightarrow g_j(d) - g_j(d') > v_j$. In this case, d' cannot be considered as *'at least as relevant as'* d.

5 Filtering Procedure

In this section, we show how it is possible to get the top k best relevant documents using acceptance profiles. In fact, acceptance profiles draw boundaries between two sets of documents: the first set consists of documents that can be considered better than the acceptance profile, and the second set consists of documents that can be considered worse than the acceptance profile. Different procedures can be used to obtain the top k best relevant documents. We give one such procedure.

Suppose that we have a set D of n documents, possibly resulting from the application of a first boolean filter, and we need to retain only the top k best documents, using an acceptance profile, i.e. acceptance thresholds a_j on each criterion g_j. The problem is to define these values a_j ($j = 1, \ldots, p$) such that the set of acceptable documents $A = \{d \in D | g_j(d) \geq a_j \ (j = 1, \ldots, p)\}$ has an approximate cardinality of k. A simple way of setting and adjusting values a_j ($j = 1, \ldots, p$) is to adjust a single parameter α corresponding to a percentile used on all criteria scales. In this case, a_j is such that a proportion $(1-\alpha)$ of the n documents satisfy $g_j(d) \geq a_j$ and a proportion α of the n documents satisfy $g_j(d) < a_j$. Considering that we want to apply the same percentile to all criteria, and we aim at retaining a proportion of $\frac{k}{n}$ of documents from D, α can be set to an initial value of $1 - \sqrt[p]{\frac{k}{n}}$. Using a dichotomic procedure, α can be adjusted so as to obtain the required size for the filtered set A.

6 Ranking Procedure

In order to get a global relevance model on the set of documents, we use *outranking approaches* [18], which are quite appropriate regarding the specificities of Section 3.2 and are based on a *partial compensatory logic*. They consist of two phases: an aggregation phase and an exploitation phase.

6.1 Aggregation Phase

Outranking approaches take as input the partial preference relations induced by the criteria family and aggregate them into one or more global preference relation(s) S. They are particularly relevant in our context since they (i) permit considering imprecision in document evaluations, (ii) can handle criteria expressed on heterogeneous scales, (iii) use all the available information on document performances, and (iv) do not necessarily require inter-criteria information.

In order to accept the assertion dSd', stating that 'document d is at least as relevant as document d'', the following conditions should be met:

- a *concordance* condition which ensures that a majority of criteria are concordant with dSd' (*majority principle*).
- a *discordance* condition which ensures that none of the discordant criteria strongly refutes dSd' (*respect of minorities principle*).

In this paper, we suppose that there is no information on the *relative importance of criteria*. In this case, to accept the assertion dSd', we use decision rules based on the criteria *supporting* (positive reasons) or *refuting* (negative reasons) this assertion. Obviously, the rules for defining this support may be more or less demanding, resulting in different outranking relations. For example, let

- $F = \{g_1, \ldots, g_p\}$ be a family of p criteria,
- H be a global preference relation, where H is P, Q, I, V or S,
- H^- be a relation such that $dH^-d' \iff d'Hd$,
- H_j be a partial preference relation, i.e. restricted to criterion g_j,
- $C(dHd') = \{j \in F : dH_jd'\}$ be the concordance coalition of criteria in favor of establishing dHd', and
- $c(dHd')$ the number of items in $C(dHd')$

A candidate outranking relation is:

$$dS^1d' \Leftrightarrow C(dSd') = F \tag{1}$$

which is a well established, but usually poor, relation since it only holds if all the criteria are concordant with dSd'.
We can also use less demanding outranking relations such as:

$$dS^2d' \Leftrightarrow c(dPd') \geq c(dP^- \cup Q^-d') \quad \text{and} \quad C(dV^-d') = \emptyset \tag{2}$$

To accept dS^2d', there should be more criteria concordant with dPd' than criteria supporting a strict or weak preference in favor of d'. At the same time, no discordant criterion should strongly disagree with this assertion.

6.2 Exploitation Phase

Outranking relations are not necessarily transitive and do not lend themselves to immediate exploitation to get the final ranking. Therefore, we need exploitation procedures in order to derive the final document ranking. We propose the

following procedure which finds its roots in [19]. It consists in partitioning the set of documents into r ranked *classes* where each class C_h contains documents with the same score. This is coherent with specificity 4 of Section 3.2. Let

- R be the set of potential relevant documents for a query,
- $F_i(d, E) = card(\{d' \in E : dS^i d'\})$ be the number of documents in $E(E \subseteq R)$ that could be considered 'worse' than d according to the global relation S^i,
- $f_i(d, E) = card(\{d' \in E : d'S^i d\})$ be the number of documents in E that could be considered 'better' than d according to S^i,
- $s_i(d, E) = F_i(d, E) - f_i(d, E)$ be the *qualification* of d in E according to S^i.

Each class C_h results from a *distillation process*. It corresponds to the last distillate of a series of sets $E_0 \supseteq E_1 \supseteq \ldots$ where $E_0 = R \setminus (C_1 \cup \ldots \cup C_{h-1})$ and E_i is a reduced subset of E_{i-1} resulting from the application of the following procedure:

1. compute for each $d \in E_{i-1}$ its qualification according to S^i, i.e. $s_i(d, E_{i-1})$,
2. choose $s_{\max} = \max_{d \in E_{i-1}}\{s_i(d, E_{i-1})\}$, then
3. $E_i = \{d \in E_{i-1} : s_i(d, E_{i-1}) = s_{\max}\}$

When one outranking relation is used, the distillation process stops after the first application of the previous procedure, i.e., C_h corresponds to distillate E_1. When different outranking relations are used, the distillation process stops when all the pre-defined outranking relations have been used or when $card(E_i) = 1$.

7 Experiments and Results

7.1 Test Setting

To facilitate empirical investigation of the proposed methodology, we developed a prototype search engine, named WIRES, that implements a preliminary version of our multiple criteria approach. In this paper, we apply our approach to the Topic Distillation (TD) task of TREC-13 Web track [20]. In this task, there are 75 topics where only a short description of each is given. For the experiments, we translated each topic to a conjunctive query following most search engine strategies. We have built an inverted index of the '.GOV' TREC test collection where we consider word stems as index terms using the Porter stemming algorithm and discard common english stopwords. We also used the hyperlink structure of this collection to build link-based criteria.

At a first level, we had to define the set F of criteria, for which we used the following elementary features which are the main factors used in the literature:

- tf_k : frequency of term t_k in document d,
- df_k : number of documents the term t_k occurs in,
- $\max tf$: maximum frequency tf_k of all terms $t_k \in d$,
- $l_{k,a}$: a binary value which equals 1 if term t_k occurs in location L_a and 0 otherwise. The considered locations are the URL ($L1$), the title (L_2), the keywords tag (L_3) and the description tag (L_4),

- $\Gamma^-(d)$: set of incoming hyperlinks to d,
- $Child(d)$: set of children documents of d. Document d' is in $Child(d)$ if it appears in a lower hierarchical level than d according to their site map,
- $prox$: proximity of query terms in document d. It corresponds to the size (number of terms) of the smallest text excerpt from the document that contains all the query terms. It equals 0 if not all the query terms are in d,
- ql : query length, i.e. the number of terms of the query,
- dl : document length, and
- $depth(d)$: depth of the URL of d, which is the number of intermediary sub-directories between document d and the root of its corresponding site map.

Based on these features, we defined the following candidate criteria:

- Frequency : For one-term queries (i.e. $q = t_k$), $g_1(d, t_k) = \frac{tf_k}{\max tf}$
- Position : For one-term queries, $g_2(d, t_k) = \sum_{a=1}^{4} l_{k,a}$
- Authority : $g_3(d) = card(\Gamma^-(d))$
- Prominence : $g_4(d) = card(Child(d))$
- Proximity : $g_5(d) = \frac{ql}{prox}$ if $prox \neq 0$, *and* 0 otherwise
- Document length : $g_6(d) = dl(d))$
- Rareness : For one-term queries, $g_7(d, t_k) = df_k$

For multi-terms queries, we used the average operator.

It is worth noting that the concrete choice of the features as well as the criteria family should be monitored to best correspond to the specific application context. For our experiments, we focused on features which capture the major well-known IR evidences. The exact definition of each criterion tries to capture intuitive preferences but remains somewhat arbitrary, thus we considered simple formulations, but more refined formulations can be used too.

In the TD task, a successful relevance ranking should favour 'good entry points' although they could contain little detailed information. This is captured by the prominence criterion (g_4).

For evaluation, we used the 'trec_eval' standard tool which is used by the TREC community to calculate the standard measures of system effectiveness which are Average precision (`AvP`), `R-p`recision (`R-p`), Reciprocal rank (`r-r`) and Success@n (`S@n`) (see, e.g., [20]).

Our approach effectiveness is compared against some high performing official results from TREC-13 using the *paired t-test* which is shown to be highly reliable (more than the *sign* or *Wilcoxon* tests) according to [21]. In the experiments, significance testing is mainly based on the `t-student` statistic which is computed on the basis of the `AvP` values of the compared runs. In the tables of the following section, we have marked with an asterisk statistically significant differences.

7.2 Results

With the criteria described before, we performed several retrieval runs. In the first set of runs, we rank documents according to each criterion and report performances in Table 1. We aim at showing which criteria are really relevant for the TD task.

Table 1. Performances of mono-criterion runs

Run Id	AvP	R-p	r-r	S@1	S@5	S@10	Δ-AvP
prominence	12.37%	16.15%	46.42%	29.33%	74.67%	85.33%	–
authority	9.27%	10.41%	31.68%	18.67%	44.00%	57.33%	-25.03%*
position	7.30%	6.66%	21.62%	12.00%	29.33%	42.67%	-40.99%*
frequency	7.01%	6.49%	16.44%	6.67%	24.00%	37.33%	-43.36%*
random	3.17%	2.42%	9.90%	4.00%	10.67%	22.67%	-74.40%*
proximity	2.78%	2.14%	4.73%	0.00%	6.67%	9.33%	-77.56%*
rareness	2.27%	1.00%	4.24%	1.33%	2.67%	9.33%	-81.65%*
length	1.76%	0.22%	2.19%	0.00%	2.67%	2.67%	-85.74%*

Table 1 shows that the run with the prominence criterion (g_4) performs significantly better than the others. Runs carried out with the first 4 criteria perform significantly better than runs carried out with the last 3. Moreover, the random run `random` performs better than the same 3. Therefore document length (g_5), proximity (g_6), and rareness (g_7) do not play an important role for the TD task.

In the second set of runs, we only considered the best four criteria, i.e. criteria g_1–g_4. In our baseline run (`mcm`), the set R of potentially relevant documents is obtained in two stages: we first use the boolean filter to identify a first set A which is then extended to a set A^+ that includes each document pointing to at least 2 documents in A. Many of the added documents should, in fact, correspond to good entry points to relevant sites. In the aggregating procedure of Section 6.1, each criterion is supposed to be a pseudo-criterion where indifference, preference and veto thresholds are set to 20%, 60% and 90%, respectively. These thresholds are set after some tunings carried with respect to TREC-12 Web track TD topics. We suppose that there is no information on the relative importance of criteria and use the outranking relation S^2 of (2). We implement the exploitation procedure of Section 6.2.

We now try to catch the impact of profile filtering on performance using the procedure presented in Section 5 which allows us to get a reasonably small set R of documents. We carried out some runs where we tried to get different numbers of filtered documents : for each run `mcm-filter-x`, x corresponds to the number of the filtered documents.

Table 2 shows that `mcm-filter-x` runs differs only with respect to `AvP` and `R-p`. All the other measures remain the same. This is because all these runs have

Table 2. Impact of filtering procedure

Run Id	AvP	R-p	r-r	S@1	S@5	S@10	Δ-AvP
mcm	17.08%	18.37%	58.04%	45.33%	74.67%	81.33%	–
mcm-filter-1000	17.00%	18.37%	58.04%	45.33%	74.67%	81.33%	-0.46%
mcm-filter-800	16.83%	18.37%	58.04%	45.33%	74.67%	81.33%	-1.45%
mcm-filter-500	16.52%	18.34%	58.04%	45.33%	74.67%	81.33%	-3.26%
mcm-filter-50	15.65%	18.40%	58.04%	45.33%	74.67%	81.33%	-8.35%*

the same ranking at the top. When we filter 50 documents, performance decreases rather significantly, whereas considering the R-p measures, performance slightly increases. Performances do not significantly decrease with respect to those of mcm when we filter 1000, 800 or 500 documents. We can conclude that filtering is beneficial for IR since it considerably reduces the size of the set of documents to be compared in the ranking procedure, and at the same time, it does not lead to significant performance drop.

We compare now our basic run mcm with other aggregation strategies.

Table 3. Different aggregation strategies

Run Id	AvP	R-p	r-r	S@1	S@5	S@10	Δ-AvP
mcm	17.08%	18.37%	58.04%	45.33%	74.67%	81.33%	–
max	8.02%	7.70%	21.40%	8.00%	33.33%	50.67%	-53.02%*
min	10.74%	12.91%	47.20%	32.00%	70.67%	77.33%	-37.13%*
prod	12.06%	14.02%	53.66%	37.33%	74.67%	80.00%	-29.41%*
sum	13.45%	14.37%	51.78%	36.00%	66.67%	82.67%	-20.73%*

In Table 3, we report performances of four aggregation operators which are max, min, sum and product operators. For these runs, documents performances are normalized so that they range in the $[0, 1]$ interval. The best performing run is the sum run, but its performances are significantly worse than those of mcm. This shows that a total compensatory logic (e.g., sum and prod runs) as well as a non compensatory logic (e.g., max and min runs) perform worse than a partial compensatory logic (e.g., mcm run) using outranking approaches for example.

We end this section by reporting performances of the official runs from TREC-13 [20] and compare our approach accordingly.

Table 4. Performance comparison with official runs

Run Id	AvP	R-p	r-r	S@1	S@5	S@10	Δ-AvP
mcm	17.08%	18.37%	58.04%	45.33%	74.67%	81.33%	–
uogWebCAU150	17.91%	20.30%	62.57%	50.67%	77.33%	89.33%	+4.84%
MSRAmixed1	17.80%	20.45%	52.79%	38.67%	72.00%	88.00%	+4.18%
MSRC04C12	16.45%	19.07%	53.39%	38.67%	74.67%	80.00%	-3.68%
humW04rdpl	16.28%	19.72%	55.31%	37.33%	78.67%	90.67%	-4.68%
THUIRmix042	14.66%	16.65%	39.54%	21.33%	58.67%	74.67%	-14.17%*
average	10.53%	12.84%	36.58%	23.87%	51.50%	61.82%	-38.37%*
median	11.52%	14.64%	39.99%	25.33%	58.00%	69.33%	-40.86%*

In Table 4, we first report performances of the best runs of the first five teams which participated to the track. Then, we computed average and median performances of all the submitted runs. From this table, we can see that mcm

has similar performances compared to those of the best ones. Moreover, `mcm` performs significantly better than the `average` or the `median` runs.

8 Conclusions

In this paper, we propose a multiple criteria framework for evidence combination where a set of candidate relevance criteria are proposed and used to determine how documents should be ranked using a set of decision rules.

From the first TREC experiments, this work seems to have the potential for high impact in the field of IR, given the possible application of evidence combination. It presents the advantage that it is applicable whatever is the collection under consideration provided that a pertinent criteria family is used. It also overcomes criteria heterogeneity problems by using a set of decision rules which are easy to grasp. Moreover, the proposed approach easily helps considering domain and context specific criteria in a natural way, rather than using complex formula which are difficult to interpret.

Approaches from multiple criteria decision theory, and especially outranking approaches, are generally used as an aid for decision makers. In the TREC context, there are various assessors judging documents with different and even conflicting preferences. This is the main explanation why it seems to be difficult to have significantly better performances. At the same time, we can outline an advantage of the proposed approach since we can easily carry out the study from the user perspective by setting a criteria family according to his/her preferences, giving rise to a personalized and valuable aid.

Future work will consist of additional experiments to strengthen the results. More specifically, applying our method in a human centered context would be an interesting extension of our work.

References

1. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM **18**(11) (1975) 613–620
2. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M., Gatford, M.: Okapi at trec-3. In: Proc. of TREC-3, Gaithersburg, Maryland, USA (1994)
3. Gao, G., Nie, J.Y., Bai, J.: Integrating word relationships into language models. In: SIGIR '05: Proc. of the 28th int. conf. on Research and development in information retrieval, New York, ACM Press (2005) 298–305
4. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM **46**(5) (1999) 604–632
5. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: WWW7: Proc. of the 7th int. conf. on World Wide Web, Amsterdam, The Netherlands, Elsevier (1998) 107–117
6. Fox, E.A., Shaw, J.A.: Combination of multiple searches. In: Proc. of TREC-3, Gaithersburg, Maryland, USA, NIST (1994)
7. Craswell, N., Robertson, S., Zaragoza, H., Taylor, M.: Relevance weighting for query independent evidence. In: SIGIR '05: Proc. of the 28th int. conf. on Research and development in information retrieval, New York, ACM Press (2005) 416–423

8. Frakes, W., Baeza-Yates, R.: Information Retrieval: Data Structures and Algorithms. Prentice Hall (1992)
9. Salton, G., Fox, E.A., Wu, H.: Extended boolean information retrieval. Commun. ACM **26**(11) (1983) 1022–1036
10. Dubois, D., Prade, H.: Criteria aggregation and ranking of alternatives in the framework of fuzzy set theory. In Zimmermann, H., Zadeh, L., Gaines, B., eds.: Fuzzy Sets and Decision Analysis. North-Holland, (1984) 209–240
11. Savoy, J., Rasolofo, Y.: Report on the trec-9 experiment: Link-based retrieval and distributed collections. In: Proc. of TREC-9, Gaithersburg, USA, NIST (2001)
12. Boughanem, M., Loiseau, Y., Prade, H.: Rank-ordering documents according to their relevance in information retrieval using refinements of ordered-weighted aggregations. In: AMR05, 3rd int. workshop on Adaptive multimedia retrieval, Glasgow, UK, Springer-Verlag (2005)
13. Anh, V.N., Moffat, A.: Vector space ranking: Can we keep it simple? In: Proc. of the Australian document computing symposium, Sydney (2002) 7–12
14. Borlund, P.: The concept of relevance in IR. J. Am. Soc. Inf. Sci. Technol. **54**(10) (2003) 913–925
15. Granka, L.A., Joachims, T., Gay, G.: Eye-tracking analysis of user behavior in www search. In: SIGIR '04: Proc. of the 27th int. conf. on Research and development in information retrieval, New York, ACM Press (2004) 478–479
16. Shannon, C.E.: Prediction and entropy of printed english. Bell Systems Technical Journal **30** (1951) 50–64
17. Roy, B.: Main sources of inaccurate determination, uncertainty and imprecision. Mathematical and Computer Modelling **12**(10-11) (1989) 1245–1254
18. Roy, B.: The outranking approach and the foundations of ELECTRE methods. Theory and Decision **31** (1991) 49–73
19. Roy, B., Hugonnard, J.: Ranking of suburban line extension projects on the Paris metro system by a multicriteria method. Transp. Research **16A**(4) (1982) 301–312
20. Craswell, N., Hawking, D.: Overview of the trec-2004 web track. In: Proc. of TREC-2004, Gaithersburg, Maryland, USA, NIST (2004)
21. Sanderson, M., Zobel, J.: Information retrieval system evaluation: effort, sensitivity, and reliability. In: SIGIR '05: Proc. of the 28th int. conf. on Research and development in information retrieval, New York, ACM Press (2005) 162–169

English to Persian Transliteration

Sarvnaz Karimi, Andrew Turpin, and Falk Scholer

School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Melbourne 3001, Australia
{sarvnaz, aht, fscholer}@cs.rmit.edu.au

Abstract. Persian is an Indo-European language written using Arabic script, and is an official language of Iran, Afghanistan, and Tajikistan. Transliteration of Persian to English—that is, the character-by-character mapping of a Persian word that is not readily available in a bilingual dictionary—is an unstudied problem. In this paper we make three novel contributions. First, we present performance comparisons of existing grapheme-based transliteration methods on English to Persian. Second, we discuss the difficulties in establishing a corpus for studying transliteration. Finally, we introduce a new model of Persian that takes into account the habit of shortening, or even omitting, runs of English vowels. This trait makes transliteration of Persian particularly difficult for phonetic based methods. This new model outperforms the existing grapheme based methods on Persian, exhibiting a 24% relative increase in transliteration accuracy measured using the top-5 criteria.

1 Introduction

Translating words of a text from a *source* language into a different *target* language can be efficiently achieved using a bilingual vocabulary, where every source word has a counterpart in the target language. In practice, however, there are often out-of-vocabulary (OOV) words that do not appear in the source dictionary. This is particularly common for proper nouns such as company, people, place and product names. In these cases *transliteration* must occur, where the OOV word is spelled out in the target language using character based methods.

Automatic transliteration of English OOV words has been studied for several languages, including Arabic [1], Korean [7,12], Chinese [14], Japanese [2,8,12], and the romantic languages [10,13]. Transliteration between English and Persian has not been previously studied.

Persian, also known as Farsi, is one of the oldest Indo-European languages, and is the official language of Iran, Afghanistan, and Tajikistan. Unlike many of the Indo-European languages (for example, English) it is not written in the Latin/Roman alphabet, but uses the same script as Arabic, with four additional characters: پ, چ, ژ, گ. So while at first glance it may seem that transliteration techniques that work for Arabic should work for Persian, the roots of the two languages are very different, and the phonetic interpretation of the Arabic script differs between the two languages. For example, ت and ط are pronounced

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 255–266, 2006.

differently in Arabic, but both are pronounced as "t" (as in "Peter") in Persian. If an Arabic name is being written in Persian, it is more likely that ط be preferred to ت; for words from other languages, authors may have an independent preference for either ت or ط, as they convey the same phonetic sound. These *redundant* characters make the transliteration of Persian a new and interesting problem to study.

Previous transliteration methods can be grouped into three main categories: grapheme-based approaches; phonetic-based approaches; and approaches that combine features of both. Grapheme-based (or *direct*) methods map groups of characters in the source word S directly to groups of characters in the target word T. Phonetic (or *pivot*) methods, on the other hand, first try and identify phonemes in the source word S, and then map those phonemes to character representations in the target language to get T, the target word. As the phonetic approaches have extra steps, they are in general more error prone than their grapheme-based counterparts, and typically success rates of phonetic based only approaches are lower than combined methods [2,12]. Given that grapheme based methods tend to be more successful than phoneme based methods, and that Persian words often omit vowels that would appear in their English counterparts making phonetic matching difficult, we adopt a grapheme based approach.

In this paper, we present results for transliterating English to Persian using a basic grapheme model as has been proposed for Arabic [1]. We extend this approach using more context characters from the source word S, and then introduce a more complex model to deal with some specifics of the Persian language. On a corpus of 1,857 proper names, our improved system achieves 66% accuracy when only one candidate transliteration is generated, compared with the baseline system at 48%. This is the first successful English to Persian transliteration system reported in the literature. We also discuss the problems in constructing an OOV corpus for Persian-English, and hypothesise that these problems will re-occur in many transliteration studies.

2 Background

In general, grapheme based transliteration is a three stage process.

Alphabet selection. In the first stage, an alphabet of source and target symbols, Σ_S and Σ_T, is created from a training corpus of known source-target word pairs. Note that symbols may be individual characters, or groups of characters, in the original alphabets. For example, the pair of English characters "ch" might be included as a single symbol in the source alphabet Σ_S. Symbol alphabets can be derived using statistical methods such as hidden Markov models [7] or translation models [3]. These have been implemented in the GIZA++ tool [11], which we use to obtain symbol alphabets for our experiments. An alternative approach

is to hand-craft the alphabets of allowable source and target graphemes, for example as adopted by AbdulJaleel and Larkey [1].

Probability generation. In the second stage, the training corpus of known source-target word pairs is used to collect statistics on the frequency of occurrence of chosen alphabet symbols. In general, we wish to compute the probability $P(t_i|s_i, c_i)$, which represents the probability of generating target symbol $t_i \in \Sigma_T$ given the source symbol $s_i \in \Sigma_S$ in the context c_i, which is a string drawn from $(\Sigma_T \cup \Sigma_S)$. In the most basic model, no context is used, so $c_i = \epsilon$, the empty string, and the transliteration is given simply by $P(t_i|s_i)$.

Use of context has been shown to improve transliteration accuracy. In his study of transliterating six romantic languages, Linden [10] use the two previous source symbols and one following source symbol as context to get $P(t_i|s_i, c_i = s_{i-2}s_{i-1}s_{i+1})$, which gave an improvement of 69% over a baseline technique. In similar work on Korean, Jung et al. [7] proposed taking advantage of past and future source symbols, and one past target language symbol, to compute $P(t_i|s_i, c_i = s_{i-2}s_{i-1}s_{i+1}t_{i-1})$, which gave a 5% improvement over their baseline.

In all methods that employ a non-empty context c_i, provision must be made for *backoff* to a smaller context. For example, if attempting to transliterate the symbol "o" in the word "lighthouse" in the context $c_i = s_{i-4}s_{i-3}s_{i-2}s_{i-1}$ = "ghth", it is likely that the context "ghth" occurred very infrequently in the training corpus, and so statistics derived in this context may not be reliable. In this case, a backoff scheme may try the context $c_i = s_{i-3}s_{i-2}s_{i-1}$ ="hth", which again may not occur frequently enough to be trusted, and so the next backoff context must be tried. This backoff method is used in the PPM data compression scheme [4].

Transliteration. The third stage of transliteration parses the source word S into symbols from Σ_S and computes, for possible target words T,

$$P(T|S) = P(T) \prod_{i=1}^{|T|} P(t_i|s_i, c_i), \tag{1}$$

where the first term $P(T)$ is a *target language model*, that is, the probability of the target word T appearing independently of any source information. For example, the word "zxqj" is extremely unlikely to appear as the English transliteration of any word in any language, and so $P($"zxqj"$)$ would be very small. Probability $P($"the"$)$, on the other hand, may sit at around 0.01. Distribution $P(T)$ can be computed using a *zero-order* model from the training corpus as $P(T) = \prod_{i=1}^{|T|}(\text{freq}(t_i)/D)$, where D is the total number of characters in the training corpus; and freq gives the frequency of its argument in the training corpus. In our experiments below we used a *first-order* model where $P(T) = \prod_{i=2}^{|T|}(\text{freq}(t_{i-1}t_i)/\text{freq}(t_i))$.

The target words can then be sorted by $P(T|S)$, given in Equation 1, to give a ranked list of the most likely transliterations of S.

3 Transliteration Methods

In this section we describe our implementation of the transliteration process. First we introduce baseline systems, followed by an explanation of our novel transliteration approach, where contexts respect vowel-consonant boundaries.

3.1 Baseline Systems

Alphabet selection. In order to select the source and target alphabets, Σ_S and Σ_T, we made use of the character-level alignment of source and target word pairs produced by GIZA++ [11]. If this resulted in a group of characters mapping to a single character in either direction, then the group was added as a single symbol to Σ_S or Σ_T, as required. Consider an example, where the word "stella" is aligned between Persian and English characters:

Source (English):	s	t	e	ll	a
Target (Persian, left-to-right):	اس	ت	ϵ	ل	ا

Here, both "te" and "ll" would be added to Σ_S, as they map to a single Persian character (ϵ represents a null character). The symbol "اس" would be added to Σ_T as these two Persian characters align with only one English character.

Probability generation. Previous Arabic transliteration systems use an empty context [1], whereas successful transliteration systems for romantic languages (arguably more close to Persian, in spite of script) use $c_i = s_{i-2}s_{i-1}s_{i+1}$ for transliterating s_i [10]. Accordingly, we tested a variety of contexts based on past and future source symbols.

For brevity, we introduce the notation $n\backslash m$ to indicate that n previous source symbols and m future source symbols make up a context, that is:

$$c_i = n\backslash m = s_{i-n} \dots s_{i-1}s_{i+1} \dots s_{i+m}$$

In order to avoid boundary conditions, we also assume that S is extended to the left and right as far as is required with a special symbol that always transliterates to an empty symbol ϵ. Therefore, $[-n, \dots, |S| + m]$ are all valid indexes into S.

During this probability generation phase, frequency counts for each mapping contained in the aligned words of the training set are gathered. We define

$$f(t_i, s_i, n, m) = \text{frequency of } t_i \text{ aligning with } s_i \text{ in context } n\backslash m.$$

Once all frequencies are gathered, they are simply normalised into probabilities to get

$$P(t_i|s_i, c_i = n\backslash m) = f(t_i, s_i, n, m) \Big/ \sum_{\forall j} f(t_j, s_i, n, m).$$

1) Let ℓ be the length of the longest context in characters to the left,
   r be the length of the longest context in characters to the right, and
   $S = [s_1 \dots s_{|S|}]$ be the input word made up of $|S|$ characters.
2) Set $X \leftarrow$ the empty set.
3) For $i \leftarrow 1$ to $|S|$ do
4) Let $p \leftarrow \ell$ and $q \leftarrow r$.
5) While $p > 0$, $q > 0$ and context $S[i - p \dots i + q]$ has a frequency $< M$
6) Set $p \leftarrow p - 1$ and set $q \leftarrow q - 1$.
7) If $p > 0$ or $q > 0$ then
8) Append all possible transliterations of s_i in context
    $S[i - \max(p, 0) \dots i + \max(q, 0)]$ to the elements of X
    and record the associated probabilities of resulting words.
9) else
    Append all possible transliterations of s_i without context
    to the elements of X and record associated probabilities.
10) Sort X and output transliterations.

Fig. 1. The transliteration algorithm.

Transliteration. While source symbols are readily identified in the aligned words of the training corpus, when presented with a source word alone for transliteration there may be several parsings possible using elements from the source alphabet Σ_S. One possible approach is to greedily choose the longest matching source symbol from left-to-right, and then use this parsing throughout the transliteration process [1]. An alternate is to define a *source language model* which will apply probabilities to various parsings, and work this probability into Equation 1 for $P(T|S)$ [7].

The approach we have taken here is to not commit to a single parsing; instead, we process the source character by character. Similarly the backoff process for contexts is on a character by character basis, rather than a symbol by symbol basis. Note that this requires us to keep track of the number of characters generated in the longest context of the probability generation phase; while it is known that this context will contain $n + m$ symbols, this will map to $\ell + r$ characters, where ℓ is the maximum number of characters in the n previous symbols (to the ℓeft), and r is the maximum number of characters in the m future symbols (to the $r$ight).

A high level description of the transliteration algorithm is given in Figure 1. An attempt is made to transliterate each character in the longest possible context, with the context reducing by one character each time at both ends until the context occurs at least M times in the training data (Steps 5 and 6). Once a context is found, all possible target symbols t_i that arise from the character s_i in that context are incorporated into the set of transliterations so far, X. If a context is not found, then a straight mapping of the individual character without context is carried out (Step 9). Since there are $|\Sigma_T|^{|T|}$ possible target words, it is not practical to exhaustively generate all variants as is shown in Steps 8 and 9. In our implementation we use the k-shortest paths algorithm due to Dijkstra [5],

which greedily expands prefixes of strings in X with the highest probabilities, and therefore guarantees that the k highest scoring transliterations are found.

There are several alternate implementations of Step 5 and 6 for reducing contexts. The approach we describe in Figure 1, where both past (p) and future (q) horizons are shrunk, we call BACKOFF-A. The second approach with which we experimented was first fully reducing future contexts, and only when q reaches zero do we begin to reduce past contexts (p). We label this approach BACKOFF-B. In all experiments we used $M = 2$.

3.2 Collapsed-Vowel Models

Purely selecting contexts according to a $n\backslash m$ model as described in the previous section ignores any characteristics that we may know of the source and target languages. For instance, Persian speakers tend to transliterate diphthongs (vowel sounds that start near the articulatory position for one vowel and moves toward the position for another) of other languages to monophthongs (two written vowels that represent a single sound). In addition, short vowel sounds in English are often omitted altogether in Persian. This suggests a model where runs of English vowels are transliterated in the context of their surrounding consonants. Rather than employ a full natural language analysis technique, as has been done for Chinese [14], we opt for a simpler segmentation approach and propose the following *collapsed-vowel* models.

Alphabet selection. We define Σ_S and Σ_T as in the baseline systems: composite symbols are identified by aligning words using the statistical translation model implemented in GIZA++.

Once this initial parsing is complete, we re-segment the words using consonant and vowel boundaries to define new symbols that we add to the existing alphabets. Each source word is parsed into a sequence of single consonants, represented by C, and runs of consecutive vowels represented by V. These are then grouped into *segments* in one of four ways:

1. VC, a run of vowels at the beginning of a word followed by a consonant;
2. CC, two consonants not separated by any vowels;
3. CVC, a run of vowels bounded by two consonants; and
4. CV, a run of vowels at the end of a word preceded by a consonant.

The first consonant in each segment overlaps the final consonant in the preceding segment. In each case the full composite symbol is added to the source alphabet (to later be used as context), and the composite symbol without the tailing consonant is added to both alphabets (to be used for transliteration).

For example, consider the word "baird". This would first be parsed as:

Source (English):	b	ai	r	d
Target (Persian, left-to-right):	ب	ی	ر	د

resulting in "ai" being added to Σ_S (as in the baseline system).

i	S	Segment	Method CV $s_i \vert c_i$	Method CV-BROAD $s_i \vert c_i$
1	e n	VC	e|en	e|eC
2	n r	CC	n|nr	n|CC
3	r i q	CVC	ri|riq	r|C and i|CiC
4	q u e	CV	que|que	q|C and ue|Cue

Fig. 2. Example of the segments, symbols and contexts generated for the source word "enrique"

In the second phase, the word would be parsed into segments as:

Source (English):	b	ai	r	d
Target (Persian, left-to-right):	ب	ی	ر	د
Segmentation:	C	V	C	C
	Segment 1			
			Segment 2	

resulting in the possible contexts "bair" and "rd" being added into Σ_S as a CVC and CC segments respectively. The additional source symbol "bai", derived by dropping the C from CVC is added to Σ_S, and "r" is already in Σ_S. In the target alphabet, ی ب is added as a truncated CVC target symbol, and the single character "ر" is already in Σ_T.

Probability generation. Probabilities are generated from frequency counts in the training data as for the baseline systems, but the contexts are now chosen according to segments, rather than runs of symbols (graphemes). We use two techniques for selecting contexts. In the first, labeled CV, each segment forms a context for the symbol that prefixes the segment. For example, the fourth column of Figure 2 shows how the contexts and symbols for the source word "enrique" are derived.

In the second technique for choosing contexts, CV-BROAD, we relax the strict matching on the consonant component of the context so that it can match all consonants, thus increasing the amount of training data for a context. In turn, this leads to rare, but existent, transliterations that would otherwise not be available. Vowels are only used in a context when transliterating a symbol that contains a vowel. This is shown in the final column of Figure 2 for the example string "enrique".

The backoff approach for each of the four segments is given by the following three rules, which are applied in order:

1. if the context is a CV segment, and does not occur at least M times in the training data, separate the symbol into a C then V context; else
2. if the context is a V segment representing a run of r vowels, and does not occur at least M times in the training data, reduce the length of the run to $r-1$ and transliterate the final vowel as a single character with no context; else

3. for all other contexts, if it occurs less than M times in the training data, drop the tailing C and try this shorter context.

Transliteration. The transliteration proceeds as described for the baseline systems in Section 3.1, where Dijkstra's k-shortest path is used to generate the k highest scoring transliterations.

4 Experiments

In this section we describe our experimental framework, including the data used for training and testing purposes, the transliteration models that are evaluated, and metrics used to quantify performance.

4.1 Data

The first two stages of transliteration require a training corpus of source and target word pairs. An English-Persian corpus was developed by taking 40,000 phrases from lists of English names that occurred on the World Wide Web. These names were names of geographical places, persons and companies, extracted from different publicly available resources. This English corpus was then transliterated by 26 native Persian speakers. Redundant and allophone characters were often used by the transliterators for Arabic words of the collection. Depending on the transliterator's assumptions about the origin of a word, different characters may have been chosen for the same word. For example, if the transliterator assumed "John Pierre" was French, then for the character "j" the transliterator might have employed the rare Persian character ژ ,pronounced same as "su" in measure, instead of widely used ج which sounds as "j" in jet. Finally the corpus was split into words, and the unique word pairs extracted, resulting in 16,952 word pairs. We refer to this collection as ENGLISH+.

The corpus selection process did not guarantee that all included words were of English origin. As a result, there are many words in the corpus that are already transliterated from other languages such as French, Arabic, and Chinese. We randomly chose a subset of 2,000 name pairs from the ENGLISH+ collection, from which all names with origins from non-English script languages were removed. This resulted a sub-collection of 1,857 name pairs, labelled ENGLISH.

In our experiments we apply ten-fold cross-validation; so for any one run, 90% of a collection is used as training data, with the remaining 10% being used as test data, and results are averages over the 10 possible runs.

4.2 Models

In our experiments we consider baseline transliteration models using a variety of context sizes and backoff options, as well as our novel collapsed-vowel models.

For the baseline models, we tested combinations of past (n) and future (m) contexts $n\backslash m$, using the following settings: 0\0, 1\0, 2\0, 0\1, 0\2, and 2\1. These contexts were all run with backoff method BACKOFF-B and a uniformly distributed target language model. Also included is the first-order target language model described in Section 2 for 0\0 and 1\0, denoted as 0\0T and 1\0T. We also experimented with BACKOFF-A for contexts 1\1, 1\1 and 2\1 using a uniform target language model, denoted 1\1A, 2\2A and 2\1A.

These models were run on both the ENGLISH and ENGLISH+ collections.

4.3 Metrics

The results of our transliteration experiments are evaluated using the standard measure of word accuracy, based on the edit distance between the transliterated word and the correct word. The edit distance measures the number of character insertions, deletions and substitutions that are required to transform one word into another [9].

Word accuracy (WA), also known as transliteration accuracy, measures the proportion of transliterations that are correct:

$$WA = \frac{Number\ of\ transliterations\ with\ ED{=}0}{Total\ number\ of\ test\ words}$$

where, ED is the edit distance between two strings [6]. WA is reported for different cutoff values. For example, *top-1* WA indicates the proportion of words in the test set for which the correct transliteration was the first candidate answer returned by the transliteration system, while *top-5* indicates the proportion of words for which the correct transliteration was returned within the first five candidate answers.

Where statistical tests are performed on accuracy, the test employed is a paired t-test on the accuracy-percentages calculated for each step of the 10-fold cross validation runs.

5 Results

In this section we present the results of our experiments into the effects of context size, target language models, and collapsed vowel models. In the tables that follow, the results of the most accurate methods are highlighted in bold.

Past and Future Context. The results of using varying past (n) and future (m) context windows are shown in Table 1. The highest performance is achieved using a context window of 1\0, that is, using a history of just one character. The difference in performance between this context and using no context is statistically significant ($p < 0.002$ for *top-1*).

Extending the historical context causes performance to deteriorate, compared to using just one historical symbol. Similarly, the addition of future context causes performance to fall when past context is also present. Using the symmetrical backoff method, BACKOFF-A, rather than BACKOFF-B, improves performance

Table 1. Mean word accuracy (%) for changing past and future context sizes

		0\0	1\0	2\0	0\1	0\2	1\1*A*	2\2*A*	2\1	2\1*A*
ENGLISH	*Top-1*	47.8	**59.2**	49.5	43.1	54.6	54.5	27.6	47.8	38.4
	Top-5	73.7	**84.9**	73.9	70.5	76.3	79.6	41.7	72.3	65.0
	Top-10	79.9	**89.3**	77.4	75.1	78.8	83.3	46.1	78.0	71.9
ENGLISH+	*Top-1*	9.0	**45.7**	15.1	32.6	12.8	31.7	6.5	9.3	22.5
	Top-5	11.7	**71.3**	23.6	60.1	21.6	55.7	14.1	14.9	41.9
	Top-10	16.6	**77.2**	26.2	68.5	23.6	63.2	17.3	16.9	51.1

Table 2. Mean word accuracy (%) when using a target language model (T) and collapsed vowel schemes (CV and CV-BROAD)

		0\0	0\0*T*	1\0	1\0*T*	CV	CV+
ENGLISH	*Top-1*	47.8	34.8	59.2	55.1	**66.4**	64.9
	Top-5	73.7	45.1	84.9	79.7	84.4	**87.3**
	Top-10	79.9	50.6	89.3	84.3	88.5	**95.0**
ENGLISH+	*Top-1*	9.0	3.3	45.7	40.8	**50.1**	49.5
	Top-5	11.7	4.6	71.3	63.6	74.0	**80.8**
	Top-10	16.6	5.2	77.2	69.3	79.4	**88.0**

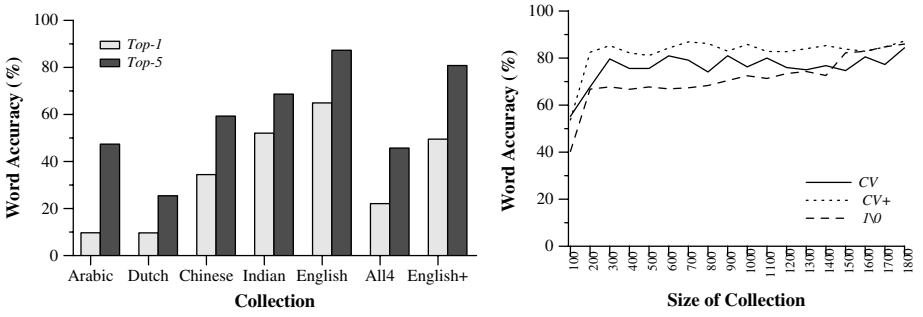
for the larger ENGLISH+ collection when both past and future context is used in the transliteration model. However, overall performance here is still significantly worse than when using a context of 1\0 ($p < 0.005$).

Target Language Models. The effect of introducing a target language model is shown in Table 2; schemes using a first-order target language model are labeled with a "T". It can be seen that using a target language model has a detrimental effect on transliteration performance for both 0\0 and 1\0 contexts.

Collapsed Vowel Approaches. The results for our novel collapsed vowel models are shown in the final columns of Table 2, labeled CV and CV-BROAD. The CV scheme shows the best performance for *top-1* transliteration for both the ENGLISH and ENGLISH+ collections, an improvement that is statistically significant over the best 1\0 baseline ($p < 0.008$). For *top-5* and *top-10* word accuracy, the CV-BROAD model shows highest performance.

Language-based Effects. The large differences between the results on the ENGLISH and ENGLISH+ collections suggest that, although a larger training collection could assist the transliteration process, the origin of the source language could have a deleterious impact on accuracy.

Figure 5(a) shows the accuracy of the CV approach on different sub-collections that we extracted from the ENGLISH+ collection. A word was assigned to a country according to its appearance on WWW pages of that country. For example, "Groenendijk" appears in many pages of the domain `.nl`, and so was assigned



(a) Different language origins (Top-1) (b) Different sizes (Top-5)

Fig. 3. Mean word accuracy of different collections

to the Dutch category. Note that the accuracy is substantially reduced for all languages individually and altogether ("All4").

Collection Size Effects. One possible reason for CV performing poorly on the country based data sets is that they are small datasets. To investigate this possibility, we randomly partitioned ENGLISH into sets of increasing size and computed the accuracy of CV on those partitions.

The results are shown in Figure 5(b). As can be seen, any data set above 200 words performed well, and so it would seem that the small size of the individual country datasets (all are larger than 200) is not to blame. There appears to be a genuine difference between the performance of CV on native English and non-English words that have already been transliterated from another language.

6 Conclusions and Future work

In this paper, we have investigated a variety of transliteration techniques and their effectiveness in transliterating words from English to Persian. We investigated the performance of existing grapheme-based approaches, and examined a range of context and target language model options. The use of context is important: using a small historical context (one past symbol) makes transliteration significantly more effective than using no context. However, using larger context sizes, or using future context, has a detrimental effect. This effect is surprising; in general a larger context should improve the predictive power of the model. The effect may be due in part to our symbol parsing approach. In future work, we plan to investigate whether incorporating a full source language model into our transliteration process can extend the range of useful contexts.

A new transliteration model of Persian, based on collapsing runs of vowels, was introduced and evaluated. Our novel techniques increased accuracy over standard grapheme-based approaches with no context by around 18% on a carefully chosen native English data set (66% up from 48% for *top-1*), and by 41% on a more general English data set that was not filtered (50% up from 9% for *top-1*). In

future work, we will evaluate our new models on other languages with similar vowel transliteration patters as Persian, such as Arabic and Indonesian.

Investigating the effects of including different source languages in training data demonstrated that this introduces noise, and can have a substantial negative impact on transliteration performance. Previous work has generally avoided this problem by using small test sets that are carefully controlled. As far as we are aware, this is the first work on transliteration that has examined the effects of data quality on accuracy. Our future research will be to further investigate and quantify the noise that is generated as part of this phenomenon.

Acknowledgments

This work was supported in part by Australian Research Council Grant DP055-8916 (AT) and the Australian government IPRS program (SK).

References

1. Nasreen AbdulJaleel and Leah S. Larkey. Statistical transliteration for English-Arabic cross language information retrieval. In *CIKM*, pages 139–146, 2003.
2. Slaven Bilac and Hozumi Tanaka. Direct combination of spelling and pronunciation information for robust back-transliteration. In *CICLing*, pages 413–424, 2005.
3. Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computional Linguistics*, 19(2):263–311, 1993.
4. John G. Cleary and Ian H. Witten. A comparison of enumerative and adaptive codes. *IEEE Transactions on Information Theory*, 30(2):306–315, 1984.
5. David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.
6. Patrick A. V. Hall and Geoff R. Dowling. Approximate string matching. *ACM Comput. Surv.*, 12(4):381–402, 1980.
7. Sung Young Jung, Sung Lim Hong, and Eunok Paek. An English to Korean transliteration model of extended markov window. In *COLING*, pages 383–389, 2000.
8. Kevin Knight and Jonathan Graehl. Machine transliteration. *Computational Linguistics*, 24(4):599–612, 1998.
9. Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965.
10. Krister Linden. Multilingual modeling of cross-lingual spelling variants. *Inf. Retrieval*, 9(3):295–310, 2005.
11. Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, 2003.
12. Jong-Hoon Oh and Key-Sun Choi. An ensemble of transliteration models for information retrieval. *Inf. Process. Manage.*, 42(4):980–1002, 2006.
13. Jarmo Toivonen, Ari Pirkola, Heikki Keskustalo, Kari Visala, and Kalervo Järvelin. Translating cross-lingual spelling variants using transformation rules. *Inf. Process. Manage.*, 41(4):859–872, 2005.
14. Stephen Wan and Cornelia Verspoor. Automatic English-Chinese name transliteration for development of multilingual resources. In *COLING-ACL*, pages 1352–1356, 1998.

Efficient Algorithms for Pattern Matching with General Gaps and Character Classes

Kimmo Fredriksson[1,⋆] and Szymon Grabowski[2]

[1] Department of Computer Science, University of Joensuu
PO Box 111, FIN–80101 Joensuu, Finland
kfredrik@cs.joensuu.fi
[2] Technical University of Łódź, Computer Engineering Department,
Al. Politechniki 11, 90–924 Łódź, Poland
sgrabow@kis.p.lodz.pl

Abstract. We develop efficient dynamic programming algorithms for a pattern matching with general gaps and character classes. We consider patterns of the form $p_0 g(a_0, b_0) p_1 g(a_1, b_1) \dots p_{m-1}$, where $p_i \subset \Sigma$, where Σ is some finite alphabet, and $g(a_i, b_i)$ denotes a gap of length $a_i \dots b_i$ between symbols p_i and p_{i+1}. The text symbol t_j matches p_i iff $t_j \in p_i$. Moreover, we require that if p_i matches t_j, then p_{i+1} should match one of the text symbols $t_{j+a_i+1} \dots t_{j+b_i+1}$. Either or both of a_i and b_i can be negative. We give algorithms that have efficient average and worst case running times. The algorithms have important applications in music information retrieval and computational biology. We give experimental results showing that the algorithms work well in practice.

1 Introduction

Background. Many notions of approximateness have been proposed in string matching literature, usually motivated by some real problems. One of seemingly underexplored problem with applications in music information retrieval (MIR) and molecular biology (MB) is *pattern matching with gaps* [3]. In this problem, gaps (text substrings) of length up to α are allowed between each pair of matching pattern characters. Moreover, in MIR applications the character matching can be relaxed with δ-matching, i.e. the pattern character matches if its numerical value differs at most by δ the corresponding text character. In MB applications the singleton characters can be replaced by classes of characters, i.e. text character t matches a pattern charater p if $t \in p$, where p is some subset of the alphabet.

Previous work. The first algorithm for the problem [3] is based on dynamic programming, and runs in $O(nm)$ time, where n and m are the lengths of the text and pattern, respectively. This basic dynamic programming solution can also be generalized to handle more general gaps while keeping the $O(nm)$ time

⋆ Supported by the Academy of Finland, grant 202281.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 267–278, 2006.

bound [11]. The basic algorithm was later reformulated [1] to allow to find all pattern occurrences, instead of only the positions where the occurrence ends. This needs more time, however. The algorithm in [2] improves the average case of the one in [1] to $O(n)$, but they assume constant α. Bit-parallelism can be used to improve the dynamic programming based algorithm to run in $O(\lceil n/w \rceil m + n\delta)$ and $O(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ time in worst and average case, respectively, where w is the number of bits in a machine word, and σ is the size of the alphabet [4].

For the α-matching with classes of characters there exists an efficient bit-parallel nondeterministic automaton solution [10]. This also allows gaps of different lengths for each pattern character. This algorithm can be trivially generalized to handle (δ, α)-matching [2], but the time complexity becomes $O(n\lceil \alpha m/w \rceil)$ in the worst case. For small α the algorithm can be made to run in $O(n)$ time on average. The worst case time can be improved to $O(n\lceil m \log(\alpha)/w \rceil)$ [4], but this assumes equal length gaps.

Sparse dynamic programming can be used to solve the problem in $O(n + |\mathcal{M}| \min\{\log(\delta + 2), \log\log m\})$ time, where $\mathcal{M} = \{(i, j) \mid |p_i - t_j| \leq \delta\}$ (and thus $|\mathcal{M}| \leq nm$) [6]. This can be extended for the harder problem variant where transposition invariance and character insertions, substitutions or mismatches are allowed together with (δ, α)-matching [7].

Our results. We develop several sparse dynamic programming algorithms. Our first algorithm is essentially a reformulation of the algorithm in [7]. The worst case running time of the algorithm is $O(n + |\mathcal{M}|)$. Our variant has the benefit that it generalizes in straight-forward way to handle general and even negative gaps, important in some MB applications [8,9]. We then give several variants of this algorithm to improve its average case running time to close to linear, while increasing the worst case time only up to $O(n + |\mathcal{M}| \log(|\mathcal{M}| + \alpha))$. This algorithm assumes fixed integer alphabet. We also present two simple and practical algorithms that run in $O(n)$ time on average for $\alpha = O(\sigma/\delta)$, but have $O(n + \min(nm, |\mathcal{M}|\alpha))$ worst case time, for any unbounded real alphabets.

These are the first algorithms that achieve good average and worst case complexities simultaneously, and they are shown to perform well in practice too.

2 Preliminaries

Let the pattern $P = p_0 p_1 p_2 \ldots p_{m-1}$ and the text $T = t_0 t_1 t_2 \ldots t_{n-1}$ be numerical strings, where $p_i, t_j \in \Sigma$ for some integer alphabet Σ of size σ. The number of distinct symbols in the pattern is denoted by σ_p.

In δ-approximate string matching the symbols $a, b \in \Sigma$ match, denoted by $a =_\delta b$, iff $|a - b| \leq \delta$. Pattern P (δ, α)-matches the text substring $t_{i_0} t_{i_1} t_{i_2} \ldots t_{i_{m-1}}$, if $p_j =_\delta t_{i_j}$ for $j \in \{0, \ldots, m-1\}$, where $i_j < i_{j+1}$, and $i_{j+1} - i_j \leq \alpha + 1$. If string A (δ, α)-matches string B, we sometimes write $A =_\delta^\alpha B$.

In all our analyses we assume uniformly random distribution of characters in T and P, and constant α and δ/σ, unless otherwise stated. Moreover, we

often write δ/σ to be terse, but the reader should understand that we mean $(2\delta + 1)/\sigma$, which is the upper bound for the probability that two randomly picked characters match.

The dynamic programming solution to (δ, α)-matching is based on the following recurrence [3,1]:

$$D_{i,j} = \begin{cases} j & t_j =_\delta p_i \text{ AND } (i = 0 \text{ OR } (i, j \geq 1 \text{ AND } D_{i-1,j-1} \geq 0)) \\ D_{i,j-1} & t_j \neq_\delta p_i \text{ AND } j > 0 \text{ AND } j - D_{i,j-1} < \alpha + 1 \\ -1 & \text{otherwise} \end{cases} \quad (1)$$

In other words, if $D_{i,j} = j$, then the pattern prefix $p_0 \dots p_i$ has an occurrence ending at text character t_j, i.e. $p_i =_\delta t_j$ and the prefix $p_0 \dots p_{i-1}$ occurs at position $D_{i-1,j-1}$, and the gap between this position and the position j is at most α. If $p_i \neq_\delta t_j$, then we try to extend the match by extending the gap, i.e. we set $D_{i,j} = D_{i,j-1}$ if the gap does not become too large. Otherwise, we set $D_{i,j} = -1$. The algorithm then fills the table $D_{0\dots m-1,0\dots n-1}$, and reports an occurrence ending at position j whenever $D_{m-1,j} = j$. This is simple to implement, and the algorithm runs in $O(mn)$ time using $O(mn)$ space.

We first present efficient algorithms to the above problem, and then show how these can be generalized to handle arbitrary gaps, tackling with both upper and lower bounded gap lengths, and even negative gap lengths, and using general classes of characters instead of δ-matching.

3 Row-Wise Sparse Dynamic Programming

The algorithm we now present can be seen as a row-wise variant of the sparse dynamic programming algorithm of the algorithm in [7, Sec. 5.4]. We show how to improve its average case running time. Our variant can also be easily extended to handle more general gaps, see Sec. 6.

3.1 Efficient Worst Case

From the recurrence of D it is clear that the interesting computation happens when $t_j =_\delta p_i$, and otherwise the algorithm just copies previous entries of the matrix or fills some of the cells with a constant.

Let $\mathcal{M} = \{(i, j) \mid p_i =_\delta t_j\}$ be the set of indexes of the δ-matching character pairs in P and T. For every $(i, j) \in \mathcal{M}$ we compute a value $d_{i,j}$. For the pair (i, j) where $d_{i,j}$ is defined, it corresponds to the value of $D_{i,j}$. If $(i, j) \notin \mathcal{M}$, then $d_{i,j}$ is not defined. Note that $d_{m-1,j}$ is always defined if P occurs at $t_{h\dots j}$ for some $h < j$. The new recurrence is

$$d_{i,j} = j \mid (i-1, j') \in \mathcal{M} \text{ AND } 0 < j - j' \leq \alpha + 1 \text{ AND } d_{i-1,j'} \neq -1,$$

and -1 otherwise. Computing the d values is easy once $\mathcal{M}$ is computed. As we have an integer alphabet, we can use table look-ups to compute $\mathcal{M}$ efficiently. Instead of computing $\mathcal{M}$, we compute lists $L[p_i]$, where $L[p_i] = \{j \mid p_i =_\delta t_j\}$.

These are obtained by scanning the text linearly, and inserting j into each list $L[p_i]$ such that t_j δ-matches t_j. Clearly, there are at most $O(\delta)$ and in average only $O(\delta\sigma_{\text{P}}/\sigma)$ symbols p_i that δ-match t_j. Therefore this can be obtained in $O(\delta n)$ worst case time, and the average case complexity is $O(n(\delta\sigma_{\text{P}}/\sigma + 1))$. Note that $|\mathcal{M}|$ is $O(mn)$ in the worst case, but the total length of all the lists is at most $O(\min\{\sigma_{\text{P}}, \delta\}n)$, hence L is a compact representation of $\mathcal{M}$. The indexes in $L[p_i]$ will be in increasing order.

Consider a row-wise computation of d. The values of the first row $d_{0,j}$ correspond one to one to the list $L[p_0]$, that is, the text positions j where $p_0 =_\delta t_j$. The subsequent rows d_i correspond to $L[p_i]$, with the additional constraint that $j - j' \leq \alpha + 1$, where $j' \in L[p_{i-1}]$ and $d_{i-1,j'} \neq -1$. Since the values in $L[p_i]$ and d_{i-1} are in increasing order, we can compute the current row i by traversing the lists $L[p_i]$ and d_{i-1} simultaneously, trying to enforce the condition that $L[p_i][h] - d_{i-1,k} \leq \alpha + 1$ for some h, k. If the condition cannot be satisfied for some h, we store -1 to $d_{i,h}$, otherwise we store the text position $L[p_i][h]$. The algorithm traverses L and $\mathcal{M}$ linearly, and hence runs in $O(n + |\mathcal{M}|)$ worst case time. We now consider improving the average case time of this algorithm.

3.2 Efficient Average Case

The basic sparse algorithm still does some redundant computation. To compute the values $d_{i,j}$ for the current row i, it laboriously scans through the list $L[p_i]$, for all positions, even for the positions close to where $p_0 \ldots p_{i-1}$ did not match. In general, the number of text positions with matching pattern prefixes decreases exponentially on average when the prefix length i increases. Yet, the list length $|L[p_i]|$ will stay approximately the same. The goal is therefore to improve the algorithm so that its running time per row depends on the number of matching pattern prefixes on that row, rather than on the number of δ-matches for the current character on that row.

The modifications are simple: (1) the values $d_{i,j} = -1$ are not maintained explicitly, they are just not stored since they do not affect the computation; (2) the list $L[p_i]$ is not traversed sequentially, position by position, but binary search is used to find the next value that may satisfy the condition that $L[p_i][h] - d_{i-1,k} \leq \alpha + 1$ for some h, k.

Consider now the average search time of this algorithm. The average length of each list $L[p_i]$ is $O(n\delta/\sigma)$. Hence this is the time needed to compute the first row of the matrix, i.e. we just copy the values in $L[p_0]$ to be the first row of d. For the subsequent rows we execute one binary search over $L[p_i]$ per each stored value in row i of the matrix. Hence in general, computing the row i of the matrix takes time $O(|d_{i-1}| \log(n\delta/\sigma))$, where $|d_i|$ denotes the number of stored values in row i. For $i > 0$ this decreases exponentially as $|d_i| = O(n(\delta/\sigma) \times \gamma^i)$, where $\gamma = 1 - (1 - \delta/\sigma)^{\alpha+1} < 1$ is the probability that a pattern symbol δ-matches in a text window of length α symbols. Summing up the resulting geometric series over all rows we obtain $O(n\frac{\delta}{\sigma(1-\delta/\sigma)^{\alpha+1}})$, which is $O(n\alpha\delta/\sigma)$ for $\delta/\sigma < 1-\alpha^{-1/(\alpha+1)}$. In particular this is $O(n)$ for $\alpha = O(\sigma/\delta)$. Hence the average search time is $O(n + n\alpha\delta/\sigma \log(n\delta/\sigma))$. However, the worst case search time is

also increased to $O(n+|\mathcal{M}|\log(|\mathcal{M}|/m))$. We note that this can be improved to $O(n+|\mathcal{M}|\log\log((mn)/|\mathcal{M}|))$ by using efficient priority queues [5].

3.3 Faster Preprocessing

The $O(\delta n)$ (worst case) preprocessing time can dominate the average case search time in some cases. Note however, that the preprocessing time can never exceed $O(n+|\mathcal{M}|)$. We now present two methods to improve the preprocessing time. The first one reduces the worst case preprocessing cost to $O(\sqrt{\delta}n)$, and improves its average case as well. The second method achieves $O(n)$ preprocessing time, but the worst case search time is slightly increased.

***O*($\sqrt{\delta}n$) time preprocessing.** The basic idea is to partition the alphabet into $\sigma/\sqrt{\delta}$ disjoint intervals $I_h, h=0\ldots\sigma/\sqrt{\delta}-1$ of size $\sqrt{\delta}$ each (w.l.o.g. we assume that δ is a square number and $\sqrt{\delta}$ divides σ). Then, for each alphabet symbol s, its respective $[s-\delta, s+\delta]$ interval wholly covers $\Theta(\sqrt{\delta})$ intervals I_h, and also can partially cover at most two I_h intervals. Two kinds of lists are computed in the preprocessing, L_{B} (for "boundary" cases) and L_{C} (for "core"). For each character of T, at most $2(\sqrt{\delta}-1)$ lists $L_{\mathrm{B}}[p_i]$ are extended with one entry, and those correspond to the alphabet symbols from the partially covered intervals I_h. But also each character t_j causes to append j to $O(\sqrt{\delta})$ lists $L_{\mathrm{C}}[p_i/\sqrt{\delta}]$, those that correspond to the I_h intervals wholly covered by $[t_j-\delta, t_j+\delta]$. Clearly, the preprocessing is done in $O(\sqrt{\delta}n)$ worst case and in $O(n\sqrt{\delta}\sigma_{\mathrm{r}}/\sigma)$ average time.

The search is again based on a binary search routine, but in this variant we binary search two lists: $L_{\mathrm{B}}[p_i]$ and $L_{\mathrm{C}}[p_i/\sqrt{\delta}]$, as the δ-matches to p_i may be stored either at some L_{B}, or at some L_{C} list. This increases both the average and worst case search cost only by a constant factor.

We can generalize this idea and have a preprocessing/search trade-off. Namely, we may have k levels, turning the preprocessing cost into $O(k\delta^{1/k}n)$, for the price of a multiplicative factor k in the search. For $k=\log\delta$ the preprocessing cost becomes $O(n\log\delta)$, and both the average and worst case search times are multiplied by $\log\delta$ as well.

***O*(n) time preprocessing.** We partition the alphabet into $\lceil\sigma/\delta\rceil$ disjoint intervals of width δ. With each interval a list of character occurrences will be associated. Namely, each list $L[i], i=0\ldots\lceil\sigma/\delta\rceil-1$, corresponds to the characters $i\delta\ldots\min\{(i+1)\delta-1, \sigma-1\}$. During the scan over the text in the preprocessing phase, we append each index j to up to three lists: $L[k]$ for such k that $k\delta\le t_j\le(k+1)\delta-1$, $L[k-1]$ (if $k-1\ge 0$), and $L[k+1]$ (if $k+1\le\lceil\sigma/\delta\rceil-1$). Note that no character from the range $[t_j-\delta\ldots t_j+\delta]$ can appear out of the union of the three corresponding intervals. Such preprocessing clearly needs $O(n)$ space and time in the worst case.

Now the search algorithm runs the binary search over the list $L[k]$ for such k that $k\delta\le p_i\le(k+1)\delta-1$, as any j such that $t_j=_\delta p_i$ must have been stored at $L[k]$. Still, the problem is there can be other text positions stored on $L[k]$ too,

Alg. 1. SDP-rows$(T, n, P, m, \delta, \alpha)$.

```
1   for j ← 0 to n − 1 do
2       for c ← max{0, ⌊t_j/δ⌋ − 1} to min{⌊(σ − 1)/δ⌋, ⌊t_j/δ⌋ + 1} do
3           L[c] ← L[c] ∪ {j}
4   for i ← 0 to |L[p_0]| − 1 do
5       j ← L[p_0][i]
6       if |t_j − p_0| ≤ δ then D'_i ← j
7   h ← |L[p_0]|
8   for i ← 1 to m − 1 do
9       c ← p_i; pl ← h; k ← 0; h ← 0; u ← 0
10      while u < |L[c]| AND k < pl do
11          j ← L[c][u]
12          do j' ← D'_k
13              if j − j' > α + 1 AND k < pl then k ← k + 1
14          while j − j' > α + 1 AND k < pl
15          if j' < j AND k < pl AND |t_j − c| ≤ δ then
16              D_h ← j; h ← h + 1
17              if i = m − 1 then report match
18          if k < pl then u ← min{v | D'_k < L[c][v], v > u}
19      Dt ← D; D ← D'; D' ← Dt
```

as the only thing we can deduce is that for any j in the list $L[k]$, t_j is $(2\delta - 1)$-match to p_i. To overcome this problem, we have to verify if t_j is a real δ-match. If $t_j \neq_\delta p_i$, we read the next value from $L[k]$ and continue analogously. After at most $\alpha+1$ read indexes from $L[k]$ we either have found a δ-match prolonging the matching prefix, or we have fallen off the $(\alpha + 1)$-sized window. As a result, the worst case time complexity is $O(n + |\mathcal{M}|(\log n + \alpha))$. The average time in this variant becomes $O(n + n\alpha\delta/\sigma \log n)$. Alg. 1 shows the complete pseudo code.

4 Column-Wise Sparse Dynamic Programming

In this section we present a novel column-wise variant. This algorithm runs in $O(n + n\alpha\delta/\sigma)$ and $O(n + \min(|\mathcal{M}|\alpha, nm))$ average and worst case time, respectively.

The algorithm processes the dynamic programming matrix column-wise. Let us define *Last Prefix Occurrence LPO* as

$$LPO_{i,j} = \begin{cases} j' & \max j' \leq j \mid p_0 \dots p_i =_\delta^\alpha t_h \dots t_{j'} \\ -\alpha - 1 & \text{otherwise} \end{cases} \tag{2}$$

Note that $LPO_{0,j} = j$ if $p_0 =_\delta t_j$. Note also that $LPO_{i,j}$ is just an alternative definition of $D_{i,j}$ (Eq. (1)). The pattern matching task is then to report every j such that $LPO_{m-1,j} = j$. As seen, this is easy to compute in $O(mn)$ time. In order to do better, we maintain a list of *window prefix occurrences* WPO_j that contains for the current column j all the rows i such that $j - LPO_{i,j} \leq \alpha$ where $i \in WPO_j$.

Assume now that we have computed LPO and WPO up to column $j - 1$, and want to compute LPO and WPO for the current column j. The invariant is that $i \in WPO_{j-1}$ iff $j - LPO_{i,j-1} \leq \alpha + 1$. In other words, if $i \in WPO_{j-1}$ and $j' = LPO_{i,j-1}$, then $p_0 \dots p_i =_\delta^\alpha t_h \dots t_{j'}$ for some h. Therefore, if $t_j =_\delta p_{i+1}$,

Alg. 2. SDP-columns$(T, n, P, m, \delta, \alpha)$.

```
1   for i ← 0 to m − 1 do LPO_i ← −α − 1
2   top ← 0
3   for j ← 0 to n − 1 do
4       c ← t_j; h ← 0
5       for i ← 0 to top − 1 do
6           pr ← WPO'_i
7           if |c − p_{pr+1}| ≤ δ then
8               if pr + 1 < m − 1 then
9                   WPO_h ← pr + 1; h ← h + 1
10              else
11                  report match
12      if |c − p_0| ≤ δ then
13          WPO_h ← 0; h ← h + 1
14      for i ← 0 to h − 1 do LPO_{WPO_i} ← j
15      for i ← 0 to top − 1 do
16          if LPO_{WPO_i} ≠ j AND j − LPO_{WPO_i} ≤ α then
17              WPO_h ← WPO'_i; h ← h + 1
18      top ← h
19      Lt ← WPO; WPO ← WPO'; WPO' ← Lt
```

then the (δ, α)-matching prefix from $LPO_{i,j-1}$ can be extended to text position j and row $i+1$. In such case we update $LPO_{i+1,j}$ to be j, and put the row number $i+1$ into the list WPO_j. This is repeated for all values in WPO_{j-1}. After this we check if also p_0 δ-matches the current text character t_j, and in such case set $LPO_{0,j} = j$ and insert j into WPO_j. Finally, we must put all the values $i \in WPO_{j-1}$ to WPO_j if the row i was not already there, and still it holds that $j - LPO_{i,j} \leq \alpha$. This completes the processing for the column j.

Alg. 2 gives the code. Note that the additional space we need is just $O(m)$, since only the values for the previous column are needed for LPO and WPO. In the pseudo code this is implemented by using WPO and WPO' to store the prefix occurrences for the current and previous column, respectively.

The average case running time of the algorithm depends on how many values there are on average in the list WPO. Similar analysis as in Sec. 3 can be applied to show that this is $O(\alpha\delta/\sigma)$. Each value is clearly processed in constant worst case time, and hence the algorithm runs in $O(n + n\alpha\delta/\sigma)$ average time. In the worst case the total length of the lists for all columns is $O(\min(|\mathcal{M}|\alpha, mn))$, and therefore the worst case running time is $O(n + \min(|\mathcal{M}|\alpha, mn))$, since every column must be visited. The preprocessing phase only needs to initialize LPO, which takes $O(m)$ time.

Finally, note that this algorithm can be seen as a simplification of the algorithm in [7, Sec. 5.4]. We avoid the computation of $\mathcal{M}$ in the preprocessing phase and traversing it in the search phase. The price we pay is a deterioration in the worst case time complexity, but we achieve simpler algorithm that is efficient on average. This also makes the algorithm alphabet independent.

5 Simple Algorithm

In this section we will develop a simple algorithm that in practice performs very well on small (δ, α). The algorithm inherits the main idea from Alg. 1, and

actually can be seen as its brute-force variant. The algorithm has two traits that distinguish it from Alg. 1: (*i*) the preprocessing phase is interweaved with the searching (lazy evaluation); (*ii*) binary search of the next qualifying match position is replaced with a linear scan in an $\alpha + 1$ wide text window. These two properties make the algorithm surprisingly simple and efficient on average, but impose an $O(\alpha)$ multiplicative factor in the worst case time bound.

The algorithm begins by computing a list L of δ-matches for p_0:

$$L_0 = \{j \mid t_j =_\delta p_0\}.$$

This takes $O(n)$ time (and solves the (δ, α)-matching problem for patterns of length 1). The matching prefixes are then iteratively extended, subsequently computing lists:

$$L_i = \{j \mid t_j =_\delta p_i \text{ AND } j' \in L_{i-1} \text{ AND } 0 < j - j' \leq \alpha + 1\}.$$

List L_i can be easily computed by linearly scanning list L_{i-1}, and checking if any of the text characters $t_{j'+1} \ldots t_{j'+\alpha+1}$, for $j' \in L_{i-1}$ δ-matches p_i. This takes $O(\alpha|L_{i-1}|)$ time. Clearly, in the worst case the total length of all the lists is $\sum_i L_i = |\mathcal{M}|$, and hence the algorithm runs in $O(n + \alpha|\mathcal{M}|)$ worst case time.

With one simple optimization the worst case can be improved to $O(\min\{\alpha|\mathcal{M}|, mn\})$ (improving also the average time a bit). When computing the current list L_i, Simple algorithm may inspect some text characters several times, because the subsequent text positions stored in L_{i-1} can be close to each other, in particular, they can be closer than $\alpha+1$ positions. In this case the $\alpha+1$ wide text windows will overlap, and same text positions are inspected more than once. Adding a simple safeguard to detect this, each value in the list L_i can be computed in $O(\alpha)$ *worst case* time, and in $O(1)$ best case time. In particular, if $|\mathcal{M}| = O(mn)$, then the overlap between the subsequent text windows is $O(\alpha)$, and each value of L_i is computed in $O(1)$ time. This results in $O(mn)$ worst case time. The average case is improved as well. Alg. 3 shows the pseudo code, including this improvement.

Consider now the average case. List L_0 is computed in $O(n)$ time. The length of this list is $O(n\delta/\sigma)$ on average. Hence the list L_1 is computed in $O(\alpha n\delta/\sigma)$ average time, resulting in a list L_1, whose average length is $O(n\delta/\sigma \times \alpha\delta/\sigma)$. In general, computing the list L_i takes

$$O(\alpha|L_{i-1}|) = O(n\alpha^i(\delta/\sigma)^i) = O(n(\alpha\delta/\sigma)^i) \tag{3}$$

average time. This is exponentially decreasing if $\alpha\delta/\sigma < 1$, i.e. if $\alpha < \sigma/\delta$, and hence, summing up, the total average time is $O(n)$.

6 Handling Character Classes and General Gaps

We now consider the case where the gap limit can be of different length for each pattern character, and where the δ-matching is replaced with character classes, i.e. each pattern character is replaced with a set of characters.

Alg. 3. SDP-simple$(T, n, P, m, \delta, \alpha)$.

```
1   h ← 0
2   for j ← 0 to n − 1 do
3       if |t_j − p_0| ≤ δ then
4           L[h] ← j; h ← h + 1
5   for i ← 1 to m − 1 do
6       pn ← h; h ← 0; L[pn] = n − 1
7       for j ← 0 to pn − 1 do
8           for j' ← L[j] + 1 to min(L[j + 1], L[j] + α + 1) do
9               if |t_j' − p_i| ≤ δ then
10                  L'[h] ← j'; h ← h + 1
11                  if i = m − 1 then report match
12      Lt ← L; L ← L'; L' ← Lt
```

6.1 Character Classes

In the case of character classes $p_i \subset \Sigma$, and t_j matches p_i if $t_j \in p_i$. For Alg. 2 and Alg. 3 we can preprocess a table $C[0 \ldots m-1][0 \ldots \sigma-1]$, where $C[i][c] := c \in p_i$. This requires $O(\sigma m)$ space and $O(\sigma \sum_i |p_i|)$ time, which is attractive for small σ, such as protein alphabet. The search algorithm can then use C to check if $t_j \in p_i$ in $O(1)$ time. For large alphabets we can use e.g. hashing or binary search, to do the comparisons in $O(1)$ or in $O(\log |p_i|)$ time, respectively.

Alg. 1 is a bit more complicated, since we need to have $\mathcal{M}$ preprocessed. First compute lists $L'[c] = \{i \mid c \in p_i\}$. This can be done in one linear scan over the pattern. Then list $L[i]$ is defined as $L[i] = \{j \mid t_j \in p_i\}$. This can be computed in one linear scan over the text appending j into each list $L[i]$ where $i \in L'[t_j]$. The total time is then $O(n\delta)$, where we can consider δ as the average size of the character classes. The search algorithm can now be used as is, the only modification being that where we used $L[p_i]$ previously, we now use $L[i]$ instead (and the new definition of L).

6.2 Negative and Range-Restricted Gaps

We now consider gaps of the form $g(a_i, b_i)$, where a_i denotes the minimum and b_i the maximum ($a_i \leq b_i$) gap length for the pattern position i. This problem variant has important applications e.g. in protein searching, see [8,9,10]. General gaps were considered in [10,11]. This extension is easy or even trivial to handle in all our algorithms, i.e. it is equally easy to check if the formed gap length satisfies $g(a_i, b_i)$ as it is to check if it satisfies $g(0, \alpha)$. The column-wise sparse dynamic programming is a bit trickier, but still adaptable. Yet a stronger model [8,9] allows gaps of *negative* lengths, i.e. the gap may have a form $g(a_i, b_i)$ where $a_i < 0$ (it is also possible that $b_i < 0$). In other words, parts of the pattern occurrence can be overlapping in the text.

Consider first the situation where for each $g(a_i, b_i)$: (i) $a_i \geq 0$; or (ii) $b_i \leq 0$. In either case we have $a_i \leq b_i$. Handling the case (i) is just what our algorithms already do. The case (ii) is just the dual of the case (i), and conceptually it can be handled in any of our dynamic programming algorithms by just scanning the current row from right to left, and using $g(-b_i - 2, -a_i - 2)$ instead of $g(a_i, b_i)$.

The general case where we also allow $a_i < 0 < b_i$ is slightly trickier. Basically, the only modification for Alg. 1 is that we change all the conditions of the form $0 \leq g \leq \alpha$, where g is the formed gap length for the current position, to form $a_i \leq g \leq b_i$. Note that this does not require any backtracking, even if $a_i < 0$.

Alg. 3 can be adapted as follows. For computing the list L_i, the basic algorithm checks if any of the text characters $t_{j'+1} \ldots t_{j'+\alpha+1}$, for $j' \in L_{i-1}$ matches p_i. We modify this to check the text characters $t_{j'+a_i+1} \ldots t_{j'+b_i+1}$. This clearly handles correctly both the situations $b_i \leq 0$ and $a_i < 0 < b_i$. The scanning time for row i becomes now $O((b_i - a_i + 1)|L_{i-1}|)$. The average time is preserved as $O(n)$ if we now require that $(b_i - a_i + 1)\delta/\sigma < 1$. The optimization to detect and avoid overlapping text windows clearly works in this setting as well, and hence the worst case time remains $O(n + \min\{(b - a + 1)|\mathcal{M}|, mn\})$, where for simplicity we have considered that the gaps are of the same size for all rows.

7 Preliminary Experimental Results

We have run experiments to evaluate the performance of our algorithms. The experiments were run on Pentium4 2GHz with 512Mb of RAM, running GNU/Linux 2.4.18 operating system. We have implemented all the algorithms in C, and compiled with `icc 7.0`.

We first experimented with (δ, α)-matching, which is an important application in music information retrieval. For the text we used a concatenation of 7543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1,828,089 bytes. The pitch values are in the range $[0 \ldots 127]$. This data is far from random; the six most frequent pitch values occur 915,082 times, i.e. they cover about 50% of the whole text, and the total number of different pitch values is just 55. A set of 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average user times. Fig. 1 shows the timings for different pattern lengths. The timings are for the following algorithms:

DP: Plain Dynamic Programming algorithm [3];
DP Cut-off: "Cut-off" version of DP (as in [2]);
SDP RW: Basic Row-Wise Sparse Dynamic Programming;
SDP RW fast: Binary search version of SDP;
SDP RW fast PP: linear preprocessing time variant of SDP RW fast (Alg. 1);
SDP CW: Column-Wise Sparse Dynamic Programming (Alg. 2);
Simple: Simple algorithm (Alg. 3);
BP Cut-off: Bit-Parallel Dynamic Programming [4];
NFA: Nondeterministic finite automaton, forward matching variant [10].

We also implemented the SDP RW variant with $O(\sqrt{\delta}n)$ worst case preprocessing time, but this was not competitive in practice, so we omit the plots.

SDP is clearly better than DP, but both show the dependence on m. The "cut-off" variants remove this dependence. The linear time preprocessing variant of the SDP "cut-off" is always slower than the plain version. This is due to the small

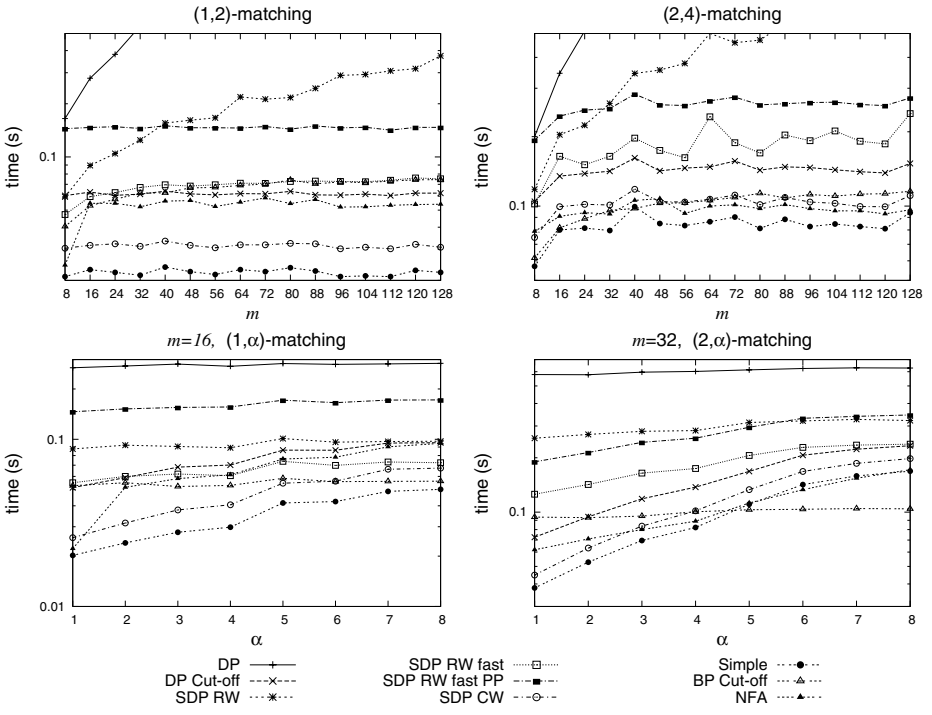


Fig. 1. Running times in seconds for $m = 8 \ldots 128$ (top) and for $\alpha = 1 \ldots 8$ (bottom). Note the logarithmic scale.

effective alphabet size of the MIDI file. For large alphabets with flat distribution the linear time preprocessing variant quickly becomes faster as m (and hence the pattern alphabet) increases. We omit the plots for lack of space. The column-wise SDP algorithm and especially Simple algorithm are very efficient, beating everything else if δ and α are reasonably small. For large (δ, α) the differences between the algorithms become smaller. The reason is that a large fraction of the text begins to match the pattern. However, this means that these large parameter values are less interesting for this application. The bit-parallel algorithm [10] is competitive but suffers from requiring more bits than fit into a single machine word.

7.1 PROSITE Patterns

We also ran preliminary experiments on searching PROSITE patterns from a 5MB file of concatenated proteins. The PROSITE patterns include character classes and general bounded gaps. Searching 1323 patterns took about 0.038 seconds per pattern with Simple, and about 0.035 seconds with NFA. Searching only the short enough patterns that can fit into a single computer word (and hence using specialized implementation), the NFA times drops to about 0.025

seconds. However, we did not implement the backward search version, which is reported to be substantially faster in most cases [10]. Finally, note that the time for Simple would be inaffected even if the gaps were negative, since only the magnitude of the gap length affect the running time.

8 Conclusions

We have presented new efficient algorithms for string matching with bounded gaps and character classes. We can handle even negative gaps efficiently. Besides having theoretically good worst and average case complexities, the algorithms are shown to work well in practice.

Acknowledgments

We thank anonymous referees for many helpful suggestions.

References

1. D. Cantone, S. Cristofaro, and S. Faro. An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceesings of WEA'05*, volume 3503 of *LNCS*, pages 428–439. Springer, 2005.
2. D. Cantone, S. Cristofaro, and S. Faro. On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with α-bounded gaps in musical sequences. In *Proceedings of ISMIR'05*, 2005.
3. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic J. of Computing*, 9(1):54–65, 2002.
4. K. Fredriksson and Sz. Grabowski. Efficient bit-parallel algorithms for (δ, α)-matching. In *Proceesings of WEA'06*, volume 4007 of *LNCS*, pages 170–181. Springer, 2006.
5. D. B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.
6. V. Mäkinen. *Parameterized approximate string matching and local-similarity- based point-pattern matching*. PhD thesis, Department of Computer Science, University of Helsinki, August 2003.
7. V. Mäkinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56(2):124–153, 2005.
8. G. Mehldau and G. Myers. A system for pattern matching applications on biosequences. *Comput. Appl. Biosci.*, 9(3):299–314, 1993.
9. G. Myers. Approximate matching of network expression with spacers. *Journal of Computational Biology*, 3(1):33–51, 1996.
10. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003.
11. Y. J. Pinzón and S. Wang. Simple algorithm for pattern-matching with bounded gaps in genomic sequences. In *Proceedings of ICNAAM'05*, pages 827–831, 2005.

Matrix Tightness: A Linear-Algebraic Framework for Sorting by Transpositions

Tzvika Hartman[1] and Elad Verbin[2]

[1] Dept. of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
hartmat@cs.biu.ac.il
[2] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel
eladv@post.tau.ac.il

Abstract. We study the problems of sorting signed permutations by reversals (SBR) and sorting unsigned permutations by transpositions (SBT), which are central problems in computational molecular biology. While a polynomial-time solution for SBR is known, the computational complexity of SBT has been open for more than a decade and is considered a major open problem.

In the first efficient solution of SBR, Hannenhalli and Pevzner [HP99] used a graph-theoretic model for representing permutations, called the *interleaving graph*. This model was crucial to their solution. Here, we define a new model for SBT, which is analogous to the interleaving graph. Our model has some desirable properties that were lacking in earlier models for SBT. These properties make it extremely useful for studying SBT.

Using this model, we give a linear-algebraic framework in which SBT can be studied. Specifically, for matrices over any algebraic ring, we define a class of matrices called *tight matrices*. We show that an efficient algorithm which recognizes tight matrices over a certain ring, $\mathbb{M}$, implies an efficient algorithm that solves SBT on an important class of permutations, called simple permutations. Such an algorithm is likely to lead to an efficient algorithm for SBT that works on all permutations.

The problem of recognizing tight matrices is also a generalization of SBR and of a large class of other "sorting by rearrangements" problems, and seems interesting in its own right as. We give an efficient algorithm for recognizing tight symmetric matrices over any field of characteristic 2. We leave as an open problem to find an efficient algorithm for recognizing tight matrices over the ring $\mathbb{M}$.

1 Introduction

One of the most promising ways to understand evolution between species is to reconstruct their evolutionary history based on genome rearrangements. In the last decade, a large body of work was devoted to a family of computational problems, called *genome rearrangement problems*. Genomes are represented by permutations, where each element stands for a gene. The basic task is, given two

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 279–290, 2006.

permutations, to find a shortest sequence of rearrangement operations (such as reversals, transpositions, translocations, etc.) that transforms one permutation into the other. Assuming (without loss of generality) that one of the permutations is the identity permutation, the problem is to find the shortest way of sorting a permutation using a given rearrangement operation (or set of operations). In this paper we mainly address the problem of sorting signed permutations by reversals and the problem of sorting (unsigned) permutations by transpositions. For more background on genome rearrangements refer to [Pev00, SEM02].

A *signed permutation* is a permutation with $+$ or $-$ on every element, which represent the direction of the corresponding gene. A *reversal* reverses the order of the elements in a segment and flips their signs. The problem of *sorting signed permutations by reversals* (SBR) is the problem of transforming a given signed permutation to the positive identity permutation using a minimum number of reversals.

A *transposition* is a rearrangement operation in which a segment is cut out of the permutation and pasted in a different location. The problem of *sorting unsigned permutations by transpositions* (SBT) is the problem of transforming a given unsigned permutation to the identity permutation using a minimum number of transpositions.[1]

Hannenhalli and Pevzner, in their seminal paper [HP99], gave a polynomial time algorithm for SBR. Subsequent works gave algorithms with better running times, and simplified the underlying theory [BH96, KST00, Ber01, KV03, TS04]. The computational complexity of SBT on the other hand is still open. There are several 1.5-approximation algorithms [BP98, Chr99, HS06], and the best algorithm to date has approximation ratio 1.375 [EH05].

To obtain a polynomial time algorithm for SBR, Hannenhalli and Pevzner used a labelled graph called the *interleaving graph* [HP99]. Each vertex of this graph is labelled either *black* or *white*. The interleaving graph models the effect of a reversal on a permutation as a graph operation on a vertex. In this operation, which we call *clicking a black vertex* v, we eliminate v while (1) replacing the subgraph induced by the neighbors of v by its complement, and (2) flipping the color of each neighbor of v.

There is a basic lower bound for the reversal distance called the *cycle lower bound* [BP96]. A central subproblem of SBR is to characterize the permutations whose reversal distance is equal to the cycle lower bound. We call these permutations *tight*. Hannenhalli and Pevzner proved that a permutation is tight if and only if each connected component of its interleaving graph contains a black vertex. This leads to an efficient algorithm for finding a minimum sorting sequence for tight permutations. They also showed how to find a minimal sorting sequence for a permutation which is not tight.

We believe that a similar approach should be used to solve SBT. Indeed there are models for SBT that try to capture the effect of performing a transposition in a way similar to the clicking operation in the interleaving graph model (e.g.

[1] See Section 3 for an explanation of why we study SBR on *signed* permutations, while SBT is studied on *unsigned* permutations.

[Chr99]). However, all existing models are incomplete, in the sense that they do not capture entirely the effect of performing a transposition on the permutation.

A cycle lower-bound exists also for SBT [BP98]. Therefore, one can generalize the notion of tightness. A characterization of tight permutations – those which can be sorted in a number of transpositions which is equal to the cycle lower bound – may be the key to a polynomial algorithm for SBT.

Our Contributions. Our first contribution is a graph-theoretic model for SBT which is analogous to the interleaving graph. In contrast with previous models, this model is *complete* in the sense that it captures the entire effect of a transposition in graph-theoretic terms. Although we can define it using a labelled graph (which has labels on both vertices and edges), it turns out that it is more natural to specify the model in linear-algebraic terms. The model is a matrix over a certain 16-element ring, which we denote by $\mathbb{M}$. A transposition is modelled by a certain *clicking* operations on a nonzero entry on the diagonal of the matrix. Equipped with this model we can state the "tightness" question for SBT in algebraic terms. This is presented in Section 3.

Our second contribution is a unified model, for which both the interleaving graph and our model for SBT are special cases. This model consists of a matrix over an arbitrary ring with a certain *clicking* operation. In Section 2 we define the matrix tightness problem, which is a generalization of both the problem of checking tightness for SBR and of checking tightness for SBT.

Next, in Sections 4 through 6 we study the matrix tightness problem. In Section 4 we give some general results on the problem, regardless of what the underlying ring is. In Section 5 we give an efficient algorithm for recognizing symmetric tight matrices over any field of characteristic 2, using an extension of the technique of Hannenhalli and Pevzner [HP99]. In Section 6 we explore the matrix tightness problem over $\mathbb{M}$, and give some starting points for solving it.

We believe that our results shed some further light on the combinatorial structure of SBR and why it is polynomially solvable. Despite the remarkable progress in recent years and the discovery of efficient algorithms, the problem is not fully understood. Better understanding may lead to even more efficient algorithms. We show a nontrivial connection between SBR and SBT in showing that both are special cases of the same algebraic question. We hope that this would trigger further progress on SBT. The algebraic clicking scheme we propose for matrices, and the question of tightness in this general context may find other applications.

Finally, we note that Meidanis and Dias [MD00] gave an algebraic model for SBT. We are not aware of any connections between their model and ours. The algebra they use is that of permutation groups, while we use mainly linear algebra.

2 The Matrix Tightness Problem

Algebra and Linear Algebra: Preliminaries. Let R be a (not necessarily commutative) ring. Recall that a ring has the properties of a field except that the

multiplication operation does not have to be commutative, and the multiplication operation does not have to be invertible. There must still be a multiplicative unit element, denoted by 1.[2]

Some elements $x \in R$, such as 1, have a multiplicative inverse (i.e. there exists $y \in R$ such that $xy = yx = 1$). We call these the *unit* elements of R. The other elements, which do not have a multiplicative inverse, are called *non-unit*. Note that an element $x \in R$ can only have a single inverse: if y_1, y_2 are both inverses of x then $y_1 = y_1 x y_2 = y_2$.

For a prime p and an integer $m \geq 1$, the *Galois Field* $\mathbb{F}_{p^m}$ is the (unique) field of size p^m. In the literature, this field is sometimes denoted by $GF(p^m)$.

The *characteristic* of a ring is the number of times 1 must be added to itself to get 0 (the characteristic is defined to be 0 if no such number exists). The characteristic of the Galois field $\mathbb{F}_{p^m}$ is p.

We now define some standard linear-algebraic concepts in somewhat non-standard ways in order to work over a non-commutative ring instead of a field. Specifically, we avoid using a determinant operator, since such an operator is hard to define over non-commutative rings. On fields our definitions coincide with the usual definitions. Let R be a ring. The vectors $v_1, \ldots, v_n \in R^m$ are called *linearly dependent* if there exist $\alpha_1, \ldots, \alpha_n \in R$ not all of which are 0 such that $\sum_{i=1}^{n} \alpha_i v_i = 0$. Otherwise, $v_1, \ldots, v_n$ are called *linearly independent*. We call a matrix A over R *regular* if its rows are linearly independent. Otherwise, we call A *singular*.

Let A be a $n \times n$ matrix over R. We always assume that the rows and columns of A are both indexed by a set V of cardinality n in a symmetric manner: row i and column i of A are indexed by the same element of V. As usual, for $S \subseteq V$ we define the *minor* of A on S to be the matrix $A[S]$ which is obtained by leaving only entries with both indices in S. $A[\emptyset]$ is the *empty matrix*, which is a matrix indexed by the set $\emptyset$. We consider the empty matrix to be regular. For $v \in V$, we denote by $A_{\downarrow v}$ and $A_{v\rightarrow}$ the column and the row indexed by v, respectively.

We denote by $\mathbb{M}$ the ring of 2×2 matrices over the binary field $\mathbb{F}_2$, with the usual matrix addition and matrix multiplication operations as ring operations. This ring consists of 16 elements. Six of them are units and ten of them are non-units.

The Matrix Tightness Problem. Let A be a $n \times n$ matrix over R indexed by V. We define the operation of *clicking* an element $v \in V$. This operation is only defined when A_{vv} is a unit of R. In this case, v is said to be *clickable*. Clicking v turns A into the matrix $A^{(*v)} \triangleq A - A_{\downarrow v} A_{vv}^{-1} A_{v\rightarrow}$. In other words, for every $i, j \in V$, $A_{ij}^{(*v)} = A_{ij} - A_{iv} A_{vv}^{-1} A_{vj}$. Equivalently, the clicking operation performs $n - 1$ elementary operations on the rows of the matrix, which turn all elements of column v, except A_{vv}, into zeroes, and then it sets row v to 0.

The matrix A is called *tight* if there is a sequence of clicking operations where each element of V is clicked exactly once. (So, a matrix is not tight if we always

[2] In some of the literature, a ring that contains a multiplicative unit element is called a *unit ring*.

"get stuck" with no clickable vertices). Note that the final result of a sequence of operations where every vertex is clicked exactly once is a matrix with all elements equal to 0. A is called *weakly-tight* if there is some sequence of clicking operations such that the final result is the zero matrix. In both cases, a sequence of clicking operations with the desired property is called a *sorting sequence*.

Let R be a fixed ring. The *tightness problem over* R is the following decision problem: The input is a $n \times n$ matrix A over R. One wishes to determine whether A is tight in time polynomial in n.

The Symmetric and Hermitian Cases. While the tightness problem is defined over any matrix, of special interest are two specific classes of matrices: the symmetric matrices and the *Hermitian* matrices. We now define the property of being Hermitian.

Let R be a ring, and let $* : R \to R$ be a function from R to R. For convenience, we write $*(x)$ as x^*. $*$ is called an *involution* on R if: (a) For any $x \in R$, $(x^*)^* = x$. (so $*$ is an invertible function); (b) For any $x, y \in R$, $(xy)^* = y^*x^*$; and (c) $0^* = 0$, $1^* = 1$.

For example, the complex conjugate is an involution over $\mathbb{C}$. For a matrix A over R, its *conjugate* with respect to involution $*$ is the matrix A^H obtained by transposing A and applying $*$ on each element of the matrix. If $A = A^H$ then A is called *Hermitian* (with respect to $*$).

If R is a ring and $*$ is an involution over it, we call the pair $(R, *)$ an *inv-ring*. The matrix tightness problem in its symmetric form over the ring R is the problem of determining whether a symmetric matrix over R is tight. The matrix tightness problem in its Hermitian form over the inv-ring $(R, *)$ is the problem of determining whether a Hermitian matrix is tight.

We often consider the matrix tightness problem over the inv-ring $(\mathbb{M}, \diamondsuit)$, where $\diamondsuit$ is the involution over $\mathbb{M}$ that exchanges the two elements on the main diagonal: $\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{\diamondsuit} = \begin{pmatrix} d & b \\ c & a \end{pmatrix}$. It is not hard to check that this is indeed an involution.

We believe that the tightness problem is of interest also outside the field of genome rearrangements. We leave open the question of applicability to other fields of research.

3 The Relation of SBR and SBT to Matrix Tightness

In this section we define the concept of r-tight and t-tight permutations, and give a short account on why checking r-tightness or t-tightness of a permutation is a sub-problem of the matrix tightness problems. A considerably more detailed explanation is included in the full version of the paper (available online).

Sorting By Reversals and Sorting by Transpositions: Preliminaries. Here we give a quick introduction to standard concepts from the theory of genome rearrangements. More relaxed presentations of these concepts can be found, e.g., in [HP99, KST00].

Let $\pi = (\pi_1 \ \dots \ \pi_n)$ be a signed permutation on n elements[3]. A *reversal* $r_{i,j}$ (for $1 \leq i \leq j \leq n$) on π reverses the order and sign of the segment of π which starts at i and ends at j, yielding: $r_{i,j} \cdot \pi = (\pi_1 \ \dots \ \pi_{i-1} \ -\pi_{j-1} \ \dots \ -\pi_i \ \pi_j \ \dots \ \pi_n)$. A *transposition* $t_{i,j,k}$ (for $1 \leq i < j < k \leq n$) exchanges between the two segments bounded by i, j and k, yielding: $t_{i,j,k} \cdot \pi = (\pi_1 \ \dots \ \pi_{i-1} \ \pi_j \ \dots \ \pi_{k-1} \ \pi_i \ \dots \ \pi_{j-1} \ \pi_k \ \dots \ \pi_n)$.

In this paper we discuss two problems. The problem of finding a shortest sequence of reversals that transforms a permutation into the identity permutation is called *Sorting by Reversals*, or *SBR*. Similarly, for transpositions the problem is *Sorting by Transpositions*, or *SBT*. Note that a transposition does not change the signs of the elements, and thus, SBT is defined only on unsigned permutations (permutations of only positive elements). The reversal (transposition) *distance* of a permutation π, denoted by $d_r(\pi)$ ($d_t(\pi)$), is the length of the shortest sorting sequence.

The Breakpoint Graph. Following Bafna and Pevzner [BP96], we first transform permutation π on n elements into a permutation $f(\pi) = \pi' = (\pi'_1 \ \dots \ \pi'_{2n})$ on $2n$ elements. $f(\pi)$ is obtained by replacing each positive element i by two elements $2i-1, 2i$ (in this order), and each negative element $-i$ by $2i, 2i-1$. In the rest of the paper we identify, in both indices and elements, $2n+1$ and 1.

Definition 1. *The* breakpoint graph $BG(\pi)$ *is an edge-colored graph on* $2n$ *vertices* $\{1, 2, \dots, 2n\}$. *For every* $1 \leq i \leq n$, π'_{2i} *is joined to* π'_{2i+1} *by a black edge, and* $2i$ *is joined to* $2i+1$ *by a gray edge.*

(see Figure 1(a) for an example).

Since the degree of each vertex is exactly 2, this graph uniquely decomposes into cycles. A cycle with k black edges and k gray edges is called a *k-cycle* and is called *odd* if k is odd. A permutation is called *r-simple* (*t-simple*) if its breakpoint graph contains only 2-cycles (3-cycles). Let $c(\pi)$ (resp. $c_{odd}(\pi)$) be the number of (odd) cycles in $BG(\pi)$. Bafna and Pevzner showed that a reversal can change $c(\pi)$ by at most one [BP96], and that a transposition can change $c_{odd}(\pi)$ by at most two [BP98]. This implies the following lower bounds for SBR and SBT, called the *cycle lower bounds*: $d_r(\pi) \geq n - c(\pi), d_t(\pi) \geq \frac{n - c_{odd}(\pi)}{2}$.

A permutation π that achieves the SBR lower bound (i.e., $d_r(\pi) = n - c(\pi)$) is called an *r-tight* permutation. The problem of determining if a permutation is r-tight is called the $TIGHT_r$ problem. Similarly, a permutation that achieves the SBT lower bound is called *t-tight*, and the decision problem is called $TIGHT_t$. The restriction of the problem $TIGHT_r$ ($TIGHT_t$) only for input permutations which are r-simple (t-simple) is denoted $TIGHT_{r,simple}$ ($TIGHT_{t,simple}$).

Two gray edges in the breakpoint graph are said to *intersect* if they intersect when one draws the breakpoint graph as in Figure 1(a). In other words, the

[3] For convenience, in this paper we consider circular permutations, that is, we consider $(\pi_1 \ \dots \ \pi_n)$ equivalent to $(\pi_2 \ \dots \ \pi_n \pi_1)$. This does not matter since both SBR and SBT are computationally equivalent for linear and circular permutations, see [MWD00, HS06].

gray edges (a, b) and (c, d) intersect if and only if the interval whose endpoints are $(\pi')^{-1}(a)$ and $(\pi')^{-1}(b)$ and the interval whose endpoints are $(\pi')^{-1}(c)$ and $(\pi')^{-1}(d)$ have a non-empty intersection, but neither of them is fully contained in the other.

The Overlap Graph. Kaplan, Shamir and Tarjan [KST00] give the *overlap graph*, a graphic model which they originally used for SBR, but that we shall use for SBT. For an unsigned permutation π, define the graph $G_{KST}(\pi)$ as follows. $G_{KST}(\pi)$ has one vertex for each gray edge of the breakpoint graph $BG(\pi)$. Two vertices are connected by an edge if their corresponding gray edges intersect. See an example in Figure 1(b).

3.1 The Relation to Matrix Tightness

We can now state our theorems that relate SBR and SBT to the matrix tightness problem:

Theorem 2. *The problem $TIGHT_{r,simple}$ is a sub-problem of the tightness problem on symmetric matrices over $\mathbb{F}_2$.*

Theorem 3. *The problem $TIGHT_{t,simple}$ is a sub-problem of the tightness problem on Hermitian matrices over $(\mathbb{M}, \diamondsuit)$.*

The proof of Theorem 2 is given implicitly in [HP99]: one simply translates an r-simple permutation π to its interleaving graph, and takes the adjacency matrix A of the interleaving graph. It holds that π is r-tight if and only if A is tight over $\mathbb{F}_2$. In a similar fashion, Kaplan, Shamir and Tarjan [KST00] implicitly prove that the problem $TIGHT_r$ is a sub-problem of the tightness problem on symmetric matrices over $\mathbb{F}_2$.

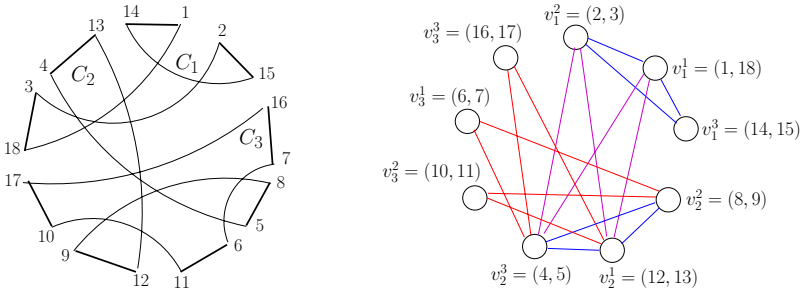
For the proof of Theorem 3 we need to transform a t-simple permutation π to a Hermitian matrix A over $(\mathbb{M}, \diamondsuit)$ such that π is t-tight if and only if A is tight. For simplicity, we give here only the transformation itself. A more detailed version can be found in the full version of the paper.

To get A we perform the following 3 steps (see example in Figure 1):

1. Draw $G = G_{KST}(\pi)$, π's overlap graph.
2. Let C_i denote the i^{th} 3-cycle of π, under some arbitrary ordering. G has 3 vertices for each such 3-cycle. Denote them by v_i^1, v_i^2, v_i^3. For any v, u which are vertices of G, denote $G(v, u) = 1$ if there is an edge in G between v and u, and $G(v, u) = 0$ otherwise.
3. Now we can write the matrix A: It is an $n/3$ by $n/3$ matrix. For every $1 \leq i, j \leq n/3$, the entry A_{ij} will be the following element of $\mathbb{M}$:

$$\begin{pmatrix} G(v_i^2, v_j^1)\ G(v_i^2, v_j^2) \\ G(v_i^1, v_j^1)\ G(v_i^1, v_j^2) \end{pmatrix}$$

Another way to view this construction is: Take the overlap graph of π. Delete one (arbitrary) vertex from each triplet. Take the adjacency matrix of the resulting graph. Exchange between each consecutive pair of rows (the first and



(a) (b)

$$
\begin{array}{c}
\\
2,3 \\
1,18 \\
8,9 \\
12,13 \\
10,11 \\
6,7
\end{array}
\begin{array}{c}
\begin{array}{cccccc} 1,18 & & 2,3 & & 12,13 & \end{array} \\
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}
\end{array}
\begin{array}{c}
\begin{array}{cccccc} 8,9 & & 6,7 & & 10,11 & \end{array} \\
\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\
\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}
\end{array}
$$

(c)

Fig. 1. (a) The breakpoint graph of the permutation $\pi = (1\ 8\ 4\ 3\ 6\ 5\ 9\ 2\ 7)$. The bold lines are the black edges, and the others are the gray edges. (b) The overlap graph, $G_{KST}(\pi)$. If you are viewing this on a color printout, the edges are colored in order to be more distinguishable. (c) The matrix A, which is our model for the permutation π. Observe that A is indeed Hermitian over $(\mathbb{M}, \diamondsuit)$.

second, the third and fourth, and so on). Now view this $(2n/3) \times (2n/3)$ matrix as a $(n/3) \times (n/3)$ matrix over $\mathbb{M}$. This is A.

4 On the Matrix Tightness Problem

In this section we explore the matrix tightness problem. We give several equivalent conditions for tightness. We show that the problem exhibits some connections to familiar linear-algebraic structures, such as gaussian elimination and matrix decompositions.

Due to lack of space we only give here a brief list of results. In the full version of the paper we give a full description of these results, and some further algebraic connections.

Theorem 4. *A is tight iff there is an ordering $(v_1, \ldots, v_n)$ of V such that for all $1 \leq k \leq n$, $A[v_1, \ldots, v_k]$ is regular. These orderings with this property are exactly the sorting sequences of A.*

(Here and in the sequel we write $A[v_1, \ldots, v_k]$ instead of $A[\{v_1, \ldots, v_k\}]$. Recall that this is the minor of A obtained by leaving only the entries with both indices in the set $\{v_1, \ldots, v_k\}$.)

Corollary 5. *If A is singular then A is not tight.*

A *permutation matrix* is a matrix with exactly one 1 in each row and each column, and the rest 0s.

Theorem 6. *Matrix A over ring R is tight iff there exist a permutation matrix P, a lower-triangular matrix L (over R) with diagonal 1, and an upper-triangular matrix U (over R) whose diagonal contains only units, such that $A = PLUP^T$. The permutation matrices P which realize this decomposition are in 1-to-1 correspondence with the sorting sequences of A.*

Analogously, for the Hermitian case, one gets that a Hermitian matrix A over inv-ring $(R, *)$ is tight iff there exist a permutation matrix P, a lower-triangular matrix L with diagonal 1, and a diagonal matrix D with only units on the diagonal, such that $A = PLDL^HP^T$. The situation for symmetric matrices is the same with $A = PLDL^TP^T$. These decompositions are similar to *Cholesky Decompositions* of positive definite matrices.

5 Tightness over Fields of Characteristic 2

In this section we give a polynomial-time checkable characterization of tightness for symmetric matrices over any field with characteristic 2. We do this by proving that the H-P theorem [HP99] generalizes, with some necessary changes, to this case.

Let A be a symmetric matrix over a field F, such that A is indexed by V. Define $G_0(A)$ to be the graph whose vertex-set is V and which has an edge (u, v) iff $A_{uv} \neq 0$. Vertex $v \in V$ is colored *black* if $A_{vv} \neq 0$, and *white* otherwise. A connected component of such a vertex-colored graph G is called *white* if all its vertices are white, and *black* otherwise.

Theorem 7. *Let A be a symmetric matrix over a field F of characteristic 2. Then A is tight iff both of the following conditions hold: (a) A is regular; (b) every connected component of $G_0(A)$ is black.*

Due to lack of space, the proof is omitted. It is included in the full version of the paper.

Theorem 7 is false over every field of characteristic other than 2. The matrix $\begin{pmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{pmatrix}$ has determinant -4, and is thus regular over every such field. Any clicking operation performed on it gives the matrix $\begin{pmatrix} 0 & -2 \\ -2 & 0 \end{pmatrix}$ which is obviously non-tight.

The problem of checking tightness over fields of characteristic other than 2, such as $\mathbb{F}_3$ or $\mathbb{R}$, seems interesting both in its own right and as a way to develop techniques that will help resolve the Hermitian problem over $\mathbb{M}$. Also of interest is the problem of checking tightness of Hermitian matrices over the inv-ring

$(F, *)$ where F is any field, and $*$ is the involution with $x^* = x^{-1}$ for $x \neq 0$ and $0^* = 0$. We could not resolve this even for fields of characteristic 2 (except $\mathbb{F}_2$, where it coincides with the symmetric case). This is especially interesting for the field $\mathbb{F}_4$, because, as we describe in the next section, it arises naturally as a sub-problem of the Hermitian problem over $\mathbb{M}$.

6 On Tightness over $\mathbb{M}$ and Other Difficult Variants

In this section we consider the problem of checking tightness of Hermitian matrices over the inv-ring $(\mathbb{M}, \diamondsuit)$. Although this ring has characteristic 2, the problem is difficult both because we are dealing with Hermitian instead of symmetric matrices, and because $\mathbb{M}$ is a ring rather than a field. It seems natural to ask whether this problem has any interesting sub-problems. One way to define a sub-problem is to restrict our matrices to have elements only from a sub-ring of $\mathbb{M}$. Every sub-ring of $\mathbb{M}$ defines a sub-problem in this manner.

In the full version we discuss all sub-rings of $\mathbb{M}$. Here we only discuss the sub-problem where all elements are taken from the sub-ring $\mathbb{M}_4 = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \right\}$. This sub-problem is equivalent to a much more elementary problem: to the tightness problem of general (i.e. not necessarily symmetric or Hermitian) matrices over $\mathbb{F}_2$. Thus, we would like to consider the tightness problem for general matrices over $\mathbb{F}_2$.

There is a natural graph formulation of the general problem over $\mathbb{F}_2$. This formulation may be easier to visualize. Let G be a directed graph with vertices colored black or white. The clicking operation on v, defined only when v is a black vertex, performs the following three operations: (1) for every vertex u, flip the color of u iff (u, v) and (v, u) are both edges of the graph; (2) For every ordered pair of different vertices u, w, if (u, v) and (v, w) are both edges of the graph then the directed edge (u, v) is created if it does not exist, and deleted if it does exist; and (3) delete from the graph the vertex v and all edges touching it. G is called tight if there is a sequence of clicking operations where every vertex is clicked exactly once. We wish to find a polynomial-time characterization of the tight graphs.

This problem seems to be difficult. We suspect that if this problem can be solved then one can use it to solve the Hermitian problem over $(\mathbb{M}, \diamondsuit)$, as well as over other fields and rings of characteristic 2.

Here is an example that shows some phenomenon that may be surprising. Let G be a graph consisting of a single directed cycle of length n. We are interested in what ways of coloring it make it tight. It can easily be seen that coloring any set of at most $n - 2$ vertices of G black and the other vertices white makes the graph non-tight, while coloring $n - 1$ vertices black and the remaining vertex white makes the graph tight. Coloring all vertices black makes the adjacency matrix of the graph non-regular and therefore non-interesting (because of Corollary 5). We see here that the tightness problem over directed graphs exhibits some "counting" properties that seem not to exist in the undirected case.

The tightness problem over directed graphs is also non-monotonic. Consider the following two matrices

$$A = \begin{pmatrix} 1\,0\,1\,1 \\ 1\,0\,1\,0 \\ 0\,1\,0\,1 \\ 1\,1\,0\,1 \end{pmatrix}, A' = \begin{pmatrix} 0\,0\,1\,1 \\ 1\,0\,1\,0 \\ 0\,1\,0\,1 \\ 1\,1\,0\,1 \end{pmatrix} .$$

Both A and A' are regular. However, A' is tight while A is not. This is surprising since this cannot happen for symmetric matrices over $\mathbb{F}_2$: if B, B' are symmetric regular matrices such that for every i, j, $B'_{ij} \leq B_{ij}$, and if B' is tight then B must also be tight. This is a consequence of Theorem 7.

7 Conclusions and Discussion

The main open problem is to resolve the tightness problem for a wider range of fields and rings, as well as involution operators, than we have done here. A specifically interesting and approachable problem is the general (i.e. not symmetric and not Hermitian) matrix tightness problem over $\mathbb{F}_2$, which has been discussed in Section 6.

We have recently discovered that the results presented in this paper can be viewed in the more general framework of the theory of delta-matroids (see e.g. [Gee96]). It seems that one can define a tightness problem over delta-matroids, and there exists a generalization of the H-P theorem to that setting.

We believe that the matrix tightness problem may be even more closely related to sorting by rearrangements problems than we show in this paper. Specifically, we show how the problem of r- or t-tightness of a permutation is a special case of the problem of tightness of a matrix. It seems interesting to try to cast the problem of finding a minimum-length sorting sequence of a permutation, or determining the length of such a sequence, into the algebraic realm that we explore here. One would perhaps first try to get a graph-theoretic model that models the effect of any reversal/transposition.[4]

Our approach here can be applied to the problems of sorting under other genome rearrangement operations (or sets of rearrangement operations). This will be discussed in the full version of this paper.

Acknowledgements. We would like to thank Noga Alon, Isaac Elias, Felix Goldberg, Haim Kaplan, Ron Shamir and Roded Sharan for fruitful discussions. Elad Verbin would like to thank Martin C. Golumbic and the Caesarea Edmond Benjamin de Rothschild Foundation for their hospitality and for the opportunity to present preliminary results leading to this research in the France-Israel Expert Workshop on Graph Classes and Graph Algorithms, 2004.

[4] Our model for SBT can represent only a limited class of transpositions. Similarly, the KST overlap graph and H-P's interleaving graph model only so-called *elementary* reversals. These restricted classes of operations suffice for checking r-/t- tightness, but are not enough for giving a minimum-length sorting sequence of a non-tight permutation.

References

[Ber01] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. In *Combinatorial Pattern Matching (CPM '01)*, pages 106–117, 2001.

[BH96] P. Berman and S. Hannenhalli. Fast sorting by reversal. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Combinatorial Pattern Matching, 7th Annual Symposium*, volume 1075 of *Lecture Notes in Computer Science*, pages 168–185, Laguna Beach, California, 10-12 June 1996. Springer.

[BP96] V. Bafna and P. A. Pevzner. Genome rearragements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[BP98] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, May 1998.

[Chr99] D. A. Christie. *Genome Rearrangement Problems*. PhD thesis, University of Glasgow, 1999.

[EH05] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. In *proceedings of the Fifth Workshop on Algorithms in Bioinformatics (WABI)*, pages 204–215, 2005.

[Gee96] Jim Geelen. *Matchings, matroids and unimodular matrices*. PhD thesis, University of Waterloo, 1996. available at `http://www.math.uwaterloo.ca/~jfgeelen`.

[HP99] S. Hannenhalli and P. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46:1–27, 1999.

[HS06] T. Hartman and R. Shamir. A simpler and faster 1.5-approximation algorithm for sorting by transpositions. *Information and Computation*, 204(2):275–290, 2006.

[KST00] H. Kaplan, R. Shamir, and R. E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, 29(3):880–892, 2000.

[KV03] H. Kaplan and E. Verbin. Effficient data structures and a new randomized approach for sorting signed permutations by reversals. In *Proc. 14th Annual Symposium on Combinaotrial Pattern Matching (CPM '03)*, pages 170–185. Springer, 2003.

[MD00] J. Meidanis and Z. Dias. An alternative algebraic formalism for genome rearrangements. In *Comparative Genomics: Gene Order Dynamics, Map Alignment and the Evolution of Gene Families, volume 1 of Series in Computational Biology*, pages 213–223, 2000.

[MWD00] J. Meidanis, M. E. Walter, and Z. Dias. Reversal distance of signed circular chromosomes. manuscript, 2000.

[Pev00] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, 2000.

[SEM02] D. Sankoff and N. El-Mabrouk. Genome rearrangement. In *Current Topics in Computational Molecular Biology*. MIT Press, 2002.

[TS04] E. Tannier and M. Sagot. Sorting by reversals in subquadratic time. In *Proc. 15th Annual Symposium on Combinaotrial Pattern Matching (CPM '04)*, pages 1–13. Springer, 2004.

How to Compare Arc-Annotated Sequences: The Alignment Hierarchy

Guillaume Blin[1] and Hélène Touzet[2]

[1] IGM-LabInfo - UMR CNRS 8049 - Université de Marne-la-Vallée
77 454 Marne-la-Vallée Cedex 2 - France
gblin@univ-mlv.fr
[2] LIFL - UMR CNRS 8022 - Université Lille 1
59 655 Villeneuve d'Ascq Cedex - France
Helene.Touzet@lifl.fr

Abstract. We describe a new unifying framework to express comparison of arc-annotated sequences, which we call *alignment of arc-annotated sequences*. We first prove that this framework encompasses main existing models, which allows us to deduce complexity results for several cases from the literature. We also show that this framework gives rise to new relevant problems that have not been studied yet. We provide a thorough analysis of these novel cases by proposing two polynomial time algorithms and an **NP**-completeness proof. This leads to an almost exhaustive study of alignment of arc-annotated sequences.

Keywords: computational biology, RNA structures, arc-annotated sequences, NP-hardness, edit distance, algorithm.

1 Introduction

In computational biology, comparison of RNA molecules has attracted a lot of interest recently. From a combinatorial perspective, one can distinguish two types of modeling that allow for various flexibility and preciseness in the encoding of RNA structures: macroscopic representations, with two-interval graphs [16,4], and microscopic representations with arc-annotated sequences, originally introduced in [6]. We focus here on arc-annotated sequences, which are raw sequences provided with related additional information in the form of arcs connecting pairs of positions. The set of arcs determines the way the sequence folds into a three-dimensional space.

Arc-annotated sequences may be refined into four main paradigms: tree edit distance [15,17,11,5], tree alignment [10], longest common arc-preserving subsequence [6,9,12], and general edit distance [8,3]. We propose a unifying framework to express comparison of arc-annotated sequences that is based on the introduction of the *common arc-annotated supersequence*. This framework has several instances depending on the definition of the embedding involved in the notion of supersequence, and the type of the supersequence (Nested, Crossing or

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 291–303, 2006.

UNLIMITED). It gives rise to a hierarchy of problems, that we called the ALIGN *hierarchy* in reference to the tree alignment. We show that this hierarchy brings together all previously mentioned comparison models for arc-annotated sequences, and leads to the introduction of new comparison models that are biologically relevant. In particular, we propose two polynomial time algorithms for the problem of comparing two NESTED arc-annotated sequences, whereas corresponding algorithms considering the same set of edit operations in other formalisms are not polynomial (Sections 4.2 and 5). We also give an **NP**-completeness result that gives some new insight on the hardness of the comparison of CROSSING arc-annotated sequences (Section 4.3). This leads to an almost exhaustive study of the ALIGN hierarchy. Due to space considerations, complete proofs are deferred to the full version of the paper.

2 Edition Models for Arc-Annotated Sequences

Given a finite alphabet Σ, an arc-annotated sequence is defined by a pair (S, P), where S is a string of Σ^* and P is a set of arcs connecting pairs of characters of S. In reference to RNA structures, characters are called *bases*. Bases with no incident arc are called *single bases*. As usually done in the study of arc-annotated sequences, we distinguish four levels of arc structure (originally proposed by Evans in [6]):

- UNLIMITED (UNLIM) – no restriction at all,
- CROSSING (CROS) – there is no base incident to more than one arc,
- NESTED (NEST) – there is no base incident to more than one arc and no arcs are crossing,
- PLAIN – there is no arc.

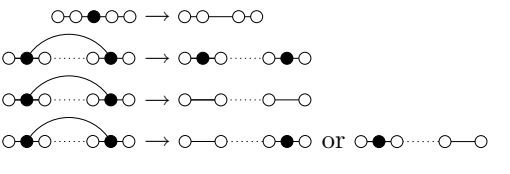
There is an obvious inclusion relation between those arc types with the $\subset$ operator (PLAIN $\subset$ NESTED $\subset$ CROSSING $\subset$ UNLIMITED). Since we focus here on structure comparison, we do not consider PLAIN sequences, which do not carry any structural information. In the remaining of this paper, we shall only deal with sequences of type NESTED, CROSSING and UNLIMITED.

In order to compare two arc-annotated sequences, we consider the set of edit operations (and their associated costs) introduced in [13] and classify it into two groups:

Substitution operations, inducing renaming of bases in the arc-annotated sequence: *base-match* ($w_m : \Sigma^2 \to \mathbb{R}$), *base-mismatch* ($w_m : \Sigma^2 \to \mathbb{R}$), *arc-match* ($w_{am} : \Sigma^4 \to \mathbb{R}$), *arc-mismatch* ($w_{am} : \Sigma^4 \to \mathbb{R}$).

Deletion operations, inducing deletion of bases and/or of arcs:

base-deletion ($w_d : \Sigma \to \mathbb{R}$)
arc-breaking ($w_b : \Sigma^4 \to \mathbb{R}$)
arc-removing ($w_r : \Sigma^2 \to \mathbb{R}$)
arc-altering ($w_a : \Sigma^3 \to \mathbb{R}$)



Given the above set of operations, we define three edit models:

I : all substitution operations, base-deletions and arc-removings are allowed,
II : the operations of model I and arc-alterings are allowed,
III : the operations of model II and arc-breakings are allowed.

In the following, given two arc-annotated sequences u and v, a *K-edit script* from u to v will refer to a series of non-oriented operations of the model K transforming u into v. The *cost of a K-edit script* from u to v, denoted $\texttt{cost}(u, v, K)$ is the sum of the costs of each operation involved in the K-edit script. We define the *K-edit distance* between u and v as the minimum cost of a K-edit script from u to v. Finding this K-edit distance is called the $\textsc{Edit}(u, v, K)$ problem.

For each model $K \in \{\text{I}, \text{II}, \text{III}\}$, we also define an *ordering relation* $\trianglelefteq_K$: if u can be obtained from v by a series of deletion and substitution operations of the model K, then $u \trianglelefteq_K v$. Provided with these notations, we propose to extend the notion of subsequence on strings to arc-annotated sequences as follows.

Definition 1 (K-subsequence). *Given two arc-annotated sequences u and v, and an edit model $K \in \{\text{I}, \text{II}, \text{III}\}$, u is said to be a K-subsequence of v if, and only if, $u \trianglelefteq_K v$.*

Given three arc-annotated sequences u, v and w such that $w \trianglelefteq_K u$ and $w \trianglelefteq_K v$, w is said to be a *common K-subsequence* of u and v. We define the cost of a common K-subsequence w of u and v as the minimum sum of operation costs needed to transform u into w and v into w: $\texttt{cost}(u, w, K) + \texttt{cost}(v, w, K)$.

When dealing with plain sequences, it is well-known that each edit script can be associated with a common subsequence of the same cost. This property is still valid with K-edit scripts on arc-annotated sequences.

Lemma 1. *Given two arc-annotated sequences u and v, and an edit model $K \in \{\text{I}, \text{II}, \text{III}\}$, solving the $\textsc{Edit}(u, v, K)$ problem is equivalent to finding a common K-subsequence w of u and v of minimal cost.*

We now turn to a novel paradigm, simply considering K-supersequences instead of K-subsequences. We shall see that this alternative point of view is a fruitful perspective and that it brings new insights on arc-annotated comparison.

Definition 2 (K-supersequence). *Given two arc-annotated sequences u and v, and an edit model $K \in \{\text{I}, \text{II}, \text{III}\}$, u is said to be a K-supersequence of v if, and only if, $v \trianglelefteq_K u$.*

In a similar way as for common subsequences, given three arc-annotated sequences u, v and w, w is a *common K-supersequence* of u and v if $u \trianglelefteq_K w$ and $v \trianglelefteq_K w$. The cost of w is defined as $\texttt{cost}(w, u, K) + \texttt{cost}(w, v, K)$. First, we prove that each Edit problem can reduce to finding an optimal supersequence.

Lemma 2. *Given two arc-annotated sequences u and v, and an edit model $K \in \{\text{I}, \text{II}, \text{III}\}$, there exists a common K-subsequence of u and v of cost α iff there exists a common K-supersequence of u and v of the same cost.*

A point worth to notice with Lemma 2 is that the type of the common supersequence is not guaranteed to be the same as the type of the common subsequence. Figure 1 illustrates such an example. The edit script associated with the optimal subsequence (which is of NESTED type) has a smaller cost than the edit script associated with the optimal NESTED supersequence. Indeed, when constructing the set of arcs of the common K-supersequence of u (above) and v (below), it is likely to create crossing arcs or multiple arcs incident to a single character that are absent in the initial sequences. In general, when considering arc-annotated sequences of NESTED types, searching for a common NESTED supersequence is more restrictive than searching for a common subsequence. In example of Figure 1, it is necessary to authorize CROSSING supersequences to get the same cost as for the EDIT problem. This observation gives rise to a family of new problems, which we call the ALIGN *hierarchy*.

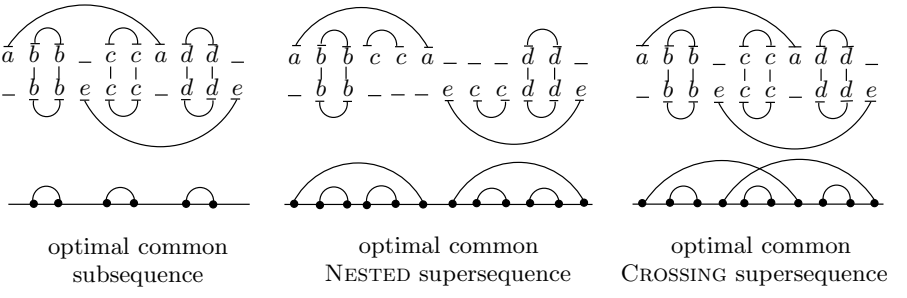


Fig. 1. Comparison of the optimal common subsequence and the optimal common supersequences. The optimal common subsequence is derived from u and v with two arc-removings. The optimal common NESTED supersequence requires four arc-removings. In this example, it is necessary to allow crossing arcs in the supersequence to get the same cost as for the subsequence (third scheme).

Definition 3 (Arc-annotated sequence alignment). *Given three types of sequences A, B and C of* $\{\text{NESTED}, \text{CROSSING}, \text{UNLIMITED}\}$ *and an edit model* $K \in \{\text{I}, \text{II}, \text{III}\}$*, the* $\text{ALIGN}(A, B, K) \to C$ *problem is defined as:*

INPUT: *two arc-annotated sequences u and v of type A and B respectively.*
OUTPUT: *a common K-supersequence w of type C of minimum cost.*

The purpose of this paper is to study exhaustively the ALIGN hierarchy and confront it to known results for existing comparison models for arc-annotated sequences. Since $\text{ALIGN}(A, B, K) \to C$ is equivalent to $\text{ALIGN}(B, A, K) \to C$, we can always assume that $B \subseteq A$. Moreover, in order for the problem to be meaningful, we impose $A \subseteq C$. Therefore, the hierarchy contains thirty distinct entries when considering all relevant possibilities for A, B, C and K.

The first result worth to notice is that the ALIGN hierarchy includes all instances of the edit distance problem, as stated in Theorem 1. This is a consequence of Lemma 1 and Lemma 2.

Theorem 1. *Given two types A, B in* $\{\text{Nested}, \text{Crossing}, \text{Unlimited}\}$ *and an edit model* $K \in \{\text{I}, \text{II}, \text{III}\}$*, the* $\text{Edit}(A, B, K)$ *and* $\text{Align}(A, B, K) \to \text{Unlim}$ *problems are equivalent.*

3 Ordered Trees and the Edit Model I

Comparing arc-annotated sequences of Nested types when considering the edit model I amounts to comparing ordered trees. Each pair of connected bases corresponds to an internal node, and each single base corresponds to a leaf. Moreover, in this model, considering arc-annotated I-supersequences of Unlimited type is meaningless as stated in Lemmas 3 and 4.

Lemma 3. *Given two types A, B in* $\{\text{Nest}, \text{Cros}\}$*, the* $\text{Align}(A, B, \text{I}) \to \text{Unlim}$ *and* $\text{Align}(A, B, \text{I}) \to \text{Cros}$ *problems are equivalent.*

Lemma 4. *Given a type B in* $\{\text{Nest}, \text{Cros}\}$*, the* $\text{Align}(\text{Unlim}, B, \text{I}) \to \text{Unlim}$ *problem has the same complexity as* $\text{Align}(\text{Cros}, B, \text{I}) \to \text{Cros}$*.*

Together with Theorem 1, these two lemmas imply that nine out of ten entries of the model I are equivalent or reduce to Edit problems. The only problem that does not reduce to an edit problem is $\text{Align}(\text{Nest}, \text{Nest}, \text{I}) \to \text{Nest}$, which fully corresponds to the ordered tree alignment, introduced by Jiang *et al.* in [10]. Therefore, the Align hierarchy is completely solved for the edit model I, as summed up in Table 1.

Table 1. Align hierarchy for the edit model I. According to Lemma 3, the ten problems of the hierarchy reduce to seven distinct instances. We indicate entries that can also be formulated as edit problems with $\times$ in the second column (see Theorem 1). Complexity results are indicated for two arc-annotated sequences u and v s.t. $\max(|u|, |v|) = n$.

$A \times B \to C$	Edit	model I
Nest × Nest → Nest		$O(n^4)$ – Jiang [10]
Nest × Nest → Cros Nest × Nest → Unlim	×	$O(n^3 \log(n))$ – Klein [11]
Cros × Nest → Cros Cros × Nest → Unlim	×	$O(n^3 \log(n))$ – Ma [14]
Cros × Cros → Cros Cros × Cros → Unlim	×	**NP**-complete – Ma [14]
Unlim × Nest → Unlim	×	$O(n^3 \log(n))$ – Lemma 4
Unlim × Cros → Unlim	×	**NP**-complete – Ma [14]
Unlim × Unlim → Unlim	×	**NP**-complete – Ma [14]

4 The Edit Model II

4.1 Some Correspondences with the LAPCS Problem

As introduced by Evans in [6], the LONGEST ARC-PRESERVING COMMON SUBSEQUENCE problem (LAPCS for short) is defined as follows: given two arc-annotated sequences u and v, find the longest – in terms of sequence length – common arc-annotated subsequence w of u and v such that an arc (i, j) in w can only be obtained from both an arc in u and an arc in v (*i.e.* arc-preserving). We prove hereafter that the LAPCS problem is a specific case of the common subsequence problem when considering the edit model II, namely the EDIT(A, B, II) problem, provided that the score system for edit operations is correctly chosen. The cost of a base-deletion or of an arc-altering is 1, the cost of an arc-removing is 2, and substitutions are prohibited, with arbitrary high costs.

Theorem 2. *Let u, v, w be three arc-annotated sequences. The sequence w is a longest arc-preserving common subsequence of u and v iff $w \trianglelefteq_{\text{II}} v$ and $w \trianglelefteq_{\text{II}} u$.*

This theorem combined with Theorem 1 allows us to derive several cases of the ALIGN hierarchy for the edit model II from recent results published in the LAPCS literature. All known results are summed up in Table 2. It remains four specific problems: ALIGN(NEST, NEST, II) $\rightarrow$ {NEST, CROS} and ALIGN(CROS, {NEST, CROS}, II) $\rightarrow$ CROS. The first two problems can be seen as a refinement of the EDIT(NESTED, NESTED, II) problem, which is not tractable. We solve them in the next two sections, and show that the first one is polynomial, whereas the second one is **NP**-complete. It follows that ALIGN(CROS, NEST, II) $\rightarrow$ CROS and ALIGN(CROS, CROS, II) $\rightarrow$ CROS are also **NP**-complete.

4.2 ALIGN(NESTED, NESTED, II) $\rightarrow$ NESTED Problem Is Polynomial

We exhibit a polynomial algorithm for the ALIGN(NEST, NEST, II) $\rightarrow$ NEST problem. This result is somehow unexpected since the associate edit problem EDIT(NESTED, NESTED, II) is **NP**-complete. It shows that imposing structural constraints on the type of the common supersequence is an adequate way for lower complexity of untractable problems.

We saw in Section 3 that in the model I the ALIGN(NEST, NEST, I) $\rightarrow$ NEST problem is polynomial, since it is equivalent to ordered tree alignment. The algorithm proposed in [10] proceeds by dynamic programming. Each step of the algorithm adds a component in the supersequence – one single base or two bases connected by an arc – that is selected so as to minimize the cost of the alignment.

We show here that the formulas for the edit model I can be extended to the edit model II by adding supplementary rules for the arc-altering operation. All rules concerning substitutions, base-deletions and arc-removings are identical.

We introduce some notations for the representation of arc-annotated sequences. Let $\circ$ be a binary operator that concatenates two arc-annotated sequences.

Table 2. ALIGN hierarchy for edit models II and III. We indicate problems that can be formulated as edit distance problem in the second column. In these cases, known results stem from the LAPCS problem for the model II (Theorems 1 and 2), and from the general edit distance for the model III (Theorem 1). Other problems are specific to the ALIGN hierarchy and are introduced and studied in this paper. Blank cells are for problems that are still open. Complexity results are indicated for two arc-annotated sequences u and v s.t. $\max(|u|, |v|) = n$.

$A \times B \to C$	EDIT	model II	model III
NEST × NEST → NEST		$O(n^4)$	$O(n^4)$
NEST × NEST → CROS		**NP**-complete	
NEST × NEST → UNLIM	×	**NP**-complete – Lin [12]	**NP**-complete – Blin [3]
CROS × NEST → CROS		**NP**-complete	
CROS × NEST → UNLIM	×	**NP**-complete – Evans [6]	Max SNP-hard – Jiang [8]
UNLIM × NEST → UNLIM	×		
CROS × CROS → CROS		**NP**-complete	
CROS × CROS → UNLIM	×	**NP**-complete – Evans [6]	Max SNP-hard – Jiang [8]
CROS × UNLIM → UNLIM	×		
UNLIM × UNLIM → UNLIM	×		

$\alpha(u) \circ v$ denotes the arc-annotated sequence composed by an arc α spanning the arc-annotated sequence u, concatenated to the arc-annotated sequence v. $b \circ u$ denotes the arc-annotated sequence composed by the single base b concatenated to the arc-annotated sequence u. The common supersequence is built from right to left. We consider five cases depending on the form of the pair of arc-annotated sequences to align, that determines which edition rules to apply. Arc-altering operation creates an arc in the common supersequence. So it should not be considered for all forms of pairs of arc-annotated sequences: At least one of the two sequences should begin with a base incident to an arc. We write A for the cost of the alignment between two arc annotated-sequences.

1. $A(\alpha(u), \beta(w)) \ =$

$$\min \begin{cases} w_{am}(\alpha, \beta) + A(u, w) \text{ – arc-(mis)match} \\ w_r(\beta) + \min\{A(y, w) + A(z, \varepsilon) | y \circ z = \alpha(u)\} \text{ – arc-removing} \\ w_r(\alpha) + \min\{A(u, y) + A(\varepsilon, z) |\ y \circ z = \beta(w)\} \text{ – arc-removing} \end{cases}$$

2. $A(\alpha(u) \circ v, \beta(w) \circ x) \ =$

$$\min \begin{cases} w_{am}(\alpha, \beta) + A(u, w) + A(v, x) \text{ – arc-(mis)match} \\ w_r(\beta) + \min\{A(y, w) + A(z, x) | y \circ z = \alpha(u) \circ v\} \text{ – arc-removing} \\ w_r(\alpha) + \min\{A(u, y) + A(v, z) |\ y \circ z = \beta(w) \circ x\} \text{ – arc-removing} \\ w_a(\alpha, b) + \min\{A(u, y) + A(v, z) |\ y \circ b \circ z = \beta(w) \circ x\} \text{ – arc-altering} \\ w_a(\beta, b) + \min\{A(y, w) + A(z, x) |\ y \circ b \circ z = \alpha(u) \circ v\} \text{ – arc-altering} \end{cases}$$

3.$A(b \circ v, \beta(w) \circ x) \ =$

$$\min \begin{cases} w_d(b) + A(v, \beta(w) \circ x) - \text{base-deletion} \\ w_r(\beta) + \min\{A(y,w) + A(z,x) | y \circ z = b \circ v\} - \text{arc-removing} \\ w_a(\beta, b) + \min\{A(y,w) + A(z,x) |\ y \circ z = v\} - \text{arc-altering} \\ w_a(\beta, b_2) + \min\{A(y,w) + A(z,x) |\ y \circ b_2 \circ z = b \circ v\} - \text{arc-altering} \end{cases}$$

and symetrically

4.$A(\alpha(u) \circ v, b \circ x) \ =$

$$\min \begin{cases} w_d(b) + A(\alpha(u) \circ v, x) - \text{base-deletion} \\ w_r(\alpha) + \min\{A(u,y) + A(v,z) | y \circ z = b \circ x\} - \text{arc-removing} \\ w_a(\alpha, b) + \min\{A(u,y) + A(v,z) |\ y \circ z = x\} - \text{arc-altering} \\ w_a(\alpha, b_2) + \min\{A(u,y) + A(v,z) |\ y \circ b_2 \circ z = b \circ x\} - \text{arc-altering} \end{cases}$$

5.$A(b \circ v, b_2 \circ x) \ =$

$$\min \begin{cases} w_d(b) + A(v, b_2 \circ x) - \text{base-deletion} \\ w_d(b_2) + A(b \circ v, x) - \text{base-deletion} \\ w_m(b, b_2) + A(v, x) - \text{base-(mis)match} \end{cases}$$

The hypothesis that the common supersequence is of NESTED type guarantees the correctness of the recurrence relations. The whole complexity remains unchanged: it is $O(n^4)$. A full analysis of this algorithm and its application to RNA structure comparison (global alignment, local alignment etc.) is presented in further detail in [7].

Theorem 3. ALIGN(NEST, NEST, II) $\rightarrow$ NEST *is polynomial.*

4.3 Hardness Result for ALIGN(NESTED, NESTED, II) $\rightarrow$ CROSSING

We show in this section that relaxing the constraint on crossing arcs in the common supersequence makes the problem difficult.

Theorem 4. ALIGN(NEST, NEST, II) $\rightarrow$ CROS *is* **NP***-complete.*

The decision problem is defined formally as follows.

INPUT*: two arc-annotated sequences u and v of* NESTED *type and an integer ℓ.*
QUESTION*: can one find an arc-annotated sequence w of* CROSSING *type which is a common* II*-supersequence of u and v of cost lower than or equal to ℓ ?*

We initially notice that this problem is in **NP** since given three arc-annotated sequences u, v and w one can check polynomially if (1) w is of CROSSING type, (2) w is a common II-supersequence of u and v, and (3) the cost of w is lower than or equal to ℓ. In order to prove that it is **NP**-complete, we propose a polynomial reduction from the **NP**-complete problem MIS-3P [2].

MIS-3P
INPUT*: a cubic planar bridgeless connected graph $G = (V, E)$ and an integer k.*
QUESTION*: is there an independent set of vertices of G* – i.e. *a set $V' \subseteq V$ such that no two vertices of V' are connected by an edge in E – of cardinality greater than or equal to k ?*

A graph $G = (V, E)$ is said to be a *cubic planar bridgeless connected* graph if any vertex of V is of degree three (cubic), G can be drawn in the plane in such a way that no two edges of E cross (planar), and there are a least two paths – with no edge in common – connecting any pair of vertices of V (bridgeless connected).

The idea of the proof is to encode a cubic planar bridgeless connected graph by two arc-annotated sequences. The construction uses first a 2-page book embedding.

Theorem 5 (Bernhart and al. [1]). *One can always find, in polynomial time, a 2-page book embedding of a cubic planar bridgeless connected graph with the following additional property: on each page, any vertex has a non-null degree.*

A *2-page book embedding* of a graph G is a linear ordering of the vertices of G along a line and an assignment of the edges of G to the two half-planes delimited by the line – called the *pages* – so that no two edges assigned to the same page cross. For convenience, we will refer to the page above (resp. below) the line as the *top-page* (resp. *bottom-page*).

Given a 2-page book embedding, we construct two arc-annotated sequences of NESTED type $u = (S, P)$ and $v = (T, Q)$ on the three-letters alphabet $\{a, b, \#\}$. The underlying raw sequences S and T are defined as follows:

$$S = \#^n\ S_1\ \#^n\ S_2\ \ldots\ \#^n\ S_n$$
$$T = \#^n\ T_1\ \#^n\ T_2\ \ldots\ \#^n\ T_n$$

where n is the number of vertices of the initial graph, and for each $1 \leq i \leq n$, S_i (resp. T_i) is a segment *baaa* if the degree of the vertex $v_i \in V$ in the top-page (resp. bottom-page) equals two, a segment *aaab* otherwise.

Now that the sequences S and T are defined, we have to copy the arc configuration of the top-page (resp. bottom-page) on S (resp. T). Each edge (v_i, v_j) of the top-page is represented by an arc in P. More precisely, this arc connects a base a of S_i and a base a of S_j. We proceed in a similar way for each edge of the bottom-page by adding, for each one, an arc in Q. Moreover, we impose that when a vertex v_i is of degree two on the top-page (resp. bottom-page), the two corresponding arcs in P (resp. Q) are incident to the rightmost two bases a of the segment S_i (resp. T_i). And, consequently, we impose that, when a vertex v_i is of degree one on the top-page (resp. bottom-page), the corresponding arc in P (resp. Q) is incident to the leftmost base a of the segment S_i (resp. T_i). It is easy to check that it is always possible to reproduce on u and v the non-crossing edge configuration of each page. An example of such a construction is given in Figure 2. The size of u and v is quadratic in n: the length of S and T is $n(n+4)$ and the total number of arcs is $3\frac{n}{2}$. In the following, we will refer to any such construction as an *align-construction*.

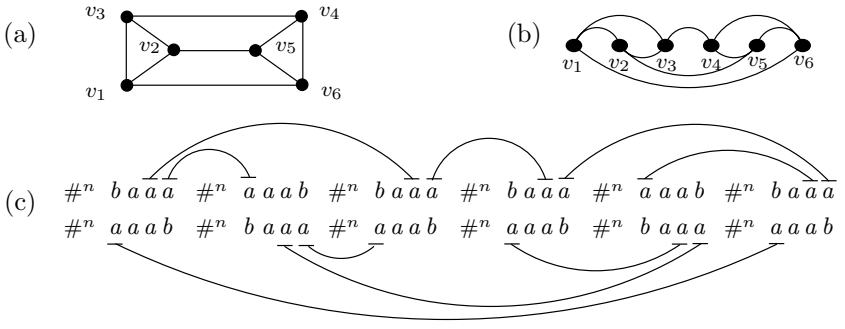


Fig. 2. Example of an align-construction. The graph (a) is a cubic planar bridgeless connected graph of 6 vertices. The graph (b) is a 2-page book embedding of the graph (a) such that, on each page, any vertex has a non-null degree. (c) The two arc-annotated sequences of NESTED type obtained from the graph (a) by an align-construction.

For the sake of simplicity, but w.l.o.g.[1], we set the score system as follows: $w_d(b) = 2, w_d(\#) = 6, w_d(a) = 1, w_a(a,a,a) = 1.5, w_r(a,a) = 2$. As a matter of fact, the proof is still valid with any combination of parameters that fullfils these two inequalities: $3w_a(a,a,a) + 2w_d(b) < 3w_r(a,a) + 3w_d(a)$ and $w_r(a,a) + 3w_d(a) < w_a(a,a,a) + 2w_d(b)$.

We first show that for any such pair of arc-annotated sequences with the given score system, there exists a "canonical" optimal common II-supersequence whose form is easy to characterize. This is the purpose of the two following Lemmas.

Lemma 5. *Let u and v be two arc-annotated sequences of* NESTED *type obtained by an align-construction for an initial graph of n vertices. There exists an optimal common* II*-supersequence $w = (U, R)$ such that U is of the form $\#^n U_1 \ldots \#^n U_n$ where for each $i \in 1..n$, $U_i = aaabaaa$ or $U_i = baaab$.*

Lemma 6. *Let u and v be two arc-annotated sequences of* NESTED *type obtained by an align-construction. In any optimal common* II*-supersequence $w = (U, R)$ of u and v, if there is an arc in R connecting a base of the segment U_i and a base of the segment U_j, then U_i and U_j cannot be both of the form baaab.*

These lemmas allow us to express the cost of an optimal NESTED supersequence between two arc-annotated sequences obtained with the align-construction.

Lemma 7. *Let u and v be two arc-annotated sequences of* NESTED *type obtained by an align-construction. The cost of any optimal common* II*-supersequence w is $3pw_a(a,a,a) + 3(\frac{n}{2} - p)w_r(a,a) + 3(n-p)w_d(a) + 2pw_d(b)$, where p is the number of segments of w of type baaab.*

We now turn to prove that ALIGN(NEST,NEST,II) $\rightarrow$ CROS is **NP**-complete with this following Lemma. This concludes the proof of Theorem 4.

[1] Since a subcase of ALIGN(NEST, NEST, II) $\rightarrow$ CROS is hard, so does the general problem.

Lemma 8. *A cubic planar bridgeless connected graph $G = (V, E)$ admits an independent set of vertices of cardinality greater than or equal to k if, and only if, there exists an arc-annotated sequence w of* Crossing *type that is a common* II*-supersequence of u and v of cost lower than or equal to $\ell = 3kw_a(a,a,a) + 3(\frac{n}{2} - k)w_r(a,a) + 3(n-k)w_d(a) + 2kw_d(b)$, where u and v are arc-annotated sequences of* Nested *type resulting from an align-construction of G and $n = |V|$.*

Remark 1. The arc-annoted sequences of the **NP**-completeness proof are not conform to the representation of an RNA molecule. It is likely to impose supplementary constraints on the encoding of the 2-page book embedding in order to get sequences that are more RNA-like: the alphabet is $\{A, U, C, G\}$, all arcs correspond to Watson-Crick pairings ($A \leftrightarrow U$ and $C \leftrightarrow G$) and base-deletion costs are more realistic. To achieve this goal, we modify the definition of u and v in the following way: replace # with twelve occurrences of C, b with $GGGGGG$ and a with AU (AU is self-complementary). Each edge in the 2-page book embedding now corresponds to two arcs between AU and AU. Figure 3 shows this new representation for the example of Figure 2.

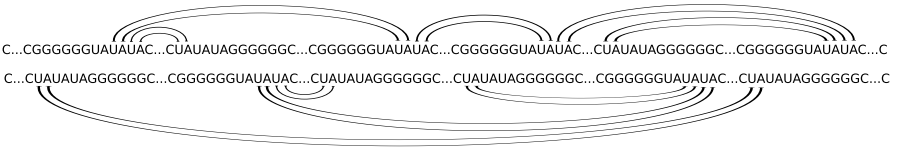


Fig. 3. RNA-like arc-annotated sequences for the example of Figure 2

5 The General Edit Distance and the Edit Model III

The edit model III corresponds to the set of operations introduced by Jiang *et al.* in the *general edit distance* problem [8]. Therefore it allows us to derive several complexity results from known results on the *general edit distance* [8,3] with Theorem 1. As illustrated in Table 2, the complexity of Align(Nest, Nest, III) $\rightarrow$ {Nest, Cros} and of Align(Cros, {Nest, Cros}, III) $\rightarrow$ Cros only is still to elucidate. We solve Align(Nest, Nest, III) $\rightarrow$ Nest.

Theorem 6. Align(Nest, Nest, III) $\rightarrow$ Nest *is polynomial.*

To prove the correctness of the above Theorem, we show that we can enrich the polynomial time algorithm defined in Section 4.2 by incorporating rules for arc-breaking operations. At each step of the construction of the common supersequence, it is necessary that one of the sequence begins with an arc, and the other one with a single base for the arc-breaking operation to be valid. So only cases 3 and 4 in the recurrence relations are concerned by the application of an arc-breaking rule.

3. $A(b \circ v, \beta(w) \circ x) \ =$
$$\min \begin{cases} \dots \\ w_b(\beta, b, b_2) + \min\{A(y, w) + A(z, x) | x \circ b_2 \circ z = v\} \end{cases}$$

4. $A(\alpha(u) \circ v, b \circ x) \ =$
$$\min \begin{cases} \dots \\ w_b(\alpha, b, b_2) + \min\{A(u, y) + A(v, z) | y \circ b_2 \circ z = x\} \end{cases}$$

6 Conclusion

In this article, we have proposed and studied a new framework for comparing arc-annotated sequences, namely the Align hierarchy. We think that this study is relevant both from a practical perspective and theoretical perspective. We have provided two polynomial time algorithms to compare arc-annotated sequences of Nested type with arc-altering and arc-breaking operations, whereas when considering other models, the problem is **NP**-complete. We also gave a new **NP**-completeness result, that enhances understanding of the complexity of arc-annotated sequences comparison. This result sheds a new light on the border between tractability and untractability when dealing with arc-annotated sequences – especially of Crossing type.

Those results, combined with the ones derived from Edit and Lapcs comparison models, have almost filled the complexity table of the Align hierarchy. As illustrated in Table 2, there still exist some open questions for the model III. But we can notice that the edit model III reduces to the edit model II when the cost of any arc-breaking is arbitrary high. As a consequence, the **NP**-completeness of Align(Nest, Nest, II) $\rightarrow$ Cros and of Align(Cros, $*$, II) $\rightarrow$ Cros shows that there exists no polynomial algorithm for arbitrary values of parameters (such as usual dynamic programming algorithms do). We, thus, conjecture that both Align(Nest, Nest, III) $\rightarrow$ Cros and Align(Cros, $*$, III) $\rightarrow$ Cros problems are **NP**-complete.

References

1. F. Bernhart and B. Kainen. The book thickness of a graph. *J. Comb. Theory Series B*, 27:320–331, 1979.
2. T.C. Biedl, G. Kant, and M. Kaufmann. On triangulating planar graphs under the four-connectivity constraint. *Algorithmica*, 19(4):427–446, 1997.
3. G. Blin, G. Fertin, I. Rusu, and C. Sinoquet. RNA sequences and the EDIT(NESTED, NESTED) problem. *technical report - LINA*, 2003.
4. M. Crochemore, D. Hermelin, G.M. Landau, and S. Vialette. Approximating the 2-interval pattern problem. In *ESA'05*, pages 426–437, 2005.
5. S. Dulucq and H. Touzet. Decomposition algorithms for the tree edit distance problem. *Journal of Discrete Algorithms*, 3(2-4):448–471, 2005.
6. P. Evans. *Algorithms and Complexity for Annotated Sequences Analysis*. PhD thesis, University of Victoria, 1999.

7. C. Herrbach, A. Denise, S. Dulucq, and H. Touzet. A polynomial algorithm for comparing RNA secondary structures using a full set of operations.
8. T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.
9. T. Jiang, G. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. *Journal of Dicrete Algorithms*, pages 257–270, 2004.
10. T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theoretical Computer Science*, 143(1):137–148, 1995.
11. P. Klein. Computing the edit-distance between unrooted ordered trees. In *6th European Symposium on Algorithms*, pages 91–102, 1998.
12. G. Lin, Z.-Z. Chen, T. jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *Journal of Computer and System Sciences*, 65:465–480, 2002.
13. G. Lin, B. Ma, and K. Zhang. Edit distance between two rna structures. In *RECOMB*, pages 211–220, 2001.
14. B. Ma, L. Wang, and K. Zhang. Computing similarity between RNA structures. *Theoretical Computer Sciences*, 276:111–132, 2002.
15. K.C. Tai. The tree-to-tree correction problem. *Journal of the Association for Comput. Machi.*, 26:422–433, 1979.
16. S. Vialette. On the computational complexity of 2-interval pattern matching. *Theoretical Computer Science*, 312(2-3):223–249, 2004.
17. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

Structured Index Organizations for High-Throughput Text Querying

Vo Ngoc Anh and Alistair Moffat

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

Abstract. Inverted indexes are the preferred mechanism for supporting content-based queries in text retrieval systems, with the various data items usually stored compressed in some way. But different query modalities require that different information be held in the index. For example, phrase querying requires that word offsets be held as well as document numbers. In this study we describe an inverted index organization that provides efficient support for all of conjunctive Boolean queries, ranked queries, and phrase queries. Experimental results on a 426 GB document collection show that the methods we describe provide fast evaluation of all three querying modes.

1 Introduction

Inverted indexes are the preferred mechanism for supporting content-based queries in text retrieval systems, with the various data items usually stored compressed in some way [Witten et al., 1999, Zobel and Moffat, 2006]. For text documents, a document-level inverted index can typically be stored in under 10% of the space of the original documents.

However, different query modalities require that different information be held in the index, a need that creates tensions in the way that the index lists are organized. For example, phrase querying requires that word offsets be maintained in the index as well as document numbers, but those same word offsets represent unwanted decoding when conjunctive Boolean queries are being processed, or when ranked queries are being handled. One solution to this dilemma – and not as wasteful as it might at first appear – is to store two indexes, one of which contains word positions, and one of which contains only document numbers. Queries of different types can then be routed to the appropriate index, and overall average query throughput rates increased at the cost of additional disk space.

In this paper we first briefly summarize different types of index organization, and consider their ability to support fast evaluation of different types of query. A uniform framework is introduced that allows different organizations to be described in a succinct and unambiguous manner. The second part of the paper then describes a blocked interleaved inverted index organization that provides efficient support for all of conjunctive Boolean queries, ranked queries, and phrase queries. The proposed representation is a hybrid between a single index and a duplicated index, and exploits the best features of

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 304–315, 2006.

both. The paper concludes with experimental results on a 426 GB document collection show that the methods we describe provide fast evaluation of all of Boolean, ranked, and phrase queries.

2 Inverted Indexes

An inverted index for a collection of documents associates with each of its distinct term an *inverted list*, that stores a *pointer* for each document the term appears in. In simplest form a pointer consists of nothing more than an ordinal document number, meaning that the inverted list for a term t can be described as

$$\langle d \rangle^{f_t} ,$$

where f_t is the number of documents containing t; d is a document number; and the $\langle x \rangle^k$ notation indicates k repetitions of objects of type x. To resolve a query, the inverted lists for the query terms are fetched, and various operations performed on them, depending on the semantics assigned to the query operators.

There are a number of factors that determine what exactly is stored in the index, and how it is used to resolve queries. The rest of this section discusses these choices.

Pointer ordering: In a *document-sorted* index, the pointers in each inverted list are stored in increasing document order, allowing differences between consecutive term appearances to be stored as d-gaps, rather than absolute document numbers. Storing differences yields a more compact inverted file once compression techniques are applied, but requires that processing of each inverted list be in document number order.

An alternative is to use a *frequency-sorted* index [Persin et al., 1996] or an *impact-sorted* [Anh et al., 2001, Anh and Moffat, 2006b]. In the former structure, each inverted list is ordered in decreasing term frequency score $f_{d,t}$. In the latter, the pointers are ordered according to the *impact* value $\omega_{d,t}$, which is a small integer representing the overall contribution of term t to the score of document d, including the factor used to normalize for document length. In both arrangements, each index list is a sequence of groups of equally weighted pointers, and within each group the document numbers are sorted, and again stored as d-gaps. Experiments by Persin et al. [1996] and Anh et al. [2001] showed that these alternative representations can be stored in approximately the same space as a standard document-sorted index, but yield significantly faster processing of ranked queries.

Processing mode: Independent of the structure of the index or the type of query being handled, there are two main approaches to processing queries: the *document-at-a-time* model in which all inverted lists are simultaneously accessed, and $|q|$-way processing is carried out to handle a query q of $|q|$ terms; and the *term-at-a-time* model, in which only one inverted list is accessed at any given time, and a sequence of $|q|-1$ binary operations are performed. Recent work has tended to focus on term-at-a-time processing as applied to ranked querying, but it is not clear that term-at-a-time processing is fundamentally better than document-at-a-time processing. For example, Strohman et al. [2005] have

Table 1. Types of index and supported query modes

Type	d	$\omega_{d,t}$	pos.	Organization	Processing modes supported
D	Y	–	–	document-sorted	term-at-a-time, document-at-a-time
DS	Y	Y	–	document-sorted	term-at-a-time, document-at-a-time
				impact or frequency-sorted	term-at-a-time, score-at-a-time
DSP	Y	Y	Y	document-sorted	term-at-a-time, document-at-a-time

experimented with document-at-a-time orderings. As well as describing an improved index organization, this paper re-examines the issue of processing model, discussing the relative advantages of the two approaches, and comparing them experimentally on the same large 426 GB text collection.

For ranked querying with impact-sorted indexes, another approach, *score-at-a-time* has been mooted [Hawking, 1998, Anh et al., 2001]. In this mode, all of the term lists are open for reading, but are processed in decreasing score contribution order rather than in increasing document number order. Similar to the term-at-a-time approach, this method requires a set of accumulator variables, but facilitates dynamic query pruning; and the parts of each inverted list that are not required are potentially never read from disk.

Word positions and index levels: Another important way of categorizing an index is whether or not it includes within-document frequencies and within-document word positions. We distinguish between three levels of detail in regard to the index information stored and summarize the three respective index types in Table 1 along with the possible index organizations and the supported processing modes. As is listed in the table, a *Type D* index (*d*ocument numbers only) contains only document numbers, and is capable of supporting Boolean queries only. A *Type DS* (*d*ocument and *s*core) index stores both document numbers and either $\omega_{d,t}$ impact scores or $f_{d,t}$ within-document frequencies (from which $\omega_{d,t}$ values can be computed), or both, and can handle Boolean and ranked queries, but not phrase-queries. A *Type DSP* index (*d*ocument, *s*core, *p*ositions) contains document numbers, $\omega_{d,t}$ values (or $f_{d,t}$ values), and, for each document that the term appears in, a list of the ordinal word positions within the document at which the term appears. Type DSP indexes are capable of supporting all of the three mentioned query types (Boolean, ranked, phrase) and some other types such as proximity queries.

Table 1 also shows that impact- and frequency-sorting applies only to Type DS indexes, and is beneficial only when ranked queries are being performed. These indexes must be processed using a term-at-a-time strategy, or a score-at-a-time approach.

One of our aims in this investigation was to evaluate the extent to which – with suitable internal organizations applied to each inverted list – Type DSP indexes can be used to efficiently support Boolean and ranked queries. We sought to determine whether including positional information necessarily degraded querying performance for Boolean and ranked queries compared to the less voluminous Type D and Type DS indexes.

3 Interleaving

One key issue that has received little previous attention is that of *interleaving*, which also relates to the internal organization of each inverted list in a document-sorted index.

Pointer interleaving: In typical descriptions of Type DS inverted indexes, each document number (usually stored as a difference, or d-gap) is immediately followed by the corresponding $\omega_{d,t}$ or $f_{d,t}$ value. This arrangement can be described as:

$$\langle d, f_{d,t} \rangle^{f_t} .$$

In a Type DSP index, the word positions are usually described as being inserted immediately adjacent to the corresponding document number and $f_{d,t}$ value.. If, in addition, impact-based ranking is required (which is possible even if the index is document-sorted), the impact score is also included. The Type DSP index then has the form:

$$\left\langle d, \omega_{d,t}, f_{d,t}, \langle p \rangle^{f_{d,t}} \right\rangle^{f_t} ,$$

where p is a positional offset, and $\omega_{d,t}$ is a small integer [Anh et al., 2001].

We categorize such indexes as being *pointer-interleaved*, since the interleaving of different quantities is at the level of the individual pointers. Pointer-interleaving is the way that Type DS and Type DSP indexes are presented in textbooks (for example, [Witten et al., 1999]); in the research literature (for example, [Williams et al., 2004]); and in public domain software systems such as `Zettair` (available from `www.seg.rmit.edu.au`). What is not clear is whether or not commercial systems use pointer-interleaved indexes – such information is, of course, kept confidential.

Term interleaving: An alternative is to keep the various related parts of each inverted list in contiguous blocks, so that (in the case of a Type DSP index), the arrangement in each inverted list becomes:

$$\left\langle \langle d \rangle^{f_t}, \langle \omega_{d,t} \rangle^{f_t}, \langle f_{d,t} \rangle^{f_t}, \langle p \rangle^{\Sigma f_{d,t}} \right\rangle .$$

Non-interleaved indexing: The final option is for all of the document numbers for all of the pointers for all of the terms to be stored, then all of the ranking weights for all of the terms, then all of the within-document frequencies for all of the terms, and finally all of the word offsets for all of the terms. The arrangement is somewhat akin to having four separate inverted files, each storing a particular type of information, with four distinct disk pointers maintained in each entry in the vocabulary.

For completeness, our experiments also cover impact-sorted Type DS indexes, despite the fact that they are primarily designed for term-at-a-time and score-at-a-time processing of ranked queries. This kind of index has a specific structure where one impact score is shared by (in general) a sequence of documents, and allows fast querying because the amount of query-time computation is kept small.

Tradeoffs: There are many tensions between these representational and execution options. In a pointer-interleaved index, the information required for phrase and mixed querying is tightly clustered, and immediately available. In a term-interleaved index, the positional information is available within the terms' lists, but Boolean and ranked queries can be processed without the need to decode, or even bypass, the positional information. However the presence of the positional information at the end of every list might erode the effectiveness of buffering and caching strategies.

In a non-interleaved index, the positional information is fetched and decoded only when phrase queries are encountered in the input stream. Non-interleaved indexes also make it somewhat easier to apply compression, as longer sequences of values likely to be drawn from the same underlying probability distribution. This latter point is especially important in our system, which employs the binary-based `slide` compression technique, in which sequences of values of similar magnitudes are identified and represented compactly [Anh and Moffat, 2006a].

However, for phrase queries, non-interleaved indexes have the drawback of requiring information to be consolidated from several different access points, or *cursors*.

4 Block-Interleaved Indexes

To allow exploration of the trade-offs possible with different interleaving strategies, we introduce a hybrid approach called *block-interleaved* indexing

Groups and fixed-size blocks: The structure of a k-block interleaved index is given by:

$$\left\langle \langle d \rangle^k, \langle \omega_{d,t} \rangle^k, \langle f_{d,t} \rangle^k, \left\langle \langle p \rangle^k \right\rangle^{\lceil \Sigma f_{d,t}/k \rceil} \right\rangle^{\lceil f_t/k \rceil}$$

Each sub-unit $\langle x \rangle^k$ for some kind of value x is a *k-block*, and each unit of k complete pointers is a *group*. That is, each inverted list is built up as a sequence of one or more groups, and within each group there are four or more k-blocks. Figure 1 gives an example list that is stored as two groups of (at most) $k = 4$ pointers; and within each group, all of the values are stored as k-blocks. Note that the set of positional offsets $\langle p \rangle^k$ are also in blocks of k, and might cross pointer boundaries. In this structure a group is a streamlined bundle of k pointers.

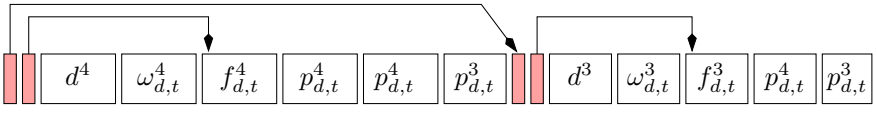


Fig. 1. An example block-interleaved index structure for an inverted list of $f_t = 7$ pointers, using $k = 4$. The list is organized in two groups. The first group contain $k = 4$ pointers, and the second group holds 3 pointers. Within the groups, all of the data items are stored in blocks. In the example, the sum of the $f_{d,t}$ values in the first group is assumed to be 11; and in the second group, 7. The shaded items are internal access structures to facilitate skipping past unneeded data.

The routines that manipulate k-blocks are optimized to support an interface that fetches and decodes the next (as many as) k values from an inverted list. A key benefit of this approach is that the decoding buffer is exactly bounded at k values for each of the blocks that are required. None of the query modes require that more than four blocks be active per term, and thus that each term requires decode buffer space of $4k$ words.

Skip pointers: The shaded items in Figure 1 are internal *skips*, that allow sections to be stepped over and not decoded. Skips are typically represented as byte or bit increments that lead to the start of some future codeword. Each group in a block-interleaved list has two skip pointers: a *group skip* that references the start position of the next group, and a *block skip* that provides access within the group, allowing the k-block of $\omega_{d,t}$ values to be stepped over, so that the set of $f_{d,t}$ values can be directly accessed. The blocks of positional offsets are never accessed without the block of $f_{d,t}$ values being retrieved first; and so there is no need for a skip to the start of the positions.

The group skips described here are similar to the skips used by Moffat and Zobel [1996] to speed up the term-at-a-time processing of Boolean and ranked queries. Similar improvement can be expected for our group skips. On the other hand, the block skips have a different function, and they allow bypassing of blocks of values that are not required in processing this query, for example, the positional information when Boolean queries are being processed. The skips of Moffat and Zobel are to facilitate searching for candidates, perhaps as a result of dynamic query pruning and accumulator limiting, and their index is stored in a pointer interleaved manner. Though it is possible, no pruning of ranked queries occurs in any of the experiments reported in this paper, so that unfair comparison between different processing modes is avoided.

Processing queries: A further issue worth elaboration is that of how queries are processed. Taking the simplest possible example, and presuming a document-sorted index and a conjunctive Boolean query, the task at hand is to identify the set of document numbers that appear in all of the terms' inverted lists.

If a term-at-a-time processing strategy is adopted, then the standard mechanism for doing this is to process the terms in increasing f_t order, using the first term to establish a set of *candidate* answers C, and then for each subsequent term t, checking each of the candidates against t's inverted list:

```
 1: open the inverted list for the term t_1 with the smallest f_t.
 2: set C ← copy_list(t_1).
 3: for i ← 2 to |q| do
 4:     open the inverted list for term t_i.
 5:     for each candidate c ∈ C, in increasing order, do
 6:         set d ← seek_list_value(t_i, c).
 7:         if d > c then
 8:             set C ← C − {c}.
 9:             if |C| = 0 then
10:                 return the empty set.
11: return C.
```

The key operation performed is that of "*seek_list_value*(t, c)", which seeks forwards in the inverted list of term t until a document number greater than or equal to c is encountered. If c is actually found, it is retained as a candidate; if the next document number d is greater than c, then c is removed as a candidate.

If document-at-a-time processing is being carried out, the set of candidates is not required, but all of the terms' lists need to be simultaneously open:

```
1: set C ← { }.
2: for i ← 1 to |q| do
3:    open the inverted list for term t_i.
4:    set d_i ← next_list_value(t_i).
5: while all lists have pointers remaining do
6:    set d ← max{d_i | 1 ≤ i ≤ |q|}.
7:    for i ← 1 to |q| do
8:       set d_i ← seek_list_value(t_i, d).
9:       set d ← max{d, d_i}.
10:   if min{d_i | 1 ≤ i ≤ |q|} = d then
11:      set C ← C + {d}.
12:      for i ← 1 to |q| do
13:         set d_i ← next_list_value(t_i).
14: return C.
```

Note that the locus of activity does not pause at every pointer in every inverted list. Instead, the focus leap-frogs down the set of lists, using *seek_list_value* operations to try and catch each list's activity zone up to the current document number in the list that to date has progressed the furthest. In particular, the costly "max" and "min" operations over the full set of $|q|$ current document numbers are relatively infrequent, and occur at most once for each of the items in the shortest inverted list. They do not occur for every pointer in every list, as would be the case in a heap-based merging process.

In a block-interleaved index, the *seek_list_value* operation still proceeds as a sequential decoding of a k-block of d-gaps, but is done within the routines handling the inverted list. They transparently use the block skip pointer in a group to access the next group if the sum of the remaining d-gaps in the current block does not reach the specified target value. While linear-time in its underlying operation, the nature of the compression process means that this operation is extremely fast, and only the first k-block of values is read and decoded in any groups that are otherwise completely skipped.

Also worth noting is that the same basic operations are used to implement phrase queries. The query is initially handled as a Boolean one, and only when a document is determined to contain all of the query terms is any access made to any of the k-blocks storing the positional information. On the other hand, the current implementation of ranked querying does require that every d-gap and every $\omega_{d,t}$ impact value be processed – dynamic pruning issues have not yet been explored.

Other blocking issues: The parameter k determines the amount of compressed information decoded in each access to the pointers in a group, and reflects the amount of decompressed information that is maintained at any point in time. Another important parameter is the unit of access to compressed information – the size of the buffers into

which inverted lists are read when required by the decoding routines. Our system does not read the whole of each inverted list in a single operation, since doing so requires compressed buffers of indeterminate (and variable) size. Instead, each inverted list is read "on demand" in compressed blocks of a fixed size, and input buffers are bounded. The drawback is that multiple seeks may be required to access a given list in its entirety, even in a term-at-a-time processing model; but in a disk-block-based file system this is likely anyway, and our arrangement simply acknowledges that reality.

Setting a size to the list buffers again involves competing tensions. Large buffers reduce the number of seek operations, but may add unnecessary decoding costs. On the other hand, use of overly short logical blocks implies multiple accesses even when relatively short inverted lists are being processed. In the experiments reported below the compressed block size (the unit read from disk) was fixed at 8 kB; and the logical block size (the amount of data decoded in each call to the decoding routines, the value k in Figure 1) was initially set at $8{,}192$ integers.

5 Experimental Evaluation

The various implementation options have been tested and compared on a large collection of typical web data – the 426 GB `GOV2` collection. This collection was created as part of the TREC initiative, see `trec.nist.gov`, and was drawn from an early-2004 crawl of the `.gov` domain. It contains approximately 25 million documents.

Queries: Two query sets, `Q1000` and `Q321`, are drawn from the set of $50{,}000$ real-life queries used in the 2005 TREC Terabyte Track experiments. The former set consists of the first $1{,}000$ queries from the superset; the latter set contains only those queries of `Q1000` that contain at least two terms and have at least one answer when considered as phrase queries against the `GOV2` collection. There are 321 queries that satisfy these two requirements. Over the two sets, the average numbers of terms per query are 2.79 and 2.41; the average number of conjunctive Boolean answers are approximately 79×10^3 and 72×10^3, respectively.

Baseline: To establish a reference point for query speed, we started with conjunctive Boolean querying. The second data column of Table 2 shows the average query processing rate (queries per second) for Boolean queries, document-at-a-time processing, the `GOV2` collection and `Q1000` queries, and several different index types.

As was anticipated, the best Boolean querying performance is achieved by a Type D index. Use of a pointer-interleaved Type DS index slows query handling by around 40%, and a pointer-interleaved Type DSP index slices more than 90% off the performance. A term-interleaved Type DS index allows the original level of performance to be regained, but even with a Type DSP term-interleaved index there is some loss of throughput.

The next column of Table 2 shows the throughput rate at which the `Q1000` queries can be processed as ranked queries. All of the Type DS indexes provide comparable performance, at throughput rates around half of what can achieved when the same queries are treated as being Boolean conjunctions. Querying rates with a Type DSP index are again slow if a pointer-interleaved index is used. Recall that these throughput rates represent exhaustive document-at-a-time processing without any form of query pruning.

Table 2. Performance on a single 2.8 GHz Intel Xeon with 1 GB RAM and the collection GOV2: index size and querying throughput for different query types with document-at-a-time processing using two query sets, measured as queries per second. The logical access unit in to the compressed streams is $k = 8{,}192$ integers.

Index arrangement		Index Size (GB)	Q1000			Q321		
			Boolean	Ranked	Phrase	Boolean	Ranked	Phrase
Type D		**5.05**	**5.73**	–	–	**5.24**	–	–
Type DS	pointer-interl.	9.21	3.23	2.46	–	3.47	**3.02**	–
	term-interl.	**6.85**	5.71	**2.49**	–	4.99	2.80	–
	non-interl.	**6.85**	5.52	2.46	–	5.12	2.69	–
Type DSP	pointer-interl.	47.18	0.48	0.46	0.46	0.58	0.58	0.60
	term-interl	**36.13**	5.35	2.33	**0.47**	4.95	2.62	**0.64**
	non-interl.	**36.13**	5.50	2.37	**0.47**	5.02	2.56	**0.64**

The fourth column of Table 2 shows the throughput rate at which the Q1000 queries can be processed as phrase queries. A Type DSP index is required, and the low throughput levels achieved reflect processing of a large volume of compressed data. All types of interleaving give comparable speeds. Note that there are many queries in Q1000 that are just a single term, and many multi-term queries for which there are no answers.

The last three columns of Table 2 show query throughput rates for the query subset Q321, for which every query has at least one Boolean answer and one phrase answer. Similar trends in performance are observed.

Index size: The first data column in Table 2 shows the size of the various inverted indexes used in these experiments. When compressed, the document components account for space equivalent to only a little over 1% of the initial collection; the $\omega_{d,t}$ components for a further 0.5% in a term-interleaved setting (which allows "runs" of like values to be exploited by the compression regime); and then the word positions account for a further approximately 6%. The total Type DSP term-interleaved index can be stored in less than 40 GB, or under 10% of the size of the data. Note also that with the particular compression mechanism used the pointer interleaved index is more expensive to store than the term-interleaved and non-interleaved variants, because pointer interleaving makes the values in each list less locally homogeneous [Anh and Moffat, 2006a].

Block-interleaved indexes: Table 3 shows the performance of Type DS and Type DSP block-interleaved indexes, in an experimental setting comparable to that used to obtain the results in Table 2, but with a logical block size of $k = 1{,}048{,}576$. Even with word positions included (the Type DSP index), all of Boolean, ranked, and phrase queries are processed at similar or better rates to those shown in Table 2.

Nor is there any penalty in terms of index size – the change to block interleaving involves an index cost for the GOV2 collection 1.59% for the Type DS index, and 8.44% for the Type DSP index.

Choosing a block size: Figure 2 shows how query throughput is affected by the choice of the logical block size parameter k. Because one of the key query processing costs

Table 3. Querying throughput for block-interleaved indexes and two query sets. The logical access unit in to the compressed streams is the same as the blocksize, $k = 1{,}048{,}576$.

Index arrangement	Q1000			Q321		
	Boolean	Ranked	Phrase	Boolean	Ranked	Phrase
Type DS block-interl.	**5.76**	**2.56**	–	5.16	**3.07**	–
Type DSP block-interl.	5.15	2.41	**1.32**	4.76	2.88	**0.69**

is disk seek times, small values of k are relatively inefficient. At the left of the graph, when $k < 1{,}000$, performance is very similar to that obtained from a pointer-interleaved Type DSP index, because multiple blocked groups fit within each 8 kB physical access block. Savings appear when k is $10{,}000$ or more, because groups now span more than one disk block, and the skip pointers mean that some of the seeks are eliminated. The best performance for all querying modalities arises at k or around one million. With this value of k only a small fraction of the inverted lists are split into more than one group, meaning that the majority of the index is stored as if it were term-interleaved. Nevertheless, buffer sizes and thus caching costs are controlled at the same time as all of the throughput gains of the term-interleaved index are attained.

Processing modes and speed: All of the experiments reported in the previous section are for document-at-a-time evaluation, which has the advantage of requiring per-query execution space proportional to the number of answers being generated rather than proportional to the length of any of the inverted lists being processed. For example, Heaps [1978, Chapter 6] describes the document-at-a-time mechanism, as do Turtle and Flood [1995] and Strohman et al. [2005]. But Section 2 mentioned two other ways in which queries can be evaluated: the term-at-a-time, and the score-at-a-time approaches. In particular, Witten et al. [1999] present query processing using the term-at-a-time paradigm, for both Boolean and ranked queries. Kaszkiel et al. [1999] evaluate both of those

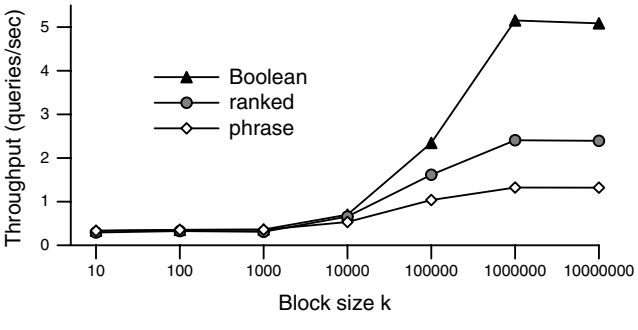


Fig. 2. Query throughput rate as a function of the logical block size parameter k, using collection GOV2, a Type DSP block-interleaved index, query set Q1000, document-at-a-time processing, and three different query types.

Table 4. Querying throughput for different query processing modes using query sets Q1000 and Q321, with other details as for Table 3. The two pointer-interleaved indexes are document-sorted.

Index arrangement	Index Size (GB)	Q1000 Ranked	Q321 Ranked
Type DS pointer-interl. document-at-a-time	9.21	2.46	3.02
Type DS pointer-interl. term-at-a-time	9.21	2.35	2.29
Type DS impact-sorted score-at-a-time	5.99	**4.12**	**3.50**

strategies in the context of passage retrieval, and conclude that document-at-a-time is superior when the number of terms is small, but that term-at-a-time is to be preferred when the number of terms in the query is more than around 3–5. Kaszkiel et al. also describe a hybrid mechanism that processes rare terms in document-at-a-time mode, then the remainder in term-at-a-time mode.

The score-at-a-time approach of Anh et al. [2001] represents a hybrid between the term-at-a-time and document-at-a-time approaches. (Anh et al. also showed that the use of integer impacts and the avoidance of floating point computations allowed fast processing or ranked queries, and an integer-based similarity calculation is used in all of the experiments reported here.) To round out our experiments, we thus carried out a final set of runs using Type DS impact-sorted indexes and the same similarity computation, and score-at-a-time processing to handle ranked queries. Table 4 shows the results.

The score-at-a-time regime provides faster query processing than either document-at-a-time or term-at-a-time processing, primarily because of the way the index is structured. In an impact-sorted index, each impact score is followed by a sequence of d-gaps representing documents that all share that impact, and so each pointer that is processed requires $1+\epsilon$ values to be decoded (the ϵ being the shared impact value) rather than the 2 values per pointer that are decoded in a pointer-interleaved document-sorted index. The index is also slightly smaller, and for the GOV2 collection the Type DS impact-sorted index occupies just 1.42% of the source files. Even faster impact-ordered processing is possible if dynamic query pruning is employed [Anh and Moffat, 2006b].

6 Conclusion

We have categorized inverted index structures in a number of ways, including with respect to the information that they contain, the way that information is organized within the index, and the way that the index is used to resolve queries. We have also carried out comprehensive experiments using a 426 GB collection of web documents, and a stream of 1,000 real-world queries.

At one level, the results we have achieved are relatively "intuitive" – it is hardly surprising that non-interleaved or term-interleaved index structures give faster Boolean query throughput than do pointer-interleaved structures. But the extent of the throughput difference is notable, and it is clear from our results that pointer interleaved structures should not be considered for practical implementation, despite their simplicity. In this

respect the careful implementation and experimentation reported in this paper represents a significant and tangible contribution. In addition, the block-interleaved index organization we have introduced allows buffering costs to be controlled, without sacrificing any querying speed. Block-interleaved indexes provide support for both document-at-a-time and term-at-a-time processing, and establish a query evaluation framework against which other proposed developments can be measured.

A key area for ongoing investigation is in the area of dynamic pruning techniques, and document-at-a-time query processing. All of the results presented here are for exhaustive evaluation of queries; further throughput improvements are likely to be possible when pruning techniques appropriate to block-interleaved indexes are tested.

Acknowledgment. This work was supported by the Australian Research Council, the ARC Special Research Center for Perceptive and Intelligent Machines in Complex Environments, and by the NICTA Victoria Laboratory.

References

V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proc. 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 35–42, New Orleans, Louisiana, Sept. 2001. ACM Press, New York.

V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, June 2006a.

V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, Aug. 2006b. ACM Press, New York. To appear.

D. Hawking. Efficiency/effectiveness trade-offs in query processing. *ACM SIGIR Forum*, 32(2): 16–22, Sept. 1998.

H. S. Heaps. *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, 1978.

M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, 17(4):406–439, Oct. 1999.

A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, Oct. 1996.

M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, Oct. 1996.

T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In G. Marchionini, A. Moffat, J. Tait, R. Baeza-Yates, and N. Ziviani, editors, *Proc. 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 219–225, Salvador, Brazil, Aug. 2005. ACM Press, New York.

H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(1):831–850, Nov. 1995.

H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems*, 22(4):573–594, 2004.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999.

J. Zobel and A. Moffat. Inverted files for text search engines. *Computing Surveys*, 2006. To appear.

Adaptive Query-Based Sampling of Distributed Collections

Mark Baillie, Leif Azzopardi, and Fabio Crestani

Department of Computing and Information Sciences,
University of Strathclyde, Glasgow, UK
{mb, leif, fabioc}@cis.strath.ac.uk

Abstract. As part of a Distributed Information Retrieval system a description of each remote information resource, archive or repository is usually stored centrally in order to facilitate resource selection. The acquisition of precise resource descriptions is therefore an important phase in Distributed Information Retrieval, as the quality of such representations will impact on selection accuracy, and ultimately retrieval performance. While Query-Based Sampling is currently used for content discovery of uncooperative resources, the application of this technique is dependent upon heuristic guidelines to determine when a sufficiently accurate representation of each remote resource has been obtained. In this paper we address this shortcoming by using the Predictive Likelihood to provide both an indication of the quality of an acquired resource description estimate, and when a sufficiently good representation of a resource has been obtained during Query-Based Sampling.

1 Introduction

An open problem that Distributed Information Retrieval systems (DIR) face is how to represent large document repositories, also known as resources, both accurately and efficiently. To facilitate resource selection, the process of assessing which collections contain relevant information with respect to a user's information request, a description of each information resource a DIR service searches is required. The obtained resource descriptions form a collection selection index that enables the DIR system to determine which online collections to search given a query [6]. Therefore, obtaining precise resource descriptions is an important phase as the quality of such representations will impact on resource selection accuracy, and ultimately retrieval performance. The acquisition and representation of an information resource presents many research challenges, particularly in uncooperative environments. When co-operation from an information resource provider cannot be guaranteed, it is necessary to obtain an unbiased and accurate description of the underlying content with respect to a number of constraints including: costs (computation and monetary), consideration of intellectual property, handling legacy and different indexing choices of the resource provider [6,11]. While Query-Based Sampling is currently used for content discovery of uncooperative resources, the application of this technique

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 316–328, 2006.

is dependent upon heuristic guidelines to determine when a sufficiently accurate representation of each remote resource has been obtained. In this paper we address this shortcoming by using the Predictive Likelihood to provide both an indication of: (i) the quality of an acquired resource description estimate, and (ii) when a sufficiently good representation of a resource has been obtained during Query-Based Sampling.

The remainder of this paper is structured as follows. First, we provide a brief outline of Query-Based Sampling and how it can be used to build resource descriptions, then we outline how Predictive Likelihood can be adopted as a measure of resource description quality with respect to the user information needs (Section 2). Next, we compare Predictive Likelihood to existing measures and show that it provides a comparable indication of resource quality despite the fact no *a priori* knowledge is used (Section 3). Finally, we demonstrate and evaluate the application of Predictive Likelihood in Query-Based Sampling on two DIR testbeds (Section 4). Our analysis validates that this unsupervised approach can substantially reduce the number of documents sampled without detracting from resource selection accuracy. We then conclude the paper with a short discussion detailing the implications of using such an approach and indicate directions for future work (Section 5).

2 Query-Based Sampling and Predictive Likelihood

A resource description is a representation of the content contained within a resource (e.g. a document collection). It can take a variety of forms depending on a number of influencing factors; such as the retrieval model used for resource selection, and the level of co-operation between a search service and information provider. Currently adopted representations include a term vector of counts or probabilities (i.e. a language model) [7], a sample of indexed documents from each collection [14], or indeed the full index [6].

The widely accepted solution for resource description acquisition is Query-Based Sampling (QBS) [7]. During QBS an estimated representation is obtained by submitting random queries to the actual collection, incrementally adding the newly retrieved documents to the estimated resource representation. Queries are randomly selected to ensure that an unbiased resource estimate is achieved. Sampling is then terminated when it is believed a sufficiently good representation of the underlying resource has been acquired, facilitating effective retrieval. Through empirical analysis, the number of documents required to be sampled, on *average*, was estimated to be approximately 300-500. This was believed to obtained a sufficiently good representation of a resource [7]. This threshold was estimated by measuring the estimated resource description against the actual resource using two indicators of quality, and then considering the corresponding retrieval selection accuracy.

While it has been shown that this criterion provides adequate resource selection accuracy under certain conditions, there are potential limitations. A fixed threshold will not always generalise across other collections and environments.

Cases when the blanket application of such a heuristic would be inappropriate include: (i) when the sizes of resources are highly skewed, and (ii) when the resources are heterogeneous. In the former, if a resource is large then undersampling may occur because not enough documents are obtained. Conversely, if a collection is small in size, then oversampling may occur, increasing costs beyond necessity. In the latter case, if the resource is varied and highly heterogeneous then to obtain a sufficiently accurate description would require more documents to be sampled than when a resource is homogenous. For both scenarios, adopting a threshold based heuristic will not ensure a sufficiently good resource description for all resources. This has been recently verified by Shokouhi *et al.* [13] over a number of different DIR testbeds.

Ideally QBS should be curtailed only when a sufficiently good description of the resource has been acquired such that the number of documents sampled is minimised and system performance preserved. In this paper we argue that the Predictive Likelihood of the user's information needs given the estimated resource description can be utilised as a measure of the goodness of a resource description estimate. We believe that the Predictive Likelihood can be used to: (i) provide an indication of the resource description quality, and (ii) to indicate when a sufficiently good representation of the resource has been obtained.

In statistical modelling the log-likelihood of a model on a held out sample of data is often applied as a measure for the "goodness of fit" of that model. This measure is also known as the Predictive Likelihood (PL) of the model [8]. PL is generally used to measure the quality of a language model in the fields of Statistical Language Modelling, but has been more recently applied to estimate language model parameters in text retrieval [1,10,16]. In these studies it has been generally assumed that those models which maximise PL will achieve better retrieval performance. Following this intuition, in the context of measuring description quality, we aim to maximise the PL of the user's information needs given the estimated resource description. By using PL we are measuring how representative each distributed information resource is when compared to the known (typical) information needs of the users of the DIR system. This is a departure from the original QBS assumption that a resource description should be a sufficient sample of the actual entire collection. Instead, by using PL descriptions are measured with respect to the information needs of the users of the system. Before discussing this main difference, we first define the Predictive Likelihood measure and how it incorporates the user's information needs.

Formally, given a sequence of queries $Q = \{q_{ij} : 1, \ldots, N; 1, \ldots, M\}$, where q_{ij} is the j^{th} term of the i^{th} query, which corresponds to a particular term t in the estimated resource description $p(t = q_{ij}|\hat{\theta})$. The likelihood of a resource description estimate $\hat{\theta}$ generating Q is given by the conditional probability:

$$p(Q|\hat{\theta}) = \prod_{i=1}^{N} \prod_{j=1}^{M} p(t = q_{ij}|\hat{\theta})$$

where,

$$p(t|\hat{\theta}) = \frac{n(t,\hat{\theta})}{\sum_{t' \in \hat{\theta}} n(t',\hat{\theta})}$$

and $n(t,\hat{\theta})$ is the number of times term t occurs in the resource estimate $\hat{\theta}$. We engage the standard assumption of independence between query terms and also between queries [16]. For computational connivence, however, we use the Predictive Log Likelihood of the estimated resource $\hat{\theta}$:

$$\ell(\hat{\theta}, Q) = \log p(Q|\hat{\theta}) = \sum_{i=1}^{N} \sum_{j=1}^{M} \log p(t = q_{ij}|\hat{\theta})$$

Using this approach for measuring the quality of a resource description is fundamentally different to existing standard approaches. Current methods measure the quality of an estimate against the actual resource, thus requiring full collection knowledge *a priori*. As mentioned previously such information is not readily available except in artificial or simulated environments. In comparison, PL requires that a set of queries Q are available for evaluating each resource description instead of the actual collection. Therefore, the selection of this set of queries is an important step in training the DIR system.

We assume that the set of queries Q are representative of the information needs of the users of that system. To elaborate, these information needs, or queries, can take the form of: (i) the previous interactions of the system obtained through the query logs [2] or (ii) profiles that represent the interests of the user-base, similar to profiles used in Information Filtering systems [4]. In the former, a query set consistent with the information needs of the user-base of the system can be obtained from query logs. For instance, the query logs of each user can be mined to extract a representative set of queries. Alternatively, if no historical queries are freely available, it is possible to access example queries from Information Retrieval test collections or a similar web based corpus. Conversely, or even supplementary, users of the system could be profiled explicitly, such as through a questionnaire or survey, where profiles represent typical topics, subject areas and tasks that the users of the system will undertake. However, both solutions for representing Q enable the DIR system to be tuned either towards an *average* user-base or even tailored towards specific users or user groups depending on the requirements of the system. Throughout the development of the system, Q can also be re-assessed with respect to the user's dynamically changing needs.

3 Predictive Likelihood as an Indicator of Quality

In this section PL is evaluated and compared as a measure of resource description quality alongside a currently adopted method. In this experiment we are motivated to evaluate whether a relationship exists between PL and the currently applied measure. If a relationship does exist, this will provide evidence that PL can be utilised as a surrogate measure of resource description quality

with the added advantage that PL does not require *a priori* knowledge of the underlying information resource statistics.

3.1 Existing Measures of Resource Description Quality

Current measures of resource description quality include the Collection Term Frequency ratio (CTF), Spearman Rank Correlation Coefficient (SRCC) [7], and the Kullback-Leibler (KL) divergence [3,11]. CTF and SRCC are normally applied in tandem, where the former provides an indication of the percentage of terms seen, while the latter is an indication of term ranking order, although neither consider the term frequency which is an important information source for all resource selection algorithms. In a recent study, the SRCC measure was shown to be unstable and unreliable [3]. As an alternative measure the KL divergence was proposed. With respect to the goal of measuring the quality of a resource description the KL divergence is appealing for a number of reasons. The term probability distributions of the actual and estimated resource descriptions capture the relative (or normalised) term frequencies, when an accurate estimation of such information is pertinent to many of the state of the art resource selection algorithms [6,11,14,15]. It also fulfils the criteria set forth in the original QBS study by Callan and Connell [7] of measuring the correspondence between the estimated and actual resource vocabulary while not overly weighting low frequency terms (CTF), and also measuring the correspondence between the estimated and actual frequency information (SRCC). Essentially the KL divergence measures this phenomena precisely, resulting in a more stable and precise measure in comparison to the surrogate indicators CTF and SRCC.

We therefore compare the KL against the PL. In this experiment we hypothesise that the PL will provide a comparable indication of the resource description quality to KL.

3.2 Kullback-Leibler Divergence

The Kullback-Leibler Divergence (KL) provides a measure for comparing the difference between two probability distributions[12]. When applied to the problem of resource description quality, KL measures the relative entropy between the probability of a term t occurring in the actual resource θ (i.e. $p(t|\theta)$), and the probability of the term t occurring in the resource description $\hat{\theta}$, i.e. $p(t|\hat{\theta})$. Formally, the KL Divergence is defined as:

$$KL(\theta|\hat{\theta}) = \sum_{t \in V} p(t|\theta) log \frac{p(t|\theta)}{p(t|\hat{\theta})}$$

where, $p(t|\theta) = \frac{n(t,\theta)}{\sum_{t \in \theta} n(t,\theta)}$, $p(t|\hat{\theta}) = \frac{\sum_{d \in \hat{\theta}} n(t,d)+\alpha}{\sum_t (\sum_{d \in \hat{\theta}} n(t,d)+\alpha)}$, $n(t,d)$ is the number of times t occurs in a document d and α is a small non-zero constant (Laplace smoothing). The smaller the KL divergence, the more accurate the description, with a score of zero indicating two identical distributions. To account for the

Table 1. Collection Statistics

Collection	# Documents	# Collection Terms	# Unique Terms	Mean Doc. Length
Aquaint	1,033,461	284,597,335	707,778	275
WT10g	1,692,096	675,181,452	4,716,811	399

sparsity within the set of sampled documents, Laplace smoothing is applied to alleviate the zero probability problem and to ensure a fair comparison between each estimated resource description.

3.3 Experimental Methodology

Our aim is to evaluate whether PL provides a similar indication of the *true quality* of a resource description estimate. Here, we assume that the KL divergence is the true measure of quality because it's measurement is taken against the actual resource description (ground truth). Our hypothesis is that for a set of estimated resource descriptions the PL measure will rank these estimated resource descriptions in the same order as the KL measure. If this is the case then PL will provide a comparable indication of the quality of that resource according to the KL measure.

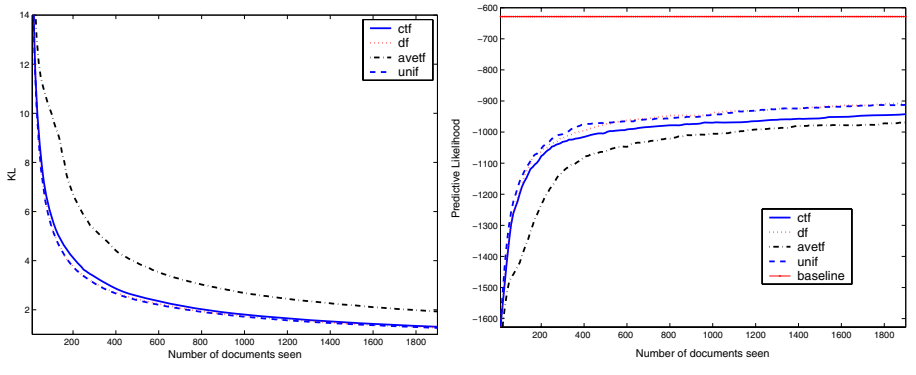


Fig. 1. Measuring the quality of resource description estimates obtained from Aquaint collection by the four QBS approaches. KL and PL measurements for each sampling approach are displayed as the number of documents sampled increases.

The experiments were performed on several different TREC collections, with varying characteristics. For brevity, though, we only report on two of these collections, the news collection Aquaint, and the Web collection WT10g (See Table 1).

Estimated resource descriptions were then created for these collections using QBS as follows:

1. A term is randomly selected from an unrelated vocabulary and is used as the first query for sampling.

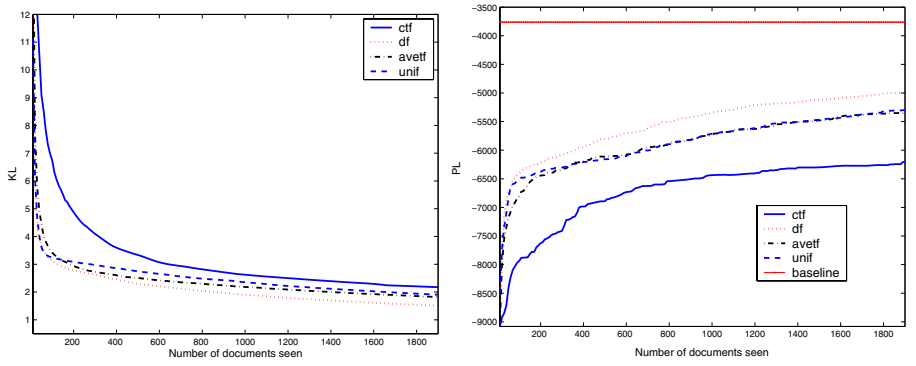


Fig. 2. Measuring resource description estimates obtained from the WT10g collection

2. The resource is queried and the top four documents returned are added to the estimated resource description.
3. The KL and PL are measured and recorded.
4. The next query is generated using the currently estimated resource description using one of the four sampling strategies: the collection frequency (*ctf*), the document frequency (*df*), the average term frequency (*avetf*), or randomly (*unif*) [7].
5. If the stopping criterion has not been satisfied, return to step (2).

We continued sampling until we obtained 2000 documents. For each sampling strategy the entire process was repeated 25 times because the initial term affects the quality of the resource description. This generated 100 estimated resource descriptions for each collection along with the corresponding measurements. The query set to compute PL for both collections consisted of TREC Topics 1-200. The title field from these topics were extracted as queries which formed Q for each collection respectively.

3.4 Experimental Results

Resource description quality. Figures 1 and 2 summarise the performance of each sampling strategy by displaying the mean quality score over the 25 runs for the Aquaint and WT10g collection respectively. In the KL plots, a score of zero indicates that the estimate is identical to the actual description. While in the PL plots, the higher the PL the better the quality, where the baseline is shown as a solid line which denotes the PL score for Q given the actual resource description θ.

We were first concerned with the rate of improvement as more documents were added to each resource description estimate. The general trend when measuring the quality of resource descriptions, as further documents were sampled, appeared to be similar (See Figures 1 and 2). As the number of documents initially sampled increased, a sharp drop in KL and a corresponding rise in PL,

Table 2. The Kendall τ Correlation of KL and PL for ranking resource description estimates in terms of quality, recorded at different intervals of documents sampled. An asterisk indicates a statistically significant correlation at $p < 0.05$.

	Documents Sampled			
Collection	**200**	**500**	**1000**	**2000**
Aquaint	0.85*	0.68*	0.62*	0.53*
WT10g	0.38*	0.51*	0.57*	0.82*

was found. As QBS sampling continued, the rate of improvement for each resource description levelled out using either measure. This trend indicated that by adding further documents to the estimate provided small gains in quality. At this point, a decision to terminate QBS based on the cost of sampling further documents versus the gain in further representation of the resource should be made. For the Aquaint collection, Figure 1, this point occurs when approximately 800-1200 documents are sampled across all term selection methods. For the WT10g collection, Figure 2, this was found at approximately 1200-1600 for KL and somewhat later when measuring with PL.

We were also concerned with which term selection method (i.e. *df, unif, ctf* or *avetf*) acquired the better resource description estimates in terms of KL and PL, and in particular if there was agreement between both measures. Focusing first on the Aquaint collection, the ordering of the mean quality of each sampling strategy was found to be identical when using KL and PL. For both measures the rank order was: *unif*, *df*, *ctf* then *avetf* (best to worst). For the WT10g collection, both measures also ranked the methods the same: *df*, *unif*, *avetf* followed by *ctf*.

Both KL and PL ranked the resource descriptions obtained from each term selection method in the same order. However, across the two collections this rank order varied with the random term selection method (*unif*) preferred for Aquaint, while the document frequency strategy (*df*) considered better for WT10g. This is an unexpected outcome as it reveals that PL can be used in a novel way for determining which sampling method will provide a better estimate on a per collection basis, potentially increasing sampling effectiveness during QBS.

Correlation between the measures. We ranked all the estimated resource descriptions, irrespective of term selection strategy, according to KL and PL producing two ranked lists. We then compared the ranked lists produced by each measure using Kendall's τ correlation test at various points in the sampling process. By doing so, we could determine if there was a strong concordance between the rankings (i.e. quantify how close in agreement each measure is when ranking the different resource description estimates in terms of quality). This approach has been used previously in IR to compare different measures of retrieval performance in [5]. The assumption is that a good estimate would be ranked highly for both measures. A correlation score close to 1 would indicate that two measures have identical rankings. A score closer to 0 would indicate no

relationship between the measures. Table 2 provides the τ correlation coefficient at 200, 500, 1000 and 2000 documents sampled.

At each of the different sampling points, shown in Table 2, the results reveal that there was a close agreement between both ranked lists complied using both measures. This relationship was found to be statistically significant across both collections, and at each of the different intervals, providing stronger evidence to support our hypothesis that the PL measure provides a comparable indication of quality with respect to the KL measure.

4 Predictive Likelihood as a Stopping Criterion

QBS is an iterative process where sampling is curtailed when a single or set of stopping criteria has been reached. In the standard approach to QBS, once n unique documents have been retrieved then sampling is stopped [7]. We propose to use the PL measure to inform the decision making process in order to decide when enough documents have been sampled. Our stopping criterion is based on the difference in the PL score for the estimated resource description, between the previous iteration $k-1$ and the current iteration k of the sampling process. The difference ϕ_k at iteration k, where $k > 1$, is given by:

$$\phi_k = \ell(\hat{\theta}_k, Q) - \ell(\hat{\theta}_{k-1}, Q) = log\left(\frac{p(Q|\hat{\theta}_k)}{p(Q|\hat{\theta}_{k-1})}\right)$$

where $\hat{\theta_k}$ is the resource description estimate at the k^{th} iteration. If ϕ_k is below a threshold ϵ, then sampling is curtailed, where ϵ indicates the necessary amount of improvement required to continue sampling. By doing so we are using a gradient ascent optimisation to maximise the Predictive Likelihood of the estimated resource description given Q [9]. The ratio of PL scores provides an indication of the rate of improvement over the previous iteration. Consequently, the free parameter ϵ is independent of the document collection characteristics (such as size and heterogeneity). Unlike the fixed n document curtailment strategy, this parameter is generalisable to other collections.

By using this technique we believe that a sufficiently good estimation of the resource will be obtained, which will minimise any unnecessary wastage from oversampling, and will also avoid obtaining an insufficient sample through undersampling. We further hypothesise that because sufficient representations of each resource will be obtained, this will translate into better selection accuracy over the fixed method. We shall now refer to the proposed method as QBS-PL and the previous threshold based approach as QBS-T.

4.1 Evaluation

The aim of the next set of experiments was to determine whether QBS-PL provided better resource selection accuracy over QBS-T. This was examined in two ways: (1) if QBS-PL improved selection accuracy when the number of sampled

documents were approximately equal, and (2) if QBS-PL provided comparable resource selection accuracy to QBS-T when the number of sampled documents were substantially less than the threshold approach.

Experimental Settings. Two DIR testbeds based on the TREC Aquaint collection were formed for these experiments, with the documents partitioned By-source and By-topic. The By-source testbed contains 112 simulated collections, with the documents arranged into collections based on both the news agency that published each document, and the month the document was published. In this testbed the size of each collection is uniform. The By-topic testbed contains 88 collections, with documents grouped by topical similarity using single pass *k-means* clustering. In this testbed, collection sizes are skewed and represent a realistic setting with respect to the distribution of content.

For QBS, sampling was performed with the term selection strategy set to *df*, with four documents retrieved per query. The thresholds used for the QBS-T ranged from 100-1000 unique documents. For QBS-PL, ϵ was set to 0.01. We also include descriptions using the full collection information ('complete') as a benchmark (i.e. all the documents in the resource to build the description). To provide an indication of how sensitive the retrieval accuracy is when applying QBS-PL with different query sets, we used four different sized query sets Q constructed from 200 TREC Topics (Topics 1-200). The number of queries in each set were 50, 100, 150 and 200. So as not to train and test using the same set of queries, another set of queries from the TREC HARD 2005 track were used for resource selection. This set contained 50 test topics, where the title field was used as the query. Resource selection was performed using the DIR benchmark algorithm CORI [6]. Resource selection accuracy was measured using the recall-based $\hat{R}$ metric. $\hat{R}$ is a measure of the overall percentage of relevant documents contained in the top r collections [7]. We measured $\hat{R}$ at $r = \{5\%, 10\%, 15\%, 20\%, 25\%\}$ of all collections searched. We also captured the average number of documents sampled per collection, and the total number of documents overall.

Experimental Results. Table 3 provides an overview of the results obtained for QBS-PL, QBS-T and also using the complete estimates (not all thresholds for QBS-T are shown). Figure 3 is a plot of QBS-PL using 200 queries, QBS with two document thresholds ($t = 500, 1000$), and the resource selection performance using complete information, compared across each of the $\hat{R}@r$ values.

The performance of the QBS-PL method varied as the size of the Q increased, see Table 3. For both collections, an increase in the Q coincided with an improvement in $\hat{R}$, across all collection cut-offs, with the QBS-PL $Q = 200$ method performing best over both testbeds. In both cases, there was a steady increase in the number of documents sampled as the size of Q grew. We suspect that this would tail off as more queries are added, but this is as yet unconfirmed due to the finite number of test queries available. As more queries are added to Q, it is sensible to expect more documents will be sampled in order to cover the new subject areas expressed in these queries. This is intuitively appealing because as the information needs of the users of a system diversify and change, a larger

Table 3. Each technique is evaluated by $\hat{R}@r$ percent of the collections searched, and the overall document statistics for each QBS technique across the two testbeds

Parameters	$\hat{R}$@5%	$\hat{R}$@10%	$\hat{R}$@15%	$\hat{R}$@20%	$\hat{R}$@25%	Ave. docs.	Total docs.
Aquaint: By-source testbed							
QBS-PL, $Q = 50$	0.093	0.162	0.231	0.283	0.341	247.9	27767
QBS-PL, $Q = 100$	0.110	0.182	0.248	0.301	0.366	347.3	38893
QBS-PL, $Q = 150$	0.116	0.188	0.250	0.308	0.360	434.8	48699
QBS-PL, $Q = 200$	**0.126**	**0.212**	**0.279**	**0.332**	**0.378**	**500.6**	**56066**
QBS-T $n = 300$	0.108	0.179	0.248	0.308	0.360	300	36960
QBS-T $n = 500$	**0.113**	**0.191**	**0.249**	**0.310**	**0.362**	**500**	**56000**
QBS-T $n = 1000$	0.124	0.207	0.291	0.353	0.415	1000	112000
Complete	0.163	0.249	0.315	0.390	0.454	11743.9	1033461
Aquaint: By-topic testbed							
QBS-PL, $Q = 50$	0.63	0.73	0.79	0.83	0.85	235.2	20466
QBS-PL, $Q = 100$	0.64	0.74	0.82	0.84	0.87	344.2	29952
QBS-PL, $Q = 150$	0.65	0.74	0.82	0.85	0.87	394.4	34315
QBS-PL, $Q = 200$	**0.66**	**0.75**	**0.82**	**0.85**	**0.86**	**456.4**	**39685**
QBS-T $n = 500$	0.54	0.69	0.75	0.81	0.84	500	44000
QBS-T $n = 1000$	0.58	0.73	0.79	0.84	0.87	1000	88000
Complete	**0.64**	**0.74**	**0.81**	**0.85**	**0.88**	**2262**	**1033461**

number of documents will be required in order to sufficiently describe a resource given those needs.

In comparison with the threshold method, QBS-T, the performance of QBS-PL provides comparable selection accuracy while reducing the number of documents sampled (See Figure 3 and Table 3). If we consider QBS-PL $Q = 200$ on the By-source testbed, the fixed threshold of $n = 500$ returns a similar number of documents sampled, but QBS-T's selection accuracy is worse. It is not until the threshold was increased to $n = 1000$ that similar selection accuracy was

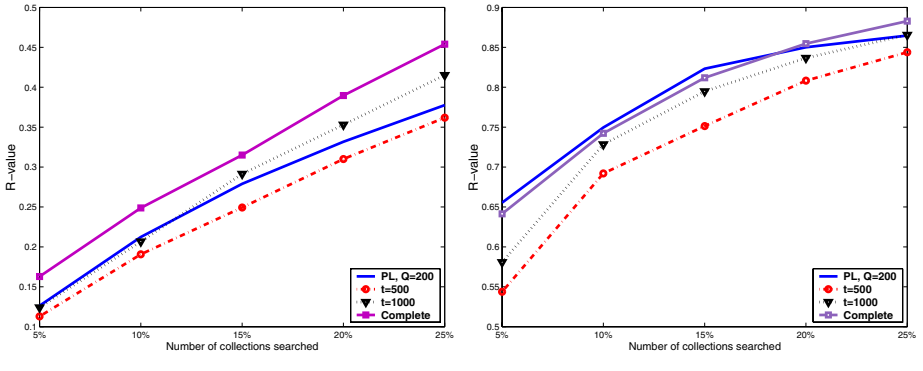


Fig. 3. QBS-PL versus QBS-T across a range of cut off values of r over By-source (left) and By-topic (right) respectively

obtained. However, this means that over 55,000 extra documents were sampled, an increase of almost 100%. On the By-topic testbed, QBS-PL provides better accuracy when compared with the QBS-T estimates. Even when QBS-T was set to $n = 1000$, with an increase of 40,000 to 50,000 extra documents sampled over the QBS-PL estimates, the selection accuracy was still 6-12% worse. It was only when complete information was used that performance similar to QBS-PL $Q = 200$ was obtained. The seems to suggest that there are problems with under and over sampling of many collections, which was not so problematic when collection size was uniform as in the By-source testbed (and in previous work [7]); but is problematic when the collection sizes are skewed.

5 Conclusions and Future Work

Both experiments have shown that PL can be effectively used as a measure of resource description quality, and as a consequence can be integrated into the QBS algorithm. It was shown that a significant relationship exists between PL and KL divergence. However, PL is a radical departure from existing measures such as KL. It is radical because it questions whether a completely unbiased representation of the underlying resource is actually required. By using PL, we are not measuring quality in terms of sampling a sufficiently good representation of the actual collection, but measuring whether the resource description estimate satisfies the information needs of the users of the DIR system. With PL we measure each resource description with respect to a set of queries that represent the typical information needs, Q, of the user-base of a system i.e. evaluating each estimate with respect to the information users want from that resource. For example, by increasing Q, it was highlighted that further documents were required to be sampled before a sufficient representation of the collection was obtained. This increase in the number of documents required to satisfy Q mirrored the addition of new information needs in Q.

We then highlighted that the problem of under and oversampling does exist when employing a QBS algorithm which uses a fixed document threshold (QBS-T). As previously posited, this is especially problematic in a situation when resources are of varying size and content. Consequently, the efficiency and effectiveness of the QBS approach is compromised when using such criteria. By employing QBS-PL, it was shown that this problem can be addressed. Using PL to measure resource description quality without *a priori* knowledge of each distributed collection, the original QBS algorithm was improved both in terms of accuracy and efficiency. QBS-PL minimised the problems of under and oversampling, and in particular when faced with collections of varying size and content, we were able to determine when a sufficiently good representation of each collection had been obtained, which in turn was reflected by performance gains. In contrast, a fixed threshold resulted both in poorer resource selection performance and also increased overheads.

A main advantage of utilising PL is that it enables the resource descriptions to be tailored specifically to the information needs of the user. This is appealing

and paves the way for the development of personalised (distributed) retrieval systems. Defining the query set Q provides an intuitive mechanism for obtaining resource descriptions that are personalised to specific users or user groups; an unexplored area of research that we are currently investigating.

Acknowledgements

This work was supported by PENG, a Specific Targeted Research Project funded within the 6th PF of the European Research Area. More information on the project can be found at `http://www.peng-project.org/`.

References

1. L. Azzopardi, M. Girolami, and C. J. Risjbergen. Investigating the relationship between language model perplexity and IR precision-recall measures. In *Proceedings of the 26th ACM SIGIR conference*, pages 369–370, 2003.
2. R. A. Baeza-Yates. Applications of web query mining. In *Proceedings of the 27th ECIR*, pages 7–22, Santiago de Compostela, Spain, 2005.
3. M. Baillie, L. Azzopardi, and F. Crestani. Towards better measures: Evaluation of estimated resource description quality for distributed IR. In *First International Conference on Scalable Information Systems*. IEEE CS Society, 2006.
4. N. J. Belkin and W. B. Croft. Information filtering and information retrieval: two sides of the same coin. *Communications of the ACM*, 35(12):29–38, 1992.
5. C. Buckley and E. M. Voorhees. Evaluating evaluation measure stability. In *Proceedings of the 23rd ACM SIGIR conference*, pages 33–40, 2000.
6. J. P. Callan. *Advances in information retrieval*, chapter Distributed information retrieval, pages 127–150. Kluwer Academic Publishers, 2000.
7. J. P. Callan and M. Connell. Query-based sampling of text databases. *ACM Transactions of Information Systems*, 19(2):97–130, 2001.
8. M. H. Degroot. *Optimal Statistical Decisions (Wiley Classics Library)*. Wiley-Interscience, April 2004.
9. R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience Publication, 2000.
10. T. Hofmann. Unsupervised learning by probabilistic latent semantic analysis. *Machine Learning*, 42(1-2):177–196, 2001.
11. P. G. Ipeirotis and L. Gravano. When one sample is not enough: improving text database selection using shrinkage. In *Proceedings of the ACM SIGMOD Conference*, pages 767–778, 2004.
12. S. Kullback. Information theoery and statistics. *Wiley, New York*, 1959.
13. M. Shokouhi, F. Scholer, and J. Zobel. Sample sizes for query probing in uncooperative distributed information retrieval. In *APWeb 2006*, volume 3841, pages 63–75. Springer Lecture Notes in Computer Science, 2006.
14. L. Si and J. P. Callan. Modeling search engine effectiveness for federated search. In *Proceedings of the 28th ACM SIGIR Conference*, pages 83–90, 2005.
15. J. Xu and W. B. Croft. Cluster-based language models for distributed retrieval. In *Proceedings of the 22nd ACM SIGIR conference*, pages 254–261, 1999.
16. C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Transaction of Information Systems*, 22(2):179–214, 2004.

Dotted Suffix Trees
A Structure for Approximate Text Indexing

Luís Pedro Coelho* and Arlindo L. Oliveira

INESC-ID/IST
{luis, aml}@algos.inesc-id.pt

Abstract. In this work, the problem we address is text indexing for approximate matching. Given a text $\mathcal{T}$ which undergoes some preprocessing to generate an index, we can later query this index to identify the places where a string occurs up to a certain number of errors k (edition distance). The indexing structure occupies space $\mathcal{O}(n \log^k n)$ in the average case, independent of alphabet size. This structure can be used to report the existence of a match with k errors in $\mathcal{O}(3^k m^{k+1})$ and to report the occurrences in $\mathcal{O}(3^k m^{k+1} + ed)$ time, where m is the length of the pattern and ed and the number of matching edit scripts. The construction of the structure has time bound by $\mathcal{O}(kN|\Sigma|)$, where N is the number of nodes in the index and $|\Sigma|$ the alphabet size.

Keywords: string algorithms, suffix trees, approximate text matching, text indexing.

1 Introduction

Since their introduction [1], suffix trees have been one of the methods of choice for text indexing. However, in many real-life problems one is interested in finding places in the text where an approximate form of the pattern occurs. In 2001, Navarro et al presented a survey of existing approaches to solving this problem [2]. More recently, Maaß [3] presents both a survey of other work and his own solution, which occupies, on average, $\mathcal{O}(|\Sigma|^k n \log^k n)$[1] space for a search time $\mathcal{O}(m)$. In this work we present an approach based on an extension of suffix trees. The main advantage of this approach is that both the search and the index size are *alphabet independent* (although the indexing time is not).

The structure presented here is superficially very similar to the one presented by Chattaraj [4] as an *inexact suffix tree*, but that work has different objectives. Cole el al [5] present a structure whose initial intuition resemble ours in that it involves error trees. However, they make different time and space tradeoffs to achieve $\mathcal{O}(m + \log\log n + occ)$ searching (Hamming distance) with a size $\mathcal{O}(n \frac{\log^k n}{k!})$ index.

* Supported by the Portuguese Science and Technology Foundation, project POSI/SRI/47778/2002 BioGrid.

[1] Maaß considers $|\Sigma|$ and k as constant and presents $\mathcal{O}(n \log^k n)$ as a complexity result. However, this analysis ignores the potentially large impact of alphabet size.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 329–336, 2006.

2 The Indexing Structure

Definition 1 (Character, string). *Given a set Σ, we say that S is a* string over Σ *if S is a (possibly empty) sequence of elements of Σ. Elements of Σ will be called* characters. *The length of the string S will be denoted by $|S|$. We shall write S_i for the i-th element of S.*

The set of all strings is denoted by Σ^* and $\Sigma^+ = \Sigma^* - \{\text{empty string}\}$.

For denoting characters we shall use letters from the beginning of the roman alphabet (a, b, c,...) and, for strings, we shall use letters from the end of the alphabet (w, x, ...). In what follows we assume that there are two special symbols (\$ and .) which are not part of Σ.

Definition 2 (Concatenation, Prefix and Suffix). *wx or aw will denote the usual concatenation operation. If $S = wxy$, then w is a* prefix *of S, x is a* substring *of S and y is a* suffix *of S (at position $|wx|$).*

Definition 3 (Patricia tree, Suffix Tree, Suffix Link). *T is a Patricia tree if T is a rooted tree with edge labels from Σ^+. For each $a \in \Sigma$ and every node n in T, there exists at most one edge leaving n whose label starts with a. Each node in a Patricia tree has a path leading to it which forms a string. If the node n has the leading path w, we shall also refer to n as $\overline{w}$. A* compact *Patricia tree omits nodes with just one child.*

A suffix tree *for a string S is a compact Patricia tree whose leaf nodes (those without children) have paths corresponding to all suffixes of the string $S\$$. A suffix link in a suffix tree is a link from the node $\overline{aw}$ to the node $\overline{w}$. This link has the label a.*

In a suffix tree, all internal nodes have a well defined suffix link. McCreight's [6] algorithm constructs a suffix tree with suffix links in linear time.

Definition 4 (Occurrence Set, Position Set). *Given a node $\overline{w}$ in a suffix tree, we call its* occurrence set *the set of indexes in the original string where the string w occurs.*

Given a node $\overline{w}$ in a suffix tree, its position set *is the set formed by taking its occurrence set and adding the length of w to each element.*

Lemma 1 (Position set at the suffix node). *Given two nodes $\overline{aw}$ and $\overline{w}$, if one takes the position set of $\overline{w}$, subtracts one from each element, one obtains a superset of the position set of $\overline{w}$. The items shared by both sets are those positions of the string which contain an a.*

The lemma is fairly obvious given that the position set of $\overline{aw}$ contains all the positions where aw occurs which are exactly those positions where w occurs preceded by a.

Definition 5 (Aproximate Match). *We say that the string s matches the string t at position p with k errors if we can make k modifications in s to obtain s' which is a substring of t at position p. A modification is either deletion, insertion or substitution of one character.*

Definition 6 (Error Tree). *For any node $\overline{w}$, its error tree is formed by taking its position set, adding one to each element and forming the Patricia tree of the suffixes starting at those positions. If the position set includes the end of the string, that element is removed.*

The leaves are labeled by the position of the string in which their paths occur minus $|w| + 1$.

Definition 7 (1-error dotted Tree). *A* 1-error dotted tree *is the tree which is formed by adding to each node in a suffix tree, a new edge labeled by $\cdot$ which points to its error tree. The edge labeled $\cdot$ shall be called a* dot link.

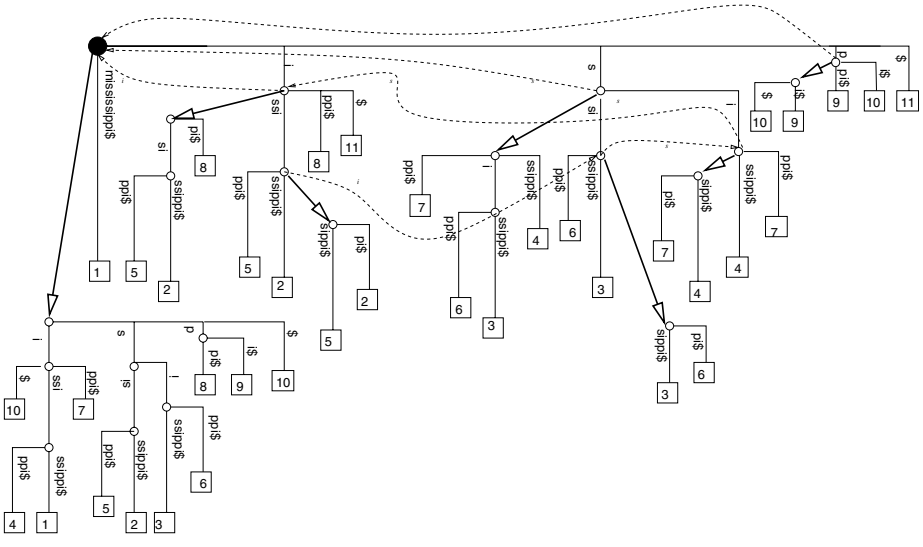


Fig. 1. 1-error dotted tree for *mississippi*

The 1-error dotted tree for *mississippi* is shown in Figure 1. The nodes are connected to their error trees by thick diagonal links. We can see some examples of the concepts above: for the node $\overline{issi}$, the occurrence set is $\{2, 5\}$ and its position set is $\{6, 9\}$. In a sense, one can say that *being* at node $\overline{issi}$ is being at positions 6 and 9 simultaneously. The error tree (at $\overline{issi}$) is formed by taking the strings starting at positions $\{7, 10\}$ (ie, *sippi$* and *pi$*) in a Patricia tree. In a leaf, the occurrence set is a singleton, and we label the leaf by its element.

The paths in the dotted tree are paths in the extended alphabet $\Sigma \cup \{., \$\}$. The notions of *occurrence set*, *position set* and *error tree* are valid for all nodes in a dotted tree.

Definition 8 (k-error dotted tree). *We define a k-error dotted tree as the tree obtained by adding error trees to each node in the $(k-1)$-error dotted tree which does not already contain one.*

3 Searching

Given a pattern to search for, we follow it character by character, descending the tree. We represent this walk by keeping a node and an offset from the start of its incoming link. Inside an edge, we consider that there is an implicit dot link which goes forward one character. At each point, we can take four possible actions: (1) **match**, where we descend according to the pattern (may not be possible); (2) **substitution**, where we follow the dot link (possibly implicit), moving in the pattern; (3) **insertion**, where we follow the (possibly implicit) dot link, *not moving* in the pattern; (4) **deletion**, where we advance in the pattern, while not moving in the tree. We limit ourselves to at most k non-matching operations (editions). Algorithm 1 implements the process just described.

Algorithm 1. Function findString($\overline{w}$, offset, s, k)

```
   Input: Current node w̄
   Input: Current offset offset
   Input: String s
   Input: Maximum errors k
   Data: The tree's string treeString
 1 if k < 0 then return string not found
 2 if s is empty then  report all w̄'s children
 3 findString(w̄,offset,s + 1,k − 1)// deletion
 4 if offset = length(w̄) then
 5 |   findString(w̄.dotLink, 0, s,k − 1)// insertion
 6 |   findString(w̄.dotLink, 0, s + 1,k − 1)// substituition
 7 |   child ←w̄.getSon(s_0)// try matching
 8 |   if child isn't null then findString(child, 0, s + 1,k)
 9 else
10 |   findString(w̄,offset +1,s,k-1)// insertion
11 |   if s_0 ≠ treeString_{start(w̄)+offset} then  k ←k − 1
12 |   findString(w̄,offset +1,s + 1,k)// either match or substituition
```

There are at most $\sum_{i=1}^{k} \binom{m}{i} = \mathcal{O}(m^k)$ ways to combine k edit operations into a string of size m. Since there are 3 operations (substitution, insertion, and deletion), we have at most $\mathcal{O}(3^k m^k)$ sequences. Each sequence has at most $m + k = \mathcal{O}(m)$ elements and therefore the total time to find matches is $\mathcal{O}(3^k m^{k+1})$. Once a match has been found in the tree, reporting the leaves below the node takes time proportional to the number of leaves, ie. to the number of edit scripts which can be used to match the pattern to a substring of the text (which can be greater than the number of occurrences).[2] The total search time is $\mathcal{O}(3^k m^{k+1} + ed)$.

[2] As often happens, strings of the form a^m serve as examples of pathological behaviour as they can match any position of a string of form a^n in a large number of ways.

4 Constructing the Dotted Tree

We start with a suffix tree and show first how to construct a one-error dotted tree. We construct the error tree for the root which is almost a copy of the entire tree, except for two properties: (1) it does not have the leaf labeled 1 in the original tree and; (2) for any other leaf $\overline{w\$}$ occurring at position p in the string, we have a new leaf $\overline{.w\$}$ which occurs at position $p-1$ in the string. For any other node $\overline{aw}$, the error tree is a copy of the error tree at node $\overline{w}$ (the node pointed to by node $\overline{aw}$'s suffix link) with the following changes: (1) the leaf labeled 1 in the original error tree is not included; (2) leaves in the copy have a label which is the original value minus one; (3) a leaf labeled p is included only if $s_{p-1} = a$.

Algorithm 2. Copying a sub tree

```
   Input: A node in a suffix tree w̄
   Input: An optional character a (not given when copying the root)
   Data: The original string string
 1 copy ←make-copy(w̄)
 2 if w̄ is a leaf then
 3 |   p ←w̄.label
 4 |   if p = 1 then return null
 5 |   if a was not given or string_{p−1} = a then
 6 |   |   copy.label ←copy.label - 1
 7 foreach n ∈ w̄.sons do
 8 |   copy of son ←copySubtree(n,a)
 9 |   if copy of son isn't null then
10 |   |   copy.sons ←copy.sons ∪ copy of son
11 if copy.sons is empty then return null
12 if copy.sons has only one element then
13 |   merge copy.sons into copy and return that
14 return copy
```

These conditions are an expression of Lemma 1 and an extension of the conditions for the root. Both are implemented by Algorithm 2. The only point to note is line 12. Since we filter some leaves, we can create nodes with only one child. These are removed by merging a child with its (single) parent. Since a typical suffix tree implementation just stores, at each node, indices to the start and end of the subtring labeling its incoming edge, merging is achieved by adjusting the start index. The construction of the tree using either Ukonnen's or McCreight's algorithm assures that this operation is correct.

Copying a tree takes time proportional to the number of nodes it contains. The error tree at the root is a straightforward copy of the whole tree. Every other error tree is a copy of an existing one. Since each node can have at most $|\Sigma|$ incoming suffix links, each error tree is transversed at most $|\Sigma|$ times. The sum

of all these operations is therefore bounded by $|\Sigma|N$. Therefore, if the number of nodes in the final tree is N, construction is done in time $\mathcal{O}(N|\Sigma|)$.

The above algorithm can be used to construct trees with any number of errors by iterating it. To construct the $(k+1)$-error tree from the k-error tree, make an adjusted copy of the tree as above (adjusting leaves and filtering the leaves with label 1) and make this the new root error tree. Then, for every other node, remove the current error tree. Finally, for every node except the root, construct its error tree as above.

Let N_k be the number of nodes of the k-error dotted tree. We will use N for N_k if k is known from context. The analysis above remains valid and we now have that the time cost is $\mathcal{O}(N_1|\Sigma| + \ldots + N_k|\Sigma|) = \mathcal{O}(kN|\Sigma|)$.

5 Space Considerations

Let l be the maximum string depth of any node in the tree.[3] We show $N_k = \mathcal{O}(nl^k)$ by induction. It is known that $N_0 = \mathcal{O}(n)$. The algorithm for turning a k-error into a $(k+1)$-error dotted tree, can be looked at the following way[4]. First it constructs the error tree at the root and clears all the other error trees. Then it proceeds in stages, making a (possibly incomplete) copy of this tree spread amongst the nodes at string-depth 1. It processes the other nodes in increasing string-depths. At each string depth, the number of nodes is increased by a maximum of N_k. Therefore, we start with N_k nodes, make an almost full copy, and copy that at most l times, $N_{k+1} = \mathcal{O}(N_k(l+1))$. Assuming $N_k = \mathcal{O}(nl^k)$ by induction we conclude $N_{k+1} = \mathcal{O}(nl^{k+1})$.

So far, we have achieved little since in the worst case $l = n - 1$ (consider $aaaa\ldots$). However, under very general assumptions (which natural language textes and DNA experimentally verify), the expected case is $l = \mathcal{O}(\log n)$ [7] and we have $N_k = \mathcal{O}(n \log^k n)$.

6 Experimental Results

Three data sets were used: English text, the DNA of yeast, and randomly generated text. Results on all sets are qualitatively similar.

To experimentally verify the average case prediction, we show in Figure 2 the ratios between the k-error and the $(k+1)$-error dotted trees regarding the number of nodes in the trees. We can easily see that the experimental values do resemble a logarithm as predicted.

Searches were then performed on top of previously indexed text. We only report whether the string exists in the text (and not all occurrences). Therefore, the number of occurrences has no influence on the search time. Figure 3 shows

[3] For a node $\overline{w}$, its string depth is $|w|$.

[4] Having the node processed in this order is, in fact, difficult to code for. However, as an analysis tool, it is a valid assumption.

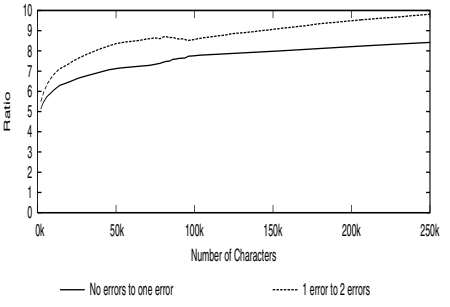


Fig. 2. Size ratio on English text

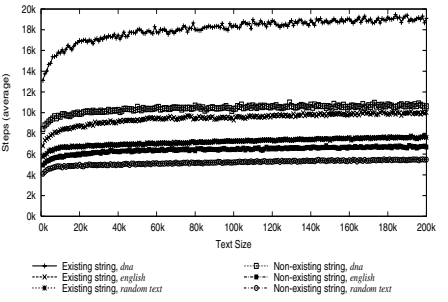


Fig. 3. Searching with 2 errors

the results of searching for 15 character long patterns with 2 errors, while varying the text size. After an initial small growth explainable by the increasing density of the tree, the search time is roughly constant.

7 Conclusions

We presented an indexing structure for approximate text matching which takes, on average, $\mathcal{O}(n \log^k n)$ space. This complexity was predicted theoretically and observed experimentally. This structure reports the existence of a match in $\mathcal{O}(3^k m^{k+1})$ and reports the positions where the matches occur in $\mathcal{O}(3^k m^{k+1} + ed)$ time. It can be constructed in $\mathcal{O}(kN|\Sigma|)$ time, N being the actual number of nodes. The structure and the algorithms to construct it are simple and easy to implement. The fact that the structure uses $\mathcal{O}(ed)$ time (instead of $\mathcal{O}(occ)$) to report the occurrences of a pattern may be a disadvantage in some applications. In other applications (eg, searching in DNA strings for degenerated occurrences of long strings), this will not be a problem since each occurrence will, in general, correspond to only one edit script.

The amount of space the index takes might limit its applicability. One direction for tackling this problem is the following remark: in the example for the string *mississippi*, presented in Figure 1, one can see that the tree below *s.i* and *ssi* are exactly the same. Whether such occurrences are the basis for a significant space saving and how to exploit them is an open question. Going further, the definition of error trees might be extended to structures such as the suffix-DAG presented by Gusfield [8, § 7.7].

Another limitation that should be addressed in future work is related with the fact that the complexity for reporting occurrences depends on the number of edit scripts, and not on the number of occurrences.

Acknowledgments. We thank L. Russo and S. Madeira for several productive discussions.

References

1. Weiner, P.: Linear pattern matching algorithms. In: FOCS, IEEE (1973) 1–11
2. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys **33** (2001)
3. Maaß, M.G., Nowak, J.: Text indexing with erros. In: Proc. 16th Annual Symp. on Combinatorial Pattern Matching (CPM). Volume 3537 of LNCS., Springer (2005) 21–32
4. Chattaraj, A., Parida, L.: An inexact-suffix-tree-based algorithm for detecting extensible patterns. Theor. Comput. Sci. **335** (2005) 3–14
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC. (2004) 91–100
6. McCreight, E.: A space-economical suffix tree construction algorithm. J. ACM **23** (1976) 262–272
7. Apostolico, A., Szpankowski, W.: Self-alignments in words and their applications. J. Algorithms **13** (1992) 446–467
8. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA (1997)

Phrase-Based Pattern Matching in Compressed Text

J. Shane Culpepper and Alistair Moffat

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Victoria 3010, Australia

Abstract. Byte codes are a practical alternative to the traditional bit-oriented compression approaches when large alphabets are being used, and trade away a small amount of compression effectiveness for a relatively large gain in decoding efficiency. Byte codes also have the advantage of being searchable using standard string matching techniques. Here we describe methods for searching in byte-coded compressed text and investigate the impact of large alphabets on traditional string matching techniques. We also describe techniques for phrase-based searching in a restricted type of byte code, and present experimental results that compare our adapted methods with previous approaches.

1 Introduction

The compressed pattern matching problem is defined as: given a *pattern* P, a *text* T, and a corresponding *compressed text* Z generated by some compression algorithm, find all occurrences of P in T, that is, determine the set $\{|x| \mid T = xPy\}$, using P and Z.

The naive approach is to decompress the text before performing the pattern matching step, and fifteen years ago, this would probably have been the fastest mechanism. But ongoing growth in CPU power compared to I/O seek times in secondary storage devices has created a hardware speed gap, which allows increasingly complex algorithms to be utilised within the time that might otherwise be spent on I/O costs. It is, however, still necessary to balance efficiency (how quickly the compressed operation can be performed) and effectiveness (how good the compression is), and to take into account practical effects such as caching performance. In this framework, word-based modelling methods, combined with byte-aligned codes, offer several benefits [de Moura et al., 2000]. In particular, the use of byte codes allows use of available exact pattern matching algorithms, with only minimal modification required. The emphasis in previous research has been on variants of the Boyer-Moore approach, particularly the Horspool modification, see, for example, Fariña [2005]. While the BMH algorithm is clearly efficient on character-based alphabets in uncompressed text, it is unclear how it performs on the extended alphabets that arise from word-based compression models.

The traditional pattern matching problem has been studied for more than thirty years, and a broad range of efficient solutions have been proposed. All of the practical approaches use one of three searching techniques, and the notion of a *search window*, that positions the pattern relative to the text. The general techniques of interest include prefix-based searching, suffix-based searching, and factor-based searching. Several empirical studies of pattern searching strategies have also been conducted, and the reader is referred to, for example, the work of Navarro and Raffinot [2002], who consider the

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 337–345, 2006.

impact of both varying pattern sizes and also varying alphabet size, and draw much of the previous work together.

However, relatively little is known about the performance impact of removing redundancy from the search text. This paper examines that question, and also evaluates the impact of large alphabets on uncompressed and compressed search times. We consider prefix-based and factor-based searching approaches as well as the favoured suffix-based approaches. We also examine the restricted-prefix byte codes introduced by Culpepper and Moffat [2005], and show that they too can be searched quickly using a modified Boyer-Moore-Horspool mechanism. Indeed, compression accelerates pattern matching so much that byte-coded sequences can be searched faster *after* compression than they can in their raw, uncompressed, form.

2 Byte-Aligned Compression

One of the first practical compressed pattern matching approaches was proposed by Manber [1997]. Manber's simple byte-pair encoding is efficient, but does not give competitive compression effectiveness. However, the idea of using bytes instead of bits was an important first step in creating algorithms that are both effective and efficient.

Simple byte coding techniques have also been used to compress sequences of integers in information retrieval systems, because they provide fast decoding compared to more principled bit-based codes. As an example application, consider the following text fragment taken from the popular children's book "Fox in Socks" [Seuss, 1965]:

Bim comes.\n Ben comes.\n Bim brings Ben broom.\n Ben brings Bim broom.\n

Instead of using a character-based approach to compression, de Moura et al. [2000] built on previous word-based approaches, and described what they called the *spaceless words model*. A spaceless words parser assumes that the text to be represented is a sequence of words followed by non-words, but with the added constraint that if any non-word is a single space, the space can be discarded by the encoder, and re-introduced later by the decoder. Words and non-words are assigned ordinal symbol identifiers as they appear, so that the sequence of words is transformed into a sequence of integer indices into a dictionary of strings. The resulting integer sequence can be represented by any coding method, including byte-aligned coding approaches. In the example, the text segment from "Fox in Socks" is transformed into the integer sequence:

$$1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 4, 5, 6, 1, 7, 4\,,$$

where the "missing" symbol number 2 represents a single space character, and is not needed anywhere in this short message. Table 1 shows the sequential codewords assigned to this text fragment and the corresponding frequencies, and radix-4 codeword assignments for a range of byte-aligned codes.

The basic byte coding method (bc) uses codes that are fully static and easy to construct. It represents input integers using a radix-256 code in which values greater than 127 are *continuers* and are always followed by another byte, while values less than 128 are *stoppers*. The codewords generated are prefix-free, and it is easy to identify codeword boundaries directly in the compressed output, since the last byte of each codeword is less than 128. Note, however, that the code is static, and that actual frequency of each

Table 1. Symbol assignments and corresponding radix-4 codewords generated using a spaceless words model on a text fragment from "Fox in Socks". In the column `bc` the codewords are assigned based on ordinal symbol ordering; all other columns take the symbol frequency into account and bypass symbol 2, which does not appear in the message.

Word	Sym.	Freq.	`bc`	`phc`	`thc`	`dbc`	`scbc`	`rpbc`
.\n	4	4	10 01	00	00 10	00	00	00
Bim	1	3	00	01	00 11	01	01	01
Ben	5	3	11 00	10	01 10 10	10 00	10	10
comes	3	2	10 00	11 00	01 10 11	10 01	11 00	11 00
brings	6	2	11 01	11 01	01 11 10	11 00	11 01	11 01
broom	7	2	10 10 00	11 10	01 11 11	11 01	11 10	11 10
(*space*)	2	0	01	—	—	—	—	—

symbol is ignored. The "`bc`" column of Table 1 shows the codewords assigned when the set of symbol identifiers are taken at face value, and a radix-4 code computed (rather than the more usual radix-256 one). In a radix-4 version of `bc`, the "byte" values 00 and 01 are stoppers and the values 10 and 11 denote continuers. Note that symbol 2, which represents a single space, is assigned a code even though it does not appear in transformed source message, and that the most frequent symbol is not necessarily assigned the shortest codeword.

Another option is to calculate a radix-256 Huffman code, denoted `phc` (for plain Huffman code) in Table 1. Now an optimal code is computed for the set of symbol frequencies, and the source message represented accordingly. However, while `phc` provides maximal flexibility in assignment of codewords, it is impossible to search directly in the compressed text because one codeword can be a suffix of another codeword. Consider the codewords assigned by `phc` for the words *Bim* and *brings* in Table 1. The codeword 01 assigned to *Bim* is a suffix of the codeword 11 01 assigned to *brings*, and a search for *Bim* will result in a match against the second part of *brings*. In the example code the ambiguity could be resolved by looking at the preceding "byte" to see if it contains 11, but in a larger code, direct searching is impossible, since codeword boundaries are not identifiable.

To reintroduce searchability, de Moura et al. [2000] described *tagged Huffman codes* (`thc`), as a variation of the arrangement used in `bc`. Tagged Huffman codes are radix-128 Huffman codes which use 7 bits in each byte to store the Huffman code and 1 bit to signal the beginning of a codeword. With the extra tag bit inserted, `thc` codes are *suffix free* and allow any string matching algorithm such as `shift-or` or `horspool` to be used directly on the compressed text. The suffix-free property ensures that no false matches occur. Note that the cost of `thc` is exaggerated in Table 1 since in two-bit nibblets, only one actual data bit can be stored. Experimentally, searching in `thc` sequences is fast [de Moura et al., 2000], and searches are two to eight times faster than if the cost of decompression is added to the cost of uncompressed searching.

Brisaboa et al. [2003b] then noted that a static byte code could also be used, and in a system they call *end-tagged dense codes* (`dbc`), applied the same `bc` coding mechanism, but with the alphabet permuted into a new ordering dictated by decreasing occurrence frequency. A *prelude* describing the permutation is then necessary, to ensure that the decoder knows which source symbol should be assigned which codeword. In

general, the cost of the permutation is recovered through the use of shorter codewords for more frequent symbols, and overall compression is improved. The prelude proposed by Brisaboa et al. [2003b] is a rank-based mapping. For example, in Table 1, the symbol 5 is the 3rd most frequent and is assigned the codeword 10 00. Note that symbol 2, which does not appear at all in the example message, is no longer allocated a codeword – this is the "dense" part of the name. A drawback of the use of a prelude is that decoding is slower than the direct use of `bc`, because each decoded symbol must now be de-permuted via a large array, and cache-miss issues intrude [Culpepper and Moffat, 2005].

Brisaboa et al. [2003a] further realised that partitioning values other than 128 are possible, and that the sets of stoppers and continuers can be of different sizes – that better compression can be achieved by calculating an optimal partition based on the probability distribution of the input symbols. Brisaboa et al. [2003a] call this method (S, C)-dense coding (`scbc`). The only constraint is that the number of stoppers plus the number of continuers must satisfy $S + C = R$, where, as before, R is the radix of the coding system. For example, if $R = 4$ (as is used in the examples shown in Table 1) there are three possible (S, C)-dense arrangements: $(1, 3)$, $(2, 2)$, and $(3, 1)$. Note that the $(2, 2)$-arrangement corresponds to `dbc`. Table 1 shows the $(3, 1)$-arrangement, the best choice for the example text.

The most recent byte code variant provides a more flexible compromise between `phc` and the `scbc` coding approach [Culpepper and Moffat, 2005]. This method, called *restricted prefix byte coding* (`rpbc`), uses the first byte of each codeword to completely describe its length. Additional bytes can then use all of the remaining codespace. This allows compression gains, because different probability distributions can be more closely approximated by codes. Culpepper and Moffat showed that optimal codes can be calculated using a simple brute force method; and that additional compression gains are possible if care is taken when constructing the prelude. In Table 1, the optimal `rpbc` code turns out to be $(1, 1, 1, 2)$-arrangement, where the set of four values describe the codeword lengths associated with each of the four possible first "bytes".

The compression gain of `rpbc` does not come without cost. It is harder to track codeword boundaries in the compressed text, and backwards decoding – starting at a given codeword, and moving backwards in the byte stream to identify preceding codewords – is not possible. These new constraints make searching directly in the compressed text more challenging, particularly when using suffix-based searching algorithms.

3 Searching in Byte-Aligned Compressed Text

The ability to apply any searching algorithm with minimal modification is one of the key strengths of the byte-aligned compression systems. For example, the only alteration necessary to search directly in a stopper-continuer byte code is to add a false match filter that tests the byte immediately prior to a proposed match location. If that prior byte is a continuer then this proposed location is a false match, since it is not aligned on a codeword boundary in the byte stream. If it is a stopper, then the proposed match can be accepted as a valid appearance of the compressed codeword sequence. On the other hand, the `rpbc` method requires a different approach to false matches, because the codeword set is assigned exhaustively rather than partially, and it no longer suffices to look at the prior byte.

Algorithm 1. Brute force searching in `rpbc`. Function *create_tables* appears in Culpepper and Moffat [2005].

input: an `rpbc`-compressed array txt of compressed length $txtlen$ bytes, a compressed pattern pat of length $patlen$ bytes when compressed, and `rpbc` control parameters v_1, v_2, v_3, and v_4, with $v_1 + v_2 + v_3 + v_4 \leq R$, where R is the radix (typically 256).

1: set $t \leftarrow 0$ and $p \leftarrow 0$ and $occurrences \leftarrow \{\}$
2: $create\_tables(v_1, v_2, v_3, v_4, R)$
3: **while** $t \leq txtlen - patlen$ **do**
4: **while** $p < patlen$ **and** $pat[p] = txt[t + p]$ **do**
5: set $p \leftarrow p + 1$
6: **if** $p = patlen$ **then**
7: set $occurrences \leftarrow occurrences \cup \{t\}$
8: set $t \leftarrow t + \mathit{suffix}[txt[t]] + 1$ and $p \leftarrow 0$

output: the set of occurrences at which pat appears in txt, presented as a set of byte offsets in the compressed text txt.

Algorithm 2. Jump-based searching in `rpbc`

input: an array txt of $txtlen$ bytes representing `rpbc`-compressed symbols being searched, with a current pattern alignment that currently associates the first byte of the `rpbc`-compressed pattern with $txt[t]$; and an integer b that represents the number of bytes by which the pattern needs to be shifted.

1: **while** $b > 0$ **do**
2: set $s \leftarrow \mathit{suffix}[txt[t]]$
3: set $t \leftarrow t + s + 1$
4: set $b \leftarrow b - s - 1$

output: pointer t indicates a new offset in txt that is again `rpbc`-symbol aligned.

The simplest way of making `rpbc`-coded sequences searchable is to ensure that false matches can never occur, by only testing valid pattern-to-text alignments. To do this, the pattern shifting step of the search process must ensure that codeword boundaries are identified and respected. Of itself this is not an onerous requirement, since in the `rpbc` code the first byte of each codeword can be used to index a table that unambiguously records how many more bytes there are in that codeword. On the other hand, the extra table lookup is required for each source symbol that is skipped, and potentially disrupts the tight searching loops that are the hallmark of efficient pattern matching algorithms.

To see the necessary modification, Algorithm 1 shows how a brute-force pattern searching mechanism is modified to maintain codeword alignments. Step 8 is the critical one; normally it would shift the text pointer t by one, in order to accommodate the next byte alignment. But, because source symbols typically span multiple bytes, the increment to t is augmented by $\mathit{suffix}[txt[t]]$, the number of trailing bytes in the codeword that commences at $txt[t]$.

The symbol-stepping approach is not possible with stopper-continuer byte codes since there is no way to compute the codeword length without examining each byte of the codeword. That is, fewer comparisons are necessary on average in the `rpbc`-brute force approach than in (say) a `scbc`-brute force approach; and false matches are impossible since a shift never places the byte-level alignment between codeword boundaries.

A similar technique can be employed in any searching approach that employs longer shifts, such as the `horspool` algorithm. For example, suppose that a pattern alignment shift of b bytes is indicated by a state-based searching process that is operating at the byte level, a shift that would normally be effected by an assignment of the form $t \leftarrow t + b$. Algorithm 2 shows how the assignment is replaced by a loop that steps at least that many bytes forward, while retaining codeword alignment. This modification can be applied to any jump-based pattern matching algorithm, including the `kmp` and `horspool` techniques, and when shift values are returned from the processing tables which fall in the middle of a codeword, the next codeword boundary is found via a longer shift. However, additional lookups of text prefix bytes are needed to find codeword boundaries, possibly affecting overall performance.

The other key issue with the `rpbc` codes is that, in the form described by Culpepper and Moffat [2005], they cannot be decoded backwards. Reverse decoding is useful when, for example, a small snippet is required, to show a context surrounding the location of an identified match in the compressed text. One possibility – viable because the first-byte of every codeword is touched during the loop shown in Algorithm 2 – is to maintain a stack or sliding window of codeword starting points. The window would need to be as long as the maximum extent of any backwards decoding.

A more elegant solution is also possible, by separating the prefix bytes and the suffix bytes into separate compressed sequences. This approach, which we denote `rpbc_pa`, offers additional pattern matching alternatives. For example, using the code shown in Table 1, the integer sequence

$$1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 4, 5, 6, 1, 7, 4,$$

can be represented as a set of first "bytes"

$$01, 11, 00, 10, 11, 00, 01, 11, 10, 11, 00, 10, 11, 01, 11, 00,$$

and a corresponding set of suffix bytes,

$$, 00, \quad , \quad , 00, \quad , \quad , 01, \quad , 10, \quad , \quad , 01, \quad , 10, \quad ,$$

where the commas show which of the first bytes each suffix byte is associated with.

Because all the first bytes are extracted out into a single sequence, they can be accessed either backwards or forwards. And the first bytes indicate the length of each codeword. That is, if the current location is known in both the sequence of first bytes and also in the sequence of suffix bytes, backwards decoding is now possible.

The searching process must change, and is carried out in two parts. First, the sequence of first bytes is searched, looking for matches against the first bytes of the codewords that make up the pattern. As the sequence of first bytes is processed, the cumulative sum of $\mathit{suffix}[\mathit{txt}[t]]$ is noted for each location t at which there is a first-byte match against the pattern. Once a set of candidate locations has been identified, the suffix bytes at those locations are checked against the suffix bytes of the pattern's codewords. Sentinels, or partial cumulative sums, can be used at predetermined locations in the prefix array to remove the requirement of inspecting each prefix byte, but at the cost of compression effectiveness. This `rpbc_pa` (prefix array) version of `rpbc` is one of the methods evaluated in the next section.

4 Experimental Results

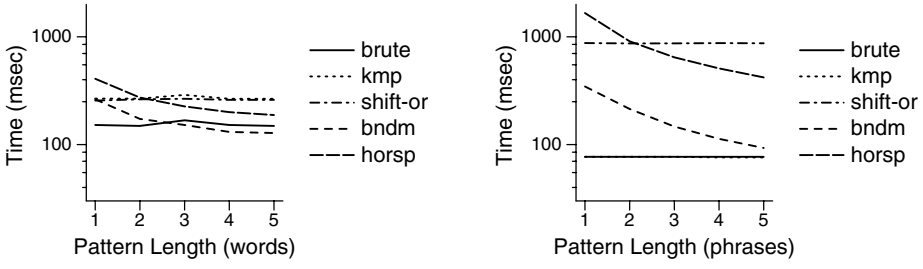
To evaluate the speed at which the various byte codes can be searched, we built two files of symbols from a 267 MB segment of SGML-tagged newspaper text, drawn from the *WSJ* component of the TREC data (see `trec.nist.gov`). The first one, `wsj267.wrd`, is the sequence of integers generated by the spaceless word model that was described earlier. The second file, `wsj267.repair`, is a sequence of integers representing phrases generated via an off-line, word-pair based encoding method called RE-PAIR [Larsson and Moffat, 2000]. Each symbol number represents a repeated phrase identified in the original word sequence, and because of the way the file is constructed, no pair of symbol numbers repeats. As well as integer-on-integer searching and character-on-character searching, five byte coding algorithms were investigated: `bc`, `dbc`, `scbc`, `rpbc`, and `rpbc_pa`. The `bc` and `dbc` methods were uniformly a little slower then `scbc`, and are not shown in the graphs below.

The average length of queries in web search systems is around 2.4 words per query [Spink et al., 2001]. To mirror this type of searching, took patterns of length 1 to 5 symbols, representing (in the case of `wsj267.wrd`) sequences of 1 to 5 words, or (in the case of `wsj267.repair`), 1 to 5 phrases. One hundred queries of each length were generated from the uncompressed integer sequences in the source files, by generating a random offset into it, and recording the sequence of symbols at that point. This process ensured that each pattern appeared at least once.

The integer pattern so generated can then be processed in different ways. For example, it can be used to measure the speed of an integer-on-integer search process; or converted back to the underlying character string and used in a character-on-character manner; or converted into codewords using any of the byte coding schemes, and then applied in a compressed codeword-on-codeword approach. For example, the three-symbol sequence $910, 2685, 153$ represents the original sequence "*offer may be*". The five different byte coding methods result in different corresponding patterns, with lengths varying from 4 bytes to 6 bytes.

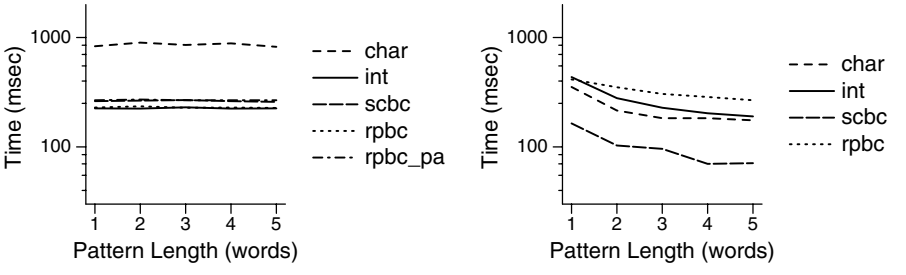
The first experiment was designed to evaluate the cost of integer-on-integer searching techniques. Figure 1 shows the measured performance of several different pattern matching techniques, without any compression having been applied. Algorithms which use preprocessed lookup tables proportional to the size of the alphabet tend to perform poorly when pattern lengths are short. Accesses to the large lookup table result in cache misses, which offset any gains achieved by the improved shifts. This effect continues until the search patterns become moderately long. In fact, brute force outperforms all of the more principled algorithms for patterns of three words or less, irrespective of the input file's probability distribution.

Figure 2 shows the speed at which the same patterns can be searched in the compressed domain, using two different pattern search algorithms, and a range of different byte-aligned coding methods. Both graphs in this figure relate to the spaceless words file `wsj267.wrd`; with the additional `char` method representing character-on-character searching in the uncompressed original form of the source file; and with the `int` method representing integer-by-integer searching in the uncompressed sequence of integers. When coupled with the brute-force searching approach (Figure 2a), `rpbc` performs



(a) Results for `wsj267.wrd` (b) Results for `wsj267.repair`

Fig. 1. Baseline searching times for uncompressed, integer-on-integer pattern matching, using a 2.8 Ghz Intel Xeon with 2 GB of RAM. The methods are brute force; the Knuth-Morris-Pratt method; Shift-Or searching; Backward Nondeterministic DAWG Matching; and the Horspool variant of the Boyer-Moore method. All of these approaches are described by Navarro and Raffinot [2002].



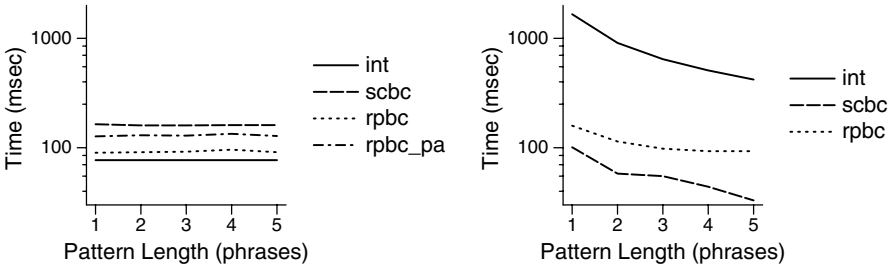
(a) Brute force search results (b) Horspool search results

Fig. 2. Searching `wsj267.wrd` using two different search techniques, and a range of uncompressed and compressed representations of text and patterns, using a 2.8 Ghz Intel Xeon with 2 GB of RAM

faster than any of the other byte code methods, and at the same speed as searching in the uncompressed integer file. With decompression costs (assuming that the data is stored in compressed form) included, the byte coding methods perform considerably better than the "decompress then search" baselines reflected in the `int` and `char` lines.

Figure 2b shows that the stopper-continuer byte code `scbc` performs better when coupled with the `horspool` searching method than when coupled with the brute force method. The `rpbc` variant has the same speed as in the brute force mode, and is clearly hampered by the additional operations involved in maintaining codeword boundaries.

Figure 3 shows the same experiment, but applied to file `wsj267.repair`. Now, when the symbol distribution is essentially flat and the alphabet size is large and dense, the integer-based `horspool` variant performs very poorly. Once again, the `rpbc` algorithm gives the same performance in the `horspool` environment as it does in the brute force one.



(a) Brute force search results

(b) Horspool search results

Fig. 3. Searching `wsj267.repair` using two different search techniques, and a range of uncompressed and compressed representations of text and patterns, using a 2.8 Ghz Intel Xeon with 2 GB of RAM.

Acknowledgement. The second author was funded by the Australian Research Council, and by the ARC Center for Perceptive and Intelligent Machines in Complex Environments. National ICT Australia (NICTA) is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In M. A. Nascimento, editor, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, volume 2857 of *LNCS*, pages 122–136, October 2003a.

N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In F. Sebastiani, editor, *Proceedings of the 25th European Conference on Information Retrieval Research*, volume 2633 of *LNCS*, pages 468–481, April 2003b.

J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In M. Consens and G. Navarro, editors, *Proceedings of the 12th International Symposium on String Processing and Information Retrieval*, volume 3772 of *LNCS*, pages 1–12, November 2005.

E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

A. Fariña. *New compression codes for text databases*. PhD thesis, Universidade de Coruña, April 2005.

N. J. Larsson and A. Moffat. Offline dictionary-based compression. *Proceedings of the IEEE*, 88 (11):1722–1732, November 2000.

U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems*, 5(2):124–136, April 1997.

G. Navarro and M. Raffinot. *Flexible pattern matching in strings*. Cambridge University Press, Cambridge, United Kingdom, first edition, 2002.

Dr. Seuss. *Fox in socks*. Random House, first edition, 1965. Written by T. Geisel.

A. Spink, D. Wolfram, B. J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science*, 52(3):226–234, 2001.

Discovering Context-Topic Rules in Search Engine Logs*

Carlos A. Hurtado[1] and Mark Levene[2]

[1] Universidad de Chile
churtado@dcc.uchile.cl
[2] Birkbeck, University of London
mlevene@dcs.bbk.ac.uk

Abstract. In this paper, we present a class of rules, called *context-topic rules*, for discovering associations between topics and contexts, where a context is defined as a set of features that can be extracted from the log file of a Web search engine. We introduce a notion of rule interestingness that measures the level of the interest of the topic within a context, and provide an algorithm to compute concise representations of interesting context-topic rules. Finally, we present the results of applying the methodology proposed to a large data log of a search engine.

1 Introduction

Search engines now receive hundreds of millions queries a day, so by inspecting their log files they are able to get an accurate picture of what users are looking for at any given moment in time. In this paper, the problem we are looking at is, in general terms: *Given a query topic, what are the "interesting" contexts for that topic?*, where a context is defined as a set of features that can be extracted from a log file of a search engine. Typical features present in search engine logs include the date and time of the query, the IP address from which the query was made, or more specific cookie information when available. The temporal contexts of day of the week (*dayOfWeek*) and time of day (*hourOfDay*) are particularly interesting, since the popularity of topics drastically changes according to these contexts. For example, broadly speaking, "chat" is more popular on the weekend during the evening, while "automoblie" is also popular during the week at various times of the day. Here we will concentrate our analysis on these temporal contexts, although our formalism is by no means limited to temporal queries and will be presented in a general setting of contexts.

Our approach to tackling the problem of context is by adapting the notion of an association rule [HGN00] to what we call a context-topic rule (or simple a *c-t rule*). In our approach, the antecedent of our rules is a set of features, while the consequent is a topic. For example, the c-t rule

$$\{dayOfWeek : \{Saturday\}, hourOfDay : \{11, 20\}\} \rightarrow automobile \qquad (1)$$

* Carlos A. Hurtado was supported by Millennium Nucleus, Center for Web Research (P04-067-F), Mideplan, Chile.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 346–353, 2006.

states that the topic "automobile" is interesting on Saturday at times 11AM and 8PM. Note that we also allow the features themselves to be sets of values; when all the features are singletons the c-t rule is called *atomic*.

Informally, for a c-t rule such as (1) to be interesting we need to ascertain that the ratio of the conditional probability, $P(t \mid c)$, of the topic (i.e. the consequent of the rule) given the context (i.e. the antecedent of the rule) to the probability, $P(t)$, of its topic in any context, is higher than a predefined threshold. Moreover, to allow for statistical variation we extend the ratio to a $100(1-\alpha)\%$ confidence interval in a standard fashion; the formal definition is given in Section 2. There remains the problem of forming maximal rules such as (1) from a set of atomic rules, in order to provide us with a compact representation of a set of interesting atomic c-t rules. We also address this problem in Section 2.

The contributions of the paper are now described. We formalise the notion of context-topic rules for finding associations between contexts and topic in a search engine log, and model a context as a region in a feature space extracted from the log. We then introduce a notion of rule interestingness as a measure of the level of interest in the topic inside a context compared with the level of interest in the topic throughout the entire log; we call this measure the *lift* of the c-t rule. We also provide an algorithm to compute c-t rules, which consists of two stages: (i) the computation of a set of atomic c-t rules, and (ii) their compression into a set of maximal c-t rules that cover the atomic rule set. Finally, we applied the proposed methodology to a large data log for a search engine.

2 Context-Topic Rules

In this section, we formalise c-t rules and present an algorithm to compute maximal rules.

2.1 Formalising Context-Topic Rules

We consider a fixed set of features $V = \{X_1, \ldots, X_n\}$, over nominal (i.e., categorical) or integer domains. We denote by $\texttt{dom}(X_i)$ the domain of the feature X_i. We assume a distinguished nominal feature called `Topic` whose domain contains the topics of interest. In the data analysis carried out here, we are assuming that the topic of a query is already known, and can be deduced from the query terms, for example, with the aid of a classifier or even manually.

A *log file* $\mathcal{L}$ is a set of events. An *event* is a vector $(x_1, \ldots, x_n, t)$ where each $x_i \in \texttt{dom}(X_i)$. An event represents a request for information on a topic t with respect to a vector, $(x_1, \ldots, x_n)$, in the feature space. In particular, a log file is a sample of the behaviour of the entire user population over a certain time period, so we make the usual distinction between a *true* probability P and an estimated probability $\hat{P}$. As an example, $\hat{P}(\texttt{Topic} = \textit{cars})$ is the fraction of events in $\mathcal{L}$ with $\texttt{Topic} = \textit{car}$ from the total number of events in $\mathcal{L}$, while $P(\texttt{Topic} = \textit{cars})$ is the corresponding true probability.

A *context-topic rule* (c-t rule) is an expression of the form $\{X_1 : S_1, X_2 : S_2, \ldots, X_n : S_n\} \rightarrow t$, where $X_1, \ldots, X_n$ are the features in V, each

$S_i \subseteq \texttt{dom}(X_i)$, and $t \in \texttt{dom}(\texttt{Topic})$. Whenever every S_i is a singleton set, we say that the rule is *atomic*, and denote $S_i = \{e_i\}$ simply by e_i.

As an example, the following c-t rule associates the topic "chat" to the context "night time during the weekend":

$$\{dayOfWeek : \{Saturday, Sunday\}, hourOfDay : \{21, 22, 23, 24, 1, 2\}\} \rightarrow chat.$$

A c-t rule $\{X_1 : S_1, \ldots, X_n : S_n\} \rightarrow t$, represents the following set of atomic c-t rules: $\{\{X_1 : e_1, \ldots, X_n : e_n\} \rightarrow t \mid e_i \in S_i\}$, which will be denoted by $\texttt{atomRules}(\{X_1 : S_1, \ldots, X_n : S_n\} \rightarrow t)$. Given two c-t rules r_1, r_2, we say that $r_1 \sqsubseteq r_2$ (r_2 contains r_1) if and only if $\texttt{atomRules}(r_1) \subseteq \texttt{atomRules}(r_2)$. Strict containment, denoted $r_1 \sqsubset r_2$, requires that $\texttt{atomRules}(r_1) \subset \texttt{atomRules}(r_2)$.

We next formalise the conditions under which a c-t rule is "interesting" in a similar fashion to other notions of interestingness in data mining such as rules or sequential patterns; see for example, [HMS01]). A c-t rule will be defined to be interesting if all the atomic rules it represents are interesting. For an atomic rule $c \rightarrow t$, the required condition is that the true probability of a request for the topic t with respect to the context c, denoted by $P(t \mid c)$ is "much larger" than the true probability of a request for the topic t with respect to the entire population of events, i.e. $P(t)$. Thus, we need a method to estimate a confidence interval of the ratio $\frac{P(t \mid c)}{P(t)}$. For this purpose, we use the lower limit of a $100(1-\alpha)\%$ Taylor series confidence interval for the ratio of two proportions [FLP03], and define the *lift* of a rule in terms of this lower limit.

Let $c \rightarrow t$ be an atomic c-t rule and $\mathcal{L}$ be a log file. The *lift* of $c \rightarrow t$ in $\mathcal{L}$ at confidence level $\beta = 100(1-\alpha)\%$, denoted $\texttt{lift}_{\mathcal{L}}(c \rightarrow t, \beta)$, is defined as follows:

$$\texttt{lift}_{\mathcal{L}}(c \rightarrow t, \beta) = \frac{\hat{P}(t \mid c)}{\hat{P}(t)} \exp\left(-Z_{1-\frac{\alpha}{2}} \sqrt{\frac{(1-\hat{P}(t \mid c))}{n_c \hat{P}(t \mid c)} + \frac{(1-\hat{P}(t))}{n\hat{P}(t)}}\right),$$

where n_c is the number of events in $\mathcal{L}$ where c holds, and n is the number of events in $\mathcal{L}$.

We now define interestingness of a c-t rule given β as above, and ρ, which is the lift threshold parameter. Let r be a c-t rule, and $\mathcal{L}$ be a log file. We say that r is (ρ, β)-*interesting* in $\mathcal{L}$ if and only if the following holds: (a) if r is atomic then $\texttt{lift}_{\mathcal{L}}(r, \beta) \geq \rho$; and (b) if r is non-atomic then all the rules in $\texttt{atomRules}(r)$ are (ρ, β)-interesting. A c-t rule r is defined to be (ρ, β)-*maximal* if it is (ρ, β)-interesting and there is no (ρ, β)-interesting rule r' such that $r \sqsubset r'$.

2.2 Algorithm for Computing c-t Rules

Given a log file $\mathcal{L}$, a minimum lift threshold ρ, and a confidence level β, our goal is to provide an algorithm to find all (ρ, β)-maximal c-t rules in $\mathcal{L}$. Intuitively, we are searching for the maximal contexts inside which the proportion of requests for a topic is ρ times larger that on average. Our algorithm utilises the following two steps:

1. Finding the set $\mathcal{A}$ of all interesting atomic rules. In a single pass of the log file we set up a hash table and count the occurrences of tuples of the form (c_i, t_j), $(c_i, *)$, and $(*, t_j)$, for each context c_i that appears in the table, and each topic $t_j \in \texttt{dom(Topic)}$. Then, we enumerate all rules $c_i \rightarrow t_j$, extracted from the hash table counters, compute the lift of each rule, and check if it is greater or equal than ρ. At the end of this step we obtain a table $\mathcal{A}(X_1, \ldots, X_2, \texttt{Topic})$ that contains all the atomic interesting rules represented in the form of tuples. This step requires two passes of the table containing the log events, and takes time linear in the size of this table.
2. Finding maximal rules r such that $\texttt{atomRules}(r)$ are the tuples in $\mathcal{A}$. In this step we iteratively compress the rule tuples in the table $\mathcal{A}$ by applying a modified version of a nesting operator.

In the remainder of the section we focus on the problem of obtaining maximal rules. As previously explained, we focus here on the discovery of c-t rules with the temporal features *dayOfWeek* and *hourOfDay*. For these two features, the problem of computing the maximal rules from a given table $\mathcal{A}$ containing atomic rules is equivalent to enumerating maximal cliques in a bipartite graph (for k features the problem reduces to enumerating cliques in k-partite graphs). A bipartite graph is a graph with two distinct vertex sets, and a maximal clique is a maximal complete bipartite subgraph of it. The problem is also equivalent to enumerating closed frequent sets in association rule mining and to constructing a Galois lattice in formal concept analysis [ZO98]. As an example, in Figure 1 table $\mathcal{A}$ is also depicted as a bipartite graph (last figure). Table C below contains the maximal rules, which can viewed as maximal cliques in the bipartite graph.

For the sake of brevity, we only give an intuitive explanation of the algorithm. We assume table $\mathcal{A}$ has two attributes denoted X_1 and X_2 and that it allows tuples to record sets of values in an attribute. The maximal cliques are computed in two steps. In a first step, we group (nest) the tuples by the values of one attribute. This operation is similar to the SQL group-by computation, where the tuples are grouped by a single attribute, and can be implemented with a single pass of the table. As an example, after this step, table $\mathcal{A}$ in Figure 1 is transformed

X_1	X_2
1	c
1	b
2	c
1	a
2	a
4	c
3	a

Table $\mathcal{A}$

X_1	X_2
{1,2,4}	c
1	b
{1,2,3}	a

Table B

X_1	X_2
{1,2,4}	c
{1,2,3}	a
1	{a,b,c}
{1,2}	{a,c}

Table C

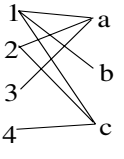


Table $\mathcal{A}$ viewed as a bipartite graph.

Fig. 1. Example showing the compression step of the c-t rule mining algorithm

into table B. In a second step, we compute all possible combination of tuples in table B such that they are maximal (a tuple is maximal if it is not subsumed by any other tuple in the table), obtaining the maximal rules in table C.

Since a bipartite graph can have an exponential number of bipartite cliques, even for two features, we may have an exponential number of maximal rules. However, when the graph is sparse we can obtain linear complexity in the size of the graph. Consider the maximal rule having context $\{X_1 : S_1, X_2 : S_2\}$, such that $\frac{|S_1|\times|S_2|}{|S_1|+|S_2|-1}$ is the maximum amongst all the maximal rules. This number, called the *arboricity* of the graph, measures the sparsity of the graph; we denote it by $r(\mathcal{A})$. From a result by Epstein [Epp94] it can be verified that the final table has $O(n2^{2r(\mathcal{A})})$ tuples, where $n = |\mathcal{A}|$; this expression is an upper bound for the number of maximal cliques in the bipartite graph. Clearly, $r(\mathcal{A})$ is constant, since $r(\mathcal{A}) \leq \frac{|\mathtt{dom}(X_i)|}{2}$, for any of the two features. Therefore our algorithm runs in $O(n)$ time, though the constant in this expression could be large. However, in practice the graph is sparse. In particular, in our experiments $r(\mathcal{A})$ did not exceed 3.

3 Experiments

We implemented the algorithm described in Section 2.2 in Java and used it to extract c-t rules from the logs of the search engine, TodoCl. This search engine covers the domain of Chile and some pages included in the `.net` high level domain that are related to ISP providers in Chile. Its index contains over 3 million web pages and currently, in mid 2006, has over 50,000 requests per day. The data we used come from the logs over a period of six months from July to December 2004. Over these six months the log registered a total of 245,170 query sessions (sessions from meta-search engines were deleted), which corresponds to 127,642 queries.

Top-5 atomic rules at conf. level 95%									
Rank	*dayOfWeek*	*hourOfDay*	`Topic`	lift at 95%	lift at 80%	lift at 60%	plain lift	$n_{c,t}$	n_c
1	*Monday*	8	*gamesOfChance*	2.86	4.99	7.11	14.26	16	187
2	*Sunday*	6	*adult*	2.57	3.06	3.41	4.20	20	46
3	*Monday*	9	*gamesOfChance*	1.38	2.66	4.02	9.00	19	349
4	*Saturday*	3	*adult*	1.34	1.71	1.99	2.69	21	76
5	*Wednesday*	5	*adult*	1.28	1.72	2.07	3.00	12	39
Atomic rules at conf. level 70% for $n_{c,t} > 50$ and $n_c - n_{c,t} > 50$									
Rank	*dayOfWeek*	*hourOfDay*	`Topic`	lift at 95%	lift at 80%	lift at 60%	plain lift	$n_{c,t}$	n_c
1	*Friday*	0	*adult*	1.15	1.37	1.54	1.92	59	299
8	*Monday*	20	*culture*	0.84	1.07	1.21	1.53	76	511
10	*Saturday*	11	*automobile*	0.82	1.07	1.17	1.52	52	314
16	*Wednesday*	12	*locations*	0.70	0.94	1.13	1.62	59	695

Fig. 2. (above) Top-5 most interesting atomic rules at confidence level 95%. (below) Best-ranked rule for each of the topics: *adult*, *culture*, *automobile*, and *locations*, at confidence level 70% and $n_{c,t} > 50$ and $n_c - n_{c,t} > 50$ (see Section 2) for the formalism.

We performed a manual classification task for the top-2000 most frequent queries into 37 topics plus a class "unclassified", which includes the queries which could not be assigned to any of the 37 topics. The top-2000 queries account for 58,681 query sessions in the log, that is a 23.9% of the total number of sessions. This set of 58,681 query sessions was used as the dataset over which we ran the experiments.

Measure	Confidence level (β)				
	60%	70%	80%	90%	95%
Mean (lift)	1.43	1.43	1.45	1.34	1.32
Standard dev. (lift)	0.76	0.72	0.68	0.55	0.50
Max (lift)	7.11	6.08	4.99	3.70	2.86
Num. atom. rules	87	63	41	30	18
Num. max. rules	55	32	23	20	11
Num. topics included	14	12	8	5	4

(A)

dayOfWeek	*hourOfDay*	`Topic`
Saturday	0	*chat*
Sunday	1	*chat*
Friday, *Saturday*	20	*goingOut*
Monday	8, 9	*gamesOfChance*
Sunday	22	*gamesOfChance*
Monday	11	*jobs*
Tuesday	9	*health*
Monday, *Tuesday*	20	*culture*
Sunday	12	*culture*
Tuesday	20, 21	*culture*
Wednesday	19	*culture*

(B)

Fig. 3. (A) Measures for the interesting atomic rules obtained at different confidence levels. (B) $(1, 70\%)$-maximal rules for six topics.

As mentioned before, we considered the two features *dayOfWeek* and *hourOfDay* to model contexts in c-t rules. Thus there are potentially $24 \times 7 = 189$ contexts and $24 \times 7 \times 37 = 6,993$ atomic c-t rules to discover. The dataset, however, registers events for only 168 contexts and 5320 atomic c-t rules, which are the ones that show activity in the log. We implemented an algorithm that scans the log table and counts the occurrences of contexts and occurrences of topics inside each context, as explained in Section 2.2. From these frequencies, the algorithm computes the lift of all possible atomic rules that arise at confidence levels $60\%, 70\%, 80\%, 90\%$, and 95%. The computation took a few seconds for each of the cases. Figure 2 (above) shows the five atomic rules with the largest lift, among the $(1, 95\%)$-interesting atomic rules obtained. The table shows lifts at confidence levels 95%, 80%, and 60%, along with its lift, i.e. the ratio $\frac{\hat{P}(t \mid c)}{\hat{P}(t)}$, as explained in Section 2. The last two columns depict the number of requests for the topic in the context, i.e. $n_{t,c}$, and the number of requests for the context, i.e. n_c. Since the normality assumption that underlies the definition of lift is more accurate for larger $n_{c,t}$ and $n_c - n_{c,t}$, in Figure 2 (below) we show $(1, 70\%)$-interesting atomic rules for which these values are greater than 50. The figure shows the best ranked rules for each of the topics *adult* (search for adult material), *culture* (queries related to cultural information), *automobile* (search for cars, car rentals, car sales, etc.), and *locations* (search for geographical areas, political divisions, cities, etc.).

We next studied the sets of atomic rules obtained at different confidence levels. Figure 3 (A) shows statistics for each of the sets of rules obtained. As

might be expected, the number of rules obtained decreases as the confidence is increased; this number goes from 87 to 18 rules in our results. In all cases the algorithm took only a few seconds to compute the rules. Figure 3 (B) shows the $(1, 70\%)$-maximal rules obtained for the topics *chat* (search for chat sites), *goingOut* (search for restaurants, theaters, pubs, etc.), *gamesOfChance* (lottery, keno, etc.), *jobs* (search for jobs), *health* (hospitals, doctors, medical services, etc.) and *culture*.

The experiments performed provide evidence for the utility of c-t rules to discover habits of users that search for information on the Web. Many of the rules found reveal clear dependencies between contexts and information needs. As an example, the rules for the topic *gamesOfChance*, Figure 3 (B), may be explained by the fact that the results of the most popular games of chance in Chile are announced every Sunday afternoon, and presumably people query the search engine on Sunday at 10pm or early on Monday morning in order to look at the results. Users requesting for information on automobiles on Saturday at 11am are probably doing research to find information on places where they can buy or look at cars.

4 Related Work

Context-topic rules are related to temporal association rules. The closest class of temporal association rules to c-t rules are the rules introduced by Li et al. [LNWJ03]. In their setting, a temporal association rule is an association rule that holds during specific time intervals. A time interval is represented using a calendar schema, which is a template that specifies a set of dates. The data mining problem studied is to find all the interesting rules that arise inside a fixed calendar schema. Our problem is different, since we focus on single topics rather than rules inside the calendar schemas (or context in our terminology), which yields a different notion of interestingness. Furthermore, our problem is to discover the calendar schemas themselves (along with compressed representations for them) inside which topics are interesting. The problem of finding the maximal c-t rules is related to the problem of obtaining a Minimum Description Length (MDL) encoding that captures the set of atomic rules. Pu and Mendelzon [PM05] studied this problems for several variations of languages that represent structured sets. For instance, if we consider ordered domains our problem is similar to the setting studied by Agrawal et al. [AGGR05], where the goal is to obtain a MDL encoding that covers a cluster within feature space. The problem of testing whether a set of maximal rules is a minimal cover is NP-complete (reduction from the minimal set cover problem), in other words, finding a MDL description for the atomic rules in our setting is computationally hard.

5 Concluding Remarks

In this paper we have introduced a method to capture the temporal contexts users search for with respect to a specified topic. We can see several potential

applications for context-topic rules. They can be used to support recommendation of web pages on a given topic according to context. This is also related to contextual advertising as the temporal context of a c-t rule may indicate the best time to advertise/recommend items related to the topic. In addition, for search engines that maintain a directory, the context of a topic could have an effect on the prominence it is given when presented to the user. Within a web site c-t rules can also be utilised in the context of web log data mining [BL00]. In this setting the topic could be obtained from a query to the local search engine, from a referral from a web search engine, or from the topic of a web page being browsed. The knowledge attained from c-t rules could provide assistance in deciding which queries a search engine should cache and when to cache them.

In the context of search engine query logs, requests are queries (related to topics) submitted to a search engine. However, our framework is much more general and could be relevant to other types of web interaction such as web page accesses or requests for topics in a web directory. Directions for future research include ranking criterion and visualisation of c-t rules, experiments over larger data sets and additional features such as IP and day of year, complexity and empirical evaluation and a more efficient implementation of algorithms.

Acknowledgments. We thank the Center for Web Research (www.cwr.cl) and Marcelo Mendoza for providing the query log database used in the experiments.

References

[AGGR05] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data. *Data Min. Knowl. Discov.*, 11(1):5–33, 2005.

[BL00] J. Borges and M. Levene. Data mining of user navigation patterns. In B. Masand and M. Spiliopoulou, editors, *Web Usage Analysis and User Profiling*, Lecture Notes in Artificial Intelligence (LNAI 1836), pages 92–111. Springer-Verlag, Berlin, 2000.

[Epp94] D. Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information Processing Letters*, pages 51:2007–211, 1994.

[FLP03] J. L. Fleiss, B. Levin, and M. C. Paik. *Statistical Methods for Rates and Proportions.* Wiley, 2003.

[HGN00] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining – A general survey and comparison. *SIGKDD Explorations*, 2:58–64, 2000.

[HMS01] David J. Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining.* The MIT Press, 2001.

[LNWJ03] Y. Li, P. Ning, X. Sean Wang, and S. Jajodia. Discovering calendar-based temporal association rules. *Data Knowl. Eng.*, 44(2):193–218, 2003.

[PM05] K. Pu and A. Mendelzon. Concise descriptions of subsets of structured sets. *ACM Trans. Database Syst.*, 30 (1), 2005.

[ZO98] Mohammed J. Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *Proceedings of 3rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, 1998.

Incremental Aggregation of Latent Semantics Using a Graph-Based Energy Model

Aditya Ramana Rachakonda and Srinath Srinivasa

IIIT-Bangalore, 26/C, Electronics City, Bangalore 560100, India
aditya.ramana@iiitb.ac.in, sri@iiitb.ac.in

Abstract. A graph-theoretic model for incrementally detecting latent associations among terms in a document corpus is presented. The algorithm is based on an energy model that quantifies similarity in context between pairs of terms. Latent associations that are established in turn contribute to the energy of their respective contexts. The proposed model avoids the polysemy problem where spurious associations across terms in different contexts are established due to the presence of one or more common polysemic terms. The algorithm works in an incremental fashion where energy values are adjusted after each document is added to the corpus. This has the advantage that computation is localized around the set of terms contained in the new document, thus making the algorithm run much faster than conventional matrix computations used for singular value decompositions.

1 Introduction

Latent semantic analysis (LSA) [3,8] is a popular mechanism for detecting semantic associations across terms by analyzing a document corpus. The underlying idea behind LSA is singular value decomposition (SVD) of the term-document (or more precisely, a term-context[1]) matrix. LSA detects similarity in context by mapping the high-dimensional term/document space to a lower dimensional *latent semantic* space, whose dimensionality corresponds to the rank of the term-document matrix. Dimensionality reduction results in associations being established across terms (and documents) based on their projections into the latent semantic space. LSA can detect associations arising due to various linguistic constructs like synonymy, co-occurrence and polysemy.

SVD computation however, suffers from a few shortcomings. Conventionally, SVD calculation is a compute-intensive process and can be performed only on a static data set. It does not scale up to extremely large and dynamic document collections like the web. In addition, LSA sometimes produces negative associations among terms, which cannot be adequately interpreted [6]. In large document collections, comprising of several polysemic terms, LSA can create spurious associations across terms from different contexts [1].

[1] We shall be using the terms "document" and "context" interchangeably. In our experiments, each paragraph from which terms were extracted, formed their context.

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 354–359, 2006.

Several enhancements have been proposed to the basic LSA model. These include the following. The HITS algorithm by Kleinberg [7] for computing hub and authority scores for web pages is shown to be equivalent to an SVD computation if it were run on a bipartite graph of terms and documents (cf. [2]). Gorell and Webb [5] propose an incremental model of the LSA algorithm based on generalized Hebbian learning, where SVD computations are incrementally performed whenever documents are added to the corpus. While this enables SVD to be computed incrementally, other problems like that of polysemy remain.

In this work, we develop an approach towards detecting latent associations, that is qualitatively different from LSA. The main idea here is to directly compute similarity in contexts between pairs of terms and add latent association weights based on this similarity measure. Similarity in context is measured by looking at the common terms occurring in their contexts and the connectivity among terms. This is a graph-based model, which can be incrementally updated, with localized computations and also does not suffer from the polysemy problem.

2 Model for the Term Graph

The central data structure in the energy model is a term-term graph. A term-document co-occurrence matrix is separately maintained in order to retrieve documents based on terms. But the energy model itself runs wholly on the term-term graph.

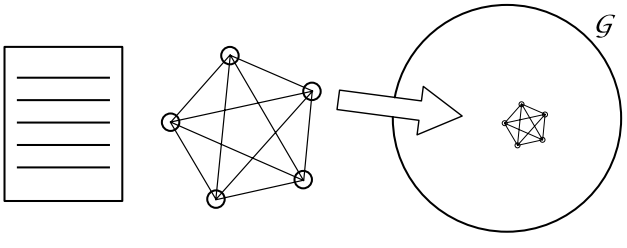


Fig. 1. Terms are extracted from a context, made into a clique and embedded into the global term-term graph $\mathcal{G}$

The term-term graph is an undirected graph $\mathcal{G} = (V, E)$, where V is set of all terms that have been found in the corpus till now and E is the set of all associations across terms that have either been found in documents or have been established by the energy model algorithm.

Edges are undirected and weighted. The weight of an edge indicates relatedness between terms which share the edge. If two terms do not share an edge then the algorithm considers the edge weight to be zero. Documents are incrementally added and the graph gets updated after each such addition.

When a new document or "bag of words" D is to be added to the corpus, it is first split into contexts[2] $D = \{C_1, C_2, \ldots C_n\}$ such that for any distinct i and

[2] A context is usually a paragraph.

j, $C_i \cap C_j = \phi$ and $\bigcup_i C_i = D$. Then, from each C_i, stopwords are eliminated[3]. The resulting terms are stemmed and terms which occur only once in the whole document are also eliminated [10].

From the resulting set of terms in each C_i, a *clique* is formed by connecting all terms in C_i to one another [4]. Once the clique C_i is formed, all edge weights in the clique are set to 1. Now C_i is embedded into the global term-term graph $\mathcal{G}$ (as shown in figure 1). For every edge of the form (u, v) found in C_i a corresponding edge is created in $\mathcal{G}$ or, if an edge already exists, its weight is incremented by 1.

The subgraph of $\mathcal{G}$ corresponding to C_i is now taken as the starting context for establishing latent associations based on the energy model. Once the energy model stops propagating, the next context from the document is added to the graph $\mathcal{G}$.

3 Energy Equations and Dynamics

Energy Equation. A context $\mathcal{C}$ in the term-term graph $\mathcal{G}$ is any subgraph comprising of terms and all associations between them that exist in $\mathcal{G}$. Energy (ξ) is the measure of relatedness defined on a set of words. The energy of a set is directly proportional to the mean (μ) of all edge-weights between every pair of words in the set. A high mean edge-weight indicates that the words in the set co-occur together a lot and hence belong to the same context. On the other hand, the energy of a set is inversely proportional to its variance (σ^2). A high variance of edge-weights implies that the set has some elements which do not quite belong to the context. So the energy of a set of words is,

$$\xi = \mu/(1 + \sigma^2) \tag{1}$$

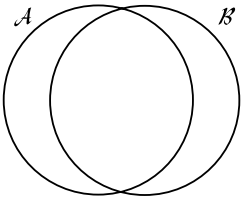


Fig. 2. Set $\mathcal{A}$ is the context in the term graph and Set $\mathcal{B}$ is the clique

Merging a Context into the Global term-term Graph. Figure 2 schematically shows two graphs $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{A}$ is some subgraph of the global term-term graph. $\mathcal{B}$ is the clique, which is being added to the term-term graph. When $\mathcal{B}$ is merged into $\mathcal{G}$, some edges in $\mathcal{B}$ only add weights to existing edges of $\mathcal{G}$, while some

[3] In this work, stopwords from the Swish-e project was used: http://swish-e.org/

[4] When there is no ambiguity, we shall use the same terminology C_i to refer to both the i^{th} bag of terms, as well as the i^{th} clique.

edges in $\mathcal{B}$ form new edges in $\mathcal{G}$. The subgraph $\mathcal{A}$ is chosen such that $\mathcal{A} - \mathcal{B}$ is the set of all terms in $\mathcal{G}$, which are related to $\mathcal{A} \cap \mathcal{B}$ but not quite related to $\mathcal{B} - \mathcal{A}$. We begin energy calculations with $\mathcal{A} \cap \mathcal{B}$ as the starting context. Terms in $\mathcal{A} - \mathcal{B}$ share the same context $\mathcal{A} \cap \mathcal{B}$ with terms from $\mathcal{B} - \mathcal{A}$. The energy of their common context is $\xi_{\mathcal{A}\cap\mathcal{B}}$. So we potentially add edges for all terms in $(\mathcal{A} - \mathcal{B}) \times (\mathcal{B} - \mathcal{A})$. The terms in $\mathcal{A} - \mathcal{B}$ and $\mathcal{B} - \mathcal{A}$ are called "pendant nodes" for the context $\mathcal{A} \cap \mathcal{B}$. They are terms that directly relate to the context. At any arbitrary propagation level, pendant nodes for establishing associations are calculated by notions called the "least set" and the "most set" explained below.

Least Set. Given a context $\mathcal{C}$, the least set ($\mathcal{L}_\mathcal{C}$), is the set of all terms, which have occured rarely in this context $\mathcal{C}$. To determine $\mathcal{L}_\mathcal{C}$, we compute the energy of $\mathcal{C}$ and note the energy change we get by removing each term of $\mathcal{C}$ individually. All those terms whose removal resulted in a rise in the energy level of $\mathcal{C}$ are added to $\mathcal{L}_\mathcal{C}$.

Most Set. Given a context $\mathcal{C}$, its most set ($\mathcal{M}_\mathcal{C}$), is the set of all terms which are not present in the context, but are nevertheless quite relavent to the context. Before computing $\mathcal{M}_\mathcal{C}$, terms in $\mathcal{C}$ is partitioned into two sets $\mathcal{R}_\mathcal{C}$ and $\mathcal{L}_\mathcal{C}$, where $\mathcal{R}_\mathcal{C} = \mathcal{C} - \mathcal{L}_\mathcal{C}$. Here, $\mathcal{L}_\mathcal{C}$ is the least set defined earlier. To determine $\mathcal{M}_\mathcal{C}$, we take set of all terms in $\mathcal{G}$ that directly connect to one or more terms in $\mathcal{R}_\mathcal{C}$, and add it to the context $\mathcal{R}_\mathcal{C}$. We then compute the energy of $\mathcal{R}_\mathcal{C}$ with the new additional term. All those terms which resulted in an increase in the energy of $\mathcal{R}_\mathcal{C}$ are added to $\mathcal{M}_\mathcal{C}$.

Adding Semantic Associations. For any context $\mathcal{C}$, once $\mathcal{L}_\mathcal{C}$ and $\mathcal{M}_\mathcal{C}$ are calculated, weighted edges are added for all nodes in $\mathcal{L}_\mathcal{C} \times \mathcal{M}_\mathcal{C}$. This is because, the words in $\mathcal{L}_\mathcal{C}$ are new to the context and the words in $\mathcal{M}_\mathcal{C}$ are already related to the context. So adding edges between them is analogous to mining new associations. If for any $(i, j) \in \mathcal{L}_\mathcal{C} \times \mathcal{M}_\mathcal{C}$, an edge already exists, then the new weight $\varepsilon_{i,j}$ that is calculated, is simply added to the existing edge weight. The edge-weights $\varepsilon_{i,j}$ between the terms $(i, j) \in \mathcal{L}_\mathcal{C} \times \mathcal{M}_\mathcal{C}$, are calculated based on the set of equations given below:

$$w = \mathcal{L}_i + \mathcal{M}_j \tag{2}$$

$$r = |\mathcal{R}_\mathcal{C}| \tag{3}$$

$$\varepsilon_{i,j} = \frac{\mu(1 - e^{-\frac{r}{\rho}})(1 - e^{-\frac{w}{\omega}})}{k} \tag{4}$$

In equation 2, $\mathcal{L}_i$ is the increase in energy by removing the term i from $\mathcal{C}$ and $\mathcal{M}_j$ is the increase in the energy by adding the term j to $\mathcal{R}_\mathcal{C}$. In equation 3, $|\mathcal{R}_\mathcal{C}|$ is the number of terms in $\mathcal{R}_\mathcal{C}$. In equation 4, μ is the mean of the edge-weights in $\mathcal{R}_\mathcal{C}$. k is the current iteration number or the propagation level. The terms ρ and ω in 4 are tweakable parameters that determine the sensitivity of the various parameters to $\varepsilon_{i,j}$.

Rationale for the Edge-weight Equation. The edge added can not be arbitrarily large, so it is limited by the mean (upper-bound). The edge-weight should be directly proportional to the size of the context. After a certain size that entity $(1 - e^{-\frac{r}{\rho}})$ saturates to one. To tweak this size there is an added parameter ρ. Similarly if w is high the edge-weight should reflect that. Here the tweakable parameter ω is used to adjust where the algorithm saturates[5]. The denominator, k is used as a dampening factor to regulate propagation. $\lim_{\rho,\omega\to 0} \varepsilon_{i,j} = \mu/k$.

Propagation. For a given context $\mathcal{C}$, after $\mathcal{L}_\mathcal{C}$ and $\mathcal{M}_\mathcal{C}$ are calculated and new edges are added, $\mathcal{M}_\mathcal{C}$ is now added to $\mathcal{C}$ and the whole process is recomputed at the next iteration level, until no edges are added in an iteration (or the maximum $\varepsilon_{i,j}$ computed at this iteration is smaller than a threshold ε). At this point the propagation stops.

4 Performance Evaluation

The energy model algorithm was tested and benchmarked against LSA. This was primarily based on the Web-KB dataset [11].

The Web-KB dataset is a large dataset comprising of 9721 documents. Both LSA and the energy model were run on this data set and the resulting term-term associations were then sorted based on their weights and compared. The values of ρ and ω in the energy model was set to 7 and 17 respectively. The value of k in LSA was set to 40. Given that Web-KB is a heterogenous document collection, it is hard to determine a good value for these parameters. For homogenous document collections some value of k between 70 and 100 was found to be most effective [4]. No such estimate seems to exist for LSA on heterogenous document collections. The energy model is yet to be fully analyzed for its performance on datasets of different levels of heterogeneity. The results were compared to those of LSA and a positive correlation between them was found.

Before comparing, all the negative values returned by LSA were set to 0. The term graph was largely maintained on disk and hence the memory requirement was significantly less when compared to LSA.

Polysems in LSA and Energy Model. In LSA during the dimension reduction step if a polysemic dimension is elimated then terms from two different contexts will merge together and spurious associations are created [9].

In the energy model, relatedness of terms with respect to context $\mathcal{C}$ is directly proportional to the size of $\mathcal{R}_\mathcal{C}$, i.e., r (equation 3). When r is considerably large and has high energy, heavier edges are added between $(\mathcal{A} - \mathcal{B})$ and $(\mathcal{B} - \mathcal{A})$[6]; and the less the value of r, or the weaker is its energy, the chances of an edge getting added are also quite small. For the polysemy problem to occur in this

[5] The saturation happens approximately at the tripled value of the tweakable parameter, i.e., if ρ is set to 10 then $(1 - e^{-\frac{r}{\rho}})$ tends to one when r is 30.

[6] As r increases the number of edges added decreases, this means to say that only if it finds contexts which are quite similar the algorithm will extract semantics.

algorithm, it is not sufficient if a term is a polysem but a whole context having high energy, should be polysemic. For more details on performance evaluation refer to [9].

5 Conclusions

The energy model presents a qualitatively different approach from LSA for establishing latent associations. The model is based on primarily tweaking a term-term graph. It can incrementally compute associations and computations are usually localized, enabling the term-term graph to be very large. It also does not suffer from the polysem problem.

In future work, we plan to integrate document retrieval into the energy model. Primarily, the issue of ranking documents and correlating document rank with the energy of co-occurring terms seems to hold promise.

References

1. Bassu, D., and Behrens, C. (2003). Distributed LSI: Scalable Concept-based Information Retrieval with High Semantic Resolution. In *Proceedings of the 3rd SIAM International Conference on Data Mining (Text Mining Workshop)*, San Francisco, CA, May 3, 2003.
2. Chakrabarti, S. (2003). Mining the web: Discovering knowledge from hypertext data. *1st Edition. Elsevier Science (USA).*
3. Deerwester, S., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the Society for Information Science* **41**, 6, 391 – 407.
4. Dumais, S. T. (1992). Enhancing Performance in Latent Semantic Indexing (LSI) Retrieval. *Technical Report,* Bellcore.
5. Gorell, G., Webb, B. (2005). Generalized Hebbian Algorithm for Latent Semantic Analysis. In *Proceedings of InterSpeech'05,* Lisbon.
6. Hoffman, T. (1999). Probabilistic Latent Semantic Analysis. *Uncertainty in Artificial Intelligence, UAI'99,* Stockholm.
7. Kleinberg J. M. (1998). Authoritative Sources in a Hyperlinked Environment. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms.*
8. Landauer, T. K., Foltz, P. W., and Laham, D. (1998). Introduction to Latent Semantic Analysis. *Discourse Processes*, 25, 259-284.
9. Rachakonda, A. R. (2006). Incremental Aggregation of Latent Semantics. *Master's Thesis. International Institute of Information Technology,* Bangalore, June 2006.
10. van Rijsbergen, C. J. (1999). Automatic Text Analysis. Information Retrieval. *2nd Edition.*
11. Carnegie Mellon University World Wide Knowledge Base (Web-KB) project, `http://www-2.cs.cmu.edu/~webkb/`, as of 15 February 2006.

A New Algorithm for Fast All-Against-All Substring Matching

Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton

University of Victoria, Canada
{mgbarsky, stege, thomo}@cs.uvic.ca and cupton@uvic.ca

Abstract. We present a new and efficient algorithm to solve the 'threshold all vs. all' problem, which involves searching of two strings (with length N and M respectively) for finding all maximal approximate matches of length at least S and with up to K differences. The algorithm is based on a novel graph model, and it solves the problem in time $O(NMK^2)$.

1 Introduction

An important problem in the field of string matching is the extraction of exact and approximate common patterns from a set of strings. In special application areas such as biological sequence analysis, finding exact patterns only can miss a great deal of useful information.

The problem can be defined as "all-against-all approximate substring matching" [1,2,4], and is notorius for its computational difficulty [5]. In practice, various constraints are set for the sought solutions, such as the maximum allowed number of approximations or "errors" and the minimum length of substrings. Despite past attempts, this problem is far from being efficiently solved. Our contribution is a fast algorithm for solving "all-against-all approximate substring matching" for two strings.

A naive approach to this problem is to exhaustively test each pair of substrings from s and t respectively. This approach has $O(N^2M^2)$ time complexity.

The best known solution was proposed by Baeza-Yates and Gonnet in [1,2], and is widely used [8]. Their solution significantly improved the average time complexity of the naive approach, by avoiding the examination of repeating substrings. In their method, the two input strings are organized into a suffix tree structure, and the order of substrings comparisons is guided by a depth-first traversal of the suffix tree nodes. The time complexity based on their practical results lies between NM (best case) and N^2M^2 (worst case), but closer to N^2M^2 [4]. Setting threshold criteria bounding the error number, i.e., allowing at most K differences in an approximate substring match, significantly improves the performance of the Baeza-Yates and Gonnet algorithm. This is because the value of K can be directly incorporated into their algorithm to cut down the depth of the suffix tree traversal. As we verify through experiments, for small values of K, the Baeza-Yates and Gonnet algorithm performs very well. However, as

F. Crestani, P. Ferragina, and M. Sanderson (Eds.): SPIRE 2006, LNCS 4209, pp. 360–366, 2006.

K increases, the number of suffix tree nodes that are examined grows almost exponentially in K, which is in accordance with Ukkonen [7].

We cast the original problem into the problem of finding "maximal paths" in a special "matching" graph.[1] Via a careful study of this graph, we are able to derive interesting and useful properties that help us in devising a higly optimized depth-first search procedure for finding "maximal paths," which correspond to the solutions of the original string problem. Our proposed algorithm runs in $O(NMK^2)$ time, which is a significant improvement over the Baeza-Yates and Gonnet algorithm. Moreover, we experimentally show that our algorithm scales linearly (as opposed to quadratically) in K and it outperforms the Baeza-Yates and Gonnet algorithm by an order of magnitude for bigger values of K. Finally, our algorithm has an additional nice feature: it reversely depends on the alphabet size. This is contrary to the behavior of the Baeza-Yates and Gonnet algorithm, whose running time worsens with the increase of the alphabet size.

2 A Graph Model for the All-Against-All Substring Matching

Let Σ be a finite alphabet. A sequence of letters $a_1 a_2 \dots a_N$, where $a_i \in \Sigma$ is called a *string* over Σ. We denote strings with s and t. Given string s, we denote its i-th letter with $s[i]$, and we denote a *substring* of s starting at position i and ending at position j with $s[i, j]$. Substring $s[i, j]$ has length $j - i + 1$.

Let the *edit distance* for two strings s and t be the minimum number of edit operations needed to transform s into t, as defined in [4] . We say the pair (s, t) is a *K-bounded approximate match* if the edit distance between s and t is at most K.

Problem 1. All error-bounded approximate matches
INPUT: *Strings s and t over alphabet Σ, and positive integers K and S.*
OUTPUT: *All error bounded approximate maximal matches $(s[i, j], t[k, l])$ such that (1) the edit distance between $s[i, j]$ and $t[k, l]$ is at most K and (2) the lengths of both $s[i, j]$ and $t[k, l]$ are at least S.*

We solve Problem 1 by casting it to an equivalent problem on graphs induced by a "matching matrix".

The *matching matrix of s and t* ($\mathcal{M}_{s,t}$) is defined as

$$\mathcal{M}_{s,t}[i, j] = \begin{cases} 1 & \text{if } s[i] = t[j] \\ 0 & \text{otherwise.} \end{cases}$$

Based on matching matrix $\mathcal{M}$, we define a weighted directed graph $G_{\mathcal{M}}$ with vertices v_{ij} corresponding to the 1-elements of the matrix, and with (directed) edges defined in a "top-down" and "left-right" fashion as follows: there is an edge $e(v_{ij}, v_{kl})$ iff $i < k$ and $j < l$ (cf. Fig. 1).

[1] The full version of an article can be downloaded from [3].

	t	a	a	c	b	a	c	a
s		0	1	2	3	4	5	6
a	0	1	1	0	0	1	0	1
b	1	0	0	0	1	0	0	0
a	2	1	1	0	0	1	0	1
b	3	0	0	0	1	0	0	0
a	4	1	1	0	0	1	0	1

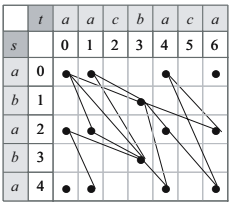


Fig. 1. Matching matrix and partial induced graph. Only edges of cost at most 3 are shown; the directions of the edges are left out.

We define the *cost* of an edge $e(v_{ij}, v_{kl})$ to be $c(v_{ij}, v_{kl}) = max(k-i, l-j)-1$. A *path* in graph $G_{\mathcal{M}}$ is a sequence of vertices connected by edges. For a path in $G_{\mathcal{M}}$, we define two characteristic properties. The *match length* of path π between v_{ij} and v_{kl} is defined as $ML(\pi) = min(k-i+1, l-j+1)$. The *error number*, $EN(\pi)$, is defined as the sum of all costs of edges in π.

Note that $G_{\mathcal{M}}$ is *not* a dynamic programming (induced) graph (*edit graph* [4]); DP graphs have been very well studied in the literature. However, to the best of our knowledge there is no work that formally studies the properties of $G_{\mathcal{M}}$ graph. Graph $G_{\mathcal{M}}$ possesses a very desirable property which is as follows.

Theorem 1. *The edit distance between $s[i,k]$ and $t[j,l]$ is equal to the error number of the cheapest path(s) from v_{ij} to v_{kl} in $G_{\mathcal{M}}$.*

Problem 2. ***All paths below threshold***
INPUT: *The graph $G_{\mathcal{M}}$ for two strings s and t, and positive integers K and S.*
OUTPUT: *All maximal paths with $EN \leq K$ and with match length at least S.*

Based on Theorem 1 we conclude that:

Corollary 1. *The problem* all bounded approximate matches *can be reduced to the* all paths below threshold *problem.*

We show next how to construct an instance for *all paths below threshold* from an instance of *all bounded approximate matches.*

3 Solving "All Paths Below the Threshold" (APBT)

We outline the logic flow of algorithm APBT, omitting all formal proofs due to space constraints. In the full version of the paper we give a simple way for building and storing the matching matrix in linear time and space. As for graph $G_{\mathcal{M}}$, we never explicitly construct and store it (remaining so linear w.r.t. space). Rather, as we show, we traverse it by constructing the needed paths "on the fly."

Path Expansion. We scan the matching matrix in row-major order. When a vertex of $G_{\mathcal{M}}$ is encountered, we initialize a path π with $EN(\pi) = 0$ and $ML(\pi) = 1$. The algorithm then builds all the possible expansions of this initial

path by adding one vertex at a time and by keeping track of the best paths found so far. As paths are constructed, the algorithm examines each partially completed path π: if no more vertices can be added without exceeding threshold K, then we stop the expansion and check whether $ML(\pi) \geq S$. If true, then we report path π as a solution.

A Single-Step Path Expansion. Since the error number of a path cannot exceed K, an edge to be appended to a path clearly has to have a cost of at most K. As a consequence all edges in $G_{\mathcal{M}}$ of cost higher than K are excluded from further consideration. Consider a path π with error number $EN(\pi)$, which ends at vertex v_{ij}. From the above discussion, it is clear that for a single-step expansion of π we need to search (in $\mathcal{M}$) for a possible "next vertex" only inside square $ABCD$, where $A = (i+1, j+1)$, and $C = (i+1+\kappa, j+1+\kappa)$, for $\kappa = K - EN(\pi)$. We call square $ABCD$ the *target square* for path π at vertex v_{ij}. The area of the target square decreases as the error number accumulated by π increases.

On the first sight, for any vertex v_{ij} in $G_{\mathcal{M}}$ there are at most $(\kappa+1) \times (\kappa+1)$ outgoing edges to be considered. We show how to reduce the number of edges for consideration. For this, we introduce the following definitions regarding diagonals in the matching matrix $\mathcal{M}$.

Let (i, j) be an arbitrary cell in $\mathcal{M}$. (1) The (i, j)-*main diagonal* for $\mathcal{M}$ is the sequence of $(i+p, j+p)$-cells in $\mathcal{M}$, where $0 \leq p \leq min\{M-i, N-j\}$. (2) Let q be a value between 0 and $N-j$. The (i, j)-q-*upper diagonal* is the $(i, j+q)$-main diagonal. (3) Let r be a value between 0 and $M-i$. The (i, j)-r-*lower diagonal* is the $(i+r, j)$-main diagonal.

Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) \leq K$. Now, assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, which lie on one of the upper diagonals w.r.t. v_{ij}. In terms of edge cost it means, that $c(v_{ij}, v_{kl}) = l - j - 1$, and $c(v_{ij}, v_{mn}) = n - j - 1$. Now, assume that $i < k < m$ and $j < l < n$. This means that if we build an edge from v_{ij} directly to v_{mn}, we "ignore" vertex v_{kl}, and unnecessarily increase $EN(\pi)$. Rather, we better expand path π to v_{kl} and later on, in the next round, continue to v_{mn}. Practically, this means that: if we find a vertex v_{kl} on an upper diagonal of the target square, then we can exclude from the search for single-step expansion all the triangular area of the target square, which is bounded by (1) row k (exclusive), and (2) the upper diagonal passing through v_{kl} (inclusive). Symmetrically, if vertex v_{kl} lies on some lower diagonal, then we can exclude from the search for expansion all the triangular area of the target square, which is bounded by (1) column l (exclusive), and (2) the lower diagonal passing through v_{kl} (inclusive). If vertex v_{kl} lies on the main diagonal of the target square, then both triangular areas are excluded at once.

In the full paper, we strengthen the above result by showing that we can safely exclude the row k (or column l) from the search for path expansion.

Optimization 1. *In search for expansions, we scan the cells of the target square in a diagonal-major order, that is: first scan the main diagonal, possibly excluding parts of the target square from further scan. Next, scan the remaining of the target*

square through the 1-upper diagonal and the 1-lower diagonal, possibly excluding other areas of the target square. Then, continue with the 2-upper diagonal and the 2-lower diagonal and so on.

Observe that, the scanning of a target square in this order guarantees that the exclusion of triangular areas takes place *as early as possible*.

Corollary 2. *Single path extension from an arbitrary vertex v_{ij} in $G_{\mathcal{M}}$ is performed at most once for each of the $2K+1$ diagonals surrounding v_{ij}, and therefore the number of possible extensions for v_{ij} is bounded by $2K+1$.*

Corollary 3. *An arbitrary cell of a matrix $\mathcal{M}[i,j]$ is accessed at most once from each of $2K+1$ diagonals. This also implies that an arbitrary vertex v_{ij} serves as a path extension for at most $2K+1$ vertices.*

Interdependence of Paths in $G_{\mathcal{M}}$. We show next how the information from previously explored paths can be reused.

Let π_1 be a previously explored path, which connects vertex v_{ij} with v_{mn}. Let π_2 be another path that we are currently exploring, which originates in v_{kl}, and is built up to vertex v_{mn}. Clearly, if $EN(\pi_2) \geq EN(\pi_1)$ and $ML(\pi_2) \leq ML(\pi_1)$, we can omit the further expansion of π_2. An illustration is given in Fig. 1, where path $v_{01}, v_{13}, v_{24}, v_{46}$ serves as π_1, which is explored earlier in a row-major order, and path v_{04}, v_{46} serves to exemplify π_2. Clearly, path π_2 will only offer a sub-solution to the solution corresponding to π_1, since the substring $t[4,6]$ is a substring of $t[1,6]$.

Thus, if we remember the smallest error number among all paths, which reached a particular vertex, then at each vertex, we will do at most $(K+1)$ expansions. The row major processing order ensures, that if ML is defined by the length of the vertical substring, then $ML(\pi_2) \leq ML(\pi_1)$. This is because both paths end at the same vertex, and π_2 starts at the same or later (greater) row than π_1. Notably, if we repeat the computation in a column-major order, all paths where ML was defined by the horizontal substring will now be defined by the vertical substring, thus ML of the later path will again be less than ML of the previously built path. For more detailed explanations see the full version.

The union of the solution sets of the two runs of the algorithm yields the final solution set.

From the above, we can conclude, that each vertex in $\mathcal{G}_{\mathcal{M}}$ is expanded at most $2(K+1)$ times.

Theorem 2. *The All Paths Below Threshold algorithm has a time complexity of $O(NMK^2)$.*

Proof. Since during path extension, any cell is accessed only once from at most $2K+1$ vertices (cf. Corollary 3) and each of these cells, if it is a vertex, is expanded at most $2(K+1)$ times (cf. discussion), the upper bound for traversing a particular cell of the matrix is at most $2(K+1)(2K+1)$. Since there are at most MN many cells in $\mathcal{M}$, the total time complexity is $O(NMK^2)$. □

The pseudocode of our algorithm is given below.

All_paths_below_threshold($\mathcal{M}$,K,S)
scan $\mathcal{M}$ in row major order
if $\mathcal{M}[i,j] = 1$ **then**
create a single-vertex v_{ij} path π
$EN(\pi) = 0$
Expand_path(π)

scan $\mathcal{M}$ in column major order
if $\mathcal{M}[i,j] = 1$ **then**
create a single-vertex v_{ij} path π
$EN(\pi) = 0$
Expand_path(π)

Expand_path(π)
if $ML(\pi) \geq S$ **then**
add π to the set of solutions

if a path with error number $EN(\pi)$
has already been extended through
v_{kl} **then** abort π and **return**

Do a single-step expansion (if possible) of π
creating new expanded path π_{exp}

Expand_path (π_{exp})

4 Experimental Evaluation

We present an experimental evaluation of our *All Paths Below Threshold* algorithm as it compares with the algorithm of Baeza-Yates and Gonnet [2].

We implemented Gusfield's variant of the Baeza-Yates and Gonnet algorithm [4].[2] We optimized it using Ukkonen's error bounded dynamic programming method [6] We abbreviate this optimized variant by $BYG + U$.

The running time was tested on the same 1.2 GHz PC with 312 MB of RAM.

Fig. 2 represents the running time of $BYG+U$ and $APBT$ on a pair of RNA sequences belonging to viruses from the same family, and where the minimum length of matches is set to $S = 50$. Notably the $APBT$ algorithm outperforms the $BYG+U$ algorithm for values of $K \geq 6$. We also show the size of the output, and this clearly shows that in order to obtain any output at all, even for similar RNA sequences, one has to set a bigger or equal to 6 value of K.

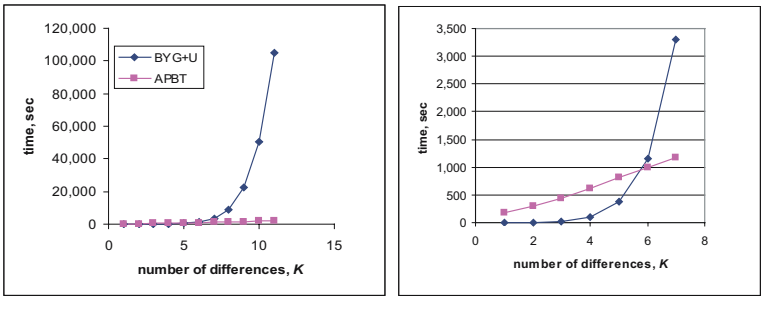


Fig. 2. Running time for two viral RNA sequences (30,000 bp): Human coronavirus 229E (27317 bp) and Human coronavirus OC43 (30738 bp) from [9]. The figure on the right is a zooming of the figure on the left for $K < 8$.

Interestingly, for $K \leq 5$, the $BYG + U$ algorithm outperforms the $APBT$ algorithm. This is because the $BYG+U$ algorithm benefits from the early stop of deeply going in the suffix trees, when the accumulated error exceeds K. However,

[2] The original code of [2] is unfortunately not available anymore.

as K grows the $BYG+U$ algorithm goes deeper in the suffix trees, and we observe an almost exponential in K increase in the running time. In contrast, $APBT$ scales on average linearly with K.

Also, we emphasize the fact that for alphabets of bigger size the $APBT$ algorithm performs better than the $BYG+U$ algorithm. The performance of $APBT$ is orders of magnitute better than $BYG+U$ for protein sequences with alphabet size of 20. This can be explained by the greater "bushiness" of the suffix trees (used by $BYG+U$) close to the root, and by the fact that with the increase of the alphabet size, our matching matrix becomes much sparser. The $APBT$ algorithm behaves so much better than the $BYG+U$ algorithm that we had to plot their behavior in different scales (cf. Fig. 3).

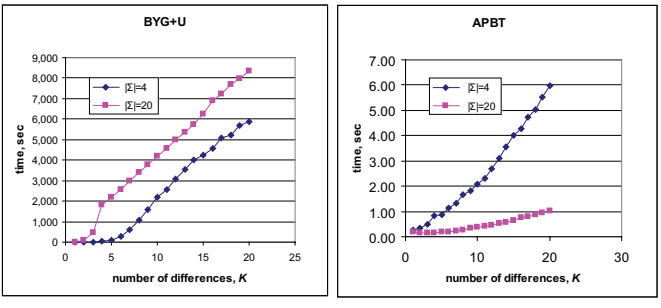


Fig. 3. Effect of alphabet size (random strings pairs of length 1000)

References

1. Baeza-Yates R.A., and Gonnet G.H. All-against-all sequence matching. *Rep. Dept. of CS, U. de Chile*, 1990.
2. Baeza-Yates R.A., and Gonnet G.H. A fast algorithm on average for all-against-all sequence matching. *Proc. SPIRE/CRIWG '99*, pp. 16–23.
3. Barsky M., Stege U., Thomo A., and Upton C.A. A New Algorithm for Fast All-Against-All Substring Matching. *http://www.cs.uvic.ca/∼mgbarksy/apbt.pdf*, 2006.
4. Gusfield D. Algorithms on Strings, Trees and Sequences. Cambridge University Press, 1997.
5. Pevzner P., and Sze S.H. Combinatorial approaches to finding subtle signals in DNA sequences. *Proc. ISMB '00*, pp. 269-278.
6. Ukkonen E. Algorithms for approximate string matching. *Information and Control* 64: 100–18, 1985.
7. Ukkonen E. Approximate string matching over suffix trees. *CPM93*, LNCS 684, 228–242, 1993.
8. Vilo J. Pattern Discovery from Biosequences. PhD Thesis, Series of Publications A, Report A-2002-3 U. of Helsinki, Finland, 2002.
9. Virus Orthologous Clusters database at *http://athena.bioc.uvic.ca* Viral Bioinformatics Resource Center, U. of Victoria, Canada.

Author Index