Gregory Kucherov
Esko Ukkonen (Eds.)

LNCS 5577

Combinatorial Pattern Matching

20th Annual Symposium, CPM 2009
Lille, France, June 2009
Proceedings

Springer

Gregory Kucherov Esko Ukkonen (Eds.)

Combinatorial Pattern Matching

20th Annual Symposium, CPM 2009
Lille, France, June 22-24, 2009
Proceedings

Springer

Volume Editors

Gregory Kucherov
Laboratoire d'Informatique Fondamentale de Lille (LIFL)
Bâtiment M3, 59655 Villeneuve d'Ascq CEDEX, France
E-mail: gregory.kucherov@lifl.fr

Esko Ukkonen
University of Helsinki, Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
00014 University of Helsinki, Finland
E-mail: esko.ukkonen@cs.helsinki.fi

Library of Congress Control Number: 2009930681

CR Subject Classification (1998): F.2, I.5, H.3.3, J.3, I.4.2, E.4, G.2.1, E.1

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

ISSN 0302-9743
ISBN-10 3-642-02440-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02440-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12696680 06/3180 5 4 3 2 1 0

Preface

It is our great pleasure to introduce the proceedings of the 20th anniversary edition of the Annual Symposium on Combinatorial Pattern Matching (CPM). The meeting was held in Lille, France, hosted by the Laboratoire d'Informatique Fondamentale de Lille (LIFL) affiliated with the Université de Lille 1 and the French Centre National de Recherche Scientifique (CNRS), as well as by INRIA Lille - Nord Europe.

Started in 1990 as a summer school with about 30 invited participants, CPM quickly evolved into a representative annual international conference. Principally motivated by combinatorial algorithms for search problems in strings (texts, sequences), the scope of CPM extended to more complex data structures such as trees, graphs, two-dimensional arrays, or sets of points. Those studies resulted in a rich collection of algorithmic techniques and data structures, making bridges to other parts of the theory of discrete algorithms and algorithm engineering. Today, the area of combinatorial pattern matching is a well-identified active subfield of algorithmic research.

Importantly, this development has been fertilized by a number of major application areas providing direct motivations and fruitful feedback to the CPM problematics. Those applications include data compression, computational biology, Internet search, data mining, information retrieval, coding, natural language processing, pattern recognition, music analysis, and others. On the one hand, all these areas make use of combinatorial pattern matching techniques and, on the other hand, raise new pattern matching problems. For example, the fast progress in computational molecular biology, triggered in the 1990s by the availability of mass genomic data, considerably influenced the combinatorial pattern matching field: as an illustration, about one-third of the papers presented in this volume deal with problems related to bioinformatics applications.

In 2009, the Combinatorial Pattern Matching symposium celebrated its 20th anniversary. Previous CPM meetings were held in Paris, London (UK), Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, Jeju Island, Barcelona, London (Canada), and Pisa. Starting from the third meeting, proceedings were published in the LNCS series, volumes 644, 684, 807, 937, 1075, 1264, 1448, 1645, 1848, 2089, 2373, 2676, 3109, 3537, 4009, 4580, and 5029. Selected papers from the first meeting in 1990 appeared in volume 92 of *Theoretical Computer Science*, from the 2000 meeting in volume 2 of *Journal of Discrete Algorithms*, from the 2001 meeting in volume 146 of *Discrete Applied Mathematics*, from the 2003 meeting in volume 3 of *Journal of Discrete Algorithms*, from the 2004 meeting in volume 368 of *Theoretical Computer Science* and from the 2005 meeting in volume 5 of *Journal of Discrete Algorithms*. Selected papers from the 2008 edition are to appear in *Theoretical Computer Science*.

To mark the 20th anniversary of CPM, all Program Committee (Co-)Chairs of previous editions of CPM were invited to serve on the 2009 Program Committee. This resulted in a committee of 31 members including many prominent researchers in the area. Moreover, the proceedings open with a special invited contribution entitled "CPM's 20th Anniversary: A Statistical Retrospective," tracing the history of the symposium and providing a collection of statistical data and factual information about all the 20 editions of CPM and the presented contributions.

The Program Committee received 63 valid submissions. Each submission was reviewed independently by three committee members, possibly assisted by external reviewers. About 70 external reviewers provided their expertise; they are listed on the pages that follow. The selection process resulted in 27 accepted papers, corresponding to an acceptance rate of about 43%. The Program Committee decided to grant two awards to selected papers: a Best Paper Award and a new Best Student Paper Award. We would like to thank the members of the Program Committee who worked very hard to ensure the timely review of all the submitted manuscripts and participated in the selection process.

The conference program also included three invited talks by Christos Faloutsos (Carnegie Mellon University), Roberto Grossi (University of Pisa), and Ravi Kumar (Yahoo! Research), who graciously accepted the Program Committee's invitation.

We are indebted to the members of the Steering Committee for their advice and tremendous help in different issues. On behalf of the entire CPM community, we would like to express our gratitude to the institutional sponsors who provided support to CPM 2009. These include the Laboratoire d'Informatique Fondamentale de Lille (UMR CNRS 8022), Université Lille 1, Région Nord-Pas de Calais, GDR Bioinformatique Moléculaire, INRIA Lille - Nord Europe, Yahoo! Research, and the University of Helsinki. The whole submission and review process was carried out with the help of the EasyChair system. Finally, we thank the local organization team headed by Hélène Touzet for carrying out all the laborious work that made the meeting possible.

March 2009 Gregory Kucherov
Esko Ukkonen

Organization

Program Committee

Steering Committee

Alberto Apostolico	Georgia Tech, USA and University of Padova, Italy
Maxime Crochemore	King's College London, UK and Université Paris-Est, France
Zvi Galil	Tel Aviv University, Israel

Organizing Committee

Sandrine Catillon
Mathieu Giraud
Stéphane Janot
Juha Kärkkäinen
Janne Korhonen
Gregory Kucherov (Co-chair)
Antoine de Monte
Laurent Noé
Maude Pupin
Hélène Touzet (Co-chair)
Jean-Stéphane Varré

External Referees

Can Alkan
Pavlos Antoniou
Hideo Bannai
Marek Biskup
Guillaume Blin
Carlos Brizuela
Shihyen Chen
Hamid Chitsaz
Manolis Christodoulakis
David Eppstein
Antonio Fariña
Thomas Fernique
Fedor Fomin
Roberto Grossi
Xi Han
Christophe Hancart
Lin He
Danny Hermelin
Farhad Hormozdiari
Fereydoun Hormozdiari
Lucian Ilie
Shunsuke Inenaga
Jesper Jansson
Inuka Jayasekera
Hossein Jowhari
Juha Kärkkäinen
Takuya Kida
Shmuel Tomi Klein
Pang Ko
Mikko Koivisto
Marcin Kubica
Oded Lachish
Giuseppe Lancia
Theodoros Lappas
Thierry Lecroq
Weiming Li
Hao Lin
Jingping Liu
Xiaowen Liu
Mercè Llabrés
Spiros Michalakopoulos
Lukasz Mikulski
Igor Nitto
Giulio Pavesi
Marcin Piatkowski
Solon Pissis
Wojciech Plandowski
Ely Porat
Jakub Radoszewski
M. Sohel Rahman
Emanuele Raineri
Antonio Restivo

Paolo Ribeca
Luís M.S. Russo
Kunihiko Sadakane
Mert Saglam
Hiroshi Sakamoto
Gilles Schaeffer
Alexander Schönhuth
Florian Sikora
Bill Smyth
Wojciech Szpankowski
Zdenek Tronícek
Bora Uyar
Rossano Venturini
Stéphane Vialette
Mark Ward
Oren Weimann
Michal Ziv-Ukelson

Sponsoring Institutions

Laboratoire d'Informatique Fondamentale de Lille (UMR CNRS 8022)
Université Lille 1
Région Nord-Pas de Calais
GDR Bioinformatique Moléculaire
INRIA Lille - Nord Europe
Yahoo! Research
University of Helsinki

Table of Contents

CPM's 20th Anniversary: A Statistical Retrospective

Elena Yavorska Harris[1], Thierry Lecroq[2],
Gregory Kucherov[3], and Stefano Lonardi[1]

[1] Dept. of Computer Science – University of California – Riverside, CA, USA
[2] University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
[3] CNRS (LIFL, Lille and J.-V.Poncelet Lab, Moscow) and INRIA Lille – Nord Europe

1 Introduction

This year the Annual Symposium on Combinatorial Pattern Matching (CPM) celebrates its 20th anniversary. Over the last two decades the Symposium has established itself as the most recognized international forum for research in combinatorial pattern matching and related applications. Contributions to the conference typically address issues of searching and matching strings and more complex patterns such as trees, regular expressions, graphs, point sets, and arrays. Advances in this field rely on the ability to expose combinatorial properties of the computational problem at hand and to exploit these properties in order to either achieve superior performance or identify conditions under which searches cannot be performed efficiently. The meeting also deals with combinatorial problems in computational biology, data compression, data mining, coding, information retrieval, natural language processing and pattern recognition.

The first edition of CPM was held in Paris in July 1990, and gathered about thirty participants. Since then the conference has been held every year, usually in June or July. Thirteen countries, over three continents, have hosted it (see Table 1). The "seed" of CPM can be traced back to a NATO-ASI Workshop in Maratea, Italy organized by Z. Galil and A. Apostolico. The volume collecting the contributions presented at the workshop [1] defined perhaps for the first time the scope of this research area, sometimes referred to as "stringology". The intent of the first two editions of CPM was to reconnect with the participants and to the spirit of the NATO-ASI meeting in Maratea. CPM'90 and CPM'91 were organized like schools with neither submission/refereeing process nor proceedings. For CPM'92, however, NSF funding was contingent upon having a Program Committee and printed proceedings, so the Symposium was born.

Selected papers from the 1990 meeting were published in a special issue of *Theoretical Computer Science* [2]. Since 1992, submitted papers have been peer-reviewed and accepted contributions have been published in Lecture Notes in Computer Science (Springer-Verlag). CPM proceedings have been published in the LNCS series, volumes 644 [8], 684 [9], 807 [10], 937 [11], 1075 [12], 1264 [13], 1448 [14], 1645 [15], 1848 [16], 2089 [17], 2373 [18], 2676 [19], 3109 [20], 3537 [21], 4009 [22], 4580 [23], and 5029 [24].

The practice of inviting a selected subset of the accepted papers for journal publication was resumed with the 11th meeting which appeared in volume 2 of *Journal of Discrete Algorithms* [5]. Then again, papers from the 12th meeting in volume 146 of *Discrete Applied Mathematics* [3], from the 14th meeting in volume 3 of *Journal of*

Table 1. Locations, dates, program chairs, and number of PC members for all CPM editions

Date	**Location**	**Chair(s)**	**|PC|**
1990, July 9-13	Paris, France	M. Crochemore	N/A
1991, April 17-19	London, UK	C.S. Iliopoulos	N/A
1992, April 29 - May 1	Tucson, AZ, USA	U. Manber	10
1993, June 2-4	Padova, Italy	A. Apostolico	10
1994, June 5-8	Asilomar, CA, USA	M. Crochemore, D. Gusfield	10
1995, July 5-7	Helsinki, Finland	Z. Galil, E. Ukkonen	10
1996, June 10-12	Laguna Beach, CA, USA	D. Hirschberg, Z. Galil	12
1997, June 30 - July 2	Aarhus, Denmark	A. Apostolico, J. Hein	12
1998, July 20-21	Piscataway, NJ, USA	M. Farach, U. Manber	15
1999, July 22-24	Warwick, UK	M. Crochemore, M. Paterson	15
2000, June 21-23	Montréal, Canada	R. Giancarlo, D. Sankoff	14
2001, July 1-4	Jerusalem, Israel	A. Amir, G. Landau	20
2002, July 3-5	Fukuoka, Japan	A. Apostolico, M. Takeda	17
2003, June 25-27	Morelia, Michocán, Mexico	R. Baeza-Yates, E. Chavez, M. Crochemore	19
2004, July 5-7	Istanbul, Turkey	U. Dogrusoz, S. Muthukrishnan, S. C. Sahinalp	16
2005, June 19-22	Jeju Island, Korea	A. Apostolico, M. Crochemore, Kunsoo Park	20
2006, July 5-7	Barcelona, Spain	M. Lewenstein, G. Valiente	19
2007, July 9-10	London, Ontario, Canada	K. Zhang, B. Ma	26
2008, June 18-20	Pisa, Italy	G. M. Landau, P. Ferragina	27
2009, June 22-24	Lille, France	G. Kucherov, E. Ukkonen	31

Discrete Algorithms [4], from the 15th meeting in volume 368 of *Theoretical Computer Science* [7] and from the 16th meeting in volume 5 of *Journal of Discrete Algorithms* [6]. Selected papers from CPM'08 are expected to appear this year in *Theoretical Computer Science*.

A total of 127 individuals has served in the 18 program committees (including 2009). The size of the PC has increased from ten in the first few years to a record thirty-one for 2009. For the twentieth anniversary of CPM all previous PC chairs were invited to serve as PC members.

2 Submitted Papers and Acceptance Rates

A total of 460 peer-reviewed papers have been published in the conference proceedings up to 2008 (including the TCS special issue for CPM'90). While the number of accepted papers has been relatively stable over the years, the number of submitted papers to the Symposium varied greatly (see Table 2). The maximum number of submission (129) was recorded for CPM'05 held in Korea, and the minimum (26) was reached in 1999. From 1992 to 2009, a total 988 papers have been submitted to CPM.

Table 2. Number of accepted and submitted papers, acceptance ratio, number of authors, number of new authors, and average number of authors per paper

	Accepted	Submitted	Ratio %	Authors	New	Avg authors/paper
1992	22	39	56.4	43	38	2.04
1993	19	34	55.9	33	16	1.89
1994	26	41	63.4	52	38	2.15
1995	29	44	65.9	52	35	2.03
1996	28	48	58.3	61	34	2.28
1997	20	32	62.5	51	33	2.6
1998	17	49	34.7	42	22	2.52
1999	21	26	80.8	44	27	2.43
2000	29	44	65.9	64	38	2.03
2001	22	35	62.9	46	23	2.5
2002	23	37	62.2	58	33	2.78
2003	28	57	49.1	63	36	2.36
2004	36	79	45.6	95	62	2.94
2005	37	129	28.7	98	63	2.73
2006	33	88	37.5	82	38	2.67
2007	32	64	50	84	36	2.91
2008	25	78	32.1	67	27	2.84
2009	27	63	42.9	84	35	3.11

While the average acceptance rate is about 56%, the spread of the distribution is quite wide. The lowest acceptance rate (29%) was recorded in 2005, the highest (81%) was reached in 1999. Table 2 shows the number of submitted papers and the acceptance rates over the years.

3 Conference Proceedings: An Analysis of Authorship

A total of 597 distinct authors have published peer-reviewed papers in the conference proceedings (including the TCS special issue for CPM'90). Out of these, 393 authors have published only once in the proceedings. There are 97 authors that published twice, 40 authors with three papers, and 25 authors with four. Authors that published more than five papers in CPM are listed in Table 3. While every effort was made to normalize the names of the author throughout the years, inaccuracies might be still present which might bias the statistics.

Table 2 reports the average number of authors for each CPM edition with proceedings published in LNCS. Note that the average is clearly increasing – it was about two authors/paper in the early nineties, and it is currently approaching an average of three authors. The increase in the number of authors is a general trend in the Sciences and has been observed in several disciplines.

We have also carried out an analysis of new authors in each CPM edition. We counted an author to be "new" if he or she had never published in CPM before. Table 2 shows that each year a large fraction of the authors publishing papers in CPM are first-timers.

Table 3. Authors with more than five papers in the CPM proceedings

Author	CPM papers	Author	CPM papers
Gonzalo Navarro	25	Tao Jiang	7
Kaizhong Zhang	15	Dong Kyue Kim	6
Amihood Amir	12	Ely Porat	6
Gad Landau	12	Jens Stoye	6
Kunsoo Park	12	John Kececioglu	6
Leszek Gąsieniec	10	Jorma Tarhio	6
Masayuki Takeda	10	Juha Kärkkäinen	6
Wojciech Rytter	10	Mathieu Raffinot	6
Bin Ma	9	Ming Li	6
Costas Iliopoulos	9	William Smyth	6
Veli Makinen	9	Wojciech Szpankowski	6
Maxime Crochemore	9	Eugene Myers	6
Ayumi Shinohara	8	Dekel Tsur	5
Michal Ziv-Ukelson	8	Mireille Régnier	5
Setsuo Arikawa	8	Rolf Backofen	5
Lusheng Wang	8	S. Muthukrishnan	5
Esko Ukkonen	7	Stéphane Vialette	5
David Sankoff	7	Tak-Wah Lam	5
Moshe Lewenstein	7	Tatsuya Akutsu	5
Pavel Pevzner	7	Wojciech Plandowski	5
Ricardo Baeza-Yates	7	Dan Gusfield	5

To evaluate the dependency between geographic location and contributions to CPM, we looked at the country of affiliation of authors over the years. More specifically, we counted how many papers have at least one author for a given country of affiliation. The resulting graph is shown in Figure 1. It is interesting to note that some countries have kept a somewhat steady stream of papers, e.g., Canada, France, UK, Chile. In contrast, contributions from the US are showing a clear decline from the early nineties. The number of papers from Israel have been showing a significant increase since 2002.

In order to further analyze the relationships between authors, we built the collaboration network $G = (V, E)$ where nodes in V correspond to authors, and $(u, v) \in E$ if u and v co-authored a paper in CPM. The graph has a total of 597 nodes and 946 edges, which results in an average degree of 3.18 edges. The resulting network has 114 connected components, of which 40 are single nodes, 39 are components of size 2, 17 of size 3, 10 of size 4 and one/two of size 5,6,7,9 and 11.

The largest connected component of the collaboration network is composed of 348 nodes (see Figure 2). In the figure, nodes and label sizes have been drawn proportional to the node degree. There are four nodes with the degree 20 or more: Costas Iliopoulos with 23 links, Amihood Amir with 21, Gonzalo Navarro with 21 and Gad Landau with 20. A high resolution picture of the graph can be downloaded from `http://www.cs.ucr.edu/~stelo/cpm/`. We computed the graph *diameter* which is the longest shortest path in the graph, the *average clustering coefficient* which measures the extent to which vertices linked to any given vertex are also linked to each other, and the *characteristic path length* which is the average shortest path distance between pairs of

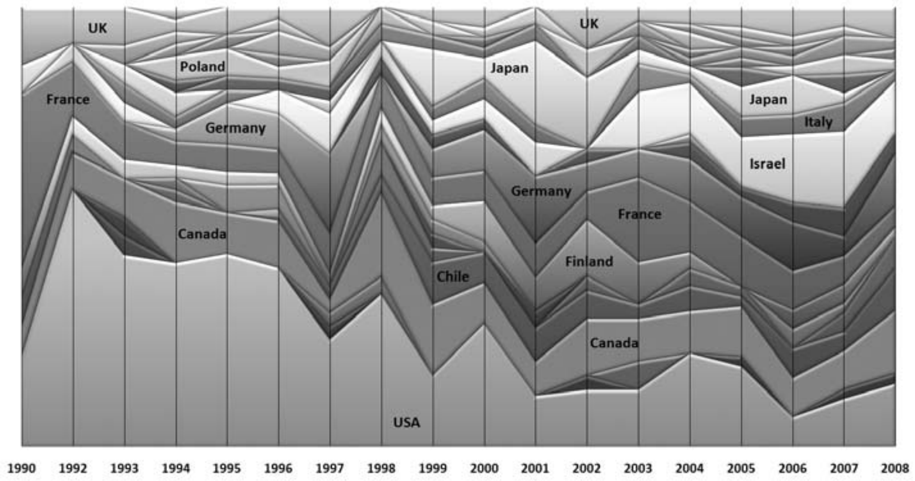


Fig. 1. Fraction of CPM papers which have at least one authors for a given country of affiliation

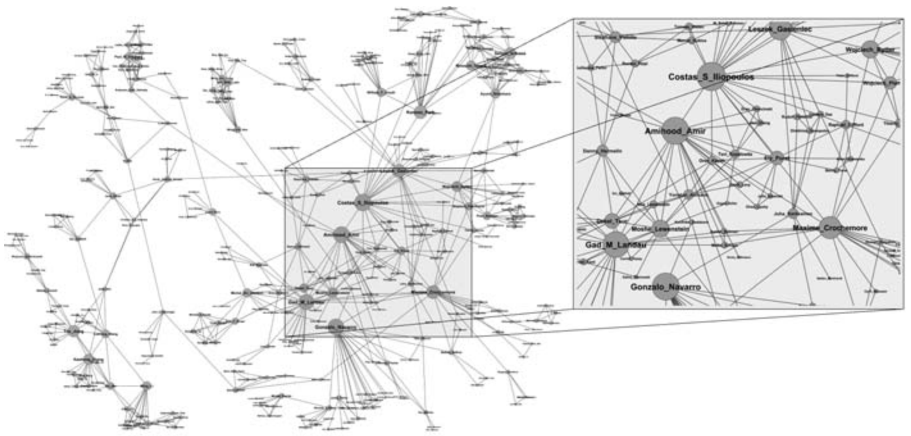


Fig. 2. The largest connected component of the CPM collaboration network. The size of each node/label is proportional to its degree. A high resolution picture of this network can be downloaded from `http://www.cs.ucr.edu/~stelo/cpm/`

vertices. The diameter of the largest connected component of the collaboration network is 17 edges, the average clustering coefficient is 0.8, and the characteristic path length is 6.49.

We have also computed *central* nodes for the largest connected component. Central nodes are the ones which have the smallest average shortest path length to all the other nodes. The top five central nodes are Maxime Crochemore (average shortest path length 3.95389), Wojciech Rytter (4.04323), Costas Iliopoulos (4.12392), Wojciech Plandowski (4.12392) and Leszek Gąsieniec (4.22767).

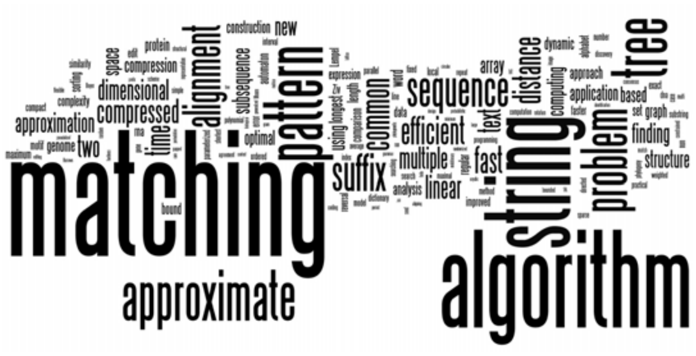


Fig. 3. Frequency analysis of CPM paper titles using Wordle (`http://www.wordle.net/`)

4 Conference Proceedings: An Analysis of Titles

We carried out a simple analysis of the most frequent words contained in the titles over the years. We considered only one occurrence of each term per title, and merged counts for singular and plural. Not surprisingly, “matching”, ”algorithm” and “string” are the most frequent terms, with 118, 92 and 81 occurrences, respectively. The word “pattern” appears less frequently (56 times), but not as rarely as “combinatorial” which occurs only three times! Figure 3 illustrates the frequency analysis using Wordle (`http://www.wordle.net/`).

Other words that appear more than twenty times in titles are: tree (53), approximate (48), problem (44), sequence (36), alignment (36), suffix (35), common (29), efficient (27), distance (27), fast (26), time (24), compressed (23), text (22), linear (22), two (21), multiple (21), and dimensional (20).

When searching for patterns composed by two words, the pairs “string-matching” (50), “pattern-matching” (41) and “approximate-matching” (37) are the most frequent. Other pairs that occur at least ten times are: algorithm-matching (28), approximate-string (26), algorithm-string (22), suffix-tree (17), array-suffix (16), algorithm-tree (16), common-subsequence (15), fast-matching (14), dimensional-matching (14), dimensional-two (13), compressed-text (13), common-longest (13), matching-two (12), linear-time (12), dimensional-pattern (12), alignment-multiple (12), algorithm-problem (12), algorithm-pattern (12), algorithm-fast (12), algorithm-approximation (12), longest-subsequence (11), alignment-sequence (11), algorithm-efficient (11), expression-regular (10), compressed-matching (10), and algorithm-alignment (10).

We also looked at patterns composed by three words. Six patterns occurs at least ten times, namely “approximate-matching-string” (24), “algorithm-matching-string” (12), “dimensional-matching-two” (11), “dimensional-matching-pattern” (11), “common-longest-subsequence” (11) and “algorithm-matching-pattern” (10). The most frequent pattern composed of four terms is “dimensional-matching-pattern-two” (8). The two most frequent patterns composed of five terms are “dimensional-matching-pattern-rotation-two” (4) and “compressed-Lempel-matching-text-Ziv” (4).

5 Conference Proceedings: An Analysis of Citations

Several papers that appeared in proceedings of CPM had a significant impact on the field of computer science, and some are consider to be seminal works. We carried out an analysis of the citation count for each of the 460 published articles using Google Scholar. We should point out that Google Scholar provides merely an approximation for the exact number of citations, and by no means our analysis or ranking should be taken literally.

Table 4. Most cited CPM papers according to Google Scholar, as of Feb 2009. The counts are the sum of the citations to the conference version and the corresponding journal version, if there is one with the exact same title. * refers to the journal version appeared in JCB 2001, with an additional author (D. Sokol).

Author(s)	Title	Citations	CPM Year
Esko Ukkonen	Approximate string-matching with q-grams and maximal matches	235	1990
Eugene Myers	A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming	172	1998
Pang Ko, Srinivas Aluru	Space Efficient Linear Time Construction of Suffix Arrays	148	2003
Tao Jiang, Lusheng Wang, Kaizhong Zhang	Alignment of Trees - An Alternative to Tree Edit	138	1994
Gad M. Landau, Jeanette P. Schmidt	An Algorithm for Approximate Tandem Repeats	137*	1993
Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, Kunsoo Park	Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications	135	2001
Esko Ukkonen	Approximate String-Matching over Suffix Trees	125	1993
Dong Kyue Kim, Jeong Seop Sim, Heejin Park, Kunsoo Park	Linear-Time Construction of Suffix Arrays	114	2003
Ricardo A. Baeza-Yates, Walter Cunto, Udi Manber, Sun Wu	Proximity Matching Using Fixed-Queries Trees	113	1994
Anne Bergeron	A Very Elementary Presentation of the Hannenhalli-Pevzner Theory	111	2001
John Kececioglu, David Sankoff	Exact and Approximation Algorithms for the Inversion Distance Between Two Chromosomes	108	1993
Udi Manber	A Text Compression Scheme That Allows Fast Searching Directly in the Compressed File	99	1994
Lucas Chi Kwong Hui	Color Set Size Problem with Application to String Matching	93	1992
Sridhar Hannenhalli	Polynomial-time Algorithm for Computing Translocation Distance between Genomes	93	1995
John Kececioglu, David Sankoff	Efficient Bounds for Oriented Chromosome Inversion Distance	92	1994
Vineet Bafna, Eugene L. Lawler, Pavel A. Pevzner	Approximation Algorithms for Multiple Sequence Alignment	90	1994
Dominique Revuz	Minimisation of acyclic deterministic automata in linear time	87	1990
Jotun Hein, Tao Jiang, Lusheng Wang, Kaizhong Zhang	On the Complexity of Comparing Evolutionary Trees	86	1995
Dan Gusfield	Haplotype Inference by Pure Parsimony	83	2003
John Kececioglu	The Maximum Weight Trace Problem in Multiple Sequence Alignment	82	1993
Stefan Burkhardt, Juha Kärkkäinen	Better Filtering with Gapped q-Grams	79	2001
William I. Chang, Jordan Lampe	Theoretical and Empirical Comparisons of Approximate String Matching Algorithms	78	1992
Ricardo A. Baeza-Yates, Chris H. Perleberg	Fast and Practical Approximate String Matching	77	1992
Vineet Bafna, S. Muthukrishnan, R. Ravi	Computing Similarity between RNA Strings	75	1995
William I. Chang, Thomas G. Marr	Approximate String Matching and Local Similarity	74	1994
Archie L. Cobbs	Fast Approximate Matching using Suffix Trees	69	1995
Gonzalo Navarro, Mathieu Raffinot	A General Practical Approach to Pattern Matching over Ziv-Lempel Compressed Text	69	1999
Jens Stoye, Dan Gusfield	Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree	69	1998
Juha Kärkkäinen	Suffix Cactus: A Cross between Suffix Tree and Suffix Array	67	1995
Steffen Heber, Jens Stoye	Finding All Common Intervals of k Permutations	66	2001
Erkki Sutinen, Jorma Tarhio	Filtration with q-Samples in Approximate String Matching	65	1996
Tzvika Hartman	A Simpler 1.5-Approximation Algorithm for Sorting by Transpositions	65	2003
Gonzalo Navarro, Erkki Sutinen, Jani Tanninen, Jorma Tarhio	Indexing Text with Approximate q-Grams	62	2000
Vincent A. Fischetti, Gad M. Landau, Jeanette P. Schmidt, Peter H. Sellers	Identifying Periodic Occurrences of a Template with Applications to Protein Structures	62	1992
Chia-Hsiang Chang, Robert Paige	From Regular Expressions to DFA's Using Compressed NFA's	61	1992
Gautam Das, Rudolf Fleischer, Leszek Gąsieniec, Dimitrios Gunopulos, Juha Kärkkäinen	Episode Matching	60	1997

Table 4 shows papers cited at least sixty times as of February 2009, according to Google Scholar. We note that Google Scholar merges citations of both conference and journal version of the same paper when both manuscripts share the same title. The number of citations in the table reflects both conference version and the corresponding journal version (if there is one). However, if the journal version was published with a different title the citation counts are not added, which introduces a bias. The table also includes the number of citations for the journal version of a paper by G. Landau and J. Schmidt initially appeared in CPM'93, and later published with the same title but with an additional author (D. Sokol) in *J. Computational Biology*.

The list in Table 4 includes 36 contributions ranging from CPM'90 to CPM'03. The top ten most cited articles include two very recent papers on the construction of suffix arrays in linear time. A cursory inspection of the titles in the table reveals a number of contributions related to approximate string matching, selected problems in computational biology (genome rearrangement, haplotyping, multiple sequence alignment, and repeat analysis), and tree matching/alignment, among others.

6 CPM Invited Speakers

Each year, two to five distinguished scientists are invited to deliver lectures at the conference in a variety of fields. The list of speakers includes Alfred V. Aho (1990), Esko Ukkonen (1990 and 2005), Alberto Apostolico (1990 and 1991), Maxime Crochemore (1990 and 1991), Zvi Galil (1990, 1991 and 2001), Uzi Vishkin (1995 and 2001), H. W. Mewes (1995), David Lipman (1996), Richard Arratia (1996), A. Dress (1997), J. B. Kruskal (1997), Ken Church (1998), Mick Noordewier (1998), Joan Feigenbaum (1999), David Jones (1999), Andrei Broder (2000), Fernando Pereira (2000), Ian H. Witte (2000), Aviezri Fraenkel (2001), Rao Kosaraju (2001), Shinichi Morishita (2002), Hiroki Arimura (2002), Vladimir Levenshtein (2003), J. Ian Munro (2003 and 2008), Evan Eichler (2004), Martin Farach-Colton (2004), Paolo Ferragina (2004), Piotr Indyk (2004), Eugene Myers (2004), Ming Li (2005), Naftali Tishby (2005), Amihood Amir (2006), Eran Halperin (2006), Steven Skiena (2006), Tao Jiang (2007), S. Muthukrishnan (2007), Frances Yao (2007), Daniel M. Gusfield (2008), Prabhakar Raghavan (2008), Christos Faloutsos (2009), Roberto Grossi (2009), Ravi Kumar (2009).

7 CPM Funding

CPM has received support from a variety of sources, including Laboratoire d'Informatique Fondamentale de Lille, INRIA Lille Nord Europe, Université Lille 1, CNRS, University of Pisa, Yahoo! Research, University of Western Ontario, Fields Institute, Department of Software - Technical University of Catalonia, Spanish Ministry of Education and Science, Yahoo! Research Barcelona, Ministry of Science and Technology, Korea, Seoul National University, SIGTCS of the Korea Information Science Society, MNG, Center for Computational Genomics, DIMACS, Consejo Nacional de Ciencia y Tecnologia, Universidad Michoacana, The Caesarea Edmond Benjamin de Rothschild Foundation, Bar Ilan University, Haifa University, Université de Montréal, MATHFIT, DIMACS, University of Aarhus, BRICS, National Research Foundation of Denmark,

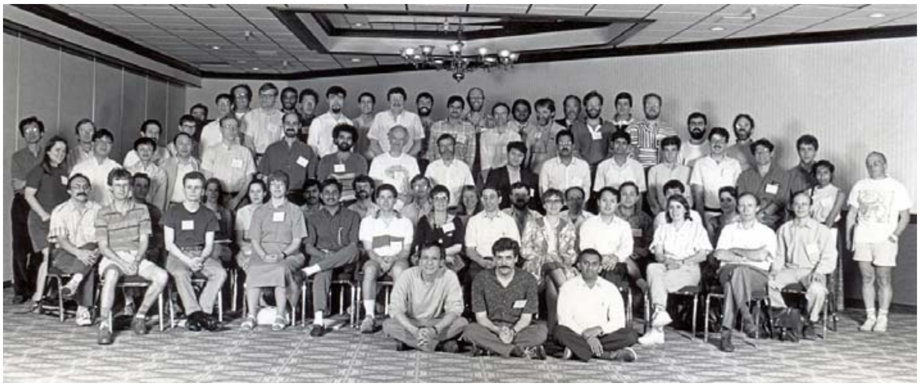


Fig. 4. Group photo of CPM'92 participants. **Standing back (right to left):** G. Benson, R. Baeza-Yates, A. Hume, J. Knight, C. Burks, H. Berghel, A. Ehrenfeucht, D. Roach, U. Manber, M. Waterman, R. Idury, A. Amir, G. Lawler, E. Port, W. Chang, M. Vingron, P. Pevzner, E. Ukkonen, F. Olken, X. Xu, A. Apostolico, T. Lecroq, and G. Havas. **Standing first row**: B. Pittel, M. Jain, D. Revuz, E. Myers, A. Schaffer, S. Seiden, D. Mehta, T. Choudhary, T. Cheung Ip, L. J. Cummings, R. Irving, S. Kannan, J. Kececioglu, P. Kilpelainen, K. Zhang, S. Wu, L. Hui, T. Warnow, and Y. D. Lyuu. **Sitting second row:** R. Paige, L. Rostami, J.Yong Kim, M. Farach, H. Wolfson, G. Landau, J. Schmidt, G. Herrmannsfeldt, D. Sankoff, R. Hariharan, L. Toniolo, C. Soderlund, D. Gusfield, and W. Szpankowski. **Sitting first row:** M. Crochemore, D. Joseph, X. Huang, M. Régnier, D. Hirschberg, M. McClure, G. Lewandowski, T. Vasi, B. Baker, C. Fraser, and P. Jacquet. **On the floor:** J. Oommen, G. Jacobson, and K. Phong Vo.

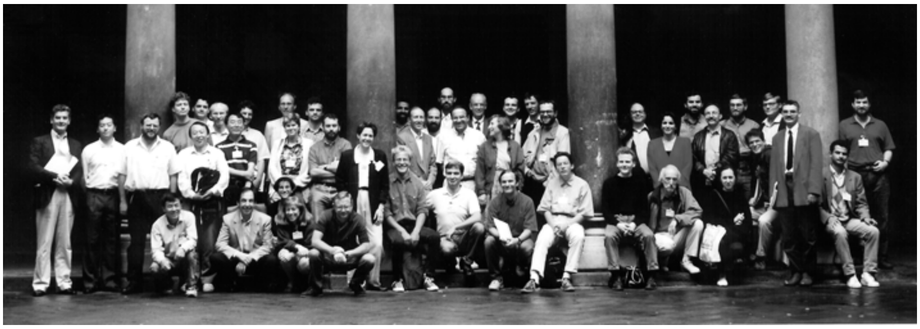


Fig. 5. Group photo of CPM'93 participants. **Standing (left to right):** A. Apostolico, T. Akutsu, G. Landau, D. Breslauer, K. Zhang, O. Delgrange, J. Tarhio, ?, ?, M. Waterman, E. Norel, ?, ?, L. Toniolo, S. Muthukrishnan? , M. Crochemore, G. Gonnet, J. Kececioglu, H. Wolfson, ?, M. Régnier, M. Frigo, W. Plandowski?, C. Iliopoulos, A. Lesk, L. Rostami, R. Baeza-Yates, W. Szpankowski, P. Pevzner, E. Ukkonen , L. Gąsienec, ?. **Sitting (left to right):** ?, L. Colussi, D. Naor, J. Schmidt, R. Giancarlo, ?, G. Bilardi, U. Manber, ?, ?, R. Irving, F. Tahi, E. Myers, S. Abdeddaïm

University of California – Irvine, Department of Information and Computer Science - UCI, Irvine Research Unit on Computer Systems Design, Academy of Finland, Ministry of Education of Finland, University of Helsinki, National Science Foundation, University of California – Davis, Padova Ricerche Consortium, National Science Foundation, University of Arizona, PRC Mathématiques et Informatique.

8 Concluding Remarks

The twentieth edition of CPM in Lille provides an opportunity to reflect on CPM's history and the impact of its research contributions to Computer Science. Most of scientists that shaped the discipline of Combinatorial Pattern Matching are still very active in this research area (see Figures 4 and 5 for a group photo of CPM'92 and CPM'93 participants). At the same time, the community around CPM has grown enough to make it a self-sustaining event, both financially and scientifically. We are certainly looking forward to the next twenty years!

Acknowledgements

We would like to thank the CPM Steering Committee for the opportunity to write this manuscript. In particular, our thanks go to Maxime Crochemore for reading an earlier version of the manuscript and suggesting improvements, and Alberto Apostolico for useful comments and for providing an account of the 1984 NATO-ASI Workshop and the early days of CPM.

References

1. Apostolico, A., Galil, Z. (eds.): Combinatorial Algorithms on Words. NATO ASI Series, Advanced Science Institutes Series, Series F, vol. 12. Springer, Heidelberg (1985)
2. Crochemore, M. (ed.): Proceedings of the 1st Annual Symposium on Combinatorial Pattern Matching. Theoret. Comput. Sci., 92(1) (1992)
3. Amir, A., Landau, G.M. (eds.): 12th Annual Symposium on Combinatorial Pattern Matching. Special issue of Discrete Applied Mathematics, vol. 146(2). Elsevier, Amsterdam (2005)
4. Baeza-Yates, R., Crochemore, M. (eds.): Indexing and Matching Strings. Special issue of J. Discrete Algorithms 3(2–4) (2005)
5. Crochemore, M., Gąsieniec, L. (eds.): Matching Patterns. Hermès (2000); Special issue of J. Discrete Algorithms
6. Park, K., Mouchard, L. (eds.): Selected papers from Combinatorial Pattern Matching, 2005. Special issue of J. Discrete Algorithms 5(4) (2005)
7. Sahinalp, S.C., Dogrusoz, U., Muthukrishnan, S. (eds.): Combinatorial Pattern Matching. Special issue of Theoretical Computer Science 368(3) (2006)
8. Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.): CPM 1992. LNCS, vol. 644. Springer, Heidelberg (1992)
9. Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.): CPM 1993. LNCS, vol. 684. Springer, Heidelberg (1993)
10. Crochemore, M., Gusfield, D. (eds.): CPM 1994. LNCS, vol. 807. Springer, Heidelberg (1994)

11. Galil, Z., Ukkonen, E. (eds.): CPM 1995. LNCS, vol. 937. Springer, Heidelberg (1995)
12. Hirschberg, D.S., Myers, E.W.: CPM 1996. LNCS, vol. 1075. Springer, Heidelberg (1996)
13. Apostolico, A., Hein, J.: CPM 1997. LNCS, vol. 1264. Springer, Heidelberg (1997)
14. Farach-Colton, M. (ed.): CPM 1998. LNCS, vol. 1448. Springer, Heidelberg (1998)
15. Crochemore, M., Paterson, M. (eds.): CPM 1999. LNCS, vol. 1645. Springer, Heidelberg (1999)
16. Giancarlo, R., Sankoff, D. (eds.): CPM 2000. LNCS, vol. 1848. Springer, Heidelberg (2000)
17. Amir, A., Landau, G.M. (eds.): CPM 2001. LNCS, vol. 2089. Springer, Heidelberg (2001)
18. Apostolico, A., Takeda, M. (eds.): CPM 2002. LNCS, vol. 2373. Springer, Heidelberg (2002)
19. Baeza-Yates, R. A., Chávez, E., Crochemore, M. (eds.): CPM 2003. LNCS, vol. 2676. Springer, Heidelberg (2003)
20. Sahinalp, S.C., Muthukrishnan, S., Dogrusoz, U. (eds.): CPM 2004. LNCS, vol. 3109. Springer, Heidelberg (2004)
21. Apostolico, A., Crochemore, M., Park, K. (eds.): CPM 2005. LNCS, vol. 3537. Springer, Heidelberg (2005)
22. Lewenstein, M., Valiente, G. (eds.): CPM 2006. LNCS, vol. 4009. Springer, Heidelberg (2006)
23. Ma, B., Zhang, K. (eds.): CPM 2007. LNCS, vol. 4580. Springer, Heidelberg (2007)
24. Ferragina, P., Landau, G.M. (eds.): CPM 2008. LNCS, vol. 5029. Springer, Heidelberg (2008)

Quasi-distinct Parsing and Optimal Compression Methods

Amihood Amir[1,2,⋆], Yonatan Aumann[1], Avivit Levy[3,4], and Yuri Roshko[3]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
{amir,aumann}@cs.biu.ac.il
[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218
[3] Shenkar College, Anna Frank 12, Ramat Gan 52526, Israel
avivitlevy@shenkar.ac.il
[4] CRI, Haifa University, Mount Carmel, Haifa 31905, Israel

Abstract. In this paper, the optimality proof of Lempel-Ziv coding is re-studied, and a much more general compression optimality theorem is derived. In particular, the property of *quasi-distinct parsing* is defined. This property is much weaker than *distinct parsing* required in the original proof, yet we show that the theorem holds with this weaker property as well. This provides a better understanding of the optimality proof of Lempel-Ziv coding, together with a new tool for proving optimality of other compression schemes. To demonstrate the possible use of this generalization, a new coding method – the *APT coding* – is presented. This new coding method is based on a principle that is very different from Lempel-Ziv's coding. Moreover, it does not directly define any parsing technique. Nevertheless, APT coding is analyzed in this paper and using the generalized theorem shown to be asymptotically optimal up to a constant factor, if APT quasi-distinctness hypothesis holds. An empirical evidence that this hypothesis holds is also given.

1 Introduction

Theoretical analysis of compression schemes is a fundamental task, albeit not an easy one. Even compression schemes that are efficient in practice are not always fully understood from the theoretical point of view. Sometimes, the search is for an appropriate measure of the compression efficiency/optimality (e.g. [4,10]). This task seems to incorporate knowledge of both the well experienced and studied properties of the specific compression scheme, and the behavior of the theoretical compression efficiency measure.

A common measure of compression optimality is *asymptotical optimality*. A well-known compression scheme achieving asymptotical optimality regarding the Shannon Entropy is the Lempel-Ziv coding [15,16]. The parsing algorithm for this encoding was first introduced by Lempel and Ziv in 1976 [8] and was proved to achieve the entropy rate by Ziv [14]. A number of different variations of the

⋆ Partly supported by ISF grant 35/05.

basic Lempel-Ziv algorithm are described in [3]. The algorithm considered in this paper is known as *LZ78* and was first described in [16]. A more transparent proof for this algorithm was provided by Wyner and Ziv in 1989 [12]. For the more powerful 1977 version of the Lempel-Ziv scheme [15], known as *LZ77*, a later proof of optimality was presented in 1994 by Wyner and Ziv [13].

In this paper, the LZ78 optimality proof of [12] is re-studied, and a more general compression-optimality theorem is derived. The advantages of such a result are two-fold:

1. Providing a generalized theorem that can serve to prove optimality of other compression schemes.
2. Better understanding the conditions that allow asymptotic optimality. Such an improved understanding can help develop other optimal compression schemes with special behavioral properties pertaining to specific applications.

Indeed, several approaches to generalize the parsing scheme of LZ78 were suggested over the years. Louchard and Szpankowski [9] study a generalization of the LZ parsing scheme with respect to the growth of the number of phrases, which is precisely the notion of *asymptotically distinct* parsing discussed in the full version of this paper. A different approach is grammar-based codes. The first significant contribution in this approach was the practical algorithm *SEQUITUR* [11]. This algorithm had excellent compression performance relative to other dictionary-based schemes. The first important theoretical contribution was in [7], which presented a class of grammar-based codes that includes LZ78 and is asymptotically optimal.

Our generalization take the approach of Louchard and Szpankowski, but give a much richer generalization. To demonstrate the possible use of our generalization, a new coding method – the *APT coding* – is presented. This coding method is based on a principle that is very different from the LZ-family codes. Moreover, it does not directly define any parsing technique. Nevertheless, the APT coding is analyzed and using our generalized theorem shown to be asymptotically optimal up to a constant factor, if APT quasi-distinctness hypothesis holds. An empirical evidence that this hypothesis holds is also given.

The APT algorithm was first introduced by [2] as a tool for convolutions of strings which avoids the use of FFT. [2] present a preliminary comparison between the number of phrases in LZ78 and the number of nodes in the APT on randomly built binary strings. Their comparison shows an advantage to APT. This preliminary result motivates a further study of the potential of APT as a compression algorithm. Our theoretical analysis further demonstrates this potential, though a more strict theoretical analysis as well as practical tests should be done on the APT coding in order to determine if, when and how it could be used as a compression scheme.

The rest of the paper is organized as follows. In Sect. 2 we give the basic definitions and the statement of the generalized theorem. In Sect. 3 we give its proof. Finally, in Sect. 4, we present the APT code and then use our generalized theorem to analyze it. Omitted proofs will appear in the full version of the paper.

2 Distinct and Quasi-distinct Parsings

2.1 Preliminaries

The Shannon Entropy Measure. Assuming the source is generated by a random process represented by a random variable X with mass function $p(x)$, the entropy of the variable X is defined by[1]

$$H(X) = -\sum p(x) \log_2 p(x).$$

The Asymptotic Equipartition Property (AEP). This property is the information theory analog of the well known law of large numbers. It is formalized in the following theorem:

Theorem 1. (AEP) [5] *If $X_1, X_2, \ldots$ are independent, identically distributed random variables drawn according to probability mass function $p(x)$, then*[2]

$$-\frac{1}{n} \log p(X_1, X_2, \ldots, X_n) \to H(X).$$

Entropy Rate of Stochastic Processes. The entropy rate is used to measure information of a series of variables that are not independent, but rather form a stochastic process.

Definition 1. [5] *The* entropy rate *of a stochastic process $\{X_i\}$ is defined by*

$$H(\chi) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n)$$

when the limit exists.

Ergodic Sources. An ergodic process is the most general dependent source for which the strong law of large numbers holds. We do not give a precise definition of an ergodic source, to avoid unnecessary technical details from probability theory. The important fact is that the AEP theorem holds for stationary ergodic processes and Markov approximations (see [5]).

Without loss of generality, we will assume that the source alphabet, χ, is binary. Thus, $\chi = \{0, 1\}$, throughout this paper.

2.2 Distinct Parsing and Optimal Compression

Parsing methods are directly connected to creation of a vocabulary of recurrent sub-words within a string. Therefore, understanding parsing techniques better may improve compression schemes or suggest others.

[1] We use logarithms to base 2. The entropy is then measured in bits.
[2] The limit can be with probability not necessarily 1.

Definition 2. [5] *A* parsing S *of a binary string* $x_1x_2\dots x_n$ *is a division of the string into phrases, separated by commas. A* distinct parsing *is such that no two phrases are identical.*

Example. Consider the following parsing method. Given a string X, for every $1 \le i$ define the i-th phrase to be the next i bits of X. Since all phrases have different length, they are obviously all distinct. Therefore, this parsing method is a distinct parsing.

The well-known LZ78 compression scheme[3] defines a distinct parsing of the source sequence. Let $c(n)$ denote the number of phrases in a parsing of a sequence of length n. Of course, $c(n)$ depends on the specific values of the variables sequence $X_1, X_2, \dots, X_n$ generated by the source. The compressed sequence in LZ78 scheme consists of a list of $c(n)$ pairs of numbers, each pair consisting of a pointer to the previous occurrence of the prefix of the phrase and the last bit of the phrase. Each pointer requires $\log c(n)$ bits, and hence the total length of the compressed sequence is $c(n)(\log c(n) + 1)$ bits. The asymptotic optimality of LZ78 coding is an immediate corollary of Theorem 2, which unites Theorem 12.10.1 and Theorem 12.10.2 given in [5].

Theorem 2. *Let* $\{X_i\}_{-\infty}^{\infty}$ *be a stationary ergodic stochastic process. Let* S *be a distinct parsing of the string* $X_1, X_2, \dots, X_n$*. Let* $l(X_1, X_2, \dots, X_n)$ *be the codeword length associated with* $X_1, X_2, \dots, X_n$ *defined on the phrases created by* S*, such that* $l(X_1, X_2, \dots, X_n) = c(n)(\log c(n) + 1)$*, where* $c(n)$ *is the number of phrases in the parsing. Then*

$$\limsup_{n\to\infty} \frac{l(X_1, X_2, \dots, X_n)}{n} \le H(\chi)$$

with probability 1, where $H(\chi)$ *is the entropy rate of the process.*

Remark. Theorem 2 specifies two conditions for a code to be optimal. The first condition focuses on the distinct parsing method producing the phrases to be coded. The second condition is a bound on the total length of the coded (distinct) phrases produced by the parsing. Note that the parsing does not specify how to efficiently code the produced phrases. Consider, for example, the following trivial parsing: the sequence $x_1x_2\dots x_n$ is parsed into one phrase $p_1 = x_1x_2\dots x_n$. Clearly, this is a distinct parsing, however, this does not define any scheme for generating the codeword length. The trivial way of taking the original string gives $l(x_1\dots x_n) = n$, which, obviously, does not satisfy Theorem 2. Therefore, this trivial parsing and coding does not define any compression scheme.

[3] Following [5], modifications and implementation details of this basic scheme are disregarded. These do not affect the asymptotic efficiency of the algorithm as considered in this paper.

2.3 Quasi-distinct Parsing

In this paper, a much weaker property is defined and surprisingly proven to be equivalent, regarding the optimality of a compression method that uses it, to the strong property of distinct parsing.

Definition 3. *Let $p_1, \ldots, p_\ell$ be a parsing of a binary string $x_1x_2\ldots x_n$. Let $D = \{p_i | 1 \leq i \leq \ell\}$, i.e. the set of all distinct phrases in the parsing. A* quasi-distinct *parsing is a parsing where $\ell - |D| = o(n/\log n)$.*

Example. Consider the following parsing method. Given a string X, for every $1 \leq i$ define the next i phrases by i times taking the next i bits of X. For example, if $X = 1111111\ldots$, then this parsing method gives the phrases: $1, 11, 11, 111, 111, 111, 1111, 1111, 1111, 1111, \ldots$ This is clearly not a distinct parsing, since the list of phrases contains repetitions of the same phrase. Moreover, let E be the multi-set that contains all the phrases that are not in D, then, the size of E grows to infinity as the length of X grows. However, this parsing method is a quasi-distinct parsing method, since $\sum(i-1)i < \sum i^2 \leq n$, and therefore, $|E| = O(n^{2/3}) = o(n/\log n)$, where n is the length of X.

The following much stronger theorem is proved in this paper.

Theorem 3. *Let $\{X_i\}_{-\infty}^{\infty}$ be a stationary ergodic stochastic process. Let S be a quasi-distinct parsing of the string $X_1, X_2, \ldots, X_n$. Let $l(X_1, X_2, \ldots, X_n)$ be the codeword length associated with $X_1, X_2, \ldots, X_n$ defined on the phrases created by S, such that $l(X_1, X_2, \ldots, X_n) = c(n)(\log c(n) + \alpha)$, where $c(n)$ is the number of phrases in the parsing and α is any positive constant. Then*

$$\limsup_{n\to\infty} \frac{l(X_1, X_2, \ldots, X_n)}{n} \leq H(\chi)$$

with probability 1, where $H(\chi)$ is the entropy rate of the process.

Since the quasi-distinct parsing property is much weaker compared to the distinct parsing property, this result is surprising and leads to a better understanding of the asymptotic optimality proof. Note that the condition on the codeword length is slightly generalized in Theorem 3 relative to Theorem 2. However, as can be deduced from the proof of Theorem 3 we present, Theorem 2 can be also restated for distinct parsing using this generalized condition. Therefore, Theorem 3 can be interpreted as showing an equivalence regarding the optimality of compression methods between the strong property of distinct parsing and the much weaker property of quasi-distinct parsing. The LZ78 compression scheme uses a distinct parsing method together with a "self-reference" method to efficiently store the vocabulary (phrases) produced by the parsing. The distinct parsing property, which does not allow repetitions of phrases, is crucial in proving that the produced vocabulary is compact, i.e., its size can be efficiently bounded. This property is also used to bound the probabilities of the phrases produced by the parsing. Yet, we show that the distinct parsing condition in the optimality proof can be relaxed to the much weaker condition of quasi-distinct parsing, which allow many (even growing to infinity) repetitions of phrases.

3 Equivalence of Quasi-distinct Parsing to Distinct Parsing

In this section we show that a quasi-distinct parsing method has the same relevant properties that a distinct parsing method has. These properties are used in the proof of Theorem 2 (see [5]). Moreover, the distinctness condition is only used in proving these properties. The first property is a bound on the number of phrases possible in a quasi-distinct of a binary sequence of length n, and is an immediate corollary of the quasi-distinctness definition and the following lemma.

Lemma 1. [Lempel and Ziv] *The number of phrases in a distinct parsing of a binary sequence $X_1, X_2, \ldots, X_n$ satisfies*

$$c(n) \leq \frac{n}{(1-\epsilon_n)\log n}$$

where $\epsilon_n \to 0$ as $n \to \infty$.

Corollary 1. *The number of phrases in a quasi-distinct parsing of a binary sequence $X_1, X_2, \ldots, X_n$ satisfies*

$$c(n) \leq \frac{n}{(1-\epsilon_n)\log n} + o(n/\log n)$$

where $\epsilon_n \to 0$ as $n \to \infty$.

The next very important property is the quasi-distinct version of Ziv's inequality (see [5,14]). Let $\{X_i\}_{i=-\infty}^{\infty}$ be a stationary ergodic process with probability mass function $P(x_1, x_2, \ldots, x_n)$. For a fixed integer k, define the kth order Markov approximation to P as

$$Q_k(x_{-(k-1)}, \ldots, x_0, x_1, \ldots, x_n) \triangleq P(x^0_{-(k-1)}) \prod_{j=1}^{n} P(x_j | x^{j-1}_{j-k})$$

where $x_i^j \triangleq (x_i, x_{i+1}, \ldots, x_j)$, $i \leq j$, and the initial state $x^0_{-(k-1)}$ will be a part of the specification of Q_k. Since $P(X_n|X^{n-1}_{n-k})$ is itself an ergodic process, we have $\frac{1}{n}\log Q_k(X_1, X_2, \ldots, X_n|X^0_{-(k-1)}) = \frac{1}{n}\sum_{j=1}^{n}\log P(X_j|X^{j-1}_{j-k}) \to -E\log P(X_j|X^{j-1}_{j-k}) = H(X_j|X^{j-1}_{j-k})$.

We will bound the rate of a code by the entropy rate of the kth order Markov approximation for all k. The entropy rate of the Markov approximation $H(X_j|X^{j-1}_{j-k})$ converges to the entropy rate of the process as $k \to \infty$ and this will prove the result.

Suppose $X^n_{-(k-1)} = x^n_{-(k-1)}$, and suppose that x_1^n is parsed into c quasi-distinct phrases, $y_1, y_2, \ldots, y_c$. Let v_i be the index of the start of the i-th phrase, i.e., $y_i = x_{v_i}^{v_{i+1}-1}$. For each $i = 1, 2, \ldots, c$, define $s_i = x^{v_i-1}_{v_i-k}$. Thus s_i is the k bits preceding y_i. Of course, $s_1 = x^0_{-(k-1)}$.

Let c_{ls} be the number of phrases with length l and preceding state $s_i = s$ for $l = 1, 2, \ldots$ and $s \in \chi^k$. We then have

$$\sum_{l,s} c_{ls} = c \tag{1}$$

and

$$\sum_{l,s} l c_{ls} = n \tag{2}$$

We now prove the following upper bound on the probability of a string based on the parsing of the string.

Lemma 2. *For any quasi-distinct parsing of the string $x_1, x_2, \ldots, x_n$, we have*

$$\log Q_k(x_1, x_2, \ldots, x_n | s_1) \leq -\sum_{l,s} c_{ls} \log c_{ls} + o(n).$$

Proof. As in Ziv's inequality proof [14,5], we begin by writing

$$Q_k(x_1, x_2, \ldots, x_n | s_1) = Q_k(y_1, y_2, \ldots, y_c | s_1) = \prod_{i=1}^{c} P(y_i | s_i),$$

or

$$\begin{aligned}\log Q_k(x_1, x_2, \ldots, x_n | s_1) &= \sum_{i=1}^{c} \log P(y_i | s_i) \\ &= \sum_{l,s} \sum_{i:|y_i|=l, s_i=s} \log P(y_i | s_i) \\ &= \sum_{l,s} c_{ls} \sum_{i:|y_i|=l, s_i=s} \frac{1}{c_{ls}} \log P(y_i | s_i)\end{aligned}$$

$$\leq \sum_{l,s} c_{ls} \log\Big(\sum_{i:|y_i|=l, s_i=s} \frac{1}{c_{ls}} P(y_i | s_i)\Big),$$

where the inequality follows from Jensen's inequality and the concavity of the logarithm.

Now, we split the phrases into two (multi-)sets: the first is the maximal set of distinct phrases, and the second contains the rest of the phrases. For the first set, since we know the y_i are distinct, we have $\sum_{i:|y_i|=l, s_i=s} P(y_i|s_i) \leq 1$. The size of the second set, by the definition of quasi-distinct parsing, is $o(n/\log n)$. Thus,

$$\log Q_k(x_1, x_2, \ldots, x_n | s_1) \leq \sum_{l,s} c_{ls} \log \frac{1}{c_{ls}} + o(\frac{n}{\log n} \log \frac{n}{\log n})$$

or

$$\log Q_k(x_1, x_2, \ldots, x_n | s_1) \leq \sum_{l,s} c_{ls} \log \frac{1}{c_{ls}} + o(n).$$

Since by (1), $\sum_{l,s} c_{ls} = c$, and by Corollary 1 we know that $c \leq \frac{n}{\log n}(1+o(1)) + o(\frac{n}{\log n})$, we get the lemma.

Using the above properties, a sufficient condition for a quasi-distinct parsing to be an optimal compression method can be proved exactly as in the original proof for distinct parsing. This gives Theorem 3.

4 Application: Analysis of APT Coding Method

In this section we present a new lossless coding method – the *arithmetic progressions tree (APT)*. We then use Theorem 3 to analyze the possible use of APT coding as a compression method. Following the framework of Theorem 3 the analysis of APT coding is done in two stages. First, we show a parsing that the APT coding defines and refer to its quasi-distinctness. Then, we analyze the APT code length.

4.1 The Arithmetic Progressions Tree

The APT coding method is based on finding arithmetic progressions in the given binary string. An arithmetic progression can then be expressed by three values: the index of its start point in the string, the difference between the elements in the progression and its length. By grouping progressions with the same difference and length parameters and applying the search recursively we get a coding of the original string. Given the APT, the original string can be reconstructed by following the paths from each leaf, which defines a starting index for a progression. The path defines the (recursive) structure of this progression. The description of APT construction follows.

APT Construction Algorithm. There can be many possible progressions to be chosen, and the choices may overlap. We are interested in a cover by disjoint arithmetic progressions of the ones[4] in the binary string, where by disjoint we mean that an element participating in one progression cannot participate in another. Our algorithm for choosing the progressions uses a very simple greedy criterion: find the least difference between ones in the string, then choose progressions of this difference (with possibly different length) first. All ones participating in a chosen progression are turned to zero, and the process of finding the least difference in the string continues until the string is all zeroes. The collection of progressions with the same difference is partitioned into sub-collections according to their length. All progressions in a sub-collection share the difference and length parameters, and differ in their start positions. Representation of the start positions in a sub-collection is done by recursively applying the algorithm until there is only one starting point to represent, which then becomes a leaf. A detailed description of the algorithm is given in Fig. 1.

[4] Zeroes can be taken instead if they are less frequent. For simplicity we assume that only indices of ones are taken.

```
APT Construction Algorithm
Input: $L$ a list of 1's indices in $X \in \{0,1\}^n$, $root$ the APT root
1  while $L$ is not empty do
2      if $L$ contains a single index $i$ then
3          allocate new $leaf$
4          $leaf.index \leftarrow i$
5          $leaf.parent \leftarrow root$
6      else
7          find, $d$, the least difference between consecutive elements in $L$
8          remove from $L$ the list $C_d$ of all progressions with difference $d$
9          split progressions in $C_d$ by length $l$ into sub-lists $C_{d,l}$
10         for each sub-list $C_{d,l}$ do
11             allocate new $node$
12             $node.difference \leftarrow d$
13             $node.length \leftarrow l$
14             $node.parent \leftarrow root$
15             run APT algorithm on $C_{d,l}$ and $node$
Output: $root$
```

Fig. 1. A recursive version of the algorithm for constructing arithmetic progressions tree

Example. Consider the string $X = 01101100101011$. APT algorithm constructs the tree with a root and two children nodes. The root does not contain any internal information. The first (it is constructed first, but it does not matter) child of the root contain the information $difference = 1$, $length = 2$, and has two children; a leaf containing $index = 13$ and an internal node with information $difference = 3$, $length = 2$, which has a leaf containing $index = 2$. The second child of the root contains the information $difference = 2$, $length = 2$, and has a leaf containing $index = 9$. Note that X can be fully reconstructed given the APT, by following the three paths from the three leaves to the root and reconstructing the (possibly recursive) arithmetic progressions they define.

Properties of APT. The following definition and lemmas are key properties of the APT structure.

Definition 4. *A node in the APT tree which is not a leaf is called an* internal *node. An internal node with degree at least 2 is called* branching*, denoted B. Non-branching nodes are denoted NB. A root of a path of length at least 2 of NB nodes is called* non-branching head*, denoted NBH.*

Lemma 3. *In a binary tree, if there are k NBH nodes, then there must be $k-1$ B nodes.*

Lemma 4. *If the APT of a string $X \in \{0,1\}^n$ has k nodes then X contains at least k zeroes.*

4.2 APT Quasi-distinctness

The Parsing Definition. The first step is to show how the APT defines a parsing of the original string. Note that each APT leaf provides a starting index of an arithmetic progression (possibly recursive) in the original string. We call this index a *start point.* The sorted list of start points indices naturally define the following parsing: The first phrase starts at the beginning of the string, and each phrase starts with the next start point and ends with the last bit before the following start point. If there are zeroes after the rightmost bit, they define the last phrase. We call this parsing *the APT parsing.*

Bounding Repetitions in the APT Parsing. We do not have a combinatorial proof that the APT parsing is indeed a quasi-distinct parsing. Nevertheless, an empirical evidence for that is presented.

Testing the APT Parsing Quasi-Distinctness. The tests were done on three data types:

1. A string generated by a pre-specified recursion relation. This type of string represents the case of highly compressible strings (at least in the Kolmogorov-Chaitin sense). The specific sequence was chosen to be Thue-Morse sequence, defined as follows: $x_0 = 0$, $x_{n+1} = x_n(x_n)^c$, where the c operator is the binary complement. For example, the first bits of this sequence are: 0010110011010010110100110010110 . . ..
2. A string generated by *srand*() function in C++-shell, which gets as an input the system time, (unsigned) time(0). This string represents a data built sequentially at random[5].
3. A string representing common texts. For this type the file *enwiki*8, which is the first 10^8 pages of *Wikipedia*, was taken. This file was downloaded from [1]. Since the current version of APT algorithm is built for binary strings the binary representation of this text is considered.

We run the APT algorithm for each of this strings for sizes 1,000,000 bits up to 205,000,000 bits by jumps of 1,000,000 bits. In each run the number of repetitions in the APT parsing were counted. The results are presented in Table 1 and Fig. 2. Table 1 shows that the APT size (in nodes not in bits) is negligible compared to the input size n and also shows the relation between internal nodes and leaves in the APT. Fig. 2 clearly shows that the number of repetitions in all these tests is $o(n/\log n)$. APT parsing quasi-distinctness is therefore referred to as the following hypothesis.

The APT Parsing Quasi-Distinctness Hypothesis. The number of repetition of phrases in the APT parsing is $o(n/\log n)$.

[5] This is surely not a random string, because APT coding can compress it and random strings cannot be compressed. Nevertheless, it passed simple tests for randomness. We, therefore, call it random.

Table 1. The APT size for different data types. All numbers were multiplied by 10^{-6}.

input size	enwiki8: inter. nodes	enwiki8: leaves	enwiki8: repet.	Thue-Morse: inter. nodes	Thue-Morse: leaves	Thue-Morse: repet.	random: inter. nodes	random: leaves	random: repet.
1	0.01	0.04	0.02	1.695E-3	1.691E-3	1.636E-3	8.235E-3	8.157E-3	2.903E-3
20	0.09	0.68	0.44	9.851E-3	9.827E-3	9.745E-3	8.235E-3	8.151E-3	2.900E-3
40	0.16	1.32	0.88	13.195E-3	17.468E-3	17.376E-3	8.235E-3	8.145E-3	2.906E-3
60	0.23	1.93	1.31	19.250E-3	20.953E-3	20.863E-3	8.235E-3	8.128E-3	2.891E-3
80	0.26	2.23	1.53	22.140E-3	22.060E-3	21.971E-3	8.235E-3	8.174E-3	2.911E-3
100	0.36	3.13	2.17	22.140E-3	29.950E-3	29.850E-3	8.235E-3	8.064E-3	2.890E-3
120	0.42	3.71	2.60	23.732E-3	33.037E-3	32.944E-3	8.235E-3	8.179E-3	2.909E-3
140	0.48	4.30	3.03	24.131E-3	37.095E-3	36.999E-3	8.235E-3	8.113E-3	2.910E-3
160	0.54	4.88	3.45	29.641E-3	39.249E-3	39.152E-3	8.235E-3	8.173E-3	2.900E-3
180	0.60	5.45	3.87	32.426E-3	43.609E-3	43.506E-3	8.235E-3	8.143E-3	2.907E-3
200	0.66	6.03	4.30	41.218E-3	37.356E-3	37.254E-3	8.235E-3	8.072E-3	2.889E-3

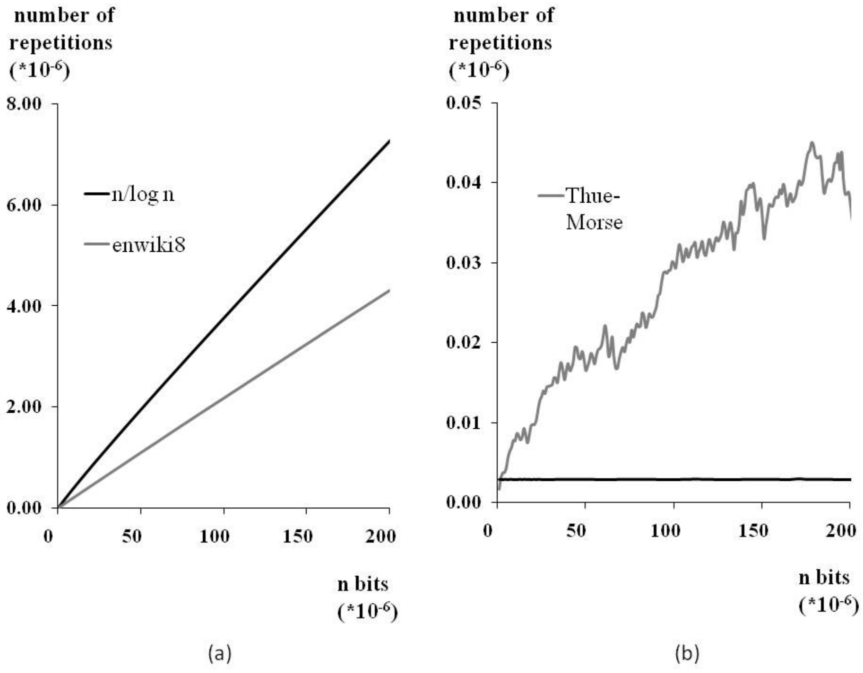


Fig. 2. (a) The number of repetitions of phrases in APT parsing of enwiki8 compared to $n/\log n$. (b) The number of repetitions of phrases in APT parsing of Thue-Morse sequence and random sequence.

Applying to All APT Nodes. Note that the APT parsing does not refer to APT internal nodes. We use the analysis of the APT structure to get bounds on all APT nodes.

Lemma 5. *Let $c(n)$ be the number of APT leaves for a string X. Under the APT parsing quasi-distinctness hypothesis, the number of APT internal nodes is at most $d \cdot c(n) + o(n/\log n)$, where $d > 0$ is a constant.*

Proof. Given a fixed number of leaves, the number of branching internal nodes in a tree is maximized in a binary tree. Also, in a binary tree it is known that the number of B nodes is at most $2c(n) - 1$, where $c(n)$ is the number of leaves. Therefore, by Lemma 3 the number of NBH nodes is at most $2c(n)$. We now show that all other NB nodes are $o(n/\log n)$ if the quasi-distinct property (q.d.p. for short) holds on the APT parsing. If q.d.p. holds then by the APT parsing definition and Corollary 1, we have $c(n) = O(n/\log n)$. Therefore, the number of NBH nodes is also $O(n/\log n)$. Assume to the contrary that there are also $\Omega(d(n) \cdot n/\log n)$ NB nodes, where $d(n)$ is any function of n growing to infinity. The NB nodes must all be on paths of NB nodes starting with NBH nodes. Therefore, the number of such paths is bounded by the number of NBH nodes, which is $O(n/\log n)$. Thus, the average path length is $\Omega(d(n))$. Since the number of ones reduces by at least half in each level of the APT, a path of length $\Omega(d(n))$ indicates that X has at least $2^{d(n)}$ one bits. Therefore, the total number of one bits in X is $\Omega(n/\log n \cdot 2^{d(n)})$. However, since the APT has in this case $\Omega(d(n) \cdot n/\log n)$ nodes, by Lemma 4, X has at least $d(n) \cdot n/\log n$ zeroes. We get: $2^{d(n)} \cdot n/\log n + d(n) \cdot n/\log n = n$, which has no solution for $d(n)$, contradiction.

4.3 The APT Code Length

We now deal with the representation of the information in the APT. First, note that the tree hierarchy can be stored efficiently with only 2 bits per tree node by using parenthesis notation [6]. Thus, we can compute the length of the representation of the information within the APT nodes and then add 2 bits per node for the tree hierarchy. The information in the APT nodes is: two values for an internal node (the difference and length), and one value for a leaf node (the starting index of a progression).

We now explain how to use a "reference" (but not exactly "self-reference") method to represent these values. Let $Y \in \{0,1\}^n$ be a string with all zeroes except the indices with values that appear in the nodes of the APT of the original string $X \in \{0,1\}^n$. We call Y the *APT reference string of* X. By Lemma 5 if $c(n)$ is the number of leaves in the APT of X, then Y contains $O(c(n))$ one bits. Now, each value in an APT node can be represented as a reference to the sequential order of the appropriate one bit in Y, i.e., the first one bit, the second one bit, etc. To represent these reference values only $\log c(n) + \alpha$ bits are needed, where α is a constant. The key property is the following:

Lemma 6. *Let $\{X_i\}_{-\infty}^{\infty}$ be a stationary ergodic stochastic process. Let X be the string $X_1, X_2, \ldots, X_n$. Then, $H(Y) \leq H(X)$, where Y is the APT reference string of X.*

Proof. We use the chain rule for the mutual information $H(X,Y)$. By the chain rule: $H(X,Y) = H(X) + H(Y|X)$ and $H(X,Y) = H(Y) + H(X|Y)$. Thus, $H(Y) \leq H(Y) + H(X|Y) = H(X,Y) = H(X) + H(Y|X) = H(X)$, since Y is completely determined when X is given, therefore, $H(Y|X) = 0$.

We can now prove that APT coding is an almost asymptotically optimal compression scheme.

Theorem 4. *If the APT parsing quasi-distinctness hypothesis holds then APT coding is asymptotically optimal up to a constant factor.*

Proof. Let $\{X_i\}_{-\infty}^{\infty}$ be a stationary ergodic stochastic process. Let X be the string $X_1, X_2, \ldots, X_n$. Let $l(X_1, X_2, \ldots, X_n)$ be the APT code length. Let $c(n)$ denote the number of phrases in the APT parsing of X. Then, by the discussion above, $l(X_1, X_2, \ldots, X_n) = \beta c(n)(\log c(n) + \alpha) + l'(Y_1, Y_2, \ldots, Y_n)$, where α, β are constants and $Y = Y_1, Y_2, \ldots, Y_n$ is the APT reference string. If the APT q.d.p. hypothesis holds then by Theorem 3, we have $\limsup \frac{1}{n} c(n)(\log c(n) + \alpha) \leq H(\chi)$. By using any asymptotically optimal compression method (such as LZ78) to represent Y, we get: $\limsup \frac{1}{n} l'(Y_1, Y_2, \ldots, Y_n) \leq H(\mathcal{Y})$ (where, $H(\mathcal{Y})$ is the entropy rate of Y), which by Lemma 6 is at most $H(\chi)$. The theorem then follows.

5 Conclusions

In this paper, two conditions together are proven to be sufficient for a code to define an optimal compression:

– Having a vocabulary produced by a quasi-distinct parsing method.
– The total length of the coded phrases in the vocabulary is bounded by $c(n)(\log c(n) + \alpha)$, where $c(n)$ is the number of phrases produced by the parsing and α is any positive constant.

These conditions are more general than the conditions met by the Lempel-Ziv code. The strength of this generalization is demonstrated by analyzing the new APT coding method. We show that this generalization is capable of handling even the APT code that is very different from the LZ-scheme, and does not even directly define a parsing. Indeed, we 'lost' constants in this analysis, and were only able to prove asymptotical optimality up to a constant factor (if APT quasi-distinctness hypothesis holds), which is a weaker notion of optimality. Nevertheless, this analysis demonstrates the flexibility that our generalization gives to the analysis of compression schemes. It also contributed to understanding the APT coding, giving a theoretical explanation for its promising behavior in the preliminary tests of its vocabulary size. A called for challenge is, therefore, to design an APT compressor and test its practical performance against existing schemes.

References

1. http://www.cs.fit.edu/~mmahoney/compression/textdata.html
2. Amir, A., Levy, A., Reuveni, L.: The practical efficiency of convolutions in pattern matching algorithms. Fundamenta Informaticae 84(1), 1–15 (2008)
3. Bell, T.C., Cleary, J.G., Witten, I.H.: Text compression. Prentice-Hall, Englewood Cliffs (1990)
4. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm., Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, Calif. (1994)
5. Cover, T.M., Thomas, J.A.: Elements of information theory. Wiley Interscience, Hoboken (1991)
6. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
7. Kieffer, J.C., Yang, E.-H.: Grammar based codes: a new class of universal lossless source codes. IEEE Transactions on Information Theory IT-46(3), 737–754 (2000)
8. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Transactions on Information Theory IT-22, 75–81 (1976)
9. Louchard, G., Szpankowski, W.: Generalized lempel-ziv parsing scheme and its preliminary analysis of the average profile. In: Data Compression Conference (DCC), pp. 262–271 (1995)
10. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM (JACM) 48(3), 407–430 (2001)
11. Nevill-Manning, C., Witten, I., Maulsby, D.: Compression by induction of hierarchical grammars. In: Proceedings of Data Compression Conference (DCC), pp. 244–253 (1994)
12. Wyner, A.D., Ziv, J.: Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression. IEEE Transactions on Information Theory IT-35(6), 1250–1258 (1989)
13. Wyner, A.D., Ziv, J.: The sliding-window lempel-ziv algorithm is asymptotically optimal. Proceedings of the IEEE 82, 872–877 (1994)
14. Ziv, J.: Coding theorems for individual sequences. IEEE Transactions on Information Theory IT-24(4), 405–412 (1978)
15. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–343 (1977)
16. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory IT-24(5), 530–536 (1978)

Generalized Substring Compression

Orgad Keller, Tsvi Kopelowitz, Shir Landau, and Moshe Lewenstein

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
{kellero,kopelot,landaus2,moshe}@cs.biu.ac.il

Abstract. In *substring compression* one is given a text to preprocess so that, upon request, a compressed substring is returned. *Generalized substring compression* is the same with the following twist. The queries contain an additional context substring (or a collection of context substrings) and the answers are the substring in compressed format, where the context substring is used to make the compression more efficient.

We focus our attention on *generalized substring compression* and present the first non-trivial *correct* algorithm for this problem. In our algorithm we inherently propose a method for finding the *bounded longest common prefix* of substrings, which may be of independent interest. In addition, we propose an efficient algorithm for substring compression which makes use of *range searching for minimum* queries.

We present several tradeoffs for both problems. For compressing the substring $S[i\,..\,j]$ (possibly with the substring $S[\alpha\,..\,\beta]$ as a context), best query times we achieve are $O(C)$ and $O\big(C \log\big(\frac{j-i}{C}\big)\big)$ for substring compression query and generalized substring compression query, respectively, where C is the number of phrases encoded.

1 Introduction

While the topic of string compression has been a viable research topic for decades, few works have been done concerning the problem of substring compression. The topic was introduced in [4], where a set of problems concerning substring compression focusing on the compression algorithm of Lempel and Ziv [17] was presented. They deal mainly with two variants of this topic, namely, given a string, what is the compressibility of different substrings of that string, both in the sense of the actual compression of the substrings and in the sense of comparing which of the substrings is the least or most compressible.

We address the following problems: in the *substring compression query* (SCQ) problem, we wish to compress a given substring of the string S, denoted by start and end location. Note that we preprocess S so that we are able to answer this query for any substring in S in an online manner. In its generalized and more powerful version, the *generalized substring compression query* (GSCQ) problem, we wish to compress the substring according to a given context taken from S as well. In both problems, our goal is to provide query times which are proportional to the size of the *compressed* substring as opposed to the size of the substring in its non-compressed form.

The issue of substring compression has interesting implications on a variety of practical applications. Recent works use compression of biological sequences as a basis of comparison between different sequences, and their information content. Compression of sub-sequences can therefore be used to perform such comparisons in a more efficient and accurate manner. Various other applications arise in the context of substring compression, such as data storage and extraction, and data transfer in a network setting.

1.1 Our Results

1. Our main result is providing an efficient and innovative algorithm for the *generalized substring compression query*, introduced in [4]. There an algorithm was suggested. However, this algorithm is incorrect [14]: it overlooked the inherent added difficulty of the generalized problem, dismissing it as trivial, while it is in fact the essence of the generalized problem. Therefore, the solution provided in [4] in fact does not solve the problem. Our solution for this problem is based on a solution to finding the *bounded longest common prefix* (BLCP) of two substrings, which is a notion we will introduce shortly.
2. In addition, we improve results shown for the *substring compression query*. Our result is based mainly on an improved solution for finding the *interval longest common prefix* (ILCP) of two substrings. This is done using an efficient solution for the problem of *range searching for minimum* [11], and not on the more classical *range reporting* problem (see, for instance [1]), used by [4] and numerous other indexing-related papers [7,2,8,12]. This constitutes a totally different method in order to reduce the substring compression query problem to the geometric problem.

Our solutions are based on a variety of tools, such as suffix trees, lowest common ancestor queries, level ancestor queries, and several kinds of range searching structures. As a result, solutions to both SCQ and GSCQ constitute tradeoffs between query times, space, and preprocessing times, due to the choice of range searching structures to be used. A comparison of the results is presented in Table 1.

Table 1. Results

Prob.	Query Time	Space	Preprocessing Time	Source
GSCQ	$O\big(C_{\alpha,\beta}(i,j)\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\big)$	$O(n^{1+\epsilon})$	w.c. $O(n^{1+\epsilon})$	new
	$O\big(C_{\alpha,\beta}(i,j)\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\log\log n\big)$	$O(n\log n)$	exp. $O(n\log n\log\log n)$	new
	$O\big(C_{\alpha,\beta}(i,j)\big(\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\log\log n+\log n\big)\big)$	$O(n\log^{\epsilon} n)$	exp. $O(n\log n)$	new
	$O\big(C_{\alpha,\beta}(i,j)\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\log n\big)$	$O(n)$	w.c. $O(n\log n)$	new
SCQ	$O(C(i,j))$	$O(n^{1+\epsilon})$	w.c. $O(n^{1+\epsilon})$	new
	$O(C(i,j)\log\log n)$	$O(n\log n)$	exp. $O(n\log n\log\log n)$	new
	$O(C(i,j)\log n)$	$O(n)$	w.c. $O(n\log n)$	new
	$O(C(i,j)\log n\log\log n)$	$O(n\log^{\epsilon} n)$	exp. $O(n\log n)$	[4]

The rest of our paper is organized as follows: in Sect. 2, we give some preliminaries and problem definitions. In Sects. 3 and 4, we describe our solutions for finding the BLCP and ILCP accordingly. In Sect. 5, we present the outline of the query algorithm's main loop, which is roughly common to both the SCQ and GSCQ problems. In Sects. 6 and 7, we present the solutions and analysis for SCQ and GSCQ.

2 Problem Definitions and Preliminaries

2.1 Preliminary Definitions and Notations

Given a string S, $|S|$ is the length of S. Throughout this paper we denote $n = |S|$. An integer i is a *location* or a *position* in S if $i = 1, \ldots, |S|$. The substring $S[i \mathinner{..} j]$ of S, for any two positions $i \leq j$, is the substring of S that begins at index i and ends at index j. Concatenation is denoted by juxtaposition. The *suffix* S_i of S is the substring $S[i \mathinner{..} n]$.

The *suffix tree* [16,15,6,13] of a string S, denoted $\mathrm{ST}(S)$, is a compact trie of all the suffixes of $S\$$ (i.e., S concatenated with a delimiter symbol $\$ \notin \Sigma$, where Σ is the alphabet set). Each of its edges is labeled with a substring of S (actually, a representation of it, e.g., the start location and its length). The "compact" property is achieved by contracting nodes having a single child. The children of every node are sorted in the lexicographical order of the substrings on the edges leading to them. Consequently, each leaf of the suffix tree represents a suffix of S, and the leaves are sorted from left to right in the lexicographical order of the suffixes that they represent. $\mathrm{ST}(S)$ requires $O(n)$ space. Algorithms for the construction of a suffix tree enable $O(n)$ preprocessing time when $|\Sigma|$ is constant, and $O(n \log \min(n, |\Sigma|))$ time when $|\Sigma|$ is not. In fact, the suffix tree can be constructed in linear time even for alphabets drawn from a polynomially-sized range, see [6].

In addition, our algorithms make use of elements from the field of computational geometry; let $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a set of n points on an $[n] \times [n]$ grid. The following query types are defined on P, for various types of a two-dimensional range R:

rangemin$_y$($R = [x, x'] \times [y, \infty]$)**:** reports the single point of P that is included in the range and has a minimal y-coordinate, i.e., the point $\arg\min_{(x,y) \in P \cap R} y$.
rangemin$_x$($R = [x, \infty] \times [y, y']$)**:** reports the point $\arg\min_{(x,y) \in P \cap R} x$.
rangemax$_x$($R = [-\infty, x'] \times [y, y']$)**:** reports the point $\arg\max_{(x,y) \in P \cap R} x$.
emptiness($R = [x, x'] \times [y, y']$)**:** returns "true" iff $P \cap R = \emptyset$.

An Overview of the Lempel-Ziv Algorithm. The LZ77 variation of the Lempel-Ziv algorithm works as follows: given an input string S of length n, the algorithm encodes the string in a greedy manner from left to right. At each step of the algorithm, suppose we have already encoded $S[1 \mathinner{..} k-1]$, we search for the location t, such that $1 \leq t \leq k-1$, for which the longest common prefix of $S[k \mathinner{..} n]$ and the suffix S_t, is maximal. Once we have found the desired location, suppose the aforementioned longest common prefix is the substring $S[t\,..\,r]$, a phrase will be added to the output which will include the encoding of the distance to the substring (i.e., the value $k - t$) and the length of the substring, (i.e., the value $r - t + 1$). The algorithm continues by encoding $S[k + (r - t + 1)\,..\,n]$. Finally, we denote the output of the LZ77 algorithm on the input S as LZ(S). The size of LZ(S), denoted $|\text{LZ}(S)|$, is the length of LZ(S) in *bits*.

The string S may be encoded within the context of the string T. We denote this by LZ($S \mid T$). The practical meaning of this is that the result is as if the algorithm was performed on the concatenated string $T\$S$, where \$ is a symbol that does not appear in neither S nor T, however, only the portion of LZ($T\$S$) which represents the compression of S is outputted by the algorithm. Some exceptions apply to this rule. They will be described later.

2.2 Problem Definitions

Given a string S of length n, we wish to preprocess S in such a way that allows us to efficiently answer the following queries:

Substring Compression Query (SCQ(i, j)): given any two indices i and j, such that $1 \leq i \leq j \leq n$, we wish to output LZ($S[i\,..\,j]$).

Generalized Substring Compression Query (GSCQ(i, j, α, β)): given any four indices i, j, α, and β, such that $1 \leq i \leq j \leq n$ and $1 \leq \alpha \leq \beta \leq n$, we wish to output LZ($S[i\,..\,j] \mid S[\alpha\,..\,\beta]$).

Query times for both of the above query types will be strongly dependent on the number of phrases actually encoded. We denote these as $C(i, j)$ and $C_{\alpha,\beta}(i, j)$ for SCQ and GSCQ, respectively. Our results will rely on the two following primitives:

Bounded Longest Common Prefix (BLCP(k, l, r)): given k, and given positions l and r which induce the context substring $S[l\,..\,r]$, we look for the longest common prefix of $S[k\,..\,j]$ and a substring which starts at some location $l \leq t \leq r$ within the context. The substring chosen must not exceed the end of context. In other words, it must be a prefix of some substring $S[t\,..\,r]$.

Interval Longest Common Prefix (ILCP(k, l, r)): given k, l, r,this time we look for the longest common prefix of $S[k\,..\,j]$ and a substring which starts at some location $l \leq t \leq r$, without further constraints.

While it may not seem so at first glance, BLCP queries constitute several problematic implications, and therefore are much more difficult to implement, in comparison to ILCP. For example, consider two suffixes S_{t_1} and S_{t_2}, such that $l \leq t_1 < t_2 \leq r$, for which $|\text{LCP}(S_k, S_{t_1})| < |\text{LCP}(S_k, S_{t_2})|$ (where LCP(S_1, S_2), for two strings S_1 and S_2, stands for the *longest common prefix* of S_1 and S_2). Some portion of the last characters of LCP(S_k, S_{t_2}) may not be eligible for consideration. Namely, if $|\text{LCP}(S_k, S_{t_2})|$ exceeds $r - t_2 + 1$ characters, LCP(S_k, S_{t_2}) exceeds location r, and therefore literally "grows out of context". In that case, it may be that S_{t_1} will eventually be the suffix to be preferred. One should take into account that such a cut-off may pertain to LCP(S_k, S_{t_1}) as well. (Note: in the case $i-1 \leq r < j$, if desired, one can allow a substring taken from the context to exceed r. This is a trivial extension to the algorithm for ILCP.)

3 Answering BLCP Queries

3.1 Preprocessing Motivation

We begin the preprocessing by constructing the suffix tree of S, $\mathrm{ST}(S)$. In the suffix tree, each leaf ℓ is associated with a suffix of $S\$$ and is therefore marked with an integer $y(\ell)$ which is the start location of that suffix. We also mark each leaf ℓ with an integer $x(\ell)$ which is the lexicographical rank of the suffix associated with ℓ within the set of all suffixes of T (this is done by using one depth-first traversal, in which we number the leaves from left to right). We then preprocess the set $P = \{(x(\ell), y(\ell)) \mid \ell \text{ is a leaf in } \mathrm{ST}(S)\} \subseteq [n+1]^2$ for emptiness and rangemin_y queries.

Suppose we search $\mathrm{ST}(S)$ for some substring $S[l\,..\,r]$, and let v be the node in which the search ended. All the leaves in the subtree rooted at v, denoted T_v, correspond to occurrences of $S[l\,..\,r]$ in S. Hence the set $Y_v = \{y(\ell) \mid \ell \text{ is a leaf in } T_v\}$ is the set of all occurrence positions of $S[l\,..\,r]$ in S. From the properties of the suffix tree it follows that the set $X_v = \{x(\ell) \mid \ell \text{ is a leaf in } T_v\}$ forms a consecutive range of values in $[n+1]$. This is exactly the range $X_v = [x(l_v), x(r_v)]$, where l_v and r_v are the leftmost and rightmost leaves in T_v, respectively. It therefore holds that for a leaf ℓ, $x(\ell) \in [x(l_v), x(r_v)]$ iff $S[l\,..\,r]$ appears in S at location $y(\ell)$.

Notice that each node u in the suffix tree has two different notions of *depth*: the ordinary perception of depth of a node in a tree, denoted $\mathrm{depth}(u)$, and the length of the string u represents denoted $\mathrm{length}(u)$. Now let S_i and S_j be two suffixes of S, and consider the *longest common prefix* of S_i and S_j, denoted $\mathrm{LCP}(S_i, S_j)$. Let ℓ_i and ℓ_j be the leaves corresponding to S_i and S_j, respectively (i.e., $i = y(\ell_i)$ and $j = y(\ell_j)$). Then $|\mathrm{LCP}(S_i, S_j)| = \mathrm{length}(\mathrm{LCA}(\ell_i, \ell_j))$, where $\mathrm{LCA}(\ell_i, \ell_j)$ is the *lowest common ancestor* of ℓ_i and ℓ_j.

3.2 Definitions and Notations

We start with the following definition:

Definition 1. *A suffix S_t is said to be* relevant *to a range $[l, r]$ if $l \leq t \leq r$.*

Before showing how to implement BLCP queries, we define the notion of *eligibility*, which forces us to define the contribution given by a specific relevant suffix properly, and is depicted in the following definition:

Definition 2. *Given a location r and a node u of the suffix tree, a relevant suffix S_t is said to be* eligible *(w.r.t. r) at node u of the suffix tree if:*

1. *u is on the path from the root to ℓ_t (Equivalently, ℓ_t is a leaf in T_u)*
2. *$t + \mathrm{length}(u) - 1 \leq r$.*

In other words, we do not want to consider any portion of the suffix S_t which exceeds the position r, as such a portion is irrelevant to us. The notion of eligibility allows us to formalize the idea of a *bounded prefix*.

3.3 Implementing BLCP Queries

Consider the suffix S_k represented by the path from the root to ℓ_k. As suffixes S_t with greater $|\mathrm{LCP}(S_k, S_t)|$ values branch out of this path at a later stage (i.e., leave this path at nodes of greater depth), we are interested in suffixes which share a large portion of this path. *However*, we are restricted by the eligibility of those suffixes. Therefore, for a suffix S_t, define $u_{k,r}(t)$ to be the node u having maximal depth such that

1. u is on the path from the root to ℓ_k (i.e. ℓ_k is a leaf in T_u).
2. S_t is eligible at u.

As we wish to find the location t, for which the portion of $\mathrm{LCP}(S_k, S_t)$ which is fully included in $S[l\,..\,r]$ is maximal, we are actually interested in the *relevant* suffix(es) S_t for which $\mathrm{length}(u_{k,r}(t))$ is maximal.

This supplies the intuition for our algorithm. Given k, we consider S_k, represented by the path from the root to ℓ_k. We search this path for the lowest node v for which there exists t such that S_t is relevant and is eligible at v.

Notice that the notion of eligibility satisfies the property that for a node u, if u is eligible for some relevant suffix, all of its ancestors are eligible for this suffix as well. In addition, if the suffix tree had been preprocessed for answering level-ancestor queries, by the methods of, for example, [3], we can find the ancestor of ℓ_k of a specific depth d in $O(1)$ time. We conclude that we can perform a binary search on the depth of nodes on this path: in each node u we probe, we efficiently test whether some relevant suffix is eligible at u, by querying for $\mathrm{emptiness}([x(l_u), x(r_u)] \times [l, r - \mathrm{length}(u) + 1])$. The x-axis range $[x(l_u), x(r_u)]$ assures us that we consider only suffixes for which the string represented by u is a prefix, and the y-axis range $[l, r - \mathrm{length}(u) + 1]$ assures us we consider only such suffixes which are relevant and are eligible at node u (see Definition 2). However, instead of the ordinary $O(\log n)$-time binary search, we use a mixed "galloping" and ordinary binary search approach: we conduct the search by iterations, where in the i-th iteration we probe the node on the path whose depth is $2^{i-1} - 1$ and conduct the proper range emptiness query on it, repeating this process until the first node whose emptiness query returned a positive result is encountered. Denote this node as q and denote the last node probed before q as p. Now we find v by direct binary searching on the sub-path between p and q.

Assume we have found the node v described before, i.e., v is the lowest node on this path for which $\mathrm{emptiness}([x(l_v), x(r_v)] \times [l, r - \mathrm{length}(v) + 1])$ returned a negative result, and let w be its child on the path. If there is only a single point (which corresponds to a single suffix) which exists in $[x(l_v), x(r_v)] \times [l, r - \mathrm{length}(v) + 1]$, the start location of the corresponding suffix will be the location to be chosen. However, this may not be the case: there might be several relevant and eligible suffixes whose corresponding grid points are in that range. In this case, since v is the node $u_{k,r}(t')$ for each such suffix $S_{t'}$, there might be an additional eligible portion of those suffixes on the edge (v, w) (figuratively speaking; we mean of course that the additional eligible portion is a prefix of the substring represented by the label of (v, w)). Furthermore, this additional portion may be of a different length for each relevant suffix which is eligible at v. Choosing the right suffix in this case is performed using a range searching for minimum query: we prefer the suffix that has the minimal (i.e. leftmost) start location of the above, as its additional eligible portion on (v, w) will be the longest. This is done by querying $\text{rangemin}_y([x(l_v), x(r_v)] \times [l, \infty])$. Let (x, y) be the point returned by the query. We return y as the start location and $\min\{\text{length}(\text{LCA}(S_k, S_y)), r - y + 1\}$ as the phrase length.

4 Answering ILCP Queries

Here our primary goal is to obtain an efficient way of finding $\text{ILCP}(k, l, r)$. Recall that in this case we are allowed to exceed location r when searching for $\text{ILCP}(k, l, r)$. This is the equivalent of finding the location $l \leq t \leq r$, for which the longest common prefix of $S[k \mathinner{.\,.} j]$ and the suffix S_t, is maximal.

Consider the suffix S_k. Clearly, it is sufficient to find the suffix S_t for which $|\text{LCP}(S_k, S_t)|$ is maximized;

In the following methods to be described, we will constantly assume the suffix S_t is *lexicographically smaller* than S_k. The process for the case where S_t is lexicographically greater than S_k is symmetric. Therefore, all we are required is to choose the best of both, i.e., the option yielding the greater $|\text{LCP}(S_k, S_t)|$ value.

Once the aforementioned location t is found, we compute $|\text{LCP}(S_k, S_t)|$. Therefore, to summarize, we have two steps: (1) finding the location t, and (2) computing $|\text{LCP}(S_k, S_t)|$.

4.1 Finding the Start Location t

We use a reduction to the problem of range searching for minimum on a grid, as opposed to *range reporting* used in [4].

Consider the suffix S_k, and consider the set of suffixes $\Gamma = \{S_l, \ldots, S_r\}$. Since $|\text{LCP}(S_k, S_t)| = \max_{t' \in [l,r]} |\text{LCP}(S_k, S_t')|$, S_t is in fact the suffix lexicographically closest to S_k, out of all the suffixes of the set Γ. Since we have assumed w.l.o.g. that S_t is lexicographically smaller than S_k, we had actually assumed $x(\ell_t) < x(\ell_k)$, or equivalently, that ℓ_t appears to the *left* of ℓ_k in the suffix tree. Incorporating the lexicographical ranks of S_k and S_t into the expression, t is actually the value which maximizes the expression $\max\{x(\ell_t) \mid l \leq t \leq r \text{ and } x(\ell_t) < x(\ell_k)\}$. Notice that $t = y(\ell_t)$.

Now consider the set $P = \{(x(\ell), y(\ell)) \mid \ell \text{ is a leaf in } \text{ST}(S)\}$. Assuming indeed $x(\ell_t) < x(\ell_k)$, we are interested in finding the maximal value $x(\ell_t)$, such that $x(\ell_t) < x(\ell_k)$, and $l \leq y(\ell_t) \leq r$. It immediately follows that the point $(x(\ell_t), y(\ell_t)) \in P$ is the point in the range $[-\infty, x(\ell_k) - 1] \times [l, r]$ having maximal x-coordinate, and therefore can be obtained efficiently by querying $\text{rangemax}_x([-\infty, x(\ell_x) - 1] \times [l, r])$. Once we have found the point $(x(\ell_t), y(\ell_t))$, we can locate ℓ_t, as it is the $x(\ell_t)$-th leaf from the left. The leaf ℓ_t will be of importance in the next section.

4.2 Computing $|\mathrm{LCP}(S_k, S_t)|$

Consider ℓ_k and ℓ_t as described above. Since $|\mathrm{LCP}(S_i, S_j)| = \text{length}(\mathrm{LCA}(\ell_i, \ell_j))$ for any i and j, it is sufficient to find the node $w = \mathrm{LCA}(\ell_k, \ell_t)$ and then to compute $\text{length}(w)$. Using the methods of Harel and Tarjan [9], an LCA query can be answered in constant time. If the value $\text{length}(u)$ for each node u has been stored in u beforehand, we conclude the value $\text{length}(w)$ is obtainable in $O(1)$ time.

The entire process for suffixes $S_{t'}$, $t' \in [l, r]$, which are lexicographically greater then S_t is symmetric. For those, the proper query which will be performed is $\text{rangemin}_x([x(\ell_k) + 1, \infty] \times [l, r])$.

5 Outline of Substring Compression Query Algorithms

Given locations i and j which induce the substring $S[i \mathinner{.\,.} j]$ to be compressed, we describe the outline of our methods, in an inductive manner:

- For the first location i, two cases exist, according to query type:
  SCQ: write the encoded representation of $S[i]$.
  GSCQ: set $k \leftarrow i$ and calculate $\mathrm{BLCP}(k, \alpha, \beta)$. For convenience, we denote $|\mathrm{LCP}| = |\mathrm{BLCP}(k, \alpha, \beta)|$.
- For a general location, assume $S[k \mathinner{.\,.} j]$ is left to be compressed. Again two cases exist:
  SCQ: the LZ method revolves around finding $\mathrm{ILCP}(k, i, k-1)$. For convenience, we denote $|\mathrm{LCP}| = |\mathrm{ILCP}(k, i, k-1)|$.
  GSCQ: here we calculate both $\mathrm{ILCP}(k, i, k-1)$ and $\mathrm{BLCP}(k, \alpha, \beta)$, and choose the longest of both. For convenience, this time we denote $|\mathrm{LCP}| = \max\{|\mathrm{ILCP}(k, i, k-1)|, |\mathrm{BLCP}(k, \alpha, \beta)|\}$.

It is important to note that in all cases we need not find the LCP itself, but rather it is sufficient to find its starting position t and its length. Once the proper $|\mathrm{LCP}|$ value is obtained, if $k + |\mathrm{LCP}| - 1 > j$, we truncate its last characters, leaving only the first $j - k + 1$.

If no such LCP exists (e.g., $|\mathrm{ILCP}| = 0$ and, if applicable, $|\mathrm{BLCP}| = 0$), we revert to writing the encoded representation of the current character, i.e., $S[k]$. Otherwise, we write the encoded representation of the distance to the starting position t (i.e., the value $k - t$) and length of LCP, and set $k \leftarrow k + |\mathrm{LCP}|$. If $k \leq j$, we repeat this process, otherwise, we stop.

6 Substring Compression Query

Given a string $S[1\,..\,n]$, it will be preprocessed to efficiently answer queries of the form SCQ(i, j), in which we are asked to find the compression of the substring $S[i\,..\,j]$. The compression of $S[i\,..\,j]$ will then be computed by performing ILCP queries in the manner described above until the compressed representation of the entire substring has been found.

6.1 Analysis

Our running times and space used are heavily affected by the choice of the range searching structure used. If we choose to use the range searching structure capable of answering range searching for minimum/maximun queries, as it is described by Lenhof and Smid [11], and modified to work on an $[n] \times [n]$ grid in [10]. The analysis is depicted in the following theorem:

Theorem 1. *SCQ(i, j) can be answered in worst-case $O(C(i, j) \log\log n)$ time, using a structure which employs $O(n \log n)$ space, and can be built in* expected *$O(n \log n \log\log n)$ time.*

Proof. The range searching for minimum (maximum) structure used [10], supports queries in worst-case $O(\log\log n)$ time, uses $O(n \log n)$ space, and can be built in overall expected $O(n \log n \log\log n)$ time.

Preprocessing Time. Consists of: $O(n \log \min(n, |\Sigma|))$ for the suffix tree construction; $O(n)$ time for a depth-first traversal in order to mark each node u and each leaf ℓ with length(u) and $x(\ell)$, respectively; $O(n)$ time for the preprocessing in order to answer future LCA queries [9]; $O(n \log n \log\log n)$ expected preprocessing time for the range searching structure [10]. We conclude the preprocessing time is overall expected $O(n \log n \log\log n)$.

Space. Consists of: $O(n)$ for the suffix tree, augmented with the additional $x(\ell)$ and length(u) values, and LCA information; $O(n \log n)$ for the range searching structure. We conclude the space used is $O(n \log n)$.

Query Time. For each of the $C(i, j)$ phrases encoded, we use: $O(\log\log n)$ for range searching for maximum (resp. minimum) queries made in order to find ℓ_{t_1} (resp. ℓ_{t_2}); $O(1)$ in order to compute both $|\text{LCA}(\ell_k, \ell_{t_1})|$ and $|\text{LCA}(\ell_k, \ell_{t_2})|$, and choose the maximum of both. We conclude the query time is overall $O(C(i, j) \log\log n)$. □

Theorem 2. *For any $\epsilon > 0$, SCQ(i, j) can be answered in worst-case $O(C(i, j))$ time, using a structure which employs $O(n^{1+\epsilon})$ space, and can be built in* worst-case *$O(n^{1+\epsilon})$ time.*

Proof. Notice that our range queries are performed on $x(\ell)$ and $y(\ell)$ values. A unique property of these values is that no $x(\ell)$ or $y(\ell)$ value occurs twice in P, i.e., the sequence of point x-coordinates, and the sequence of point y-coordinates, are both permutations of $[n+1]$. Using the *range next value* structure of [5] allows us to obtain the following tradeoff: preprocessing time and space used are dominated by the preprocessing time and space used by the range searching structure. For the query time, since a single range searching for minimum/maxiumum query can now be answered in $O(1)$ time, the overall query time is now worst-case $O(C(i,j))$. □

As mentioned, we offer a tradeoff which is based upon replacing the range searching structure with the one of Mäkinen and Navarro [12]. This structure can support range searching for minimum/maxiumum in $O(\log n)$ time. While this functionality does not appear explicitly in [12], it can be inferred using standard techniques and is not presented here due to lack of space. Furthermore, this structure requires space of only $O(n)$, and can be constructed in $O(n \log n)$ time. The analysis therefore follows:

Theorem 3. *SCQ(i,j) can be answered in worst-case $O(C(i,j)\log n)$ time, using a structure which employs $O(n)$ space, and can be built in* worst-case *$O(n \log n)$ time.*

Proof. Again, preprocessing time and space used are dominated by the preprocessing time and space used by the range searching structure. For the query time, since a single range searching for minimum/maxiumum query can now be answered in $O(\log n)$ time, the overall query time is now worst-case $O(C(i,j)\log n)$. □

7 General Substring Compression Query

For GSCQ, in addition to the two locations i and j, which denote the substring $S[i\,..\,j]$ to be compressed, we receive two more indices α and β, which induce a *context* substring $S[\alpha\,..\,\beta]$. This time we are asked to provide $\mathrm{LZ}(S[i\,..\,j] \mid S[\alpha\,..\,\beta])$.

Here, when trying to compress $S[k\,..\,j]$ for some $i \leq k \leq j$, we have two options: for the first we consider phrases having a start position $i \leq t \leq k-1$. This option is the one solved in Sect. 6, using ILCP queries. The second, is to consider phrases taken from $S[\alpha\,..\,\beta]$. This will be done using a BLCP query.

7.1 Analysis

The analysis is depicted in the following theorem:

Theorem 4. GSCQ(i,j,α,β) *can be answered in worst-case*

$$O\left(C_{\alpha,\beta}(i,j)\left(\log\left(\frac{j-i}{C_{\alpha,\beta}(i,j)}\right)Q_{\mathrm{empt}} + Q_{\mathrm{rmin}}\right)\right)$$

time, using a structure which takes $O(S_{\mathrm{empt}} + S_{\mathrm{rmin}})$ space, and is built in $O(n \log \min\{n, |\Sigma|\} + P_{\mathrm{empt}} + P_{\mathrm{rmin}})$ time, where Q_{empt}, P_{empt}, and S_{empt} (resp. Q_{rmin}, P_{rmin}, and S_{rmin}) are the query time, preprocessing time and space of the range emptiness (resp. range searching for minimum) structure, respectively.

Proof. As follows:

Preprocessing Time. Consists of: $O(n \log \min(n, |\Sigma|))$ for the suffix tree construction; $O(n)$ time for a depth-first traversal in order to mark each node u with $x(l_u)$, $x(r_u)$, and length(u); $O(n)$ time for the preprocessing in order to answer future LCA queries [9]; $O(n)$ time for the preprocessing in order to answer future level-ancestor queries [3]. In addition we have the preprocessing times associated with the range searching structures.

Space. Consists of: $O(n)$ for the suffix tree, augmented with the additional $x(l_u)$, $x(r_u)$ and length(u) values, LCA and level-ancestor structure information. These bounds will be dominated by the range searching structures chosen.

Query Time. Consider the query's main loop described in Sect. 5 and Consider the d-th iteration of the query algorithm main loop, and let len_d be the length of the phrase encoded in this iteration ($d = 1, \ldots, C_{\alpha,\beta}(i,j)$). Assume $S[k \mathinner{..} j]$ is the portion left to be compressed at before this iteration, and let nodes v, p, and q be as defined before. It holds that $\text{depth}(p) \leq \text{length}(p) \leq \text{length}(v) \leq |\text{BLCP}(k, \alpha, \beta)|$. Node q was found one iteration after node p. Therefore:

$$\text{depth}(q) \leq 2(\text{depth}(p) + 1) \leq 2(|\text{BLCP}(k, \alpha, \beta)| + 1)\ . \tag{1}$$

We conclude that finding q was done by performing $O(\log|\text{BLCP}(k, \alpha, \beta)|)$ node accesses, and the following binary search, was supported by performing

$$O(\log(\text{depth}(q) - \text{depth}(p))) = O(\log|\text{BLCP}(k, \alpha, \beta)|) \tag{2}$$

node accesses. Since

$$|\text{BLCP}(k, \alpha, \beta)| \leq \max\{|\text{ILCP}(k, i, k-1)|, |\text{BLCP}(k, \alpha, \beta)|\} = \text{len}_d\ , \tag{3}$$

and when accessing each node, a range emptiness query was conducted, overall time for the mixed search described is $O(\log(\text{len}_d) \cdot Q_{\text{empt}})$, where Q_{empt} is the query time used for the emptiness query. We conclude that a $\text{BLCP}(k, \alpha, \beta)$ query can be answered in $O(\log(\text{len}_d) \cdot Q_{\text{empt}} + Q_{\text{rmin}})$, where Q_{rmin} is the time required for the final range searching for minimum query performed. Recall that an $\text{ILCP}(k, i, k-1)$ query is also made — however, since the time for this query is only $O(Q_{\text{empt}})$, it is dominated by the time used for the $\text{BLCP}(k, \alpha, \beta)$ query.
We conclude that GSCQ can be answered in overall

$$O\left(Q_{\text{empt}} \sum_{d=1}^{C_{\alpha,\beta}(i,j)} \log(\text{len}_d) + C_{\alpha,\beta}(i,j) Q_{\text{rmin}}\right) \tag{4}$$

time. $\{\text{len}_d\}_{d=1}^{C_{\alpha,\beta}(i,j)}$ is a partition of $|S[i\,..\,j]| = j - i + 1$, therefore the above expression is maximized when $\text{len}_1 = \cdots = \text{len}_{C_{\alpha,\beta}(i,j)} = \frac{j-i+1}{C_{\alpha,\beta}(i,j)}$. We conclude that GSCQ$(i, j, \alpha, \beta)$ can be answered in time.

$$O\left(C_{\alpha,\beta}(i,j)\left(\log\left(\frac{j-i}{C_{\alpha,\beta}(i,j)}\right)Q_{\text{empt}} + Q_{\text{rmin}}\right)\right)$$

□

Table 2. GSCQ tradeoffs

empt	rmin	Query Time	Space	Preproc. Time
[5]	[5]	$O\big(C_{\alpha,\beta}(i,j)\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\big)$	$O(n^{1+\epsilon})$	$O(n^{1+\epsilon})$
[12]	[12]	$O\big(C_{\alpha,\beta}(i,j)\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\log n\big)$	$O(n)$	$O(n\log n)$
[1]	[12]	$O\big(C_{\alpha,\beta}(i,j)\big(\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\log\log n + \log n\big)\big)$	$O(n\log^{\epsilon} n)$	$O(n\log n)$
[1]	[10]	$O\big(C_{\alpha,\beta}(i,j)\log\big(\frac{j-i}{C_{\alpha,\beta}(i,j)}\big)\log\log n\big)$	$O(n\log n)$	$O(n\log n\log\log n)$

The choice of range emptiness and range searching for minimum structures will determine the time bounds for their respective queries. Tradeoff results are given in Table 2, where the column labeled "empt" denotes the range emptiness structure used, and the column labeled "rmin" denotes the range searching for minimum structure used.

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. In: FOCS 2000: IEEE Symposium on Foundations of Computer Science, pp. 198–207 (2000)
2. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. J. Algorithms 37(2), 309–325 (2000)
3. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
4. Cormode, G., Muthukrishnan, S.: Substring compression problems. In: SODA 2005: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, pp. 321–330. Society for Industrial and Applied Mathematics (2005)
5. Crochemore, M., Iliopoulos, C.S., Kubica, M., Rahman, M.S., Walen, T.: Improved algorithms for the range next value problem and applications. In: STACS, pp. 205–216 (2008)
6. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS 1997: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997), Washington, DC, USA, p. 137. IEEE Computer Society Press, Los Alamitos (1997)
7. Ferragina, P.: Dynamic text indexing under string updates. J. Algorithms 22(2), 296–328 (1997)

8. Ferragina, P., Muthukrishnan, S., de Berg, M.: Multi-method dispatching: A geometric approach with applications to string matching problems. In: STOC 1999: Proceedings of the thirty-first annual ACM Symposium on Theory of Computing, pp. 483–491 (1999)
9. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
10. Keller, O., Kopelowitz, T., Lewenstein, M.: Range non-overlapping indexing and successive list indexing. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 626–631. Springer, Heidelberg (2007)
11. Lenhof, H.-P., Smid, M.: Using persistent data structures for adding range restrictions to searching problems. RAIRO Theoretical Informatics and Applications 28, 25–49 (1994)
12. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 703–714. Springer, Heidelberg (2006)
13. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
14. Muthukrishnan, S.: Personal communication with the second author
15. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
16. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, pp. 1–11. IEEE, Los Alamitos (1973)
17. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)

Text Indexing, Suffix Sorting, and Data Compression: Common Problems and Techniques

Roberto Grossi

Università di Pisa, Italy
grossi@di.unipi.it

Abstract. The talk is a guided tour on text indexing data structures, suffix sorting, and data compression. We discuss how they share common problems on text suffixes, showing the interplay among some of the algorithmic techniques that have been devised so far. In the following, given a text $T = T[1, n]$ of n symbols, we denote by s_i its suffix $s_i = T[i, n]$ for $1 \le i \le n$.

A text indexing data structure stores the suffixes $s_1, s_2, \ldots, s_n$ of T at preprocessing time, in a suitable format that can support pattern matching queries over T. For example, given a pattern string P of m symbols, one type of query is that of computing how many times P appears in T, whose $O(m + \log n)$ time complexity in the comparison model compares favorably with the $O(m + n)$ cost required by full text scanning [8]. Notable examples of text indexing data structures are suffix trees [10,14] and suffix arrays [9] for usage in main memory, string B-trees [4] and cache-oblivious string B-trees [1] for usage in external and hierarchical memory, to name a few.

Suffix sorting requires to arrange the suffixes $s_1, s_2, \ldots, s_n$ in lexicographic order. This is the major computational bottleneck in suffix-based algorithms, and can be solved in $O(n \log n)$ time in the comparison model (e.g. [7]). Having sorted the suffixes, it is not difficult to build a text indexing data structure in (nearly) linear time. Suffix sorting is crucial also in data compression, as witnessed by the importance of the Burrows-Wheeler transform [3]. The techniques adopted in the aforementioned topics converged in several ways into the rich fields of compressed text indexing [5,6,11,13] and succinct data structures [2,12], with some old and new open problems.

References

1. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-oblivious string B-trees. In: ACM (ed.) Proc. of the 25th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems, Chicago, IL, USA 2006, June 26–28, 2006, pp. 233–242. ACM Press, New York (2006)
2. Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM Journal on Computing 28(5), 1627–1640 (1999)

3. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Research Report 124, Digital SRC, Palo Alto, CA, USA (May 1994)
4. 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), 236–280 (1999)
5. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52(4), 552–581 (2005)
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)
7. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM 53(6), 918–936 (2006)
8. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6, 323–350 (1977)
9. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
10. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2), 262–272 (1976)
11. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), 2:1–2:61 (2007)
12. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4), 1–43 (2007)
13. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms 48(2), 294–313 (2003)
14. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)

Contracted Suffix Trees: A Simple and Dynamic Text Indexing Data Structure

Andrzej Ehrenfeucht[1], Ross M. McConnell[2], and Sung-Whan Woo[2]

[1] Dept. of Computer Science, 430 UCB, University of Colorado at Boulder, Boulder, CO 80309-0430, USA
[2] Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523-1873 USA

Abstract. We address the problem of finding the locations of all instances of a string P in a text T, where of T is allowed to facilitate the queries. Previous data structures for this problem include the suffix tree, the suffix array, and the compact DAWG. We modify a data structure called a *sequence tree*, which was proposed by Coffman and Eve for hashing, and adapt it to the new problem. We can then produce a list of k occurrences of any string P in T in $O(||P|| + k)$ time. Because of properties shared by suffixes of a text that are not shared by arbitrary hash keys, we can build the structure in $O(||T||)$ time, which is much faster than Coffman and Eve's algorithm. These bounds are as good as those for the suffix tree, suffix array, and the compact DAWG. The advantages are the elementary nature of some of the algorithms for constructing and using the data structure and the asymptotic bounds we can give for updating the data structure when the text is edited.

1 Introduction

In this paper, we consider the problem of finding occurrences of a pattern string P in a text T, where preprocessing of T is allowed in order to create a data structure that speeds up the search.

Let m denote the length $||P||$ of P, let n denote the length $||T||$ of T, and let k be the number of positions in T where P occurs as a substring. For simplicity, we assume for the moment that the size of the alphabet Σ is fixed.

Previous data structures for this problem include the *suffix tree* [1], the *compact directed acyclic word graph (compact DAWG)* [2], and the *suffix array* [3]. The first two approaches take $O(n)$ time to build the data structure, and $O(m + k)$ time to find the k positions where the pattern string occurs.

The suffix array can be constructed in $O(n)$ time, and takes $O(m + \log n)$ time to produce a pointer to a list of occurrences of of P in T. A slightly slower approach takes $O(m \log n)$ time, and this approach is of practical interest because of its simplicity.

In this paper, we describe an alternative to these data structures, which we call a **contracted suffix tree.** It can also be constructed in $O(n)$ time, and takes $O(m + k)$ time to find the k occurrences of the pattern string. Unlike the suffix array, it does not give a pointer to a list of occurrences, but it can be made the basis of a data type that takes $O(m)$ time to input P and $O(1)$ time to produce an occurrence each time one is requested, even if the user does not ask for all occurrences. This is just as good as producing a list, since it provides what amounts to a list iterator. (The suffix tree can also be augmented with extra pointers to provide such an interface.)

A special case of the contracted suffix tree, called the *position heap* can be made the basis of a list interface that takes $O(m)$ time to input P and produces a list iterator that gives the positions in left-to-right order of the occurrences in the text, at a cost of $O(\log k)$ time per element. The previous approaches give the occurrences in an order that has no relation to their left-to-right order in the text.

Like the suffix tree and the compact DAWG, and unlike the suffix array, our bounds must be increased by a $\log |\Sigma|$ factor when the size of the alphabet, Σ, is introduced as a variable. This factor comes from the time required to find the child of a node on the child edge labeled b. This can be improved to $O(1)$ *expected* time with a hash table that returns the child given a hash key consisting of the parent and a letter. This is nevertheless a disadvantage when compared to suffix arrays.

The proposed approach has the advantage that it has a good time bound for modifying the data structure after arbitrary text edits on T. The *generalized suffix tree* allows search for a pattern string in a collection of texts. In [4], it is shown that it is possible to implement it to allow insertion and removal of any text X in the collection in $O(||X||)$ time. However, X must be inserted or removed in its entirety and smaller edits on X are not supported. Very recently, Salson *et. al.* have given an approach that takes $O(n)$ worst-case time to modify a variant of the suffix array after an arbitrary edit operation on T [5]. This is as bad as the cost as discarding the suffix array and rebuilding it from the beginning. However, they argue that their approach is much more efficient in practice, and support this with empirical studies on benchmarks.

Except when T has very low entropy, our proposed data structure can be updated efficiently when the text T is modified. Let $h(T)$ be the length of the largest substring X of T that is repeated more than $||X||$ times in T. A few moment's consideration reveals that $h(T)$ can be expected to be quite small for most practical applications. The expected value of $h(T)$ is $O(\log n)$ when T is a randomly-generated string. Since few applications deal with random strings, a more important observation is that long repeated substrings in T have little impact on the value of $h(T)$ unless they are repeated an inordinate number of times.

Updating the proposed structure after deletion or insertion of a consecutive block of b characters takes $O((h(T)^2 + bh(T)))$ time. The tradeoff of implementing it in this way is that searches take $O(m \log k + k)$ time, rather than $O(m + k)$ time, to produce the k occurrences of the pattern string. In the worst case, as when $T = a^n$, $h(T) = \Theta(n)$, and one can resort to the $O(n)$ bound obtained by discarding and rebuilding the structure. However, the bound is a stronger one than $O(n)$ because it characterizes analytically the relationship between the running time and an easily-understood property of the text.

We can give a somewhat stronger bound as follows. Let $h(T, i)$ be the length of the longest string X that has more than $||X||$ occurrences in T and an occurrence containing a pointer to position i in T. We can update the structure after deletion of a block of b characters in $O((h'^2 + bh') \log n)$ time, where h' is the maximum of $h(T, i)$ and $h(T', i)$ over the positions i that were affected by the insertion or deletion and T' is the final string. Thus, the update is efficient unless it occurs inside a very large section of the text that is repeated very many times. Updates only take significant time when they occur in regions of the text that have very low entropy. Indeed, the proposed structure has a relationship to a structure used in the Lempel-Ziv data compression algorithm [6].

An even stronger result is that moving a consecutive block of b characters from one place to another in the text takes $O((h(T)^2)$ time, independently of the number b of characters in the block. This is a common editing operation and the correction to a common type of error in databases of genetic sequences.

2 Preliminaries

Let λ be the null string. If $X = x_1 x_2 ... x_k$ is a string, we let $||X||$ denote the length k of x. The **reverse** of X is the string $X^R = x_k x_{k-1} ... x_1$. If Y is a prefix $x_1 x_2 ... x_i$ of X, let $X - Y$ be the result $x_{i+1} ... x_k$ of removing Y from the front of X.

For reasons that will become clear shortly, we adopt the convention of numbering the positions of the text T from right to left, so $T = t_n t_{n-1} ... t_1$. Let T_i denote the suffix $t_i t_{i-1} \dots t_1$ beginning at position i. Let us distinguish a **substring** $P = p_1 p_2 ... p_m$ of a T from an **instance** (P, i) of P in T, where $P = t_i t_{i-1} ... t_{i-m+1}$. The null substring, λ, is considered to occur at every position.

Definition 1. *A* **trie** *on alphabet Σ denotes a rooted tree T with the following properties:*

1. *Each edge is labeled with a character;*
2. *For each node u and letter $b \in \Sigma$, there is at most one edge with label b from u to a child of u.*

Given a trie, let us say that the **label of a path** from the root to a node u is the string given by the sequence X of characters that occur on edges of the path. This is the **path label** of u. Because of the second property, the path label uniquely identifies u. We therefore adopt the convention of treating the node and its path label as interchangeable objects. For example, we may consider whether a *string* X is a *node* of the trie, or whether one *node* is a *substring* of another. Note that one node is a prefix of another if and only if it is an ancestor in the trie.

A basic operation on a trie takes an input string $P = p_1 p_2 ... p_m$ and finds the largest prefix P' of P that is a node of the trie. Since $|\Sigma|$ is fixed, this is easily accomplished in $O(||P'||)$ time by starting at the root and iteratively taking edges labeled with the sequence of letters from P, until P is exhausted or a node is encountered that doesn't have a child on the next letter of P. Let us call this operation **indexing** into the trie.

3 Sequence Hash Trees

A data structure of Coffman and Eve [7], called a **sequence hash tree**, was designed for the problem of implementing hash tables (dictionaries) whose keys are strings. It consists of a trie for indexing into the table. The structure of the tree depends on the order in which the strings are inserted. We describe a small variant that is easier to adapt to our substring matching problem, below.

Let $\mathcal{S} = (S_1, S_2, \ldots S_n)$ be a given ordering of the strings. Without loss of generality for our purposes, we may assume that no string is $\mathcal{S}$ is a prefix of any other. The trie that they construct is defined by induction, as follows. If $i = 1$, the trie H_1 is just a root node with a pointer to S_1. If $i > 1$, then H_i is obtained from H_{i-1} by finding the shortest prefix Xb of S_i that is not already a node of the trie. A new node Xb is added as the child of node X on edge labeled b, and a pointer is installed from it to S_i.

Figure 1 gives an example. To find an occurrence of a string S in the hash table, they index into the trie $H_n = H$ on the longest prefix X of S that is a node of H. For each node on the path from the root to X, they check whether the hash-table entry the node points to matches S.

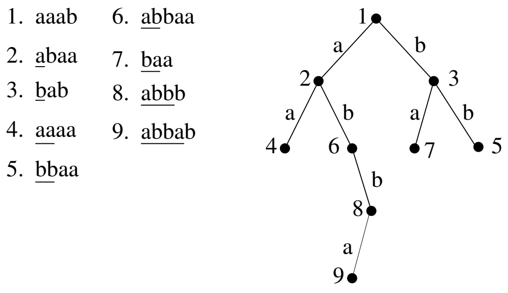


Fig. 1. The sequence hash tree of a set of strings. Each string is installed at the shortest prefix that isn't already a node of the sequence hash tree. The shape of the tree depends on the order in which the strings are inserted.

Their algorithm for deleting string from the tree consists of finding a leaf descendant Y of the node X that points to the string, copying Y's pointer to X, and deleting Y. The insertion algorithm indexes into the new string S until it reaches a null pointer, creates a new leaf child X on that pointer, and makes it point to S. Both of these operations take time proportional to the depth of the tree.

4 Contracted Suffix Trees

We consider how to apply Coffman and Eve's sequence hash trees to our problem of finding all occurrences of P in T. We build the sequence hash tree for the set of *suffixes* of T, and record, in the node generated by each suffix, a pointer to where the suffix begins in T. P occurs at location i in T if and only if it is a prefix of suffix T_i. We append a special character \$ to the end of T to ensure that no suffix of T is a prefix of another. Instead of looking for an entry that P matches, we look for all entries that P is a prefix of; the locations where these suffixes begin in T gives the locations where P occurs.

Coffman and Eve's paper has received little attention since it was published in 1970, due, in no doubt, to the existence of superior ways of implementing a hash table. In the present paper, we show that this data structure is a much richer when considered in the context of the new problem. The structure of the set of suffixes of a text T allows us to derive interesting and algorithmically useful properties that do not apply in the general case addressed by Coffman and Eve. In particular, we show that it has height at most $h(T)$, and show that if the suffixes are inserted in ascending order of length, it is now possible to build the data structure in time that is linear in $n = ||T||$, that is, in $O(1)$ time, amortized, per hash key. We show how the tree can be augmented with *maximal reach pointers* so that finding all k entries that have P as a prefix takes $O(m+k)$ worst-case time, independently of the height of the tree.

The $h(T)$ height bound gives an $O(h(T))$ bound for Coffman and Eve's operations for inserting and deleting a suffix. This gives the $O((b + h(T))h(T))$ bound for updating the data structure when a consecutive block of b characters is deleted and the $O((h(T))^2)$ bound for moving a block of b characters from one place in T to another.

To distinguish this special case of their structure, we will call it the **contracted suffix tree**; in contrast to the suffix tree, which has a path for each suffix, the contracted suffix tree only has paths for prefixes of the suffixes.

4.1 A Naive Query Algorithm for Contracted Suffix Trees

In this section, we give the **naive query algorithm**, which determines the k occurrences of P in T in $O(m^2 + k)$ time, where m is the length of P. Below, we show how to improve this to $O(m + k)$. As explained below, the $O(m^2)$ component is overly pessimistic in practice, however, and the naive approach takes $O(m + k)$ expected time on random strings. It may be competitive with the $O(m + k)$ worst-case approach for many applications, due to its simplicity and lower space requirements.

Lemma 1. *If P is not a node of a contracted suffix tree of T, it has fewer than $||P||$ occurrences in T.*

Proof. Every suffix of T that has P as a prefix results in a new node of the tree that is either a proper prefix of P or that has P as a prefix. Since P does not occur in the tree, it is not a prefix of any node in the tree. Therefore, the number of suffixes of T that have P as a prefix, hence the number of occurrences of P, is bounded by the number of proper prefixes of P.

Lemma 2. *The height of a contracted suffix tree of T is at most $h(T)$.*

Proof. Let $X = x_k x_{k-1} \ldots x_1$ be a deepest leaf of the tree. Let X_i denote the prefix $x_k x_{i-1} \ldots x_i$ of X. For each i from 1 through k, X_i occurs at least i times in T because it has at least i descendants, $\{X_i, X_{i-1}, \ldots X_1\}$, and each of these points to an occurrence of a substring of which X_i is a prefix. Therefore, $X_{\lceil k/2 \rceil}$ has length $\lfloor k/2 \rfloor$ and occurs at least $\lceil k/2 \rceil$ times in T. It must be that $k/2$ is a lower bound on $h(T)$, so the height k is $O(h(T))$.

A node X contains a pointer to a position i where X occurs. It follows that if P is a node of the contracted suffix tree, the positions in T pointed to by ancestors of P may or may not be occurrences of P, while the positions pointed to by descendants of n are all occurrences of P. This gives a simple algorithm for a special case of the query:

– **Case 1.** *P is a node in H.*
  The applicability of the case is detected by indexing into H on P. For each ancestor (prefix) X' of P, determine whether P occurs at the position i pointed to by X'. In addition to these, report positions contained in all descendants of P. (See Figure 2.)

For each ancestor X' of P, it takes $O(m)$ time in the worst case to check whether P occurs at the position pointed to by X'. There are at most m ancestors of P, so the total time spent on these checks is $O(m^2)$. Finding the k' descendants of x takes $O(k')$ time, since no checking is required. The total is $O(m^2 + k)$.

– **Case 2.** P is not a node in H.
  Let X be the longest prefix of P that is a node in H. For each ancestor (prefix) X' of X, determine whether P occurs at the position i pointed to by X', and report the ones that do. (See Figure 2.)

Since Case 2 applies, X is a proper prefix of P. Let b be the character that follows this instance of X in P. Since X has no child on edge labeled b, any descendant of X contains a pointer to a position that is an occurrence of Xc for $c \neq b$. Therefore, no descendant of X is an occurrence of P, and we may drop the k from the $O(m^2 + k)$ bound we got in Case 1 to get an $O(m^2)$ bound.

4.2 A Naive Construction Algorithm

Coffman and Eve' algorithm for inserting an entry to the sequence hash tree takes $O(h)$ time, where h is the height of the tree. By Corollary 2, we may use this algorithm to insert each suffix of T, yielding a contracted suffix tree in $O(nh(T))$ time.

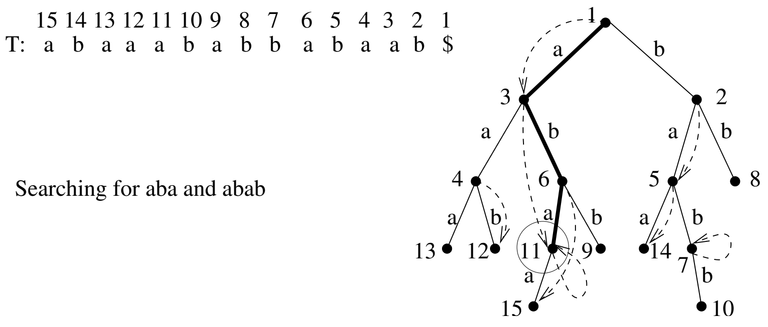


Fig. 2. The string *aba* is the node labeled with a pointer to position 11, so it falls into Case 1. The position labels of ancestors of this node, $\{1, 3, 6\}$ may or may not be occurrences of *aba*, and this must be checked. A naive approach checks whether the occurrence of *ab* at position 6 is followed by an *a* and the occurrence of *a* at position 3 is followed by *ba*, and the occurrence of the null string at position 1 is followed by *aba*. This approach takes $O(m)$ time per ancestor. This can be improved to $O(1)$ time per ancestor by checking whether the *maximal-reach pointer* of the ancestor (dashed arrows) points to a descendant of node *aba* (Section 5). The position labels of descendants of *aba* $\{11, 15\}$ must be occurrences of *aba*, and do not have to be checked. The string *abab* falls into Case 2. Node *aba* is the largest prefix that is a node of the tree. Only the ancestors $\{1, 3, 6\}$ of this node can be occurrences of *abab*. This can be checked by verifying that the maximal reach pointer points to *aba*, indicating an occurrence of *aba* not followed by *a*, and, if so, checking whether the occurrence is followed by *b*.

5 An $O(m + k)$ Bound for Searches

At a node X pointing to position i of T, let Y be the longest prefix of T_i that is a node of a contracted suffix tree H of T. Clearly, X is a prefix of Y, though it is possible that it is not a proper prefix. We install a **maximal reach pointer** from node X to node Y. (See figure 2.)

Lemma 3. *The position i in an ancestor X of P is an occurrence of P if and only if X's maximal reach pointer points to a (not necessarily proper) descendant of P.*

Proof. The nodes of the tree that have P as a prefix are the descendants of P. If X's maximal reach pointer points to a descendant of P, then T_i has P as a prefix, and P occurs at position i. If X's maximal reach pointer does not point to a descendant of P, then since P is a node of the tree, P is not a prefix of T_i, which means that P does not occur at position i.

The **naive algorithm** for installing the maximal-reach pointers is to revisit each position i of T after the suffix heap tree H has been built, indexing into H on the suffix beginning at i, passing through the node containing i, and stopping when a node encountered that has no child on the next letter of T. This is the node that must be pointed to by the node containing position i. Since the height of H is $O(h(T))$ this gives an $O(nh(T))$ algorithm for installing the maximal-reach pointers, which adds nothing to the asymptotic time bound for building H with the naive construction algorithm.

After we construct the tree, we perform a depth-first search of the tree to label each node with a discovery and finishing time, as described in [8]. These are essentially preorder and postorder numbers. Their purpose is to allow us to determine, given nodes X and Y, whether X is an ancestor of Y; this is the case if and only if X has an earlier discovery time and a later finishing time than Y does. We also keep a pointer from each position in T to the node of H that points to it.

Case 1 queries. As before, to find the k' occurrences of P listed in descendants of P, we visit P's subtree in $O(k')$ time. The difference now is that we can determine at each ancestor X' of P whether the position i it contains is an occurrence of P by checking whether the maximal reach pointer of X' points to a (not necessarily proper) descendant of P. This test takes $O(1)$ time using the preorder and postorder numbers, giving the $O(m)$ bound for finding the remaining occurrences of P that its ancestors point to.

Case 2 queries. For Case 2, there are $O(m)$ occurrences of P by Lemma 1. We partition P into substrings by finding the maximal prefix P_1 of P that is a node of H, then the maximal prefix of $P - P_1$ that is a node of H, etc., yielding $(P_1, P_2, \ldots, P_k)$.

Since we are in Case 2, there are $k \geq 2$ strings in this sequence. Since $||P_1||$ is the longest prefix of P that is a node of the tree, any proper descendant X of P_1 fails to be a prefix of P, hence the position at X is not an occurrence of P. Only (not necessarily proper) ancestors of P_1 can contain pointers to occurrences of P. Since P_1 is a prefix of P, only nodes that point to an occurrence of P_1 are candidates to be occurrences of P. An ancestor is a candidate if and only if its maximal-reach pointer points to P_1; if its maximal reach pointer points to a proper descendant of P_1, it points to a node that is not a prefix of P.

By induction on j, we now find which candidate locations are occurrences of $P_1 P_2 \ldots P_j$, and show that there are $O(||P_j||)$ of them, as follows. If $j = 1$, the candidate positions are the $O(||P_1||)$ candidates described above. If $1 < j < k - 1$, the candidate positions are the positions of $P_1 P_2 \ldots P_j$, and we assume by induction that there are $O(||P_j||)$ of them. We test for each candidate position i whether i is an occurrence of P_{j+1} by checking whether $i' = i - ||P_1 P_2 \ldots P_j||$ is an occurrence of P_{j+1} that's followed by $P_{j+2} P_{j+3} \ldots P_k$. We do this by determining whether i' occurs at a (not-necessarily proper) ancestor Y of P_{j+1} and Y's maximal reach pointer points to a (not-necessarily proper) descendant of P_{j+1}. This takes $O(||P_j||)$ time using the preorder and postorder numbers, and since there are $O(||P_{j+1}||)$ ancestors of P_{j+1}, it yields $O(||P_{j+1}||)$ positions. If $j = k-1$, $P_{j+2} P_{j+3} \ldots P_k$ is empty, so we also test whether i' occurs in a descendant of $P_{j+1} = P_k$.

The time bound for finding all occurrences of $P = P_1 P_2 \ldots P_k$ is thus $O(||P_1|| + ||P_2|| + \ldots + ||P_k||) = O(m)$.

6 Building a Contracted Suffix Tree in $O(n)$ Time

We begin by installing the text in an array of characters in $O(n)$ time, so that we can access the letter in any position number i in $O(1)$ time.

Let $H(T)$ denote the unique result of inserting the suffixes of T into the contracted suffix tree in ascending order of their length. This special case of the contracted suffix tree is called the **position heap** [9].

One advantage of the position heap is that it can be used to produce in $O(||P||)$ time an iterator on a list of positions in order in which they occur in the text, at a cost of spending $O(\log k)$, rather than $O(1)$, for returning the next element of the list. The positions are in heap order, that is, the position at each node is smaller than the positions at its children. In Case 1, the positions in ancestors of P therefore occur in sorted order, and for the descendants, a priority queue can be used to manage the topmost descendants that haven't already been returned. In Case 2, the candidate positions are found at ancestors of P_1, which occur in sorted order, and elements are subsequently deleted from this list to give the list of occurrences of P. This gives them in right-to-left order; if left-to-right order is desired, the position heap for the reverse of the text can be used.

Let the **dual** $D(T)$ of the position heap $H(T)$ be the trie where for each node X of $H(T)$, the **reverse** X^R of X is a node of $D(T)$ (see Figure 3).

It is tempting to think that the dual is just the position heap of the reverse of the text, but it is easily verified that this is not the case.

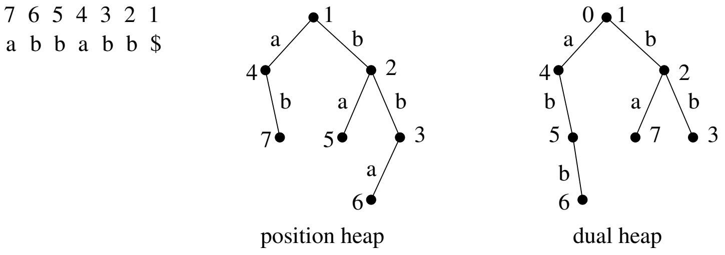


Fig. 3. The position heap and its dual for the text *abbabbb*. The labels of the path leading to a node in the dual is the reverse of the labels of the path leading to it in the position heap.

Let us say that a set of S of strings is **hereditary** if, whenever $X \in S$, every substring of X is also in S.

Lemma 4. *The nodes of the position heap are a hereditary family of strings.*

Proof. Let us show this by induction on the length of $T_i = t_i t_{i+1} ... t_1$. The lemma is trivially true for $H(T_1)$, which has only one node, the empty string. Otherwise, we adopt as the induction hypothesis that the nodes of $H(T_{i-1})$ have the hereditary property. Since $H(T_i)$ differs from $H(T_{i-1})$ only by the addition of a node X, $H(T_i)$ can only fail to have the hereditary property if some proper substring of X fails to be a node of T_i.

This can't be the case if $||X|| < 2$, since λ is a node of $H(T_i)$. Suppose $||X|| \geq 2$. We can then write X as $aX'b$. The parent of $aX'b$ is aX', hence it is a node of $H(T_{i-1})$. Since aX' is longer than X', X' is a node in T_{i-2}. Also, $X'b$ is a prefix of T_{i-1}, and since X' is a node of T_{i-2}, $X'b$ is either added at step $i-1$ or is already a node of T_{i-2}. In either case, it is a node of $H(T_{i-1})$. We conclude that aX' and $X'b$ are nodes of T_{i-1}. By the induction hypothesis, every substring of aX' and $X'b$ is a node of T_{i-1}, hence of T_i, and these are every proper substring of the new node $X = aX'b$.

The lemma is not true for sequence hash trees: the substring *bba* of the node *abba* labeled 9 in Figure 1 is not a node of the tree.

Corollary 1. *The set of nodes of $D(T)$ is the same as the set of nodes of $H(T)$.*

Proof. By definition, every node of $H(T)$ is a node of $D(T)$. It remains to show that every node of $D(T)$ is a node of $H(T)$. Let X be an arbitrary node of $H(T)$. By Lemma 4, not only is every prefix of a node X of $H(T)$ a node of $H(T)$, but so is every suffix. This implies that every ancestor of X in $D(T)$ is a node of $H(T)$. There are no nodes on any path of $D(T)$ that fail to be a node of $H(T)$.

We implement $H(T)$ and $D(T)$ on the same set of nodes, so that each node has a parent in $H(T)$ and a parent in $D(T)$. We continue to refer to each node by its path label X in $H(T)$, even when considering it as a node of $D(T)$. Equivalently, each node of $D(T)$ is denoted by the sequence X of labels on edges from the node to the root of $D(T)$.

We get an $O(n)$ time bound for constructing the position heap by simultaneously constructing the position heap and its dual. During construction, we need the edges to go from child to parent in the position heap and from parent to child in the dual. After the tree is constructed, the dual heap can be discarded and the edges of the position heap can be reversed in $O(n)$ time, to go from parent to child, by bucket sorting edges according to destination vertex.

When going from $H(T_{i-1})$ to $H(T_i)$, a new node must be added to hold a pointer to position i. Let a be the first letter of T_i and let X be the node added at step $i-1$. If position i is the first time a was encountered, we add a new child of the root on edge labeled a in both the position heap and the dual. Otherwise, let aX' be the longest prefix of T_i that is a node of $H(T_{i-1})$. Because X was added at step $i-1$, it follows from the hereditary property that aX was not a node of $H(T_{i-1})$. Therefore, X' is a proper prefix of X. We find X' by ascending through proper ancestors (prefixes) of X in the position heap. X' is the first one that has a child on edge labeled a in the dual.

Let $b = t_{i-||aX'||}$. Since the text is in an array of characters, we may look up b in $O(1)$ time. According the constructive definition of H, the new node in H is $aX'b$. In H, this is the child of aX' on edge labeled b. According to the definition of the dual, $aX'b$ is also the new node of the dual. It must be the child of $X'b$ on edge labeled a. $X'b$ is already known: it was the child of X' encountered on the path from X to X'.

Let us now show that iterating this procedure from 1 through n gives an $O(n)$ time bound for finding $H(T)$ and $D(T)$. We use an amortized analysis to bound this cost over all iterations. The only difficulty is bounding the amount of time traversing the path from X up to X'. Let the *current depth* be the depth of the last node added. The key to the analysis is that each node traversed on this path decreases the current depth by one, and adding the new node $aX'b$ then increases the depth by two. Since the initial depth is 0, the total number of times the depth decreases is bounded by the number of times it increases, which is $O(n)$.

The procedure for adding the maximal-reach pointers to $H(T)$ in $O(n)$ time is similar. We install maximal reach pointers in nodes in the same order in which those nodes were added to $H(T)$. This time, we let the current depth be the depth of the node X pointed to by the next node's maximal reach pointer. The next node pointed to is obtained by searching upward from X to find the lowest ancestor X' such that aX' is a node; this is the node pointed to by the next maximal-reach pointer. The amortized analysis of the running time is the same as it is for the construction.

7 Dynamic Texts

When T changes dynamically, we can no longer assume that the characters of T are in an array.

The nodes of the contracted suffix tree were previously labeled with discovery and finishing times in a depth-first search on the tree. We replace these with pointers into a data structure for dynamic ordered lists that is suitable for looking up in $O(\log n)$ time which of two elements is earlier in the list. The order of elements in this list are the discovery and finishing times of nodes in the current tree. A balanced binary tree where the elements appear in inorder suffices, for example. Each node points to the two elements corresponding to its discovery and finishing times. When a new node is created, its discovery-time element is inserted immediately after the finishing-time element of its left neighbor, or of its parent if it has no left neighbor. Its finishing element is handled symmetrically. When it is deleted, its discovery- and finishing-time elements are simply removed from the list. This list allows one to determine whether one node is an ancestor of another in $O(\log n)$ time, rather than $O(1)$ time. This raises the worst-case time bound for finding which ancestors of P point to occurrences of P from $O(1)$ to $O(\log n)$. This increases the worst-case bound for finding all k occurrences of P from $O(m + k)$ to $O(m \log n + k)$.

Similarly, the text must be implemented with a data structure that allows insertion and deletion of blocks of text in $O(\log n)$, and indexing the character that is currently in position i. This can be carried out with similar schemes; one simple scheme which takes $O(\log n)$, amortized, is based on splay trees, and is described in [10]. If this structure is used, finding all k occurrences of P takes $O(m \log n + k)$, amortized.

Figure 4 shows how to modify the tree when a single character is deleted, in the case the b at position p_6. Each underline under the text represents the occurrence of a node X pointed to X. We can remove p_6 with Coffman and

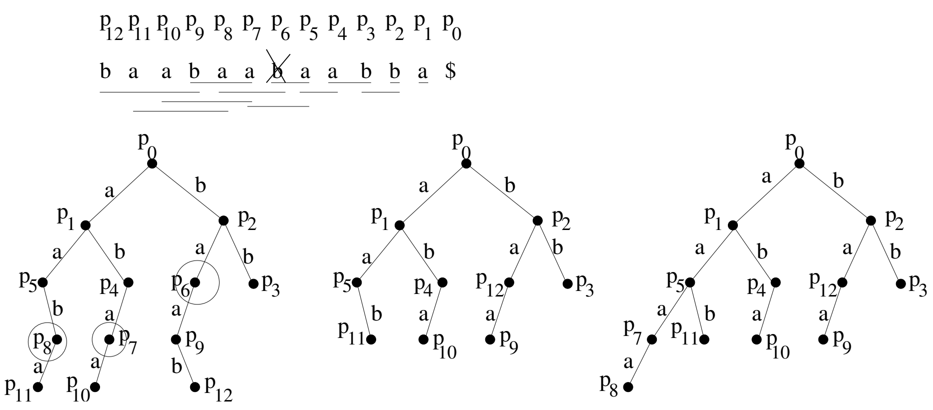

p12 p11 p10 p9 p8 p7 p6 p5 p4 p3 p2 p1 p0
b a a b a a b a a b b a $
p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12
a b

Fig. 4. Modifying a contracted suffix tree when a character is deleted from T

Eve's deletion operation, which replaces it with a leaf descendant, in this case p_{12}. However, p_7 is at node *aba*, which indicates that there is an occurrence of *aba*, represented by an underline, which extends from position p_7 to p_5. This is no longer the case, because the middle letter of *aba* is the *b* that was deleted at position p_6. We must therefore remove p_7, because it is at a node that is no longer a prefix of the suffix that begins at p_7. Similarly, we must remove p_8, which resides at *aab*, whose last letter is the deleted one. Let us call positions such as p_7 and p_8 that are not deleted but must be moved to a new location the *affected positions.*

The middle tree of Figure 4 shows the tree after the deleted and affected positions have been removed. We then reinsert the new suffixes beginning at the affected positions with Coffman and Eve's insertion operation.

Since each node is a string of length $O(h(T))$, there are $O(h(T))$ affected positions, and each one takes $O(h(T) + \log n)$ amortized time to remove. The $\log n$ time comes from the need to remove and insert its discovery and finishing times in the dynamic lists (and the $O(\log n)$ amortized bound comes if you use the splay-tree data structure in [10] for representing the text). $O(h(T) + \log n)$ amortized is $O(h(T))$, amortized, since $\log n = O(h(T))$. Similarly, each affected position takes $O(h(T'))$ amortized time to insert, where T' is the revised text. An easy way to see that $h(T')$ is $O(h(T))$ is to observe that the insertion of the $h(T)$ positions can add at $O(h(T))$ to the height of the middle tree. This gives a total of $O([h(T)]^2)$ time to delete a character. Insertion of a character works similarly and has the same time bound.

When a consecutive block of b characters is deleted, there are still $O(h(T))$ affected characters that precede the b characters of the block. Each of the b deleted characters and the $O(h(T))$ affected characters takes $O(h(T))$ amortized time to process, so the time to delete b consecutive characters from T is

$O((b+h(T))h(T))$, amortized. Similarly, the insertion of b consecutive characters takes $O((b+h(T'))h(T'))$ amortized time, where T' is the new text that results from the insertion.

To get the $O([h(T)]^2)$ amortized bound for moving a consecutive block of b characters from one position to another in the text, we observe that we only need to move affected positions. These are $O(h(T))$ positions at the end of the block, $O(h(T))$ positions preceding the initial location of the block, and $O(h(T))$ positions preceding the new location of the block. The remaining positions in the block are at nodes that continue to be prefixes of the suffixes that they point to, so they don't need to be moved.

References

1. Weiner, P.: Linear pattern-matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory, pp. 1–11. Institute of Electrical Electronics Engineers, London (1973)
2. Blumer, A., Blumer, J., Ehrenfeucht, D., Haussler, D., McConnell, R.: Complete inverted files for efficient text retrieval and analysis. Journal of the ACM 34, 578–595 (1987)
3. Manber, U., Myers, E.: Suffix arrays: a new method for on-line search. SIAM J. Comput. 22, 935–948 (1993)
4. Ferragina, P., Grossi, R., Montangero, M.: On updating suffix tree labels. Theor. Comput. Sci. 201(1-2), 249–262 (1998)
5. Salson, M., Lecroq, T., Lonard, M., Mouchard, L.: Dynamic burrows-wheeler transform. Theoretical Computer Science (accepted, 2009)
6. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24, 530–536 (1978)
7. Coffman, E., Eve, J.: File structures using hashing functions. Communications of the ACM 13, 427–432 (1970)
8. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. McGraw-Hill, Boston (2001)
9. Ehrenfeucht, A., McConnell, R.M.: String searching. In: Mehta, D., Sahni, S. (eds.) Handbook of Data Structures and Applications. CRC Press, Boca Raton (2005)
10. Tarjan, R.E.: Data structures and network algorithms. Society for Industrial and Applied Math., Philadelphia (1983)

Linear Time Suffix Array Construction Using D-Critical Substrings

Ge Nong[1,⋆], Sen Zhang[2], and Wai Hong Chan[3,⋆⋆]

[1] Computer Science Department, Sun Yat-Sen University, P.R.C.
issng@mail.sysu.edu.cn
[2] Dept. of Math., Comp. Sci. and Stat., SUNY College at Oneonta, U.S.A.
zhangs@oneonta.edu
[3] Department of Mathematics, Hong Kong Baptist University, Hong Kong
dchan@hkbu.edu.hk

Abstract. In this paper we present in detail a new efficient linear time and space suffix array construction algorithm(SACA), called the D-Critical-Substring algorithm. The algorithm is built upon a novel concept called fixed-size D-Critical-Substrings, which allow us to compute suffix arrays through a balanced combination of the bucket-sort and the induction sort. The D-Critical-Substring algorithm is very simple, a fully-functioning sample implementation of which in C++ is embodied in only about 100 effective lines. The results of the experiment that we conducted on the data from the Canterbury and Manzini-Ferragina corpora indicate that our algorithm outperforms the two previously best-known linear time algorithms: the Kärkkäinen-Sanders (KS) and the Ko-Aluru (KA) algorithms.

1 Introduction

Background

Suffix arrays were first proposed by Manber and Myers in their seminal SODA'90 paper [1] as a memory efficient alternative data structure to suffix trees. While a suffix tree physically contains all the suffixes of the text, a suffix array is only an integer array consisting of specially arranged indexes which references all the suffixes of the text in their lexicographically ascending order. Due to their light-weighted nature, suffix arrays have been used in large-scale indexing, compressing and pattern matching applications, e.g., internet-scale searching and genome sequence databases, where the magnitudes of data are measured often in billions of characters [2,3]. The original suffix array construction algorithm introduced in [1] is superlinear, it, however, has well stimulated several attempts in designing linear time suffix array construction algorithms during the past two decades. Among them are three well-known linear SACAs contemporarily reported in

⋆ Nong was partially supported by the National Science Foundation of P.R.C. (Grant No. 60873056).

⋆⋆ Chan was partially supported by the Faculty Research Grant (FRG/07-08/II-30).

2003. They are attributed to Kim, Sim, Park and Park (KSP) [4], Kärkkäinen and Sanders (KS) [5], and Ko and Aluru (KA) [6] respectively. While KSP is an array version of Farach's work [7] on suffix trees, the KA and KS algorithms share a same recursive framework but employ two different block compressing schemes and two different sorting methods: the KA algorithm compresses the text using varying length substrings at induction sorting friendly positions, but the KS algorithm compresses the text using fixed size substrings without utilizing induction sort at all. Yuta Mori has independently evaluated the KS and KA algorithms [8] to demonstrate that the KA is over one time faster than the KS. However, the implementation of the KS algorithm is dramatically simpler than that of the KA. This is well evidenced by the sample implementations of the two algorithms from their respective original inventors (The KS can be coded using about 100 lines of C code while the KA uses much more than 1000 lines). The reason is that the fixed size substrings used in the KS algorithm can be easily sorted by the bucket-sort, while the KA has to use very involved logics to maintain S-list in its crucial substring renaming phase.

Our Contributions

We proposed a new algorithm to combine the benefits of both fixed size substrings and the induction sort. The algorithm is called the D-Critical Substring algorithm and has been outlined in our poster at DCC'08 [9]. In this paper, we further extend it by including a detailed analysis and evaluation of the D-critical substring algorithm. The key idea of our D-Critical algorithm is to compress the original string using fixed D-Critical substrings (hence the name of the algorithm), which in turn are defined based on D-Critical characters. As will be seen in the following sections, D-critical characters are induction sort friendly and D-critical-substrings are fixed size. As a result, this new algorithm outperforms both the KA and the KS algorithms.

2 Preliminary

Unless explicitly specified otherwise, the following notations will be used throughout the paper.

- S: a string with n characters consecutively arranged as an array with the index starting from 0.
- $S[i..j]$: the consecutive substring starting at ith position and ending at jth position of S, where $i \leq j$.
- \$: The size-n input string S is terminated by a sentinel \$, which is the unique lexicographically smallest character in S.
- $suf(S, i)$: the suffix in S starting at $S[i]$ and running into the last character.
- $\Sigma(S)$ and $SA(S)$: the alphabet and the suffix array of S, respectively.
- Type-S or type-L suffix: a suffix $suf(S, i)$ is of type-S (type-L respectively) if $suf(S, i) < suf(S, i+1)$ ($suf(S, i) > suf(S, i+1)$ respectively). The last suffix $suf(S, n-1)$ consisting of only the single character \$ (the sentinel) is specifically defined as type-S.

- Type-S or type-L character: a character $S[i]$ is of type-S or type-L if $suf(S,i)$ is type-S or type-L, respectively.
- Leftmost type-S (LMS) character: $S[i]$ is a LMS character if $S[i]$ is type-S and $S[i-1]$ is type-L, where $i \in (0, n-1]$.
- Leftmost type-S (LMS) suffix: $suf(S,i)$ is a LMS suffix if $S[i]$ is a LMS character.
- t: let $t(S,i)$ be the LS-type function of $S[i]$, defined as $t(S,i) = 1$ for $S[i]$ is type-S or else $t(S,i) = 0$. For simplicity, $t(S,i)$ is also denoted as $t[i]$.
- ω: let $\omega(S,i)$ be the ω-weighting function of $S[i]$, defined as $\omega(S,i) = 2S[i] + t[i]$.
- S_ω: the ω-weighted string of S, where $S_\omega[i] = \omega(S,i)$.

3 D-Critical Substring Algorithm

3.1 Algorithm Framework

Our proposed D-Critical substring algorithm, which we simply refer to as the DCS algorithm, is outlined in Fig. 1. Lines 1-3 first produce the reduced problem, which is then solved recursively by Lines 4-8. From the solution of the reduced problem, Line 9 induces the final solution for the original problem. The time and space bottleneck of this algorithm resides at reducing the problem in Lines 1-3, which is $O(n)$, achieved by using D-Critical substrings to re-represent the original text.

3.2 Critical Characters

Definition 1. *(Critical Character) A character $S[i]$ is D-Critical, where $d \geq 2$, iif (1) $S[i]$ is a LMS character; or else (2) $S[i-d]$ is a D-Critical character, $S[i+1]$ is not a LMS character and no character in $S[i-d+1..i-1]$ is D-Critical.*

Definition 2. *(Neighboring Critical Characters) A pair of D-Critical characters $S[i]$ and $S[j]$ are said to be two neighboring D-Critical characters in S, if there is no other D-Critical characters between them.*

Definition 3. *(Critical Substring) The substring $S[i..i+d+1]$ is a D-Critical substring for the D-Critical character $S[i]$ in S. For $i \geq n-(d+1)$, $S[i..i+d+1] = S[i..n-2]\{S[n-1]\}^{d+1-(n-2-i)}$, where $\{S[n-1]\}^x$ denotes that $S[n-1]$ is repeated x times.*

From the above definitions, we have the following immediate observations.

Proposition 1. *In S, (1) all LMS characters are D-Critical characters, (2) the last character must be a D-Critical character, and (3) the first character must not be a D-Critical character.*

Proposition 2. *If $S[i]$ is a D-Critical character, neither $S[i-1]$ nor $S[i+1]$ is a D-Critical character.*

Lemma 1. *The distance between any two neighboring D-Critical characters* $S[i]$ *and* $S[j]$ *in* S *must be in* $[2, d+1]$, *i.e.,* $j - i \in [2, d+1]$, *where* $d \geq 2$ *and* $i < j$.

Proof. From Proposition 2, given $S[i]$ is a D-Critical character, $S[i+1]$ must not be a D-Critical character. In other words, the first d-character on the right hand of $S[i]$ may be any in $S[i+2, i+d+1]$, but must not be $S[i+1]$.

```
DCS-Sort($S, SA$)
     ▷ $S$ is the input string;
     ▷ $SA$ is the output suffix array of $S$;
     $P_1$, $S_1$: array $[0..n_1]$ of integer;
     $bkt$: array $[0..\|\Sigma(S)\| - 1]$ of integer;
 1   Find the sample pointer array $P_1$ for all the fixed-size D-Critical substrings in $S$;
 2   Bucket sort all the fixed-size D-Critical substrings in $P_1$;
 3   Name each D-Critical substring in $S$ by its bucket index to get
     a new shortened string $S_1$;
 4   if $\|S_1\|$ = Number of Buckets
 5      then
 6            Directly compute $SA_1$ from $S_1$;
 7      else
 8            DCS-Sort($S_1, SA_1$);
 9   Induce $SA$ from $SA_1$;
10   return
```

Fig. 1. Our DCS algorithm

3.3 Reducing the Problem

To simplify the discussion, we use $\Psi_{C-d}(S)$ to denote the D-Critical substring array for S, which contains all the D-Critical substrings in S, one substring per item, consecutively arranged according to their original positional order in S.

Definition 4. *(Sample Pointer Array) The array* P_1 *contains the sample pointers for all the D-Critical substrings in* S *preserving their original positional order, i.e.* $S[P_1[i]..P_1[i] + d + 1]$ *is a D-Critical substring.*

From the definitions of P_1 and Ψ_{C-d}, immediately we can conclude $\Psi_{C-d} = \{S[P_1[i]..P_1[i] + d + 1] | i \in [0, n_1)\}$, where n_1 denotes the size (or cardinality) of Ψ_{C-d}. Hereafter, we simply consider P_1 at pointer level, but the underneath comparisons for its items lie in the substrings in Ψ_{C-d}. To compute P_1 from S, we need to know the LS-type of each character in S. This can be done by scanning S once from right to left in $O(n)$ time, by utilizing these properties: (i) $S[i]$ is type-S if (i.1) $S[i] < S[i+1]$ or (i.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is type-S; and (ii) $S[i]$ is type-L if (ii.1) $S[i] > S[i+1]$ or (ii.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is type-L. Provided with the LS-type of each character is known, we can traverse S once from right to left to compute P_1 in $O(n)$ time.

Definition 5. *(Siblings) $P_1[i]$ and $S[P_1[i]..P_1[i]+d+1]$ are said as a pair of siblings.*

Let's bucket sort all the items of P_1 by their ω-weighted siblings (i.e. $S_\omega[P_1[i]..P_1[i]+d+1]$ for $P_1[i]$) in increasing order, where all the buckets are indexed from 0. Then we can replace each item of P_1 with the index of its bucket to obtain a new string S_1. Notice this process is equivalent to a 1-to-1 mapping relationship between a D-Critical string and a unique integer. It is the so-called renaming (or simply naming) step. Here, we have the following observations on S_1.

Lemma 2. *(Sentinel) The last character of S_1 must be the unique smallest character in S_1.*

Proof. From Proposition 1, we know that $S[n-1]$ must be a D-Critical character and the D-Critical substring starting at $S[n-1]$ must be the unique smallest among all sampled by P_1.

Lemma 3. *(1/2 Reduction Ratio) $\|S_1\|$ is at most half of $\|S\|$, i.e. $n_1 \leq \lfloor n/2 \rfloor$.*

Proof. From Proposition 1, $S[0]$ must not be a D-Critical character. We know from Lemma 1 the distance between any two neighboring D-Critical character is at least 2, which immediately completes the proof.

The above two lemmas state that, S_1 is at least half smaller than S and terminated by an unique smallest sentinel too.

Theorem 1. *(Coverage) For any two characters $S_1[i] = S_1[j]$, there must be $P_1[i+1] - P_1[i] = P_1[j+1] - P[j] \in [2, d+1]$, where $d \geq 2$, i.e., $P_1[i+1]$ and $P_1[i]$ are at distance no more than $d+1$, and so do $P_1[j+1]$ and $P_1[j]$.*

Proof. Given $S_1[i] = S_1[j]$, from the definition of S_1, there must be (1) $S[P_1[i]..P_1[i+1]] = S[P_1[j]..P_1[j+1]]$ and (2) $t[P_1[i]..P_1[i+1]] = t[P_1[j]..P_1[j+1]]$. Given (1) and (2) are satisfied, let $i' = P_1[i]+1$ and $j' = P_1[j]+1$, we have the below observations:

- Any in $S[i'..i'+d+1]$ is a LMS character. In this case, given $S_1[i] = S_1[j]$, we must have $P_1[i+1] = P_1[j+1]$.
- None in $S[i'..i'+d+1]$ is a LMS character. In this case, both $i'+d$ and $j'+d$ must be in P_1.

In either case, we have $P_1[i+1] - P_1[i] = P_1[j+1] - P[j]$.

Theorem 2. *(Order Preservation) The relative order of any two suffixes $suf(S_1, i)$ and $suf(S_1, j)$ in S_1 is the same as that of $suf(S, P_1[i])$ and $suf(S, P_1[j])$ in S.*

Proof. The proof is due to the following consideration on two cases:

– Case 1: $S_1[i] \neq S_1[j]$. This case can be further split into two cases in respect to whether the two critical substrings in S starting at $S[P_1[i]]$ and $S[P_1[j]]$ are equal or not. If they are different, the statement is obviously correct. If they are identical, we must have $t[P_1[i]+d+1] \neq t[P_1[j]+d+1]$ (or else we must have $S_1[i] = S_1[j]$), which implies that the statement is correct too.
– Case 2: $S_1[i] = S_1[j]$. In this case, the order of $suf(S_1, i)$ and $suf(S_1, j)$ is determined by the order of $suf(S_1, i+1)$ and $suf(S_1, j+1)$. The same argument can be recursively conducted on $S_1[i+1] = S_1[j+1]$, $S_1[i+2] = S_1[j+2]$,...$S_1[i+k-1] = S_1[j+k-1]$ until a k is reached that makes $S_1[i+k] \neq S_1[j+k]$. Because that $S_1[i..i+k-1] = S_1[j..j+k-1]$, from Theorem 1, we must have $P_1[i+k] - P_1[i] = P_1[j+k] - P_1[j]$, i.e., the substrings $S[P_1[i]..P_1[i+k]]$ and $S[P_1[j]..P_1[j+k]]$ are of the same length. This suggests that sorting $S_1[i..i+k]$ and $S_1[j..j+k]$ is equal to sorting $S[P_1[i]..P_1[i+k]+d+1]$ and $S[P_1[j]..P_1[j+k]+d+1]$. Hence, the statement is correct in this case, too.

This theorem suggests that in order to find the orders for all D-Critical suffixes in S, we can sort S_1 instead. Because S_1 is at least 1/2 smaller than S, the computation on S_1 can be done within about one half the complexity for S. In the following subsections, we show how to bucket sort and name the items of P_1, i.e. the two crucial subtasks of computing S_1.

3.4 Sorting and Naming P_1

To bucket sort and name all the items of P_1, intuitively, we need at least three integer arrays of at most $2n_1 + n$ integers in total: two size-n_1 used as the alternating buffers for bucket sorting P_1, and another size-n for the bucket pointers, where $2n_1 \leq n$. The array of bucket pointers needs a size of n because each character of P_1 is in the range $[0, n-1]$. The space needed for sorting P_1 constitutes the space bottleneck for our algorithm. To further improve the space efficiency, we can use the following γ-weighting scheme for bucket sorting P_1 instead.

Definition 6. *(γ-Weighted Substring) The γ-weighted substring $S_\gamma[i..j]$ in S is defined as $S_\gamma[i..j] = S[i..j-1]S_\omega[j]$.*

For any two γ-weighted substrings, we immediately have the below result.

Lemma 4. *Given $S_\gamma[i..i+k] < S_\gamma[j..j+k]$ and $S[i..i+k] = S[j..j+k]$, we must have $t(S, i+x) \leq t(S, j+x)$ for any $x \in [0, k]$.*

By replacing $S_\omega[i..j]$ with $S_\gamma[i..j]$ as the weight of $P_1[i]$ for bucket sorting P_1 to produce S_1, we have the following result.

Theorem 3. *(γ-Order Equivalence) (1) Given $S_\gamma[P_1[i]..P_1[i]+d+1] = S_\gamma[P_1[j]..P_1[j]+d+1]$, there must be $S_\omega[P_1[i]..P_1[i]+d+1] = S_\omega[P_1[j]..P_1[j]+d+1]$; and (2) Given $S_\gamma[P_1[i]..P_1[i]+d+1] < S_\gamma[P_1[j]..P_1[j]+d+1]$, there must be $S_\omega[P_1[i]..P_1[i]+d+1] < S_\omega[P_1[j]..P_1[j]+d+1]$.*

Proof. Let $i' = P_1[i]$ and $j' = P_1[j]$. If $S_\gamma[i'..i'+d+1] = S_\gamma[j'..j'+d+1]$, we must have $S[i'..i'+d+1] = S[j'..j'+d+1]$ and $t(S, i'+d+1) = t(S, j'+d+1)$, i.e., $S_\omega[i'..i'+d+1] = S_\omega[j'..j'+d+1]$. Further, if $S_\omega[i'+d+1] = S_\omega[j'+d+1]$ and $S[i'+d] = S[j'+d]$, we must have $t(S, i'+d) = t(S, j'+d)$ as well as $S_\omega(i'+d) = S_\omega(j'+d)$, and so on for the other characters in the two substrings. Therefore, we must have $S_\omega[i'..i'+d+1] = S_\omega[j'..j'+d+1]$. When $S_\gamma[i'..i'+d+1] < S_\gamma[j'..j'+d+1]$, we consider these two cases:

- If $S[i'..i'+d+1] \neq S[j'..j'+d+1]$, given $S_\gamma[i'..i'+d+1] < S_\gamma[j'..j'+d+1]$, there must be $S[i'..i'+d+1] < S[j'..j'+d+1]$ from the definition of γ-weighted substring (Definition 6), which immediately yields $S_\omega[i'..i'+d+1] < S_\omega[j'..j'+d+1]$ from the definition of S_ω.
- If $S[i'..i'+d+1] = S[j'..j'+d+1]$, we must have $t(S, i'+d+1) = 0$ and $t(S, j'+d+1) = 1$. Further, from Lemma 4, we have $t(S, i'+x) \leq t(S, j'+x)$ for any $x \in [0, d+1]$, resulting in $S_\omega[i'..i'+d+1] < S_\omega[j'..j'+d+1]$.

Hence, we complete the proof.

Theorem 3 suggests that, to determine the order of two ω-weighted D-Critical substrings, we can use their γ-weighted counterparts instead. As a result, we need to compare the characters' types only for the last characters of the D-Critical substrings. Therefore, sorting all the items of P_1 according to the last characters of their γ-weighted siblings can be decomposed into two passes sequentially: (1) bucket sort according to the types of these characters; and (2) bucket sort according to these characters themselves. Using this method, we only need an array of $\Sigma(S)$ or n_1 integers for maintaining the bucket information at the 1st or 2nd iterations, respectively.

Now, provided with P_1, t and S, we can compute S_1, i.e. the reduced problem, using the two-step algorithm described below.

- Step 1: Bucket sort all the elements of P_1 into another array P_1' by their corresponding siblings (i.e. fixed-size D-Critical substrings) in S, with $\Sigma(S)$ buckets. The sorting is done through $d+2$ passes, in a manner of least-significant-character-first. This step requires a time complexity of $O(dn_1) = O(n_1)$, for $d = O(1)$.
- Step 2: Compute the names for all elements in P_1' (as well as P_1). This job can be done by a simple algorithm described as following: (i) allocate a size-n array *tmp*, where each item is an integer in $[0, n-1]$; (ii) initialize all items of *tmp* to be -1; (iii) scan P_1' once from left to right to compute all the names for the items of P_1', by setting $tmp[P_1'[i]]$ with the index of bucket that $P_1'[i]$ belonging to; (iv) pack all non-negative elements in *tmp* into the buffer of P_1', by traversing *tmp* once. Now, the buffer of P_1' stores the string of S_1.

One problem with Step 2 in the above algorithm is that in addition to P_1' and S_1, it uses a large space of n integers (each integer is of $\lceil \log n \rceil$ bits) for *tmp*. Alternatively, we can use another space-efficient algorithm for this by reusing *tmp* for P'_1 and S_1, described as following. Let's define a logical array $\widetilde{tmp}_e = \{tmp[i] | i\%2 = 0\}$ for the first n_1 even items of *tmp*, where $\widetilde{tmp}_e$ is said to be a logical array for its physical buffer is distributed into the first n_1 even items of *tmp*, i.e., its physical buffer is not spatially continuous.

Suppose that P'_1 is initially stored in the first n_1 items of *tmp*, we first copy P'_1 into $\widetilde{tmp}_e$ and set $tmp[j] = -1$ for any $tmp[j] \notin \widetilde{tmp}_e$, i.e., distribute P'_1 into the first even items of *tmp*. Next, we scan $\widetilde{tmp}_e$ from left to right to compute the names for all the the items of $\widetilde{tmp}_e$. For each $\widetilde{tmp}_e[i]$, we record its name as following: (1) if $\widetilde{tmp}_e[i]$ is even, set $tmp[\widetilde{tmp}_e[i] - 1]$ with the name; or else set $tmp[\widetilde{tmp}_e[i]]$ with the name. Now, all the items of S_1 are stored in the non-negative odd items of *tmp* in their correct relative positional orders. Last, we traverse *tmp* once to compact all the non-negative odd items into S_1. Using this method for Step 2, *tmp* is reused for accommodating both P'_1 and S_1, resulting in that only one n-integer array is required for all of them.

3.5 Inducing SA from $SA(S_1)$

For denotation simplicity, let $SA_0 = SA(S)$ and $SA_1 = SA(S_1)$. Furthermore, let $SA_{lms} = \{SA_0[i] | \text{S[SA}_0\text{[i]] is a LMS character}\}$ and similarly $SA_l = \{SA_0[i] | \text{S[SA}_0\text{[i]] is a type} - \text{L character}\}$. From SA_1, we can derive SA_{lms} and then induce SA_l from SA_{lms} and SA_0 from SA_l, as described below. The algorithm for inducing SA_0 from SA_1 consists of four sequential stages in $O(n)$ time/space:

1. Initialization: (1) Set all the items of SA_0 to be negative. (2) Scan S at most twice to find the buckets in SA_0 for all the suffixes in S according to their first characters.
2. Deriving SA_{lms} from SA_1: (1) Initialize all the buckets in SA_0 as empty by setting the start of each bucket as its end. (2) Scan SA_1 once from right to left, if $S[P_1[SA_1[i]]]$ is a LMS character then put $P_1[SA_1[i]]$ to the current start of its bucket in SA_0 and move the bucket start one item to the left.
3. Inducing SA_l from SA_{lms}: (1) For each bucket in SA_0, set the end as its start. (2) Scan SA_0 from left to right, for each non-negative item $SA_0[i]$, if $S[SA_0[i] - 1]$ is type-L then put $SA_0[i] - 1$ to the current end of its bucket and move the bucket end one item to the right.
4. Inducing SA from $SA_l(S)$: (1) For each bucket in SA_0, set the start as its end. (2) Scan SA_0 from right to left, for each item $SA_0[i]$, if $S[SA_0[i] - 1]$ is type-S then put $SA_0[i] - 1$ to the current start of its bucket and move the bucket start one item to the left.

In the above algorithm, in addition to SA_0, we need another array *bkt* for maintaining the start/end of each bucket on-the-fly in each stage, where *bkt* has $\|\Sigma(S)\|$ items and each item is of $\lceil \log n \rceil$ bits.

We now consider the correctness of the inducing algorithm for the stages 2-4. The correctness of stage 4, i.e. inducing SA from $SA_l(S)$, has been proven in [6] (Lemma 3), which is quoted and re-stated in this paper's terms as below.

Lemma 5. [6] *Given all the type-L (or type-S) suffixes of S sorted, all the suffixes of S can be sorted in $O(n)$ time.*

From the above lemma, it is trivial for us to have the below corollary for our previous work [10]. This corollary supports the correctness of stage 3 and restated here.

Corollary 1. *Given all the LMS suffixes of S sorted, all the type-L suffixes of S can be sorted in $O(n)$ time.*

Proof. From Lemma 5, we know that given SA_s, we can induce SA as well as SA_l in $O(n)$ time, by traversing SA once from left to right. Notice that not every type-S suffix is needed for sorting SA_l (here we use SA_l to denote all type-L suffixes); instead a type-S suffix is needed only when it is also a LMS suffix. In order words, knowing the order of all the LMS suffixes, we can traverse once from left to right to populate the order of SA_l in $O(n)$ time.

Given SA_1, the correctness of stage 2 is obvious, for we just simply copy all LMS items of SA_1 into the ends of their corresponding buckets (notice that in a bucket, a type-L suffix is less than a type-S suffix), keeping their relative orders unchanged.

4 Practical Strategies

We propose some techniques to further improve the time/space efficiencies of our algorithm in practice. Without loss of generality, we assume a 32-bit machine and each integer consumes 4 bytes.

General Strategy: Reusing the Buffer for $SA(S)$

From our algorithm framework in Fig.1, we see that the algorithm consists of three steps in sequence: (1) sorting P_1; (2) naming P_1 to obtain S_1; and (3) inducing $SA(S)$ from $SA(S_1)$. Notice that $SA(S)$ is an array of n integers, and both P_1 and S_1 have n_1 integers, where $2n_1 \leq n$, we can re-use the buffer for $SA(S)$ for the steps (1) and (2) too. For more details, the reader is referred to the sample codes in the appendix.

Strategy 1: Storing the LS-Type Array

Each element of the LS-type array for S is one-bit and a total of at most $n(1 + 1/2 + 1/4 + ... + \log^{-1} n) < 2n$ bits are required by the LS-arrays for all recursions. Hence, we can use the two most-significant-bits (MSBs) of $SA(S)[i]$ for storing the LS-type of $S[i]$. Recalling that the space for each integer is allocated in units of 4-byte instead of bits, the two MSBs of an integer is always available for us in this case. This is because that to compute $SA(S)$, our algorithm running on a 32-bit machine requires at least $5n$ bytes, where $4n$ for the items (each is a 4-byte integer) in $SA(S)$ and n for the input string (usually one byte per character). Therefore, the maximum size n_{max} of the input string must satisfy $5n_{max} < 2^{32}$, resulting in $n_{max} < 2^{32}/5$ and $\log n_{max} < 30$. In order words, 30 bits are enough for each item of $SA(S)$. However, for implementation convenience, we can simply store the LS-type arrays using bit arrays of maximum $2n$ bits in total, i.e. $0.25n$ bytes.

Strategy 2: Bucket Sorting P_1

Given the buffers for P_1 and S_1, to bucket sort P_1, we can use another array *bkt* in Fig. 1 for maintaining the buckets, where the size of *bkt* is determined by the alphabet size of the input string S. Even the original input string S is of a constant alphabet, after the first iteration, we will have S_1 as the input S for the next iteration. Since S_1 has an integer alphabet that can be as large as n_1 in the worst case, *bkt* may require a maximum space up to $n_1 \leq \lfloor n/2 \rfloor$ integers. To prevent *bkt* from growing with n_1, instead of sorting characters—each character is of 4 bytes—in each pass of bucket sorting the d-critical substrings, we simply sort each character with two passes, i.e., the bucket sorting is performed on units of 2-byte. The time complexity for bucket sorting all the fixed-size d-critical substrings at each iteration is linear proportional to the total number of characters for these substrings. Since each d-critical substring is of fixed-size $d+2$ characters and the number of substrings decreases at least half per iteration, the total number of characters sorted at each iteration is upper bounded by $O((d+2)(1/2+1/4+...+\log^{-1} n)) = O(dn)$, which is $O(n)$ given $d = O(1)$. Hence, the time complexity for bucket sorting in this way remains linear $O(n)$. For $n \leq 2^{32}$, the entire bucket sorting process will be half slowed down. However, the space for *bkt* can be fixed to 65536 integers, i.e. $O(1)$.

Strategy 3: Inducing the Final Result

In the inducing algorithm we described before, a buffer *bkt* is needed for dynamically recording the current start/end of each bucket. However, in order to save more space, we can use an alternative inducing algorithm which requires only the buffer for $SA(S_1)$ and needs no *bkt* when inducing $SA(S_1)$. This idea is to name the elements of P_1 in a different way: once all the items of P_1 have been sorted into their buckets, we can name each item of P_1 by the end of of its bucket to produce S_1. To be more precise, this is because the MSB of each item in SA_1 and S_1 is unused (when the strategy 1 is not applied). Given that each item of S_1 points to the end of its bucket in SA_1, the inducing can be done in this way: when an empty bucket in SA_1 is inserted the first item $S_1[i]$ at $SA_1[j]$, we set $SA_1[j] = i$ and $S_1[i] = j$, and mark the MSBs of $SA_1[j]$ and $S_1[i]$ by 1 to indicate they are borrowed for maintaining the bucket end. At the end of each inducing stage, we can restore the items in S_1 and SA_1 to their correct values in this way: scan SA_1 from left to right, for each $SA_1[i]$ with its MSB as 1, let $S_1[SA[i]] = i$ and and reset the MSBs of $SA_1[i]$ and $S_1[SA_1[i]]$ as 0.

5 Main Results

Theorem 4. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and the space complexities for the algorithm DCS in Fig. 1 to compute $SA(S)$ are $O(n)$ and $O(n\lceil\log n\rceil)$ bits, respectively.*

Proof. Because the problem is reduced at least 1/2 at each recursion, we have the time complexity governed by Eq. 1, where the reduced problem is of size at most $\lfloor n/2\rfloor$. The first $O(n)$ in the equation accounts for reducing the problem and inducing the final solution from the reduced problem.

$$\mathcal{T}(n) = \mathcal{T}(\lfloor n/2\rfloor) + O(n) = O(n) \qquad (1)$$

The space complexity is obvious $O(n\lceil\log n\rceil)$ bits, for the size of each array used at the first iteration is upper bounded by $n\lceil\log n\rceil$ bits, and decreases at least a half for each iteration thereafter.

Theorem 5. *The DCS algorithm can construct the suffix array for a size-n string with a constant or integer alphabet using $O(n)$ time and a working space of only $0.25n + O(1)$ bytes.*

Proof. (Sketch) The key technique is to design the algorithm DCS with the general strategy and the strategies 2-3 in Section 4, the details of which are omitted here due to the limited space. We have coded in C++ a sample implementation for this (i.e. the DCS2 algorithm in the experiment section). Interested readers are welcome to contact us for the details/codes.

In [2], a working space of $0.03n$ bytes was reported for the algorithm proposed by Manzini and Ferragina, as a statistical result from experiments. Compared to this, our worst-case result of $0.25n + O(1)$ is approaching the existing best result for working space required by suffix array construction algorithms.

6 Experiments

For comparison convenience, we adopt some data sets from the Canterbury and the Manzini-Ferragina [2] corpora that are widely used for performance evaluations of various suffix array algorithms. All the input strings are of constant alphabets smaller than 256, and one byte is consumed by each character. The experiments were performed on a machine with AMD Athlon(tm) 64x2 Dual Core Processor 4200+ 2.20GHz and 2.00GB RAM, and the operating system is Linux (Sabayon Linux distribution).

The algorithms investigated in our experiments are limited to *linear* algorithms only, i.e. DCS1, DCS2, KS and KA, where DCS1 and DCS2 are the DCS algorithm with $d = 3$ and enhanced by the practical strategies proposed in Section 4. Specifically, the algorithms DCS1 and DCS2 use different settings of strategies: DCS1 uses the general strategy only, while DCS2 uses the strategies 2-3 in addition to the general strategy. The KS algorithm was downloaded

Table 1. Data Used in the Experiments

Data	$\|\Sigma\|$	Characters	Description
world192.txt	94	2 473 400	CIA world fact book
bible.txt	63	4 047 392	King James Bible
chr22.dna	4	34 553 758	Human chromosome 22
E.coli	4	4 638 690	Escherichia coli genome
sprot34.dat	66	109 617 186	Swissprot V34 protein database
etext99	146	105 277 340	Texts from Gutenberg project
howto	197	39 422 105	Linux Howto files

from Sanders's website [11], and the KA algorithm was downloaded from Ko's website [12]. All the programs were compiled by *g++* with the -O3 option. The performance measurements to be investigated are: the time/space complexities, the recursion depth and the mean reduction ratio. The time for each algorithm is the mean of 3 runs; and the space is the heap peak measured by using the *memusage* command to fire the running of each program. The total time (in seconds) and space (in million bytes, MBytes) for each algorithm are the sums of the times and spaces consumed by running the algorithm for all the input data, respectively. The mean time (measured in seconds per MBytes) and space (in bytes per character of the input string) for each algorithm are the total time and space divided by the total number of characters in all input data. The recursion depth is defined as the number of iterations, and the mean reduction ratio is the sum of reduction ratios for all iterations divided by the recursion depth.

Table 2. Time, Space, Recursion Depth and Reduction Ratio

Data	Time (Seconds)				Space (MBytes)				Recursion Depth			Reduction Ratio		
	DCS1	DCS2	KS	KA	DCS1	DCS2	KS	KA	DCS	KS	KA	DCS	KS	KA
world	*1.61*	2	4.8	1.9	12.91	*12.50*	55.24	21.24	7	*6*	*6*	*.37*	.67	.42
bible	*3.11*	3.9	8.9	3.51	21.50	*20.30*	90.40	34.45	*6*	*6*	*6*	*.37*	.67	.45
chr22	*31.5*	39.6	92.8	33.41	184.44	*171.41*	819.25	289.97	10	12	*8*	*.36*	.67	.43
E.coli	*3.53*	4.3	10	3.98	25.15	*23.23*	105.93	40.01	8	*7*	8	*.36*	.67	.42
sprot	*111.59*	139.6	356	132.89	560.44	*543.26*	2591.62	930.06	*8*	9	9	*.37*	.67	.45
etext	*123.2*	150.4	428.1	149.67	559.55	*521.85*	2369.92	907.34	*12*	*12*	*12*	*.37*	.67	.45
howto	*36.3*	44.05	130.4	42.85	208.08	*195.55*	932.07	331.54	*10*	11	13	*.36*	.67	.45
Total	*310.84*	383.85	1031	368.21	1572.07	*1488.08*	6964.44	2554.61	*61*	63	62	-	-	-
Mean	*1.09*	1.34	3.60	1.29	5.49	*5.20*	24.34	8.93	*8.71*	9.0	8.85	*0.366*	0.67	0.438
Norm.	*1*	1.23	3.32	1.18	1	*0.95*	4.43	1.63	*1*	1.03	1.02	*1*	1.83	1.19

We show in Table 2 the statistic results collected from the experiments, where the best results are styled in the italic fonts. For comparison convenience, we normalize all the results by that from the DCS1 algorithm. In the program for the KS algorithm, each character of the input string S is stored as a 4-byte integer, and the buffer for $SA(S)$ is not reused for the others. To be fair, we subtract $7n$ bytes from the space results we measured for the KS algorithm in the experiments, for we are sure $7n$ space can be trivially saved using some engineering tricks. From these results, we see that the best time and space performances are achieved by our algorithms DCS1 and DCS2, respectively. Specifically, the DCS1 algorithm is the fastest, which in average is more than twice (232%) faster than the KS, and 9% faster than the KA. The best space performance is achieved by the DCS2, which is 23% slower than the DCS1, however, still more than 1.5 times ((3.32-1.23)/1.23=169%) faster than the KS algorithm. The mean space of $24.34n$ for the KS algorithm in our experiments is about twice of the 10-$13n$ for another space efficient implementation of the KS algorithm by Puglisi [13]. Even suppose the better 10-$13n$ space, the KS algorithm still uses a space more than twice of that for our algorithms DCS1 and DCS2. Similar to the observations from the others [13,14], the KA algorithm in our experiments is more space efficient than the KS algorithm, however, which uses about 60-70% more space than our algorithms DCS1 and DCS2, respectively.

The results for recursion depths and reduction ratios are machine-independent and deterministic for each given input string. Obviously, the smaller the reduction ratio is, the faster the algorithm is. For an overall comparison, we also give the total for the recursion depth and the reduction ratio for each algorithm and the mean for both, where the former is the sum of all corresponding results and the later is the former divided by the number of individual input data, i.e. 7 for the DCS and the KS algorithms and 5 for the KA algorithm in this case, respectively in this case. In this table, clearly, the DCS algorithm achieves the best reduction ratio for all the input data. The DCS algorithm has a mean reduction ratio about a half ((0.67-0.366)/0.67=45%) smaller than that of the KS algorithm. This well coincides with their time results. where the DCS1 algorithm is more than twice faster than the KS algorithm.

7 Closing Remarks

In this paper, we have analyzed the D-Critical substring algorithm [9], a new linear time/space SACA which uses a novel concept called D-Critical Substrings to encode the original text. The implementation of our algorithm is as intuitive as that of the KS algorithm and our algorithm runs even faster than the KA algorithm. The complete C++ source code (around 100+ effective lines) of the sample implementation of the proposed algorithms used in our experiments is available upon readers' request.

Recently, after the development of the D-Critical-Substring algorithm, we proposed another SA algorithm using the almost pure induced sorting technique (i.e. the SA-IS algorithm in [15]), and both algorithms are running in linear time. Currently, the two algorithms have been investigated under the assumption that the whole input string is completely stored in the main memory. However, as we have noticed that, there are increasing needs for building the SAs of huge corpora, e.g., the ever growing genome databases. When the input string demands more space than the main memory can provide, we have to seek for help from using the slower but far more larger external memory such as flash memory or harddisk. Our recent study on using external memory for our proposed linear SA algorithms indicated that, compared with the SA-IS algorithm, the D-Critical-Substring algorithm has greater potentials to be improved for using external memory, due to its distinct advantage of sorting fixed-size d-critical substrings, in contrast to sorting variable-size substrings in the SA-IS algorithm.

References

1. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms, pp. 319–327 (1990)
2. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica 40(1), 33–50 (2004)
3. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: The 32nd Annual ACM Symposium on Theory of Computing (STOC 2000), pp. 397–406 (2000)
4. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
5. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
6. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
7. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS 1997: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997), p. 137 (1997)
8. Mori, Y.: divsufsort (2007), http://homepage3.nifty.com/wpage/software/libdivsufsort.html
9. Zhang, S., Nong, G.: Fast and space efficient linear suffix array construction. In: IEEE Data Compression Conference 2008 (DCC 2008), p. 553 (2008)
10. Nong, G., Zhang, S.: Optimal lightweight construction of suffix arrays. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 613–624. Springer, Heidelberg (2007)
11. Sanders, P.: A driver program for the KS algorithm (2007), http://www.mpi-inf.mpg.de/~sanders/programs/suffix/
12. Ko, P.: Source codes for the KA algorithm (2007), http://kopang.public.iastate.edu/homepage.php?page=source
13. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. 39(2), 1–31 (2007)
14. Lee, S., Park, K.: Efficient implementations of suffix array construction algorithms. In: Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms, pp. 64–72 (2004)
15. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: IEEE Data Compression Conference 2009 (DCC 2009), pp. 193–202 (2009)

On the Value of Multiple Read/Write Streams for Data Compression

Travis Gagie

University of Eastern Piedmont
Alessandria, Italy
travis@mfn.unipmn.it

Abstract. We study whether, when restricted to using polylogarithmic memory and polylogarithmic passes, we can achieve qualitatively better data compression with multiple read/write streams than we can with only one. We first show how we can achieve universal compression using only one pass over one stream. We then show that one stream is not sufficient for us to achieve good grammar-based compression. Finally, we show that two streams are necessary and sufficient for us to achieve entropy-only bounds.

1 Introduction

Massive datasets seem to expand to fill the space available and, in situations when they no longer fit in memory and must be stored on disk, we may need new models and algorithms. Grohe and Schweikardt [15] introduced read/write streams to model situations in which we want to process data using mainly sequential accesses to one or more disks. As the name suggests, this model is like the streaming model (see, e.g., [22]) but, as is reasonable with datasets stored on disk, it allows us to make multiple passes over the data, change them and even use multiple streams (i.e., disks). As Grohe and Schweikardt pointed out, sequential disk accesses are much faster than random accesses — potentially bypassing the von Neumann bottleneck — and using several disks in parallel can greatly reduce the amount of memory and the number of accesses needed. For example, when sorting, we need the product of the memory and accesses to be at least linear when we use one disk [21,14] but only polylogarithmic when we use two [7,15]. Similar bounds have been proven for a number of other problems, such as checking set disjointness or equality; we refer readers to Schweikardt's survey [26] of upper and lower bounds with one or more read/write streams, Heinrich and Schweikardt's recent paper [17] relating read/write streams to classical complexity theory, and Beame and Huỳnh-Ngọc's recent paper [3] on the value of multiple read/write streams for approximating frequency moments.

Since sorting is an important operation in some of the most powerful data compression algorithms, and compression is an important operation for reducing massive datasets to a more manageable size, we wondered whether extra streams could also help us achieve better compression. In this paper we consider the problem of compressing a string s of n characters over an alphabet of size σ when we are restricted to using $\log^{\mathcal{O}(1)} n$ bits of memory and $\log^{\mathcal{O}(1)} n$ passes over the data. In Section 2, we show how we can achieve universal compression using only one pass over one stream. Our approach is to break the string into blocks and compress each block separately, similar to what is done in practice to compress large files. Although this may not usually significantly worsen the compression itself, it may stop us from then building a fast compressed index [11] (unless we somehow combine the indexes for the blocks) or clustering by compression [8] (since concatenating files should not help us compress them better if we then break them into pieces again). In Section 3 we use a vaguely automata-theoretic argument to show one stream is not sufficient for us to achieve good grammar-based compression. Of course, by 'good' we mean here something stronger than universal compression: we want the size of our encoding to be at most polynomial in the size of the smallest context-free grammar than generates s and only s. We still do not know whether any constant number of streams is sufficient for us to achieve such compression. Finally, in Section 4 we show that two streams are necessary and sufficient for us to achieve entropy-only bounds. Along the way, we show we need two streams to find strings' minimum periods or compute the Burrows-Wheeler Transform. As far as we know, this is the first paper on compression with read/write streams, and among the first papers on compression in any streaming model; we hope the techniques we have used will prove to be of independent interest.

2 Universal Compression

An algorithm is called universal with respect to a class of sources if, when a string is drawn from any of those sources, the algorithm's redundancy per character approaches 0 with probability 1 as the length of the string grows. The class most often considered, and which we consider in this section, is that of stationary, ergodic Markov sources (see, e.g., [9]). Since the kth-order empirical entropy $H_k(s)$ of s is the minimum self-information per character of s with respect to a kth-order Markov source (see [25]), an algorithm is universal if it stores any string s in $nH_k(s) + o(n)$ bits for any fixed σ and k. The kth-order empirical entropy of s is also our expected uncertainty about a randomly chosen character of s when given the k preceding characters. Specifically,

$$H_k(s) = \begin{cases} (1/n) \sum_a \mathsf{occ}(a,s) \log \frac{n}{\mathsf{occ}(a,s)} & \text{if } k = 0, \\ (1/n) \sum_{|w|=k} |w_s| H_0(w_s) & \text{otherwise,} \end{cases}$$

where $\mathsf{occ}(a,s)$ is the number of times character a occurs in s, and w_s is the concatenation of those characters immediately following occurrences of k-tuple w in s.

In a previous paper [13] we showed how to modify the well-known LZ77 compression algorithm [27] to use sublinear memory while still storing s in $nH_k(s) + \mathcal{O}(n \log\log n / \log n)$ bits for any fixed σ and k. Our algorithm uses nearly linear memory and so does not fit into the model we consider in this paper, but we mention it here because it fits into some other streaming models (see, e.g., [22]) and, as far as we know, was the first compression algorithm to do so. In the same paper we proved several lower bounds using ideas that eventually led to our lower bounds in Sections 3 and 4 of this paper.

Theorem 1 (Gagie and Manzini, 2007). *We can achieve universal compression using one pass over one stream and $\mathcal{O}\big(n/\log^2 n\big)$ bits of memory.*

To achieve universal compression with only polylogarithmic memory, we use a recent algorithm due to Gupta, Grossi and Vitter [16]. Although they designed it for the RAM model, we can easily turn it into a streaming algorithm by processing s in small blocks and compressing each block separately.

Theorem 2 (Gupta, Grossi and Vitter, 2008). *In the RAM model, we can store any string s in $nH_k(s) + \mathcal{O}\big(\sigma^k \log n\big)$ bits, for all k simultaneously, using $\mathcal{O}(n)$ time.*

Corollary 1. *We can achieve universal compression using one pass over one stream and $\mathcal{O}\big(\log^{1+\epsilon} n\big)$ bits of memory.*

Proof. We process s in blocks of $\lceil \log^\epsilon n \rceil$ characters, as follows: we read each block into memory, apply Theorem 2 to it, output the result, empty the memory, and move on to the next block. (If n is not given in advance, we increase the block size as we read more characters.) Since Gupta, Grossi and Vitter's algorithm uses $\mathcal{O}(n)$ time in the RAM model, it uses $\mathcal{O}(n \log n)$ bits of memory and we use $\mathcal{O}\big(\log^{1+\epsilon} n\big)$ bits of memory. If the blocks are $s_1, \ldots, s_b$, then we store all of them in a total of

$$\sum_{i=1}^{b} \big(|s_i| H_k(s_i) + \mathcal{O}\big(\sigma^k \log\log n\big)\big) \leq nH_k(s) + \mathcal{O}\big(\sigma^k n \log\log n / \log^\epsilon n\big)$$

bits for all k simultaneously. Therefore, for any fixed σ and k, we store s in $nH_k(s) + o(n)$ bits. $\Box$

A bound of $nH_k(s) + \mathcal{O}\big(\sigma^k n \log\log n / \log^\epsilon n\big)$ bits is not very meaningful when k is not fixed and grows as fast as $\log\log n$, because the second term is $\omega(n)$. Notice, however, that Gupta *et al.*'s bound of $nH_k(s) + \mathcal{O}\big(\sigma^k \log n\big)$ bits is also not very meaningful when $k \geq \log n$, for the same reason. As we will see in Section 4, it is possible for s to be fairly incompressible but still to have $H_k(s) = 0$ for $k \geq \log n$. It follows that, although we can prove bounds that hold for all k simultaneously, those bounds cannot guarantee good compression in terms of $H_k(s)$ when $k \geq \log n$.

By using larger blocks — and, thus, more memory — we can reduce the $\mathcal{O}\big(\sigma^k n \log\log n / \log^\epsilon n\big)$ redundancy term in our analysis, allowing k to grow faster than $\log\log n$ while still having a meaningful bound. In the full version of this paper we will prove the resulting tradeoff is nearly optimal. Specifically, using an argument similar to those we use to prove the lower bounds in Sections 3 and 4, we will prove that the product of the memory, passes and redundancy must be nearly linear in n. It is not clear to us, however, whether we can modify Corollary 1 to take advantage of multiple passes.

Open Problem 1. *With multiple passes over one stream, can we achieve better bounds on the memory and redundancy than we can with one pass?*

3 Grammar-Based Compression

Charikar *et al.* [6] and Rytter [24] independently showed how to build a context-free grammar APPROX that generates s and only s and is an $\mathcal{O}(\log n)$ factor larger than the smallest such grammar OPT, which is $\Omega(\log n)$ bits in size.

Theorem 3 (Charikar *et al.*, 2005; Rytter, 2003). *In the RAM model, we can approximate the smallest grammar with* $|\mathsf{APPROX}| = \mathcal{O}\big(|\mathsf{OPT}|^2\big)$ *using* $\mathcal{O}(n)$ *time.*

In this section we prove that, if we use only one stream, then in general our approximation must be superpolynomially larger than the smallest grammar. Our idea is to show that periodic strings whose periods are asymptotically slightly larger than the product of the memory and passes, can be encoded as small grammars but, in general, cannot be compressed well by algorithms that use only one stream. Our argument is based on the following two lemmas.

Lemma 1. *If s has period ℓ, then the size of the smallest grammar for that string is $\mathcal{O}(\ell + \log n \log\log n)$ bits.*

Proof. Let t be the repeated substring and t' be the proper prefix of t such that $s = t^{\lfloor n/\ell \rfloor} t'$. We can encode a unary string $X^{\lfloor n/\ell \rfloor}$ as a grammar G_1 with $\mathcal{O}(\log n)$ productions of total size $\mathcal{O}(\log n \log\log n)$ bits. We can also encode t and t' as grammars G_2 and G_3 with $\mathcal{O}(\ell)$ productions of total size $\mathcal{O}(\ell)$ bits. Suppose S_1, S_2 and S_3 are the start symbols of G_1, G_2 and G_3, respectively. By combining those grammars and adding the productions $S_0 \rightarrow S_1 S_3$ and $X \rightarrow S_2$, we obtain a grammar with $\mathcal{O}(\ell + \log n)$ productions of total size $\mathcal{O}(\ell + \log n \log\log n)$ bits that maps S_0 to s. □

Lemma 2. *Consider a lossless compression algorithm that uses only one stream, and a machine performing that algorithm. We can compute any substring from*

- *its length;*
- *for each pass, the machine's memory configurations when it reaches and leaves the part of the stream that initially holds that substring;*
- *all the output the machine produces while over that part.*

Proof. Let t be the substring and assume, for the sake of a contradiction, that there exists another substring t' with the same length that takes the machine between the same configurations while producing the same output. Then we can substitute t' for t in s without changing the machine's complete output, contrary to our specification that the compression be lossless. □

Lemma 2 implies that, for any substring, the size of the output the machine produces while over the part of the stream that initially holds that substring, plus twice the product of the memory and passes (i.e., the number of bits needed to store the memory configurations), must be at least that substring's complexity. Therefore, if a substring is not compressible by more than a constant factor (as is the case for most strings) and asymptotically larger than the product of the memory and passes, then the size of the output for that substring must be at least proportional to the substring's length. In other words, the algorithm cannot take full advantage of similarities between substrings to achieve better compression. In particular, if s is periodic with a period that is asymptotically slightly larger than the product of the memory and passes, and s's repeated substring is not compressible by more than a constant factor, then the algorithm's complete output must be $\Omega(n)$ bits. By Lemma 1, however, the size of the smallest grammar that generates s and only s is bounded in terms of the period.

Theorem 4. *With one stream, we cannot approximate the smallest grammar with* $|\mathsf{APPROX}| \leq |\mathsf{OPT}|^{\mathcal{O}(1)}$.

Proof. Suppose an algorithm uses only one stream, m bits of memory and p passes to compress s, with $mp = \log^{\mathcal{O}(1)} n$, and consider a machine performing that algorithm. Furthermore, suppose s is periodic with period $\lceil mp \log n \rceil$ and its repeated substring t is not compressible by more than a constant factor. Lemma 2 implies that the machine's output while over a part of the stream that initially holds a copy of t, must be $\Omega(mp \log n - mp) = \Omega(mp \log n)$. Therefore, the machine's complete output must be $\Omega(n)$ bits. By Lemma 1, however, the size of the smallest grammar that generates s and only s is $\mathcal{O}(mp \log n + \log n \log \log n) \subset \log^{\mathcal{O}(1)} n$ bits. Since $n = \log^{\omega(1)} n$, the algorithm's complete output is superpolynomially larger than the smallest grammar. □

As an aside, we note that a symmetric argument shows that, with only one stream, in general we cannot decode a string encoded as a small grammar. To prove this, instead of considering a part of the stream that initially holds a copy of the repeated substring t, we consider a part that is initially blank and eventually holds a copy of t. We can compute t from the machine's memory configurations when it reaches and leaves that part, so the product of the memory and passes must be greater than or equal to t's complexity. Also, we note that Theorem 4 has the following corollary, which may be of independent interest.

Corollary 2. *With one stream, we cannot find strings' minimum periods.*

Proof. Consider the proof of Theorem 4. If we could find s's minimum period, then we could store s in $\log^{\mathcal{O}(1)} n$ bits by writing n and one copy of its repeated substring t. □

In fact, a more careful argument shows we cannot even check whether a string has a given period. In the full version of this paper, we will give proper proofs of lower bounds for decoding grammars and checking periodicities.

Unfortunately, as we noted in the introduction, our results for this section are still incomplete, as we do not know whether multiple streams are helpful for grammar-based compression.

Open Problem 2. *With $\mathcal{O}(1)$ streams, can we approximate the smallest grammar well?*

4 Entropy-Only Bounds

Kosaraju and Manzini [19] pointed out that proving an algorithm universal does not necessarily tell us much about how it behaves on low-entropy strings. In other words, showing that an algorithm encodes s in $nH_k(s) + o(n)$ bits is not very informative when $nH_k(s) = o(n)$. For example, although the well-known LZ78 compression algorithm [28] is universal, $|\mathsf{LZ78}(1^n) = \Omega(\sqrt{n})$ while $nH_0(1^n) = 0$. To analyze how algorithms perform on low-entropy strings, we would like to get rid of the $o(n)$ term and prove bounds that depend only on $nH_k(s)$. Unfortunately, this is impossible since, as the example above shows, even $nH_0(s)$ can be 0 for arbitrarily long strings.

It is not hard to show that only unary strings have $H_0(s) = 0$. For $k \geq 1$, recall that $H_k(s) = (1/n) \sum_{|w|=k} |w_s| H_0(w_s)$. Therefore, $H_k(s) = 0$ if and only if each distinct k-tuple w in s is always followed by the same distinct character. This is because, if a w is always followed by the same distinct character, then w_s is unary, $H_0(w_s) = 0$ and w contributes nothing to the sum in the formula. Manzini [20] defined the kth-order modified empirical entropy $H_k^*(s)$ such that each context w contributes at least $\lfloor \log |w_s| \rfloor + 1$ to the sum. Because modified empirical entropy is more complicated than empirical entropy — e.g., it allows for variable-length contexts — we refer readers to Manzini's paper for the full definition. In our proofs in this paper, we use only the fact that

$$nH_k(s) \leq nH_k^*(s) \leq nH_k(s) + \mathcal{O}\big(\sigma^k \log n\big) \ .$$

Manzini showed that, for some algorithms and all k simultaneously, it is possible to bound the encoding's length in terms of only $nH_k^*(s)$ and a constant that depends only on σ and k; he called such bounds 'entropy-only'. In particular, he showed that an algorithm based on the Burrows-Wheeler Transform (BWT) [5] stores any string s in at most $(5 + \epsilon)nH_k^*(s) + \log n + g_k$ bits for all k simultaneously (since $nH_k^*(s) \geq \log(n - k)$, we could remove the $\log n$ term by adding 1 to the coefficient $5 + \epsilon$).

Theorem 5 (Manzini, 2001). *Using the BWT, move-to-front coding, run-length coding and arithmetic coding, we can achieve an entropy-only bound.*

The BWT sorts the characters in a string into the lexicographical order of the suffixes that immediately follow them. When using the BWT for compression, it is customary to append a special character \$ that is lexicographically less than any in the alphabet. For a more thorough description of the BWT, we again refer readers to Manzini's paper. In this section we first show how we can compute and invert the BWT with two streams and, thus, achieve entropy-only bounds. We then show that we cannot achieve entropy-only bounds with only one stream. In other words, two streams are necessary and sufficient for us to achieve entropy-only bounds.

One of the most common ways to compute the BWT is by building a suffix array. In his PhD thesis, Ruhl introduced the StreamSort model [23,1], which is similar to the read/write streams model with one stream, except that it has an extra primitive that sorts the stream in one pass. Among other things, he showed how to build a suffix array efficiently in this model.

Theorem 6 (Ruhl, 2003). *In the StreamSort model, we can build a suffix array using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(\log n)$ passes.*

Corollary 3. *With two streams, we can compute the BWT using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

Proof. We can compute the BWT in the StreamSort model by appending \$ to s, building a suffix array, and replacing each value i in the array by the $(i-1)$st character in s (replacing either 0 or 1 by \$, depending on where we start counting). This takes $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(\log n)$ passes. Since we can sort with two streams using $\mathcal{O}(\log n)$ bits memory and $\mathcal{O}(\log n)$ passes (see, e.g., [26]), it follows that we can compute the BWT using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes. $\square$

Now suppose we are given a permutation π on $n+1$ elements as a list $\pi(1), \ldots, \pi(n+1)$, and asked to rank it, i.e., to compute the list $\pi^0(1), \ldots, \pi^n(1)$. This problem is a special case of list ranking (see, e.g., [2]) and has a surprisingly long history. For example, Knuth [18, Solution 24] described an algorithm, which he attributed to Hardy, for ranking a permutation with two tapes. More recently, Bird and Mu [4] showed how to invert the BWT by ranking a permutation. Therefore, reinterpreting Hardy's result in terms of the read/write streams model gives us the following bounds.

Theorem 7 (Hardy, c. 1967). *With two streams, we can rank a permutation using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

Corollary 4. *With two streams, we can invert the BWT using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

Proof. The BWT has the property that, if a character is the ith in $\mathsf{BWT}(s)$, then its successor in s is the lexicographically ith in $\mathsf{BWT}(s)$ (breaking ties by order of appearance). Therefore, we can invert the BWT by replacing each character by its lexicographic rank, ranking the resulting permutation, replacing each value i by the ith character of $\mathsf{BWT}(s)$, and rotating the string until \$ is at the end. This takes $\mathcal{O}(\log n)$ memory and $\mathcal{O}\big(\log^2 n\big)$ passes. $\square$

Since we can compute and invert move-to-front, run-length and arithmetic coding using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(1)$ passes over one stream, by combining Theorem 5 and Corollaries 3 and 4 we obtain the following theorem.

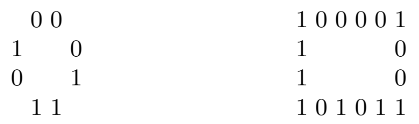


Fig. 1. Examples of 3rd-order and 4th-order De Bruijn cycles

Theorem 8. *With two streams, we can achieve an entropy-only bound using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}\big(\log^2 n\big)$ passes.*

To show we need at least two streams to achieve entropy-only bounds, we use De Bruijn cycles in a proof similar to the one for Theorem 4. A kth-order De Bruijn cycle [10] is a cyclic sequence in which every possible k-tuple appears exactly once. For example, Figure 1 shows a 3rd-order and a 4th-order De Bruijn cycle. (In this paper we need consider only binary De Bruijn cycles.) Our argument this time is based on Lemma 2 and the following results about De Bruijn cycles.

Lemma 3. *If $s \in d^*$ for some kth-order De Bruijn cycle d, then $nH_k^*(s) = \mathcal{O}\big(2^k \log n\big)$.*

Proof. By definition, each distinct k-tuple is always followed by the same distinct character; therefore, $nH_k(s) = 0$ and $nH_k^*(s) = \mathcal{O}\big(2^k \log n\big)$. □

Theorem 9 (De Bruijn, 1946). *There are $2^{2^{k-1}-k}$ kth-order De Bruijn cycles.*

Corollary 5. *We cannot store most kth-order De Bruijn cycles in $o(2^k)$ bits.*

Since there are 2^k possible k-tuples, kth-order De Bruijn cycles have length 2^k, so Corollary 5 means that we cannot compress most De Bruijn cycles by more than a constant factor. Therefore, we can prove a lower bound similar to Theorem 4 by supposing that s's repeated substring is a De Bruijn cycle, then using Lemma 3 instead of Lemma 1.

Theorem 10. *With one stream, we cannot achieve an entropy-only bound.*

Proof. As in the proof of Theorem 4, suppose an algorithm uses only one stream, m bits of memory and p passes to compress s, with $mp = \log^{\mathcal{O}(1)} n$, and consider a machine performing that algorithm. This time, however, suppose s is periodic with period $2^{\lceil \log(mp \log n) \rceil}$ and that its repeated substring t is a kth-order De Bruijn cycle, $k = \lceil \log(mp \log n) \rceil$, that is not compressible by more than a constant factor. Lemma 2 implies that the machine's output while over a part of the stream that initially holds a copy of t, must be $\Omega(mp \log n - mp) = \Omega(mp \log n)$. Therefore, the machine's complete output must be $\Omega(n)$ bits. By Lemma 3, however, $nH_k^*(s) = \mathcal{O}\big(2^k \log n\big) = \mathcal{O}\big(mp \log^2 n\big) \subset \log^{\mathcal{O}(1)} n$. □

Notice Theorem 10 implies a lower bound for computing the BWT: if we could compute the BWT with one stream then, since we can compute move-to-front, run-length and arithmetic coding using $\mathcal{O}(\log n)$ bits of memory and $\mathcal{O}(1)$ passes over one stream, we could thus achieve an entropy-only bound with one stream, contradicting Theorem 10.

Corollary 6. *With one stream, we cannot compute the BWT.*

In the full version of this paper we will show that computing the BWT of a ternary string is at least as hard as sorting $n/\log n$ numbers; therefore, it takes $\Omega(\log n)$ passes with $\mathcal{O}(1)$ streams.

In another paper [12] we improved the coefficient in Manzini's bound from $5+\epsilon$ to 2.7, using a variant of distance coding instead of move-to-front and run-length coding. We conjecture this algorithm can also be implemented with two streams.

Open Problem 3. *With $\mathcal{O}(1)$ streams, can we achieve the same entropy-only bounds that we achieve in the RAM model?*

Acknowledgments. Many thanks to Giovanni Manzini, Ferdinando Cicalese, Paolo Ferragina and Roberto Grossi for helpful discussions. This research was supported by the Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

References

1. Aggarwal, G., Datar, M., Rajagopalan, S., Ruhl, M.: On the streaming model augmented with a sorting primitive. In: Proceedings of the 45th Symposium on Foundations of Computer Science, pp. 540–549 (2004)
2. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM Journal on Computing 36(6), 1672–1695 (2007)
3. Beame, P., Huỳnh-Ngọc, D.-T.: On the value of multiple read/write streams for approximating frequency moments. In: Proceedings of the 49th Symposium on Foundations of Computer Science, pp. 499–508 (2008)
4. Bird, R.S., Mu, S.-C.: Inverting the Burrows-Wheeler transform. Journal of Functional Programming 14(6), 603–612 (2004)
5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 24, Digital Equipment Corporation (1994)
6. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., shelat, a.: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
7. Chen, J., Yap, C.-K.: Reversal complexity. SIAM Journal on Computing 20(4), 622–638 (1991)
8. Cilibrasi, R., Vitányi, P.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (2005)
9. Cover, T.M., Thomas, J.A.: Elements of Information Theory, 2nd edn. Wiley, Chichester (2006)
10. de Bruijn, N.G.: A combinatorial problem. Koninklijke Nederlandse Akademie van Wetenschappen 49, 758–764 (1946)

11. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
12. Gagie, T., Manzini, G.: Move-to-front, distance coding, and inversion frequencies revisited. In: Proceedings of the 18th Symposium on Combinatorial Pattern Matching, pp. 71–82 (2007)
13. Gagie, T., Manzini, G.: Space-conscious compression. In: Proceedings of the 32nd Symposium on Mathematical Foundations of Computer Science, pp. 206–217 (2007)
14. Grohe, M., Koch, C., Schweikardt, N.: Tight lower bounds for query processing on streaming and external memory data. Theoretical Computer Science 380(1–3), 199–217 (2007)
15. Grohe, M., Schweikardt, N.: Lower bounds for sorting with few random accesses to external memory. In: Proceedings of the 24th Symposium on Principles of Database Systems, pp. 238–249 (2005)
16. Gupta, A., Grossi, R., Vitter, J.S.: Nearly tight bounds on the encoding length of the Burrows-Wheeler Transform. In: Proceedings of the 4th Workshop on Analytic Algorithmics and Combinatorics, pp. 191–202 (2008)
17. Hernich, A., Schweikardt, N.: Reversal complexity revisited. Theoretical Computer Science 401(1–3), 191–205 (2008)
18. Knuth, D.E.: The Art of Computer Programming, 2nd edn., vol. 3. Addison-Wesley, Reading (1998)
19. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM Journal on Computing 29(3), 893–911 (1999)
20. Manzini, G.: An analysis of the Burrows-Wheeler Transform. Journal of the ACM 48(3), 407–430 (2001)
21. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)
22. Muthukrishnan, S.: Data Streams: Algorithms and Applications. In: Foundations and Trends in Theoretical Computer Science. Now Publishers (2005)
23. Ruhl, J.M.: Efficient Algorithms for New Computational Models. PhD thesis, Massachusetts Institute of Technology (2003)
24. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science 302(1–3), 211–222 (2003)
25. Savari, S.: Redundancy of the Lempel-Ziv incremental parsing rule. IEEE Transactions on Information Theory 43(1), 9–21 (1997)
26. Schweikardt, N.: Machine models and lower bounds for query processing. In: Proceedings of the 26th Symposium on Principles of Database Systems, pp. 41–52 (2007)
27. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)
28. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Reoptimization of the Shortest Common Superstring Problem*

(Extended Abstract)

Davide Bilò[1], Hans-Joachim Böckenhauer[2], Dennis Komm[2], Richard Královič[2], Tobias Mömke[2], Sebastian Seibert[2], and Anna Zych[2]

[1] Department of Computer Science, University of L'Aquila
dbilo@inf.ethz.ch

[2] Department of Computer Science, ETH Zurich, Switzerland
{hjb,dennis.komm,richard.kralovic, tobias.moemke,sseibert,anna.zych}@inf.ethz.ch

Abstract. A reoptimization problem describes the following scenario: Given an instance of an optimization problem together with an optimal solution for it, we want to find a good solution for a locally modified instance.

In this paper, we deal with reoptimization variants of the shortest common superstring problem where the local modifications consist of adding or removing a single string. We show NP-hardness of these reoptimization problems and design several approximation algorithms for them.

1 Introduction

In classical algorithmics, one is interested in finding good feasible solutions to input instances about which nothing is known in advance. Unfortunately, many practically relevant problems are computationally hard, and so different approaches such as approximation algorithms or heuristics are used for computing good approximations for optimal solutions. In the real world, however, some extra knowledge about the instance at hand might be already known. The concept of reoptimization employs a special kind of additional knowledge: Under the assumption that we are given an instance of an optimization problem together with an optimal solution for it, we want to efficiently compute a good solution for a locally modified input instance.

This concept of reoptimization was mentioned for the first time in [13] in the context of postoptimality analysis for some scheduling problem. Postoptimality analysis deals with the related question of how much an instance may be altered without changing the set of optimal solutions, see, e. g., [17]. Since then, the concept of reoptimization has been successfully applied to various problems like

* This work was partially supported by SNF grant 200021-121745/1 and SBF grant C 06.0108 as part of the COST 293 (GRAAL) project funded by the European Union.

the traveling salesman problem [3,1,7,10], the Steiner tree problem [4,8,11], the knapsack problem [2], and various covering problems [5]. A survey of reoptimization problems can be found in [9].

In this paper, we investigate some reoptimization variants of the *shortest common superstring problem*, SCS for short. Given a substring-free set of strings, the SCS asks for a shortest common superstring of S, i. e., for a minimum-length string containing all strings from S as substrings. The SCS is one of the most prominent hard problems in stringology with many applications, e. g., in computational biology where it is used for modeling certain aspects of the DNA fragment assembly problem (see, for instance, [14,6] for more details). The SCS is known to be NP-hard [12] and even APX-hard [18]. Many approximation algorithms have been devised for the SCS, the best-known being a greedy algorithm proposed by Tarhio and Ukkonen [16] which can be proven to achieve an approximation ratio of 4, but is conjectured to be 2-approximative. The currently best known approximation algorithm achieves a ratio of 2.5 [15].

In this paper, we deal with reoptimizing the SCS under the local modifications of adding or removing a single string. Our main results are the following. We show that both reoptimization versions of the SCS are NP-hard and propose some approximation algorithms for them. First, we devise an iteration technique for improving the approximation ratio of any SCS algorithm in the presence of a long string in the input which might be of independent interest. Then, we use this iteration technique to design an algorithm for SCS reoptimization which gives an approximation ratio arbitrarily close to 1.6 for adding a string and a ratio arbitrarily close to 13/7 for removing a string. This algorithm uses some known approximation algorithm for the original SCS (without reoptimization), and its approximation ratio depends on the ratio of this SCS algorithm. Thus, any improvement over the best known ratio of 2.5 for the SCS immediately yields also an improvement of these reoptimization results. Since the running time of this iterative algorithm is rather high, we also analyze a simple and fast reoptimization algorithm for adding a string and prove an approximation ratio of 11/6 for it.

The paper is organized as follows. In Section 2, we formally define the reoptimization variants of the SCS and fix our notation. Section 3 is devoted to the hardness results, in Section 4, we present the iterative reoptimization algorithms, and Section 5 contains the analysis of the fast approximation algorithm for adding a string.

2 Preliminaries

We start with defining some notations for dealing with strings that we will use throughout the paper. By λ we denote the empty string. The concatenation of two strings s and t will be written as $s \cdot t$, or as st for short. Let s, t, x, and y be some (possibly empty) strings such that $t = xsy$. Then s is a *substring* of t, we write $s \sqsubseteq t$, and t is a *superstring* of s. If x is empty, we say that s is a *prefix* of t, if y is empty, then s is a *suffix* of t. We say that a set S of strings is *substring-free* if $s \not\sqsubseteq t$, for all $s, t \in S$.

For two strings s_1 and s_2, the *overlap* $\mathrm{ov}(s_1, s_2)$ of s_1 and s_2 is the maximum-length *proper* suffix of s_1 which is also a *proper* prefix of s_2, i. e., we additionally require that $s_1, s_2 \not\sqsubseteq \mathrm{ov}(s_1, s_2)$. The corresponding prefix of s_1, i. e., the string p such that $s_1 = p \cdot \mathrm{ov}(s_1, s_2)$, is denoted by $\mathrm{pref}(s_1, s_2)$. The *merge* of s_1 and s_2 is defined as $\mathrm{merge}(s_1, s_2) := \mathrm{pref}(s_1, s_2) \cdot s_2$. We inductively extend this notion of merge to more than two strings by defining

$$\mathrm{merge}(s_1, \dots, s_m) = \mathrm{merge}(\mathrm{merge}(s_1, \dots, s_{m-1}), s_m).$$

We call a string s *periodic* with period π, if there exist a suffix $\underline{\pi}$ and a prefix $\overline{\pi}$ of the string π and some $k \in \mathbb{N}$ such that $s = \underline{\pi} \cdot \pi^k \cdot \overline{\pi}$. In this case, we also write $s \sqsubseteq \pi^\infty$.

The problem we are investigating in this paper is to find the shortest common superstring for a given set $S = \{s_1, \dots, s_m\}$ of strings. If S is substring-free, then the shortest common superstring can be unambiguously described by the order in which the strings appear in it: If $s_{i_1}, \dots, s_{i_m}$ is the order of appearance in a shortest superstring t, then $t = \mathrm{merge}(s_{i_1}, \dots, s_{i_m})$. This observation leads to the following formal definition of the problem.

Definition 1. *The* shortest common superstring problem*, SCS for short, is the following optimization problem: Given a substring-free set of strings $S = \{s_1, \dots, s_m\}$, the feasible solutions are all permutations $(s_{i_1}, \dots, s_{i_m})$ of S. For any feasible solution* $\mathrm{Sol} = (s_{i_1}, \dots, s_{i_m})$*, the cost is* $|\mathrm{Sol}| = |\mathrm{merge}(s_{i_1}, \dots, s_{i_m})|$*, i. e., the length of the shortest superstring for S containing the strings from S in the order as given by* Sol*. The goal is to find a permutation minimizing the length of the corresponding superstring.*

In this paper, we deal with two reoptimization variants of the SCS. The local modifications we consider here are adding a string to our set of input strings or deleting one string from it. The corresponding reoptimization problem can be formally defined as follows.

Definition 2. *The input for the* SCS reoptimization problem with adding a string*, SCS+ for short, consists of a substring-free set $S_O = \{s_1, \dots, s_m\}$ of strings, an optimal SCS-solution* Opt_O *for it, and a string s_{new} such that also $S_N = S_O \cup \{s_{new}\}$ is substring-free.*

Analogously, the input for the SCS reoptimization problem with removing a string*, SCS– for short, consists of a substring-free set of strings $S_O = \{s_1, \dots, s_m\}$, an optimal SCS-solution* Opt_O *for it, and a string $s_{old} \in S_O$. In this case, $S_N = S_O \setminus \{s_{old}\}$. For both problems, the goal is to find an optimal SCS-solution* Opt_N *for S_N.*

In addition to the maximum overlap and merge as defined above, we also consider the overlap and merge inside a given solution. Let Sol be some solution for an SCS instance given by a set of strings S and let s and t be two strings from S which are not necessarily overlapping in Sol. Then $\mathrm{ov}_{\mathrm{Sol}}(s, t)$ denotes the overlap of s and t in Sol, and we use $\mathrm{merge}_{\mathrm{Sol}}(s, t) = \mathrm{merge}(s, \dots, t)$ as an abbreviation for the merge of s and t together with all input strings lying between them in Sol. By $\mathrm{prefM}_{\mathrm{Sol}}(s,t)$, we denote the prefix of $\mathrm{merge}_{\mathrm{Sol}}(s,t)$ such that $\mathrm{prefM}_{\mathrm{Sol}}(s,t) \cdot t = \mathrm{merge}_{\mathrm{Sol}}(s,t)$. Note that s may be a proper prefix of $\mathrm{prefM}_{\mathrm{Sol}}(s,t)$. For $\mathrm{Sol} = \mathrm{Opt}_O$, we use the notations ov_O, merge_O, and prefM_O for $\mathrm{ov}_{\mathrm{Opt}_O}$, $\mathrm{merge}_{\mathrm{Opt}_O}$, and $\mathrm{prefM}_{\mathrm{Opt}_O}$, respectively. Analogously, we use ov_N, merge_N, and prefM_N for $\mathrm{Sol} = \mathrm{Opt}_N$. Note that, for two consecutive strings s and t inside some solution Sol, $\mathrm{merge}_{\mathrm{Sol}}(s,t) = \mathrm{merge}(s,t)$, but this equality does not necessarily hold for non-consecutive strings.

3 Hardness Results

In this section, we show that the considered reoptimization problems are NP-hard. Similarly to [9], we use a polynomial-time Turing reduction since we rely on repeatedly applying reoptimizations.

Theorem 1. *The problems SCS+ and SCS– are NP-hard.*

Proof. We split the reduction into several steps. Given an input instance I for SCS, we define a corresponding easily solvable instance I'. Then we show that I' is indeed solvable in polynomial time. Finally, we show how to use polynomially many reoptimization steps in order to transform the optimal solution for I' into an optimal solution for I.

For any SCS+ instance I, the easy instance I' consists of no strings. Obviously, the empty vector is an optimal solution for I'. Now, I' can be transformed into any instance I by adding all strings from I one after the other. Thus, SCS+ is NP-hard.

Now, let us consider the local modification of removing strings. Let I be an instance for SCS that consists of m strings $s_1, s_2, \ldots, s_m$. For any i, let s_i^f be the first symbol of s_i, let s_i^l be its last symbol, and let s_i^c be s_i without the first and last symbol. Without loss of generality, we exclude strings of length 1 since they cannot significantly increase the hardness of any input instance.

Now, we construct I' as follows. Let $\#_0, \#_1, \ldots, \#_m$ be $m+1$ special symbols that do not appear in I. Then, we introduce the set of strings $S' := \{s'_0, s'_1, \ldots, s'_m\}$, where $s'_0 := \#_0 s_1^f s_1^c$, $s'_m := s_m^c s_m^l \#_m$, and $s'_i := s_i^c s_i^l \#_i s_{i+1}^f s_{i+1}^c$, for each $i \in \{1, \ldots, m-1\}$. Let the instance I' be the set of the strings from I together with the strings from S'. It is clear that $m+1$ local modifications, each removing one of the new strings, transform I' into I. Thus, it only remains to

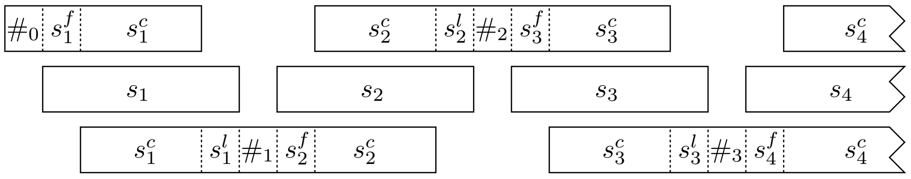


Fig. 1. An optimal solution for the easily solvable instance I'

show that I' is efficiently solvable. To this end, we claim that no algorithm can do better than alternating the new and the old strings as depicted in Fig. 1.

We now formally prove the correctness of the construction above. First, observe that the constructed instance is substring-free. Now, let us only consider the strings from S'. We will show that at each position of any common superstring at most two of these strings can overlap. Suppose, conversely, that more than two of the strings overlap. Then there are pairwise disjoint numbers i,j, and k between 0 and m such that s'_i, s'_j, and s'_k overlap in at least one symbol. Let, without loss of generality, s'_i be the leftmost string and let s'_k be the rightmost string in an overlapping setting. But then each symbol of the middle string, s'_j, is overlapped by at least one of the other strings — a contradiction, because the symbol $\#_j$ only appears in s'_j.

Following the construction of S', the overall length $\sum_{i=0}^{m} |s'_i|$ of the strings from S' is $m+1+2\cdot\sum_{i=1}^{m}(|s_i|-1)$. Since only non-special symbols can overlap, any shortest superstring is at least $m+1+\sum_{i=1}^{m}(|s_i|-1)$ symbols long; otherwise, there would be some position in the superstring that overlaps with three strings from S'. Finally, we have to include the strings from I. To this end, we will show that adding m strings not containing special symbols to S' results in a lower bound of $m+1+\sum_{i=1}^{m}|s_i|$ on the length of any common superstring.

Note that, given a substring-free set of k strings, where w is the longest one, it cannot have a common superstring t_1 that is shorter than $|w|+(k-1)$, i. e.,

$$|t_1| \geq |w| + k - 1. \tag{1}$$

Similarly, given a substring-free set of $k+2$ strings exactly two of which contain special characters, namely $w_l = u_l\#_u u_r$ and $w_r = v_l\#_v v_r$, any common superstring starting with w_l has at least $|w_l|+k$ symbols and, analogously, any common superstring ending with w_r has at least $|w_r|+k$ symbols.

Given a common superstring t_2 which starts with w_l and ends with w_r, we have

$$|t_2| \geq |u_l| + 1 + |v_r| + 1 + \max\{|u_r|, |v_l|\} + k. \tag{2}$$

Now let us consider I'. Let t be a common superstring for I'. We decompose t into $w_0 w'_0 w_1 w'_1 w'_2 w_2 \dots w_m w'_m w_{m+1}$ such that each w'_i consists of exactly one special symbol. Therefore, each string from I is contained in at least one of the strings between the special symbols. Let k_i be the number of strings from I that is contained in w_i. Then, according to (1) and (2), w_i is at least k_i symbols longer than the longer end of the two special strings belonging to w'_i and w'_{i+1}. Due to the estimation of the length of a shortest common superstring above, and since all m strings of I have to appear somewhere, i. e., $\sum_{i=0}^{m+1} k_i \geq m$, the length of t is at least

$$\sum_{i=1}^{m}(|s_i|-1) + m + 1 + \sum_{i=0}^{m+1} k_i \geq \sum_{i=1}^{m}(|s_i|-1) + m + 1 + m = \sum_{i=1}^{m}|s_i| + m + 1.$$

But this is exactly the length of our constructed common superstring. Therefore, we conclude that SCS– is NP-hard.

4 Iterative Algorithms for Adding or Removing a String

Consider any polynomial approximation algorithm A for SCS with approximation ratio γ. We show how to construct a polynomial reoptimization algorithm for SCS+ with approximation ratio arbitrarily close to $(2\gamma-1)/\gamma$. Furthermore, we show a similar result for SCS– with approximation ratio $(3\gamma-1)/(\gamma+1)$. Since the best known polynomial approximation algorithm for SCS gives $\gamma = 2.5$, see [15], we obtain an approximation ratio arbitrarily close to $8/5 = 1.6$ for SCS+ and an approximation ratio arbitrarily close to $13/7 < 1.86$ for SCS–.

The core part of our reoptimization algorithms is an approximation algorithm for SCS that works well if the input instance contains at least one long string. More precisely, let $S = \{s_1, \ldots, s_m\}$ be an instance of SCS such that $\mu_0 \in S$ is a longest string in S, and let $|\mu_0| = \alpha_0|\mathrm{Opt}|$, for some $\alpha_0 > 0$, where Opt is an optimal solution of S.

The algorithm A_1 guesses the leftmost string l_1 and the rightmost string r_1 which overlap with μ_0 in the string corresponding to Opt, together with the respective overlap lengths. Afterwards, it computes a new instance S_1 by eliminating all substrings of $\mathrm{merge}_{\mathrm{Opt}}(l_1, \mu_0, r_1)$ from the instance S, calls the algorithm A on S_1 and appends l_1, μ_0, r_1 to the approximate solution returned by A.

Now we generalize A_1 by iterating this procedure k times. For arbitrary k, we construct a polynomial-time approximation algorithm A_k for SCS that computes a solution of length at most

$$\left(1 + \frac{\gamma^k(\gamma-1)}{\gamma^k-1}(1-\alpha_0)\right)|\mathrm{Opt}|.$$

For every $i \in \{1, \ldots, k\}$, we define strings l_i, r_i, and μ_i as follows: Let l_i be the leftmost string that overlaps with μ_{i-1} in Opt. If there is no such string, $l_i := \mu_{i-1}$. Similarly, let r_i be the rightmost string that overlaps with μ_{i-1} in Opt. We define μ_i as $\mathrm{merge}_{\mathrm{Opt}}(l_i, \mu_{i-1}, r_i)$.

The algorithm A_k uses exhaustive search to find strings l_i, r_i and μ_i for every $i \in \{1, \ldots, k\}$. This can be done by assigning every possible string of S to l_i and r_i, and trying every possible overlap between l_i, μ_{i-1} and r_i. For every feasible candidate set of strings and for every i, the algorithm computes the candidate solution Sol_i corresponding to the string $\mathrm{merge}(u_i, \mu_i)$, where u_i is the string corresponding to the result of algorithm A on the input instance S_i obtained by removing all substrings of μ_i from S. Algorithm A_k then outputs the best solution among all candidate solutions.

Theorem 2. *Let n be the total length of all strings in S, i. e., $n = \sum_{j=1}^{m} |s_j|$. Algorithm A_k works in time $O(m^{2k}n^{2k}(kmn + kT(m, n)))$, where $T(m, n)$ is the time complexity of algorithm A on an input instance with at most m strings of total length at most n.*

Proof. Algorithm A_k needs to test all $O(m^{2k})$ possibilities for choosing $2k$ strings $l_1, r_1, \ldots, l_k, r_k$ from the m strings of S. For every such possibility, it must test all possible overlaps between the strings in order to obtain strings $\mu_1, \dots, \mu_k$. Hence, the lengths of $2k$ overlaps must be tested. As the length of each overlap can be in the range from 0 to n, there are $O(n^{2k})$ possibilities. For each of the $O(m^{2k}n^{2k})$ possibilities, A_k tests if it is feasible (this can be done in time $O(n)$) and computes the corresponding k candidate solutions. To compute one candidate solution Sol_i, the instance S_i is prepared in time $O(mn)$ and algorithm A is executed in time $T(m,n)$. □

Theorem 3. *Algorithm A_k finds a solution of S of length at most*

$$\left(1 + \frac{\gamma^k(\gamma-1)}{\gamma^k - 1}(1-\alpha_0)\right)|\mathrm{Opt}|.$$

Proof. Assume that A_k outputs a solution of length greater than $(1+\beta)|\mathrm{Opt}|$, for some $\beta > 0$. In the analysis, we focus on the part of the computation of A_k where the correct assignment of strings l_i, r_i, and μ_i is analyzed. By our assumption, every candidate solution Sol_i has length greater than $(1+\beta)|\mathrm{Opt}|$. The solution Sol_i corresponds to the string $\mathrm{merge}(u_i, \mu_i)$, where $|\mu_i| = \alpha_i|\mathrm{Opt}|$, for some $\alpha_i > 0$, and u_i is the result of algorithm A on the input instance S_i. Hence, $|\mathrm{Sol}_i| \le |u_i| + |\mu_i|$.

It is not difficult to check that, if we remove all substrings of μ_i from Opt, we obtain a feasible solution for S_i of length at most $|\mathrm{Opt}| - |\mu_{i-1}| = (1-\alpha_{i-1})|\mathrm{Opt}|$: By definition of μ_i, we have removed every string that overlapped with μ_{i-1}. Hence, $|u_i| \le \gamma(1-\alpha_{i-1})|\mathrm{Opt}|$, and

$$(1+\beta)|\mathrm{Opt}| < |\mathrm{Sol}_i| \le (\gamma(1-\alpha_{i-1}) + \alpha_i)|\mathrm{Opt}|. \tag{3}$$

Inequality (3) implies

$$\alpha_i > 1 + \beta - \gamma + \gamma\alpha_{i-1}. \tag{4}$$

Solving the system of recurrent equations (4) yields

$$\alpha_k > (1+\beta-\gamma)\frac{\gamma^k - 1}{\gamma - 1} + \gamma^k\alpha_0. \tag{5}$$

Since μ_i is a substring of Opt for every i, it holds that $\alpha_k \le 1$. Putting this together with (5) yields

$$\beta \le \frac{\gamma^k(\gamma-1)}{\gamma^k-1}(1-\alpha_0).$$

□

4.1 Reoptimization of SCS+

We now employ the iterative SCS algorithm described above for designing an approximation algorithm for SCS+. For every k, we define the algorithm A_k^+ for SCS+ as follows. Given an input instance S_O, its optimal solution Opt_O, and a new string s_{new}, the algorithm A_k^+ returns the solution Sol_1 corresponding to $\mathrm{merge}(\mathrm{Opt}_O, s_{new})$ or the solution Sol_2 computed by A_k for the input instance $S_N := S_O \cup \{s_{new}\}$, whichever is better.

Theorem 4. *Algorithm A_k^+ yields a solution of length at most*

$$\frac{2\gamma^{k+1}-\gamma^k-1}{\gamma^{k+1}-1}|\mathrm{Opt}_N|.$$

Proof. Let $|s_{new}| = \alpha|\mathrm{Opt}_N|$. Then $|\mathrm{Sol}_1| \leq (1+\alpha)|\mathrm{Opt}_N|$. Since S_N contains a string of length at least $\alpha|\mathrm{Opt}_N|$, Theorem 3 ensures that

$$|\mathrm{Sol}_2| \leq \left(1+\frac{\gamma^k(\gamma-1)}{\gamma^k-1}(1-\alpha)\right)|\mathrm{Opt}_N|.$$

Hence, the minimum of $|\mathrm{Sol}_1|$ and $|\mathrm{Sol}_2|$ is maximal if

$$(1+\alpha)|\mathrm{Opt}_N| = \left(1+\frac{\gamma^k(\gamma-1)}{\gamma^k-1}(1-\alpha)\right)|\mathrm{Opt}_N|,$$

which happens if

$$\alpha = \frac{\gamma^{k+1}-\gamma^k}{\gamma^{k+1}-1}.$$

In this case, A_k^+ yields a solution of length at most

$$(1+\alpha)|\mathrm{Opt}_N| = \frac{2\gamma^{k+1}-\gamma^k-1}{\gamma^{k+1}-1}|\mathrm{Opt}_N|. \qquad \square$$

By choosing k sufficiently large, the approximation ratio of A_k^+ can be made arbitrarily close to $(2\gamma-1)/\gamma$. Algorithm A_k^+ is polynomial for every k, but the degree of the polynomial grows with k.

4.2 Reoptimization of SCS–

Similarly as for the case of SCS+, we define algorithm A_k^- for SCS– as follows. Given an input instance S_O, its optimal solution Opt_O and a string $s_{old} \in S_O$ to be removed, A_k^- returns the solution Sol_1 obtained from Opt_O by leaving out s_{old}, or the solution Sol_2 computed by A_k for input instance $S_N := S_O \setminus \{s_{old}\}$, whichever is better.

Theorem 5. *Algorithm A_k^- yields a solution of length at most*

$$\frac{3\gamma^{k+1}-\gamma^k-2}{\gamma^{k+1}+\gamma^k-2}|\mathrm{Opt}_N|.$$

Proof. Let $l \in S_O$ $(r \in S_O)$ be the string that immediately precedes (follows) s_{old} in Opt_O, respectively. We focus on the case where both l and r exist, the other cases are analogous. It is easy to see that

$$|\mathrm{Sol}_1| \leq |\mathrm{Opt}_O| - |s_{old}| + |\mathrm{ov}(l, s_{old})| + |\mathrm{ov}(s_{old}, r)|.$$

Since augmenting Opt_N with s_{old} yields a feasible solution for S_O, we have $|\mathrm{Opt}_O| \leq |\mathrm{Opt}_N| + |s_{old}|$.

Without loss of generality, assume that $|\mathrm{ov}(s_{old}, r)| \leq |\mathrm{ov}(l, s_{old})| = \alpha|\mathrm{Opt}_N|$. Hence, $|\mathrm{Sol}_1| \leq (1 + 2\alpha)|\mathrm{Opt}_N|$. Furthermore, S_N contains the string l of length at least $\alpha|\mathrm{Opt}_N|$, so Theorem 3 ensures that

$$|\mathrm{Sol}_2| \leq \left(1 + \frac{\gamma^k(\gamma - 1)}{\gamma^k - 1}(1 - \alpha)\right)|\mathrm{Opt}_N|.$$

The minimum of $|\mathrm{Sol}_1|$ and $|\mathrm{Sol}_2|$ is maximal if

$$(1 + 2\alpha)|\mathrm{Opt}_N| = \left(1 + \frac{\gamma^k(\gamma - 1)}{\gamma^k - 1}(1 - \alpha)\right)|\mathrm{Opt}_N|,$$

which happens if

$$\alpha = \frac{\gamma^{k+1} - \gamma^k}{\gamma^{k+1} + \gamma^k - 2}.$$

In this case, A_k^- yields a solution of length at most

$$\frac{3\gamma^{k+1} - \gamma^k - 2}{\gamma^{k+1} + \gamma^k - 2}|\mathrm{Opt}_N|. \qquad \square$$

Similarly as in the case of SCS+, the approximation ratio of A_k^- can be made arbitrarily close to $(3\gamma - 1)/(\gamma + 1)$ by choosing k sufficiently large.

5 One-Cut Algorithm for Adding a String

In this section, we present a simple and fast algorithm OneCut for SCS+ and prove that it achieves an 11/6-approximation ratio. The algorithm cuts Opt_O at all positions one by one. Recall that the given optimal solution Opt_O is represented by an ordering of the input strings, thus cutting Opt_O at some position yields a partition of the input strings into two sub-orderings. The two corresponding strings are then merged with s_{new} in between. The algorithm returns a shortest of the strings obtained in this manner, see Algorithm 1.

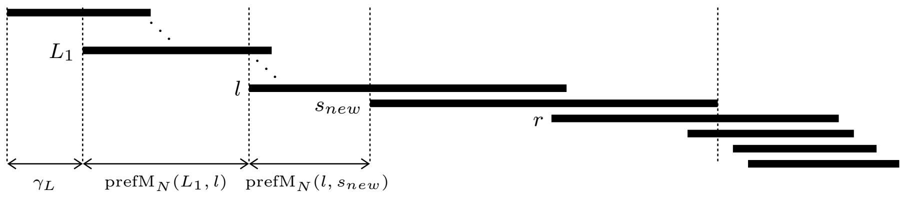
Algorithm 1. OneCut

```
Input: A set of strings $S = \{s_1, \ldots, s_m\}$, an optimal solution $\mathrm{Opt}_O = (s_1, \ldots, s_m)$ for
    $S$, and a string $s_{new}$
1: for $i \in \{0, \ldots, m\}$ do
2:     Let Solution$_i := (s_1, \ldots, s_i, s_{new}, s_{i+1}, \ldots, s_m)$.
Output: A best of the obtained solutions Solution$_i$, for $0 \leq i \leq m$
```

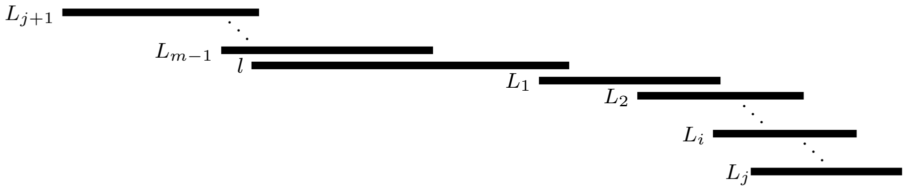
Theorem 6. *The algorithm* OneCut *is an* $11/6$*-approximation algorithm for SCS+ running in time* $\mathcal{O}(n \cdot m)$ *for inputs consisting of* m *strings of total length* n *over a constant-size alphabet.*

Proof sketch. We first analyze the running time of OneCut. Using suffix trees, we can compute all pairwise overlaps of $\{s_{new}, s_1, \ldots, s_m\}$ in time $\mathcal{O}(n \cdot m)$, see e. g. [16]. Using these precomputed overlaps, each of the $m+1$ iterations of OneCut can be performed in constant time. Thus, the overall running time of OneCut is also in $\mathcal{O}(n \cdot m)$.

We now show that OneCut provides an approximation ratio of 11/6 for SCS+. The proof is constructed in the following manner. One by one, we eliminate cases in which we can prove a ratio of 11/6 for OneCut, until all cases are covered. Each time we prove a ratio of 11/6 under some condition, we can deal in the following with the remaining cases under the assumption that this condition does not hold. In this way, we construct a list of assumptions which eventually lead to some final case. Due to the space limitations, the proofs of the lemmas are omitted in this extended abstract.



(a) The new optimal solution Opt_N (in the case that L_1 precedes l)



(b) The old optimal solution Opt_O (in the case that $L_i \neq \lambda$)

Fig. 2. The new and old optimal solution

Lemma 1. *If* $|s_{new}| \leq \frac{5}{6}|\text{Opt}_N|$, *then the algorithm* OneCut *provides an* 11/6-*approximation ratio.*

Lemma 1 shows that the desired approximation ratio can be reached whenever the string s_{new} is short. This leads to the first assumption.

Assumption 1. $|s_{new}| > \frac{5}{6}|\text{Opt}_N|$.

Let l be the string directly preceding s_{new} in Opt_N and let r be the direct successor of s_{new} in Opt_N (see Fig. 2 (a)). Lemma 2 proves that we may assume, without loss of generality, that l and r almost completely cover the string s_{new}.

Lemma 2. *If* OneCut *returns an* 11/6-*approximation for all instances where there is at most one letter from* s_{new} *not covered in* Opt_N *by either* l *or* r, *then it returns an* 11/6-*approximation in general.*

Assumption 2. *In* Opt_N*, at most one letter of the string* s_{new} *is not covered by either* l *or* r*.*

Lemma 3. *Assumption 2 implies that either* $|s_{new}| \le \frac{1}{2}|\mathrm{Opt}_N| + |\mathrm{ov}(l, s_{new})|$ *or* $|s_{new}| \le \frac{1}{2}|\mathrm{Opt}_N| + |\mathrm{ov}(s_{new}, r)|$.

By Lemma 3 and Assumption 2, without loss of generality, we may assume the following.

Assumption 3. $|s_{new}| \le \frac{1}{2}|\mathrm{Opt}_N| + |\mathrm{ov}(l, s_{new})|$.

We now enumerate the strings in Opt_O according to the position of l as shown in Fig. 2 (b):

$$\mathrm{Opt}_O = (L_{j+1}, \ldots, L_{m-1}, l, L_1, \ldots, L_i, \ldots, L_j)$$

Thus, let L_1 be the direct successor of l in Opt_O. If l has no successor in Opt_O, let $L_1 = \lambda$ be the empty string. In this case, the strings preceding l in Opt_O are $L_2, \ldots, L_m$, and we may assume that L_1 is located at the end of Opt_O.

In Lemma 4, we resolve the case where L_1 follows s_{new} in Opt_N.

Lemma 4. *Under Assumptions 1 and 3, if* L_1 *is located after* s_{new} *in* Opt_N*, then* OneCut *returns an* 11/6*-approximation.*

If $L_1 = \lambda$, we may assume that it follows l in Opt_N. Thus, we can add the following assumption.

Assumption 4. $L_1 \neq \lambda$ *and* L_1 *precedes* s_{new} *in* Opt_N*.*

We define $\pi_L = AB$, where $A = \mathrm{prefM}_N(L_1, l)$ and $B = \mathrm{prefM}_O(l, L_1) = \mathrm{pref}(l, L_1)$. Note that $L_1 = (AB)^g p_1$ and $l = (BA)^h p_2$ for some natural numbers g, h, where p_1 and p_2 denote some prefixes of AB and BA, respectively (see Fig. 3). Thus, $L_1, l \sqsubseteq \pi_L^\infty$. Now let L_i be the first string after L_1 in Opt_O which is not periodic with period π_L, i. e., $L_i \not\sqsubseteq \pi_L^\infty$. If there is no such string, let $L_i = \lambda$ be the empty string. Let $L = \mathrm{merge}_O(l, L_{i-1})$. Let γ_L denote the prefix of Opt_N preceding L_1 (see Fig. 2 (a)).

Lemma 5. *Assumption 4 and* $|\pi_L| \ge \frac{1}{6}|\mathrm{Opt}_N| - |\gamma_L|$ *give approximation ratio* 11/6 *for* OneCut*.*

In Lemma 5, we have handled the case that the period π_L is relatively long, yielding the following assumption for the rest of the proof.

Assumption 5. $|\pi_L| + |\gamma_L| \le \frac{1}{6}|\mathrm{Opt}_N|$.

The 11/6-approximation is proven in Lemma 6 for the case where L_i follows s_{new} in Opt_N.

Lemma 6. *Under Assumptions 1, 2, 3, 4, and 5, and if* L_i *follows* s_{new} *in* Opt_N*,* OneCut *is an* 11/6*-approximation algorithm for SCS+.*

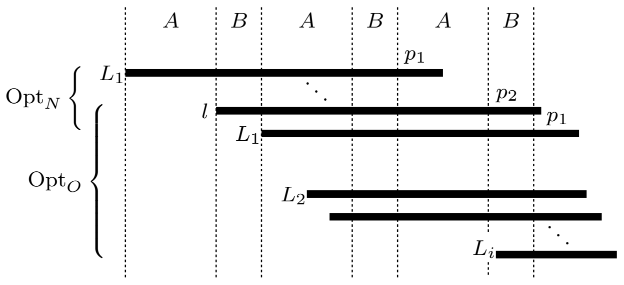


Fig. 3. Periodicity of l and L_1

Thus, we can make the following assumption for our final case. (In the case where $L_i = \lambda$, we may assume that L_i follows s_{new} in Opt_N.)

Assumption 6. *$L_i \neq \lambda$ and L_i precedes s_{new} in* Opt_N.

In the final case of the proof, as presented in Lemma 7, we will use Assumptions 1 to 6 to prove our claim for all remaining situations not previously dealt with.

Lemma 7. *Under Assumptions 1, 2, 3, 4, 5, and 6,* OneCut *provides an* 11/6-*approximation ratio for SCS+.*

This completes the proof of Theorem 6. □

We now show that the analysis in the proof of Theorem 6 is tight.

Theorem 7. *Algorithm* OneCut *cannot achieve an* $(\frac{11}{6} - \varepsilon)$*-approximation, for any* $\varepsilon > 0$.

Proof. For any $n \in \mathbb{N}$, we construct an input instance that consists of the following strings:

$$S_O = \{\vdash, xa^{n+2}x, a^{n+1}xa^{n+1}, a^n xa^{n+1}xa^n, \\ b^n yb^{n+1}yb^n, b^{n+1}yb^{n+1}, yb^{n+2}y, \dashv\}.$$

Obviously, arranging the strings in the order as presented forms an optimal solution Opt_O of length $6n + O(1)$:

$$\begin{array}{llllllllll}
 & & a^n & x\ a^{n+1} & x\ a^n & b^n & y\ b^{n+1} & y\ b^n & & \\
 & & a^{n+1} & x\ a^{n+1} & & & b^{n+1} & y\ b^{n+1} & & \\
 & x & a^{n+2} & x & & & & y\ b^{n+2} & y & \\
\vdash & & & & & & & & & \dashv
\end{array}$$

The corresponding superstring is $\vdash xa^{n+2}xa^{n+1}xa^n b^n yb^{n+1}yb^{n+2}y \dashv$. Let

$$s_{new} := b^{n-1}yb^{n+1}yb^n\#a^n xa^{n+1}xa^{n-1}.$$

It is easy to see that there is a solution for $S_N = S_O \cup \{s_{new}\}$ which has asymptotically the same length as Opt_O:

$$\begin{array}{ccccccccccccccc}
 & & b^{n-1} & y & b^{n+1} & y & b^{n} & \# & a^{n} & x & a^{n+1} & x & a^{n-1} & & \\
 & & b^{n} & y & b^{n+1} & y & b^{n} & & a^{n} & x & a^{n+1} & x & a^{n} & & \\
 & & b^{n+1} & y & b^{n+1} & & & & & & a^{n+1} & x & a^{n+1} & & \\
 & y & b^{n+2} & y & & & & & & & & x & a^{n+2} & x & \\
\vdash & & & & & & & & & & & & & & \dashv
\end{array}$$

This new optimal solution Opt_N is obviously *unique* (except for the placement of the symbols $\vdash$ and $\dashv$ at the beginning or the end). Applying algorithm OneCut for inserting s_{new} into the instance when Opt_O is given, however, does not find a common superstring that is shorter than $11n + O(1)$ symbols.

Here, the crucial observation is that all strings in S_O need to be rearranged to construct Opt_N from Opt_O (which then means that no information is gained by the given additional knowledge). To be optimal, 7 cuts are necessary. Being allowed to only cut once, however, cannot yield a solution better than $11n+\mathcal{O}(1)$. Finally, we easily verify that $|\mathrm{Opt}_N| = 6n + \mathcal{O}(1)$. □

6 Conclusion

In this paper, we have investigated the reoptimization of SCS according to two different local modifications. Besides the results presented here, there is a straight-forward generalization of the algorithm OneCut. For any constant k, we can also allow k cuts. We expect that additional cuts lead to improved approximation ratios. It is not hard, however, to show some lower bounds on the approximation ratio for k cuts. Using the same hard instance as in the proof of Theorem 7, we can show that two cuts do not improve the approximation ratio. In general, for any $\varepsilon > 0$ and $k \geq 3$, we have constructed a hard input instance such that the approximation ratio of the k-cut algorithm is bounded from below by $1 + 2/(k + 1) - \varepsilon$.

References

1. Archetti, C., Bertazzi, L., Speranza, M.G.: Reoptimizing the traveling salesman problem. Networks 42(3), 154–159 (2003)
2. Archetti, C., Bertazzi, L., Speranza, M.G.: Reoptimizing the 0-1 knapsack problem. Technical Report 267, University of Brescia (2006)
3. Ausiello, G., Escoffier, B., Monnot, J., Paschos, V.T.: Reoptimization of minimum and maximum traveling salesman's tours. In: Arge, L., Freivalds, R. (eds.) SWAT 2006. LNCS, vol. 4059, pp. 196–207. Springer, Heidelberg (2006)
4. Bilò, D., Böckenhauer, H.-J., Hromkovič, J., Královič, R., Mömke, T., Widmayer, P., Zych, A.: Reoptimization of Steiner trees. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 258–269. Springer, Heidelberg (2008)
5. Bilò, D., Widmayer, P., Zych, A.: Reoptimization of weighted graph and covering problems. In: Bampis, E., Skutella, M. (eds.) WAOA 2008. LNCS, vol. 5426, pp. 201–213. Springer, Heidelberg (2009)
6. Böckenhauer, H.-J., Bongartz, D.: Algorithmic Aspects of Bioinformatics. Natural Computing Series. Springer, Heidelberg (2007)

7. Böckenhauer, H.-J., Forlizzi, L., Hromkovič, J., Kneis, J., Kupke, J., Proietti, G., Widmayer, P.: Reusing optimal TSP solutions for locally modified input instances (extended abstract). In: Proc. of the Fourth IFIP International Conference on Theoretical Computer Science (IFIP TCS 2006), vol. 209, pp. 251–270. Springer, Heidelberg (2006)
8. Böckenhauer, H.-J., Hromkovič, J., Královič, R., Mömke, T., Rossmanith, P.: Reoptimization of Steiner trees: Changing the terminal set. Theoretical Computer Science (to appear)
9. Böckenhauer, H.-J., Hromkovič, J., Mömke, T., Widmayer, P.: On the hardness of reoptimization. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 50–65. Springer, Heidelberg (2008)
10. Böckenhauer, H.-J., Komm, D.: Reoptimization of the metric deadline TSP. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 156–167. Springer, Heidelberg (2008)
11. Escoffier, B., Milanic, M., Paschos, V.T.: Simple and fast reoptimizations for the Steiner tree problem. Technical Report 2007-01, DIMACS (2007)
12. Gallant, J., Maier, D., Storer, J.A.: On finding minimal length superstrings. Journal of Computer and System Sciences 20(1), 50–58 (1980)
13. Schäffter, M.W.: Scheduling with forbidden sets. Discrete Applied Mathematics 72(1-2), 155–166 (1997)
14. Setubal, C., Meidanis, J.: Introduction to Computational Molecular Biology. Natural Computing Series. PWS Publishing Company (1997)
15. Sweedyk, Z.: A $2\frac{1}{2}$-approximation algorithm for shortest superstring. SIAM Journal on Computing 29(3), 954–986 (2000)
16. Tarhio, J., Ukkonen, E.: A greedy approximation algorithm for constructing shortest common superstrings. Theoretical Computer Science 57(1), 131–145 (1988)
17. Van Hoesel, S., Wagelmans, A.: On the complexity of postoptimality analysis of 0/1 programs. Discrete Applied Mathematics 91(1-3), 251–263 (1999)
18. Vassilevska, V.: Explicit inapproximability bounds for the shortest superstring problem. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 793–800. Springer, Heidelberg (2005)

LCS Approximation via Embedding into Local Non-repetitive Strings

Gad M. Landau[1], Avivit Levy[2,3], and Ilan Newman[1]

[1] Department of Computer Science, Haifa University, Haifa 31905, Israel
{landau,ilan}@cs.haifa.ac.il
[2] Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel
avivitlevy@shenkar.ac.il
[3] CRI, Haifa University, Mount Carmel, Haifa 31905, Israel

Abstract. A classical measure of similarity between strings is the length of the *longest common subsequence*(LCS) between the two given strings. The search for efficient algorithms for finding the LCS has been going on for more than three decades. To date, all known algorithms may take quadratic time (shaved by logarithmic factors) to find large LCS. In this paper the problem of approximating LCS is studied, while focusing on the hard inputs for this problem, namely, approximating LCS of near-linear size in strings over relatively large alphabet (of size at least n^ϵ for some constant $\epsilon > 0$, where n is the length of the string). We show that, any given string over relatively large alphabet can be embedded into a local non-repetitive string. This embedding has a negligible additive distortion for strings that are not too dissimilar in terms of the edit distance. We also show that LCS can be efficiently approximated in locally-non-repetitive strings.

1 Introduction

Measuring similarity plays an important role in data analysis. As strings are a common data representation, similarity measures defined on strings are widely used. A classical measure of similarity between strings is the length of the *longest common subsequence* (LCS) between the two given strings. The search for efficient algorithms for finding the LCS has been going on for more than three decades. The classical dynamic programming algorithm takes quadratic time [21,22] and this complexity matches the lower bound in comparison model [1]. Many other algorithms have been suggested over the years [12,13,19,4,5,16,18,9] (see also [11]). However, the state of the art is still not satisfying. To date, all known algorithms may take near-quadratic time to find large LCS. None of the known algorithms can find LCS of linear size in time polynomially smaller than quadratic. Analysis of large data bases storing very long strings cannot settle with such methods.

A possible approach is to trade accuracy for speed and employ faster algorithms that approximate the LCS. In fact, for measuring similarity a sufficiently long common subsequence as an evidence of similarity might be as good as the LCS itself. Thus, a good approximation of the LCS that can be found fast is of great importance.

Approximating LCS in Strings Over Small Alphabet. Strings over small alphabet have large LCS. Thus, LCS in strings over small alphabet can be trivially approximated to a factor of $1/|\Sigma|$, where Σ is the alphabet, by just picking the letter that has the highest frequency. If the alphabet size is $o(n^{\epsilon})$ for every constant $\epsilon > 0$, this trivial algorithm achieves sub-polynomial approximation ratio, which is roughly the best known approximation ratio for the closely related edit distance [20][1]. However, when the alphabet of the strings gets larger this approximation becomes useless. Therefore, our goal is to design efficient algorithms approximating LCS over strings with relatively large alphabet, i.e., alphabet of size at least n^{ϵ}.

Sparse vs. Large LCS. Relatively large alphabet may reduce the number of matching symbols between the two given strings. In such cases the sparse LCS techniques of Hunt-Szymanski can be used to give efficient exact solutions that depend on the matchings set size [13,5]. However, the input strings may have a quadratic size of matching pairs of symbols even if the alphabet is relatively large. In these cases, these sparse LCS algorithms take quadratic time. Other methods for finding sparse LCS quickly are known. Specifically, LCS of size $O(n^{\alpha})$, where n is the string size and $0 < \alpha < 1$ is a constant, can be found by algorithms that take time $O(n^{1+\alpha})$ [12,16,18]. However, these algorithms take quadratic time for finding LCS of linear size. Thus, the focus of this paper is on efficiently approximating large LCS, typically, LCS of near linear size, in strings over relatively large alphabet.

Related Work. LCS is closely related to the *edit distance* (ED). The edit distance is the number of insertions, deletions, and substitutions needed to transform one string into the other. This distance is of key importance in several fields such as text processing, Web search and computational biology, and consequently computational problems involving ED have been extensively studied. The ED is the dissimilarity measure corresponding to the LCS similarity measure. The ED can also be computed by a quadratic time dynamic programming procedure. In fact, using the methods of Landau and Vishkin [17], ED can be computed in time $\max\{k^2, n\}$, where k is the bound on ED and n the length of the strings. Thus, a fast algorithm can find if the ED is small or not. Approximating ED efficiently has proved to be quite challenging [3]. Currently, the best quasi-linear time algorithm due to Batu, Ergün and Sahinalp [7], achieves approximation factor $n^{1/3+o(1)}$, where n is the length of the strings.

[1] [20] show an embedding into ℓ_1, which is stronger than an approximation algorithm. However, the time complexity of the embedding is high. It can, therefore, be used for various tasks such as sketching and nearest neighbor search, but not as an edit-distance approximation algorithm.

Results. In this paper it is shown that large LCS can be efficiently approximated in strings with relatively large alphabet if the ED is not too large. In particular, LCS of linear size can be approximated to a constant factor, if the edit distance is $o(\frac{n|\Sigma|}{t \ln t})$, where $|\Sigma|$ is the alphabet size and t is the period size ($t = n$ in aperiodic strings). It is important to note, that our algorithm does not need to verify that the requirement on the ED is indeed fulfilled. A large LCS detected by the algorithm is an evidence of similarity. For alphabet of size at least n^{ϵ}, our algorithm complexity is always $O(n^{2-\epsilon} \log\log n)$ but can be much better (for some parameters it is $O(n \log\log n)$). Our contribution to the computation of large common subsequences is, therefore, a strictly sub-quadratic time algorithm (i.e., of complexity $O(n^{2-\epsilon})$ for some constant ϵ) which can find common subsequences of linear (and near linear) size that cannot be detected efficiently by the existing tools.

The approximation ratio of our algorithm depends on the size of the LCS. It is better as the LCS is longer. Table 1 demonstrates the worst case performance of our algorithm for LCS of different sizes. The complexity guarantees presented in the table are a result of combining the theorems proved in this paper (Theorem 1 and Corollary 2 combined with Theorems 2 and 3 and Theorem 5). We stress that these are worst case performances also in the sense that they demonstrate the worst case parameters for given LCS size, alphabet size and period length, but the true parameters for a given pair of strings can be much better. The complexity of our method is superior compared to sparse LCS techniques when LCS of near-linear size is concerned, as the first 6 lines of the table indicate. Moreover, even for strictly sub-quadratic size LCS, our method gives a faster approximation algorithm if the alphabet is large enough. As lines 7 and 9 of the table indicate, for LCS of size $\Theta(n^{3/4})$ we get a faster algorithm for every $\epsilon > 1/2$. Line 8 of the table represents a case where our technique should not be used due to the requirement on the edit distance. In such a case, sparse LCS techniques should be preferred.

Our method works well for strings A and B where the ED is $o(LCS(A, B) \cdot \frac{|\Sigma|}{t \ln t})$, where Σ is the alphabet size and t depends on the periodicity of the input strings (can be of size n in aperiodic strings). The effect of these parameters is also demonstrated in Table 1. Note that, if the edit distance is $\Theta(n^{\epsilon})$, the exact LCS can be found in time $\max\{n^{2\epsilon}, n\}$, by finding the edit positions and taking the complement positions. However, for edit distance that is $\Omega(n/\log^c n)$, for some $c > 1$, our algorithm is strictly sub-polynomial, while computing the ED yields a near-quadratic time algorithm. Moreover, even for edit distance that is $\Theta(n^{\epsilon})$, our algorithm complexity is always superior when $\epsilon > 2/3$ and can be superior also for smaller ϵ, depending on the parameters of the strings.

Techniques. We exploit low distortion embedding of strings over relatively large alphabet into local non-repetitive strings. Local non-repetitiveness has been used for approximating ED [6] and for embedding ED [8]. In [6] and [8], efficient algorithms for input strings that are non-repetitive or locally-non-repetitive with good parameters are designed. Here, we show that any string over relatively large alphabet can be embedded into a locally non-repetitive string. We prove

Table 1. Worst Case Performance of Our Algorithm: Examples

LCS	Alphabet Size	Period Length	ED	Approximation Ratio	Complexity
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$o(n/\ln n)$	$\Theta(1)$	$O(n\log n)$
$\Theta(n)$	$\Theta(n^{\epsilon})$	$\Theta(n)$	$o(n^{\epsilon}/\ln n)$	$\Theta(1)$	$O(n^{2-\epsilon}\log\log n)$
$\Theta(n)$	$\Theta(n^{\epsilon})$	$\Theta(n^{\epsilon})$	$o(n/\ln n)$	$\Theta(1)$	$O(n^{2-\epsilon}\log\log n)$
$\Theta(n/\log^{c} n)$	$\Theta(n)$	$\Theta(n)$	$o(n/\log^{c+1} n)$	$\Theta(1/\log^{c} n)$	$O(n\log n)$
$\Theta(n/\log^{c} n)$	$\Theta(n^{\epsilon})$	$\Theta(n)$	$o(n^{\epsilon}/\log^{c+1} n)$	$\Theta(1/\log^{c} n)$	$O(n^{2-\epsilon}\log\log n)$
$\Theta(n/\log^{c} n)$	$\Theta(n^{\epsilon})$	$\Theta(n^{\epsilon})$	$o(n/\log^{c+1} n)$	$\Theta(1/\log^{c} n)$	$O(n^{2-\epsilon}\log\log n)$
$\Theta(n^{3/4})$	$\Theta(n)$	$\Theta(n)$	$o(n^{3/4/}/\ln n)$	$\Theta(1/n^{1/4})$	$O(n\log n)$
$\Theta(n^{3/4})$	$\Theta(n^{\epsilon})$	$\Theta(n)$	$o(n^{\epsilon-1/4}/\ln n)$	$\Theta(1/n^{1/4})$	$O(n^{2-\epsilon}\log\log n)$
$\Theta(n^{3/4})$	$\Theta(n^{\epsilon})$	$\Theta(n^{\epsilon})$	$o(n^{3/4}/\ln n)$	$\Theta(1/n^{1/4})$	$O(n^{2-\epsilon}\log\log n)$

that this embedding has an additive negligible (contraction) distortion, if $ED = o(LCS(A,B)\cdot\frac{|\Sigma|}{t\ln t})$. We then show that local non-repetitiveness can be used to significantly speed-up LCS approximation. The speed-up in the efficiency of our algorithm depends on the local non-repetitiveness parameters of the given strings. We show that local non-repetitiveness can be efficiently sketched so that the best parameters for any two strings can be found by looking at a poly-logarithmic sketch.

The paper is organized as follows. Sect. 2 presents basic definitions and properties. Sect. 3 presents the embedding of strings over relatively large alphabet into locally-non-repetitive strings, namely, (1,n/c)-non-repetitive strings, for some c. In Sect. 4 we present approximation algorithms for this special case of (1,n/c)-non-repetitive strings, where c is a parameter. Finally, in Sect. 5 we show that the best parameters for a given pair of strings can be quickly found by looking at local non-repetitiveness sketches (LNR-sketches) of the strings. It is shown that our LNR-sketch size matches the lower bound, and a lower bound on the space needed by a LNR-sketching algorithm in the streaming model is also given.

2 Preliminaries

In this section we give the basic definitions and properties used in this paper.

Problem Definition. Let A and B be two strings of length n over alphabet Σ. The *longest common subsequence problem* is to find the longest subsequence, denoted by $LCS(A,B)$, appearing in both A and B. We will abuse notation throughout the paper by letting $LCS(A,B)$ denote both the longest common subsequence and its length. It will be clear from the context which is referred to. The well-known Property 1 specifies the relation between the LCS and ED.

Property 1. Let A, B be two n-long strings, then

$$n - LCS(A,B) \leq ED(A,B) \leq 2\cdot(n - LCS(A,B)).$$

Definition 1. (LCS preserving embedding) *Let* $\mathbf{X}$ *and* $\mathbf{Y}$ *be two classes of* n*-long strings. A* LCS preserving embedding *of* $\mathbf{X}$ *into* $\mathbf{Y}$ *with* distortion ρ*, is an injective mapping* $f : \mathbf{X} \mapsto \mathbf{Y}$*, such that for every pair* $A, B \in \mathbf{X}$*,* $\rho \cdot LCS(A,B) \leq LCS(f(A), f(B)) \leq LCS(A,B)$*, where* $\rho \leq 1$*.*

Note that we require the embedding to be non-expanding. It is only allowed to have a bounded contraction factor.

Periodicity and Non-Repetitiveness. Periodicity and non-repetitiveness are two basic properties of a given string that, as we formally state in the sequel, are closely related.

Definition 2. *Let* S *be a string of length* n*.* S *is called* periodic *if* $S = P^i P'$*, for some* $2 \leq i \leq n$*, where* P *is a prefix of* S *such that* $|P| \leq n/2$*, and* P' *is a prefix of* P*. The smallest such prefix* P *is called the* period *of* S*. If* S *is not periodic it is called* aperiodic*.*

Definition 3. (A t**-substring).** *Let* S *be a string of length* n*. The* t-substring of S *starting at position* i*,* $i \leq n - t + 1$*, is the string* $S[i]S[i+1]\ldots S[i+t-1]$*.*

Definition 4. (Locally non-repetitive strings). *A string* S *is called* (t, w)-non-repetitive *if every* w *successive* t*-substrings in* S *are distinct, i.e., for each interval* $\{i, \ldots, i + w - 1\}$*, the* w *substrings of length* t *that start in this interval are distinct. If* $t = 1$ *then* S *is simply called* locally-non-repetitive*.*

In the next definition of non-repetitiveness it is required that t-substrings in the range are not only distinct, but also different enough with respect to an additional parameter d.

Definition 5. (Locally strong non-repetitiveness). *A string* S *is called* (t, w, d)-non-repetitive *if for each interval* $\{i, \ldots, i + w - 1\}$ *every pair of* t*-substrings* s_i*,* s_j *in* S *starting in this interval have* $\mathcal{H}(s_i, s_j) \geq d$*, where* $\mathcal{H}(s_i, s_j)$ *is the hamming distance between* s_i *and* s_j *(i.e. the number of indices in which* s_i *differ from* s_j*).*

Remark. Throughout the paper we refer to a wrap-around of the given string S, i.e. indices are taken modulo n, the length of the string. Thus, all t-substrings are well-defined for every t. If S is periodic then the wrap-around is defined as to continue the period from the point it is cut in the string S.

Property 2. Let S be a (t, w)-non-repetitive string, then:

1. S is a (t', w)-non-repetitive string, for every $t' > t$.
2. S is a (t, w')-non-repetitive string, for every $w' < w$.

Property 3. Let S be a string of length n, then:

1. If S is a periodic string with period length p then S is a (p, p)-non-repetitive string.
2. If S is aperiodic then S is a (n, w)-non-repetitive string, where $n/2 \leq w \leq n$.

Lemma 1. *Let S be a n-long string over alphabet Σ with period length p, then S is a $(p, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string. If S is aperiodic then S is a $(n, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string.*

Note that, Lemma 1 gives a guarantee for worst case parameters of locally strong non-repetitiveness. For a given pair of strings, the best parameters, i.e., the larger parameters w and d for which the t-substrings are strongly non-repetitive, can be much better. For example, consider a n-long string over alphabet n^ϵ, with period $p > n^\epsilon$. The lemma only assures that it is $(p, n^\epsilon/2, n^\epsilon/2)$-non-repetitive, however, it can actually be (p, p, d)-non-repetitive, for $d \geq n^\epsilon/2$.

3 Embedding Strings over Relatively Large Alphabet into Local Non-repetitive Strings

By Lemma 1, relatively large alphabet assures the existence of a large enough parameter w and a parameter t such that the t-substrings are locally strong non-repetitive, for a large enough parameter d. We will exploit this to define an embedding into (1,n/c)-non-repetitive strings, for which the solutions of Sect. 4 are applicable. This embedding has only an additive negligible distortion, if the ED is asymptotically negligible compared to the LCS size and the ratio between the alphabet size and the periodicity parameter of the string. Thus, it enables approximating large LCS in general strings over relatively large alphabet with effectively the same approximation ratio as the algorithms for (1,n/c)-non-repetitive strings, provided that the ED is not large. For clarity of exposition, a simple idea of an embedding that may have an unbearable distortion is described first. After analyzing its weaknesses it is shown how these can be overcome by defining our embedding. Finally, we discuss the algorithmic applications of this embedding.

A Naive Embedding. The idea is to exploit Property 3, namely, that every n long string S over alphabet Σ is a (t, w)-non-repetitive string for some $|\Sigma| \leq t \leq n$, $|\Sigma| \leq w \leq n$. Each new t-substring defines a new symbol (overall, a linear number of new symbols). This embedding yields a (1,n/c)-non-repetitive string where $c \leq \frac{2n}{|\Sigma|}$, and since $|\Sigma|$ is relatively large the algorithms of Sect. 4 are efficient.

We now analyze the distortion of this embedding. Given the original n-long strings A and B, denote by A', B' the strings after employing the embedding. Clearly, $LCS(A', B') \leq LCS(A, B)$ because positions with different symbols remain different. Also, each of the $n - LCS(A, B)$ symbols that do not participate in $LCS(A, B)$ affects only t substrings, thus,

$$LCS(A', B') \geq n - t(n - LCS(A, B)) = LCS(A, B) - (t - 1)(n - LCS(A, B)).$$

By Property 1 we get

$$LCS(A', B') \geq LCS(A, B) - \frac{t-1}{2} \cdot ED(A, B).$$

Thus, this embedding has an additive distortion affected both by t and $ED(A,B)$, which can both be $\Omega(n)$.

The Embedding f. Fix a random binary vector v of length $t-1$, where each coordinate is 1 with probability $\frac{2d\ln t}{|\Sigma|}$ for an arbitrarily chosen constant $d > 2$, and 0 otherwise. Note that v is well defined for relatively large alphabet, since for $|\Sigma| \geq n^{\epsilon}$ and $t \leq n$, $\frac{2d\ln t}{|\Sigma|} = o(1)$. Given an n-long string S over alphabet Σ define $f(S)$ as follows. Each location i is given a symbol $\sigma(i)$ which identifies the string $S_i, S_{i_1}, \ldots, S_{i_k}$, where $S_{i_1}, \ldots, S_{i_k}$ are the locations in the $(t-1)$-substring starting at position $i+1$ in S for which the corresponding coordinates in v are 1. Note, that there is no assumption whatsoever on any property of the original string S. Lemma 2 and Corollary 1 give the local non-repetitiveness guarantee on the string produced by the embedding f. Lemma 3 bounds the distortion of the embedding f.

Lemma 2. *Let S be a n-long string over alphabet Σ then, there exists a parameter t, $|\Sigma| \leq t \leq n$ such that $f(S)$ is $(1, |\Sigma|/2)$-non-repetitive string with probability at least $1 - 1/t^{d-2}$.*

Proof. By Lemma 1, there exists a t, $|\Sigma| \leq t \leq n$, such that S is a $(t, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string. Let i, j be any indices in S such that $|i-j| < |\Sigma|/2$, and let s_i be the t-substring starting at position i in S. By Lemma 1 we have $\mathcal{H}(s_i, s_j) \geq |\Sigma|/2$. We first claim that

$$Prob[\mathcal{H}(f(s_i), f(s_j)) = 0] \leq 1/t^{d}.$$

This is because $Prob[\mathcal{H}(f(s_i), f(s_j)) = 0] = (1 - \frac{2d\ln t}{|\Sigma|})^{|\Sigma|/2}$, if none of the $|\Sigma|/2$ coordinates in which s_i and s_j differ are chosen. Thus, by the union bound

$$Prob[\exists i,j : \mathcal{H}(f(s_i), f(s_j)) = 0] \leq 1/t^{d-2}.$$

The lemma follows.

The resulting string $f(S)$ can be checked if it is indeed a locally non-repetitive string in linear time. If it is not, the choice of v can be repeated until the result is a locally non-repetitive string. The expected number of vectors v that should be chosen is less than 2. Corollary 1 follows.

Corollary 1. *Let S be a string over alphabet Σ then, there exists a deterministic embedding f such that $f(S)$ is $(1, |\Sigma|/2)$-non-repetitive string.*

Lemma 3. *Let A, B be n-long strings over alphabet Σ, then*

$$LCS(A,B) \geq LCS(f(A), f(B)) \geq LCS(A,B) - \frac{d(t-1)\ln t}{|\Sigma|} \cdot ED(A,B)$$

Proof. First note that $LCS(A,B) \geq LCS(f(A), f(B))$, because positions with different symbols in A and B remain different in $f(A)$ and $f(B)$. We now bound the contraction factor of f. Since by the definition of the randomized embedding

f the first symbol of the i-th t-substring is always taken and the rest $i+1,\ldots,i+t-1$ locations of the i-th t-substring are taken with probability $\frac{2d\ln t}{|\Sigma|}$ for a constant $d>2$, we have:

$$\begin{aligned}
LCS(f(A),f(B)) &\geq n-(1+\frac{2(t-1)d\ln t}{|\Sigma|})(n-LCS(A,B))\\
&= LCS(A,B)-\frac{2(t-1)d\ln t}{|\Sigma|}\cdot(n-LCS(A,B))\\
&\geq LCS(A,B)-\frac{d(t-1)\ln t}{|\Sigma|}\cdot ED(A,B),
\end{aligned}$$

where the last inequality is due to Property 1.

Let $\mathbf{RL}(n,\Sigma)$ be the class of n-long strings over alphabet Σ, $|\Sigma|\geq n^{\epsilon}$, for some $\epsilon>0$. Let $\mathbf{LNR}(n)$ be the class of locally-non-repetitive n-long strings. Theorem 1 follows.

Theorem 1. *There exists an embedding $f:\mathbf{RL}(n,\Sigma)\mapsto\mathbf{LNR}(n)$ such that for every $A,B\in\mathbf{RL}(n,\Sigma)$, there exists a parameter t, $|\Sigma|\leq t\leq n$, such that $f(A),f(B)\in\mathbf{LNR}(n)$ and if $ED(A,B)=o(LCS(A,B)\cdot\frac{|\Sigma|}{t\ln t})$ then f has distortion $1-o(1)$.*

Implementation and Algorithmic Application. The discussion of efficient algorithms for computing the embedding f is postponed to Sect. 5, since the algorithms we present are also sketching algorithms, and therefore, require the relevant study from this point of view. Denote by $\gamma(n)$, the time for computing f. In Sect. 5 it is shown that $\gamma(n)=\tilde{O}(n)$. Corollary 2 is the algorithmic application of the embedding f.

Corollary 2. *Let A,B be two n-long strings over alphabet Σ. Then, there exists a parameter t, $|\Sigma|\leq t\leq n$, such that if $ED(A,B)=o(LCS(A,B)\cdot\frac{|\Sigma|}{t\ln t})$, any algorithm approximating $LCS(f(A),f(B))$ to a factor of α in $O(\beta(n))$ steps, can be used to approximate $LCS(A,B)$ to a factor of $\alpha-o(1)$ in $O(\beta(n))+\tilde{O}(n)$ steps.*

4 Approximating LCS in (1,n/c)-Non-repetitive Strings

In this section we present efficient algorithms to approximate the LCS if both strings are (1,n/c)-non-repetitive strings. The algorithms framework is based on the observation that a (1,n/c)-non-repetitive string for small values of parameter c is sufficiently close to being a permutation string (i.e., a string with distinct characters). Finding the LCS in n-long permutation strings is actually finding the Longest Increasing Subsequence (LIS) of a string over the alphabet $\{1,\ldots,n\}$, which can be done fast.

4.1 $\Theta(1/c)$-Approximation Algorithm

The algorithm first divides both input strings A and B into c blocks of size $O(n/c)$. Since A and B are (1,n/c)-non-repetitive, each of their blocks is a permutation string. Therefore, the LCS between any block of A and any block of B can be found fast using the LIS algorithm. Our algorithm exploits this fact by finding the LIS between all c^2 pairs of block of A and block of B, and chooses the pair with the best score. A detailed description of the algorithm is given in Fig. 1. Lemma 5 and Corollary 3 assure the approximation ratio of this algorithm. Lemma 4 gives its complexity guarantee. Theorem 2 follows.

```
Algorithm Approx1LCS
Input: Two strings $A$, $B$ of length $n$, a parameter $c$
1   divide $A$, $B$ into $c$ blocks of size $O(n/c)$.
2   for each pair of blocks $A_i$, $B_j$ do
3           transform into blocks $A'_i$, $B'_j$ containing only the joint alphabet symbols.
4           $\ell_{i,j} \leftarrow LIS(A'_i, B'_j)$
5   $L_{alg} \leftarrow \max \ell_{i,j}$
Output:
6   $L_{alg}$
```

Fig. 1. $\Theta(1/c)$-Approximation Algorithm for LCS in (1,n/c)-Non-Repetitive Strings

Lemma 4. *Algorithm Approx1LCS runs in $O(cn \log\log(n/c) + c^2)$ steps.*

Proof. It is a well-known fact that LIS can be computed in $(n \log\log n)$ time for n-length strings. Algorithm *Approx1LCS* computes c^2 times LIS on strings of size n/c. Therefore, the total time for steps 2-4 is $O(cn \log\log(n/c))$. Step 5 takes another c^2 steps. The lemma then follows.

Lemma 5. *Let A and B be two strings of length n, then there exists a pair of blocks A_i, B_j such that $l_{i,j} \geq \Theta(1/c) \cdot LCS(A,B)$.*

Corollary 3. *The approximation ratio of algorithm Approx1LCS is $\Theta(1/c)$.*

Theorem 2. *Let A,B be two (1,n/c)-non-repetitive strings then $LCS(A,B)$ can be approximated to a factor of $\Theta(1/c)$ in $O(c \cdot n \log\log(n/c) + c^2)$ steps.*

4.2 $\Theta(k/c)$-Approximation Algorithm

The $\Theta(1/c)$ approximation ratio of algorithm *ApproxLCS*1 is quite well if c is constant. However, as c grows it gets worse. In fact, for $c = \sqrt{n}$ it gives nothing but a trivial approximation. We thus give another algorithm with the same framework as algorithm *ApproxLCS*1, in which additional work is done (but asymptotically takes the same time) in order to improve the approximation ratio. This new algorithm does not choose only one pair of blocks with best score, but rather gather a legal sequence of pairs of blocks with total best score. A legal sequence does not contain crossing pairs. Clearly, any legal sequence defines a common subsequence of A and B. Fortunately, such a legal sequence of pairs can be found by a dynamic programming procedure in $O(c^2)$ time. We refer to this procedure by $MaximumWeightLegalSequence$. A detailed description of the algorithm is given in Fig. 2. Lemma 7 assures the approximation ratio of this algorithm. Lemma 6 gives its complexity guarantee. Theorem 3 follows.

```
ALGORITHM APPROX2LCS
Input: Two strings $A$, $B$ of length $n$, a parameter $c$
1  divide $A$, $B$ into $c$ blocks of size $O(n/c)$.
2  for each pair of blocks $A_i$, $B_j$ do
3        transform into blocks $A'_i$, $B'_j$ containing only the joint alphabet symbols.
4        $\ell_{i,j} \leftarrow LIS(A'_i, B'_j)$
5  construct a weighted bipartite graph $G =< V1 \cup V2, E >$ with weight function
         $W : E \rightarrow \mathcal{N}$, where:
         $V1 = \{i \mid A_i \textit{ is a block in } A\}$
         $V2 = \{j \mid B_j \textit{ is a block in } B\}$
         $E = \{(i,j) \mid i \in V1 \textit{ and } j \in V2\}$
         $W(i,j) = \ell_{i,j}$
6  $L_{alg} \leftarrow MaximumWeightLegalSequence(G, W)$
Output:
7  $L_{alg}$
```

Fig. 2. $\Theta(k/c)$-Approximation Algorithm for $LCS \geq kn/c$ in (1,n/c)-Non-Repetitive Strings

Lemma 6. *Algorithm Approx2LCS runs in $O(cn \log\log(n/c) + c^2)$ steps.*

Lemma 7. *Algorithm Approx2LCS approximates $LCS(A,B) \geq kn/c$ to a factor of $\Theta(k/c)$.*

Theorem 3. *Let A,B be two (1,n/c)-non-repetitive strings then $LCS(A,B) \geq kn/c$ can be approximated to a factor of $\Theta(k/c)$ in $O(c \cdot n \log\log(n/c) + c^2)$ steps.*

5 Sketching Local Non-repetitiveness

Since the performance of our method for approximating LCS rely on the extent of local non-repetitiveness parameters of the given strings, it is natural to ask how quickly can these parameters be found. The almost linear time algorithms presented in this section do not require any pre-computed information on the strings (e.g., the periodicity), and approximate the best parameters to a factor of 2. For our method of approximating the LCS of two given strings this is sufficient. However, the strength of these algorithms lies in the fact that they are sketching algorithms, i.e., they are only used once for a given string and produce a small (poly-logarithmic) size information from which the best parameters can be deduced. This use is valuable for data-bases applications, in which a query string is typically compared with many stored strings to find a similar (or the most similar) stored string. Short one-time pre-computed sketches of the stored strings save many repeated linear time scans, and thus speed-up computations.

In this section, we show that the best parameters t and w for a given pair of strings can be found by looking at $O(\log^2 n)$ size independently pre-computed *local non-repetitiveness sketches* (LNR-sketch) of the strings. The LNR-sketch gives the exact parameter w for which the best t parameter is approximated to a factor of 2. The implementation of the embedding f from Sect. 3 using the construction of *strong local non-repetitiveness sketches* (SLNR-sketch) is then described. We also show that our LNR-sketch size matches the lower bound. Finally, a lower bound on the space needed by a LNR-sketching algorithm in the streaming model is also given.

5.1 The LNR-Sketching Algorithms

If both t and w are given in advance, a trivial sketch of one bit can be built. Simply, keep the one bit answer of the check if S is a (t, w)-non-repetitive string. This check can obviously be done in time $O(tn)$, and therefore the sketching algorithm is efficient (i.e., has a polynomial time complexity). In the sequel, we assume that the t and w parameters are unknown when the sketching is done, which is the interesting case. We explain the algorithms for a given t parameter, and then use them for the case that t is not given.

Sketching with a Given t. The sketching algorithms are based on finding the minimum distance between any repeating t-substrings. This distance is returned as the w parameter. The correctness of this returned value is ensured by Property 2. The number of bits needed to store this value is $O(\log n)$. Finding the minimum distance between any repeating t-substrings can be found either by a $O(n \log^2 t)$ time deterministic algorithm or by a $O(n)$ time randomized algorithm. The deterministic algorithm uses a renaming process as in the string matching algorithm of Karp-Miller-Rosenberg [14]. It is usually assumed, for convenience, that t is a power of 2. This assumption can be removed by using standard splitting techniques, while adding only a $O(\log t)$ factor to the $O(n \log t)$ complexity. The randomized algorithm uses the Rabin-Karp string matching algorithm [15] to produce a distinct polynomial representing each t-substring with high probability. In both the deterministic and the randomized algorithm after the "names" representing the n t-substrings are determined all is needed is a linear scan to find the minimum distance between repeating "names".

Sketching with Unknown t. In order to have the w for every t, we find the exact parameter w for every $t = 2^i$, $0 \leq i \leq \log n$. For each such t we use the algorithms described above for a given t. Since we only do that for $O(\log n)$ values of t, and for each the sketch size is $O(\log n)$ we get a total $O(\log^2 n)$ sketch size. For each value t , the w parameter is the one stored for the closest power of two that is less than or equal to t. The correctness of this value is ensured by Property 2.

Theorem 4. *Let A, B be n long strings, then, there exist (almost) linear algorithms giving LNR-sketch of size $O(\log^2 n)$ enabling finding the maximum w and approximating to a factor of 2 the minimum t for which A and B are both (t, w)-non-repetitive.*

5.2 Sketching Strong Local Non-repetitiveness

The embedding from strings over relatively large alphabet into (1,n/c)-non-repetitive strings described in Sect. 3 requires local non-repetitiveness under the choices of the randomized vectors v, which is not detected by the algorithms described in Sect. 5.1. Nevertheless, we show that the ideas of the sketching algorithms described in Sect. 5.1 can be used also for this case. We call it *strong local non-repetitiveness sketch* (SLNR-sketch)[2]. By Corollary 1, a constant number of vectors v are enough so that two given strings can be compared using the same vector v. Therefore, in the sequel we ignore the fact that the algorithm is repeated for each choice of v and keep each of the resulting sketches[3].

To this end, the substrings as defined by the binary vector v (defined in Sect. 3), are considered. Observe that both the deterministic and randomized sketching algorithms described in Sect. 5.1 work as well for non-contiguous strings. Such non-standard use of the KMR algorithm also appears in [2]. Note that the binary vector v depends only on Σ and t and is independent of S. Thus, the definition of the vector can be done in the sketching time. Also, note that in order to be able to compare any two strings (with possibly different size of joint alphabet and different t parameter) we must define a v vector for each possible pair. To cover all possible values of Σ, for each t a power of two, $O(\log^2 n)$ vectors v (for each Σ a power of two and t a power of two) are computed. Once a specific vector v is defined, the sketch for non-repetitiveness can be done as explained in Sect. 5.1. This would take $O(n \log t)$ because here t is a power of 2. Since $O(\log^2 n)$ sketches of size $O(\log n)$ are used, Theorem 5 follows.

Theorem 5. (The embedding implementation) *Let A, B be n long strings, then, there exist (almost) linear algorithms giving SLNR-sketch of size $O(\log^3 n)$ enabling finding the maximum w and approximating to a factor of 2 the minimum t for which $f(A)$ and $f(B)$ are both $(1, w)$-non-repetitive.*

5.3 Lower Bound on LNR-Sketch Size

Note that the w parameter as a function of t is a nondecreasing monotone function that take values on the range $\{1, \ldots, n\}$. We show a feasible set of

[2] Should not be confused with the local strong non-repetitiveness.

[3] A data-base application requires another logarithmic factor in the size of the database to assure that every pair of strings can be compared using the same vector v.

monotone sequences, i.e., monotone sequences that represent w as a function of t for some string. The size of this set gives a lower bound on the number of bits needed to represent a LNR-sketch.

Lemma 8. *The size of the feasible set is at least* $(\frac{n}{\log n})^{\log n}$.

The next theorem is an immediate corollary of Lemma 8.

Theorem 6. *Any LNR-sketch of* n*-length string requires* $\Omega(\log^2 n)$ *bits.*

5.4 A $\Omega(n/\log n)$ Space Lower Bound of LNR-Sketching Algorithms in Streaming Model

We now show that LNR-sketch cannot be done in streaming model. Consider the following one-round two-party communication setting for the problem. Alice has a string $S1$ of length n and Bob has a string $S2$ of length n. Alice and Bob should decide whether there exists a t-substring in $S1$ repeating in $S2$ while Alice may pass at most k bits to Bob. We call this setting the *repeating t-substring problem.* Lemma 9 shows that $k = \Omega(n)$. Theorem 7 follows.

Lemma 9. *The repeating* t*-substring problem requires passing* $\Omega(n)$ *bits.*

Theorem 7. *Any LNR-sketching deterministic algorithm in streaming model requires* $\Omega(n/\log n)$ *space.*

6 Conclusions

We show how embedding strings over relatively large alphabet into local non-repetitive strings can be exploited for approximating LCS in strictly sub-quadratic time. An important contribution of the paper is also conceptual in suggesting a different point of view that make the problem algorithmically easier. Our technique works well provided that the dissimilarity in terms of the edit distance of the given strings is not too large. It is still an open question wether LCS can be well-approximated in strings over relatively large alphabet with large dissimilarity.

References

1. Aho, A.V., Hirschberg, D.S., Ulman, J.D.: Bounds on the complexity on the longest common subsequence problem. JACM 23(1), 1–12 (1976)
2. Amir, A., Aumann, Y., Kapah, O., Levy, A., Porat, E.: Approximate string matching with address bit errors. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 118–129. Springer, Heidelberg (2008)
3. Andoni, A., Krauthgamer, R.: The computational hardness of estimating edit distance. In: FOCS, pp. 724–734 (2007)
4. Apostolico, A., Guerra, C.: The longest common subsequence problem revisited. Algorithmica 2, 315–336 (1987)

5. Baker, B.S., Giancarlo, R.: Longest common subsequence from fragments via sparse dynamic programming. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 79–90. Springer, Heidelberg (1998)
6. Bar-Yossef, Z., Jayram, T.S., Krauthgamer, R., Kumar, R.: Approximating edit distance efficiently. In: FOCS, pp. 550–559 (2004)
7. Batu, T., Ergün, F., Sahinalp, C.: Oblivious string embeddings and edit distance approximation. In: SODA, pp. 792–801 (2006)
8. Charikar, M., Krauthgamer, R.: Embedding the Ulam metric into ℓ_1. Theory of Computing 2, 207–224 (2006)
9. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. SIAM J. Comput. 32(5), 1654–1673 (2003)
10. Crochemore, M., Porat, E.: Computing a longest increasing subsequence of length k in time o(n log log k). In: Gelenbe, E., Abramsky, S., Sassone, V. (eds.) Visions of computer science, Swindon, UK, pp. 69–74. The British Computer Society (2008)
11. Gusfield, D.: Algorithms on strings, trees and sequences. Cambridge University Press, Cambridge (1997)
12. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. JACM 24(4), 664–675 (1977)
13. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. CACM 20, 350–353 (1977)
14. Karp, R., Miller, R., Rosenberg, A.: Rapid identification of repeated patterns in strings, arrays and trees. Symposium on the Theory of Computing 4, 125–136 (1972)
15. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
16. Landau, G.M., Scheiber, B., Ziv-Ukelson, M.: Sparse LCS commom substring alignment. Information Processing Letters 88(6), 259–270 (2003)
17. Landau, G.M., Vishkin, U.: Fast string matching with k differences. Journal of Computer and System Sciences 37(1), 63–78 (1988)
18. Landau, G.M., Ziv-Ukelson, M.: On the common substring alignment problem. Journal of Algorithms 41(2), 338–359 (2001)
19. Masek, W.J., Paterson, M.S.: A faster algorithm for computing string edit distances. JCSS 20, 18–31 (1980)
20. Ostrovsky, R., Rabani, Y.: Low distortion embeddings for edit distance. In: Proceedings of the 37th annual ACM symposium on Theory of computing (STOC), pp. 218–224 (2005)
21. Sankoff, D.: Matching sequences under deletion/insertion constraints. Pro. Nat. Acad. Sct. USA 69, 4–6 (1972)
22. Wagner, R.A., Fischer, M.J.: The string to string correction problem. JACM 21(1), 168–173 (1974)

An Efficient Matching Algorithm for Encoded DNA Sequences and Binary Strings

Simone Faro[1] and Thierry Lecroq[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania, Italy
[2] University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
`faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr`

Abstract. We present a new efficient algorithm for exact matching in encoded DNA sequences and on binary strings. Our algorithm combines a multi-pattern version of the Bndm algorithm and a simplified version of the Commentz-Walter algorithm. We performed also experimental comparisons with the most efficient algorithms presented in the literature. Experimental results show that the newly presented algorithm outperforms existing solutions in most cases.

Keywords: string matching, binary strings, DNA sequences, experimental algorithms, compressed text processing.

1 Introduction

In this article we consider the problem of searching for all exact occurrences of a pattern p, of length m, in a text t, of length n, where p and t are both bitstreams. In particular we consider the cases where each character of p and t consists in a single bit (binary sequences) or a couple of bits (encoded DNA sequences).

Matching binary data is an interesting problem in computer science, since binary data are omnipresent in telecom and computer network applications. Many formats for data exchange between nodes in distributed computer systems as well as most network protocols use binary representations. Binary images often arise in digital image processing as masks or as the results of certain operations such as segmentation, thresholding and dithering. Moreover some input/output devices, such as laser printers and fax machines, can only handle binary images.

Also DNA sequences can be handled as bitstreams. Since a DNA sequence is constructed with four bases (`A`, `C`, `T`, and `G`), an efficient fixed-length encoding method [7] can be used, where only two bits for each character are needed.

In molecular biology, DNA sequences are the fundamental information for each species and a comparison between DNA sequences is an interesting and basic problem. There are various kinds of comparison tools which provide approximate matching. However most of them are based on exact matching in order to speed up the process. Moreover because the total number of sequences is rapidly increasing, efficient methods are needed, not only for fast matching but also for efficient sequences storage. Thus the need for fast matching algorithms on encoded sequences.

The first non trivial algorithm for searching on bitstreams was presented in [8] by Klein and Ben-Nissan. They proposed an efficient variant of the Boyer-Moore [2] algorithm for the binary case without referring to bits. The algorithm is projected to process only entire blocks such as bytes or words and achieves a significantly reduction in the number of text character inspections.

Recently in [5] two efficient algorithms have been presented for the problem adapted to completely avoid any reference to bits allowing to process pattern and text byte by byte. The first solution, called Binary-Hash-Matching algorithm, is an adaptation of the q-Hash family [10] for exact pattern matching to the case of binary strings. The second solution, called Binary-Skip-Search algorithm, extends the Skip-Search algorithm [3] for the single exact pattern matching problem. Experimental results conducted in [5] on various conditions showed that the proposed algorithms perform better than existing solutions and even than the most effective algorithms for standard pattern matching.

All previous solutions have been proposed for searching on binary data but can be easily adapted to the case of encoded DNA sequences with minor changes.

The Fed algorithm [7] (Fast matching with Encoded DNA sequences) is a string matching algorithm specifically tuned for matching DNA sequences compressed using a fixed-length encoding. Specifically, the Fed algorithm combines a multi-pattern version of the Quick-Search algorithm [12] and a simplified version of the Commentz-Walter algorithm [4]. However, its strategy is general enough to be adapted also for matching binary sequences.

In this article we present a new efficient algorithm for matching on binary strings and encoded DNA sequences which, despite its $\mathcal{O}(nm)$ worst case time complexity, obtains very good results in practical cases.

The rest of the paper is organized as follows. In Section 2, we describe in details the new solution tuned for binary data. Next, in Section 3, we explain how to handle also encoded DNA sequences. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 4. Finally, we draw our conclusions in Section 5.

2 A New Efficient Algorithm

Before entering into details, we need a bit of notations and terminology. A string p of length $m \geq 0$ is represented as a finite array $p[0\,..\,m-1]$ of characters from a finite alphabet Σ. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $p[i]$ we denote the $(i+1)$-th character of p, for $0 \leq i < m$. Likewise, by $p[i\,..\,j]$ we denote the substring of p contained between the $(i+1)$-th and the $(j+1)$-th characters of p, for $0 \leq i \leq j < m$. A substring of p of length ℓ is called a ℓ-substring of p.

A string p over the binary alphabet $\Sigma = \{0, 1\}$ is said to be a *binary string* and is represented as a binary vector $p[0\,..\,m-1]$, whose elements are bits. Binary vectors are usually structured in blocks of k bits, typically bytes ($k = 8$), halfwords ($k = 16$) or words ($k = 32$), which can be processed at the cost of a single operation. If p is a binary string of length m we use the symbol $P[i]$ to indicate the $(i+1)$-th block of p and use $p[i]$ to indicate the $(i+1)$-th bit of p. If B is a block of k bits we indicate with symbol B_j the j-th bit of B, with $0 \leq j < k$. Thus, for $i = 0, \ldots, m-1$ we have $p[i] = P[\lfloor i/k \rfloor]_{i \bmod k}$.

In this section we present a new efficient algorithm, called Bfl algorithm (Binary-Faro-Lecroq), for searching on binary strings and encoded DNA sequences. The proposed algorithm exploits the block structure of text and pattern to speed up the searching phase avoiding to work with bitwise operations. The core of the algorithm is based on a multiple-pattern version of the Bndm algorithm [11] (Backward Nondeterministic Dawg Match) for the single exact pattern matching problem, which makes use of bit-parallelism [1]. Moreover the algorithm implements also an efficient shift strategy based on a simplified version of the Commentz-Walter algorithm [4].

During the preprocessing phase the algorithm constructs a set of tables that can be accessed later, during the searching phase, in order to speed up the overall performances. The procedures used in the preprocessing phase are presented in Figure 2. Specifically the algorithm computes: **(1)** a table of several copies of p, in order to process text and pattern block by block (as in [8]); **(2)** a bit mask-vector used to implement a multi-pattern version of the Bndm algorithm; **(3)** an index-list table, λ, in order to identify candidate alignments during the searching phase; **(4)** a shift table ls, based on the bad-character heuristic, with the aim of increasing the shift advancements.

In what follows we suppose that the block size k is fixed, so that all references to both text and pattern will only be to entire blocks of k bits. We refer to a k-bit block as a *byte*, though larger values than $k = 8$ could be supported as well. Moreover we suppose that $T[i]$ and $P[i]$ denote, respectively, the $(i+1)$-th byte of the text and of the pattern, starting for $i = 0$ with both text and pattern aligned at the leftmost bit of the first byte. Since the lengths in bits of both text and pattern are not necessarily multiples of k, the last byte may be only partially defined. In particular if the pattern has length m then its last byte is that of position $\lceil m/k \rceil$ and only the leftmost $(m \bmod k)$ bits of the last byte are defined. We suppose that the undefined bits of the last byte are set to 0.

Finally we notice that the following description of the algorithm is tuned for processing binary data where each character consists in a single bit. In the next section we explain how to handle also encoded DNA sequences.

Preprocessing of the pattern

In the preprocessing phase we define several copies of the pattern, identified by a set of indexes $\mathbf{P}$, and memorized in the form of a matrix of bytes, *Patt*, of size $k \times (\lceil m/k \rceil + 1)$. Each index $i \in \mathbf{P}$ refers to a row of the matrix *Patt* containing a copy of the pattern shifted by i position to the right. In each pattern $Patt[i]$, for $i \in \mathbf{P}$, the i leftmost bits of the first byte remain undefined and are set to 0. Similarly the rightmost $((k - ((m+i) \bmod k)) \bmod k)$ bits of the last byte are set to 0. Formally the j-th bit of the byte $Patt[i, h]$ is defined by

$$Patt[i,h]_j = \begin{cases} p[kh - i + j] & \text{if } 0 \leq kh - i + j < m \\ 0 & \text{otherwise} \end{cases}.$$

Patt i	0	1	2	3
0	11001011	00101100	10110000	
1	01100101	10010110	01011000	
2	00110010	11001011	00101100	
3	00011001	01100101	10010110	
4	00001100	10110010	11001011	00000000
5	00000110	01011001	01100101	10000000
6	00000011	00101100	10110010	11000000
7	00000001	10010110	01011001	01100000

m_i	s_i	m'_i
2	0	2
2	1	1
2	1	1
2	1	2
3	1	2
3	1	2
3	1	2
3	1	2

$F1$	$F2$
11111111	11111000
01111111	11111100
00111111	11111110
00011111	11111111
00001111	10000000
00000111	11000000
00000011	11100000
00000001	11110000

Fig. 1. Precomputed tables for a pattern p =110010110010110010110 of length $m = 21$ and block size $k = 8$. Blocks containing a k-substring of p are presented with light gray background color.

for $0 \le j < k$, $0 \le h < \lceil (m + i)/k \rceil$ and $i \in \mathbf{P}$.

For binary strings we have $\mathbf{P} = \{i \mid 0 \le i < k\}$. Observe that each k-substring of the pattern appears once in the table *Patt*. In particular, the k-substring starting at position j of p is memorized in $Patt[k - (j \mod k), \lceil j/k \rceil]$.

We indicate with symbol m_i, for each $i \in \mathbf{P}$, the index of the last byte of the pattern $Patt[i]$, i.e., $m_i = \lceil (m + i)/k \rceil - 1$. Moreover we define the values s_i and m'_i which represent, respectively, the position of the first byte in $Patt[i]$ containing a k-substring of p and the number of bytes in $Patt[i]$ containing k-substrings of p. Observe that $s_i = 0$ if $i = 0$, while $s_i = 1$ when $i > 0$.

The Bfl algorithm uses bytes in the matrix *Patt* to compare the pattern block by block against the text, for any possible alignment of the pattern. However when comparing the first or the last byte of P against its counterpart in the text, the bit positions not belonging to the pattern have to be neutralized. For this purpose we define two vectors, $F1$ and $F2$, containing binary masks of length k. Formally, for each $i \in \mathbf{P}$,

$$F1[i]_j = \begin{cases} 1 & \text{if } i \le j < k \\ 0 & \text{otherwise} \end{cases}, \; F2[i]_j = \begin{cases} 1 & \text{if } 0 \le j \le (m + i - 1) \mod k \\ 0 & \text{otherwise} \end{cases}.$$

Figure 1 shows the precomputed tables defined above for a pattern of length $m = 21$ and $k = 8$.

Definition of the NFA and construction of the bit mask vector

The Bfl algorithm uses bit-parallelism to simulate the behavior of a nondeterministic finite state automaton constructed over the set of patterns identified by $\mathbf{P}$. However, in order to let the automaton fit in a single machine word of size ω, only the substrings $Patt[i][s_i \dots s_i + m' - 1]$ are handled by the automaton, for each $i \in \mathbf{P}$, where $m' = \min(\{m'_i \mid i \in \mathbf{P}\} \cup \{\omega\})$.

Specifically the NFA constructed by the Bfl algorithm has $m' + 1$ different states, say $Q = \{0, 1, 2, 3, \dots, m'\}$, and m' different transitions. In particular each state q, with $0 < q \le m'$, has a transition towards state $q - 1$ labeled with the class of characters $\{Patt[i][s_i + q] \mid i \in \mathbf{P}\}$. State m' is the initial state.

In order to simulate the NFA the algorithm initializes a bit mask $M[B]$ of dimension ω, for each block $B \in \{0 \dots 2^k - 1\}$. In particular the j-th bit of $M[B]$

```
Preprocess $(Patt, k)$
1. for $i \leftarrow 0$ to $k-1$ do
2.    if $i = 0$ then $s_i \leftarrow 0$ else $s_i \leftarrow 1$
3.    $m_i \leftarrow \lceil (m+i)/k \rceil - 1$
4.    for $h \leftarrow 0$ to $m_i$ do
5.       $Patt[i,h] \leftarrow (P[h] \gg i)$
6.       if $h > 0$ then
7.          $Patt[i,h] \leftarrow Patt[i,h] \mid (P[h-1] \ll (k-i))$
8.    $F1[i] \leftarrow 1^k \gg i$
9.    $F2[i] \leftarrow 1^k \ll k - ((m+i) \mod k)$
10.   if $F2[i] = 1^k$ then $m'_i \leftarrow m_i - s_i + 1$
11.   else $m'_i \leftarrow m_i - s_i$
12. return $(Patt, Mask)$

Initialize-Bit-Mask$(Patt, k)$
1. for $B \leftarrow 0$ to $2^k - 1$ do $M[B] = 0$
2. for $i \leftarrow 0$ to $k$ do
3.    $A \leftarrow 10^{\omega-1}$
4.    for $j = 0$ to $m' - 1$ do
5.       $B \leftarrow Patt[i, s_i + m' - j - 1]$
6.       $M[B] \leftarrow M[B] \mid A$
7.       $A \leftarrow A \gg 1$
8. return $M$

Compute-Long-Shift$(Patt, k)$
1. for $B \leftarrow 0$ to $2^k - 1$ do
2.    $ls[B] = 2m' - 1$
3. $i \leftarrow 0$
4. $h \leftarrow 0$
5. for $j \leftarrow 0$ to $m - k$ do
6.    $ls[Patt[i,h]] \leftarrow s_i + 2m' - h - 2$
7.    $i \leftarrow i - 1$
8.    if $i < 0$ then
9.       $i \leftarrow 7$
10.      $h \leftarrow h + 1$
11. return $ls$

Compute-Index-List$(Patt, k)$
1. for $B \leftarrow 0$ to $2^k - 1$ do
2.    $\lambda[B] = \emptyset$
3. $i \leftarrow 0$
4. $h \leftarrow 0$
5. for $i \leftarrow 0$ to $k$ do
6.    $B \leftarrow Patt[i, s_i + m' - 1]$
7.    $\lambda[B] \leftarrow \lambda[B] \cup \{i\}$
8. return $\lambda$
```

Fig. 2. Procedures in the preprocessing phase of the Bfl algorithm for binary strings

is set to 1 if the block B appears at position $s_i + m' - j - 1$ in, at least, one of the patterns $Patt[i]$, with $i \in \mathbf{P}$. Otherwise the j-th bit of $M[B]$ is set to 0.

For each block $B \in \{0 \dots 2^k - 1\}$, the definition of the bit mask $M[B]$ can be done in two steps. First we define, for each $i \in \mathbf{P}$, a bit mask $M_i[B]$ where, for $0 \le j < \omega$,

$$M_i[B]_j = \begin{cases} 1 & \text{if } j < m' \text{ and } Patt[i, s_i + m' - j - 1] = B \\ 0 & \text{otherwise.} \end{cases}$$

Then the j-th bit of the mask $M[B]$, with $0 \le j < \omega$, is defined as

$$M[B]_j = (M_0[B]_j \mid M_1[B]_j \mid \dots \mid M_{k-1}[B]_j)\,.$$

Construction of the index list

The automaton defined above recognizes also words that are not substrings of the pattern. Formally the automaton recognizes any block sequence x, of length $\ell \le m'$, of the form $x = x_0.x_1 \dots x_{\ell-1}$ where, for $0 \le j < \ell$,

$$x_j \in \{Patt[i][s_i + m' - \ell + j] \mid i \in \mathbf{P}\}.$$

Despite this fact, the automaton can be used to search for a pattern in a text. In particular when a candidate substring is found, the algorithm could naively check for the occurrence of any pattern $Patt[i]$, with $i \in \mathbf{P}$.

However, in order to make a filter the algorithm maintains, for each block $B \in \{0 \dots 2^k - 1\}$, a linked list λ which is used to find candidate patterns.

In particular, for each block $B \in \{0 \dots 2^k - 1\}$, the entry $\lambda[B]$ is a set of indexes, defined by $\lambda[B] = \{i \mid i \in \mathbf{P} \text{ and } P[i][s_i + m' - 1] = B\}$.

Thus, when a block sequence is recognized by the automaton, ending at block position j of the text, the algorithm naively checks for the occurrence of any pattern $Patt[i]$, with $i \in \lambda[T[j]]$.

In practical cases each set in the table can be implemented as a linked list.

Construction of the shift table

The BFL algorithm makes also use of a *long shift* rule which is a multi-pattern version of the original bad-character shift heuristic, improved with an efficient look-ahead. The shift rule defined here is used by the algorithm when no substring is recognized while scanning the window of the text from right to left. In such a case the current window of the text can be safely advanced by $m' - 1$ positions to the right.

Observe that, if j is the ending position of the current window of the text, the block at position $j + m' - 1$ is always involved in the next alignment. Thus we can use it to compute the next window alignment.

Specifically the algorithm computes a *long shift* table $ls() : \{0, \dots, 2^k - 1\} \rightarrow \{m' - 1, \dots, 2 \times m' - 1\}$, whose definition can be done in two steps.

First, for each pattern $Patt[i]$, with $i \in \mathbf{P}$, we compute a shift table $sh[i, B]$, where, for each $B \in \{0 \dots 2^k - 1\}$,

$$sh[i, B] = \min(\{m'\} \cup \{s_i + m' - h - 1 \mid Patt[i, h] = B \text{ and } s_i \leq h < s_i + m'\}).$$

Then, for each $B \in \{0 \dots 2^k - 1\}$, the long shift table ls is defined by

$$ls[B] = \min\{sh[i, B] \mid i \in \mathbf{P}\} + m' - 1.$$

Thus the next alignment to be processed in the text is that ending at position $j + ls[T[s + m' - 1]]$.

The searching phase

The searching phase of the BFL algorithm can be divided into two parts: a match phase and a shift phase. The pseudocode of the algorithm is shown in Figure 3.

The NFA is represented by a state vector D of size m' (the first state of the automaton is not represented). Like in the SBNDM algorithm [6], the BFL algorithm starts each iteration with a test of two consecutive text characters and implements a fast-loop to obtain better results on average (LINE 7). Such a fast loop makes use of the long shift table to compute the next window alignment.

In the match phase the same kind of right to left scan in a window of size m', ending at position j in the text, is performed as in the BNDM algorithm (LINE 9). The state vector is updated in a similar fashion as in the SHIFT-AND algorithm [1]. If the state vector D is equal to 0 after $\ell + 1$ updates of D, then a word of length ℓ has been recognized by the automaton. If $\ell = m'$ a candidate alignment has been found (LINE 12) and the algorithm naively checks the occurrence of any pattern contained in the index list $\lambda[T[j]]$ (LINE 13).

```
BFL $(P, m, T, n)$
1. $Patt \leftarrow$ Preprocess($P$, $m$, $k$)
2. $M \leftarrow$ Initialize-Bit-Mask($Patt$, $k$)
3. $\lambda \leftarrow$ Compute-Index-List($Patt$, $k$)
4. $ls \leftarrow$ Compute-Long-Shift($Patt$, $k$)
5. $j \leftarrow \lfloor m/k \rfloor$
6. while $j < \lceil n/k \rceil$ do
7.     while $((M[T[j]] \ll 1)$ & $M[T[j-1]]) = 0$ do $j \leftarrow j + ls[T[j + m' - 1]]$
8.     $pos \leftarrow j$
9.     $D \leftarrow (M[T[j]] \ll 1)$ & $M[T[j-1]]$
10.    while $D \leftarrow (D \ll 1)$ & $M[T[j-2]]$ do $j \leftarrow j - 1$
11.    $j \leftarrow j + m'$
12.    if $j = pos$ then
13.        for each $i \in \lambda[T[j]]$ do
14.            $h \leftarrow 1$, $s \leftarrow j - m' - s_i + 1$
15.            while $h < m_i$ and $Patt[i, h] = T[s + h]$ do $h \leftarrow h + 1$
16.            if   $h = m_i$ and
17.                 $Patt[i, 0] = (T[s]$ & $F1[i])$ and
18.                 $Patt[i, h] = (T[s + h]$ & $F2[i])$
19.            then Output($s$)
```

Fig. 3. The searching phase of the BFL algorithm

In all cases, after the match phase, the index j is advanced of $m' - \ell + 1$ to the right and the algorithm restarts its computation with the shift phase.

If a candidate alignment is found for a text position j, for each index $i \in \lambda[T[j]]$, the algorithm uses the precomputed table $Patt[i]$ to check whether $s = j - m' - s_i + 1$ is a valid shift. Specifically the algorithm reports a match if the following three conditions hold (LINES 13-18):

1. $Patt[i, h] = T[s + h]$, for $h = s_i, ..., s_i + m_i - 1$ and
2. $Patt[i, 0] = T[s]$ & $F1[i]$ and
3. $Patt[i, m_i] = T[s + m_i]$ & $F2[i]$.

After the matching phase the algorithm restarts with the shift phase.

Complexity issues

In this section we analyze the time complexity of the newly presented algorithm: **(1)** Procedure PREPROCESS requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space; **(2)** procedure INITIALIZE-BIT-MASK, used to initialize the vector of bit masks M, requires $\mathcal{O}(2^k + km)$ time and $\mathcal{O}(2^k)$ extra-space; **(3)** procedure COMPUTE-INDEX-LIST, used to construct the index list λ, requires $\mathcal{O}(2^k + k)$ time and $\mathcal{O}(2^k)$ extra-space; **(4)** procedure COMPUTE-LONG-SHIFT, used to construct the long shift table ls, requires $\mathcal{O}(2^k + m)$ time and $\mathcal{O}(2^k)$ extra-space; finally **(5)** the searching phase, shown in Figure 3, takes $\mathcal{O}(\lceil n/k \rceil \lceil m/k \rceil k) = \mathcal{O}(nm)$ time. Thus the BFL algorithm has a $\mathcal{O}(2^k + nm)$ overall time complexity and requires $\mathcal{O}(2^k)$ extra-space in the worst case.

3 Handling Encoded DNA Sequences

In a fix-length encoded DNA sequence each base is represented by a couple of bits. Specifically we define a map ζ which associates to any character in the set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ an element in the set $\{\mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}\}$. Thus a DNA sequence γ can be represented with a bitstream $t = \zeta(\gamma)$ of $(2 \times |\gamma|)$ bits.

Due to the structure of the encoded sequence t, any occurrence of a given encoded pattern p, starts at an even position of the text. This suggests that only even alignments of the pattern have to be processed.

The only change to be applied, when handling encoded DNA sequences, is in the preprocessing of the set of patterns. Specifically the set $\mathbf{P}$ is defined by

$$\mathbf{P} = \{i \mid 0 \leq i < k \text{ and } (i \mod 2) = 0\}$$

For instance, if each block consists of $k = 8$ bits, we have $\mathbf{P} = \{0, 2, 4, 6\}$.

4 Experimental Results

Here we present experimental data which allow to compare, in terms of running time, the following string matching algorithms on binary strings and encoded DNA sequences: the Binary-Boyer-Moore algorithm (BBM) [8] by Klein and Ben-Nissan, the Binary-Hash-Matching algorithm (BHM) [5], the Binary-Skip-Search algorithm (BSKS) [5], Fed algorithm (FED) [7] and the new Bfl (BFL) algorithm. All algorithms have been implemented in the **C** programming language and were used to search for the same binary strings in large fixed text buffers on a PC with Intel Core2 processor of 1.66GHz with 1GB memory. Moreover we use a word size $\omega = 32$ and a block size $k = 8$.

To simulate the different conditions which can arise when processing binary data the algorithms have been tested on two $\mathsf{Rand}(1/0)_\gamma$ problems, with a different distribution of zeros and ones. For the case of compressed strings it is quite reasonable to assume a uniform distribution of characters. For compression scheme using Huffman coding, such randomness has been shown to hold in [9]. In contrast when processing binary images we expect a non-uniform distribution of characters. For instance in a fax-image usually more than 90% of the total number of bits is set to zero.

In particular each $\mathsf{Rand}(1/0)_\gamma$ problem consists of searching a set of 1000 random patterns of a given length in a random binary text of 4×10^6 bits. The distribution of characters depends on the value of the parameter γ: bit 0 appears with a percentage equal to γ%.

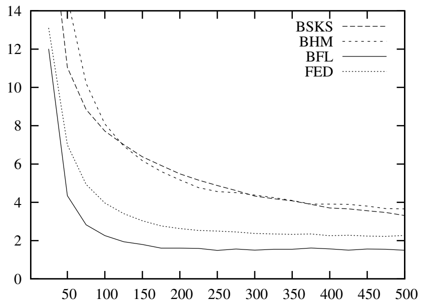
The genome we used for tests on encoded DNA sequences is a sequence of 4, 638, 690 base pairs of *Escherichia coli*. We used the encoded version of the file E.coli of the Large Canterbury Corpus[1].

Searching have been performed for patterns, of length m from 25 to 500, which have been taken as substring of the text at random starting positions.

[1] http://www.data-compression.info/Corpora/CanterburyCorpus/

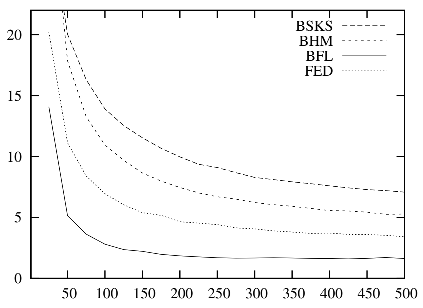
In the following tables, running times are expressed in hundredths of seconds. Best results are bold faced.

m	BBM	BSKS	BHM	BFL	FED
25	161.905	19.484	28.625	**12.014**	13.107
50	90.109	11.047	14.671	**4.344**	7.000
75	70.718	8.846	10.220	**2.828**	4.936
100	65.797	7.720	8.094	**2.264**	3.968
125	58.780	7.016	6.939	**1.938**	3.406
150	52.593	6.375	6.171	**1.798**	3.032
200	42.032	5.484	5.171	**1.609**	2.625
250	50.751	4.875	4.563	**1.485**	2.500
300	47.564	4.327	4.375	**1.498**	2.375
350	45.498	4.079	4.094	**1.546**	2.328
400	42.502	3.702	3.904	**1.564**	2.253
450	45.344	3.562	3.800	**1.562**	2.234
500	44.345	3.311	3.658	**1.497**	2.267



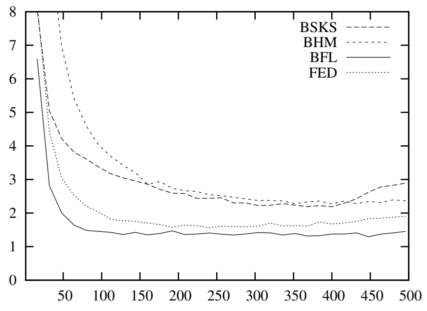
Experimental results for a $\mathsf{Rand}(0/1)_{50}$ problem

m	BBM	BSKS	BHM	BFL	FED
25	188.469	29.842	33.110	**14.095**	20.219
50	112.720	20.031	17.860	**5.142**	11.125
75	88.953	16.299	13.251	**3.624**	8.390
100	82.360	13.909	10.938	**2.797**	6.938
125	74.671	12.531	9.686	**2.358**	6.032
150	69.875	11.531	8.641	**2.218**	5.389
200	58.952	9.967	7.452	**1.843**	4.641
250	64.921	9.093	6.690	**1.689**	4.406
300	61.219	8.283	6.218	**1.671**	4.063
350	58.141	7.921	5.908	**1.670**	3.796
400	54.420	7.595	5.563	**1.624**	3.718
450	57.402	7.284	5.423	**1.642**	3.594
500	55.296	7.077	5.281	**1.625**	3.405



Experimental results for a $\mathsf{Rand}(0/1)_{70}$ problem

m	BBM	BSKS	BHM	BFL	FED
16	41.266	8.062	19.407	**6.594**	8.249
32	28.955	5.046	10.046	**2.814**	4.422
64	29.485	3.813	5.420	**1.641**	2.533
96	26.764	3.375	4.031	**1.453**	2.032
128	26.436	3.047	3.422	**1.361**	1.766
160	24.577	2.859	2.862	**1.347**	1.701
192	25.624	2.592	2.733	**1.469**	1.578
224	33.170	2.438	2.641	**1.373**	1.623
256	28.595	2.453	2.517	**1.372**	1.608
288	26.421	2.299	2.421	**1.377**	1.593
320	27.596	2.234	2.374	**1.407**	1.703
352	24.251	2.235	2.281	**1.391**	1.625
384	23.593	2.221	2.359	**1.327**	1.734
448	24.063	2.626	2.343	**1.294**	1.830
496	24.659	2.891	2.362	**1.452**	1.906



Experimental results for an encoded DNA sequence

Experimental results show that the Bfl algorithm obtains the best run-time performance in all cases. In particular it turns out that the Bfl algorithm is two times faster than the Fed algorithm when searching on binary data.

In the case of encoded DNA sequences such a difference is less evident and the Bfl algorithm turns out to be 15% faster than the Fed algorithm which is specifically tuned for matching encoded DNA sequences.

Finally we report that, in most cases, the FED algorithm turns out to be more efficient than the BINARY-SKIP-SEARCH and BINARY-HASH-MATCHING algorithms, for both binary data and encoded DNA sequences.

5 Conclusion

An efficient algorithm for exact matching on binary strings and encoded DNA sequences has been presented. The algorithm combines a multi-pattern version of the BNDM algorithm with a simplified shift strategy of COMMENTZ-WALTER algorithm. Moreover it exploits the block structure of the binary strings and process text and pattern with no use of any bit manipulations. From our experimental results it turns out that the presented algorithm is the most effective in practical cases and outperforms existing solutions especially in the case of binary data.

References

1. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Commun. ACM 35(10), 74–82 (1992)
2. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
3. Charras, C., Lecroq, T., Pehoushek, J.D.: A very fast string matching algorithm for small alphabets and long patterns. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 55–64. Springer, Heidelberg (1998)
4. Commentz-Walter, B.: A string matching algorithm fast on the average. In: Maurer, H.A. (ed.) ICALP 1979. LNCS, vol. 71, pp. 118–132. Springer, Heidelberg (1979)
5. Faro, S., Lecroq, T.: Efficient pattern matching on binary strings. In: Current Trends in Theory and Practice of Computer Science, Poster (2009)
6. Holub, J., Durian, B.: Fast variants of bit parallel approach to suffix automata. Talk given in: The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation (2005), `http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf`
7. Kim, J.W., Kim, E., Park, K.: Fast matching method for DNA sequences. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 271–281. Springer, Heidelberg (2007)
8. Klein, S.T., Ben-Nissan, M.K.: Accelerating Boyer Moore searches on binary texts. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 130–143. Springer, Heidelberg (2007)
9. Klein, S.T., Bookstein, A., Deerwester, S.: Storing text retrieval systems on cdrom: Compression and encryption considerations. ACM Trans. on Information Systems 7, 230–245 (1989)
10. Lecroq, T.: Fast exact string matching algorithms. Inf. Process. Lett. 102(6), 229–235 (2007)
11. Navarro, G., Raffinot, M.: A bit-parallel approach to suffix automata: Fast extended string matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 14–33. Springer, Heidelberg (1998)
12. Sunday, D.M.: A very fast substring search algorithm. Commun. ACM 33(8), 132–142 (1990)

Fast Searching in Packed Strings

Philip Bille*

Technical University of Denmark
phbi@imm.dtu.dk

Abstract. Given strings P and Q the (exact) string matching problem is to find all positions of substrings in Q matching P. The classical Knuth-Morris-Pratt algorithm [SIAM J. Comput., 1977] solves the string matching problem in linear time which is optimal if we can only read one character at the time. However, most strings are stored in a computer in a packed representation with several characters in a single word, giving us the opportunity to read multiple characters simultaneously. In this paper we study the worst-case complexity of string matching on strings given in packed representation. Let $m \leq n$ be the lengths P and Q, respectively, and let σ denote the size of the alphabet. On a standard unit-cost word-RAM with logarithmic word size we present an algorithm using time

$$O\left(\frac{n}{\log_\sigma n} + m + \mathrm{occ}\right).$$

Here occ is the number of occurrences of P in Q. For $m = o(n)$ this improves the $O(n)$ bound of the Knuth-Morris-Pratt algorithm. Furthermore, if $m = O(n/\log_\sigma n)$ our algorithm is optimal since any algorithm must spend at least $\Omega(\frac{(n+m)\log\sigma}{\log n} + \mathrm{occ}) = \Omega(\frac{n}{\log_\sigma n} + \mathrm{occ})$ time to read the input and report all occurrences. The result is obtained by a novel automaton construction based on the Knuth-Morris-Pratt algorithm combined with a new compact representation of subautomata allowing an optimal tabulation-based simulation.

1 Introduction

Given strings P and Q of length m and n, respectively, the *(exact) string matching problem* is to report all positions of substrings in Q matching P. The string matching problem is perhaps the most basic problem in combinatorial pattern matching and also one of the most well-studied, see e.g. [5, 7, 12, 14] for classical textbook algorithms and the surveys in [11, 17]. The first worst-case $O(n)$ algorithm (we assume w.l.o.g. that $m \leq n$) is the classical Knuth-Morris-Pratt algorithm [14]. If we assume that we can read only one character at the time this bound is optimal since we need $\Omega(n)$ time to read the input. However, most strings are stored in a computer in a *packed representation* with several characters in a single word. For instance, DNA-sequences have an alphabet of size 4 and are therefore typically stored using 2 bit per character with 32 characters in a

* Supported by the Danish Agency for Science, Technology, and Innovation.

64-bit word. On packed strings we can read multiple characters in constant time and hence potentially do better that the $\Omega(n)$ lower bound for string matching. In this paper we study the worst-case complexity of packed string matching and present an algorithm to beat the $\Omega(n)$ lower bound for almost all combinations of m and n.

1.1 Setup and Results

We assume a standard unit-cost word RAM with word length $w = \Theta(\log n)$ and a standard instruction set including arithmetic operations, bitwise boolean operations, and shifts. The space complexity is the number of words used by the algorithm, not counting the input which is assumed to be read-only. All strings in this paper are over an alphabet Σ of size σ. The *packed representation* of a string S is obtained by storing $\Theta(\log n/\log \sigma)$ characters per word thus representing S in $O(|S|\log \sigma/\log n) = O(|S|/\log_\sigma n)$ words. If S is given in the packed representation we simply say that S is a *packed string*. The *packed string matching problem* is defined as above except that P and Q are packed strings. In the worst-case any algorithm for packed string matching must examine all of the words in the packed representation of the input strings. The algorithm must also report all occurrences of P in Q and therefore must spend at least $\Omega\left(\frac{n}{\log_\sigma n} + \mathrm{occ}\right)$ time, where occ denotes the number of occurrences of P in Q. In this paper we present an algorithm with the following complexity.

Theorem 1. *For packed strings P and Q of length m and n, respectively, with characters from an alphabet of size σ, we can solve the packed string matching problem in time $O\left(\frac{n}{\log_\sigma n} + m + \mathrm{occ}\right)$ and space $O(n^\varepsilon + m)$ for any constant ε, $0 < \varepsilon < 1$.*

For $m = o(n)$ this improves the $O(n)$ bound of the Knuth-Morris-Pratt algorithm. Furthermore, if $m = O(n/\log_\sigma n)$ our algorithm matches the lower bound and is therefore optimal. In practical situations m is typically much smaller than n and therefore this condition is almost always satisfied.

1.2 Techniques

The KMP-algorithm [14] may be viewed as simulating an automaton K according to the characters from Q in a left-to-right order. At each character in Q we use K to maintain the longest prefix of P matching the current suffix of Q. Improvements of automaton-based algorithms can often be obtained by partitioning the automaton into many small subautomata, tabulate relevant information for the subautomata, and use the tables to speed-up the simulation in each subautomaton [15, 16, 21]. This idea is also known as the "Four Russian Technique" after Arlazarov et al. [4].

However, if we attempt to apply this idea to the KMP-algorithm two major problems appear. First, the structure of the transitions in K does not in general allow us to partition K into subautomata such that a simulation does not change subautomata too often. Indeed, for any partition we might be forced to repeatedly change subautomaton after every group of $O(1)$ characters of Q and hence end up using $\Omega(n)$ time. Secondly, even if we could design a suitable partition of K into subautomata we have to compactly encode the transitions of the subautomata in order for the tabulation to be efficient. An explicit list of such transitions will not suffice to achieve the bound of Theorem 1. The main contribution of this paper are two new ideas to overcome these problems.

First, we present the *segment automaton*, C, derived from K. In C, the states of K are grouped into overlapping intervals of $r = \Theta(\log n / \log \sigma)$ states from K such that (almost all of) the states in K are duplicated in C. We show how to selectively "copy" the transitions from K to C such that the total number of transitions between subautomata never exceeds $O(n/r)$ in the simulation on Q. Secondly, we show how to exploit structural properties of the transitions to represent subautomata optimally. This allows us to tabulate paths of transitions for all subautomata of size $< r$ using $O(\sigma^r + m) = O(n^\varepsilon + m)$ space and preprocessing time for a suitably chosen r. The simulation can then be performed in time $O(n/r + \text{occ}) = O(n/\log_\sigma n + \text{occ})$ leading to Theorem 1.

This main contribution of this paper is theoretical, however, we believe that both the segment automaton and the compact representation of automata may prove very useful in practice if combined with ideas from other algorithms for packed matching.

1.3 Related Work

Exploiting packed string representations to speed-up string matching is not a new idea and is even mentioned in the early papers by Knuth et al. and Boyer and Moore [7, 14]. More recently, several packed string matching algorithms have appeared [6, 8, 9, 10, 13, 19]. However, none of these improve the worst-case $O(n)$ bound of the classical KMP-algorithm.

It is possible to extend the "super-alphabet" technique by Fredriksson [9, 10] to obtain a simple trade-off for packed string matching. The idea is to build an automaton that, similar to the KMP-automaton, maintains the longest prefix of P matching the current suffix of Q but allows Q to be processed in groups of r characters. Each state has σ^r outgoing transitions corresponding to all combinations of r characters. This algorithm uses $O(n/r + m\sigma^r)$ time and $O(m\sigma^r)$ space. Choosing $r = \varepsilon \log_\sigma n$ this is $O(n/\log_\sigma n + mn^\epsilon)$ time and $O(mn^\epsilon)$ space. Compared to Theorem 1 this is a factor $\Theta(m)$ worse in space and only improves the $O(n)$ time bound of the KMP-algorithm when $m = o(n^{1-\epsilon})$.

Packed string matching is closely related to the area of *compressed pattern matching* introduced by Amir and Benson [1, 2]. Here the goal is to search for a uncompressed pattern in a compressed text without decompressing it first. Furthermore, the search should be faster than the naive approach of decompressing the text first and then using the fastest algorithm for the uncompressed problem. In *fully compressed pattern matching* the pattern is also given in compressed form. Several algorithms for (fully) compressed string matching are known, see e.g., the survey by Rytter [18]. For instance, if Q is compressed with the

Ziv-Lempel-Welch scheme [20] into a string Z of length z, Amir et al. [3] showed how to find all occurrences of P in time $O(m^2 + z)$. The packed representation of a string may be viewed as the most basic way to compress a string. Hence, in this perspective we are studying the fully compressed string matching problem for packed strings. Note that our result is optimal if the pattern is not packed.

1.4 Outline

In Section 2 we first review the KMP-algorithm before presenting the segment automaton in Section 3. In Section 4 we show how to compactly represent and efficiently tabulate subautomata and finally, in Section 5 we present the complete algorithm.

2 The Knuth-Morris-Pratt Automaton and String Matching

In this section we briefly review KMP-algorithm [14], which will be the starting point of our new algorithm.

Let S be a string of length $|S|$ on an alphabet Σ. The character at position i in S is denoted $S[i]$ and the substring from position i to j is denoted by $S[i, j]$. The substrings $S[1, j]$ and $S[i, |S|]$ are the *prefixes* and *suffixes* of S, respectively.

The Knuth-Morris-Pratt automaton (KMP-automaton), denoted $K(P)$, for P consists of $m+1$ states identified by the integer $\{0, \dots, m\}$ each corresponding to a prefix of P. From state s to state $s+1$, $0 \leq s < m$ there is a *forward transition* labeled $P[s]$. We call the rightmost forward transition from $m - 1$ to m the *accepting transition*. From state s, $0 < s \leq m$, there is a *failure transition* to a state denoted $\text{fail}(s)$ such that $P[1, \text{fail}(s)]$ is the longest prefix of P matching a proper suffix of $P[1, s]$. Fig. 1(a) depicts the KMP-automaton for the pattern $P = \text{ababca}$.

The failure transitions form a tree with root in state 0 and with the property that $\text{fail}(s) < s$ for any state s. Since the longest prefixes of $P[1, s]$ and $P[1, s+1]$ matching a suffix of P can increase by at most one character we have the following property of failure transitions.

Lemma 1. *Let P be a string of length m and $K(P)$ be the KMP-automaton for P. For any state* $1 < s < m$, $\text{fail}(s + 1) \leq \text{fail}(s) + 1$.

We will exploit this property in Section 4.1 to compactly encode subautomata of the KMP-automaton. The KMP-automaton can be constructed in time $O(m)$ [14].

To find the occurrences of P in Q we read the characters of Q from left-to-right while traversing $K(P)$ to maintain the longest prefix of P matching a suffix of the current prefix of Q as follows. Initially, we set the state of $K(P)$ to 0. Suppose that we are in state s after reading the $k-1$ characters of Q, i.e., the longest prefix of P matching a suffix of $Q[1, k-1]$ is $P[1, s]$. We process the next character $\alpha = Q[k]$ as follows. If α matches the label of the forward transition

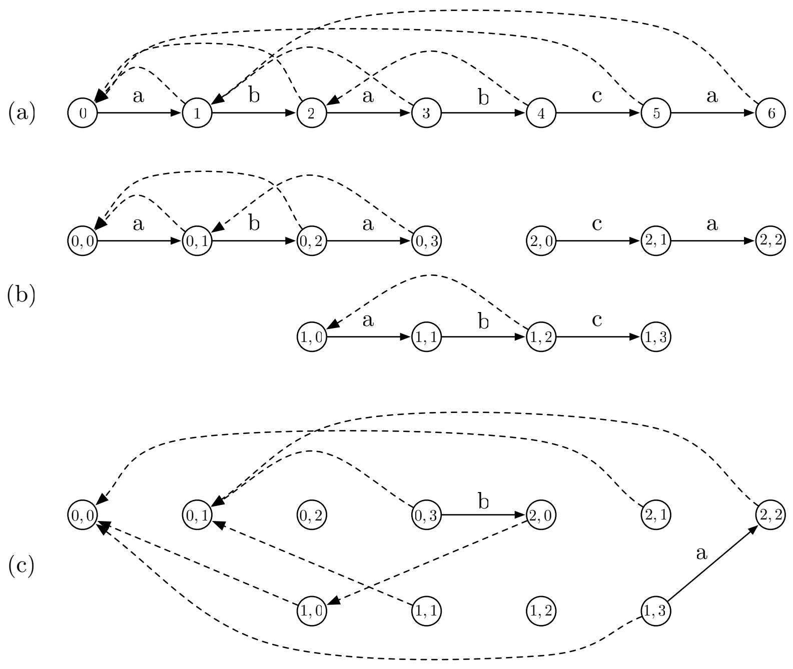


Fig. 1. (a) The Knuth-Morris-Pratt automaton $K(P)$ for the pattern P = ababca. Solid lines are forward transitions and dashed lines are failure transitions. (b)-(c) The corresponding segment automaton $C(P, 4)$ for P consisting of 3 segments with 4, 4, and 3 states. The light transitions are shown in (b) and the heavy transition transitions in (c).

from s the next state is $s + 1$. Furthermore, if this transition is the accepting transition then $k + 1$ is the endpoint of a substring of Q matching P and we therefore report an occurrence. Otherwise, (α does not match the label of the forward transition from s to $s + 1$) we recursively follow failure transitions from s until we find a state s' whose forward transition is labeled α in which case the next state is $s' + 1$, or if no such state exist we set the next state to be 0. We define the *simulation* of $K(P)$ on Q to be sequence of transitions traversed by the algorithm.

Each time the simulation on Q follows a forward transition we continue to the next character and hence the total number of forward transitions is at most n. Each failure transition strictly decrease the current state number while forward transitions increase the state number by 1. Since we start in state 0 the number of failure transition is therefore at most the number of forward transitions. Hence, the total number of transitions is at most $2n$ and therefore the searching takes $O(n)$ time. In total the KMP-algorithm uses time $O(n + m) = O(n)$.

3 The Segment Automaton

In this section we introduce a simple automaton called the *segment automaton*. The segment automaton for P is equivalent to $K(P)$ in the sense that the simulation on Q at each step provides the longest prefix P matching a suffix of the current prefix of Q. The segment automaton allows to easily decompose $K(P)$ into subautomata of a given size r such that the simulation on Q passes through at most $O(n/r)$ subautomata.

Let $K = K(P)$ be the KMP-automaton for P. For a even integer parameter r, $1 < r \leq m + 1$ we define the segment automaton with parameter r, denoted $C(P, r)$, as follows. Define a *segment* S to be an interval $S = [l, r]$, $0 \leq l \leq r \leq m$, of states in $K(P)$ and let $|S| = r - l + 1$ denote the size of S. Divide the $m + 1$ states of K into a set of $z = \lceil (m + 1)/r \rceil$ overlapping segments, denoted $SS = \{S_0, \ldots, S_{z-1}\}$, where $S_i = [l_i, r_i]$ is defined by

$$l_i = i \cdot \frac{r}{2} \qquad r_i = \min(l_i + r - 1, m).$$

Thus, each segment in SS consists of r consecutive states from K, except the last segment, S_{z-1}, which may be smaller. Any state s in K appears in at most 2 segments and adjacent segments share $r/2$ states.

The segment automaton $C = C(P, r)$ is obtained by adding $|S|$ states for each segment $S \in SS$ and then selectively "copying" transitions from K to C. Specifically, the states of C is the set of pairs given by

$$\{(i, j) \mid 0 \leq i < z, 0 \leq j < |S_i|\}.$$

We view each state (i, j) C as the jth state of the ith segment, i.e., state (i, j) corresponds to the state $l_i + j$ in K. Hence, each state in K is represented by 1 or 2 states in C and each state in C uniquely corresponds to a state in K.

We copy transitions from K to C in the following way. Let $t = (s, s')$ be a transition in K. For each segment S_i such that $s \in [l_i, r_i]$ we have the following transitions in C:

– If $s' \in [l_i, r_i]$ there is a *light transition* from $(i, s - l_i)$ to $(i, s' - l_i)$.
– If $s' \notin [l_i, r_i]$ there is a *heavy transition* from $(i, s - l_i)$ to $(i', s' - l_{i'})$, where either $S_{i'}$ is the unique segment containing s' or if two segments contain s, then $S_{i'}$ is the segment such that $s' \in [l_{i'}, l_{i'} + r/2]$, i.e., the segment containing s' in the leftmost half.

If t is a forward transition with label $\alpha \in \Sigma$ it is also a forward transition in C with label α, if t is a failure transition it is also a failure transition in C, and if t the accepting transition it is also an accepting transition in C. The segment automaton with $r = 4$ corresponding to the KMP-automaton of Fig. 1(a) is shown in Fig. 1(b) and (c) showing the light and heavy transitions, respectively. From the correspondence between C and K we have that each accepting transition in a simulation of C on Q corresponds to an occurrence of P in Q. Hence, we can solve string matching by simulating C instead of K.

We will use the following key property of the C.

Lemma 2. *For a string P of length m and even integer parameter $1 < r \leq m + 1$, the simulation of the segment automaton, $C(P, r)$, on a string Q of length n contains at most $O(n/r + \mathrm{occ})$ heavy and accepting transitions.*

Proof. Consider the sequence T of transitions in the simulation of $C = C(P, r)$ on Q. Let N_{accept} denote the number of accepting transitions, and let N_{hforward} and N_{hfail} denote the number of heavy forward and heavy failure transitions, respectively. Each accepting transition in T corresponds to an occurrence and therefore $N_{\mathrm{accept}} = \mathrm{occ}$. For a state (i, j) in C we will refer to i as the *segment number*. Since a forward transition in K increases the state number by 1 in K a heavy forward transition increases the segment number by 1 or 2 in C. A heavy failure transition strictly decrease the segment number. Hence, since we start the simulation in segment 0, we can have at most 2 heavy failure transitions for each heavy forward transition in T and therefore

$$N_{\mathrm{hfail}} \leq 2N_{\mathrm{hforward}}. \tag{1}$$

If $N_{\mathrm{hforward}} = 0$ the results trivially follows. Hence, suppose that $N_{\mathrm{hforward}} > 0$. Before the first heavy forward transition in T there must be at least $r - 1$ light transitions in order to reach state $(0, r - 1)$. Consider the subsequence of transitions t in T between an arbitrary heavy transition h and a forward heavy transition f. The heavy transition h cannot end in segment $z - 1$ since there is no heavy forward transition from here. All other heavy transitions have an endpoint in the leftmost half of a segment and therefore at least $r/2$ light transitions are needed before a heavy forward transition can occur. Recall that the total number of transition in T is at most $2n$ and therefore the number of heavy forward transitions in T is bounded by

$$N_{\mathrm{hforward}} \leq 2n/(r/2) = 4n/r. \tag{2}$$

Combining the bound on N_{accept} with (1) and (2) we have that the total number of heavy and accepting transitions is

$$N_{\mathrm{hforward}} + N_{\mathrm{hfail}} + N_{\mathrm{accept}} \leq 3N_{\mathrm{hforward}} + \mathrm{occ} = O(n/r + \mathrm{occ}). \qquad \square$$

4 Representing Segments

4.1 A Compact Encoding

Let S be a segment with r states over an alphabet of size σ. We show how to compactly represent all light transitions in S using $O(r \log \sigma)$ bits. To represent forward transitions we simply store the labels of the $r-1$ light forward transitions in S using $(r - 1) \log \sigma = O(r \log \sigma)$ bits. Next consider the failure transitions. A straightforward approach is to explicitly store for each state $s \in S$ a bit indicating if its failure pointer is light or heavy and, if it is light, a pointer to $\mathrm{fail}(s)$. Each pointer requires $\lceil \log r \rceil$ bits and hence the total cost for this representation is $O(r \log r)$ bits. We show how to improve this to $O(r)$ bits in the following.

First, we locally enumerate the states in S to $[0, r-1]$. Let $I = \{i_1, \ldots, i_\ell\}$, $0 \leq i_1 < \cdots < i_\ell < r$, be the set of states in S with a light failure transition and let $F = \{f_{i_1}, \ldots, f_{i_\ell}\}$ be the set of failure pointers for the states in I. We encode I as a bit string B_I of length r such that $B_I[j] = 1$ iff $j \in I$. This uses r bits. To represent F compactly we encode f_1 and the sequence of differences between consecutive elements $D = d_{i_2}, \ldots, d_{i_\ell}$, where $d_{i_j} = f_{i_j} - f_{i_{j-1}}$. We represent f_1 explicitly using $\lceil \log r \rceil$ bits. Our representation of D consists of 2 bit strings. The first string, denoted B_D, is the concatenation of the binary encoding of the numbers in D, i.e., $B_D = \mathrm{bin}(d_{i_2}) \cdots \mathrm{bin}(d_{i_\ell})$, where $\mathrm{bin}(\cdot)$ denotes standard two's complement binary encoding (the differences may be negative) and $\cdot$ denotes concatenation. Each number d_j uses at most $1 + \log |d_j|$ bits and therefore the size of the B_D is at most

$$|B_D| \leq \sum_{j \in I'} (|\log(d_j)| + 1) < r + \sum_{j \in I'} |\log(d_j)|, \tag{3}$$

where $I' = I \backslash \{i_1\}$. The second bit string, denoted $B_{D'}$, represents the boundaries of the numbers in B_D, i.e., $B_{D'}[k] = 1$ iff k is the start of a new number in B_D. Thus, $|B_{D'}| = |B_D|$. Note that with f_1, B_D, and $B_{D'}$ we can uniquely decode F. The total size of the representation is $\lceil \log r \rceil + 2|B_D|$ bits.

To bound the size of the representation we show that $|B_D| = O(r)$ implying that the representation uses $\lceil \log r \rceil + 2 \cdot O(r) = O(r)$ bits as desired. We first bound the sum $\sum_{j \in I'} |d_j|$. Recall from Lemma 1 that the failure function increases by at most 1 between consecutive states in K. Hence, over the subsequence F of $< r$ of failure pointers in the range $[0, r-1]$ the total increase of the failure function can be at most r. Hence, $\sum_{j \in I'} d_j \leq r$. Furthermore, if $f_1 = x$, for some $x \in [0, r-1]$, the total decrease of F over a segment of r states is at most x plus the total increase and therefore $\sum_{j \in I'} d_j \geq -(x + r) \geq -2r$. Hence,

$$\sum_{j \in I'} |d_j| \leq 2r \tag{4}$$

Combining (3) and (4) we have that

$$\begin{aligned} |B_D| &< r + \sum_{j \in I'} \log |d_j| = r + \log \left(\prod_{j \in I'} |d_j| \right) \leq r + \log \left((2r/|I'|)^{|I'|} \right) \\ &< r + \log \left((2r/r)^r \right) = O(r). \end{aligned}$$

Thus, we have shown the following result.

Lemma 3. *All light forward and failure transitions of a segment of size r can be encoded using $O(r \log \sigma)$ bits.*

4.2 Simulating Light Transitions

Let $C = C(P, r)$ be the segment automaton, and consider the path p of states in the simulation on C from a state (i, j) on some string q. Then, the *longest light* *path* from (i, j) on q is defined as the longest prefix of p consisting entirely of light non-accepting transitions in segment i. For example, consider state $(1, 1)$ in segment 1 in Fig. 1. The longest light path on the string $q = \mathrm{bac}$ is the path $p = (1, 1), (1, 2), (1, 0), (1, 1)$. The transition on c from state $(1, 1)$ is heavy and therefore not included in p.

We show how to quickly compute the length and endpoint of longest light paths. Let S^{enc} be the compact encoding of a segment S as described above including the label of the forward heavy transition from the rightmost state in S (if any) and a bit indicating whether or not the rightmost light transition is accepting or not. Furthermore, let j be a state in S, let q be a string, and define

Next(S^{enc}, j, q): Return the pair (l, j'), where l and j' is the length and final state, respectively, of the longest light path in S from j matching a prefix of q.

We can efficiently tabulate Next for arbitrary strings q of length r as follows. Let b be the total number of bits needed to represent the input to Next. The string q uses $r \lceil \log \sigma \rceil$ bits and by Lemma 3 S^{enc} uses $O(r \log \sigma + \log \sigma + 1) = O(r \log \sigma)$ bits. Furthermore, the state number j uses $\lceil \log r \rceil$ bits and hence $b = O(r \log \sigma + \log r) = O(r \log \sigma)$. Using a table T with 2^b entries we can store all results of Next. Each entry is computed using a standard simulation in $O(r)$ time and therefore we can construct T in $2^b \cdot O(r) = 2^{O(b)}$ time and space. Hence, if we have $t < 2^w$ space available for T we may set $r = \frac{1}{c} \cdot \frac{\log t}{\log \sigma}$, where $c > 0$ is an upper bound on the constant appearing in the $2^{O(b)}$ expression above. Hence, the total space and preprocessing time now becomes $2^{O(b)} = 2^{\frac{1}{c} \frac{c \log t}{\log \sigma} \log \sigma} = O(t)$.

With T precomputed and stored in memory we can now answer arbitrary Next queries for arbitrary encoded segments and strings of length at most q in constant time.

5 The Algorithm

We now put the pieces from the previous sections together to obtain our main result of Theorem 1. Assume that we have $t < 2^w$ space available and choose $r = \Theta(\log t / \log \sigma)$ as above for the tabulation. We first preprocess P by computing the following information:

– The segment automaton $C(P, r)$ with parameter r and $z = \lceil m + 1/r \rceil$ segments $SS = \{S_0, \ldots, S_{z-1}\}$.
– The compact encoding S^{enc} for each segment $S \in SS$.
– The tabulated Next function for segments with r states and input string of length r.

We compute the segment automaton and the compact encodings in $O(m)$ time and space. The tabulation for Next uses $O(t)$ time and space and hence the preprocessing uses $O(t + m)$ time and space.

We find the occurrences of P in Q using the algorithm described below. The main idea is to simulate the segment automaton using the tabulated Next function with the segment automaton. At each iteration of the algorithm we traverse light transitions until we either have processed r characters from Q or encounter a heavy or accepting transition. We then follow the next transition reporting an occurrence if the transition is accepting and repeat until we have read all of Q.

Algorithm S (*Packed String Search*). Let P be a string of preprocessed for parameter r as above. Given a string Q of length n this algorithm finds all occurrences of P in Q.

S1. [Initialize] Set $(i, j) \leftarrow (0, 0)$ and $k \leftarrow 1$.

S2. [Do light transitions] Compute $(l, j') \leftarrow \textsc{Next}(S_i^{\text{enc}}, j, Q[k, \min(k + r, n)])$. At this point (i, j') is the state in the traversal of C on Q after reading the prefix $Q[1, k + l]$. All transitions on the string $Q[k, k + l]$ are light and non-accepting by the definition of $\textsc{Next}$.

S3. [Done?] If $k = n$ the algorithm terminates.

S4. [Do next transition] Compute (i^*, j^*) by following the transition from (i, j') on character $Q[k + l + 1]$. If the transition is a failure transition we set $k^* \leftarrow k + l$ and otherwise set $k^* \leftarrow k + l + 1$. If this is the accepting transition report an occurrence ending at position k^*.

S5. [Repeat] Update $(i, j) \leftarrow (i^*, j^*)$ and $k \leftarrow k^*$ and repeat from step S2.

It is straightforward to verify that Algorithm S simulates $C(P, r)$ on Q and reports occurrences whenever we encounter an accepting transition. In each iteration we either read r character from Q and/or perform a heavy or accepting transition. We can process r characters from Q on light transitions at most $\lceil n/r \rceil$ and by Lemma 2 the total number of heavy and accepting transitions is $O(n/r + \text{occ})$. Hence, the total number of iterations is $O(n/r + \text{occ})$. Since each iteration takes constant time this also bounds the running time. Adding the preprocessing time and plugging in $r = \Theta(\log t / \log \sigma)$ the time becomes

$$O\left(\frac{n}{r} + t + m + \text{occ}\right) = O\left(\frac{n}{\log_\sigma t} + t + m + \text{occ}\right)$$

with space $O(t + m)$. Hence we have the following result.

Theorem 2. *Let P and Q be packed strings of length m and n, respectively. For a parameter $t < 2^w$ we can solve the packed string matching problem in time $O\left(\frac{n}{\log_\sigma t} + t + m + \text{occ}\right)$ and space $O(t + m)$.*

Note that the tabulation is independent of P and we therefore only need to compute it once for multiple searches. If we plugin $t = n^\varepsilon$, for $0 < \varepsilon < 1$, we obtain an algorithm using time $O\left(\frac{n}{\log_\sigma(n^\varepsilon)} + n^\varepsilon + m + \text{occ}\right) = O\left(\frac{n}{\log_\sigma n} + m + \text{occ}\right)$ and space $O(n^\varepsilon + m)$ thereby showing Theorem 1.

References

1. Amir, A., Benson, G.: Efficient two-dimensional compressed matching. In: Proceedings of the 2nd Data Compression Conference, pp. 279–288 (1992)
2. Amir, A., Benson, G.: Two-dimensional periodicity and its applications. In: Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 440–452 (1992)
3. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: pattern matching in Z-compressed files. J. Comput. System Sci. 52(2), 299–307 (1996)
4. Arlazarov, V.L., Dinic, E.A., Kronrod, M.A., Faradzev, I.A.: On economic construction of the transitive closure of a directed graph (in russian). english translation in soviet math. dokl. 11, 1209–1210 (1975); Dokl. Acad. Nauk. 194, 487–488 (1970)
5. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Commun. ACM 35(10), 74–82 (1992)
6. Baeza-Yates, R.A.: Improved string searching. Softw. Pract. Exper. 19(3), 257–271 (1989)
7. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
8. Faro, S., Lecroq, T.: Efficient pattern matching on binary strings. In: Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (2009)
9. Fredriksson, K.: Faster string matching with super-alphabets. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 44–57. Springer, Heidelberg (2002)
10. Fredriksson, K.: Shift-or string matching with super-alphabets. Inf. Process. Lett. 87(4), 201–204 (2003)
11. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge (1997)
12. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31(2), 249–260 (1987)
13. Klein, S.T., Ben-Nissan, M.: Accelerating Boyer Moore searches on binary texts. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 130–143. Springer, Heidelberg (2007)
14. Knuth, D.E., James, J., Morris, H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
15. Masek, W., Paterson, M.: A faster algorithm for computing string edit distances. J. Comput. System Sci. 20, 18–31 (1980)
16. Myers, E.W.: A four-russian algorithm for regular expression pattern matching. J. ACM 39(2), 430–448 (1992)
17. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences, 280 pages. Cambridge University Press, Cambridge (2002)
18. Rytter, W.: Algorithms on compressed strings and arrays. In: Bartosek, M., Tel, G., Pavelka, J. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 48–65. Springer, Heidelberg (1999)
19. Tarhio, J., Peltola, H.: String matching in the DNA alphabet. Softw. Pract. Exp. 27, 851–861 (1997)
20. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (1984)
21. Wu, S., Manber, U., Myers, E.W.: A subquadratic algorithm for approximate regular expression matching. J. Algorithms 19(3), 346–360 (1995)

New Complexity Bounds for Image Matching under Rotation and Scaling*

Christian Hundt[1] and Maciej Liśkiewicz[2,**]

[1] Institut für Informatik, Universität Rostock, Germany
christian.hundt@uni-rostock.de
[2] Institut für Theoretische Informatik, Universität zu Lübeck, Germany
liskiewi@tcs.uni-luebeck.de

Abstract. The problem of image matching under rotation is to find for two given digital images A and B a rotation that converts image A as close as possible to B. The research in combinatorial pattern matching led to a series of improving algorithms which commonly attack this problem by searching the complete set $\mathcal{R}(A)$ of all rotations of A. We present the first optimal algorithm of this kind, i.e, one that solves image matching in $O(|\mathcal{R}(A)|) = O(n^3)$ time for images of size $n \times n$. Subsequently, for image matching under compositions of rotation and scaling we show a new lower bound $\Omega(n^6/\log n)$ on the cardinality of $\mathcal{SR}(A)$, the set of rotated and scaled transformations of A. This bound almost matches the upper bound $O(n^6)$.

1 Introduction

In general, the image matching problem (IMP, for short) is to find for two given digital images A and B an admissible transformation f that changes A closest to B. If $\mathcal{F}$ is a fixed set of admissible transformations $f : \mathbb{R}^2 \to \mathbb{R}^2$ then let $\mathcal{D}_{\mathcal{F}}(A)$ denote the set of all images $f(A)$ which result from transformations of image A according to $f \in \mathcal{F}$. So, the image matching problem is that of finding in the data base $\mathcal{D}_{\mathcal{F}}(A)$ the image most close to B. In this paper we investigate IMP for two basic subsets of linear transformations, namely rotations $\mathcal{F}_{\mathtt{r}}$ and compositions of rotation and scaling $\mathcal{F}_{\mathtt{sr}}$. For simplification of notation we write $\mathcal{R}(A)$ instead of $\mathcal{D}_{\mathcal{F}_{\mathtt{r}}}(A)$ and $\mathcal{SR}(A)$ instead of $\mathcal{D}_{\mathcal{F}_{\mathtt{sr}}}(A)$. Image matching applied to these transformations has a wide range of applications in various image processing settings e.g. in computer vision ([19]), medical imaging (see, for example [7,23,24]), pattern recognition, digital watermarking ([8]), etc.

The image matching problem was intensively studied both experimentally and theoretically. In the image processing community a common approach to solve the problem uses techniques based on continuous analysis. Though the used techniques guarantee achieving satisfactory local optima, the disadvantage of such methods is the high complexity to find the global optimum (for more

* Partially supported by DFG research grant RE 672/5-1.
** On leave from Instytut Informatyki, Uniwersytet Wrocławski, Poland.

discussion, see e.g. [14]). Another approach is to use discrete methods and models. Prominent examples of this are feature based techniques (see e.g. [1,21]) which consist in extracting salient features (points, lines, regions etc. in the real plane) from images A and B and subsequently, in finding a transformation f which transforms the geometrical objects of A closest to those of B. But, this technique relies heavily on the quality of feature extraction and feature matching, two highly non-trivial tasks. Feature matching, e.g., remains difficult even for points (see [17] for a survey and [20,27] for some related problems). In fact, known algorithms for this problem give only approximate solutions and particularly they do not guarantee to find the global optimum, even for such a simple class of transformations as compositions of rotation and translation [18].

Recently, discretization techniques developed in the combinatorial pattern matching research (CPM, for short) have been used successfully for IMP. Apart from algorithmic achievements, this led to improved techniques for the analysis of the problem. Essentially, all algorithms developed in CPM for computing a best match $f(A)$ with B share the same plane idea to perform exhaustive search of the entire set $\mathcal{D}_{\mathcal{F}}(A)$. Surprisingly, the fastest known methods which determine the provably best image match under rotations and under combinations of rotation and scaling come from this simple approach. In fact, the main challenge here is to find a discretization of $\mathcal{F}$ to get $\mathcal{D}_{\mathcal{F}}(A)$. Although CPM research (see e.g. [22,13,12,2,5,4,3]) concentrates mainly on the problem of locating an *exact* match $f(A)$ in B rather than on computing the *best* one like in IMP, the developed discretization techniques form a convenient start point for our research on IMP. In particular, Amir et al. [2] have proven that the cardinality of $\mathcal{R}(A)$, the set of different possible rotated images, is $\Theta(n^3)$ and subsequently they have provided a pattern matching algorithm working in time $O(n^2m^3)$ for images A and B of size $n \times n$ and $m \times m$, respectively. Using this approach one can get that the image matching problem for rotations is solvable in time $O(n^5)$. Roughly speaking, this corresponds to the complexity of a method which searches all $\Theta(n^3)$ rotated images $f(A)$ in $\mathcal{R}(A)$ and evaluates in $O(n^2)$ time the distortions between $f(A)$ and B to find the closest image.

However, one can improve this naive method of searching all rotated images in $\mathcal{R}(A)$ in an appropriate way. Based on the techniques by Amir et al. Nouvel and Rémila [26] describe a method which computes *incrementally* all the rotated images of A in time $O(n^3 \log n)$. Thus, applying the incremental rotation algorithm we get that image matching allowing rotations can be done in time $O(n^3 \log n)$. Using a new discretization method for $\mathcal{R}(A)$ we provide in [16] an algorithm for IMP of the same time complexity. In both algorithms the sorting of values corresponding to rotation angles plays a crucial role and causes the $\log n$ factor in the computational complexity. One of the main contributions of this paper is an improvement of the algorithm presented in [16] which allows getting rid of the $\log n$ factor. In this way we get the first algorithm for the image matching problem under rotations that runs in $O(n^3)$ time, which is optimal in the sense that all $\Theta(n^3)$ elements of $\mathcal{R}(A)$ are computed in $O(n^3)$ steps.

In [15,16] we have investigated image matching also for the class $\mathcal{F}_{\mathtt{sr}}$ of compositions of rotation and scaling. For these transformations we have managed to provide an image matching algorithm which works in time linear in the cardinality of $\mathcal{SR}(A)$, the set of all possible rotated and scaled images. But, it has been left as an open question what is the exact estimation of this number. In [15,16] we show that the cardinality is $|\mathcal{SR}(A)| = O(n^6) \cap \Omega(n^5)$.

Achieving a more exact estimation of the cardinality of $\mathcal{SR}(A)$ seems to be an interesting new task in combinatorial geometry. Due to Amir et al. [2] we know that $|\mathcal{R}(A)| = \Theta(n^3)$. On the other hand $|\mathcal{S}(A)|$ – the number of all possible scaled images – is $\Theta(n^2)$ (see e.g. [3] or [16]). Thus, a natural conjecture would be that the corresponding cardinality of $\mathcal{SR}(A)$ is $\Theta(n^5)$. But using a straightforward approach to get all images in $\mathcal{SR}(A)$ combining either $\mathcal{S}(\mathcal{R}(A))$ or $\mathcal{R}(\mathcal{S}(A))$ does not work. Neither the set of all images obtained by scaling all rotated images nor the set obtained by rotating all scaled images does coincide with the set $\mathcal{SR}(A)$. In contrast to the continuous compositions of scaling and rotation, their compositions on digital images are neither commutative nor transitive. In this paper we continue the research of IMP in the combinatorial setting using the algebraic approach we introduced in [14] and refined in [15,16] and give a new, rather surprising, lower bound on the cardinality: $|\mathcal{SR}(A)| = \Omega(n^6/\log n)$.

This paper presents results which heavily build on previous papers [14,15,16]. We organize the presentation as follows: We start with technical preliminaries. Next, in Section 3 we briefly provide the basics of our approach introduced in [14,15,16] which are necessary to understand the new results of this paper. In Section 4 we provide an $O(n^3)$ time algorithm for image matching under rotations and next in Section 5 we prove the $\Omega(n^6/\log n)$ lower bound on the cardinality $|\mathcal{SR}(A)|$.

2 Technical Preliminaries

Through the whole paper, an image is a two-dimensional array of pixels, i.e., of unit squares covering a certain area of the real plane $\mathbb{R}^2$. The pixels of a size-n image A are indexed over the set $\mathcal{N} = \{(i,j) \mid -n \le i,j \le n\}$, which we call the support of A. The pixel with index (i,j) has its geometric center point at coordinates (i,j). Each pixel (i,j) has a color $A\langle i,j\rangle$ that is an element from a finite set $\Sigma = \{0,1,\ldots,\sigma\}$ of color values. To simplify the dealing with A's borders we let $A\langle i,j\rangle = \perp$ if $(i,j) \notin \mathcal{N}$, where $\perp$ is a special color marking the exterior of A. We measure the distortion between two given size-n images A and B by $\Delta(A,B) = \sum_{(i,j)\in\mathcal{N}} \delta(A\langle i,j\rangle, B\langle i,j\rangle)$ where $\delta(a,b)$ is a function charging mismatches, for example, $\delta(a,b) = |a-b|$ if $a \ne \perp$ and $b \ne \perp$ and 0 otherwise.

In the general case, image transformations are injective functions $f : \mathbb{R}^2 \to \mathbb{R}^2$. Applying a transformation f to A we get a transformed image $f(A)$, a new two-dimensional array of pixels with support $\mathcal{N}$. To define a color value for any pixel (i,j) in $f(A)$, let f^{-1} be the inverse function of f. Then we define the color value $f(A)\langle i,j\rangle$ as the color $A\langle i',j'\rangle$ of the pixel $(i',j') = [f^{-1}(i,j)]$, where $[(x,y)] := ([x],[y])$ denotes rounding both components of a vector $(x,y) \in \mathbb{R}^2$.

Hence, to determine $f(A)\langle i,j\rangle$ we choose the pixel of A which geometrically contains the point $f^{-1}(i,j)$ in its square area. With this setting we model *nearest-neighbor* interpolation, commonly used in image processing.

Now, for any image A and set $\mathcal{F}$ we may define the set $\mathcal{D}_{\mathcal{F}}(A) = \{f(A) \mid f \in \mathcal{F}\}$ that contains all possible image transformations of A with respect to $\mathcal{F}$. Subsequently, we are ready to give the definition of the image matching problem under $\mathcal{F}$: *For given images A and B with support $\mathcal{N}$ find in the set $\mathcal{D}_{\mathcal{F}}(A)$ an image A' minimizing the distortion $\Delta(A', B)$.* For the analysis of the structural complexity aspects of this problem we apply the unit cost model for arithmetic operations, a complexity measure which is sometimes referred to as arithmetic complexity. Therefore, we assume that mathematical basic integer operations like addition, subtraction and multiplication can be done in constant time.

In this paper we are basically interested in two transformation classes, namely $\mathcal{F}_{\mathtt{r}}$ which contains all rotations and $\mathcal{F}_{\mathtt{sr}}$ the transformations combining scaling and rotation. Any transformation f in $\mathcal{F}_{\mathtt{sr}}$ can be uniquely described by

$$f(x,y) = \begin{pmatrix} s\cos\phi & s\sin\phi \\ -s\sin\phi & s\cos\phi \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{1}$$

for some $s, \phi \in \mathbb{R}$, with $s \neq 0$. Since $\mathcal{F}_{\mathtt{r}} \subset \mathcal{F}_{\mathtt{sr}}$ all rotations can be represented accordingly with the restriction to $s = 1$.

3 Previous Results

In our previous works [15,16] we presented a new algorithmic approach to solve image matching under $\mathcal{F}_{\mathtt{sr}}$ in time linear with respect to the cardinality $|\mathcal{SR}(A)|$. However, using our methods for rotations we get a time complexity asymptotically bigger than the cardinality of $\mathcal{R}(A)$ by a $\log n$ factor. In this section we will briefly discuss some basics of our approach which will be extended in this paper to give a more precise estimation on $|\mathcal{SR}(A)|$ and to get rid of the $\log n$ factor for solving the IMP with rotations.

By equation (1) all transformations in $\mathcal{F}_{\mathtt{sr}}$ can be characterized by the two parameters $p = s\cos\phi$ and $q = s\sin\phi$. Hence, each such transformation f can be described by a point $(p,q)^T$ in the two-dimensional parameter space $\mathbb{R}^2$. Note that $(0,0)^T$ is the only point in $\mathbb{R}^2$ which corresponds to a non-injective transformation and by this it does not characterize a transformation in $\mathcal{F}_{\mathtt{sr}}$. However, for convenience we will simply ignore this exception. The transformations in $\mathcal{F}_{\mathtt{r}}$ are represented by the points $(p,q)^T$ with $p^2 + q^2 = 1$ and lie on the unit circle denoted here by C, which is a non-linear one-dimensional subspace of $\mathbb{R}^2$.

The central concept of our approach is a new discretization technique for $\mathcal{F}_{\mathtt{sr}}$. We introduce a subdivision of the parameter space $\mathbb{R}^2$ into a finite number of subspaces $\varphi_1, \ldots, \varphi_t$ such that any pair of transformations $f_1, f_2 \in \mathcal{F}_{\mathtt{sr}}$ gives the same transformed image $f_1(A) = f_2(A)$ if their inverses f_1^{-1} and f_2^{-1} are represented by points $(p_1,q_1)^T$, resp. $(p_2,q_2)^T$ contained in the same subspace φ_i for some $i \in \{1, \ldots, t\}$. This means that each of the t subspaces represents one transformed image in $\mathcal{SR}(A)$. In this way we provide a discrete characterization of the uncountable set $\mathcal{F}_{\mathtt{sr}}$ which characterizes $\mathcal{SR}(A)$ exactly. In the case of rotations an analogous property holds for a subdivision of C into segments.

In the basic principle of searching the whole set $\mathcal{SR}(A)$ our approach is consistent with the common practice in the CPM. However, the used new discretization method enables a very fast enumeration of the set. We use our geometric characterization to traverse all the subspaces φ_1 to φ_t of the parameter space. With each subspace we find one of the possible transformed images A' in $\mathcal{SR}(A)$. Subsequently, the distortion between such A' and B is evaluated to eventually find the best match.

For images of size n the subdivision of the parameter space into the spaces φ_1 to φ_t is determined by a set $\mathcal{H}_n$ of straight lines. Each line in $\mathcal{H}_n$ is described by one or more equations from the set H_n which we define as follows:

$$H_n = \left\{ h_{ijk} : ip + jq = k - 0.5 \mid (i,j) \in \mathcal{N}, k \in \{-n, \ldots, n+1\} \right\}$$

To describe a line, let for an equation $h_{ijk} \in H_n$ and for any point $(p,q) \in \mathbb{R}^2$, $h_{ijk}(p,q) = ip + jq - (k - 0.5)$. Then we say that h_{ijk} describes the line ℓ which contains all the points $(p,q) \in \mathbb{R}^2$ satisfying the equation $h_{ijk}(p,q) = 0$, i.e., we get $\ell = \{(p,q) \in \mathbb{R}^2 \mid h_{ijk}(p,q) = 0\}$. Note the difference: h_{ijk} is an algebraic expression whereas ℓ is a subspace of $\mathbb{R}^2$.

The set of straight lines $\mathcal{H}_n$ plays a crucial role in our approach. The meaning of h_{ijk} and the described line ℓ can be understood as follows: Suppose we have a *continuous* arbitrarily formed path Π of points in $\mathbb{R}^2$ which starts at $P(0)$ and ends at $P(1)$. Let us describe the points forming the path Π as $P(t)$ for $t \in [0,1]$. Consider the transformation $f_{P(t)}$ which is obtained by taking the inverse of the transformation described by the point $P(t)$. Now imagine a walk from $P(0)$ to $P(1)$ by the continuous increment of t. Due to the discrete nature of images it turns out that $f_{P(t)}(A)$ undergoes abrupt changes while $f_{P(t)}$ changes continuously with t. The crucial property is that $f_{P(t)}(A)$ remains unchanged as long as Π does not cut a straight line from $\mathcal{H}_n$. Particularly, at the moment t when $P(t) \in \ell$, i.e, when Π crosses ℓ, then $f_{P(t)}(A)$ changes at the pixels (i,j) and $(j,-i)$. Thereby, the pixel of the input image A which is responsible for the new color value of $f_{P(t)}(A)\langle i,j\rangle$, resp. $f_{P(t)}(A)\langle j,-i\rangle$, can be derived from parameter k. Notice in this context that the constant 0.5 in the equations is due to the rounding used for the realization of nearest-neighbor interpolation.

Hence, the straight lines of $\mathcal{H}_n$ cut the parameter space into the pieces φ_1 to φ_t. To describe this partition we need the following additional subspaces of $\mathbb{R}^2$ for all $h_{ijk} \in H_n$:

$$\ell^+ = \{(p,q) \mid h_{ijk}(p,q) > 0\} \quad \text{and} \quad \ell^- = \{(p,q) \mid h_{ijk}(p,q) < 0\}.$$

In geometry the subspaces $\ell^+ \cup \ell$ and ℓ^- are called half-planes. Now we are ready to define the partition of $\mathbb{R}^2$ by the set of equations H_n:

$$\mathcal{A}(H_n) = \left\{ \varphi \subseteq \mathbb{R}^2 \mid \varphi = \bigcap_{\ell \in \mathcal{H}_n} \ell^{s_w} \text{ for some } s_1, \ldots, s_{|\mathcal{H}_n|} \in \{+,-,0\}, \varphi \neq \emptyset \right\},$$

where ℓ^0 denotes just ℓ. For an example see Figure 3. In literature the set $\mathcal{A}(H_n)$ is called the line arrangement given by the lines $\mathcal{H}_n$. For detailed information on line arrangements see [9]. Accordingly, $\mathcal{H}_n$ partitions the circle C into segments:

$$\mathcal{A}_C(H_n) = \{\varphi \mid \exists\varphi' \in \mathcal{A}(H_n), \varphi = \varphi' \cap C \neq \emptyset\}$$

We call the elements of $\mathcal{A}(H_n)$ and $\mathcal{A}_C(H_n)$ faces. A face is called a d-face if its dimension is d for $d \in \{0, 1, 2\}$. Thus, a 0-face is a point, a 1-face is a line (segment) or curve (segment), and a 2-face is a convex region on the plane given by the intersection of a finite number of half-planes. A face φ' is a subface of another face φ if the dimension of φ' is one less than of φ and φ' is contained in the boundary of φ. We also say that φ and φ' are incident and that φ is a superface of φ'. The relation between $\mathcal{A}(H_n)$ and $\mathcal{SR}(A)$ is our most important property which we formulated as a main result in [15,16]:

Theorem 1 ([15,16]). *For all n and every image A of size n there exist surjective mappings $\Gamma_{n,\mathtt{sr}} : \mathcal{A}(H_n) \to \mathcal{SR}(A)$ and $\Gamma_{n,\mathtt{r}} : \mathcal{A}_C(H_n) \to \mathcal{R}(A)$.*

Thus, Theorem 1 reduces the enumeration of $\mathcal{SR}(A)$, a set with no obvious structure, to the enumeration of $\mathcal{A}(H_n)$. Respectively, $\mathcal{R}(A)$ can be obtained by the set of unit circle segments $\mathcal{A}_C(H_n)$. In turn, the efficient enumeration of all faces in $\mathcal{A}(H_n)$, resp. $\mathcal{A}_C(H_n)$, can be realized easily. We conveniently use the incidence graph $\mathcal{I}(H_n)$, which contains a node $v(\varphi)$ for each face $\varphi \in \mathcal{A}(H_n)$ and two nodes $v(\varphi)$ and $v(\varphi')$ are connected by an edge if the faces φ and φ' are incident. For a detailed description of incidence graphs for line arrangements and the complexity of computing them see [9] and [10]. For the partition of the circuit C one can construct analogously the incidence graph $\mathcal{I}_C(H)$ that is just a cyclic list of vertices $v(\varphi)$ faces. The image matching algorithm for $\mathcal{F}_{\mathtt{sr}}$ proposed in [15,16] works as follows

```
The IM Algorithm
 1. Construct the incidence graph $\mathcal{I}(H_n)$;
 2. Perform depth first searching to traverse all nodes $v(\varphi)$ in $\mathcal{I}(H_n)$;
 3.     For each enumerated face $\varphi$ apply $\Gamma_{n,\mathtt{sr}}(\varphi)$ to compute $f(A)$;
 4. Return $f$ which induces the minimum distortion $\Delta(f(A), B)$.
```

For rotations we proceed analogously using the incidence graph $\mathcal{I}_C(H)$ instead of $\mathcal{I}(H_n)$ and the mapping $\Gamma_{n,\mathtt{r}}(\varphi)$ instead of $\Gamma_{n,\mathtt{sr}}(\varphi)$. Ragnar Nevries [25] studied the practical realization and applications of this approach in his diploma thesis.

The enumeration of transformed images implied by the geometrical incidence between the corresponding faces allows that successively enumerated transformed images differ only in few pixels. Hence, our algorithm holds the current transformation A' of A and with every traversing step on $\mathcal{I}(H_n)$, rep. on $\mathcal{I}_C(H_n)$ it updates A' on a constant number of pixels in average. The coordinates of pixels to be updated are computed in advance and are stored as a label *Update*(v) to each node v. This enables that our algorithm finds the best image match under $\mathcal{F}_{\mathtt{sr}}$ in $O(|\mathcal{A}(H_n)|)$ time plus the complexity needed to compute the incidence graph that is linear with respect to $O|\mathcal{A}(H_n)|$, too. For rotations the time complexity is made up similarly. However, in contrast to $\mathcal{I}(H_n)$ the

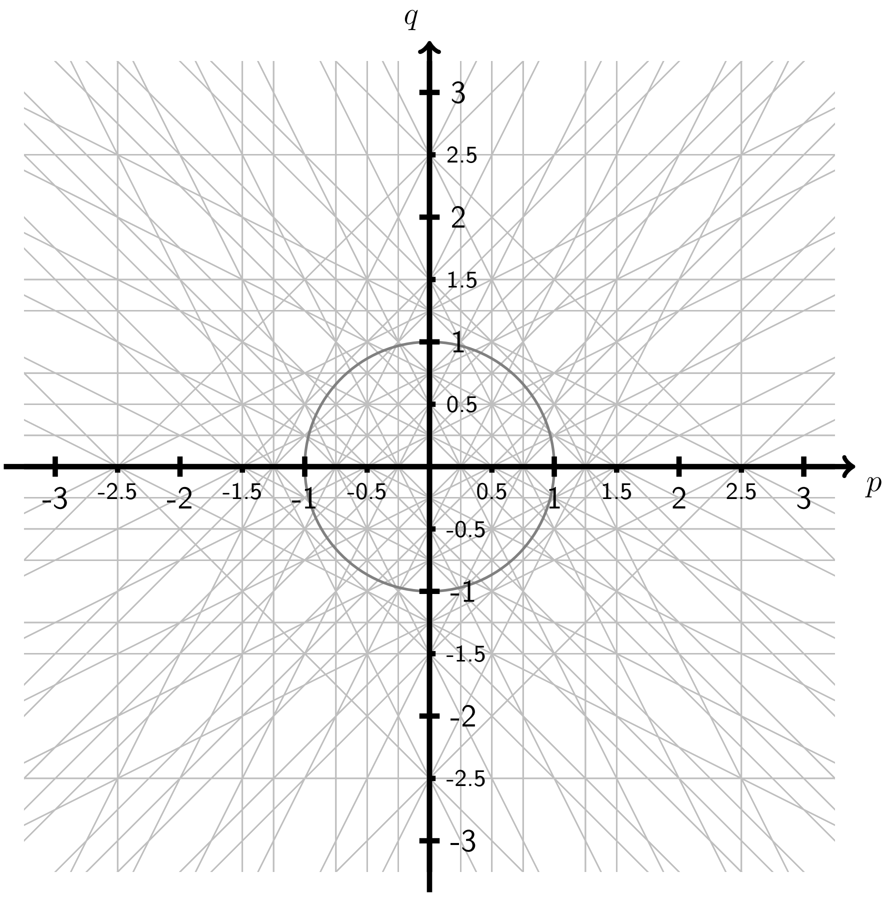


Fig. 1. The parameter space $\mathbb{R}^2$ partitioned by $\mathcal{H}_n$, with $n = 2$, i.e. for images of $5^2 = 25$ pixels. The points in $\mathbb{R}^2$ represents compositions of real scaling and rotation. For $P, P' \in \mathbb{R}^2$ representing f and f', respectively, the transformed images $f(A)$ and $f'(A)$ are equal if P and P' belong to the same face. The unit circle represents all rotations or equivalently all compositions with scaling factor $s = 1$.

computation of $\mathcal{I}_C(H_n)$ needs asymptotically more time than $O(|\mathcal{A}_C(H_n)|)$, i.e., in the previous solution it is higher by a $\log n$ factor. In [15] we were able to give the following estimations:

Theorem 2 ([15]). *The cardinality $|\mathcal{A}(H_n)|$ grows in n by $\Omega(n^5) \cap O(n^6)$ and $|\mathcal{A}_C(H_n)|$ grows in n by $\Theta(n^3)$. As a consequence IM for $\mathcal{F}_{\mathtt{sr}}$ can be done in time bounded by $O(n^6)$ and for $\mathcal{F}_{\mathtt{r}}$ in time bounded by $O(n^3 \log n)$.*

Through the rest of this paper we will show how to eliminate the additional $\log n$ term for rotations using a more sophisticated approach and next we will improve the lower bound on the number of rotated and scaled images to $\Omega(n^6/\log n)$.

4 Fast Computation of the Incidence Graph for Rotations

Figure 3 presents an optimal (i.e., output-linear time) algorithm for the computation of the incidence graph $\mathcal{I}_C(H_n)$ needed for IMP with rotations. The structure of $\mathcal{I}_C(H_n)$ is simply cyclic and given by the alternating sequence of 0- and 1-faces on C which arises from intersections of lines in $\mathcal{H}_n$ with C (cf. Figure 3). To compute $\mathcal{I}_C(H_n)$ our algorithm determines all $\Theta(n^3)$ intersection points and sorts them according to their order on C. The problem is that every intersection point has irrational components. We provide that the right order of them can already be obtained by the highest $O(\log n)$ bits of each coordinate. Moreover, we show that the needed bits can be evaluated in $O(1)$ time with respect to the unit cost model for basic integer operations. Due to this short representations the algorithm finds the order of the intersection points in linear time $O(n^3)$ by the use of Radix Sort. Subsequently, the incidence graph is computed from the obtained structural information.

We first establish equations for the two intersection points $(p_{ijk\pm}, q_{ijk\pm})$ between a line ℓ determined by $h_{ijk} \in H_n$ and C,

$$p_{ijk\pm} = \tfrac{i(k-0.5)\pm j\sqrt{i^2+j^2-(k-0.5)^2}}{i^2+j^2} \quad \text{and} \quad q_{ijk\pm} = \tfrac{j(k-0.5)\mp i\sqrt{i^2+j^2-(k-0.5)^2}}{i^2+j^2}.$$

Here $\pm$ and $\mp$ symbols stand for signs $+$ or $-$. Through the rest of this section we will always assume that $i^2+j^2 \geq (k-0.5)^2$. Otherwise the line determined by h_{ijk} does not cut C and so we do not need to consider it at all. Since we are working on the unit circle an order-preserving unique representation of intersection points is given by only $p_{ijk\pm}$ and the sign of $q_{ijk\pm}$ that we will denote by $\sigma_{ijk\pm}$. We will now show that every two successive intersection points on C have a minimum discrepancy of at least $\Omega(n^{-10})$ in their $p_{ijk\pm}$ components.

Lemma 1. *For all $(i,j),(i',j') \in \mathcal{N}$, $k,k' \in \{-n,\dots,n+1\}$ and $\pm,\pm' \in \{+,-\}$ it is true that the value $|p_{ijk\pm} - p_{i'j'k'\pm'}|$ is either zero or greater than $2^{-14}n^{-10}$.*

Proof (Sketch). Let $P = (p_{ijk\pm}, q_{ijk\pm})$ and $P' = (p_{i'j'k'\pm'}, q_{i'j'k'\pm'})$ be two different intersection points between the unit circle and lines ℓ, given by h_{ijk}, and ℓ', given by $h_{i'j'k'}$, such that the distance between their p-coordinates, i.e., $\Delta p = |p_{ijk\pm} - p_{i'j'k'\pm'}|$, is minimum. The proof finds a lower bound on Δp by the help of a lower bound on the angle α between the vectors P and P'.

Because P and P' are situated on the unit circle it becomes evident that α is at least $d = \|P - P'\|$, the distance between P and P'. To approximate d we have to consider the three possible relations between ℓ and ℓ' which are depicted in Figure 2). In each case we show that $d \geq \frac{1}{68n^5}$.

Case 1: ℓ and ℓ' are identical. We can assume that $i = i'$, $j = j'$, $k = k'$ and that $\pm$ is opposite to $\pm'$. Since $4i^2 + 4j^2 \geq (2k-1)^2 > 0$ we get:

$$d = \|P - P'\| = \sqrt{\frac{4i^2 + 4j^2 - (2k-1)^2}{i^2 + j^2}} \geq \sqrt{\frac{1}{2n^2}} \geq \frac{1}{\sqrt{2}n}.$$

In this simple case d is huge compared to $\Omega(n^{-5})$.

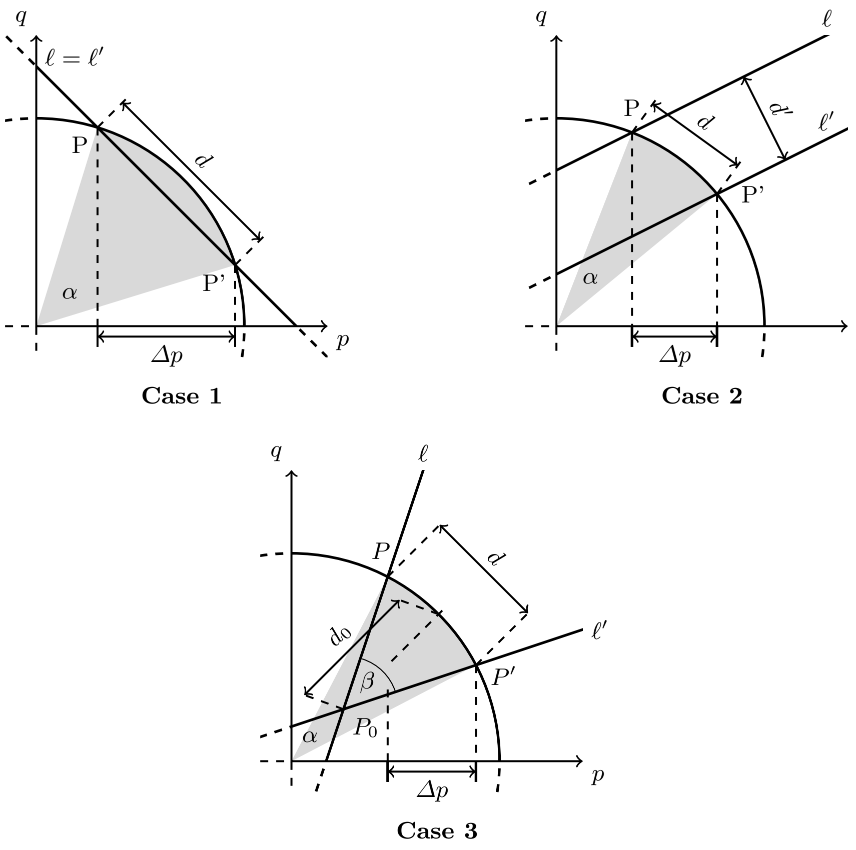


Fig. 2. Three cases for the relative positions of two lines ℓ and ℓ' cutting the unit circle at points $P = (p_{ijk\pm}, q_{ijk\pm})$ and $P' = (p_{i'j'k'\pm'}, q_{i'j'k'\pm'})$. The minimum distance $\Delta p = |p_{ijk\pm} - p_{i'j'k'\pm'}|$ is estimated using the angle α between P and P' which is in turn bounded from below by d. In Case 1 d can be determined directly. Case 2 applies the minimum distance d' between parallel lines as lower bound on d. Finally, Case 3 estimates d by the minimum distance d_0 of the unit circle to the intersection P_0 of ℓ and ℓ' as well as the minimum angle β between ℓ and ℓ'.

Case 2: ℓ and ℓ' are parallel. The second case immediately leads to $d \geq d'$, the distance between parallel lines ℓ and ℓ'. To find the minimum (but non zero) distance d' we choose an arbitrary point on the second line ℓ', e.g., let $(p_0, q_0) = \left(0, \frac{2k'-1}{2j'}\right)$, and measure the distance between (p_0, q_0) and ℓ. Assuming that (p_0, q_0) is not on ℓ we get:

$$d \geq d' = \frac{|ip_0 + jq_0 - (k - 0.5)|}{\sqrt{i^2 + j^2}} = \left| \frac{j(2k' - 1) - j'(2k - 1)}{2j'\sqrt{i^2 + j^2}} \right| \geq \frac{1}{\sqrt{8}n^2}.$$

Thus, in Case 2, d is again huge with respect to $\Omega(n^{-5})$.

Case 3: ℓ and ℓ' intersect in exactly one point P_0. In the most complicated case a lower bound on d is found by the minimum distance d_0 between P_0 and circle C as well as the minimum angle β between ℓ and ℓ'. It can be shown that $d_0 \geq \frac{1}{34n^4}$ for both P_0 inside and outside of C. Moreover, one can prove that the angle $\beta \geq \frac{1}{n}$. If $\beta \geq \frac{\pi}{2}$ then it can be shown directly that $d \geq \Omega(n^{-4})$ and thus, we assume that the opposite inequality: $\beta < \frac{\pi}{2}$ is true. Then, in the worst case it happens that the triangle determined by P, P' and the base point P_0 is isosceles. It P_0 is outside of C this leads easily to:

$$d \geq 2h \tan \frac{\beta}{2} \geq \frac{2}{34n^4} \tan \frac{1}{2n} \geq \frac{1}{34n^5}$$

where $h < d_0$ is the height of the triangle. If P_0 is inside C and if $\beta < \frac{\pi}{2}$, then $h < \frac{d_0}{2}$ and it follows that $d \geq \frac{1}{68n^5}$.

The lower bound on d gives that $\alpha \geq \frac{1}{68n^5}$ which provides the necessary requirements to estimate the minimum nonzero gap Δp between p-coordinates. In worst case situation one of the values $p_{ijk\pm}$ or $p_{i'j'k'\pm'}$ is either 1 or -1 and it follows:

$$\Delta p = 1 - \cos\alpha \geq 1 - \cos\left(\frac{1}{68n^5}\right) \geq 2^{-14}n^{-10}. \qquad \square$$

According to the previous lemma it is sufficient to compute the p-coordinates with an precision of at most $\Omega(n^{-10})$. Since the p-components are between -1 and 1 our algorithm thus evaluates only $O(\log n)$-bits-representations, though the exact values are even non-rational. In particular, $14 + 10\log_2 n$ bits and one sign bit are enough. The following lemma argues that this short number representations can even be computed by a constant number of basic arithmetic integer operations and thus, in constant time.

Lemma 2. *Let $b = \lceil \log_2 n \rceil$. Then for all $(i,j) \in \mathcal{N}$, $k \in \{-n, \ldots, n+1\}$ and $\pm \in \{+,-\}$ a $25b$-bit approximation $\tilde{p}_{ijk\pm}$ of $p_{ijk\pm}$ can be computed by a constant number of integer addition and multiplication operations.*

It remains to show that the algorithm in Figure 3 works correctly in time $O(n^3)$, which is quite trivial now.

Theorem 3. *The incidence graph $\mathcal{I}_C(H_n)$ can be computed in time $O(n^3)$.*

Proof. Let $b = \lceil \log_2 n \rceil$. The algorithm computes the $25b$-bit representations $\tilde{p}_{ijk\pm}$ and the signs $\sigma_{ijk\pm}$ for all $(i,j) \in \mathcal{N}$, $k \in \{-n, \ldots, n+1\}$ and $\pm \in \{+,-\}$. Lemma 1 guarantees that maintaining the $25b$-bit precision is sufficient to differentiate between the intersection points since $25b \geq 14 + 10\log_2 n$. Moreover by Lemma 2 the representations can be computed in time $O(|\mathcal{A}_C(H_n)|)$. Next, according to the signs $\sigma_{ijk\pm}$, the algorithm splits the representations $\tilde{p}_{ijk\pm}$ into two subsets R_+ and R_-. This can be done in $O(|\mathcal{A}_C(H_n)|)$ time, too.

Subsequently, R_+ and R_- are sorted individually by Radix Sort. The algorithm uses the radix 2^b and thus, iterates 25 times over all elements of R_+ and R_-, respectively. Hence, the whole sorting takes $O(|\mathcal{A}_C(H_n)|)$ time.

Algorithm `ComputeRotationIncidenceGraph` /* *Computation of* $\mathcal{I}_C(H_n)$ */
Input: Integer number n.
Output: The incidence graph $\mathcal{I}_C(H_n)$.

```
begin                                   /* Main( ) for Computation of $\mathcal{I}_C(H_n)$ */
  $R_- = \emptyset$; $R_+ = \emptyset$;
  for all $(i,j) \in \mathcal{N}, k \in \{-n,\dots,n+1\}, \pm \in \{+,-\}$ do begin
    $\tilde{p}_{ijk\pm} \approx \frac{i(k-0.5)\pm j\sqrt{i^2+j^2-(k-0.5)^2}}{i^2+j^2}$;    /* get $25b$-bit representation */
    if $(j(k-0.5) < \mp i\sqrt{i^2+j^2-(k-0.5)^2})$ then        /* compute $\sigma_{ijk\pm}$ */
      $R_- = R_- \cup \{(\tilde{p}_{ijk\pm}, i, j, k)\}$ else $R_+ = R_+ \cup \{(\tilde{p}_{ijk\pm}, i, j, k)\}$;
  end;
  call RADIXSORT($R_-, n$); call RADIXSORT($R_+, n$);
  append list REVERSE($R_-$) to list $R_+$ and obtain sorted list $R$;
  $V = \{v_0\}$; $E = \emptyset$; $\tilde{p}_{prev} = \infty$; $w = 1$;
  for $(\tilde{p}, i, j, k)$ in $R$ do begin
    if $(\tilde{p} \neq \tilde{p}_{prev})$ then begin
      $V = V \cup \{u_w, v_w\}$;            /* add $u$ for 0-face and $v$ for 1-face */
      add $(i,j,k)$ to the Update set of $v_w$;
      $E = E \cup \{(v_{w-1}, u_w), (u_w, v_w)\}$; $\tilde{p}_{prev} = \tilde{p}$; $w = w + 1$;
    end else add $(i,j,k)$ to the Update set of $v_w$;
  end;
  unify $v_0$ and $v_{w-1}$; return $\mathcal{I}_C(H_n) = (V, E)$;
end.
```

Fig. 3. The algorithm computing $\mathcal{I}_C(H_n)$. Firstly, all intersection points between lines and C are determined. Then the method `RADIXSORT` sorts the lists of intersection points by the use of 2^b buckets where $b = \lceil \log_2 n \rceil$. Finally the incidence graph is generated from the sorted lists.

Finally, the graph $\mathcal{I}_C(H_n)$ is generated from traversing the two sorted lists which takes again time linear in $|\mathcal{A}_C(H_n)|$. This completes the proof because $|\mathcal{A}_C(H_n)| \in O(n^3)$ by [2]. $\square$

5 The Lower Bound on the Number of Different Compositions of Scaling and Rotation

In [15] we presented an image matching algorithm under combinations of scaling and rotation which essentially tries out all possible transformations of an image A by traversing all faces in the set $\mathcal{A}(H_n)$. Although we performed this searching strategy optimally, i.e., in time linear in $|\mathcal{A}(H_n)|$, we could not tell what is the exact time complexity (as a function of the input size) of our algorithm. Since we have shown that $|\mathcal{A}(H_n)| \in O(n^6) \cap \Omega(n^5)$ we claimed a worst case running time of $O(n^6)$. However it was open whether its time complexity is asymptotically smaller than n^6. In this section we provide strong evidence that this upper bound is very accurate and that in the worst case no exhaustive search of $\mathcal{SR}(A)$ could work significantly faster.

Theorem 4. *The cardinality of $|\mathcal{A}(H_n)|$ grows with $n \in \mathbb{N}$ by at least $\Omega(n^6/\log n)$.*

Proof (Sketch). To prove the theorem we show that there are at least $\Omega(n^6/\log n)$ 1-faces in $\mathcal{A}(H_n)$. Though this would already be enough to complete the proof the application of planar graph properties deduces shortly that there are also $\Omega(n^6/\log n)$ 0- and 2-faces in $\mathcal{A}(H_n)$. This is in particular useful when estimating $|\mathcal{SR}(A)|$ since at least every 2-face of $\mathcal{A}(H_n)$ corresponds to a unique transformation of image A (though not necessarily to some unique transformed image A', which heavily depends on A).

We will now outline how to establish the given number of 1-faces in $\mathcal{A}(H_n)$. We assume that $n \geq 100$ and for convenience we let $m = \lceil 0.01n \rceil$. Note that 1-faces are created by the mutual intersections of lines in $\mathcal{H}_n$. Let consider two subsets of $\mathcal{H}_n$, namely $\mathcal{H}_n^{\mathsf{a}}$ and $\mathcal{H}_n^{\mathsf{b}}$, which contain all lines given by the equations in

$$H_n^{\mathsf{a}} = \{h_{ijk} : ip + jq = k - 0.5 \mid 0.5n < i, -j \leq n, 1 \leq k \leq m, \gcd(i,-j) = 1\}$$
$$H_n^{\mathsf{b}} = \{h_{ijk} : ip + jq = k - 0.5 \mid (i,j) \in \mathcal{N}, 1 \leq k \leq m\}.$$

Subsequently let $\mathcal{A}_n$ be the set of all one-dimensional subspaces which are gained from cutting a line in $\mathcal{H}_n^{\mathsf{a}}$ by lines in $\mathcal{H}_n^{\mathsf{b}}$. Clearly, the cardinality $|\mathcal{A}_n|$ must be less than the number of all 1-faces because any 1-face is either a subspace of exactly one $\varphi \in \mathcal{A}_n$ and thus, counted only once, or not a subspace of any line in $\mathcal{H}_n^{\mathsf{a}}$ and then not counted at all.

To get $|\mathcal{A}_n|$ we look at every line $\ell \in \mathcal{H}_n^{\mathsf{a}}$ and determine the number of subspaces it is cut into by the lines $\ell' \in \mathcal{H}_n^{\mathsf{b}}$ (for an illustration, see Figure 4). This number is greater (by one) than the number of intersection points on ℓ. All relevant intersection points (p, q) on line ℓ, given by equation $h_{ijk} \in H_n^{\mathsf{a}}$, are created by some line ℓ', given by equation $h_{i'j'k'} \in H_n^{\mathsf{b}}$, and have the form

$$p = \frac{j'(2k-1) - j(2k'-1)}{2ij' - 2i'j} \quad \text{and} \quad q = \frac{i(2k'-1) - i'(2k-1)}{2ij' - 2i'j}.$$

The following lemma plays a crucial role in our proof.

Lemma 3. *For every line $\ell \in \mathcal{H}_n^{\mathsf{a}}$ the number of intersections with lines in $\mathcal{H}_n^{\mathsf{b}}$ is $\Omega(n^3/\log n)$.*

To show the lemma, note that since ℓ is not vertical it is sufficient to estimate the number of different p-coordinates. For this purpose we estimate a lower bound on how many lines $\ell' \in \mathcal{H}_n^{\mathsf{b}}$ give a fraction $\frac{s}{t}$ where (1) $s = j'(2k-1) - j(2k'-1)$, (2) $t = 2ij' - 2i'j$, (3) $t = 2x$ with x a prime number and (4) $s < x$. Firstly, on basis of $\gcd(i, -j) = 1$ we get that x can become any prime number in $\{4mn, \ldots, -jn\}$. Secondly, we provide that for any choice of t we can still get m different numerators s with $s < x$. Obviously it is true that all fractions constructed like this correspond to some intersection point on ℓ and are mutually not equal.

Thus we derive the $\Omega(n^3/\log n)$ bound on the number intersection points for every $\ell \in \mathcal{H}_n^{\mathsf{a}}$. Summed over all $\Omega(n^3)$ lines in H_n^{a} we obtain $\Omega(n^6/\log n)$ elements in $\mathcal{A}_n$. This completes the proof. $\square$

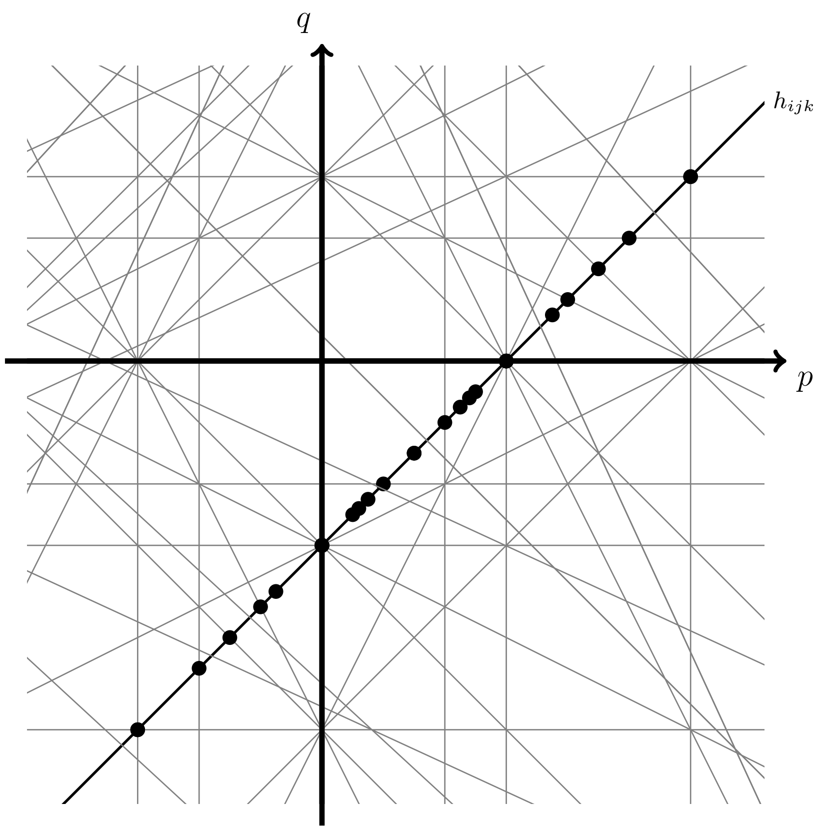


Fig. 4. The intersection points between the line ℓ described by $h_{ijk} \in H_3^{\mathsf{a}}$ and all lines in $\mathcal{H}_3^{\mathsf{b}}$, where $(i, j, k) = (2, -2, 1)$. For clearness of presentation we let $m = 1$ instead of $m = \lceil 0.01n \rceil$ as has been assumed in the proof.

Acknowledgment

The authors thank Andreas Jakoby, Ragnar Nevries and Ulf Ochsenfahrt for fruitful discussions and the anonymous reviewers for helpful suggestions.

References

1. Bovik, A. (ed.): Handbook of Image and Video Processing. Academic Press, San Diego (2000)
2. Amir, A., Butman, A., Crochemore, M., Landau, G., Schaps, M.: Two-dimensional pattern matching with rotations. Theor. Comput. Sci. 314(1-2), 173–187 (2004)
3. Amir, A., Butman, A., Lewenstein, M., Porat, E.: Real two dimensional scaled matching. Algorithmica 53(3), 314–336 (2009)

4. Amir, A., Chencinski, E.: Faster two-dimensional scaled matching. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 200–210. Springer, Heidelberg (2006)
5. Amir, A., Kapah, O., Tsur, D.: Faster two-dimensional pattern matching with rotations. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 409–419. Springer, Heidelberg (2004)
6. Brent, R.P.: Fast multiple-precision evaluation of elementary functions. Journal of the Association for Computing Machinery 23(2), 242–251 (1976)
7. Brown, L.G.: A survey of image registration techniques. ACM Computing Surveys 24(4), 325–376 (1992)
8. Cox, I.J., Bloom, J.A., Miller, M.L.: Digital Watermarking, Principles and Practice. Morgan Kaufmann, San Francisco (2001)
9. Edelsbrunner, H.: Algorithms in Combinatorial Geometry. Springer, Berlin (1987)
10. Edelsbrunner, H., O'Rourke, J., Seidel, R.: Constructing arrangements of lines and hyperplanes with applications. SIAM J. Comput. 15, 341–363 (1986)
11. Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers. Oxford University Press, Oxford (1954)
12. Fredriksson, K., Navarro, G., Ukkonen, E.: Optimal exact and fast approximate two-dimensional pattern matching allowing rotations. In: Apostolico, A., Takeda, M. (eds.) CPM 2002. LNCS, vol. 2373, pp. 235–248. Springer, Heidelberg (2002)
13. Fredriksson, K., Ukkonen, E.: A rotation invariant filter for two-dimensional string matching. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 118–125. Springer, Heidelberg (1998)
14. Hundt, C., Liśkiewicz, M.: On the complexity of affine image matching. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 284–295. Springer, Heidelberg (2007)
15. Hundt, C., Liśkiewicz, M.: Two-dimensional pattern matching with combined scaling and rotation. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 5–17. Springer, Heidelberg (2008)
16. Hundt, C., Liśkiewicz, M., Nevries, R.: A combinatorial geometric approach to two-dimensional robustly pattern matching with scaling and rotation. To appear in Theor. Comput. Sci.
17. Indyk, P.: Algorithmic aspects of geometric embeddings. In: Proc. FOCS 2001, pp. 10–33 (2001)
18. Indyk, P., Motwani, R., Venkatasubramanian, S.: Geometric matching under noise: Combinatorial bounds and algorithms. In: Proc. SODA 1999, pp. 354–360 (1999)
19. Kasturi, R., Jain, R.C.: Computer Vision: Principles. IEEE Computer Society Press, Los Alamitos (1991)
20. Kenyon, C., Rabani, Y., Sinclair, A.: Low distortion maps between point sets. In: Proc. STOC 2004, pp. 272–280 (2004)
21. Kropatsch, W.G., Bischof, H. (eds.): Digital Image Analysis - Selected Techniques and Applications. Springer, Berlin (2001)
22. Landau, G.M., Vishkin, U.: Pattern matching in a digitized image. Algorithmica 12(3/4), 375–408 (1994)

23. Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. Medical Image Analysis 2(1), 1–36 (1998)
24. Modersitzki, J.: Numerical Methods for Image Registration. Oxford University Press, Oxford (2004)
25. Nevries, R.: Entwicklung und Analyse eines beschleunigten Image Matching-Algorithmus für natürliche Bilder, Diplomarbeit, Universität Rostock (2008)
26. Nouvel, B., Rémila, E.: Incremental and transitive discrete rotations. In: Reulke, R., Eckardt, U., Flach, B., Knauer, U., Polthier, K. (eds.) IWCIA 2006. LNCS, vol. 4040, pp. 199–213. Springer, Heidelberg (2006)
27. Papadimitriou, C., Safra, S.: The complexity of low-distortion embeddings between point sets. In: Proc. SODA 2005, pp. 112–118 (2005)

Online Approximate Matching with Non-local Distances

Raphaël Clifford and Benjamin Sach

University of Bristol, Dept. of Computer Science, Bristol, BS8 1UB, UK
{clifford,sach}@cs.bris.ac.uk

Abstract. A black box method was recently given that solves the problem of online approximate matching for a class of problems whose distance functions can be classified as being local. A distance function is said to be local if for a pattern P of length m and any substring $T[i, i+m-1]$ of a text T, the distance between P and $T[i, i+m-1]$ is equal to $\Sigma_j \Delta(P[j], T[i+j-1])$, where Δ is any distance function between individual characters. We extend this line of work by showing how to tackle online approximate matching when the distance function is non-local. We give solutions which are applicable to a wide variety of matching problems including function and parameterised matching, swap matching, swap-mismatch, k-difference, k-difference with transpositions, overlap matching, edit distance/LCS, flipped bit, faulty bit and L_1 and L_2 rearrangement distances. The resulting unamortised online algorithms bound the worst case running time *per input character* to within a log factor of their comparable offline counterpart.

1 Introduction

A great deal of progress has been made in finding fast algorithms for a variety of important forms of approximate matching in the last few decades. The most common computational model in which these algorithms have been analysed assumes that the text and pattern are to be held in fast primary storage and that each query to the data has constant cost. However, increasingly it has become apparent that new applications such as those found in telecommunications or monitoring Internet traffic require a fresh approach. It may no longer be possible to store the entirety of the text and the worst case time per input character is often more important than the overall running time of any algorithm.

The model that we consider is a deterministic variant of data streaming where we assume we are given a pattern in advance and the text to which it is to be matched arrives one character at a time. The overall task is to report matches between the pattern and text as soon as they occur and to bound the worst case time *per input character*. Previous work in this model showed how to convert offline algorithms for approximate pattern matching problems with simple distance functions into efficient online ones using a black box approach [8]. It is an important feature of both our approach and the previous work that the running time of the resulting algorithms is not amortised.

The main restriction for this black box solution was that the distance function defined by the approximate matching problem had to have the property of being *local*. A local distance function is defined to be one where the distance between a pattern P and a substring of the text T can be written as $\Sigma_j \Delta(P[j], T[i+j-1])$, where Δ is any distance function between individual characters in the input alphabet. In other words, the distance was simply measured as the sum of the distances between individual symbols. Although a number of interesting problems including exact matching with wildcards, matching under the Hamming norm and numerical measures such as the L_2 and L_1 norm have distance functions which are local, this left open the problem of how to handle the many matching problems with more sophisticated distance measures.

To appreciate the challenges that arise in online pattern matching when the distance function is non-local, consider for example the problem of function matching [3]. There is a function match between pattern P and $T[i, i+m-1]$ if there exists a function f (possibly distinct for each i) from the input alphabet Σ to itself such that $T[i+j] = f(P[j])$ for all $0 \leq j < m$. For example aba has a function match with $T = xyx$ but not $T' = xyy$ as a can not be mapped to two different letters. In the previous black box approach to creating online algorithms which we briefly describe in Section 2, distances would be found independently for different substrings of the pattern and the results combined. However, in this case whether $P[3] = a$ matches $T'[3] = y$ depends on the function chosen to map the characters in $P[1, 2]$ and vice versa. By definition only one choice of function is permitted for the whole of the pattern. As a result any matchings for the substrings of P would appear to have to depend on the results for all other substrings. In general for non-local distance functions, we must find a way efficiently to handle the dependencies between different parts of the pattern.

Our contribution is to present three general methods which can be applied successfully to convert a wide variety of non-local approximate matching problem into efficient unamortised online ones. We will refer to such algorithms as pseudo-realtime (PsR) throughout the paper by analogy to the realtime model for linear time algorithms. The techniques are necessarily no longer 'black box', depending in detail on the specific offline algorithm being considered. As with the previous work for local distance functions, the running time per input character is guaranteed to be within a log factor of its offline counterpart.

2 Preliminaries and Previous Work

Throughout the paper, T and P will be used to denote the text and pattern strings respectively. By convention, $|T| = n$ and $|P| = m$. For any other strings, a lowercase letter is used to denote length, for example $|A| = a$. $A||B$ denotes the concatenation of strings A and B. The character alphabet is denoted Σ (and Σ_P for the pattern alphabet). When discussing the alignment of the pattern and text we will often refer to *right alignments*. Right alignment i of P and T aligns the final character of P with the ith character of T. This is a natural way to discuss alignments when the text is being streamed. The usual offline notion

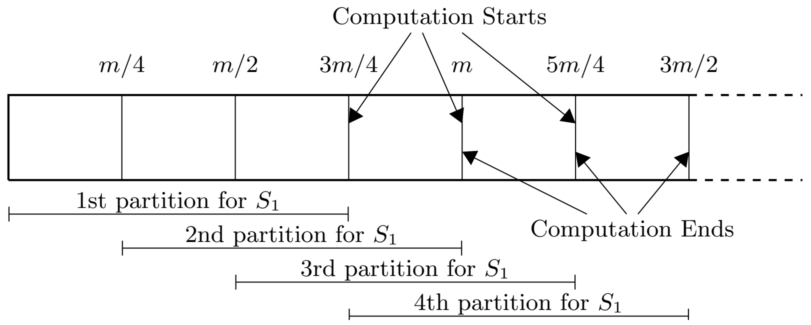


Fig. 1. Partitioning of the text for subpattern S_1

where the first character of the pattern is aligned at a position in the text will be termed left alignment to avoid confusion.

The ideas we present build on the black box algorithm of [8] for local distance functions which we briefly describe here. The basic idea is to split the pattern into $O(\log m)$ consecutive subpatterns each having half the length of the previous one. The first subpattern $S_1 = P[1, m/2]$ and subpattern S_j has length $m2^{-j}$ for $1 \leq j < s$ where $s = \log_2(m)+1$. S_s is set to be the last character of the pattern. The offline algorithm is then run for each subpattern against the whole of the text with the distances found added to an auxiliary array C. In this way, for any subpattern starting at position j of the pattern, its distance to a substring starting at position i of the text will be added to the count at $C[i-j+1]$. At the end of this step C will contain $\Delta(P, T[i, i+m-1])$ for every location i in t.

To ensure that the work for each subpattern is completed before its result is needed to report a match, the text was partitioned into overlapping substrings. Each of the $O(\log m)$ subpatterns has a different length and induces a different and independent partitioning of the text. Each partition of the text is set to be of size to $3|S_j|/2$, with an overlap of length $|S_j|$. For each subpattern, the work of a search does not have to be completed until $|S_j|/2$ characters after it starts and so we can set this work to be performed over the period between arrival of $T[i]$ and $T[i+|S_j|/2]$ as shown in Figure 1. This gives a space requirement of $O(m)$. Let $T(n,m)$ be the time complexity of the offline algorithm used as a black box. The running time per text input character was shown to be $O(\Sigma_{j=1}^{\log_2 m} T(n, 2^{j-1})/n))$ which is bounded above by $O(T(n,m)\log(m)/n)$.

3 Our Results

We present an overview of problems that we solve in pseudo-realtime and the methods developed. Due to space restrictions we are not able to discuss each problem in detail, but to explain the main ideas we present examples for each of the main techniques.

- Some approximate matching problems with non-local distance functions translate immediately to the pseudo-realtime setting with no asymptotic time penalty and minimal modification. For example, edit distance and Longest Common Subsequence (LCS) where the offline algorithm completes the full dynamic programming table and faulty bits and L_1 rearrangement distance, where the offline solutions considers each alignment of the pattern and text separately are all naturally pseudo-realtime. As another example, the algorithm of Amir et al. [6] for the parameterised matching problem is a modification of KMP that allows a direct translation by applying the realtime modifications of Galil [9].
- Many pattern matching algorithms rely heavily on cross-correlations implemented via the fast Fourier transform (FFT). We show in Section 4.1 how these problems can be made pseudo-realtime efficiently by application of a PsR cross-correlation algorithm, using the L_2 rearrangement distance problem as an example.
- We next develop a method we call 'Split and Correct' in Section 4.2. The aim is to split the pattern into subpatterns as in Section 2 and then correct for the non-local effects that occur across the boundaries between adjacent subpatterns. Pseudo-realtime swap-mismatch is given as an example.
- Finally we present a method we call 'Split and Feed' in Section 4.3. This method is applied to problems where matching a subpattern to the text can affect the alignment of other subpatterns with the text. We explain this method using the k-differences problem as an example. We also comment that the k-difference with transpositions problem can be solved by combining this technique with that of Split and Correct.

Table 1. Summary of main pseudo-realtime pattern matching results

Problem	Offline per char time	Online/PsR penalty	**Method**
local matching	various	$O(\log m)$	Splitting [8]
function	various [3]	$O(\log m)$	PsR Cross-correlations
parameterised	$O(\log \|\Sigma\|)$ [6]	$O(1)$	Realtime KMP
edit distance/LCS	$O(m)$ [11]	$O(1)$	Immediate
k-differences	$O(k)$ [10]	$O(\log m)$	Split & Feed
swap-mismatch	$O(\sqrt{m \log m})$ [5]	$O(1)$	Split & Correct
swap	$O(\log m \log \|\Sigma\|)$ [4]	$O(\log m)$	Split & Correct
overlap	$O(\log n)$ [4]	$O(\log m)$	Split & Correct
k-diff with transpositions	$O(k)$ [10]	$O(\log m)$	Split & Correct + Feed
self normalised	$O(\log m)$ [7]	$O(\log m)$	PsR Cross-correlations
faulty bits	$O(m \log m)$ [2]	$O(1)$	Immediate
flipped bits	$O(\log m)$ [2]	$O(\log m)$	PsR Cross-correlations
L_1 rearrangement	$O(m)$ [1]	$O(1)$	Immediate
L_2 rearrangement	$O(\log m)$ [1]	$O(\log m)$	PsR Cross-correlations

A brief summary of these results along with the multiplicative time penalty incurred for the corresponding online/PsR algorithms is given in Table 1. This list is intended to be exemplary rather than comprehensive and in particular we anticipate that our methods can be applied equally successfully to a larger range of problems we have yet to consider. In each case we require at most $O(m)$ space. Note that in the case of edit distance there also exists an $O(n(1 + \frac{m}{\log n}))$ time offline solution for finite alphabets [12].

4 Algorithms for Pseudo-realtime Translation

We now give an overview of each method along with examples of their application.

4.1 Pseudo-realtime Cross-Correlation Method

The cross-correlation is an important technique in pattern matching and lies at the heart of many of the fastest algorithms known. We discuss a class of non-local problems that can be made pseudo-realtime simply and efficiently by using as its main tool the replacement of the offline cross-correlation with a pseudo-realtime version. Lemma 1 gives the running time per input character.

Lemma 1 (Pseudo-realtime Cross-correlation). *Let X be an array received online and Y an array received in advance. For any i, when character $X[i]$ arrives, we can compute $(X \otimes Y)[i - m + 1] = \sum_{j=0}^{y-1} X[i + j - m + 1]Y[j]$ in $O(\log^2 m)$ time. As the cross-correlation is local, this is immediate from application of the black box method of [8].*

For the function matching problem, Amir et al.[3] give a solution for small pattern alphabets in $O(n|\Sigma_P| \log m)$ and a randomised solution to the general problem that runs in $O(n \log m)$ time with failure probability $1/n$ of declaring a false positive. Both algorithms can be made pseudo-realtime in $O(|\Sigma_P| \log^2 m)$ and $O(\log^2 m)$ time per character respectively using PsR cross-correlations and by reordering the computation of the offline algorithms.

The L_2 rearrangement distance problem first introduced by Amir et al. [1] allows us to describe a slightly more sophisticated application of this general method. At right alignment i, consider all permutations π so that $T[i-m+j] = P[\pi(j)]$ for all j and define $cost(\pi) = \sum_j |j - \pi(j)|^2$. The L_2 rearrangement distance is defined to be the minimum *cost* over all such permutations (or ∞ if no such permutation exists). It is clear from the definition that this is a highly non-local problem. Analysis of the offline $O(n \log m)$ solution shows that the main challenge lies in its cross-correlation stage but that the remaining work still requires careful scheduling for the overall technique to be successful. We present a pseudo-realtime version of their algorithm which runs in $O(\log^2 m)$ time per character using $O(m)$ total space, equalling the space requirements for the offline solution.

For all $a \in \Sigma$, let $\psi_a(X)$ be an array of the indices of all occurrences of character a in X; further we define $occ_a(X) = |\psi_a(X)|$. Consider the following functions:

$$F_x(P', T')[i'] = \sum_{j=0}^{|P'|-1} (P'[j] - T'[i' + j - |P'|] + x)^2 . \tag{1}$$

$$G_x(P, T)[i] = \sum_{a \in \Sigma} F_x(\psi_a(P), \psi_a(T), a)[occ_a(T[1, i])] . \tag{2}$$

Amir et al. show that if we set $x = (i - m)$ then $G_{(i-m)}(P, T)[i]$ is exactly the distance between $T[i-m+1, i]$ and P. This assumes that the distance is less than ∞ which can be checked in $O(\log m)$ time per character. Further, it is shown that if we can calculate $G_x(P, T)[i]$ for fixed $x = 0, 1, 2$ then $G_{(i-m)}(P, T)[i]$ can be computed by polynomial interpolation in constant time per character. Therefore, in the remainder we need only consider a fixed x.

Observe that the sums, $G_x(P, T)[i - 1]$ and $G_x(P, T)[i]$ differ only at the term where $a = T[i]$. Thus, if we can update the corresponding F_x when we receive $T[i]$ in pseudo-realtime, a sliding window approach will allow us to compute $G_x(P, T)[i]$. To compute the F_x terms in pseudo-realtime we split the data and computation by symbol. When a symbol $T[i] = a$ arrives we consider this as the arrival of a new index for the array $\psi_a(T)$. In this way we create one array of indices per character in the input alphabet and can consider each separately. It is important for the pseudo-realtime algorithm that when a symbol a arrives, the only work that is carried out relates to the array $\psi_a(T)$ and no others. The computation of F_x can therefore be computed independently for each symbol. The classification of the arriving character can be handled using a binary search tree in $O(\log m)$ time.

It remains to show how to compute $F_x(P', T')[i']$ efficiently in pseudo-realtime. By multiplying out F_x observe that it can be computed using PsR cross-correlations and a sliding window. Applying Lemma 1 the resulting pseudo-realtime algorithm runs in $O(\log^2(|P'|))$ time per character and $O(|P'|)$ space. However, $|P'| \leq m$ so $O(\log^2(|P'|)) \in O(\log^2 m)$ time per character and this dominates the overall time complexity. The total space is dominated by the working space of computing each F_x. For each a, $|P'| = occ_a(P)$ giving a total space of $\sum_a O(occ_a(P)) \in O(m)$. Theorem 1 summarises the result.

Theorem 1. *The L_2 rearrangement distance problem can be solved in pseudo-realtime in $O(\log^2 m)$ time per character and $O(m)$ space.*

4.2 Split and Correct

The 'Split and Correct' method we develop in this Section can most easily be applied to non-local pattern matching problems where the distance function between the pattern and substrings of the text can be expressed as the cost of a sequence of moves. In the pseudo-realtime setting, a non-local move is defined to be one which changes characters in more than one of the subpatterns in the split pattern. We consider in particular, problems where the number of possible non-local moves with respect to a given subpattern is bounded by a constant. For this class of problems, we split the pattern into subpatterns as before and create a set of transformed subpatterns by applying all valid combinations of non-local moves to each subpattern. Matches of all of these patterns with the text can be found with no effect on time complexity as the number of such moves is constant per subpattern. For each boundary between two adjacent subpatterns, we will now need to compute the number and type of non-local moves that would occur in a globally optimal alignment between pattern and text. This allows us to select the appropriate transformed subpatterns at each alignment and recombine the results.

To make the explanation concrete, we show how this general method can be applied to the Swap-Mismatch problem. The related *Swap* matching and *Overlap* matching problems, first addressed by Amir et al. [4] can also be solved by the method detailed above although in the latter case a slight generalisation of the notion of a move is required.

Swap-Mismatch. Swap-Mismatch distance between equal length strings is the minimum number of moves required to transform P into T (referred to as $cost(P, T)$). The valid moves are *swap* (swap two adjacent characters) and *mismatch* (replace a character). As overlapping swap and mismatch operations can always be replaced by two mismatches at no extra cost, the minimal cost transformation need never apply two moves to the same character. On non-equal length strings, at right alignment i, the distance is defined to be $cost(P, T[i-m+1, i])$. The solution we present gives an $O(\sqrt{m \log m})$ time per character solution if applied to the best known offline method of Amir et al. [5].

Let l_j and r_j be the leftmost and rightmost indices of subpattern S_j respectively (split as before). Following the Split and Correct method, we define a set of 'boundary indicators' for all $0 < j < s$: $b_{ij} = 1$ if $P[r_j]$ and $P[l_{j+1}]$ are swapped in some minimal cost transformation of P into $T[i - m + 1, i]$ and 0 otherwise. Trivially, we let $b_{i0} = b_{is} = 0$ for all i. The remainder of the section explains first how to use these indicators and secondly, how to compute them efficiently.

A black box solution using boundary indicators. For any subpattern S_j, the valid non-local moves are swaps at each end, giving a total of four transformed subpatterns. For $x, y \in \{0, 1\}$, let $S_j^{(x)(y)} = P[l_j + x, r_j - y]$ represent these transformed subpatterns. We ignore the swapped characters at the boundaries at this stage as the costs incurred by them will be accounted for by the boundary indicators. Recall that in the black box method of [8] there is a final stage of accumulation of the distances found between subpatterns and the different substrings of the text into an auxiliary array C. To compute the swap-mismatch distance from $S_j^{(x)(y)}$ to T for all j and all x, y we apply this method to an offline swap-mismatch algorithm but modify this final stage. Having computed the distances for each $S_j^{(x)(y)}$, we use the boundary indicators to pick which $S_j^{(x)(y)}$ to include in the sum at each alignment and therefore to add to C.

Lemma 2 shows that we are therefore able to calculate $cost(P, T[i-m+1, i])$ at each right alignment i with additive $O(\log m)$ time per alignment.

Lemma 2. *At right alignment i, the distance from P to T is equal to $\sum_{j=1}^{(s-1)} b_{ij}$ plus for each j, the distance from $S_j^{(b_{i(j-1)})(b_{ij})}$ to T at right alignment $i - m + r_j - b_{ij}$.*

Computing the boundary indicators. Lemma 3 allows us to find the boundaries across which swaps will occur in an optimal transformation. Both conditions can readily be checked in constant time per character. In the overall algorithm, we wish to compute boundary indicators for $O(\log m)$ different boundaries, requiring $O(\log m)$ time per text character. We define $y(xy)^*$ to be a "y" followed by zero or more copies of "xy".

Lemma 3. *If $n = m$, there is an optimal swap-mismatch transformation of P into T where a swap occurs across the boundary between $P[i] = x$ and $P[i+1] = y$ iff*

1. *$P[i] = T[i+1]$ and $P[i+1] = T[i]$*
2. *There exists an odd ℓ such that $T[i-\ell+1..i] = y(xy)^*$, $P[i-\ell+1..i] = x(yx)^*$ and $P[i-\ell] \neq y$ or $T[i-\ell] \neq y$*

Overall, the algorithm performs three steps, all in pseudo-realtime:

1. Calculate matches of T against $S_j^{00}, S_j^{01}, S_j^{10}$ and S_j^{11} for all j at all alignments. This is done using the black box method applied to the offline method of Amir et al. in $O(\sqrt{m \log m})$ time per character.
2. Calculate the boundary indicators at all alignments. This is computed using the method above in $O(\log m)$ time per character.
3. Combine the results of steps one and two using the relation stated in Lemma 2. This is computed directly and requires $O(\log m)$ time per character.

Theorem 2 gives the running time for pseudo-realtime swap-mismatch.

Theorem 2. *The swap-mismatch problem can be solved pseudo-realtime in $O(\sqrt{m \log m})$ time per character and $O(m)$ space.*

4.3 Split and Feed

The final technique that we discuss is termed 'Split and Feed'. Here we consider pattern matching problems where the non-local nature of the distance function affects the alignment of subpatterns. Where the distance function is local, the positions of alignments of all subpatterns is fixed for a given alignment of the whole pattern and text. However for problems where insertion and deletion are permitted, for example, this no longer holds and we can no longer apply the previously described Split and Correct method. Consider matching a pattern $P = A||B$ against some text T where A and B are substrings. Under such distance functions, optimal matches of P against T may be composed of sub-optimal matches of A and B. Edit distance and the k-differences problem have this property. Therefore, we cannot compute matches of P by separately computing matches of its sub-patterns.

As before the method splits the pattern, P, into sub-patterns, $P = S_1||S_2||S_3 \dots ||S_s$. The overall idea of the method is to iteratively use the distances from $R_{j-1} = S_1||S_2|| \dots ||S_{j-1}$ to T to compute the distances from $R_j = S_1||S_2|| \dots ||S_{j-1}||S_j$ to T[1]. We refer to this process as 'feeding' the results from distances to R_{j-1} into the input of the computation of distances to R_j. The computation associated with R_j is termed level j. Note that level s computes distances against $R_s = P$ as required. This feeding of results ensures that optimal matches composed of sub-optimal sub-pattern matches are computed correctly. We motivate the Split and Feed method by considering a pseudo-realtime solution for the k-difference problem.

k-differences. The edit distance between two strings is the minimum number of moves required to transform P into T. We refer to this distance as $cost(P, T)$. The valid moves are *insert* (insert a character), *delete* (delete a character) and *mismatch* (replace a character). In the pattern matching case, we define an array Cost: at right alignment i, the distance $\text{Cost}(P, T)[i] = \min_{\ell<i} \text{cost}(P, T[\ell, i])$. The k-difference problem is to output the distance at all positions i where $\text{Cost}(P, T)[i] \leq k$, we call this a match. We also refer to a match of P as shorthand for a substring of T that P can be transformed into in $\leq k$ moves. Observe that both *insert* and *delete* operations are non-local and affect alignment of other characters.

A straightforward approach to solving this problem would split the pattern into halving lengths and consider each separately. However, even if a solution for the previously mentioned problems were found, there is an added difficulty we have to consider. Subpatterns much shorter than k still require text partitions that are $\Theta(k)$, increasing the overall time complexity of the algorithm. As a result we insist that the smallest subpattern has size larger than k. However we are now required to carry out extra work in order to find the distances that include this final subpattern. We assume throughout that $k \leq m/8$. If this is not the case then the direct dynamic programming solution runs in $O(m) \in O(k)$ time per character without modification.

Our solution splits the pattern into subpatterns of halving length $S_1, S_2, S_3 \dots S_s$ so that $P = S_1||S_2||S_3 \dots ||S_s$. In this case s is selected to be largest integer so that $|S_s| \geq 4k$. The final subpattern, S_s, is therefore of size $4k \leq |S_s| < 8k$. As discussed in the method overview, computation occurs in levels where level j computes the distances from $R_j = R_{j-1}||S_j$ to T using the distances from $R_{j-1} = S_1||S_2|| \dots ||S_{j-1}$ to T.

Level j can also be viewed as computing distances from S_j to T but incorporating a *starting cost array.* The starting cost array for level j gives the cost of beginning a match of S_j at each left alignment (remembering that the output is in terms of right alignments). We let the starting cost at left alignment i equal

[1] Where $R_1 = S_1$.

the distance from R_{j-1} to T at right alignment $(i-1)$. The distances computed from S_j to T using this starting cost array can be shown to equal the distances from R_j to T. Intuitively, this states that a match of S_j must be proceeded directly by a match of R_j.

A modification to the hybrid dynamic programming solution of [10] allows us to compute these distances offline in $O(|S_j|k)$ time if the portion of the text we are considering is of length $O(|S_j|)$. We will use this modified offline algorithm as a tool in our pseudo-realtime algorithm, distributing its work over the time taken for portions of the text to arrive. Details of the modification are left for the full version of the paper.

We are now able to describe the structure of the pseudo-realtime k-difference algorithm.

Feeding one level into another. For all levels $j < s$, we split T into partitions of length $7|S_j|/4$ with overlap $3|S_j|/2$. We will ensure that $\text{Cost}(R_j, T)[i]$ has been computed when $T[i+5|S_j|/8]$ arrives. As $k < 4|S_j|$, if a match of $R_j = R_{j-1}||S_j$ ends in $T[y-|S_j|/2+1, y]$ for some partition ending at y then the corresponding match of R_{j-1} must end in $T[y-7|S_j|/4+1, y-3|S_j|/4]$. This splits the partition into two disjoint sections marked in Figure 2 as *matches start* and *matches end.* To compute distances from R_j in the *matches end* section using the modified hybrid algorithm we only need the distances from R_{j-1} in the *matches start* section. Computing R_j distances in the *matches end* section is sufficient as the text partitions overlap.

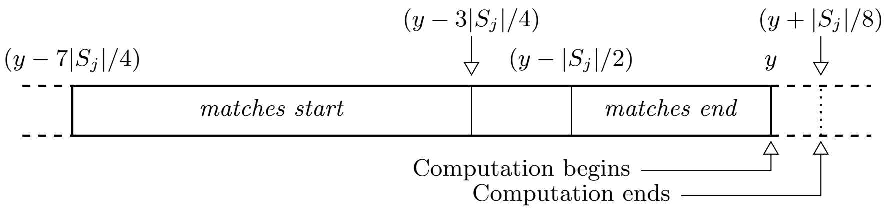


Fig. 2. The structure of a text partition for subpattern S_j ending at position y

Using $\text{Cost}(R_{(j-1)}, T)[y-7|S_j|/4+1, y-3|S_j|/4]$ as the starting cost array, text partition, $T[y-7|S_j|/4+1, y]$ and pattern S_j we can compute $\text{Cost}(R_j, T)[(y-|S_j|/2+1, y]$ in $O(|S_j|\,k)$ time (offline) using the modified hybrid method above[2].

We begin computation as soon as the last text character of the partition is received. However, we distribute the work over the time allocated to the next $|S_j|/8$ character arrivals. As we only process one partition at a time, we use $O(k)$ time per text character (per level). This computation requires that level $j-1$ has outputted $\text{Cost}(R_{(j-1)}, T)[y-7|S_j|/4+1, y-3|S_j|/4]$ before character y is received. In the worst case, a level j partition ending at y needs the output of the level $j-1$ partition ending at $y-|S_j|/2$ as shown in Figure 3. This partition will finish computing after $|S_{j-1}|/8 = |S_j|/4$ characters arrive which is

[2] For $j = 1$, we let $R_0 = \emptyset$ (the empty string) so that $\text{Cost}(R_0, T)[x] = 0$ for all x.

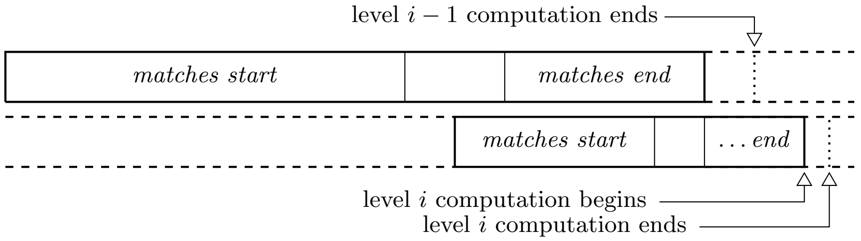

level $i-1$ computation ends
matches start
matches end
matches start
... end
level i computation begins
level i computation ends

Fig. 3. Alignment of a level $i-1$ partition (above) and a level i partition (below)

before character y is received. As a result, computation of the relevant section of $\text{Cost}(T, R_{(j-1)})$ completes before it is needed by level j.

The final level of computation. The aim of this stage is to produce the bottom $|S_s|+1$ rows of the dynamic programming table for the edit distance between P and T so that we can output $\text{Cost}(P,T)[i]$ when $T[i]$ arrives. Level $(s-1)$ provides us with the top row of this table in the form of $\text{Cost}(R_{(s-1)}, T)$. If a match of P ends at right alignment i, in the worst case, the corresponding match of prefix $R_{(s-1)}$ ends at position $i-|S_s|-k \le i-5|S_s|/4$, for example where there are k inserts into S_s. Therefore, we only need $\text{Cost}(R_{(s-1)}, T)[x]$ for $x < i - 5|S_s|/4$ to compute $\text{Cost}(P,T)[i]$ which we have before $T[i]$ arrives from level $s-1$. Use of this fact, coupled with careful work scheduling allows us to fill the dynamic table a constant number of columns per text character so that column i is filled as $T[i]$ is seen. $\text{Cost}(P,T)[i]$ is then the value of the bottom cell of column i. Each column is of height $O(S_s) \in O(k)$ as $|S_s| \le 8k$, so we perform $O(k)$ work per character as required.

Theorem 3. *The time complexity for the pseudo-realtime k-differences algorithm is $O(k \log m)$ per character. Each level requires $O(|S_j|)$ space, giving a total of $O(m)$ space.*

Combining Split and Feed and Split and Correct. The k-difference problem with transpositions allows an additional move, *transposition*; a restricted *swap* which can only occur before all other moves types. This problem can be solved offline in $O(kn)$ time by a simple modification of the method of Landau and Vishkin [10]. Although no single method we have discussed will convert this algorithm to be pseudo-realtime, by applying the Split and Feed and Split and Correct methods simultaneously an $O(k \log m)$ time per character algorithm results.

Acknowledgements. The authors would like to thank Benny and Ely Porat for many helpful discussions at an early stage of the work on this paper and Dave Arthur for his comments on the final draft.

References

[1] Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: rearrangement distances. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 1221–1229 (2006)

[2] Amir, A., Aumann, Y., Kapah, O., Levy, A., Porat, E.: Approximate string matching with address bit errors. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 118–129. Springer, Heidelberg (2008)

[3] Amir, A., Aumann, Y., Lewenstein, M., Porat, E.: Function matching. SIAM Journal on Computing 35(5), 1007–1022 (2006)

[4] Amir, A., Cole, R., Hariharan, R., Lewenstein, M., Porat, E.: Overlap matching. Inf. Comput. 181(1), 57–74 (2003)

[5] Amir, A., Eisenberg, E., Porat, E.: Swap and mismatch edit distance. Algorithmica 45(1), 109–120 (2006)

[6] Amir, A., Farach, M., Muthukrishnan, S.: Alphabet dependence in parameterized matching. Inf. Process. Lett. 49(3), 111–115 (1994)

[7] Clifford, P., Clifford, R.: Self-normalised distance with don't cares. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 63–70. Springer, Heidelberg (2007)

[8] Clifford, R., Efremenko, K., Porat, B., Porat, E.: A black box for online approximate pattern matching. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 143–151. Springer, Heidelberg (2008)

[9] Galil, Z.: String matching in real time. Journal of the ACM 28(1), 134–149 (1981)

[10] Landau, G.M., Vishkin, U.: Fast string matching with k differences. J. Comput. Syst. Sci. 37(1), 63–78 (1988)

[11] Levenshtein, I.V.: Binary codes capable of correcting deletions, insertions, and reversals. Cybernetics and Control Theory (1966)

[12] Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. 20(1), 18–31 (1980)

Faster and Space-Optimal Edit Distance "1" Dictionary

Djamal Belazzougui

Ecole Nationale Supérieure d'Informatique, Algiers, Algeria
Ineodev Company
djamel.belazzougui@ineodev.com

Abstract. In the approximate dictionary search problem we have to construct a data structure on a set of strings so that we can answer to queries of the kind: find all strings of the set that are similar (according to some string distance) to a given string. In this paper we propose the first data structure for approximate dictionary search that occupies optimal space (up to a constant factor) and able to answer an approximate query for edit distance "1" (report all strings of dictionary that are at edit distance at most "1" from query string) in time linear in the length of query string. Based on our new dictionary we propose a full-text index for approximate queries with edit distance "1" (report all positions of all sub-strings of the text that are at edit distance at most "1" from query string) answering to a query in time linear in the length of query string using space $O(n(\lg(n)\lg\lg(n))^2)$ in the worst case on a text of length n. Our index is the first index that answers queries in time linear in the length of query string while using space $O(n \cdot poly(log(n)))$ in the worst case and for any alphabet size.

1 Introduction

In this paper we are interested in solving the problem of approximate search queries on dictionaries of strings. In this problem, we are given an input string and we must report all the strings of the dictionary that are similar to that string in some specified string distance. In our paper we are interested in the most commonly used distance for this problem which is the *edit distance*. The edit distance between a string x and a string y is the minimal number of edit operations that must be applied to string x in order to obtain string y (or equivalently edit operations that permit to transform string x into string y). In the so-called *Levenstein distance* three operations are considered: deletion of a character, insertion of a character and substitution of a character by another. In the variant known as the *Damerau-Levenshtein distance* an additional operation is considered which is the swap of two consecutive characters. Another edit error that occurs frequently in queries involving multiple words consists in merging two consecutive words into a single one.

The main contribution of this paper is a dictionary construction that efficiently supports queries where we have to report all strings that are at edit distance at most “1” from a given query string. Our dictionary could handle all the five edition errors mentioned above. However for lack of space and ease of presentation, we will limit the presentation only to the three first errors.

As far as we know our solution is the first solution for this problem that answers to queries in time proportional to query string length while using space that is only a constant factor larger than space occupancy of a raw dictionary. Both query time and space usage of our dictionary are worst case bounds and do not include an extra factor depending on alphabet size.

Based on our dictionary for edit distance “1”, we build an index for approximate search on a text of length n with edit distance “1”. The index occupies space $O(n(\lg(n)\lg\lg(n))^2)$ and answers queries in time proportional to query string length.

1.1 Preliminaries

Our model is the RAM model with word size w. We assume that pointers to manipulated elements (manipulated elements can be strings, characters or integers) fit in a machine word. That is if N is the number of manipulated elements, then $\lg(N) < w$. We also assume that standard operations like shifts, division and multiplication all take constant time.

In our problem, we have a collection S of n strings of characters (elements of S are sometimes referred as keys) where each character occupies b bits and we assume that $b = O(w)$ (we note the size of alphabet by α where $\alpha = 2^b$). This assumption is made so that we can manipulate a character in $O(1)$ time. However character length can be as small as 1 bit. The total number of characters in all strings is denoted by m.

1.2 Our Results

Our main result is the following theorem which is proved in section 3.

Theorem 1. *For any set S of n strings of total length m characters where each character is of length b bits, we can construct a dictionary occupying space $O(mb)$ bits and supporting approximate queries that return all strings of S that are at edit distance at most “1” from a string p in time $O(k + occ)$ where $k = |p|$ and occ is number of reported strings. Expected construction time of our dictionary is $O(m)$. Additionally our dictionary supports the counting of number of strings at distance at most “1” in time $O(k)$.*

The following theorem is proved in section 4.

Theorem 2. *For any text of n characters each of length b bits, we can build an index occupying space $O(n(\lg(n)\lg\lg(n))^2 b)$ bits and supporting queries that return all positions of all sub-strings of the text that are at edit distance at most “1” from a string p in time $O(k + occ)$ where $k = |p|$ and occ is the number of reported strings.*

1.3 Related Work

The problem of approximate string matching has been extensively studied. We only describe most significant results with close to optimal query times. For the approximate dictionary search over edit distance "1", we mention the work in paper [4] which proposes two solutions to solve the problem of approximate dictionary search over hamming distance with one error. The first solution solves the problem only over binary alphabet (alphabet of length 1 bit) in time $O(|p| + occ)$ while using space $O(m)$ words. The solution consists essentially in using a compacted trie that stores all strings that are at hamming distance "1" from the strings of the original dictionary. While this solution answers to queries in the same time as our solution, it has no known construction algorithm that runs in reasonable time. When extended naively to alphabets of size α, query time will remain the same, but the space usage becomes $O(m\alpha)$ words which will be prohibitive for very large alphabets. The second solution of [4] answers to queries in time $O(|p|+occ+\lg(n))$ for binary alphabets. This solution has some similarity with our work in that it uses two tries to solve the problem. Similarly to the first solution, the second solution presented in [4] deals only with hamming distance, but it can easily be extended to work with edit distance. Its space usage is $O(m)$ words while we use only $O(mb)$ bits. If b is constant, space usage of that solution will thus be a factor $\lg(n)$ away from optimal. Additionally the second solution presented in [4] answers to queries in time $O(|p| + occ + \lg(n))$ only for binary alphabet (alphabet $\{0, 1\}$). For non binary alphabets of size $2^b = \alpha$, the query time becomes $O(|p| \cdot b + occ + \lg(n))$. Peter Brass suggests in [3, section 8.2], the use of hashing instead of the binary search used to answer queries in the second solution of [4]. With this modification, query time becomes $O(|p|\alpha + occ)$ which is the same as our solution for constant time alphabets, but prohibitive for large alphabets.

Another solution for binary alphabet is presented in [5]. The solution presented there deals also with Hamming distance but strings have fixed length L. Query time of the solution is better than ours in that it is only $O(L/w)$. However unlike our solution, the space usage is not optimal as it uses $O(Ln\lg(L))$ bits supposing that all strings are of same length L. By contrast our solution uses only $O(Ln)$ bits of space on a binary alphabet. Moreover the solution in [5] can only return a single matching string from the dictionary while our solution returns all matching strings.

Another way to solve approximate dictionary search problem is through the use of solutions for the more general problem of approximate full-text indexing. All known solutions for approximate full-text indexing for edit distance "1" incur at least a factor $\Omega(\lg(n))$ in space usage and/or an additive $\Omega(lg(n))$ term in query time. We can conclude that those solutions are not competitive with our solution for approximate dictionary search problem.

We now compare those solutions with our solution for full-text indexing problem with edit distance "1". The only solution (we are aware about) for that problem with query time $O(|p|+occ)$ that uses reasonable space is the solution described in [11] which uses $O(n(\alpha\lg(n)))$ words of space which for small alphabets is lower than space used by our solution which is $O(n(\lg(n)\lg\lg(n))^2 b)$ bits. However in solution of [11] either the query time or the space bound hold only on average. By contrast our bounds are both worst case bounds.

2 Tools

We now present the tools that will be used for our construction.

2.1 Minimal Perfect Hashing

Given a set S of n elements, a minimal perfect hash function is a function that maps every element of the set S to a distinct number in interval $[0, n-1]$. It is a well known result that a minimal perfect hash function occupying $O(n)$ bits can be constructed in $O(n)$ expected time so that a query takes time $O(1)$. Examples of optimal theoretical and practical solutions are presented in [10] and [2].

2.2 Exact String Dictionary with Minimal Perfect Hashing

Given a set S of n strings of total length m, using minimal perfect hashing we can construct a dictionary (for exact lookups) that uses optimal space and supports exact lookups in time proportional to the length of input string. That is, the exact dictionary occupies $O(mb)$ bits of space and an exact lookup for a string of length k takes time $O(k)$. The first step of our construction consists in building a minimal perfect hash function on the set S. This minimal perfect hash function (which we note by *mphf*) will map each string of S to a distinct number in interval $[0, n-1]$ (usually construction of *mphf* involves a preliminary step that consists in the use of a hash function that maps the strings to integers of $O(w)$ bits). We store the strings one after another in a consecutive array T in the order given by the *mphf*. If we note by s_i the string mapped to the number i by the *mphf*, then the first character of string s_i in T will be given by the formulae

$$Position[s_i] = \sum_{j=0}^{j<i} |s_j|$$

The space used by the array T is mb bits. In order to recover the position of a string in T, we use a *prefix-sum* data structure that will give for any i the sum of lengths of all strings mapped to numbers below i by the *mphf*. The data structure we use is the one described in papers [8] and [9] which will use space $n(\lceil\lg(m/n)\rceil + 2 + o(1)) = O(mb)$ bits and will permit to recover position of string s_i in constant time.

2.3 Retrieval Lists Dictionary

We have a set $S' = \{x_1, x_2, \ldots, x_{n'}\}$ of n' keys. We also have a collection of non empty lists $L = \{l_1, l_2, \ldots, l_{n'}\}$ where each l_i is a list of elements that we associate with the key x_i from S'. The total number of elements stored in the lists is m' and each element is of length b bits. We wish to construct a data structure that uses space $O(m'b)$ bits while supporting the following operations in constant time:

- *list_element*(x, i). This operation returns element number i (elements of the lists are numbered from 1) of the list associated with key x if $x \in S'$. The returned element is undefined If $x \notin S'$.
- *size_of*(x). This operation returns number of elements in the list associated with x if $x \in S'$. The result of operation is undefined if $x \notin S'$.

We call this problem by *retrieval lists dictionary* problem as we aim to retrieve elements of a list associated with a key x without the ability to check whether $x \in S'$. We will give a simple construction for the *retrieval lists dictionary* occupying $O(m'b)$ bits of space. This solution is based on prefix sum data structure and perfect hashing and supports the two operations in constant time. The solution has some similarity to the solution for exact string dictionary described in subsection 2.2. The first step is to construct a minimal perfect hash function on the set S'. We note by x_i the element of S' mapped by *mphf* to position i. We store elements of each list contiguously in an array T'. The position of element number j of the list associated with key x_i will be given by

$$Position(i, j) = (\sum_{k=0}^{k<i} |x_k|) + j$$

The space used by the array T' is of course $m'b$ bits. Similarly to 2.2, we also use a *prefix-sum* data structure in order to recover the position of the first element of each list in the array T'. This *prefix-sum* data structure will thus use space $n'(\lceil \lg(m'/n') \rceil + 2 + o(1))$ bits. The total space is thus $O(m'b)$. The traversal of the list associated with a key q will begin by querying the *mphf* which will give a number i. By querying the *prefix-sum* data structure we will obtain a position p_i. The element number j of the list associated with q will thus be located at position $T[p_i + j]$. To get the number of elements in the list associated with key q, we simply query *prefix-sum* data structure with the numbers i and $i+1$ giving the numbers p_i and p_{i+1}. The number of elements in the list associated with q will simply be $p_{i+1} - p_i$.

2.4 Succinctly Encoded Tries

We now describe another standard tool that will be used for our solution. Given a collection of strings S of total length m characters (mb bits), we wish to construct a trie on S that occupies space $O(m)$ characters ($O(mb)$ bits) and that can be traversed in time $O(|p|)$ for a given pattern p. For a query on a pattern p we require that the trie returns a unique integer identifier for each node traversed during the lookup for the pattern $|p|$. This identifier must occupy $O(w)$ bits of space. This problem can be solved through the use of a data structure described in [1]. The solution for cardinal trees presented in [1] is perfectly suitable for our purpose as our trie can be considered as a cardinal tree. The space usage and traversal time for the trie from solution in [1] are both optimal. Additionally the data structure is able to return a unique integer in the range $[1, c]$ for each traversed node where c is the total number of nodes. We note that the numbers in the range $[1, c]$ fit in $O(w)$ bits (This is implied by the fact that $c < m$ and $\lg(m) = O(w)$).

3 Data Structure

We are given a collection S of n strings (keys) of total length m characters. We note the query string by q and we note its length by $k = |q|$. The straightforward solution to answer approximate queries on a string q is to query exhaustively for all strings that can be obtained by one edit operation on the string q. We notice that we have k candidate strings that can be obtained by deleting one character from q. We also have $k\alpha$ and $(k+1)\alpha$ strings that can be obtained through substitutions and insertions respectively. If an exact dictionary were used we would spend time $O(k)$ time to answer each query and thus a total time $O(k^2\alpha)$ to answer queries for all candidate strings.

The first trick that we use to reduce query time consists in reducing the number of candidate strings for insertions and substitution from $O(k\alpha)$ to $O(k)$, by using a retrieval lists dictionary that gives us a candidate character for each possible position of insertion or substitution in the string q.

Now that we have reduced number of candidate strings to just $O(k)$, we will exploit the fact that candidate strings are only slightly different from string q in order to get query time $O(k)$. Our approach will consist in doing a preprocessing step that involves the string q, taking time $O(k)$, after which, querying for each candidate string will take time $O(1)$. Checking for each candidate string involves two steps: computing a hash value that points to a string s of the dictionary and comparing that pointed string with candidate string. We will see that using standard string hash functions, the computation of the hash value of each candidate string can be done in time $O(1)$ using some arrays that have been initially computed on string q. What remains is to improve query time of the second step which is to compare candidate string to the string s located (by the hash value) in the dictionary. One potential solution would be to use string hash functions that produce long hash values and store the hash value associated with each string of dictionary. Then a comparison of candidate string with string s would be reduced to the comparison of their respective hash values. However this would only work with high probability as there is still some probability that the candidate string and the string s have the same hash value without being equal. The idea we use to improve comparison time in a deterministic way relies on the observation that we can compare two blocks of u characters where $ub = O(w)$ in just $O(1)$ time. This implies that strings of less than u characters can be compared in time $O(1)$. What remains is thus to improve comparison time for long strings. Our solution for that is to compute signatures of all prefixes and suffixes of strings in the dictionary whose lengths are multiple of u. Those signatures will occupy less than w bits, and thus total space used by signatures will be of the same order as the space occupied by the strings. The comparison of a candidate string will involve comparison of signatures of a prefix and a suffix of q with signatures of a prefix and a suffix of s along with a constant number of comparisons involving blocks of less than $2u$ characters. In order to get deterministic signatures for prefixes and suffixes, we will use a trie and a reverse tries built on the set of strings in the dictionary, which will be able to return deterministic signatures for any prefix and suffix of every string in the dictionary.

Our data structure will use the following components:

- A trie Tr built on the set S using data structure from [1]. The main role of this trie is to permit to compare prefixes of keys in S with prefix of any string q just by comparing the identifiers returned by the trie for the two strings. This is the essential part of our algorithm that permits to achieve a $O(1)$ query time bound for each candidate string.
- A trie $\overline{Tr}$ built on the set $\overline{S}$ using data structure from [1]. The set $\overline{S}$ is the set of all strings of S written in reversed order. The main role of this trie is to permit to compare suffixes of keys in S with suffix of any string q just by comparing the identifiers returned by the trie for the two strings.
- A *retrieval lists dictionary* constructed using the data structure described in 2.3. This *retrieval lists dictionary* will associate a list of characters with pairs of integers (l, r) both in the range $[0, m]$.
- A dictionary based on minimal perfect hashing constructed according to general ideas described in subsection 2.2. The details of the construction of this dictionary will be given in next subsection.

3.1 Construction

Construction of the tries. The construction of the trie Tr and reversed trie $\overline{Tr}$ will both take time $O(m)$. The two tries will both use $O(mb)$ bits which is optimal. We note that traversing the trie or the reverse trie for a pattern p will return at most $|p|$ identifiers in the range $[1, m]$ (the maximal number of nodes in the trie is m) corresponding to the traversed nodes.

Construction of the lists dictionary. We now turn out to the construction of the *retrieval lists dictionary*. We process successively every string s of S. Let k be the length of string s. We will store two temporary arrays $L[0..k]$ and $R[0..k]$ of integers in the range $[0, m]$. We set $L[0] = R[0] = 0$. We first traverse the trie Tr for the string s. The traversal will take time $O(k)$. During the traversal of the trie, we store in the cell $L[i]$ the identifier of the node reached at step i (steps are numbered from one). We do a symmetric processing to generate elements $R[1..k]$. We traverse the reverse trie $\overline{Tr}$ for the string $\overline{s}$ (the reverse of string s) and then we store the identifier of the node reached at step i in the cell $R[i]$. The construction of the *retrieval lists dictionary* will proceed in the following way: for each string s of length k for which we have computed arrays L and R we add the character $s[i]$ to the list corresponding to the pair $(L[i-1], R[k-i])$ for each i such that $1 \leq i \leq k$. In fact we insert elements in a lists dictionary implemented using a temporary dynamic hash table (implemented using some dynamic hashing technique that supports insertions in constant expected time) and linked lists. The temporary storage of the dynamic hash table and associated linked lists will be $O(m)$ words of space. Later we can construct our *retrieval lists dictionary* from the temporary hash table and linked lists using the method described in 2.3 which will take expected $O(m)$ time. The space occupied by the *retrieval lists dictionary* will then be optimal $O(mb)$ bits.

Construction of perfect hashing based dictionary. We finally turn out to the implementation of perfect hashing based dictionary. The first step in our construction is to map our set S of n strings to distinct hash values in the range $[0, P-1]$ for a prime P such that $P > mn^2$ and $P > 2^b$. To that end we will use a hash function h parametrized with a randomly chosen integer t such that $t \in [0, P-1]$. For a given string s of S of length k, we compute the hash value associated with s using the following formulae: $h(s) = (s[1] \otimes t) \oplus (s[2] \otimes t^2) \oplus \cdots \oplus (s[k] \otimes t^k)$ where additions and multiplications are done modulo P (characters of string s are considered as integers in the range $[0, P-1]$). Computation of hash values for all strings will thus take time $O(m)$ (we can compute each hash value in optimal time using Horner scheme). Once we have computed the hash values associated with all strings of S, we check whether we have collisions between the hash values generated for the keys of the set S (we have at least one pair of strings with the same hash value) which will happen with probability at most $1/2$ (see appendix A for justification). If this is the case, we repeat computation of the set of hash values associated with keys of the set S using new randomly chosen value for t until we have no collision. The probability that a given t maps keys without collisions is at last $1/2$, which means that we will have to do an expected $O(1)$ attempts before succeeding. The expected total time for generating the hash values without collisions will thus be $O(m)$ as each attempt takes $O(m)$ time and we do expected $O(1)$ attempts. Once we have mapped all strings of S to distinct numbers, we will use those numbers as keys to build our *mphf*.

The final detail that we have to deal about is how to store our strings in the dictionary. We will deal differently with short and long strings (the reason for this distinction will become clear in next subsection). Short strings (strings of length $\leq w$) will be stored unmodified in the dictionary. For a long string s of length $k > w$ we will store a modified string s' of length $3k$ characters ($3kb$ bits). We divide the string s' into three consecutive parts s'_1,s'_l and s'_r each of length kb bits. The string s'_1 will contain a copy of the string s. We now describe how we set the strings s'_l and s'_r. To that end we first set a value $u = \lceil \frac{\lg(m)}{b} \rceil$. We consider the strings s'_l and s'_r as arrays of $k' = \lfloor k/u \rfloor$ elements of ub bits each (note that $\lg(m) \leq ub < \lg(m) + b$) ignoring the padding bits (the last $kb - k'ub$ bits). We set $s'_l[i]$ (elements of s'_l and s'_r are numbered from 1) to the value $L[ui]$ (the identifier of the node reached at the step ui when traversing the trie Tr for the string s) and we set $s'_r[i]$ to the value $R[ui]$ (the identifier of the node reached at the step ui when traversing the trie $\overline{Tr}$ for the string $\overline{s}$).

Of course we will store our strings (both short unmodified and long modified strings) in a contiguous array in the order given by the *mphf*, and we use *prefix-sum* data structure to store locations of each string.

Satellite data. In many applications we may need to associate a satellite data with each string of the dictionary. This can easily be done in our case, simply by adding a table of n cells indexed by *mphf* used for the dictionary.

3.2 Queries

We are given on input a string q of length k. We first compute the string $\overline{q}$ which is the reverse of string q. Then we compute the arrays $L[0..k]$ and $R[0..k]$. We first set $L[0] = R[0] = 0$. We traverse the trie Tr for the string q and we put in cell $L[i]$ the identifier of the node reached at step i. If the search stops at step i we will set to the special value $\perp$ every cell $L[j]$ for $j \in [i+1, k]$. Likewise we set cells of array R by traversing the trie $\overline{Tr}$ for the string $\overline{q}$ and putting in cell $R[i]$ the identifier of the node reached at step i. If the search stops at step i we will set to the special value $\perp$ every cell $R[j]$ for $j \in [i+1, k]$.

We also compute three arrays noted by $A_t[0, k+1]$, $F[0..k]$ and $G[1..k+1]$:

1. The array A_t will store all powers of t up to t^{k+1}. We first set $A_t[0] = 1$. Then we set $A_t[i] = A_t[i-1] \otimes t$ for each i in interval $[1, k+1]$.
2. To generate F we first set $F[0] = 0$. Then we set $F[i] = F[i-1] \oplus (q[i] \otimes A_t[i])$ for each i in the interval $[1..k]$.
3. To generate G we first set $G[k+1] = 0$. Then we set $G[i] = (G[i+1] \oplus q[i]) \otimes t$ for each i in the interval $[1..k]$.

It can easily be seen that time required for computing each of the five arrays is $O(k)$. We have four kinds of strings in dictionary that could match a query string: a string that is at distance "0" (exact match) and strings of the dictionary that can be obtained with just one of the three errors. We only describe matching of strings obtained by one kind of error (insertion). The remaining types of errors which are quite similar to the one we describe here are deferred to the appendix:

1. String at edit distance "0". In this case we simply probe the dictionary for the string q. We know that $h(q) = (q[1] \otimes A_t[1]) \oplus \cdots \oplus (q[k] \otimes A_t[k]) = F[k] = G[1]$. Probing the dictionary will return a string s'. If it is a short string (s' is a short string if its length is less than w) we set s to to s', otherwise (s' is a long string and it length is at least $3w$) we set s to be the first $|s'|/3$ bits of s'. Now we return the string s as a match if and only if we find that $s = q'$. Comparison between s and q' takes $O(k)$ time.
2. Strings obtained by inserting one character in q. We will do $k+1$ steps for $i \in [0, k]$. At each step i we probe the retrieval lists dictionary for the list associated with pair $(L[i], R[k-i])$ if and only if $L[i] \neq \perp$ and $R[k-i] \neq \perp$ (we suppose we are inserting a character after position i). To check for validity of returned list, we only need to check for its first element. If first element is valid we conclude that the list really exists and that all remaining elements

are also valid. Let c be the first element of the list associated with pair $(L[i], R[k-i])$(c is of course a character). We should now probe the dictionary for the string $q' = q[1..i]cq[i+1..k]$. We have $h(q') = F[i] \oplus (c \oplus G[i+1]) \otimes A_t[i+1]$. As previously we use $h(q')$ in order to query the *mphf*. Then we use the position returned by the *mphf* to probe the dictionary. If the returned string s' is a short string, we can directly compare it with q' in $O(1)$ time and return a match in case they are equal (this comparison takes $O(1)$ time as the two strings are of length $O(w)$). If it is a long string, we divide s' into three equal parts s'_1, s'_l and s'_r. What we need now is to be able to compare strings s'_1 and q'. However we can not compare s'_1 and q' in $O(1)$ time as they do not fit in a constant number of words. In order to compare s'_1 and q' in constant time we will use the strings s'_l and s'_r which are considered as vectors of elements of length $ub \geq \lg(m)$ bits each. We set the two variables $l_{q'} = \lfloor \frac{i}{u} \rfloor$ and $r_{q'} = \lfloor \frac{k-i}{u} \rfloor$. We will return a match if and only if the following conditions are all met:

- Length of s' is $3(k+1)$.
- $l_{q'} = 0 \vee s'_l[l_{q'}] = L[l_{q'}]$. This test is used to check whether the first $u \cdot l_{q'}$ characters of s'_1 and q are the same. This test takes $O(1)$ time. It can easily be seen that $s'_l[l_{q'}] = L[l_{q'}]$ will hold if and only if the identifier returned from the trie Tr is the same for the first $u \cdot l_{q'}$ characters of s'_1 and the first $u \cdot l_{q'}$ characters of q which can only happen if those characters are the same.
- $q[u \cdot l_{q'} + 1..i] = s'_1[u \cdot l_{q'} + 1..i]$. This test can be done in $O(1)$ time as we are comparing two strings of length at most $(u-1)b = O(w)$ bits.
- $s'_1[i+1] = c$. This test clearly takes $O(1)$ time.
- $q[i+1..k - u \cdot r_{q'}] = s'_1[i+2..k+1-u \cdot r_{q'}]$. This test takes $O(1)$ time as we are comparing two strings of length $O(w)$ bits.
- $r_{q'} = 0 \vee s'_r[r_{q'}] = R[r_{q'}]$. This test is used to check whether the last $u \cdot r_{q'}$ characters of s'_1 and q are the same. This test takes $O(1)$ time. It can easily be seen that $s'_r[r_{q'}] = R[r_{q'}]$ will hold if and only if the identifier returned from the trie $\overline{Tr}$ is the same for the last $u \cdot r_{q'}$ characters of s'_1 and the last $u \cdot r_{q'}$ characters of q which can only happen if those characters are identical.

We emphasize that at each step i, we need only to check for the string obtained by inserting at position i the first character of the list returned by retrieval lists dictionary. If we have a match, we can continue to retrieve strings obtained by inserting remaining characters from the list at position i and we are sure to have a match for those obtained strings. That is because the fact that the first returned character was valid means that there exists a list associated with the pair $(L[i], R[k-i])$ and so all elements of the list are valid. Reporting each additional element of the list takes $O(1)$ time.

Checking for the two remaining types of errors will also take $O(k)$ time. The procedure for checking those errors is similar to the procedure for checking for insertion. For completeness the procedure for checking for those two kinds of errors is described in appendix B. Reporting valid strings for each error takes additional $O(1)$ time per element. Thus the total query time is $O(k + occ)$.

Counting queries. Counting the number of matching occurrences can be done in time $O(k)$. This is done by summing up the count of matching strings from each kind of error. For deletions, we can have at most k matching strings and checking for each candidate string takes $O(1)$ time. For insertions and substitutions we need to do $k+1$ and k steps. At each step we need only to check first element of a list returned by retrieval lists dictionary and if this element is a matching element, we add the size of the list (obtained using *size_of* operation) to the total count of matching strings.

4 Approximate Full-Text Indexing with Edit Distance "1"

In the approximate full-text indexing problem, we have to build a data structure on a text of length n so that we can answer to queries of the kind: find all locations in the text of strings that are at prescribed distance from a string q. Our data structure for approximate dictionary lookups for edit distance "1" can be used to get an improved solution for full-text indexing with edit distance "1". For that we combine our approximate dictionary with solution from [6]. We construct the index of [6] which uses $O(n \lg(n))$ words of space (which is $O(n \lg(n)^2)$ bits as we have $w = \Theta(\lg(n))$ with query time $O(|q| + \lg(n) \lg\lg(n) + occ)$. We select all sub-strings of the text of lengths up to $\lg(n) \lg\lg(n)$ characters, and store them in our approximate dictionary which will thus store at most $n(\lg(n) \lg\lg(n))$ strings. Strings that appear more than once in the text are stored only once in the dictionary. We associate with each string (as satellite data) in the dictionary a pointer to a vector that stores all locations where string appears in the text. Each vector pointer occupies $\lg(n)$ bits and each string occupies at most $\lg(n) \lg\lg(n)$ characters making a total space usage of $O(n(\lg(n) \lg\lg(n))^2 b)$ bits for the dictionary. We can pack vectors of locations of strings into a single contiguous array of $O(n \lg(n) \lg\lg(n))$ pointers occupying a total $O(n \lg(n)^2 \lg\lg(n))$ bits of space. Summing up space usage of all components of our data structure we get a total space usage of $O(n(\lg(n) \lg\lg(n))^2 b)$ bits of space.

To make a query for a string q, we simply check whether $|q| > \lg(n) \lg\lg(n)$ in which case we use data structure from [6] to solve the query. Otherwise, we use our dictionary to solve the query.

5 Concluding Remarks

We have described a dictionary that answers approximate queries with edit distance "1", in time proportional to length of query strings, while using space that is only a constant factor away from the raw dictionary. The space usage of our dictionary is optimal up to a constant factor, but the query time is only optimal for alphabets of lengths $\Theta(w)$ bits. Ideally query time should be proportional to length of query string expressed in terms of memory words and not in terms of number of characters. Thus, in the case of constant sized alphabets query time of our dictionary is a factor $\Theta(w)$ larger than a hypothetical optimal.

We can cite few open questions for future work:

– Is it possible to devise a data structure that runs in time $O(k/B)$ in the cache oblivious model or in the I/O model where B is the block size. A direct adaptation of our data structure would give queries that run in time $\Theta(k)$. This is the consequence of the fact that a query in our dictionary uses $\Theta(k)$ non contiguous memory probes.
– Is it possible to devise a dynamic version of our data structure with reasonable update times.
– Is our query time the best possible for data structures that use optimal space.

We now briefly discuss the practicality of our dictionary. The space usage of our dictionary is upper bounded by $6mb + O(m)$ bits. We remark that the trie, reverse trie and the arrays s'_l and s'_r computed for each long string s are only used in order to permit checking candidate strings in time $O(1)$. However directly comparing candidate strings with strings of the dictionary would involve access to characters that are stored contiguously in memory. This implies that such comparisons could be done quickly on modern architectures provided that strings are not too long. This is the case because on modern architectures, execution time is often dominated by number of probes to non consecutive memory locations. We can conclude from that,that unless strings are extremely long, we can eliminate the trie, reverse trie and the arrays s'_l and s'_r, reducing space to about $2mb+O(m)$ bits. This reduces our data structure to just two elements: an exact dictionary based on minimal perfect hashing and retrieval lists dictionary. We remark that the lists dictionary will only be useful for large alphabets as for very short alphabets, we have so few candidates for insertions and substitutions that using the lists dictionary will not asymptotically reduce query time. We can conclude from that, that the dictionary presented in this paper is mostly useful for strings over relatively large alphabets like natural languages or ASCII alphabets.

Acknowledgements

The author is grateful Gonzalo Navarro for his valuable comments on a previous draft and for helpful discussion. The author is also grateful to Martin Dietzfelbinger for showing him the solution for hash functions on strings and to Gerth Brodal for valuable comments on a previous draft.

References

1. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
2. Botelho, F.C., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 139–150. Springer, Heidelberg (2007)
3. Brass, P.: Advandced Data structures. Cambridge University Press, Cambridge (2008)

4. Brodal, G.S., Gąsieniec, L.: Approximate dictionary queries. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 65–74. Springer, Heidelberg (1996)
5. Brodal, G.S., Srinivasan, V.: Improved bounds for dictionary look-up with one error. Information Processing Letters 75(1-2), 57–59 (2000)
6. Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC, pp. 91–100 (2004)
7. Dietzfelbinger, M., Gil, J., Matias, Y., Pippenger, N.: Polynomial hash functions are reliable (extended abstract). In: ICALP, pp. 235–246 (1992)
8. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM 21(2), 246–260 (1974)
9. Fano, R.M.: On the number of bits required to implement an associative memory. In: Memorandum 61, Computer Structures Group, Project MAC, MIT Press, Cambridge (1971)
10. Hagerup, T., Tholey, T.: Efficient minimal perfect hashing in nearly minimal space. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 317–326. Springer, Heidelberg (2001)
11. Maaß, M.G., Nowak, J.: Text indexing with errors. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 21–32. Springer, Heidelberg (2005)

A Hash Function for Strings

We now give a brief analysis of the string hash function used for our dictionary. The reader can refer to [7] for more detailed analysis. The total length of the n strings of our collection is m, which means that the length of the longest string is at most $m-n+1$. The computation of our hash function uses multiplications and additions modulo P, where P is a prime number chosen such that $P > mn^2$ and $P > \alpha$. For a string s of S of length k, the hash function is computed by the formulae $h(s) = (s[1] \otimes t) \oplus (s[2] \otimes t^2) \oplus \cdots \oplus (s[k] \otimes t^k)$, where t is a number randomly chosen from interval $[0, P-1]$. For the purpose of hash function computation, we can consider our strings as polynomials of degree $d = m-n+1$ over a finite field. It is well known that polynomials over finite field of degree d can have at most d roots. Thus two distinct strings can hash to the same value for at most $d = m-n+1$ choices of t. Our set has n distinct strings which means that we have $n(n-1)/2$ pairs of distinct strings and each pair of strings hash values collide for at most $m-n+1$ choices of t. Thus we have at most $n(n-1)(m-n+1)/2$ choices of t that generate at least one collision. So the probability of having at least one collision for a t randomly chosen from $[0, P-1]$ is at most $\frac{n(n-1)(m-n+1)/2}{P} < \frac{n(n-1)(m-n+1)/2}{mn^2} < 1/2$.

B Dictionary Queries for Deletion and Substitution

For completeness, we describe in this section how the dictionary is checked for strings resulting from one deletion or one substitution:

1. Strings obtained by deleting one character from q. In this case we will try to probe the dictionary with every possible string that can be obtained by deleting one of the characters of q. We have exactly k such strings which means that we need to spend only $O(1)$ time for each probe to check whether there is a match or not. More concretely if we wish to probe the dictionary for string q' obtained by deleting character number i from q ($q' = q[1..i-1]q[i+1..k]$), we do the following steps: we first check that $L[i-1] \neq \perp$ and $R[k-i] \neq \perp$. If this is not true, we immediately conclude that we do not have a match for the string q'. Otherwise we compute the hash value $h(q') = F[i-1] \oplus G[i+1] \otimes A_t[i-1]$ in constant time. We can now query the *mphf* using the hash value $h(q')$. Using the position returned by the *mphf*, we query the dictionary which will return a string s'. If it is a short string we can directly compare s' and q' in $O(1)$ time (before comparing the strings we compare their lengths) and return a match in case they are equal. If s' is a long string, we divide the string s' into three equal parts s'_1,s'_l and s'_r. What we need now is to compare s'_1 with q' and return a match if $s'_1 = q'$ (recall that s'_1 is a copy of a string of S). We set $l_{q'} = \lfloor \frac{i-1}{u} \rfloor$ and $r_{q'} = \lfloor \frac{k-i}{u} \rfloor$. We will return a match if and only if the following conditions are all met:
   - Length of s' is $3(k-1)$.
   - $l_{q'} = 0 \vee s'_l[l_{q'}] = L[l_{q'}]$.
   - $q[u \cdot l_{q'} + 1..i-1] = s'_1[u \cdot l_{q'} + 1..i-1]$.
   - $q[i+1..k - u \cdot r_{q'}] = s'_1[i..k-1-u \cdot r_{q'}]$.
   - $r_{q'} = 0 \vee s'_r[r_{q'}] = R[r_{q'}]$.

   It is clear that testing for the candidate string for each i takes $O(1)$ time.
2. Strings obtained by substituting one of the characters of q. The case of substitution is very similar to that of insertion. First, we check the *retrieval lists dictionary* for each pair $(L[i-1], R[k-i])$ for $i \in [1,k]$ at the condition that $L[i-1] \neq \perp$ and $R[k-i] \neq \perp$. Let c be the first element of the list associated with $(L[i-1], R[k-i])$. We let $q' = q[1..i-1]cq[i+1..k]$. We have $h(q') = F[i-1] \oplus (c \oplus G[i+1]) \otimes A_t[i]$. We now probe the dictionary using $h(q')$. If the returned string s' is a short one, we directly compare it with q' in $O(1)$. Otherwise we decompose s' into three equal parts s'_1,s'_l and s'_r. We now need to compare s'_1 with q'. For that, we first set the two variables $l_{q'} = \lfloor \frac{i-1}{u} \rfloor$ and $r_{q'} = \lfloor \frac{k-i}{u} \rfloor$. We will return a match if and only if the following conditions are all met:
   - Length of s' is $3k$.
   - $l_{q'} = 0 \vee s'_l[l_{q'}] = L[l_{q'}]$.
   - $q[u \cdot l_{q'} + 1..i-1] = s'_1[u \cdot l_{q'} + 1..i-1]$.
   - $s'_1[i] = c$.
   - $q[i+1..k - u \cdot r_{q'}] = s'_1[i+1..k - u \cdot r_{q'}]$.
   - $r_{q'} = 0 \vee s'_r[r_{q'}] = R[r_{q'}]$.

   It is clear that checking every condition takes time $O(1)$. Similarly to the case of insertions, we need to do the check only for first element of the list. If we do not have a match for the first element, we conclude that the list does not exist and we stop immediately. Otherwise, we report the first element and continue to traverse remaining elements of the list without checking in $O(1)$ time per element and we report each element as a match.

Approximate Matching for Run-Length Encoded Strings Is 3SUM-Hard

Kuan-Yu Chen[1], Ping-Hui Hsu[1], and Kun-Mao Chao[1,2,3]

[1] Department of Computer Science and Information Engineering
[2] Graduate Institute of Biomedical Electronics and Bioinformatics
[3] Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106
kmchao@csie.ntu.edu.tw

Abstract. In this paper, we consider a commonly used compression scheme called run-length encoding (abbreviated RLE). We provide lower bounds for problems of approximately matching two RLE strings. Specifically, we show that the wildcard matching and k-mismatches problems for RLE strings are 3SUM-hard. For two RLE strings of m and n runs, such a result implies that it is very unlikely to devise an $o(mn)$-time algorithm for either problem. We then propose an $O(mn + p \log m)$-time sweep-line algorithm for their combined problem, i.e. wildcard matching with mismatches, where $p \leq mn$ is the number of matched or mismatched runs. Furthermore, the problem of aligning two RLE strings is also shown to be 3SUM-hard.

Keywords: run-length encoding, 3SUM, wildcard matching, k-mismatches, alignment.

1 Introduction

Run-length encoding (RLE) is a very simple and efficient coding scheme of data compression. It compresses a string into several runs, each run consisting of identical symbols. A run can be represented by a pair, usually denoted by σ^i, where σ is an alphabet symbol and i is its repetition times. For example, string *bbcccddaaaaa* can be compressed into the RLE format as $b^2c^3d^2a^5$. The most well-known application of RLE is perhaps the fax transmission since most faxed documents are composed of a great portion of white space and a small portion of black space [23]. This coding is also widely used in optical character recognition and image compression where the contents usually contain large scales of identically valued pixels [9]. In 1992, Amir and Benson [2] considered a new paradigm of accessing the compressed data directly without decompression. In particular, they considered the pattern matching problem in RLE texts. Since then this has been an active research field, and several papers have been devoted to designing efficient algorithms for problems of matching or comparing the RLE strings [2,4,5,6,10,14,16,17,20,21,22,24]. In this paper, we work in the other direction by providing lower bounds for the approximate matching and alignment problems for RLE strings. Specifically, we show that the two fundamental problems, the wildcard matching and k-mismatches problems, for RLE strings are 3SUM-hard.

The 3SUM problem is to decide whether there exists a triple a, b, c in a set of n integers such that $a + b + c = 0$. Currently, the fastest known algorithms for 3SUM require $\Theta(n^2)$ time. The 3SUM problem serves as a base problem of a class of problems conjectured to require $\Omega(n^2)$ time to solve. We say problem PR is 3SUM-hard if every instance of 3SUM can be reduced to a constant number of instances of PR in $o(n^2)$ time. Many fundamental problems in computational geometry are proved to fall in the class of 3SUM-hard problems [8,15]. Recently, Baran *et al.* [7] show that the 3SUM problem on integers and rationals can be solved in expected $o(n^2)$ time in several models of computation.

In this paper, we investigate the following problems. Note that, we assume the alignment problem is in the linear-gap model with arbitrary scoring matrix.

1. The wildcard matching problem for RLE strings (abbreviated RLE-WILDCARD).
2. The k-mismatches problem for RLE strings (abbreviated RLE-MISMATCH).
3. The local and global alignment problems for RLE strings (abbreviated RLE-ALIGNMENT).

We begin, in Section 3, by considering a 3SUM-hard problem, called *discrete segment containing points* (abbreviated DISCRETE-SCP) [8]. We adopt the notation used in [15]. Given problems PR1 and PR2, we say PR1 $\lll_{f(n)}$ PR2 if every instance of PR1 of size n can be solved by using an instance of PR2 of size $O(n)$ with additional $f(n)$ time. Thus, problem PR is 3SUM-hard if 3SUM $\lll_{f(n)}$ PR, where $f(n) = o(n^2)$. In Sections 3, 4 and 6, we basically establish 3SUM' $\lll_n$ DISCRETE-SCP $\lll_{n \log n}$ RLE-WILDCARD $\lll_n$ RLE-ALIGNMENT and DISCRETE-SCP $\lll_{n \log n}$ RLE-MISMATCH, where 3SUM' is a variant of the 3SUM problem. As a result, the problems listed above all exist an $\Omega(mn)$ barrier, where m and n are the number of runs of the two input RLE strings. In Section 5, we give an upper bound of $O(mn \log m)$ for the combined problem of RLE-WILDCARD and RLE-MISMATCH. It should be noted that when we prove the lower bounds we assume that the input strings are over a finite alphabet, whereas the algorithm we give is capable of handling input strings over an infinite alphabet. Moreover, our algorithm runs in the extremely weak model, as that of [18], in which the alphabet is unordered and only equality of symbols can be tested.

2 Related Work and Previous Results

Given two RLE strings P and T of m and n runs, respectively. Let M and N denote the uncompressed string lengths of P and T. The existing fastest algorithms for both the wildcard matching and k-mismatches problems all rely on the technique of fast Fourier transforms (FFTs). The fastest algorithm for the wildcard matching problem runs in $O(N \log M)$ time [11,13], while the fastest algorithm for the k-mismatches problem runs in $O(N\sqrt{k \log k})$ time [2], which is

$O(N\sqrt{M \log M})$ in the worst case. As for their combined problem, i.e. wildcard matching with mismatches, there exists an $O(N\sqrt{M \log M})$-time solution by extending the algorithms given in [1,19]. Very recently, Clifford *et al.* [12] proposed an $O(Nk \log^2 M(\log^2 k + \log \log M))$-time algorithm which is more efficient for small k. For these problems, we propose in Section 5 a compressed matching algorithm running in $O(mn + p \log m)$ time, where $p \leq mn$ is the number of matched or mismatched runs. The advantage of our approach is two-fold. First, it is a "fully" compressed matching algorithm, meaning that it can cope with RLE strings directly without any decompression. Second, since given two uncompressed strings, one can always compress them into the RLE format in $O(M+N)$ time and then run our algorithm, our result implies an $O(M+N+mn+p \log m)$-time algorithm, $p \leq mn$, for uncompressed strings. For cases where the compression ratio is good enough, i.e. when $O(mn + p \log m)$ is close to $O(M + N)$, our algorithm is close to the trivial lower bound of $O(M+N)$-time, and thus outperforms all the existing algorithms mentioned above. Furthermore, we give 3SUM reductions to demonstrate that there exists an $\Omega(mn)$ barrier in these problems.

On the other hand, for the problem of aligning two RLE strings, Mäkinen *et al.* [22] and Crochemore *et al.* [14] independently proposed an upper bound of $O(mN + Mn)$. A recent work improved the bound to $O(\min\{mN, Mn\})$ [16]. In this paper, we give the 3SUM-hardness result of the problem, which also suggests an $\Omega(mn)$ barrier.

3 Segments Containing Points on Discrete Lines

The 3SUM' problem defined below is a variant of 3SUM, and both problems are proved linearly reducible to each other [15]. In what follows, we use this problem as the base problem of our reductions.

Problem 1. 3SUM': Given three sets of integers A, B, and C, each of size n, are there $a \in A$, $b \in B$, and $c \in C$ with $a + b = c$?

Now we introduce a problem called *discrete segment containing points* (DISCRETE-SCP), defined formally as below.

Problem 2. DISCRETE-SCP: Given a set U of m integers and a set V of n pairwise-disjoint intervals of integers, where $m = O(n)$, is there an integer number (translation) u such that $U + u \subseteq V$? Here, an interval, denoted by $[i, j]$, $i \leq j$, is a set of consecutive integers $\{i, i+1, \ldots, j\}$.

Note that this problem is a discrete version of SCP given in [8]. Following the same paradigm as [8], we show that DISCRETE-SCP is 3SUM-hard.

Theorem 1. 3SUM' $\lll_n$ DISCRETE-SCP.

Proof. Let (A, B, C) be an instance of 3SUM', where $|A| = |B| = |C| = n$. Assume without loss of generality that $A \cup B \cup C$ contains only positive numbers. (Observe that $a + b = c$ if and only if $(a + u) + (b + u) = (c + 2u)$ for any integer number u, thus we can perform translations to sets A, B, and C.) Let $d = \max\{x \mid x \in A \cup B \cup C\} + 1$. We then create two sets U and V, taken as a corresponding instance of DISCRETE-SCP:

$$U = \{10id, 10id + 3d - c_i \mid c_i \in C, i = 1, \ldots, n\}.$$

$$V = [-10nd + 9d, -5d] \cup A \cup \{3d - b \mid b \in B\} \cup [9d, 10nd - 5d].$$

As you can see, each number $c \in C$ corresponds to a pair of integers in U with difference $3d - c$. And each pair $a \in A$, $b \in B$ corresponds to a pair of integers in V with difference $3d - (a + b)$. The two intervals $[-10nd + 9d, -5d]$ and $[9d, 10nd - 5d]$ in V are carefully designed such that (1) there does not exist an integer u such that $U + u \subseteq [-10nd + 9d, -5d] \cup [9d, 10nd - 5d]$ and (2) if there exists a translation u such that $\{10id+u, 10id+3d-c_i+u\} \subseteq A \cup \{3d-b \mid b \in B\}$ for some i, the rest $2n - 2$ points of $U + u$ will be contained by the two intervals. Thus, whether there exists a triple $a \in A$, $b \in B$, $c \in C$ with $a + b = c$ can be seen as whether there exists a translation such that segments in V contain all the points of $U + u$. □

4 Approximate Matching for Run-Length Encoded Strings

The approximate matching problem is, given a pattern $\mathcal{P}$ and a text $\mathcal{T}$, to locate the positions of all "approximate" occurrences of $\mathcal{P}$ in $\mathcal{T}$. There are two commonly used criteria for "approximation": (1) introducing wildcard symbols into the pattern or the text and (2) allowing small mismatches between an occurrence and the pattern. The former is known as the wildcard matching problem, and the latter is known as the k-mismatches problem. In this section, we show that these two problems for RLE strings are 3SUM-hard.

4.1 The Wildcard Matching Problem

The wildcard matching problem is defined as follows. Let Σ denote the alphabet and $*$ denote a wildcard symbol (sometimes called a don't care symbol) which can match any symbol in Σ. Let pattern $\mathcal{P} = p_1 p_2 \ldots p_M$ and text $\mathcal{T} = t_1 t_2 \ldots t_N$ be two strings over $\Sigma \cup \{*\}$. The pattern $\mathcal{P}$ is said to occur at position i in $\mathcal{T}$ if, for every position j in the pattern, either $p_j = t_{i+j-1}$ or at least one of p_j and t_{i+j-1} is a wildcard symbol. In the following, we define the wildcard matching problem for RLE strings.

Problem 3. RLE-WILDCARD: Assume that pattern $\mathcal{P}$ and text $\mathcal{T}$ are strings over $\Sigma \cup \{*\}$. Let P and T be the RLE strings of $\mathcal{P}$ and $\mathcal{T}$, respectively. The problem is to, given P and T, locate all occurrences of $\mathcal{P}$ in $\mathcal{T}$.

Suppose that $P = X_1 X_2 \ldots X_m$ and $T = Y_1 Y_2 \ldots Y_n$, where X_i and Y_j are the i-th and j-th runs of P and T, respectively. If there is no run of wildcard symbols in either string, we can solve the problem optimally in $O(m+n)$ time. First observe that if $\mathcal{P}$ occurs at some position in run Y_j, run X_1 (resp., X_m) is not necessarily identical to run Y_j (resp., Y_{j+m-1}). But runs $X_2X_3\ldots X_{m-1}$ of P must be identical to runs $Y_{j+1}Y_{j+2}\ldots Y_{j+m-2}$ of T, respectively. By viewing a run as a symbol from $\Sigma \times \mathbb{N}$, we can adopt any linear-time string matching algorithm whose running time independent of the alphabet size like KMP algorithm [18], to identify all the occurrences of $X_2X_3\ldots X_{m-1}$ in T in $O(m+n)$ time. For each occurrence candidate, we need an extra examination of run X_1 (resp., X_m) with run Y_j (resp., Y_{j+m-1}). This gives a linear-time solution for matching two RLE strings without wildcard symbols. We now show that when the input contains some runs of wildcard symbols, even when they only appear in the pattern, the problem becomes 3SUM-hard, suggesting an $\Omega(mn)$ time barrier. Specifically, we reduce DISCRETE-SCP to a special case of RLE-WILDCARD, in which the alphabet is binary, i.e. $\Sigma = \{0,1\}$, and wildcard symbols appear only in the pattern.

Theorem 2. DISCRETE-SCP $\lll_{n\log n}$ RLE-WILDCARD.

Proof. We show that the decision problem of RLE-WILDCARD, i.e. whether there exists an occurrence of $\mathcal{P}$ in $\mathcal{T}$, is 3SUM-hard. Given an instance U and V of the DISCRETE-SCP problem with sizes m and n, we construct in the following two RLE strings P and T of at most $2m-1$ and $2n-1$ runs respectively, taken as an instance of the RLE-WILDCARD problem. We first sort U and V in $O(n\log n)$ time. Let $U' = \langle p_1, p_2, \ldots, p_m\rangle$ be the sorted sequence of U, i.e. $p_i < p_{i+1}$, and $V' = \langle [q_1, r_1], [q_2, r_2], \ldots, [q_n, r_n]\rangle$ be the sorted intervals of V, i.e. $q_i \le r_i < q_{i+1} \le r_{i+1}$. We then construct two RLE strings $P = 1^1 *^{p_2-p_1-1} 1^1 *^{p_3-p_2-1} \ldots 1^1 *^{p_m-p_{m-1}-1} 1^1$, and $T = 1^{r_1-q_1+1}0^{q_2-r_1-1}1^{r_2-q_2+1}0^{q_3-r_2-1}\ldots 0^{q_n-r_{n-1}-1}1^{r_n-q_n+1}$. Here, run $0^{q_i-r_{i-1}-1}$ is absent if $q_i - r_{i-1} - 1 = 0$ for some i, and run $*^{p_j-p_{j-1}-1}$ is absent if $p_j - p_{j-1} - 1 = 0$ for some j. Let $\mathcal{P}$ and $\mathcal{T}$ be the uncompressed strings of P and T. It is easily seen that there is a translation u such that $U + u \subseteq V$ if and only if there is an occurrence of $\mathcal{P}$ in $\mathcal{T}$. □

4.2 The *k*-Mismatches Problem

Given two strings $A = a_1a_2\ldots a_\ell$ and $B = b_1b_2\ldots b_\ell$ of equal length, the Hamming distance of A and B is defined as $d_H(A,B) = \ell - |S|$, where $S = \{(i,i) \mid a_i = b_i, 1 \le i \le \ell\}$. Note that $d_H(A,B) = \infty$ if $|A| \ne |B|$. Now we define the *k*-*mismatches* problem for RLE strings.

Problem 4. RLE-MISMATCH: Assume that pattern $\mathcal{P}$ and text $\mathcal{T}$ are strings over alphabet Σ. Let P and T be the RLE strings of $\mathcal{P}$ and $\mathcal{T}$, respectively. The problem is to, given P, T and a threshold number K, locate all substrings $\mathcal{T}'$ of $\mathcal{T}$ such that $d_H(\mathcal{P}, \mathcal{T}') \le K$.

Theorem 3. DISCRETE-SCP $\lll_{n\log n}$ RLE-MISMATCH.

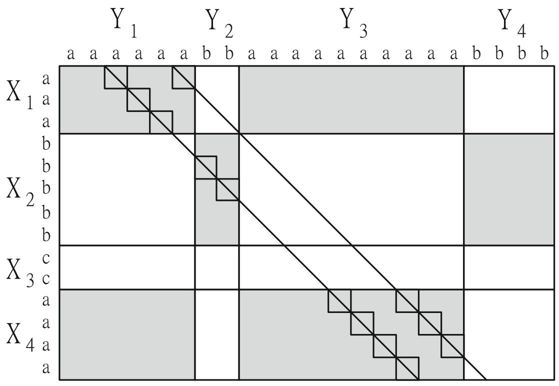
Proof. Given an instance U and V of DISCRETE-SCP, we construct two RLE strings P and T like Theorem 2 except that all the wildcard symbols in $\mathcal{P}$ are replaced by θ, where θ is an extra symbol not appearing in $\mathcal{T}$. Note that the number of symbol θ in $\mathcal{P}$ is $|\mathcal{P}| - m$. Thus, there is a translation u such that $U + u \subseteq V$ if and only if there is an approximate occurrence of $\mathcal{P}$ in $\mathcal{T}$ with at most $|\mathcal{P}| - m$ mismatches. □

5 A Sweep-Line Algorithm for Wildcard Matching with Mismatches

In this section, we give an upper bound for the combined problem of RLE-WILDCARD and RLE-MISMATCH. Assume that pattern $\mathcal{P}$ and text $\mathcal{T}$ are strings over $\Sigma \cup \{*\}$, where Σ is the alphabet (possibly infinite). Let $P = X_1X_2 \dots X_m$ and $T = Y_1Y_2 \dots Y_n$ be the RLE strings of $\mathcal{P}$ and $\mathcal{T}$, respectively. Given P, T and a threshold number K, below we show how to find all substrings $\mathcal{T}'$ of $\mathcal{T}$ such that $d_H(\mathcal{P}, \mathcal{T}') \leq K$.

Let $\mathcal{P}[i]$ (resp., $\mathcal{T}[i]$) denote the i-th symbol of $\mathcal{P}$ (resp., $\mathcal{T}$), and let M and N be the lengths of $\mathcal{P}$ and $\mathcal{T}$, respectively. We define matrix $D[i,j] = \delta(\mathcal{P}[i], \mathcal{T}[j])$, $i = 1, 2, \dots, M$ and $j = 1, 2, \dots, N$, where $\delta(a,b) = 1$ if symbol a matches symbol b and $\delta(a,b) = 0$ otherwise. For those entries $D[i,j]$ with $j - i = d$, they are said to be on diagonal d. To see if $\mathcal{P}$ occurs at position i in $\mathcal{T}$, for some $i = 1, \dots, N - M + 1$, one can accumulate the entries of D on diagonal $i - 1$, since $d_H(\mathcal{P}[1 \dots M], \mathcal{T}[i \dots i + M - 1]) = M - \sum_{j=1}^{M} \delta(\mathcal{P}[j], \mathcal{T}[i + j - 1])$. Thus, our goal becomes to identify those diagonals on which the number of 1's is no less than $M - K$.

Each run pair X_i and Y_j corresponds a sub-matrix $B_{i,j}$ of D. All entries in $B_{i,j}$ are 1's if X_i and Y_j encode the same symbol or either one encodes the wildcard symbol, and are 0's otherwise, so we can partition matrix D into black blocks (of 1's) and white blocks (of 0's). See Figure 1 for an example. Instead



Y1
Y2
Y3
Y4
a a a a a a b b a a a a a a a a a a b b b b
X1
X2
X3
X4
a a a b b b b b c c a a a a

Fig. 1. The matrix D of pattern $P = a^3b^5c^2a^4$ and text $T = a^6b^2a^{10}b^4$. The matrix D is partitioned into black and white blocks. As you can see, diagonals 2 and 5 contain nine and four entries 1's, respectively.

of accumulating the number of 1's diagonal by diagonal, we compute how many 1's a black block contributes to each diagonal of D.

We define $\mu(B_{i,j}, d)$ to be the number of entries in $B_{i,j}$ on diagonal d. Let (x_1, y_1) and (x_2, y_2) denote the positions of the upper-left and lower-right corners of $B_{i,j}$, respectively. The following formula calculates $\mu(B_{i,j}, d)$. To simplify the presentation, we assume that there exists two dummy diagonals $-M$ and N.

– Case I: Suppose $y_1 - x_1 \le y_2 - x_2$.

$$\mu(B_{i,j}, d) = \begin{cases} 0, & \text{for } d = -M, \dots, y_1 - x_2 - 2; \\ d - y_1 + x_2 + 1, & \text{for } d = y_1 - x_2 - 1, \dots, y_1 - x_1 - 1; \\ x_2 - x_1 + 1, & \text{for } d = y_1 - x_1, \dots, y_2 - x_2 - 1; \\ y_2 - x_1 - d + 1, & \text{for } d = y_2 - x_2, \dots, y_2 - x_1; \\ 0, & \text{for } d = y_2 - x_1 + 1, \dots, N. \end{cases}$$

– Case II: Suppose $y_1 - x_1 > y_2 - x_2$.

$$\mu(B_{i,j}, d) = \begin{cases} 0, & \text{for } d = -M, \dots, y_1 - x_2 - 2; \\ d - y_1 + x_2 + 1, & \text{for } d = y_1 - x_2 - 1, \dots, y_2 - x_2 - 1; \\ y_2 - y_1 + 1, & \text{for } d = y_2 - x_2, \dots, y_1 - x_1 - 1; \\ y_2 - x_1 - d + 1, & \text{for } d = y_1 - x_1, \dots, y_2 - x_1; \\ 0, & \text{for } d = y_2 - x_1 + 1, \dots, N. \end{cases}$$

Given a black block $B_{i,j}$, if we plot points $(d, u(B_{i,j}, d))$ on a plane for each $d = -M, \dots, N$, the points will be contained in at most five adjacent line segments of slopes 0, 1, 0, -1, 0, from left to right, respectively. Note that some horizontal line segments may be absent if $y_1 - x_1 = y_2 - x_2$ or $y_1 - x_2 - 1 = -M$ or $y_2 - x_1 = N$. For the correctness of our algorithm, in these cases we assume that there exists a line segment of length 0 therein. The resulting diagram formed by these line segments is in shape of a hat. We call the intersections of two adjacent line segments the *turning points*. Algorithm FindChange (Figure 2) computes all the turning points of diagrams generated by the black blocks.

If we depict the diagrams generated by every black block on the same plane, what we need to do is to "accumulate" those diagrams, see Figure 3. The accumulated diagram is also composed of adjacent line segments. We propose a

```
Algorithm FindChange
Input: Two RLE strings, $P = X_1X_2 \dots X_m$ and $T = Y_1Y_2 \dots Y_n$.
Output: A set $S$ of pairs $(x, \ell)$, where $x$ denotes the $x$-coordinate of a turning point
        and $\ell$ denotes the slope change.
Initialization: $S \leftarrow \phi$;
1  for each $i$,$j$ such that $B[i, j]$ is a black block do
2      Compute the positions of the upper-left corner $(x_1, y_1)$ and the lower-right
3      corner $(x_2, y_2)$ of $B[i, j]$;
4      $S \leftarrow S \cup (y_1 - x_2 - 1, 1) \cup (y_1 - x_1, -1) \cup (y_2 - x_2, -1) \cup (y_2 - x_1 + 1, 1)$;
5  end for
6  Output $S$;
```

Fig. 2. Finding all turning points in diagrams generated by the black blocks

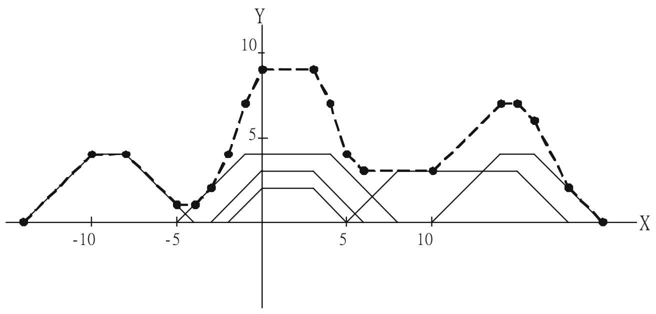


Fig. 3. The solid lines are the diagrams generated by the black blocks. The dashed lines are the result of accumulating them.

```
Algorithm ACCUMULATING
Input: Set $S$ from algorithm FINDCHANGE.
Output: A list $L$ of points.
Initialization: $pos \leftarrow -M-1$; $slope \leftarrow y \leftarrow 0$;
1   Sort $S$ according to the first attribute. Let $S'$ be the sorted list;
2   while $S'$ is not exhausted do;
3       Retrieve the next pair $(x, \ell)$ from list $S'$;
4       if $x \neq pos$ then
5           $y \leftarrow y + (x - pos) \times slope$;
7           Insert $(x, y)$ into the end of list $L$;
6           $pos \leftarrow x$;
8       end if
9       $slope \leftarrow slope + \ell$;
10  end while
11  Output $L$;
```

Fig. 4. A sweep-line algorithm for accumulating the number of matches on each diagonal

sweep-line algorithm, ACCUMULATING (Figure 4), for computing the intersections of these line segments. Once the intersections are computed, the number of matches at each position can be computed in constant time using linear interpolation.

Theorem 4. *There exists an $O(mn + p \log m)$-time algorithm for the combined problem of* RLE-WILDCARD *and* RLE-MISMATCH*, where $p \leq mn$ is the number of matched or mismatched runs.*

Proof. By an $O(mn)$-time preprocessing, one can retrieve all matched runs (black blocks) efficiently. Thus, the for-loop in algorithm FINDCHANGE is executed p times, where p is the number of matched runs. For line 2 of algorithm FINDCHANGE, since $x_1 = \sum_{k=1}^{i-1} |X_k| + 1$, $y_1 = \sum_{k=1}^{j-1} |Y_k| + 1$, $x_2 = \sum_{k=1}^{i} |X_k|$, and $y_2 = \sum_{k=1}^{j} |Y_k|$, it is easy to compute their positions in a total of $O(m+n+p)$

time. The sorting procedure in algorithm ACCUMULATING requires $O(p \log p)$, which leads to a total running time of $O(mn + p \log p)$ for our algorithm, which is $O(mn \log mn)$ in the worst case. To further improve the running time to $O(mn + p \log m)$, one can observe that the output of FINDCHANGE, list S, is partially sorted, and sorting S is essentially merging at most m sorted lists of total size $O(p)$. Moreover, an alternative choice is to count the number of mismatches instead of the number of matches. Thus, p can be the number of mismatched runs (white blocks). □

6 Alignment for Run-Length Encoded Strings

Given two strings $\mathcal{P}$ and $\mathcal{T}$ over a finite alphabet Σ. An alignment of $\mathcal{P}$ and $\mathcal{T}$ is obtained by inserting spaces, denoted by $-$, into or at the ends of $\mathcal{P}$ and $\mathcal{T}$ such that the two resulting strings, $\mathcal{P}'$ and $\mathcal{T}'$, have an identical length, say ℓ. Moreover, if we place $\mathcal{P}'$ and $\mathcal{T}'$ one upon the other, getting a sequence of aligned pairs, there are no two spaces aligned together. Let $\mathcal{A}$ be the aligned sequence of $\mathcal{P}'$ and $\mathcal{T}'$, denoted by $\mathcal{A} = \langle {\mathcal{P}'[1]\ \mathcal{P}'[2] \ldots \mathcal{P}'[\ell] \atop \mathcal{T}'[1]\ \mathcal{T}'[2] \ldots \mathcal{T}'[\ell]} \rangle$. We say $\mathcal{A}$ is an alignment of $\mathcal{P}$ and $\mathcal{T}$, and the score of $\mathcal{A}$ is defined by $score(\mathcal{A}) = \Sigma_{i=1}^{\ell} \delta(\mathcal{P}'[i], \mathcal{T}'[i])$, where $\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$ is a scoring function, sometimes called the scoring matrix. The global alignment problem aims at finding an alignment of $\mathcal{P}$ and $\mathcal{T}$ such that the alignment score is maximized. The local alignment problem aims at finding an alignment of a substring of $\mathcal{P}$ and a substring of $\mathcal{T}$ such that the alignment score is maximized. The two alignment problems are known to be solvable in subquadratic time [14]. In what follows, we consider the compressed optimal alignment problems in which only the compressed forms of $\mathcal{P}$ and $\mathcal{T}$ are given.

Problem 5. RLE-ALIGNMENT: Assume that $\mathcal{P}$ and $\mathcal{T}$ are strings over a finite alphabet Σ. Given P and T, which are the RLE strings of $\mathcal{P}$ and $\mathcal{T}$, the problem is to find the optimal local (global) alignment of $\mathcal{P}$ and $\mathcal{T}$.

Given two RLE strings of m and n runs, one can always decompress them and then run the subquadratic algorithm of [14]. Thus, for cases where the compression ratio is constant, i.e. the length of the decompressed string is proportional to number of runs, we can solve the RLE alignment problems in $o(mn)$ time. In the following, we show that the RLE alignment problems are actually 3SUM-hard, suggesting that any algorithm for these problems may require $\Omega(mn)$ time.

We will show that computing the score of an optimal alignment is 3SUM-hard. Let $score_L(\mathcal{P}, \mathcal{T})$ and $score_G(\mathcal{P}, \mathcal{T})$ denote the scores of an optimal local alignment and an optimal global alignment of $\mathcal{P}$ and $\mathcal{T}$, respectively, and let M and N be the lengths of $\mathcal{P}$ and $\mathcal{T}$.

Theorem 5. RLE-WILDCARD $\lll_n$ RLE-ALIGNMENT.

Proof. An instance P and T of RLE-WILDCARD is taken as an instance of RLE-ALIGNMENT without any modification. The alphabet of both the RLE-WILDCARD and RLE-ALIGNMENT problems are the same, i.e. $\Sigma \cup \{*\}$. We define the scoring matrix $\delta : (\Sigma \cup \{*, -\}) \times (\Sigma \cup \{*, -\}) \to \mathbb{R}$ as follows. For symbols $x, y \in \Sigma$, let $\delta(x, y) = 1$ if $x = y$ and $\delta(x, y) = -1$, otherwise. Let $\delta(x, *) = \delta(*, x) = 1$ for every symbol $x \in \Sigma$. Let $\delta(x, -) = \delta(-, x) = -1$ for every symbol $x \in \Sigma \cup \{*\}$.

We want to show that there exists an occurrence of $\mathcal{P}$ in $\mathcal{T}$ if and only if $score_L(\mathcal{P}, \mathcal{T}) = M$. First observe that $score_L(\mathcal{P}, \mathcal{T}) \leq M$ since only match pairs can get a positive score of 1 and there are at most M matches between $\mathcal{P}$ and $\mathcal{T}$. If there exists an occurrence $\mathcal{T}[i \dots i + M - 1]$ of $\mathcal{P}$ in $\mathcal{T}$, we can construct a local alignment $\mathcal{A} = \langle \begin{smallmatrix} \mathcal{P}[1] & \mathcal{P}[2] & \dots & \mathcal{P}[M] \\ \mathcal{T}[i] & \mathcal{T}[i+1] & \dots & \mathcal{T}[M+i-1] \end{smallmatrix} \rangle$ of score M. Thus, we have $score_L(\mathcal{P}, \mathcal{T}) \geq M$, which leads to $score_L(\mathcal{P}, \mathcal{T}) = M$. On the other hand, if $score_L(\mathcal{P}, \mathcal{T}) = M$, we know that an optimal local alignment must contain exactly M match pairs and no gap or mismatch pair, for otherwise $score_L(\mathcal{P}, \mathcal{T})$ would be strictly less than M. Thus, an optimal local alignment corresponds to an exact occurrence of $\mathcal{P}$ in $\mathcal{T}$.

To see the reduction holds for the global alignment problem, we introduce a redundant symbol ϵ into the alphabet. That is, the alphabet of the RLE-ALIGNMENT becomes $\Sigma \cup \{*, \epsilon\}$. An instance P and T of RLE-WILDCARD is replaced by P' and T. Here, P' is obtained by appending two runs of redundant symbols ϵ to both ends of P. More specifically, let $P' = \epsilon^{N-M} \cdot P \cdot \epsilon^{N-M}$, an RLE string of $m+2$ runs, where $\cdot$ denotes a concatenation. Let $\mathcal{P}'$ be the decompressed string of P'. The scoring function $\delta : (\Sigma \cup \{\epsilon, *, -\}) \times (\Sigma \cup \{\epsilon, *, -\}) \to \mathbb{R}$ is defined in the same way as before with additional mapping of $\delta(\epsilon, x) = \delta(x, \epsilon) = 0$ for every symbol $x \in \Sigma \cup \{*, -\}$. We know $score_G(\mathcal{P}', \mathcal{T}) \leq M$ since there are at most M matches. If there exists an occurrence of $\mathcal{P}$ in $\mathcal{T}$, say $\mathcal{T}[i \dots i + M - 1]$, we can construct a global alignment

$$\mathcal{A} = \langle \begin{smallmatrix} \epsilon & \dots & \epsilon & \epsilon & \dots & \epsilon & \mathcal{P}[1] & \dots & \mathcal{P}[M] & \epsilon & \dots & \epsilon & \epsilon & \dots & \epsilon \\ - & \dots & - & \mathcal{T}[1] & \dots & \mathcal{T}[i-1] & \mathcal{T}[i] & \dots & \mathcal{T}[M+i-1] & \mathcal{T}[M+i] & \dots & \mathcal{T}[n] & - & \dots & - \end{smallmatrix} \rangle$$

of score M. Thus, $score_G(\mathcal{P}', \mathcal{T}) = M$. On the other hand, suppose $score_G(\mathcal{P}', \mathcal{T}) = M$. An optimal global alignment of $\mathcal{P}'$ and $\mathcal{T}$must contain exactly M match pairs, which implies that there exists an exact occurrence of $\mathcal{P}$ in $\mathcal{T}$. □

7 Concluding Remarks

In this paper, we give a lower bound and an upper bound for RLE-WILDCARD and RLE-MISMATCH. There exists a log-factor gap between the bounds. Moreover, the gap between the existing upper bound and our lower bound of RLE-ALIGNMENT is still large. Bridging the gaps between the bounds remains open.

Acknowledgements

The authors would like to thank I-Hsuan Yang and the anonymous reviewers for their helpful comments. Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao were supported in part by NSC grants 95-2221-E-002-126-MY3 and 97-2221-E-002-097-MY3 from the National Science Council, Taiwan.

References

1. Abrahamson, K.: Generalized String Matching. SIAM Journal on Computing 16(6), 1039–1051 (1987)
2. Amir, A., Benson, G.: Efficient Two-Dimensional Compressed Matching. In: DCC, pp. 279–288 (1992)
3. Amir, A., Lewenstein, M., Porat, E.: Faster Algorithms for String Matching with k Mismatches. Journal of Algorithms 50(2), 257–275 (2004)
4. Amir, A., Landau, G.M., Sokol, D.: Inplace Run-Length 2d Compressed Search. Theoretical Computer Science 290(3), 1361–1383 (2003)
5. Apostolico, A., Landau, G.M., Skiena, S.: Matching for Run-Length Encoded Strings. Journal of Complexity 15(1), 4–16 (1999)
6. Arbell, O., Landau, G.M., Mitchell, J.S.B.: Edit Distance of Run-Length Encoded Strings. Information Processing Letters 83(6), 307–314 (2002)
7. Baran, I., Demaine, E.D., Patrascu, M.: Subquadratic Algorithms for 3SUM. Algorithmica 50(4), 584–596 (2008)
8. Barequet, G., Har-Peled, S.: Polygon Containment and Translational Min-Hausdorff-Distance Between Segment Sets are 3SUM-Hard. International Journal of Computational Geometry and Applications 11(4), 465–474 (2001)
9. Berghorn, W., Boskamp, T., Lang, M., Peitgen, H.-O.: Fast Variable Run-Length Coding for Embedded Progressive Wavelet-Based Image Compression. IEEE Transactions on Image Processing 10(12), 1781–1790 (2001)
10. Bunke, H., Csirik, J.: An Improved Algorithm for Computing the Edit Distance of Run-Length Coded Strings. Information Processing Letters 54(2), 93–96 (1995)
11. Clifford, P., Clifford, R.: Simple Deterministic Wildcard Matching. Information Processing Letters 101(2), 53–54 (2007)
12. Clifford, R., Efremenko, K., Porat, E., Rothschild, A.: From Coding Theory to Efficient Pattern Matching. In: SODA (2009)
13. Cole, R., Hariharan, R.: Verifying Candidate Matches in Sparse and Wildcard Matching. In: STOC, pp. 592–601 (2002)
14. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. SIAM Journal on Computing 32(6), 1654–1673 (2003)
15. Gajentaan, A., Overmars, M.H.: On a Class of $O(n^2)$ Problems in Computational Geometry. Computational Geometry 5, 165–185 (1995)
16. Huang, G.-S., Liu, J.J., Wang, Y.-L.: Sequence Alignment Algorithms for Run-Length-Encoded Strings. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 319–330. Springer, Heidelberg (2008)
17. Kim, J.W., Amir, A., Landau, G.M., Park, K.: Computing Similarity of Run-Length Encoded Strings with Affine Gap Penalty. Theoretical Computer Science 395(2–3), 268–282 (2008)
18. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast Pattern Matching in Strings. SIAM Journal on Computing 6(2), 323–350 (1977)
19. Kosaraju, S.R.: Efficient String Matching (manuscript, 1987)
20. Liu, J.J., Huang, G.-S., Wang, Y.-L., Lee, R.C.T.: Edit Distance for a Run-Length-Encoded String and an Uncompressed String. Information Processing Letters 105(1), 12–16 (2007)

21. Liu, J.J., Wang, Y.-L., Lee, R.C.T.: Finding a Longest Common Subsequence between a Run-Length-Encoded String and an Uncompressed String. Journal of Complexity 24(2), 173–184 (2008)
22. Mäkinen, V., Ukkonen, E., Navarro, G.: Approximate Matching of Run-Length Compressed Strings. Algorithmica 35(4), 347–369 (2003)
23. Bodson, D., McConnell, K.R., Schaphorst, R.: FAX: Digital Facsimile Technology and Applications. Artech House, Norwood (1989)
24. Mitchell, J.S.B.: A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings. Technical Report, SUNY Stony Brook (1997)

Modeling and Algorithmic Challenges in Online Social Networks

Ravi Kumar

Yahoo! Research, 701 First Ave., Sunnyvale, CA 94089, USA
`ravikumar@yahoo-inc.com`

Abstract. Online social networks have become major and driving phenomena on the web. In this talk we will address key modeling and algorithmic questions related to large online social networks. From the modeling perspective, we raise the question of whether there is a generative model for network evolution. The availability of time-stamped data makes it possible to study this question at an extremely fine granularity. We exhibit a simple, natural model that leads to synthetic networks with properties similar to the online ones. From an algorithmic viewpoint, we focus on challenges posed by the magnitude of data in these networks. In particular, we examine topics related to influence and correlation in user activities and compressibility of such networks.

Permuted Longest-Common-Prefix Array*

Juha Kärkkäinen[1], Giovanni Manzini[2], and Simon J. Puglisi[3]

[1] Department of Computer Science,
University of Helsinki, Finland
juha.karkkainen@cs.helsinki.fi
[2] Department of Computer Science,
University of Eastern Piedmont, Italy
manzini@mfn.unipmn.it
[3] School of Computer Science and Information Technology,
Royal Melbourne Institute of Technology, Australia
simon.puglisi@rmit.edu.au

Abstract. The longest-common-prefix (LCP) array is an adjunct to the suffix array that allows many string processing problems to be solved in optimal time and space. Its construction is a bottleneck in practice, taking almost as long as suffix array construction. In this paper, we describe algorithms for constructing the *permuted LCP* (PLCP) array in which the values appear in position order rather than lexicographical order. Using the PLCP array, we can either construct or *simulate* the LCP array. We obtain a family of algorithms including the fastest known LCP construction algorithm and some extremely space efficient algorithms. We also prove a new combinatorial property of the LCP values.

1 Introduction

The suffix array (SA) [13] is a lexicographically sorted list of all the suffixes in a string. The longest-common-prefix (LCP) array stores the lengths of the longest-common-prefixes of adjacent suffixes in SA. Augmenting SA with LCP allows many problems in string processing to be solved in optimal time and space. In particular the LCP array is key for: efficiently simulating traversals of the suffix tree [22,5] (top-down, bottom up, suffix link walks) with the suffix array [1]; pattern matching on the suffix array in attractive theoretical bounds [13,1]; fast disk based suffix array arrangements [3,21]; and compressed suffix trees [4].

Various methods for suffix arrays have been extensively investigated but the LCP array has received much less attention. Several very fast SA construction algorithms have been developed in recent years [17], but there has been no improvement in LCP construction time since the original LCP-from-SA algorithm [8]. Furthermore, the fastest SA construction algorithms are also space economical, which is not the case with LCP construction. This is a problem since space is

* This work is supported by the Academy of Finland grant 118653 (ALGODAN), by the Italy-Israel Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics", and by the Australian Research Council.

an even bigger concern than time. For large documents, full representations of the text, the SA, and the LCP array cannot be stored (simultaneously) in RAM. There are many methods for SA addressing this problem, including compressed representations [15], external storage [21], and external and semi-external construction [2,6]. The few LCP related improvements have been concerned with the space too (see Section 2).

In this paper we show that a natural and effective way to reduce the time and space costs of LCP computation is the use of a simple alternative representation of LCP values. Instead of storing them in the classical LCP array we store them in the *permuted LCP* (PLCP) array in which the values appear in position order, rather than lexicographical order. The PLCP array has played a role in previous algorithms implicitly (see Lemma 1) or even explicitly [19,11], but we bring it to the center stage. We use the PLCP array as the central piece that connects a number of techniques (old and new) into a family of algorithms.

One advantage of the PLCP array over the LCP array is that it supports compact representation — in two different ways, in fact: sparse array [11] and succinct bitarray [19]. Each representation can be used for *simulating* the LCP array. The properties of the representations are summarized in Table 1.

Table 1. Properties of PLCP representations. The construction space does not include the space for the text and the SA which are the input to the construction algorithms.

	Space	LCP random access	Construction Time	Construction Space
Full array	n words	$\mathcal{O}(1)$	$\mathcal{O}(n)$	n words
Sparse array	n/q words	$\mathcal{O}(q)$ amortized	$\mathcal{O}(n)$	n/q words
Succinct bitarray	$2n + o(n)$ bits	$\mathcal{O}(1)$	$\mathcal{O}(n)$	n words + $2n$ bits
			$\mathcal{O}(n \log n)$	$3n$ bits

The other advantage of the PLCP array over the LCP array is faster construction. The main contribution of this paper are efficient and space economical construction algorithms. The first one (called Φ in Section 5) is a linear time algorithm that is extremely fast in practice. It constructs the full PLCP array roughly 2.2 times faster than the fastest LCP array construction algorithm. Futhermore, combining the algorithm with an LCP-from-PLCP construction yields the fastest known algorithm for computing the LCP array. More space efficient variants of the algorithm (called Φip and Φx in Section 5) are also faster than other comparable algorithms.

Another contribution of the paper is a novel, non-trivial combinatorial property of the (P)LCP array (Theorem 1), which leads to another PLCP construction algorithm (called IB in Section 5). It enables the construction of the succinct bitarray representation with a peak space usage of only $3n$ bits in addition to the text and the SA. Although the algorithm runs in $\mathcal{O}(n \log n)$ time, it is quite fast in practice.

The speed of the algorithms comes from their mostly sequential access patterns, which enable prefetching and avoid cache misses. In particular, the Φ algorithm accesses only one array non-sequentially in any stage. The sequential access patterns help in reducing space, too. With the exception of Φip, all our construction algorithms process SA in sequential order and can produce the LCP array in sequential order. Thus, as in [18], we can keep SA and LCP on disk without an excessive slow-down obtaining *semi-external* algorithms that need RAM only for the text and the PLCP representation, which is only $3n$ bits for the bitarray and even less for the sparse array.

2 Background and Related Work

Throughout we consider a string $t = t[0..n] = t[0]t[1] \ldots t[n]$ of $n+1$ symbols. The first n symbols of t are drawn from a constant ordered alphabet, Σ. The final character $t[n]$ is a special "end of string" character, \$, distinct from and lexicographically smaller than all the other characters in Σ.

For $i = 0, \ldots, n$ we write $t[i..n]$ to denote the ***suffix*** of t of length $n-i+1$, that is $t[i..n] = t[i]t[i+1] \cdots t[n]$. For convenience we will frequently refer to suffix $t[i..n]$ simply as "suffix i". Similarly, we write $t[0..i]$ to denote the ***prefix*** of t of length $i+1$. We write $t[i..j]$ to represent the ***substring*** $t[i]t[i+1] \cdots t[j]$ of t that starts at position i and ends at position j.

The ***suffix array*** of t, denoted SA_t or just SA when the context is clear, is an array $\mathrm{SA}[0..n]$ which contains a permutation of the integers $0..n$ such that $t[\mathrm{SA}[0]..n] < t[\mathrm{SA}[1]..n] < \cdots < t[\mathrm{SA}[n]..n]$. In other words, $\mathrm{SA}[j] = i$ iff $t[i..n]$ is the j^{th} suffix of t in ascending lexicographical order.

The ***lcp array*** $\mathrm{LCP} = \mathrm{LCP}[0..n]$ is an array defined by t and SA_t. Let $\mathrm{lcp}(y, z)$ denote the length of the longest common prefix of strings y and z. For every $j \in 1..n$,

$$\mathrm{LCP}[j] = \mathrm{lcp}(t[\mathrm{SA}[j-1]..n], t[\mathrm{SA}[j]..n]),$$

that is, $\mathrm{LCP}[j]$ is the length of the longest common prefix of suffixes $\mathrm{SA}[j-1]$ and $\mathrm{SA}[j]$. $\mathrm{LCP}[0]$ is undefined.

The ***permuted lcp array*** — $\mathrm{PLCP}[0..n-1]$ — has the same contents as LCP but in different order. Specifically, for every $j \in 1..n$,

$$\mathrm{PLCP}[\mathrm{SA}[j]] = \mathrm{LCP}[j]. \tag{1}$$

The LCP array first appeared in the original paper on suffix arrays [13], where Manber and Myers show how to compute the LCP array as a byproduct of their $\mathcal{O}(n \log n)$ time SA construction algorithm. The linear time SA construction algorithm of Kärkkäinen and Sanders can also be modified to produce the LCP array [7]. Other SA construction algorithms could probably be modified so too, but a more attractive approach was introduced by Kasai et al. [8], who gave a simple algorithm to construct the LCP array from an already constructed SA in

$\Theta(n)$ time using $2n$ words of extra space[1]. In the rest of the paper we follow this approach and assume that we compute lcp values knowing the text and SA.

The first improvements to Kasai et al.'s algorithm reduced the extra space to n words by eliminating the need for an extra working array through essentially reordering computation. Two different ways to achieve this have been described by Mäkinen [12] and Manzini [14]. Manzini also gives another algorithm that saves space by overwriting SA with LCP, which is not possible with earlier algorithms. It needs n bytes plus n words of extra space in the worst case but usually significantly less than that.

Recently, Puglisi and Turpin [18] gave another algorithm that can overwrite SA with LCP, using $\mathcal{O}(nv)$ time and $\mathcal{O}(n/\sqrt{v})$ extra space, where v is a parameter that controls a space-time tradeoff. The algorithm accesses SA and produces LCP in a strictly left to right manner allowing SA and LCP to reside on disk without a large penalty in speed. This is the first semi-external LCP construction algorithm. In practice, it needs less than twice the size of the text in primary memory.

The PLCP array first appeared in its succinct bitarray form in [19], where Sadakane introduced it as a concise representation of the LCP array, but he did not address the problem of constructing it. The full or sparse PLCP array is used in the LCP construction algorithm by Khmelev, which has not been published in literature but an implementation is available [11].

3 Storing and Using the PLCP Array

The standard way of storing the PLCP array is an array of integers — $\mathrm{PLCP}[0..n-1]$. This ***full array*** representation takes n words of storage and because of (1) supports random access to LCP values in $\mathcal{O}(1)$ time.

Two concise representations of the PLCP array are based on the following key property of the PLCP array.

Lemma 1. *For every* $i \in 1..n-1$, $\mathrm{PLCP}[i] \geq \mathrm{PLCP}[i-1] - 1$. □

The lemma was proven by Kasai et al. [8] in a slightly different form. All efficient (P)LCP construction algorithms (including ours) rely on this property.

The ***sparse PLCP array*** — PLCP_q — of $\lceil n/q \rceil$ integers contains only every q^{th} entry of the PLCP array, i.e., $\mathrm{PLCP}_q[i] = \mathrm{PLCP}[iq]$. Lemma 1 allows us to estimate the missing PLCP entries as follows.

Lemma 2. *For any* $i \in 0..n-1$, *let* $a = \lfloor i/q \rfloor$ *and* $b = i \bmod q$, *i.e.,* $i = aq + b$. *If* $(a+1)q \leq n-1$, *then* $\mathrm{PLCP}_q[a] - b \leq \mathrm{PLCP}[i] \leq \mathrm{PLCP}_q[a+1] + q - b$. *If* $(a+1)q > n-1$, *then* $\mathrm{PLCP}_q[a] - b \leq \mathrm{PLCP}[i] \leq n - i \leq q$. □

Using these bounds, we can compute the actual value of $\mathrm{PLCP}[i]$ by doing at most $q + \mathrm{PLCP}_q[a+1] - \mathrm{PLCP}_q[a]$ comparisons (or at most q comparisons if

[1] Throughout we will use "extra space" to mean space in addition to t and SA, including the n words required to hold LCP, unless otherwise stated.

$(a+1)q > n-1$). This requires an access to the text and the SA (or Φ_q, see Section 4). The number of comparisons can be close to n for some i but the average number, over all LCP array entries, is at most $\mathcal{O}(q)$ as shown by the following lemma.

Lemma 3. *Assuming the text and the* SA *are available, the sparse* PLCP *array* PLCP_q *supports random access to* LCP *values in $\mathcal{O}(q)$ amortized time.*

Proof. For $1 \le i \le n$ it is $\mathrm{LCP}[i] = \mathrm{PLCP}[\mathrm{SA}[i]]$. Hence, if $\mathrm{SA}[i] = qk$ then $\mathrm{LCP}[i] = \mathrm{PLCP}_q[k]$ and computing it takes $\mathcal{O}(1)$ time. Otherwise let $\mathrm{SA}[i] = aq+b$, with $0 \le b < q$. By Lemma 2 we can compute $\mathrm{LCP}[i]$ comparing at most $q+\mathrm{PLCP}_q[a+1]-\mathrm{PLCP}_q[a]$ symbols of the suffixes $t[\mathrm{SA}[i-1]..n]$ and $t[\mathrm{SA}[i]..n]$ (or at most q symbols if $(a+1)q > n-1$). For $i = 1,\dots,n$ let $c(i)$ denote the number of comparisons needed for computing $\mathrm{LCP}[i]$. We prove the lemma by showing that

$$\frac{1}{n}\sum_{i=1}^{n} c(i) \le (q-1) + q^2/n.$$

Let $a' = \lfloor (n-1)/q \rfloor$ so that $n' = a'q$ is the largest multiple of q smaller than n. For the above observation, the indexes i such that $\mathrm{SA}[i] \ge n'$ contribute to the sum $\sum_{i=1}^{n} c(i)$ by at most q^2. To complete the proof we show that

$$\sum_{i:\ SA[i]<n'} c(i) \ \le\ (q-1)n.$$

Since for $k = 1,\dots,a'-1$ there are exactly $q-1$ indexes i such that $kq < \mathrm{SA}[i] < (k+1)q$ we have

$$\sum_{i:\ SA[i]<n'} c(i) \ \le\ (q-1)\sum_{k=0}^{a'-1}(q+\mathrm{PLCP}_q[k+1]-\mathrm{PLCP}_q[k]). \tag{2}$$

Define $w(k) = \mathrm{PLCP}_q[k] + kq$. Then, (2) can be rewritten as

$$\begin{aligned}\sum_{i:\ SA[i]<n'} c(i) \ &\le\ (q-1)\sum_{k=0}^{a'-1}(w(k+1)-w(k))\\ &=\ (q-1)(w(a')-w(0))\\ &\le\ (q-1)(\mathrm{PLCP}_q[a']+a'q)\\ &=\ (q-1)(\mathrm{PLCP}(n')+n')\\ &\le\ (q-1)n.\end{aligned}$$

and the lemma follows. □

The ***succinct bitarray*** representation of the PLCP array [19] consists of a bit array $\mathrm{B}[0..2n-1]$, where the $\mathrm{B}[j] = 1$ if and only if $j = 2i + \mathrm{PLCP}[i]$ for some $i \in [0..n-1]$. Note that due to Lemma 1, $2i + \mathrm{PLCP}[i]$ has a different value for each i. Then $\mathrm{PLCP}[i] = select_1(\mathrm{B}, i+1) - 2i$, where $select_1(\mathrm{B}, j)$ returns the position of the j^{th} 1-bit in B. The select-query can be answered in $\mathcal{O}(1)$ time given an additional data structure of $o(n)$ bits. Several such select-structures are known in the literature, the most practical of these for our purposes is probably the *darray* [16]. Summing up, the succinct bitarray representation takes $2n+o(n)$ bits and supports random access to LCP values in $\mathcal{O}(1)$ time.

We have implemented our own optimized variant of the darray structure using the fact that we do not need support for the rank operation. The data structure consists of the bitvector B (modified as described below) and a sparse PLCP array, PLCP_q. The PLCP_q entries indicate the positions of every q^{th} 1-bit in B, dividing B into n/q blocks (of varying sizes). The select operation finds the position of the j^{th} 1-bit. Locating the correct block is easy with PLCP_q. If the block is small, we simply scan the block bitvector to find the correct bit. A bitvector of $\mathcal{O}(\log n)$ bits can be scanned in $\mathcal{O}(1)$ time using appropriate lookup tables (of size $o(n)$). If the block is large enough, we replace the bitvector for the block with a full PLCP array for that block, i.e., with an array of $q-1$ integers pointing to all the 1-bits within the block. A single lookup is enough to locate the bit we want. The main difference to the darray is that the full PLCP arrays for large blocks are stored over the bitvector B overwriting the bits there.

There are several ways of using the PLCP array depending on the application. For some applications, the PLCP array is just as good as the LCP array. Computing the average LCP value is an example. Most applications, though, need the LCP array. We have two options in this case. First, we can simulate the LCP array using (1). Second, we can compute the LCP array from the PLCP array, which is then discarded.

When turning the PLCP array into the LCP array, there are some space saving techniques worth mentioning. First, it is possible to do this by an in-place permutation, i.e., by using a single array that contains the PLCP array at start and the LCP array at finish. In addition to this array, the in-place permutation needs the SA and n bits used as markers. Second, it is possible to store the LCP array and the SA on disk memory. All our PLCP construction algorithms do just a single sequential scan over the SA. Similarly, constructing the LCP array from the PLCP array can be done in sequential order with respect to the LCP and SA. Thus, in main memory we only need to keep the text and the PLCP array, the latter possibly using a compact representation.

4 Constructing the PLCP Array

4.1 Computing PLCP Using the ϕ Array

Our first PLCP construction algorithm uses another array $\Phi[0..n-1]$.[2] For every $j \in 1..n$,

$$\Phi[\mathrm{SA}[j]] = \mathrm{SA}[j-1].$$

[2] The Φ array is so named because it is in some way symmetric to the Ψ array [20].

```
      — Compute $\Phi_q$
 1: for $i \leftarrow 0$ to $n-1$ do
 2:     if $\mathrm{SA}[i] \bmod q = 0$ then
 3:         $\Phi[\mathrm{SA}[i]/q] \leftarrow \mathrm{SA}[i-1]$
      — Turn $\Phi_q$ into $PLCP_q$
 4: $\ell \leftarrow 0$
 5: for $i \leftarrow 0$ to $\lfloor (n-1)/q \rfloor$ do
 6:     $j \leftarrow \Phi_q[i]$
 7:     while $t[iq+\ell] = t[j+\ell]$ do
 8:         $\ell \leftarrow \ell+1$
 9:     $\mathrm{PLCP}_q[i] \leftarrow \ell$
10:     $\ell \leftarrow$ max $(\ell - q, 0)$
```

(a)

```
      — Compute irreducible lcp values
 1: for $i \leftarrow 1$ to $n$ do
 2:     $j \leftarrow \mathrm{SA}[i-1]$
 3:     $k \leftarrow \mathrm{SA}[i]$
 4:     if $t[j-1] \neq t[k-1]$ then
 5:         $\ell \leftarrow lcp(t[j..n], t[k..n])$
 6:         $k' \leftarrow \lceil k/q \rceil \cdot q$
 7:         if $\mathrm{PLCP}_q[k'] < \ell - k' + k$ then
 8:             $\mathrm{PLCP}_q[k'] \leftarrow \ell - k' + k$
      — Fill in the other values
 9: for $i \leftarrow 1$ to $\lfloor (n-1)/q \rfloor$ do
10:     if $\mathrm{PLCP}_q[i] < \mathrm{PLCP}_q[i-1] - q$ then
11:         $\mathrm{PLCP}_q[i] \leftarrow \mathrm{PLCP}_q[i-1] - q$
```

(b)

Fig. 1. Algorithms for computing Sparse PLCP with sample rate q. The algorithm in (a) uses Φ; the one in (b) uses Irriducible LCPs and assumes $t[-1] = t[n] = \$$. Setting $q = 1$ produces the Full PLCP representation.

Clearly, the Φ array can be computed easily with a scan of the suffix array. The Φ array is also used in [11], though differently from the way we use it.

The Φ array is closely related to the PLCP array: for every $i \in [0..n-1]$,

$$\mathrm{PLCP}[i] = lcp(i, \Phi[i]).$$

Thus to compute $\mathrm{PLCP}[i]$ we just need to compare the suffixes i and $\Phi[i]$, and similar to the algorithm of Kasai et al., Lemma 1 allows us to skip the first $\mathrm{PLCP}[i-1] - 1$ characters in the comparison. Also note that we can save space by overwriting Φ with PLCP.

The technique generalizes easily to computing the sparse PLCP array by using the sparse version of the Φ array — Φ_q. The full algorithm is given in Fig. 1(a). We can also easily compute the bitarray B this way. The technique is not very space efficient, though, since we need the full Φ array.

4.2 Computing PLCP Using Irreducible LCPs

Our second technique of computing the PLCP array is based on the concept of ***irreducible*** lcp values. We say that $\mathrm{PLCP}[i] = \mathrm{lcp}(i, \phi[i])$ is *reducible* if $t[i-1] = t[\phi[i]-1]$. Reducible values are easy to compute via the next lemma.

Lemma 4. *If* $\mathrm{PLCP}[i]$ *is reducible, then* $\mathrm{PLCP}[i] = \mathrm{PLCP}[i-1] - 1$.

In essence the same result appeared as Lemma 1 in [14].

The idea of the algorithm is to first compute the irreducible lcp values and then use Lemma 4 to fill in the reducible values. The irreducible lcp values are computed *naively*, i.e., by comparing the suffixes i and $\Phi[i]$ from the beginning. The efficiency of the algorithm is based on the following surprising property of the irreducible lcp values. The property is a conjecture by Dmitry Khmelev [10], but we provide the first proof for it.

Theorem 1. *The sum of all irreducible lcp values is* $\leq 2n \log n$.

Proof. Let $\ell = \mathrm{PLCP}[i] = lcp(i,j)$ (i.e., $j = \Phi[i]$) be an irreducible lcp value, i.e., $t[i-1] \neq t[j-1]$, $t[i..i+\ell-1] = t[j..j+\ell-1]$ and $t[i+\ell] \neq t[j+\ell]$. For every $k \in 0..\ell-1$, the matching pair of characters $t[i+k] = t[j+k]$ contributes to the lcp value and we account for it as follows.

Consider the suffix tree of the reverse of t, and let v_{i+k} and v_{j+k} be the leafs corresponding to the prefixes $t[0..i+k]$ and $t[0..j+k]$. The nearest common ancestor u of v_{i+k} and v_{j+k} represents the reverse of $t[i..i+k]$ (because $t[i-1] \neq t[j-1]$). If v_{i+k} is in a smaller subtree of u than v_{j+k}, the cost of the pair $t[i+k] = t[j+k]$ is assigned to v_{i+k}, otherwise to v_{j+k}.

Now we show that each leaf v carries a cost of at most $2 \log n$. Whenever v is assigned a cost, this is associated with an ancestor u of v and another leaf w under u. We call u a costly ancestor of v and w a costly cousin of v. We will show that (a) each leaf v has at most $\log n$ costly ancestors, and that (b) for each costly ancestor, there is at most two costly cousins.

To show (a), we use the “smaller half trick”. Consider the path from v to the root. At each costly ancestor u, the size of the subtree at least doubles with the addition of the subtree containing w. Thus there are at most $\log n$ costly ancestors. Let v be leaf, u a costly ancestor of v and w a corresponding costly cousin representing the reverse of the strings $t[0..i+k]$, $t[i..i+k]$ and $t[0..j+k]$, respectively. Then either $i = \Phi[j]$ or $j = \Phi[i]$. Suppose the former and assume there is another costly cousin $w' \neq w$ of v with the same costly ancestor. Then w' must represent $t[0..j'+k]$ for $j' = \Phi[i]$. Adding a third costly cousin is then no more possible, which proves (b). □

The theorem is asymptotically tight as shown by the next lemma.

Lemma 5. *For a binary de Bruijn sequence of order* k*, the sum of all irreducible lcp values is* $(k-1)2^{k-1} - \Theta(1)$*. As* $n = 2^k + k - 1$ *is the length of the sequence, the sum of irreducible lcp values is* $(n/2)\log n - \mathcal{O}(n)$*.*

Proof. Let x be any sequence on $\Sigma = \{0,1\}$ of length $k-1$. $x0$ and $x1$ both appear in the de Bruijn sequence so they are in contiguous positions of the suffix array. The symbols preceding $x0$ and $x1$ cannot be identical, otherwise the de Bruijn sequence would contain two identical length-k subsequences. Thus, we have an irreducible lcp value $lcp(x0, x1) = k-1$. The lemma follows since there are 2^{k-1} such x's. □

The notion of irreducible lcp can be used to compute all our PLCP representations with the same idea: first compute the irreducible lcp values naively and then fill in the rest using Lemma 4. When computing the full PLCP array the algorithm is trivial and by Theorem 1 requires $\mathcal{O}(n \log n)$ time. With the sparse PLCP array, we use the following fact

$$\mathrm{PLCP}[i] = \max\{\mathrm{PLCP}[j] - (i-j) : j \leq i \text{ and } \mathrm{PLCP}[j] \text{ is irreducible.}\}$$

Table 2. Data files used for empirical tests, sorted in ascending order of average LCP

Name	Mean LCP	Max LCP	Size (bytes)	Description
sprot	89	7,373	109,617,186	Swiss prot database
rfc	93	3,445	116,421,901	RFC text files
linux	479	136,035	116,254,720	Linux kernel 2.4.5 source
jdk13	679	37,334	69,728,899	html/java files from JDK 1.3
etext	1,109	286,352	105,277,340	Gutenberg etext99/*.txt files
chr22	1,979	199,999	34,553,758	Human chromosome 22
gcc	8,603	856,970	86,630,400	gcc 3.0 source files
w3c	42,300	990,053	104,201,579	HTML files from w3c.org

The first stage updates the nearest sparse entry following each irreducible lcp value and the second stage fills in the rest. The full algorithm is in Fig. 1(b).

With the bitarray representation, we need a second bit array $\mathrm{C}[0..n-1]$ to store the positions of the irreducible entries. For each irreducible entry $\mathrm{PLCP}[i]$, we set the bit i in C and the bit $\mathrm{PLCP}[i]+2i$ in B. Once done, setting the bits for reducible values is easy. This algorithm needs only $3n$ bits in addition to the text and the suffix array and requires $\mathcal{O}(n \log n)$ time to construct.

5 Experimental Results

For testing we used the files listed in Table 2[3]. All tests were conducted on a 3.0 GHz Intel Xeon CPU with 4Gb main memory and 1024K L2 Cache. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 3.4.4) executed with the -O3 option. Times given are the minima of three runs and were recorded with the standard C `getrusage` function.

Experiments measured the time to compute various PLCP representations, and the classical LCP array. All methods tested take as input t and SA, which are not modified. All data structures reside in primary memory. The algorithms and their space requirements are described in Table 3. Included are three previous approaches. For consistency we modified the pt$x$ code to produce LCP in a separate array, not overwriting SA as in [18].

Runtimes for PLCP and LCP array construction are given in Table 4. The runtime of a fast SA construction algorithm[4], is included as a reference. Of the PLCP/LCP algorithms there are several interesting pairings to consider.

Φ vs. klaap. Overall, Φ is clearly the fastest route to the LCP array, being around 1.5 times faster than klaap on all inputs, and roughly 2.2 times faster if one stops at the PLCP. This is no doubt due better locality of memory reference: at any time Φ only access one array in a non-sequential fashion, whereas klaap makes random accesses to two arrays throughout its execution. Φ also consistently shades method I.

[3] Available at `http://web.unipmn.it/~manzini/lightweight/corpus/`

[4] Available at `http://www.michael-maniscalco.com/msufsort.htm`

Table 3. Algorithms and their space requirements. The space requirements do not include arrays that all algorithms share, i.e., text and SA for PLCP construction and text, SA and LCP for LCP construction.

Alg.	PLCP Space	LCP Space	PLCP representation	Description
Φ	n words	n words	full	Φ algorithm
Φx	n/x words	n/x words	sparse ($q = x$)	Φ algorithm
Φip	n words	0	full	Φ using in-place permutation in LCP construction
I	n words	n words	full	Irreducible lcp algorithm
IB	$3n$ bits	$3n$ bits	bitarray	Irreducible lcp algorithm
klaap		n words		Kasai et al [8]
m9		0		Manzini's Lcp9 [14]
pt$x$		$3n/\sqrt{x}$ words		Puglisi and Turpin [18]

Table 4. Runtimes (in milliseconds) for the various LCP construction algorithms

Alg.	sprot	rfc	linux	jdk13	etext	chr22	gcc	w3c	Avg.
Φ PLCP	13.57	14.22	13.12	6.64	15.86	4.89	9.41	9.98	10.96
I PLCP	20.46	20.85	18.64	10.52	23.24	7.91	13.37	15.37	16.30
Φ16 PLCP	3.11	3.58	3.17	1.66	3.41	1.11	2.32	2.47	2.60
Φ32 PLCP	2.45	2.64	2.55	1.45	2.47	0.81	1.88	2.14	2.05
Φ64 PLCP	2.16	2.27	2.23	1.33	2.06	0.67	1.65	1.98	1.79
IB PLCP	17.78	18.40	16.33	6.41	23.91	9.51	10.84	9.68	14.11
Φ LCP	20.55	21.39	19.99	10.62	23.21	6.87	14.24	15.93	16.60
Φip LCP	37.25	39.11	38.31	20.80	37.93	10.88	26.93	32.03	30.41
Φ16 LCP	35.71	41.25	33.01	17.66	43.93	13.94	23.69	25.45	29.33
Φ32 LCP	33.90	37.87	31.36	17.64	36.75	11.77	22.48	25.87	27.21
Φ64 LCP	36.77	37.53	32.07	20.54	34.36	10.55	23.23	29.75	28.10
IB LCP	104.93	108.83	89.27	46.48	123.60	31.22	57.74	63.51	78.20
klaap	34.10	32.70	28.67	16.01	38.29	10.94	21.24	26.23	26.02
m9	54.59	53.86	44.34	27.11	59.76	16.63	32.95	43.02	41.53
pt64	56.50	53.40	46.11	42.45	42.22	11.14	35.03	62.05	43.61
pt256	49.96	47.59	42.10	45.48	41.26	9.96	33.60	67.64	42.20
SA	42.42	38.31	36.09	24.85	44.22	12.83	26.52	35.08	32.54

Φip vs. m9. These algorithms use no extra space for LCP construction. The significantly faster speed of Φip can be largely attributed to an optimization that exploits the out-of-order execution capabilities of the CPU. When performing the in-place permutation, Φip follows multiple chains simultaneously. Execution alternates between the active chains allowing the CPU to proceed with one chain while waiting for a cache miss on another.

Φx and IB vs. pt$x$. These algorithms use very little extra space. Φx is clearly the fastest of these algorithms, nearly as fast as klaap, in fact. On the other hand, IB is quite slow when computing the LCP array due to the slowness of the select structure. We have not spent much time optimizing the implementation and a significant speed-up may be possible.

Φx, IB and pt$x$ can effectively work with disk resident SA and LCP. Then only the text and the space in Table 3 need to reside in primary memory. We have implemented such *semi-external* versions of the Φx algorithms. Similar to results in [18] we found that the sequential access to SA and LCP of these algorithms meant that runtimes increased by at most 10% (from those in Table 4). For brevity, we do not report actual times here. However, we remark that the semi-external version of $\Phi 64$ constructs the LCP array for a 1Gb prefix of the Human Genome in 526 seconds (ie. under 10 minutes) on our test machine, and allocates, including the space for the text, just 1.06Gb of RAM. For the same file pt64 requires 595 seconds and allocates 2.72Gb.

6 Concluding Remarks

In this paper we have investigated the PLCP array, a simple alternative representation of lcp values in which the values appear in position order, rather than lexicographical order, as they do in the LCP array. The PLCP array is very fast to construct, offers interesting space/time tradeoffs and can be used to simulate the LCP array or efficiently construct a full representation of it.

A drawback of current LCP construction algorithms, including those we describe here, is that they require primary memory at least equal to the size of the input text, so that the random accesses to it do not become expensive disk seeks. An I/O efficent algorithm for LCP array construction is an important direction for future work. I/O efficient algorithms for SA construction are described in [2].

Acknowledgements. Some of the key ideas in this paper were originated by Dmitry Khmelev who tragically died at young age in 2004 [9]. In particular, Theorem 1 is his conjecture [10] and the idea of using the sparse PLCP array comes from his algorithm [11].

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2, 53–86 (2004)
2. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM Journal of Experimental Algorithmics 12, 1–24 (2008)
3. 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, 236–280 (1999)
4. Fischer, J., Mäkinen, V., Navarro, G.: An(other) entropy-bounded compressed suffix tree. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 152–165. Springer, Heidelberg (2008)
5. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)

6. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science 387, 249–257 (2007)
7. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
8. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
9. Kelbert, A.: Memorial website of Dima Khmelev (2006), `http://mgg.coas.oregonstate.edu/~anya/dima/index-eng.html`
10. Khmelev, D.: Personal communication (2004)
11. Khmelev, D.: Program lcp version 0.1.9 (2004), `http://www.math.toronto.edu/dkhmelev/PROGS/misc/lcp-eng.html`
12. Mäkinen, V.: Compact suffix array — a space efficient full-text index. Fundamenta Informaticae 56, 191–210 (2003); Special Issue - Computing Patterns in Strings
13. Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22, 935–948 (1993)
14. Manzini, G.: Two space saving tricks for linear time LCP computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39 (2007)
16. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX 2007). SIAM, Philadelphia (2007)
17. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39, 1–31 (2007)
18. Puglisi, S.J., Turpin, A.: Space-time tradeoffs for Longest-Common-Prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008)
19. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 225–232. ACM/SIAM (2002)
20. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 294–313 (2003)
21. Sinha, R., Puglisi, S.J., Moffat, A., Turpin, A.: Improving suffix array locality for fast pattern matching on disk. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 661–672. ACM Press, New York (2008)
22. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th annual Symposium on Foundations of Computer Science, pp. 1–11 (1973)

Periodic String Comparison

Alexander Tiskin[⋆]

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

Abstract. In our previous work, we introduced the concept of semi-local string comparison, and developed for it an efficient method called the seaweed algorithm. In the current paper, we introduce its extension, called the periodic seaweed algorithm. The new algorithm allows efficient exploitation of the periodic structure in one of the input strings. By application of the periodic seaweed algorithm, we obtain new algorithms for the tandem LCS problem and the tandem cyclic alignment problem, improving on existing algorithms in running time.

1 Introduction

In [14,15], we introduced the concept of semi-local string comparison, and developed for it an efficient method called the seaweed algorithm. For completeness, we give a summary of our approach in Sections 2–6.

Algorithms on periodic strings is an important sub-area of string algorithms, motivated in part by applications to computational molecular biology. In the current paper, we introduce an extension of the seaweed algorithm, called the periodic seaweed algorithm. The new algorithm, described in Section 7, allows efficient exploitation of the periodic structure in one of the input strings.

Two particular algorithmic problems in periodic string comparison are the tandem LCS problem and the tandem cyclic alignment problem. We give full definitions of these problems in Section 8. By application of the periodic seaweed algorithm, we obtain new algorithms for these problems, improving on existing algorithms in running time, and answering a question by Landau [6].

Due to space constraints, we omit some proofs, which can be found in the full version of this paper [16].

2 Terminology and Notation

In addition to integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, we will use *odd half-integers* $\{\ldots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots\}$. For ease of reading, odd half-integer variables will be indicated by hats (e.g. $\hat{\imath}$, $\hat{\jmath}$). Ordinary variable names (e.g. i, j, with possible subscripts or superscripts), will normally indicate integer variables, but can sometimes indicate a variable that may be either integer, or odd half-integer.

[⋆] Research supported by the Centre for Discrete Mathematics and Its Applications (DIMAP), University of Warwick.

We denote integer and odd half-integer *intervals* by

$$[i : j] = \{i, i+1, \ldots, j-1, j\}$$
$$\langle i : j \rangle = \bigl\{i + \tfrac{1}{2}, i + \tfrac{3}{2}, \ldots, j - \tfrac{3}{2}, j - \tfrac{1}{2}\bigr\}$$

Note that in this notation, both an integer and an odd half-integer interval is defined by integer endpoints. To denote infinite intervals of integers and odd half-integers, we will use $-\infty$ for i and $+\infty$ for j where appropriate, so e.g. $[-\infty : +\infty]$ denotes the set of all integers, and $\langle -\infty : +\infty \rangle$ the set of all odd half-integers.

When dealing with pairs of numbers, we will often use geometrical language and call such pairs *points*. When visualising points, we use the matrix indexing convention: the first coordinate in a pair increases downwards, and the second coordinate rightwards. We say that a point (i_0, j_0) *dominates* point (i, j), if $i_0 < i$ and $j < j_0$. Visually, the dominated point is "below and to the left" of the dominating point.

We will make extensive use of finite and infinite matrices, with integer elements, and with integer or odd half-integer indices. Unless indicated otherwise, all definitions below apply to both finite and infinite matrices. Given finite or infinite index ranges I, J, a *matrix over* $I \times J$ is indexed by $i \in I$ and $j \in J$. A matrix is *nonnegative*, if all its elements are nonnegative.

We will use parenthesis notation for indexing matrices, e.g. $A(i, j)$. We will use straightforward notation for selecting subvectors and submatrices; for example, given a matrix A over $[-\infty : +\infty]^2$, we denote by $A[i_0 : i_1, j_0 : j_1]$ the submatrix defined by the given intervals. A star $*$ will indicate that for a particular index, its whole range is selected implicitly, e.g. $A[*, j_0 : j_1] = A[-\infty : +\infty, j_0 : j_1]$.

The matrices we consider (in particular, infinite matrices) can be *implicit*, i.e. represented by a compact data structure that allows access to every element in a specified (not necessarily constant) time.

Definition 1. *Let D be a matrix over $\langle i_0 : i_1 \rangle \times \langle j_0 : j_1 \rangle$. Its* distribution matrix *D^Σ over $[i_0 : i_1] \times [j_0 : j_1]$ is defined by*

$$D^\Sigma(i, j) = \sum\nolimits_{\hat\imath > i, \hat\jmath < j} D(\hat\imath, \hat\jmath)$$

for all $i \in [i_0 : i_1]$, $j \in [j_0 : j_1]$, $\hat\imath \in \langle i_0 : i_1 \rangle$, $\hat\jmath \in \langle j_0 : j_1 \rangle$.

Definition 2. *Let A be a matrix over $[i_0 : i_1] \times [j_0 : j_1]$. Its* density matrix *$A^\square$ over $\langle i_0 : i_1 \rangle \times \langle j_0 : j_1 \rangle$ is defined by*

$$A^\square(\hat\imath, \hat\jmath) = A\bigl(\hat\imath + \tfrac{1}{2}, \hat\jmath - \tfrac{1}{2}\bigr) - A\bigl(\hat\imath - \tfrac{1}{2}, \hat\jmath - \tfrac{1}{2}\bigr) - A\bigl(\hat\imath + \tfrac{1}{2}, \hat\jmath + \tfrac{1}{2}\bigr) + A\bigl(\hat\imath - \tfrac{1}{2}, \hat\jmath + \tfrac{1}{2}\bigr)$$

for all $\hat\imath \in \langle i_0 : i_1 \rangle$, $\hat\jmath \in \langle j_0 : j_1 \rangle$.

The definitions of distribution and density matrices extend naturally to matrices over an infinite index range, as long as the sum in Definition 1 is always defined. This will be the case for all matrices considered in this work.

Definition 3. *Matrix A will be called* simple*, if* $\left(A^{\square}\right)^{\Sigma} = A$.

Definition 4. *Matrix A is called* a Monge matrix*, if $A^{\square}$ is nonnegative.*

3 Permutation and Unit-Monge Matrices

A *permutation matrix* is a (finite or infinite) square zero-one matrix, containing exactly one nonzero in every row and every column. Typically, permutation matrices will be over odd half-integer intervals. An *identity matrix* is a (finite or infinite) permutation matrix Id, such that $Id(\hat{\imath}, \hat{\jmath}) = 1$, iff $\hat{\imath} = \hat{\jmath}$. Further, given an $h \in [-\infty : +\infty]$, we define an infinite *offset identity matrix* as a permutation matrix Id_h, such that $Id_h(\hat{\imath}, \hat{\jmath}) = 1$, iff $\hat{\jmath} - \hat{\imath} = h$. We have $Id_0 = Id$. Clearly, a finite or infinite identity or offset identity matrix, can be represented implicitly in constant space and with constant query time.

An infinite permutation matrix P over $\langle -\infty : +\infty \rangle^2$ has *core* $\langle i_0 : i_1 \rangle \times \langle j_0 : j_1 \rangle$ and *offset* h, for given $i_0, i_1, j_0, j_1, h \in [-\infty : +\infty]$, if

$$\begin{gathered} i_1 - i_0 = j_1 - j_0 \\ j_0 - i_0 = j_1 - i_1 = h \\ P(\hat{\imath}, \hat{\jmath}) = Id(\hat{\imath} - h, \hat{\jmath}) = Id(\hat{\imath}, \hat{\jmath} + h) \end{gathered}$$

for all $\hat{\imath} \in \langle -\infty : i_0 \rangle \cup \langle i_1 : +\infty \rangle$, $\hat{\jmath} \in \langle -\infty : j_0 \rangle \cup \langle j_1 : +\infty \rangle$. In particular, an offset identity matrix Id_h has empty core and offset h. Clearly, a permutation matrix with a finite core can be represented implicitly in finite space and with constant query time.

From now on, instead of "index pairs corresponding to nonzeros", we will write simply "nonzeros", where this does not lead to confusion. We will normally assume that a permutation matrix is given by an efficient data structure that allows random access to the nonzeros both by rows and by columns.

Given a permutation matrix P over $I \times J$, and a set $I' \subseteq I$, we will denote by $P(I', \cdot)$ the permutation submatrix *row-induced by* I', i.e. the permutation submatrix obtained by deleting from P all columns in $I \setminus I'$, and then deleting from the remaining submatrix all zero rows. A column-induced permutation submatrix $P(\cdot, J')$ is defined analogously. Both these operations can be implemented in linear time by a sweep of the nonzeros of matrix P.

Definition 5. *A square matrix A is called* a unit-Monge matrix*, if $A^{\square}$ is a permutation matrix.*

Matrices that are both unit-Monge and simple will be our main tool for the rest of this work. Note that a square matrix A is simple unit-Monge, if and only if $A = P^{\Sigma}$, where P is a permutation matrix. The value $A(i_0, j_0) = P^{\Sigma}(i_0, j_0)$ is the number of (odd half-integer) nonzeros that the (integer) point (i_0, j_0) dominates in matrix P.

A permutation matrix P of size n can be regarded as an implicit representation of the simple unit-Monge matrix P^{Σ}. An individual element of P^{Σ} can be queried in time $O(n)$ by a single sweep of the nonzeros of P, counting those that are dominated. Thinking of the nonzeros of P as odd half-integer points in the plane, this procedure is known as geometric *dominance counting*.

Existing methods of computational geometry allow us to answer dominance counting queries much more efficiently than by a direct linear sweep, as long as a preprocessing of the point set is allowed. In this work, we will deal with *incremental queries*, which are given an element of an implicit simple unit-Monge matrix, and return the value of an adjacent element. This kind of query can be answered directly from the permutation matrix, without any preprocessing.

Theorem 1. *Given a permutation matrix P of size n, and the value $P^{\Sigma}(i,j)$, $i,j \in [0:n]$, the values $P^{\Sigma}(i\pm 1,j)$, $P^{\Sigma}(i,j\pm 1)$, where they exist, can be queried in time $O(1)$.*

Proof. Consider a query of the type $P^{\Sigma}(i+1,j)$; other query types are obtained by symmetry. Let $\hat{\jmath} \in \langle 0:n \rangle$ be such that $P(i+\frac{1}{2},\hat{\jmath}) = 1$; value $\hat{\jmath}$ can be obtained from the permutation representation of P in time $O(1)$. We have

$$P^{\Sigma}(i+1,j) = P^{\Sigma}(i,j) - \begin{cases} 1 & \text{if } \hat{\jmath} < j \\ 0 & \text{otherwise} \end{cases} \qquad \square$$

Incremental queries described by Theorem 1 can be used to answer *batch queries*, returning a set of elements in a row, column or diagonal of the implicit simple unit-Monge matrix. In particular, all elements in a given row, column or diagonal of matrix P^{Σ} can be obtained by a sequence of incremental queries in time $O(n)$, and a subset of r consecutive elements in time $O\big(r + \log^2 n\big)$.

4 Semi-Local LCS and Highest-Score Matrices

We will consider strings of characters taken from an alphabet. Two alphabet characters α, β *match*, if $\alpha = \beta$, and *mismatch* otherwise. We extend the alphabet by a special *wildcard character* '?', which by definition matches all the other characters in the alphabet. We denote by $\backsim$ (respectively, $\sim$) a string of wildcard characters extending infinitely to the left (respectively, right).

It will be convenient to index strings by odd half-integer, rather than integer indices, e.g. $a = \alpha_{\frac{1}{2}}\alpha_{\frac{3}{2}}\ldots\alpha_{m-\frac{1}{2}}$. We will index strings similarly to matrices, writing e.g. $a(\hat{\imath}) = \alpha_{\hat{\imath}}$, $a\langle i:j \rangle = \alpha_{i+\frac{1}{2}}\ldots\alpha_{j-\frac{1}{2}}$. String concatenation will be denoted by juxtaposition.

Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. Special cases of a substring are *a prefix* and *a suffix* of a string.

Definition 6. *Given strings a, b, the* longest common subsequence (LCS) problem *asks for the length of the longest string that is a subsequence of both a and b. We will call this length the* LCS score *of strings a, b.*

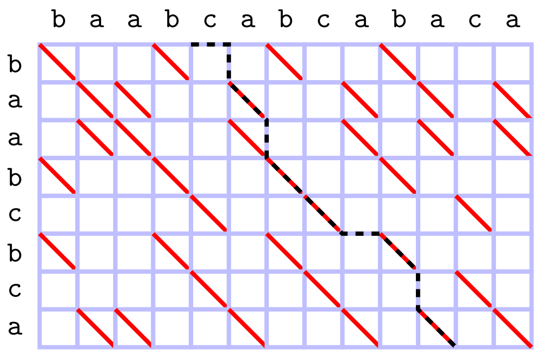


Fig. 1. Alignment dag $G_{a,b}$ and a highest-scoring path

Definition 7. *Given strings a, b, the* semi-local LCS problem *asks for the LCS scores as follows:*

- *a against every substring of b (the* string-substring LCS problem*);*
- *every prefix of a against every suffix of b (the* prefix-suffix LCS problem*);*
- *symmetrically, the* substring-string LCS problem *and the* suffix-prefix LCS problem*, defined as above but with the roles of a and b exchanged.*

Definition 8. *An* alignment dag *is a weighted dag, defined on the set of nodes $v_{l,i}$, $l \in [l_0 : l_1]$, $i \in [i_0, i_1]$. The edge and path weights are called* scores*. For all $l \in [l_0 : l_1]$, $\hat{l} \in \langle l_0 : l_1 \rangle$, $i \in [i_0, i_1]$, $\hat{\imath} \in \langle i_0 : i_1 \rangle$, the alignment dag contains:*

- *the horizontal edge $v_{l,\hat{\imath}-\frac{1}{2}} \to v_{l,\hat{\imath}+\frac{1}{2}}$ and the vertical edge $v_{\hat{l}-\frac{1}{2},i} \to v_{\hat{l}+\frac{1}{2},i}$, both with score 0;*
- *the diagonal edge $v_{\hat{l}-\frac{1}{2},\hat{\imath}-\frac{1}{2}} \to v_{\hat{l}+\frac{1}{2},\hat{\imath}+\frac{1}{2}}$ with score either 0 or 1.*

An alignment dag can be viewed as an $(l_1 - l_0) \times (i_1 - i_0)$ grid of *cells*. An instance of the semi-local LCS problem on strings a, b corresponds to an $m \times n$ alignment dag $G_{a,b}$; a cell indexed by $\hat{l} \in \langle 0 : m \rangle$, $\hat{\imath} \in \langle 0 : n \rangle$ is called a *match cell*, if $a(\hat{l}) = b(\hat{\imath})$, and a *mismatch cell* otherwise. The diagonal edges in match cells have score 1, and in mismatch cells score 0. Clearly, the diagonal edges with score 0 do not affect maximum node-to-node scores, and can therefore be ignored. Figure 1 shows the alignment dag corresponding to strings a = "`baabcbca`", b = "`baabcabcabaca`" (an example borrowed from [1]).

A particular example of an alignment dag is the *full-match dag*, which consists entirely of match cells.

The solution to the semi-local LCS problem is equivalent to finding the highest-scoring paths in the alignment dag.

Definition 9. *Consider an $m \times \infty$ alignment dag $G_{a,\backsim b\sim}$. The corresponding* highest-score matrix *is a matrix over $[-\infty : +\infty]^2$, defined by*

$$H_{a,\backsim b\sim}(i,j) = \max score(v_{0,i} \rightsquigarrow v_{m,j})$$

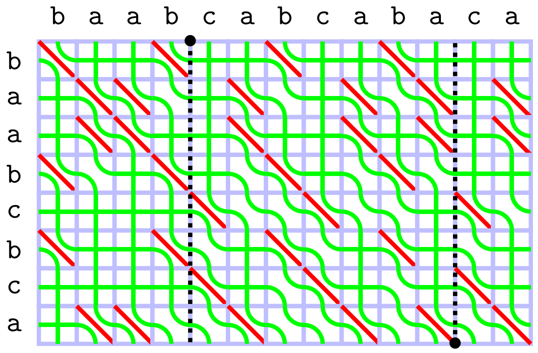


Fig. 2. Alignment dag $G_{a,b}$ and nonzeros of $P_{a,\backsim b\sim}$ as seaweeds

where $i, j \in [-\infty : +\infty]$, and the maximum is taken across all paths between the given endpoints. If $i = j$, we have $H_{a,\backsim b\sim}(i, j) = 0$. By convention, if $j < i$, then we let $H_{a,\backsim b\sim}(i, j) = j - i < 0$.

In Figure 1, the highlighted path has score 5, and corresponds to the value $H_{a,\backsim b\sim}(4, 11) = 5$, which is equal to the LCS score of string a against substring $b\langle 4 : 11\rangle$ = "`cabcaba`".

Theorem 2 ([14,15]). *Consider an $m \times \infty$ alignment dag $G_{a,\backsim b\sim}$ Its corresponding highest-score matrix $H_{a,\backsim b\sim}$ is unit-anti-Monge. In particular, we have*

$$H_{a,\backsim b\sim}(i, j) = j - i - P^{\Sigma}_{a,\backsim b\sim}(i, j)$$

where $P_{a,\backsim b\sim}$ is a permutation matrix over $\langle -\infty : +\infty\rangle^2$.

The key idea of our approach is to regard the highest-score matrix $H_{a,\backsim b\sim}$ as implicitly represented by the permutation matrix $P_{a,\backsim b\sim}$.

Figure 2 shows a graphical representation of the implicit highest-score matrix, given directly on the alignment dag. Nonzeros of $P_{a,\backsim b\sim}$ are represented by *seaweeds*, laid out as paths in the dual graph. In particular, every nonzero $P_{a,\backsim b\sim}(i, j) = 1$, where $i, j \in \langle 0 : n\rangle$, is represented by a seaweed originating between the nodes $v_{0,i-\frac{1}{2}}$ and $v_{0,i+\frac{1}{2}}$, and terminating between the nodes $v_{m,j-\frac{1}{2}}$ and $v_{m,j+\frac{1}{2}}$. The remaining seaweeds, originating or terminating at the sides of the dag, correspond to nonzeros $P_{a,\backsim b\sim}(i, j) = 1$, where either $i \in \langle -m : 0\rangle$ or $j \in \langle n : n + m\rangle$ (or both). In particular, every nonzero $P_{a,\backsim b\sim}(i, j) = 1$, where $i \in \langle -m : 0\rangle$ (respectively, $j \in \langle n : m + n\rangle$) is represented by a seaweed originating between the nodes $v_{-i-\frac{1}{2},0}$ and $v_{-i+\frac{1}{2},0}$ (respectively, terminating between the nodes $v_{m+n-j-\frac{1}{2},n}$ and $v_{m+n-j+\frac{1}{2},n}$). For the purposes of this section, the specific layout of the seaweeds between their endpoints is not important. However, this layout will become meaningful in the context of the algorithms described in the following sections.

5 Weighted Scores and Edit Distances

The concept of LCS score is generalised by that of *weighted alignment score* (see e.g. [5]). An *alignment* of strings a, b is obtained by putting a subsequence of a into one-to-one correspondence with a (not necessarily identical) subsequence of b, character by character and respecting the index order. The corresponding pair of characters, one from a and the other from b, are said to be *aligned.* A character not aligned with a character of another string is said to be aligned with a *gap* in that string. Hence, four types of character alignment arise, each of which is given a real *weight*:

- a pair of matching characters, with weight $w_=$;
- a pair of mismatching characters, with weight $w_\#$;
- a gap against a character, with weight $w_\dashv$;
- a character against a gap, with weight $w_\vdash$.

Some of these weights may be negative. Aligning a matching pair of characters is considered to be better than aligning a mismatching pair of characters, which in its turn is not worse than aligning each of the two characters against a gap. Therefore, we assume $w_= > w_\# \geq w_\dashv + w_\vdash$. In particular, the LCS score corresponds to taking $w_= = 1$, $w_\# = w_\dashv = w_\vdash = 0$.

Definition 10. *The* alignment score *for strings a, b is the maximum total weight of character alignments in an alignment of a and b.*

Clearly, the alignment score corresponds to a shortest path in a generalised alignment dag, where diagonal match, diagonal mismatch, horizontal and vertical edges have weight $w_=$, $w_\#$, $w_\dashv$, $w_\vdash$, respectively.

We show that without loss of generality, we can restrict ourselves to alignment scores with $w_= = 1$, $w_\dashv = w_\vdash = 0$. Indeed, given general weights, we solve the alignment score problem with *normalised weights*

$$w^*_= = 1 \qquad w^*_\# = \frac{w_\# - w_\dashv - w_\vdash}{w_= - w_\dashv - w_\vdash} \qquad w^*_\dashv = w^*_\vdash = 0$$

Then the score w of any alignment with the original weights can be found from the score w^* of the corresponding alignment with normalised weights as

$$w = w^* \cdot (w_= - w_\dashv - w_\vdash) + m \cdot w_\vdash + n \cdot w_\dashv$$

In this work, we restrict ourselves to alignment scores that satisfy the following rationality condition.

Definition 11. *A set of alignment score weights will be called* rational *if the corresponding normalised weights (in particular, $w^*_\#$) are rational numbers.*

Given a rational set of normalised weights, the semi-local alignment score problem on strings a, b can be easily reduced to the semi-local LCS problem. For details, see [16].

6 The Seaweed Algorithm

A classical solution to the global LCS problem is given by the dynamic programming algorithm, discovered independently by Needleman and Wunsch (without an explicit analysis) [11], and by Wagner and Fischer [17]. The dynamic programming algorithm runs in time $O(mn)$. Based on the ideas of Schmidt [12], Alves et al. [1] gave an algorithm for the string-substring LCS problem, also running in time $O(mn)$.

In [15], we gave a simple algorithm for the semi-local LCS problem, that matches the above algorithms in and asymptotic running time, and improves on them in functionality. We call it the *seaweed algorithm*, since it has a simple interpretation in terms of seaweeds (paths in the dual graph, see Figure 2).

Algorithm 1 (Semi-local LCS: the seaweed algorithm [15]).
Input: strings a, b of length m, n, respectively.
Output: implicit highest-score matrix P_A on strings a, b.
Description. The output permutation matrix $P_{a,\backsim b\sim}$ has core $\langle -m : n\rangle \times \langle 0 : m+n\rangle$ and offset m. We will maintain a variable matrix P with the same core and offset. Let initially $P \leftarrow Id_m$. We sweep the cells of the alignment dag in an arbitrary order compatible with the top-to-bottom and left-to-right partial order of the cells. For each cell, we perform an update on matrix P. At the end of the sweep, we will have $P = P_{a,\backsim b\sim}$.

Consider a cell indexed by $\hat{l} \in \langle 0 : m\rangle$, $\hat{\imath} \in \langle 0 : n\rangle$. We define the cell's *parameters* to be characters $a(\hat{l})$, $b(\hat{\imath})$. Let $i^* = \hat{\imath}+m-\hat{l}$. The update is performed on a 2×2 induced permutation submatrix of P as follows:

$$P\langle \cdot, i^*-1 : i^*+1\rangle \leftarrow \begin{cases} \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right) & \text{if } a(\hat{l}) \neq b(\hat{\imath}) \text{ and } P\langle \cdot, i^*-1 : i^*+1\rangle = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right) \\ \text{unchanged otherwise} \end{cases} \quad (1)$$

The current cell can be regarded as an automaton, performing the update on the submatrix from an *input state* into the *output state*.

The sequence of updates on matrix P can be interpreted as the following sequence of updates on the alignment dag. We start with a trivial full-match dag, which consists entirely of match cells. We then sweep the cells in the order described above. In each step, we transform a match cell into a mismatch cell, if the corresponding characters mismatch in the input strings. By Theorem 2, the algorithm maintains the invariant "current state of matrix P is the implicit highest-score matrix for the current state of the alignment dag". Therefore, at the end of the sweep, we have $P = P_{a,\backsim b\sim}$.

Cost analysis. For every cell, the 2×2 column-induced submatrix $P\langle \cdot, i^*-1 : i^*+1\rangle$ can be obtained from matrix P in time $O(1)$. The cell update also runs in time $O(1)$. Therefore, the overall running time is $O(mn)$. □

In the course of the computation by Algorithm 1, the semi-local LCS problem is solved implicitly for all prefixes of a against all prefixes of b. The algorithm can

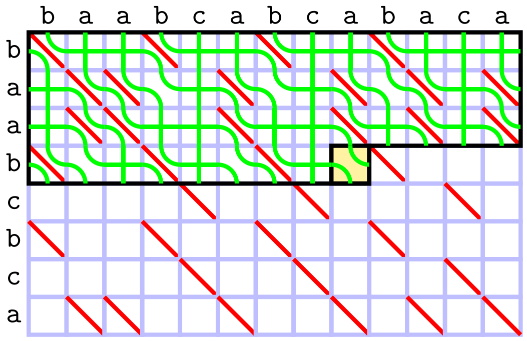


Fig. 3. A snapshot of Algorithm 1 (the seaweed algorithm)

be interpreted in terms of seaweeds as follows. Each seaweed is traced across the alignment dag in the top-to-bottom or left-to-right direction. A seaweed runs in a straight line by default; however, its direction may be affected by match cells, and by other seaweeds. Every cell has two seaweeds passing through it, one entering across the top edge and another across the left-hand edge. In a match cell, both seaweeds "bend away" from each other, so the seaweed entering at the top exits on the right, and the seaweed entering on the left exits at the bottom. In a mismatch cell, the two seaweeds keep straight and cross each other, if and only if this pair of seaweeds have not previously crossed; otherwise, they bend away as in a match cell. Therefore, any given pair of seaweeds are only allowed to cross at most once in the course of the computation. Notice that the same property of crossing at most once also holds for any pair of highest-scoring paths in the dag.

Figure 3 shows a snapshot of Algorithm 1. The dag area that has already been processed is shown by the dark border; the cell currently being processed is shaded. Since the two seaweeds crossing in the current cell have previously crossed, the current step will leave the implicit highest-score matrix unchanged, so that the second crossing is not allowed. The final layout of the seaweeds is the one given in Figure 2, which describes the full sequence of states of the implicit highest-score matrix in Algorithm 1.

7 The Periodic Seaweed Algorithm

In many string comparison applications, one or both of the input strings may have periodic structure. In this section, we show how to exploit such structure efficiently, using a variant of the seaweed method.

Consider the problem of comparing a finite string a of length m against a string b, which is infinite in both directions and *periodic*: $b = u^{\pm\infty} = \ldots uuuu \ldots$ The *period string* u is finite of length p.

Definition 12. *Given strings a, u, the* periodic string-substring LCS problem *asks for the LCS score of a against every finite substring of $b = u^{\pm\infty}$.*

Without loss of generality, we assume that every character of a occurs in u at least once. Clearly, the length of the substring of b in Definition 12 can be restricted to be at most mp.

The definition of the alignment dag (Definition 8) extends naturally to the periodic string-substring LCS problem. The alignment dag is itself *periodic*: the edges $v_{l,\hat{\imath}-\frac{1}{2}+kp} \to v_{l,\hat{\imath}+\frac{1}{2}+kp}$ (respectively, $v_{\hat{l}-\frac{1}{2},i+kp} \to v_{\hat{l}+\frac{1}{2},i+kp}$, $v_{\hat{l}-\frac{1}{2},\hat{\imath}-\frac{1}{2}+kp} \to v_{\hat{l}+\frac{1}{2},\hat{\imath}+\frac{1}{2}+kp}$) have equal scores for all $l \in [l_0 : l_1]$, $\hat{l} \in \langle l_0 : l_1 \rangle$, $i \in [i_0, i_1]$, $\hat{\imath} \in \langle i_0 : i_1 \rangle$, $k \in [-\infty : +\infty]$. Such an alignment dag can also be regarded as a horizontal composition of an infinite sequence of *period subdags*, each of which is isomorphic to the $m \times p$ alignment dag $G_{a,u}$.

Consider the highest-score matrix $H_{a,b}$ and its implicit representation $P_{a,b}$; note that, since string b is already infinite, it does not require any extension by wildcards. Matrices $H_{a,b}$, $P_{a,b}$ are again *periodic*: we have $H_{a,b}(i,j) = H_{a,b}(i+p, j+p)$ for all $i,j \in [-\infty : \infty]$, and $P_{a,b}(\hat{\imath},\hat{\jmath}) = H_{a,b}(\hat{\imath}+p, \hat{\jmath}+p)$ for all $\hat{\imath},\hat{\jmath} \in \langle -\infty : \infty \rangle$. To represent such matrices, it is sufficient to store the p nonzeros of the *horizontal period submatrix* $P_{a,b}\langle 0 : p, * \rangle$, or, symmetrically, of the *vertical period submatrix* $P_{a,b}\langle *, 0 : p \rangle$. The nonzero sets of the two period submatrices can be obtained from one another in time $O(p)$; we will be using both of them simultaneously where necessary.

The periodic string-substring LCS problem can be solved by a simple extension of the seaweed algorithm (Algorithm 1). Following the periodic structure of the highest-score matrix, the seaweed pattern is also periodic. Hence, the seaweeds only need to be traced within a single period subdag, with appropriate wraparound.

Algorithm 2 (Periodic string-substring LCS: the periodic seaweed algorithm).
Input: strings a, u of length m, p, respectively.
Output: implicit highest-score matrix $P_{a,b}$, represented by nonzeros of (say) vertical period submatrix $P_{a,b}\langle *, 0 : p \rangle$, where $b = u^{\pm\infty}$.
Description. The output matrix is periodic with period p. We will maintain a variable matrix P with the same period. We let initially $P \leftarrow Id_m$ (which is a periodic matrix). Then, we sweep the cells of the period subdag as follows. In the outer loop, we run through the rows of cells top-to-bottom. For the current row $\hat{l} \in \langle 0 : m \rangle$, we start the inner loop at an arbitrary match cell $\hat{\imath}_0 \in \langle 0 : p \rangle$, so we have $a(\hat{l}) = b(\hat{\imath})$. Such a match cell is guaranteed to exist by the assumption that every character of a occurs in u at least once. Then, we sweep the cells from $\hat{\imath} = \hat{\imath}_0$ left-to-right, wrapping around from $\hat{\imath} = p - \frac{1}{2}$ to $\hat{\imath} = \frac{1}{2}$, and continuing the sweep left-to-right up to $\hat{\imath} = \hat{\imath}_0 - 1$. For each cell, we perform an update on matrix P. At the end of the sweep, we will have $P = P_{a,b}$.

Consider a cell indexed by $\hat{l} \in \langle 0 : m \rangle$, $\hat{\imath} \in \langle 0 : p \rangle$. We define the cell's *parameters* to be characters $a(\hat{l})$, $b(\hat{\imath})$. Let $i^* = \hat{\imath} + m - l$. As in Algorithm 1, the update is performed on a 2×2 column-induced permutation submatrix of P by (1). Note that the first update in an inner loop is always trivial: we have $a(\hat{l}) = b(\hat{\imath}_0)$, therefore P remains unchanged.

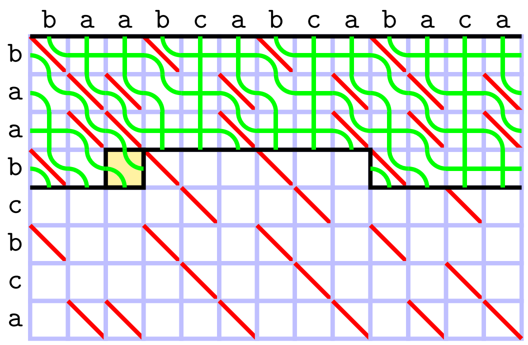


Fig. 4. A snapshot of Algorithm 2 (the periodic seaweed algorithm)

The sequence of updates on matrix P can be interpreted as a sequence of updates on the alignment dag, as described in Algorithm 1, but now including the wraparound. Therefore, at the end of the sweep, we have $P = P_{a,b}$.

Cost analysis. As in Algorithm 1, the total running time is $O(mp)$. □

Figure 4 shows a snapshot of Algorithm 2, using the same conventions as Figure 3.

Note that the cell updating order in Algorithm 2 is significantly more restricted than in Algorithm 1, due to the extra data dependencies caused by the wraparound. This seems to rule out the possibility of a micro-block version of the algorithm, that would be analogous to the micro-block seaweed algorithm [16].

8 Tandem Alignment

The periodic LCS problem has many variations that can be solved by an application of the periodic seaweed algorithm.

The first such variation is the *tandem LCS problem.* The problem asks for the LCS score of a string a of length m against a tandem k-repeat string $b = u^k$ of length $n = kp$. As before, we assume that every character of a occurs in u at least once. We may assume that $k \leq m$ (since for $k \geq m$, every character of a can be matched to a different copy of u in b, and therefore the LCS score between a and b is equal to m).

The tandem LCS problem can be solved naively by considering the LCS problem directly on strings a and b, ignoring the periodic structure of string b. The standard dynamic programming algorithm [11,17] solves the problem in time $O(mn) = O(mkp)$. This running time can be slightly improved by the micro-block precomputation method [9].

The tandem LCS problem can also be regarded as a special case of the common-substring LCS problem [8,3]. Using this technique, the problem can be solved in time $O\bigl(m(k + p)\bigr)$. The techniques of Landau et al. [3,7] give an algorithm for the tandem LCS problem, parameterised by the LCS score of the input strings; however, the worst-case running time of this algorithm is still $O\big(m(k+p)\big)$. Landau [6] asked if the running time for the tandem LCS problem can be improved to $O\big(m(\log k + p)\big)$.

We now give an algorithm that improves on the current algorithms in time and functionality, and even exceeds Landau's expectation. First, we call Algorithm 2 on strings a and u. Then, we count the number of nonzeros dominated by point $(0, n)$, i.e. nonzeros in the submatrix $P_{a,b}\langle 0 : +\infty, -\infty : n\rangle$. Given the (say) horizontal period submatrix $P_{a,b}\langle 0 : p, *\rangle$, this can be done by a sweep of its p nonzeros, counting every nonzero with appropriate multiplicity. More precisely, every nonzero $P_{a,b}(i, j) = 1$, $i \in \langle 0 : p\rangle$, $j \in \langle -\infty : \infty\rangle$, is counted with multiplicity $k - \lfloor j/p \rfloor$, if $j \in \langle 0 : n\rangle$, and is skipped (counted with multiplicity 0) otherwise. The solution to the tandem LCS problem is then obtained by Theorem 2. The overall running time is dominated by the call to Algorithm 2, which runs in time $O(mp)$.

Another set of variations on the periodic LCS problem was introduced by Benson [2] as the *tandem alignment problem.* Instead of asking for all string-substring LCS scores of a against $b = u^{\pm\infty}$, the tandem alignment problem asks for a substring of b that is closest to a in terms of alignment score (or edit distance), under different restrictions on the substring. In particular:

- the *pattern global, text global (PGTG) tandem alignment problem* restricts the substring of b to consist of a whole number of copies of u, i.e. to be of the form $u^k = uu \ldots u$ for an arbitrary integer k;
- the *tandem cyclic alignment problem* restricts the substring of b to be of length kp for an arbitrary integer k (but it may not consist of a whole number of copies of u);
- the *pattern local, text global (PLTG) tandem alignment problem* leaves the substring of b unrestricted.

All three versions of the tandem alignment problem can be regarded as special cases of the approximate pattern matching problem on strings a of length m and $b' = u^m$ of length $n = kp$ (but with the roles of the text and the pattern reversed). Therefore, the tandem LCS problem can be solved naively by considering the approximate pattern matching problem directly on strings a and b', ignoring the periodic structure of string b'. Given an arbitrary (real) set of alignment weights, the classical algorithm by Sellers [13] solves the problem in time $O(mn) = O(mkp)$. For a rational set of weights, the running time can again be slightly improved by the micro-block precomputation method.

The PGTG and PLTG tandem alignment problems can be solved more efficiently by the technique of *wraparound dynamic programming* [10,4] (see also [2]) in time $O(mp)$. For the tandem cyclic alignment problem, Benson [2] modified this technique to give an algorithm running in time $O(mp \log p)$ and memory $O(mp)$.

We now give a new algorithm for the tandem cyclic alignment problem, which improves on the existing algorithm in running time, assuming a rational set of alignment weights. The running time of the new algorithm matches the current algorithms for the PGTG and PLTG tandem alignment problems.

Given input strings a, u, we first solve the periodic string-substring problem by calling Algorithm 2. This gives us a period submatrix of matrix $P_{a,b}$, where $b = u^{\pm\infty}$. Then, for each k, $0 < k < m$, we perform independently the following procedure. We solve the tandem LCS problem for strings a and u^k by the method described earlier in this section, counting every nonzero in the period submatrix $P_{a,b}$ with an appropriate multiplicity. This gives us the LCS score for a against u^k for every k. We then update this score incrementally, obtaining the LCS score for string a against a window of length p in b, sliding through p successive positions. This is equivalent to querying p successive elements in a diagonal of matrix $P_{a,b}$, which can be achieved by $2p$ incremental dominance counting queries. By Theorem 1, every one of these queries can be performed in time $O(1)$.

More precisely, let $P^{\Sigma}_{a,b}(i, i+kp)$, $i \in [0 : p]$, be the current query element. Let $\hat{\imath}_0 = i + \frac{1}{2}$, $\hat{\jmath}_1 = i + kp + \frac{1}{2}$. Let $\hat{\jmath}_0$, $\hat{\imath}_1$ be such that $P_{a,b}(\hat{\imath}_0, \hat{\jmath}_0) = P_{a,b}(\hat{\imath}_1, \hat{\jmath}_1) = 1$. The value $\hat{\jmath}_0$ (respectively, $\hat{\imath}_1$) can be obtained from the horizontal (respectively, vertical) period submatrix of $P_{a,b}$ in time $O(1)$. Then the next query element is

$$P^{\Sigma}_{a,b}(i+1, i+kp+1) = P^{\Sigma}_{a,b}(i, i+kp) - \begin{cases} 1 & \text{if } \hat{\jmath}_0 < i+kp \\ 0 & \text{otherwise} \end{cases} + \begin{cases} 1 & \text{if } \hat{\imath}_1 > i+1 \\ 0 & \text{otherwise} \end{cases}$$

The call to Algorithm 2 runs in time $O(mp)$; its output is shared by the tandem LCS computation for all k. For each k, the running time of both the remaining part of the tandem LCS computation and the sliding window (batch query) computation is $O(p)$; therefore, the combined running time for all values of k is $m \cdot O(p) = O(mp)$. Overall, the algorithm runs in time $O(mp)$.

9 Conclusions

In this work, we have introduced a new method for periodic string-substring comparison. By application of the new method, we have obtained improved algorithms for the tandem LCS problem and the tandem cyclic alignment problem. Since string periodicity is a fundamental concept in many areas of computer science and computational molecular biology, it is likely that our method will have other interesting applications.

Acknowledgement

The periodic string comparison problem was introduced to the author by Gad Landau. The author thanks the anonymous referees, whose comments helped to improve this work.

References

1. Alves, C.E.R., Cáceres, E.N., Song, S.W.: An all-substrings common subsequence algorithm. Discrete Applied Mathematics 156(7), 1025–1035 (2008)
2. Benson, G.: Tandem cyclic alignment. Discrete Applied Mathematics 146(2), 124–133 (2005)
3. Crochemore, M., Landau, G.M., Schieber, B., Ziv-Ukelson, M.: Re-use dynamic programming for sequence alignment: An algorithmic toolkit. In: String Algorithmics. Texts in Algorithmics, vol. 2. College Publications, King's (2004)
4. Fischetti, V.A., Landau, G.M., Sellers, P.H., Schmidt, J.P.: Identifying periodic occurrences of a template with applications to protein structure. Information Processing Letters 45(1), 11–18 (1993)
5. Jackson, B.N., Aluru, S.: Pairwise sequence alignment. In: Handbook of Computational Molecular Biology. Chapman and Hall/CRC Computer and Information Science Series, ch. 1, pp. 1–1 – 1–31. Chapman and Hall/CRC, Boca Raton (2006)
6. Landau, G.: Can DIST tables be merged in linear time? An open problem. In: Proceedings of the Prague Stringology Conference, p. 1. Czech Technical University in Prague (2006)
7. Landau, G.M., Myers, E., Ziv-Ukelson, M.: Two algorithms for LCS consecutive suffix alignment. Journal of Computer and System Sciences 73(7), 1095–1117 (2007)
8. Landau, G.M., Ziv-Ukelson, M.: On the common substring alignment problem. Journal of Algorithms 41(2), 338–359 (2001)
9. Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. Journal of Computer and System Sciences 20, 18–31 (1980)
10. Myers, E.W., Miller, W.: Approximate matching of regular expressions. Bulletin of Mathematical Biology 51(1), 5–37 (1989)
11. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48(3), 443–453 (1970)
12. Schmidt, J.P.: All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. SIAM Journal on Computing 27(4), 972–992 (1998)
13. Sellers, P.H.: The theory and computation of evolutionary distances: Pattern recognition. Journal of Algorithms 1(4), 359–373 (1980)
14. Tiskin, A.: Semi-local longest common subsequences in subquadratic time. Journal of Discrete Algorithms 6(4), 570–581 (2008)
15. Tiskin, A.: Semi-local string comparison: Algorithmic techniques and applications. Mathematics in Computer Science 1(4), 571–603 (2008)
16. Tiskin, A.: Semi-local string comparison: Algorithmic techniques and applications. Technical Report 0707.3619, arXiv (2009)
17. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM 21(1), 168–173 (1974)

Deconstructing Intractability: A Case Study for Interval Constrained Coloring

Christian Komusiewicz[⋆], Rolf Niedermeier, and Johannes Uhlmann[⋆⋆]

Institut für Informatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, D-07743 Jena, Germany
{c.komus,rolf.niedermeier,johannes.uhlmann}@uni-jena.de

Abstract. The NP-hard Interval Constrained Coloring problem appears in the interpretation of experimental data in biochemistry dealing with protein fragments. Given a set of m integer intervals in the range 1 to n and a set of m associated multisets of colors (specifying for each interval the colors to be used for its elements), one asks whether there is a "consistent" coloring for all integer points from $\{1, \ldots, n\}$ that complies with the constraints specified by the color multisets. We initiate a study of Interval Constrained Coloring from the viewpoint of combinatorial algorithmics, trying to avoid polyhedral and randomized rounding methods as used in previous work. To this end, we employ the method of systematically deconstructing intractability. It is based on a thorough analysis of the known NP-hardness proof for Interval Constrained Coloring. In particular, we identify numerous parameters that naturally occur in the problem and strongly influence the problem's practical solvability. Thus, we present several positive (fixed-parameter) tractability results and, moreover, identify a large spectrum of combinatorial research challenges for Interval Constrained Coloring.

1 Introduction

Althaus et al. [1, 2] recently identified the Interval Constrained Coloring problem as an important combinatorial problem in the context of automated mass spectrometry and the determination of the tertiary structure of proteins. It builds the key to replace a manual interpretation of exchange data for peptic fragments with computer-assisted methods, see Althaus et al. [2] for more on the biochemical background and further motivation. The decision problem Interval Constrained Coloring (ICC) deals with matching color multisets with integer intervals and can be formalized as follows.[1] To this end, for two positive integers i, j with $i \leq j$, let $[i, j] := \{k \in \mathbb{N} \mid i \leq k \leq j\}$. Further, for $i \geq 1$ let $[i]$ denote the interval $[1, i]$.

[⋆] Supported by a PhD fellowship of the Carl Zeiss Foundation.

[⋆⋆] Supported by the DFG, research project PABI, NI 369/7.

[1] Note that, compared with Althaus et al. [1, 2], we choose a somewhat different but equivalent formalization here; this problem definition turns out to be more suitable for our subsequent studies.

Input: A positive integer n, a set of m integer intervals $\mathcal{F} = \{F_1, \ldots, F_m\}$, all within $[n]$, a multiset of m multisets of colors $\mathcal{C} = \{C_1, \ldots, C_m\}$ over k different colors.
Question: Is there a coloring $c : [n] \to [k]$ such that for each interval $F_i \in \mathcal{F}$ it holds that $C_i = c(F_i)$?

Herein, $c(F_i)$ denotes the *multiset* of colors assigned by c to the integer points in the interval F_i. Concerning the biochemical background, the intervals correspond to (typically overlapping) fragments of a protein with n residues, and the k colors correspond to k different exchange rates that need to be assigned consistently to the n residues [1, 2]. The color multisets correspond to experimentally found bulk information that needs to be matched with the residues and can be interpreted as constraints that describe a set of valid colorings of the interval $[n]$ or, alternatively, a set of strings of length n over the "color alphabet". Note that from an applied point of view it is also important to investigate the corresponding optimization problems where one wants to maximize the number of requirements (that is, intervals that completely match with a given color multiset) that can be fulfilled [1]. However, in this paper we focus on analyzing the complexity of the decision problem.

Known results. To our knowledge, so far ICC has only been studied in the two papers by Althaus et al. [1, 2]. It has been shown to be NP-complete by a reduction from the Exact Cover problem [1]. In the more applied paper [2], besides first introducing and formalizing the problem, an algorithm based on integer linear programming and branch-and-bound was presented that enumerates all valid (fulfilling all constraints) color mappings c. In particular, it was shown that in the case of $k = 2$ colors a direct combinatorial algorithm leads to polynomial-time solvability; the computational complexity of the case $k = 3$ was left open. Finally, successful experiments with real-world instances with $n < 60$, $m \leq 50$, $k = 3$ and randomly generated instances with $n \leq 1000$, $m = n/2$, and $k = 3$ have been performed. In the more theoretical paper [1], besides the NP-completeness proof, the preceding work [2] has been continued by providing results concerning polynomial-time approximability. In particular, there is an algorithm producing a coloring where all requirements are matched within a mere additive error of one if the LP-relaxation of the presented integer program for ICC has a feasible solution. This algorithm is based on a sophisticated polyhedral approach combined with recent randomized rounding techniques.

Our contributions. This work proposes a fresh view on ICC and the development of exact algorithms for NP-hard combinatorial problems in general. The fundamental starting point here is to deconstruct proofs of NP-hardness in order to obtain new insights into the combinatorial structure of problems. More specifically, the point is to analyze how different parameters occurring in a problem contribute to its computational complexity. Having identified (some of) these parameters, the next step is to determine the complexity behavior in dependence on these parameters and combinations thereof. This is where parameterized algorithmics [5, 6, 8] comes into play. In case of ICC, there is an enormous number of useful parameterizations, all naturally deduced from deconstructing the known NP-hardness proof. In this line, for instance, we can show a fixed-parameter tractability result with respect to the parameter "maximum interval length". Whereas we do not know whether the problem is fixed-parameter tractable with respect to the color parameter k alone, it is with respect to the combined parameter (n, k), that is, there is an algorithm with time complexity of the form $O^*((k-1)^n)$.[2] These algorithms are of practical interest when the corresponding parameter values are sufficiently small. For instance, note that all experiments of Althaus et al. [2] were performed having $k = 3$ and $n \leq 60$ for real-world instances. Indeed, in case of $k = 3$ we can further improve the running time to $O^*(1.89^n)$. In this spirit, in Section 3 we investigate a number of "single parameterizations", and in Section 4 we consider an even larger number of "combined parameterizations". Moreover, whereas ICC is NP-complete for "cutwidth" three [1], we present a combinatorial polynomial-time algorithm for cutwidth two. Tables 1 and 2 in Sections 3 and 4 survey the current state of the art and our new results concerning (combinatorial) algorithms that can efficiently solve ICC in case of favorable parameter constellations.

Due to the lack of space, several proofs are deferred to the long version of this paper.

2 Parameterization and Deconstruction of NP-Hardness

Parameterized algorithmics [5, 6, 8] aims at a multivariate complexity analysis of problems. The hope lies in accepting the seemingly inevitable combinatorial explosion for NP-hard problems, but to confine it to some parameter p. In this paper, p always is a positive integer or a vector of positive integers. A given parameterized problem (I, p) is *fixed-parameter tractable (FPT)* with respect to the parameter p if it can be solved within running time $f(p) \cdot \text{poly}(|I|)$ for some computable function f only depending on p.

A standard question of people unfamiliar with parameterized algorithmics is how to define respectively find "the" parameter for an NP-hard problem. There are the following (partly overlapping) "standard answers" to this question:

1. The standard parameterization typically refers to the size of the solution set of the underlying problem (whenever applicable).
2. A parameter describes a structural property of the input; for instance, the treewidth of a graph or the number of input strings.
3. Finding useful parameters to some extent is an "art" based on analyzing what typical real-world instances could look like.

Perhaps the most natural and constructive answer, however, is to look at the corresponding proof(s) of NP-hardness and what "parameter assumptions" they (do not) make use of. Indeed, this is nothing but what we mean by *deconstructing NP-hardness proofs for parameter identification*. In this work, we deconstruct the (only known) NP-hardness proof of ICC and gain a rich scenario of combinatorially and practically interesting structural parameterizations.

[2] The O^*-notation suppresses polynomial-time factors [9].

Let us now take a closer look at ICC. We first have to briefly review the known NP-hardness reduction from Exact Cover due to Althaus et al. [1]: The input of Exact Cover is a set $\mathcal{S}$ of subsets of a ground set $U := \{1, 2, \ldots, u\}$ and a positive integer t, and the question is whether there are t subsets from $\mathcal{S}$ such that every element from U is contained in exactly one such subset. Althaus et al.'s polynomial-time many-one reduction (using an approach by Chang et al. [4]) from Exact Cover to ICC works as follows.

1. The number of colors k is set to $s := |\mathcal{S}|$.
2. The interval range n is set to $(u + 1) \cdot s - t$.
3. For each element from U, there are exactly three corresponding integer intervals. Indeed, one can speak of three types of intervals, and all intervals of one type can be placed consecutively into one interval $[n]$ without overlap.
   (a) Type 1: Intervals of the form $[(i - 1)s + 1, is]$ for all $1 \leq i \leq u$.
   (b) Type 2: Intervals of the form $[is - t + 1, (i + 1)s - t]$ for all $1 \leq i \leq u$.
   (c) Type 3: Intervals of the form $[is - t - f_i + 1, is - t + 1]$ for all $1 \leq i \leq u$, where f_i denotes the number of occurrences of i in the sets of $\mathcal{S}$.
4. Every type-1 and every type-2 interval is assigned the color set $\{1, \ldots, k\}$. A type-3 interval corresponding to $i \in U$ is assigned the color set consisting of the colors associated with the subsets in $\mathcal{S}$ that contain i.

After having described the construction behind the NP-hardness proof, the deconstruction begins by making several observations about its properties:

1. The interval range n and the number m of intervals both are unbounded.
2. The number of colors k is s, hence unbounded, but all color multisets indeed are sets. That is, no interval shall be assigned the same color twice.
3. The maximum interval length is s, hence unbounded.
4. The maximum overlap between intervals is $\max\{t, s - t\}$, hence unbounded.
5. Only three different surrounding intervals $[n]$ are needed for comprising all intervals without overlap, hence the *cutwidth* of the constructed instance is bounded by three.

From the fifth observation we can conclude that there is no hope for fixed-parameter tractability with respect to the parameter "cutwidth" unless P=NP. On the positive side, we will show that ICC is polynomial-time solvable for cutwidth two. However, from the other four observations we directly obtain the following questions concerning a parameterized complexity analysis of ICC.

1. Is ICC fixed-parameter tractable with respect to the parameters n or m?
2. Is ICC fixed-parameter tractable with respect to k, or is it already NP-hard for constant k-values? Indeed, the complexity for the practically relevant case $k = 3$ is still unsettled. How does the parameter "maximum number of different colors per color multiset" influence the complexity? In the constructed instance this parameter is unbounded.
3. Is ICC fixed-parameter tractable with respect to the parameter "maximum interval length"?
4. Is ICC fixed-parameter tractable with respect to the parameter "maximum overlap between intervals"?

The central point underlying the above derived algorithmic questions is that whenever a quantity (that is, parameter) in an NP-hardness proof is unbounded (non-constant), then this evokes the quest to know what happens if this quantity is constant or considered to be small compared to the overall input size. Clearly, one way to answer is to provide a different proof of NP-hardness where this quantity is bounded. Otherwise, the main tool in answering such questions is parameterized algorithmics. Indeed, the story goes even further by combining different parameterizations. More specifically, it is, for instance, natural to ask whether ICC is fixed-parameter tractable when parameterized by both cutwidth and the number of colors k (the answer is open), or whether it is fixed-parameter tractable when parameterized by both n and k (the answer is "yes") and what the combinatorial explosion $f(n, k)$ then looks like. In this way, one ends up with an extremely diverse and fruitful ground to develop practically relevant combinatorial algorithms.

In the remainder of this paper, besides the already defined parameters n (range), m (number of intervals), and k (number of colors), we will consider the following parameters and combinations thereof:

– maximum interval length l;
– cutwidth $c := \max_{1 \leq i \leq n} |\{F \in \mathcal{F} : i \in F\}|$;
– maximum pairwise overlap between intervals $o := \max_{1 \leq i < j \leq m} |F_i \cap F_j|$;
– maximum number of different colors Δ in the color multisets.

Note that one of the integer linear programs devised by Althaus et al. [2] has $O(m \cdot k)$ variables. Using Lenstra's famous result [7] on the running time of integer linear programs with a fixed number of variables then implies that ICC is fixed-parameter tractable with respect to the (combined) parameter (m, k). Due to the huge combinatorial explosion in Lenstra's theorem, however, this result is of purely theoretical interest and more efficient combinatorial algorithms are of big interest.

In the next two sections, we present several fixed-parameter tractability results with respect to the above parameters (Section 3) and combinations of each time two of them (Section 4).

Let us spot some challenges for future research concerning the multivariate complexity analysis of ICC. The complexity behavior with respect to the combined parameter (c, k) is widely open. A breakthrough would be to show the tractability with respect to k—note that we have intractability with respect to c. Basically along the same lines as k, also Δ gives an interesting parameterization with almost no results so far. The parameter m also seems of particular interest. The fixed-parameter tractability with respect to m is completely open and with respect to the combined parameter (m, k) the running time needs improvement.

We close this section with some simple observations about a helpful "normal form" that one may assume without loss of generality for all ICC input instances. More precisely, based on simple and efficient preprocessing rules, one can perform a data reduction that yields this normal form.

Table 1. Complexity of ICC for 1-dimensional parameterizations. Herein, "P" means that the problem is polynomial-time solvable, "NPc" means that the problem is NP-complete, and "?" means that the complexity is unknown. For fixed-parameter algorithms, we only give the function of the exponential term, omitting polynomial factors. The results for $k = 2$ and $c = 3$ are due to Althaus et al. [1, 2], the rest is new.

Parameter	k	Δ	l	c	m	n	o
Complexity	$k = 2$: P $k \geq 3$: ?	$\Delta \geq 2$: ?	$l!$	$c = 2$: P $c = 3$: NPc	?	$n!$	$o = 1$: P $o \geq 2$: ?

Proposition 1 (Normal form for ICC)
In $O(lmn)$ time, one can transform every ICC *instance into an equivalent one such that*

1. *at every position $i \in [n]$, there is at most one interval starting at i and at most one interval ending at i, and*
2. *if the maximum interval length is l, then every position $i \in [n]$ is contained in at most $2l$ intervals.*

3 Single Parameters

In Section 2, we identified various parameters as meaningful "combinatorial handles" to better assess the computational complexity of ICC. Concerning cutwidth c, whereas $c = 3$ is known to be NP-complete [1], here we show that $c = 2$ is polynomial-time solvable. Obviously, $l \leq n$, so the fixed-parameter tractability with respect to l (as we will prove subsequently) implies the fixed-parameter tractability with respect to n. Table 1 surveys known and new results with respect to single parameters.

Theorem 1. ICC *can be solved in $O(l! \cdot lmn)$ time.*

Proof. We present a dynamic programming algorithm. We use the following notation. First, let A denote the set of intervals contained in some other intervals, that is, $A := \{F \in \mathcal{F} \mid \exists_{F' \in \mathcal{F}} : F \subsetneq F'\}$, and $B := \mathcal{F} \setminus A$. Let $\mathcal{K} = \{1, \ldots, k\}$ denote the set of all colors. We say that a coloring c' *satisfies* an input interval $F_i \in \mathcal{F}$ if $c'(F_i) = C_i$. For an interval $[s, t]$, a coloring is represented by a vector in $\mathcal{K}^{t-s+1}$. For an input interval $F_i \in \mathcal{F}$, the set K_i of all satisfying colorings is given by $K_i := \{c' \in \mathcal{K}^{|F_i|} \mid c' \text{ satisfies } C_i\}$. In the worst case that every color occurs at most once in the multiset C_i, there are $|C_i|!$ satisfying colorings of an input interval F_i. In the following, we assume that the intervals in B are ordered in increasing order of their start points (and, hence, also in increasing order of their endpoints). Let $B = \{B_1, \ldots, B_{m'}\}$ and $B_j = [s_j, t_j]$ for all $1 \leq j \leq m'$. The intervals in B cover $[n]$, that is, $\bigcup_{j=1}^{m'} B_j = [n]$. For every B_j, the algorithm maintains a table T_j with an entry for every satisfying coloring of B_j. More specifically, for every coloring $c' = (c'_1, \ldots, c'_{|B_j|}) \in K_j$

we set $T_j(c') = \text{true}$ iff there exists a coloring $c'' = (c''_1, \ldots, c''_{t_j}) \in \mathcal{K}^{t_j}$ of the interval $[t_j]$ with $(c''_{s_j}, \ldots, c''_{t_j}) = c'$ such that c'' satisfies each interval $F \in \mathcal{F}$ with $F \subseteq [t_j]$.

For $j = 1$ and for every $c' \in K_1$, this is achieved by setting $T_1(c') := \text{true}$ iff c' satisfies every interval $[s, t] \in \mathcal{F}$ with $[s, t] \subseteq [s_1, t_1]$.

We say that a coloring $c' = (c'_1, \ldots, c'_{|B_j|}) \in K_j$ for B_j is consistent with a coloring $c'' = (c''_1, \ldots, c''_{|B_{j-1}|}) \in K_{j-1}$ for B_{j-1} if c' and c'' agree in $B_{j-1} \cap B_j$, that is, $(c''_{s_j - s_{j-1}+1}, \ldots, c''_{|B_{j-1}|}) = (c'_1, \ldots, c_{t_{j-1}-s_j+1})$. We write $c'|c''$ to denote that c' is consistent with c''.

For $j = 2, \ldots, n$ and for every $c' = (c'_1, \ldots, c'_{|B_j|}) \in K_j$, we set

$$\begin{aligned} T_j(c') = \text{true} \iff\, & c' \text{ satisfies all } F \in \mathcal{F} \text{ with } F \subseteq B_j \text{ and} \\ & \exists_{c'' \in K_{j-1},\, c'|c''} : T_{j-1}(c'') = \text{true}. \end{aligned}$$

The correctness can be seen as follows. The "$\Rightarrow$"-direction follows directly by definition. For the "$\Leftarrow$"-direction, observe the following. A coloring of $[t_j]$, composed of a coloring of $[t_{j-1}]$ satisfying all $F \in \mathcal{F}$ with $F \subseteq [t_{j-1}]$ and a coloring c' of $[s_j, t_j]$ satisfying all $F \in \mathcal{F}$ with $F \subseteq [s_j, t_j]$, satisfies all $F \in \mathcal{F}$ with $F \subseteq [t_j]$; clearly, all $F \in \mathcal{F}$ with $F \subseteq [t_{j-1}]$ are satisfied. Moreover, all other $F \in \mathcal{F}$ with $F \subseteq [t_j]$ are satisfied since for every fragment $[s, t] \in \mathcal{F}$ with $t_{j-1} < t \leq t_j$ it holds that $[s, t] \subseteq [s_j, t_j]$.

As to the running time, there are at most $|B_j|!$ satisfying colorings of B_j; at most one for every permutation of the associated color multiset. Hence, one has to consider at most $l!$ colorings for every B_j. For every $j = 1, \ldots, m' - 1$, the algorithm works as follows. When building the table T_j for every $c' = (c'_1, \ldots, c'_{|B_j|}) \in K_j$, the algorithm computes an auxiliary table Q_j with an entry $Q_j(c'_r, c'_{r+1}, \ldots, c'_{|B_j|})$, where $r := s_{j+1} - s_j + 1$, indicating whether $T_j(c') = \text{true}$. Herein, in order to ensure that the size of Q_j does not exceed $l!$ and to allow fast access to its elements, table Q_j can for example be realized as an array of size $|B_j|!$ where the entry for $c_s = (c'_r, c'_{r+1}, \ldots, c'_{|B_j|})$ is stored at the position corresponding to the number of the lexicographically smallest permutation of C_j with "prefix" c_s. Then, to check whether $\exists_{c'' \in K_{j-1},\, c'|c''} : T_{j-1}(c'') = \text{true}$ for a $c' = (c'_1, \ldots, c'_{|B_j|}) \in K_j$, the algorithm can check whether $Q_{j-1}(c'_1, \ldots, c'_{|B_j|}) = \text{true}$ in $O(l)$ time. Hence, for every position $1 \leq j \leq m'$, it needs at most $O(l! \cdot (l + lm))$ time, where the factor lm is due to checking whether c' satisfies all $F \in \mathcal{F}$ with $F \subseteq [s_j, t_j]$. In summary, the total running time is $O(l! \cdot lmn)$ since $m' \leq n$. $\square$

Next, we show that ICC is solvable in $O(n^2)$ time in case the cutwidth $c = 2$. This contrasts the case $c = 3$ shown to be NP-complete [1]. Our algorithm is based on four data reduction rules that are executable in polynomial time. The application of these rules either leads to a much simplified instance that can be colored without violating any interval constraints or shows that the instance is a no-instance.

Reduction Rule 1. *For any two intervals F_i and F_j,*

- *if $|F_i \cap F_j| = |C_i \cap C_j|$, then set $c(F_i \cap F_j) = C_i \cap C_j$;*
- *if $|F_i \cap F_j| > |C_i \cap C_j|$, then return "No".*

Rule 1 is obviously correct: if two intervals "share" more positions than colors, then there is no coloring that satisfies both intervals, and if the number of shared positions is equal to the number of shared colors, then we have to color the overlapping intervals exactly with these colors.

Note that when we set $c(i) = c_x$ for some position i, we can simplify the instance as follows. For all $F_j = [s, t]$ with $s \leq i \leq t$, we set $C_j := C_j \setminus \{c_x\}$ and $t := t - 1$. For all $F_j = [s, t]$ with $i < s$, we set $s := s - 1$ and $t := t - 1$. "Empty" intervals F_j with $C_j = \emptyset$ are removed from the input. After Rule 1 and this subsequent reduction of the instance, we can assume that no interval is completely contained in any other interval.

In the following, assume that the intervals are ordered with respect to their startpoints, that is, for $F_i = [s_i, t_i]$ and $F_j = [s_j, t_j]$ with $i < j$ we have $s_i < s_j$. Let t be the endpoint of the first interval $F_j \neq F_1$ that overlaps only with one other interval. Clearly, we can color $[t]$ independently from $[t + 1, n]$. Together with Rule 1, and the fact that $c = 2$, we can thus assume that all intervals except for F_1 and F_m overlap with exactly two other intervals. Hence, we can partition each interval F_j, $1 < j < m$, into at most three subintervals: the first subinterval overlaps with F_{j-1}, the second (possibly empty) subinterval does not overlap with any other interval, and the third subinterval overlaps with F_{j+1}. The following notation describes this structural property. For an interval F_j, $1 < j < m$, define $F_j^1 := F_j \cap F_{j-1}$, $F_j^3 := F_{j+1}^1$, and let $F_j^2 := F_j \setminus (F_j^1 \cup F_j^3)$. For a coloring c' of all intervals, let $C_j^1 := c'(F_j^1)$. Define C_j^2 and C_j^3 accordingly. For F_1, define $F_1^3 := F_2^1$, and $F_1^2 := F_1 \setminus F_1^3$; for F_m define $F_m^1 := F_m \cap F_{m-1}$ and $F_m^2 := F_m \setminus F_m^1$; C_1^3, C_1^2, C_m^1, and C_m^2 are defined analogously. Whether a coloring violates an interval F_j only depends on the sets C_j^1,C_j^2, and C_j^3. Hence, when we know that a color c_x must belong to some C_j^l, $1 \leq l \leq 3$, then we can color an arbitrary $i \in F_j^l$ with c_x. Finally, let $\mathrm{occ}(x, C)$ denote the multiplicity of an element x in a multiset C.

The next rule reduces intervals F_j that have no "private" middle interval F_j^2 but more elements of a color c_x than the previous interval.

Reduction Rule 2. *For any interval F_j, if $F_j^2 = \emptyset$ and there is a color c_x such that $\mathrm{occ}(c_x, C_{j-1}) < \mathrm{occ}(c_x, C_j)$, then set $c(i) = c_x$ for some arbitrary $i \in F_j^3$.*

The rule is obviously correct. After its application, for every interval F_j with $F_j^2 = \emptyset$, we have $|F_{j-1}| > |F_j|$. Next, we reduce triples of intervals F_{j-1}, F_j, F_{j+1} that have identical color multisets in case $F_j^2 = \emptyset$.

Reduction Rule 3. *For intervals F_{j-1}, F_j, and F_{j+1} such that $C_{j-1} = C_j = C_{j+1}$ and $F_j^2 = \emptyset$, remove F_{j-1} and F_j from the input and for all intervals $F_{j+l} = [s, t]$ with $l \geq 1$ set $F_{j+l} = [s', t']$, where $s' := s - |F_j|$ and $t' := t - |F_j|$.*

The correctness proof for Rule 3 is omitted.

The following is our final data reduction rule.

Reduction Rule 4. *Let I be an instance that is reduced with respect to Rules 1, 2, and 3, and let F_j be the first interval of I such that there is a color c_x with* $\mathrm{occ}(c_x, C_j) > \mathrm{occ}(c_x, C_{j+1})$. *Do the following:*

- *if $j = 1$, then set $c(i) = c_x$ for some $i \in F_1^2$;*
- *if $j > 1$ and $c_x \notin C_{j-1}$, then set $c(i) = c_x$ for some $k \in F_j^2$ in case $F_j^2 \neq \emptyset$ and otherwise return "No";*
- *if $j > 1$ and $c_x \in C_{j-1}$, then set $c(i) = c_x$ for some $i \in F_j^1$.*

Lemma 1. *Rule 4 is correct.*

Proof. Let I be an instance, reduced with respect to Rules 1, 2, and 3, to which Rule 4 is applied, and let I' be the resulting instance. We only show that if I is a yes-instance, then I' is a yes-instance, since the reverse direction trivially holds.

If $j = 1$, this is easy to see: since c_x occurs more often in F_1 than in F_2 one of the positions in $F_1 \setminus F_2$ must be colored with c_x.

If $j > 1$ and $c_x \notin C_{j-1}$, then it is clear that one of the positions in F_j^2 must be colored with c_x. We either perform this forced coloring or return "No" if this is not possible.

Finally, if $j > 1$ and $c_x \in C_{j-1}$, the situation is more complicated. Let c' be a coloring that fulfills the interval constraints of I, we call such a coloring *proper*. If there is a position $i \in F_j^1$ such that $c'(i) = c_x$, then the claim obviously holds. Otherwise, we show that we can transform c' into an alternative coloring c'' that is proper and there is an $i \in F_j^1$ such that $c''(i) = c_x$. Whether coloring c'' is proper depends only on the multisets C_l^1, C_l^2, and C_l^3, $1 \leq l \leq m$, that are defined by the coloring function c'. Hence, we describe the transformation applied to c' with respect to these multisets. Note that we do not modify the sets C_l^y for any $l > j$.

We face the following situation: $c_x \notin C_j^1$, but since c' is a coloring that does not violate any interval constraints and by the precondition of Rule 2, $c_x \in C_j^2$. By the precondition of Rule 4, we have $C_1 \subseteq C_2 \subseteq \ldots \subseteq C_j$. We show that we can always find a series of exchange operations such that the resulting coloring is proper and $c_x \in C_j^1$. We perform a case distinction. Due to the lack of space, we show only some cases, the other cases are similar, albeit more complicated.

Case 1: $F_{j-1}^2 \neq \emptyset$. There are three subcases of this case.
Case 1.1: $c_x \in C_{j-1}^2$. In this case, we exchange $c_x \in C_{j-1}^2$ and some arbitrary $c_l \in C_j^1$. Furthermore, we remove c_x from C_j^2 and add c_l to C_j^2. The exchange is shown in Fig. 1a; the resulting coloring is clearly proper and $c_x \in C_j^1$.
Case 1.2: $c_x \in C_{j-1}^1$ and $F_{j-2}^2 \neq \emptyset$. Clearly, C_{j-2} must be involved in the exchange. We choose an arbitrary element $c_l \in C_{j-2}^2$. Since $C_{j-2} \subseteq C_{j-1}$, we also have $c_l \in C_{j-1} \setminus C_{j-2}$. We distinguish two subcases.
Case 1.2.1: $c_l \in C_{j-1}^3$. We perform a *direct* exchange of c_l and c_x between C_{j-2}^2 and C_{j-1}^1 and also between C_j^1 and C_j^3. The exchange is shown in Fig. 1b; the resulting coloring is clearly proper and $c_x \in C_j^1$.

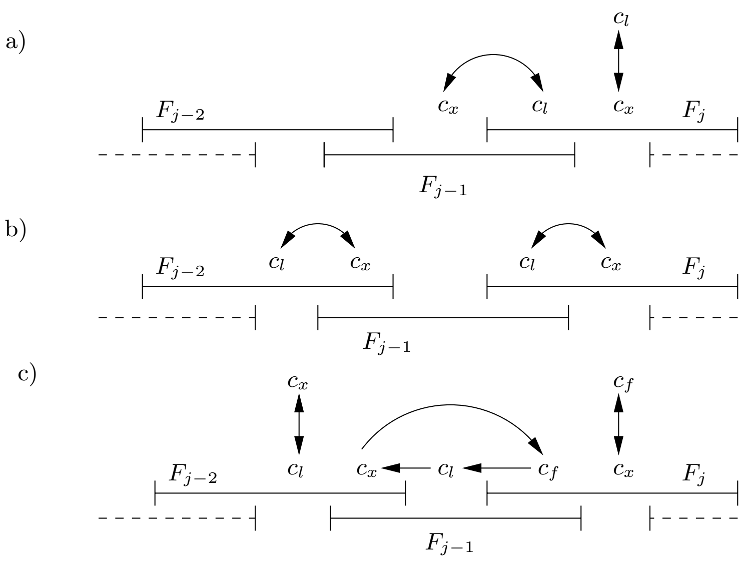


Fig. 1. Exchange operations used in the proof of Lemma 1

Case 1.2.2: $c_l \in C^2_{j-1}$**.** We remove c_l from C^2_{j-2} and add c_x to C^2_{j-2}. Furthermore, we perform a *circular* exchange between C^1_{j-1}, C^2_{j-1}, and C^3_{j-1}: move c_x from C^1_{j-1} to C^3_{j-1}, move an arbitrary element c_f from C^3_{j-1} to C^2_{j-1}, and move c_l from C^2_{j-1} to C^1_{j-1}. Finally, we remove c_x from C^2_j and add c_f to C^2_j. The exchange is shown in Fig. 1c; the resulting coloring is clearly proper and $c_x \in C^1_j$.

The correctness of the final two cases is deferred to a long version of this paper.

Case 1.3: $c_x \in C^1_{j-1}$ **and** $F^2_{j-2} = \emptyset$**.**

Case 2: $F^2_{j-1} = \emptyset$**.**

In all cases, we can construct an alternative coloring that is proper and $c_x \in C^1_j$, which means that we can assume that if I is yes-instance, then there is some $i \in C^1_j$ such that $c'(i) = c_x$. In summary, this shows that I is a yes-instance iff I' is a yes-instance. □

With these four reduction rules at hand, we can describe a simple quadratic-time algorithm for ICC with cutwidth two.

Theorem 2. ICC *can be solved in* $O(n^2)$ *time when the input has cutwidth two.*

Proof. The algorithm starts with exhaustively applying Rules 1 to 4. Note that the rules have to be applied in the correct order, that is, after each reduction step, we always check first whether Rule 1 can be applied, then whether Rule 2 can be applied, and so on. The rules either return "No" or we obtain an instance that is reduced with respect to all data reduction rules. In such an instance we have $C_1 \subseteq C_2 \subseteq \ldots \subseteq C_m$. Otherwise, Rule 4 would apply, because there would be some $F_i \in \mathcal{F}$ and a color c_x such that $\text{occ}(c_x, C_i) > \text{occ}(c_x, C_i + 1)$. This instance can be easily colored as follows. For the first interval F_1, we choose an arbitrary coloring that does not violate C_1. Clearly, this also does not violate C_2 since $C_1 \subseteq C_2$. Then we remove the colored parts from the input, adjust the color multisets accordingly, and choose an arbitrary coloring that does not violate C_2. Clearly, this does not violate C_3, since $C_2 \subseteq C_3$. After this, we again reduce the colored parts and continue with coloring F_3. This is repeated until all positions are colored and clearly produces a coloring that does not violate any interval constraints. This proves the correctness of the algorithm.

For the running time of the algorithm consider the following. First, since the input has cutwidth two, we have $m = O(n)$. For each data reduction rule, checking whether it can be applied and the application itself can be performed in $O(n)$ time. Furthermore, the application of any of the reduction rules removes at least one position from the interval $[n]$. Hence, each rule can be applied at most n times. Together with the $O(n)$ time that is clearly sufficient for coloring any instance reduced with respect to the reduction rules, this leads to a total running time of $O(n^2)$. $\square$

Using the previous algorithm, we also obtain polynomial-time solvability in case the maximum overlap o between fragments is at most one. This follows from the observation that after achieving the normal form of the instance, each instance with overlap at most one also has cutwidth at most two, which can be seen as follows. Suppose an instance that has the normal form has overlap one and cutwidth at least three. Then there must be an interval F_i that overlaps with two other intervals F_j, F_l at some position x. Since at each position at most one interval starts, at most one of F_j and F_l, say F_j, starts at position x. This, however, means that F_l starts at some position $u < x$ and hence it has overlap at least two with F_i, leading to a contradiction.

Corollary 1. ICC *can be solved in* $O(n^2)$ *time when the input has overlap one.*

4 Combined Parameters

In the following, as already indicated in Section 2, we turn to the study of some relevant pairs of single parameters which form a "combined parameter". Table 2 summarizes our current knowledge about combined parameterizations of ICC—there are many questions left open. Here, we study the three different combined parameters (l, k), (l, c), and (n, k) that seem to be of immediate practical interest and all allow for fixed-parameter tractability results. The proof of Theorem 3 is somewhat similar to the proof of Theorem 1 and therefore omitted.

Theorem 3. ICC *can be solved in* $O(k^l \cdot (k + lm)n)$ *time.*

Theorem 4. ICC *can be solved in* $(c + 1)^l \cdot \text{poly}(n, m)$ *time.*

Proof. We present a dynamic programming algorithm. We use the following notation. For every position i, $1 \leq i \leq n$, let $\mathcal{F}_i = \{F_{i_1}, \ldots, F_{i_{n_i}}\}$ denote the input intervals containing i. Further, let $\mathcal{C}_i = \{C_{i_1}, \ldots, C_{i_{n_i}}\}$ denote the color

Table 2. Complexity of ICC for combined parameters. We only give the function of the exponential term, omitting polynomial factors. Herein, $(k,*)$ and $(k,*,*)$ refer to combined parameters that feature k and one or two additional parameters, $(l,*)$ refers to combined parameters that feature l and one additional parameter. Note that for $k=3$, we achieve an improvement from 2^n to 1.89^n. The result for parameter (k,m) is due to Althaus et al. [2], the rest is new.

Parameter	Running times
$(k,*)$	k^l, $(k-1)^n$, $f(k,m)$ (ILP)
$(k,*,*)$	$l^{c\cdot(k-1)}$, $n^{c\cdot(k-1)}$
$(l,*)$	Δ^l, $(c+1)^l$

multisets associated with the intervals in $\mathcal{F}_i$, where C_{i_j} is the color multiset associated with F_{i_j}, $1 \leq j \leq n_i$. Note that $n_i \leq c$. Further, let $F_{i_j} = [s_{i_j}, t_{i_j}]$ for all $1 \leq j \leq n_i$. By $\mathcal{K} = \{1,\ldots,k\}$ we refer to the set of all colors. Further, a tuple $(M_1,\ldots,M_q)$ of (multi)sets is called a *chain* if there exists a permutation π of $\{1,\ldots,q\}$ such that $M_{\pi(1)} \subseteq M_{\pi(2)} \subseteq \ldots \subseteq M_{\pi(q)}$.

For every position i, the algorithm maintains a table T_i with a Boolean entry for every possible tuple of color multisets $(A_1,\ldots,A_{n_i})$ with $A_j \subseteq C_{i_j}$ and $|A_j| = i - s_{i_j} + 1$ for all $1 \leq j \leq n_i$. More specifically, $T_i(A_1,\ldots,A_{n_i})$ is true iff there exists a coloring $c' : [i] \to \mathcal{K}$ such that $c'([s_{i_j}, i]) = A_j$ for all $1 \leq j \leq n_i$ and for every $F_l \in \mathcal{F}$ with $F_l \subseteq [i]$ it holds that $c'(F_l) = C_l$. Note that an instance is a yes-instance iff there is a true entry in T_n. The goal of the dynamic programming is to compute the tables T_i to fulfill this definition.

According to Proposition 1, at each position in $[n]$ there starts at most one input interval and ends at most one input interval. Hence, there is exactly one interval in $\mathcal{F}_1$. Let $\mathcal{F}_1 = \{F\}$ and let C be the color multiset associated with F. We set $T_1(\{c'\}) = \text{true}$ for every $c' \in C$.

For every position i, $2 \leq i \leq n$, there is at most one interval in $\mathcal{F}_{i-1} \setminus \mathcal{F}_i$ and at most one in $\mathcal{F}_i \setminus \mathcal{F}_{i-1}$. Thus, assume that $\mathcal{F}_{i-1} = \{F', F_1, \ldots, F_q\}$ and $\mathcal{F}_i = \{F_1,\ldots,F_q,F''\}$, that is, $\mathcal{F}_{i-1} \cap \mathcal{F}_i = \{F_1,\ldots,F_q\}$ (if $\mathcal{F}_{i-1} \setminus \mathcal{F}_i = \emptyset$ or $\mathcal{F}_i \setminus \mathcal{F}_{i-1} = \emptyset$, then skip F' or F'' in the following formulas). Let $F_j = [s_j, t_j]$ for all $1 \leq j \leq q$.

For every tuple $(A_1,\ldots,A_q,A'')$ that forms a chain and fulfills $A_j \subseteq F_j$, $|A_j| = i - s_j + 1$ for $1 \leq j \leq q$ and $A'' \subseteq F''$, $|A''| = 1$, set

$$T_i(A_1,\ldots,A_q,A'') = \text{true} \iff \exists_{x \in (\bigcap_{j=1}^{q} A_j) \cap A''} : T_{i-1}(F', A_1 \setminus \{x\}, \ldots, A_q \setminus \{x\}). \quad (I)$$

The correctness of this recursion can be seen as follows. On the one hand, if $T_i(A_1, A_2, \ldots, A_q, A'') = \text{true}$, that is, if there exists a coloring $c' : [i] \to \mathcal{K}$ fulfilling the above conditions, then clearly c' restricted to $[i-1]$ fulfills the above properties for $i-1$ and $F', A_1 \setminus \{c'(i)\}, \ldots, A_q \setminus \{c'(i)\}$.

On the other hand, if $\exists_{x \in (\bigcap_{j=1}^{q} A_j) \cap A''} : T_{i-1}(A', A_1 \backslash \{x\}, \ldots, A_q \backslash \{x\}) = \text{true}$, that is, if there exists a coloring $c'' : [i-1] \to \mathcal{K}$ fulfilling the above properties for $i-1$ and $A', A_1 \backslash \{x\}, \ldots, A_q \backslash \{x\}$, then the extension c' of c'' with $c'(j) = c''(j)$ for $1 \leq j < i$ and $c'(i) = x$ fulfills the above properties for i and $A_1, \ldots, A_q, A''$.

Finally, note that we only consider tuples $(A_1, \ldots, A_q, A'')$ that form chains. This is correct, since for a coloring $c' : [i] \to \mathcal{K}$ it clearly holds that the tuple $(c'([s_1, i]), \ldots, c'([s_j, i]), \{c(i)\})$ forms a chain.

In accordance with the equivalence (I), the algorithm computes the tables T_i for increasing values of i (starting with $i = 2$). Finally, it outputs "Yes" if T_n contains a true entry and "No", otherwise.

As to the running time, for every position there are at most $(c+1)^l$ tuples of color multisets $(A_1, \ldots, A_{n_i})$ with $A_j \subseteq C_{i_j}$ and $|A_j| = i - s_{i_j} + 1$, $1 \leq j \leq n_i$, that form a chain. This can be seen as follows. Let F_l denote the interval in $\mathcal{F}_i$ with the smallest starting point. Clearly, a tuple of color multisets $(A_1, \ldots, A_{n_i})$ that forms a chain corresponds to a partition of C_j into $(c+1)$ subsets. Since, for every color in F_j there are $(c+1)$ choices, there are at most $(c+1)^l$ such partitions. □

For a multiset M that contains k different colors and for an integer $q \geq 1$ there are at most q^{k-1} size-q submultisets of M. This is true since if we have chosen the occurrence number of the first $k-1$ colors in a size-q subset (there are at most q^{k-1} choices), then the occurrence number of the kth color is fixed. With this observation, the running time of the algorithm presented in the proof of Theorem 4 can also be bounded by $O^*(l^{c \cdot (k-1)})$.

Corollary 2. ICC *can be solved in* $l^{c \cdot (k-1)} \cdot \text{poly}(n, m)$ *time.*

Trivially, we can solve any instance in $O(k^n)$ time by trying all k colors for all n positions. Now, we use the fact that for two colors the problem is polynomial-time solvable [2]. Hence, we need to "guess" only $k-2$ colors and the positions that have one of the two remaining colors. For these positions, we then use the polynomial-time algorithm for ICC with two colors, giving the following result.

Theorem 5. ICC *can be solved in* $O((k-1)^n \cdot g(n, m))$ *time, where* $g(n, m)$ *is the time needed to solve* ICC *for* $k = 2$.

For the practically relevant case where $k = 3$ [2], we can achieve a speed-up by the following simple observation: At least one of the colors appears at most on $n/3$ positions.

Theorem 6. *For* $k = 3$, ICC *can be solved in* $O(1.89^n \cdot g(n, m))$ *time, where* $g(n, m)$ *is the time needed to solve* ICC *for* $k = 2$.

Beigel and Eppstein [3] gave a thorough study of exact exponential-time algorithms for the NP-complete 3-Coloring problem. It is tempting to investigate whether some of their tricks can be applied to ICC with three colors; in particular, a simple randomized strategy presented by Beigel and Eppstein might be promising.

5 Conclusion

Conceptually, we presented a systematic development of the method of "deconstructing intractability". We exhibited this approach using the NP-complete ICC problem as a particularly fertile application case. Through deconstruction and using methods of parameterized algorithmics, we started a diverse multivariate complexity analysis of ICC. Refer to Tables 1 and 2 in Sections 3 and 4 for an overview and numerous challenges for future research. There remain many challenges for future work: Even combinations of three or more parameters may be relevant. Besides that, already for pairs of two single parameters there are several qualitatively different fixed-parameter tractability results one can strive for and which typically are independent from each other. For instance, for a combined parameter (p_1, p_2) combinatorial explosions such as $p_1^{p_2}$, $p_2^{p_1}$, $2^{p_1 \cdot p_2}$ etc. all can be useful for solving specific real-world instances. Finally, we focussed attention on the decision version, but the investigations should clearly be extended to the optimization variants. Summarizing, the research challenges offered by ICC and, more generally, the deconstructive approach to intractability, seem to be (almost) inexhaustible.

Acknowledgment. We thank Michael R. Fellows for early discussions about the deconstructive approach to NP-hard problems and Nadja Betzler for pointing us to the ICC problem.

References

[1] Althaus, E., Canzar, S., Elbassioni, K., Karrenbauer, A., Mestre, J.: Approximating the interval constrained coloring problem. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 210–221. Springer, Heidelberg (2008)
[2] Althaus, E., Canzar, S., Emmett, M.R., Karrenbauer, A., Marshall, A.G., Meyer-Baese, A., Zhang, H.: Computing H/D-exchange speeds of single residues from data of peptic fragments. In: Proceedings of the 23rd ACM Symposium on Applied Computing (SAC 2008), pp. 1273–1277. ACM Press, New York (2008)
[3] Beigel, R., Eppstein, D.: 3-coloring in time $O(1.3289^n)$. Journal of Algorithms 54(2), 168–204 (2005)
[4] Chang, J., Erlebach, T., Gailis, R., Khuller, S.: Broadcast scheduling: Algorithms and complexity. In: Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA 2008), pp. 473–482. ACM-SIAM (2008)
[5] Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
[6] Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)
[7] Lenstra, H.W.: Integer programming with a fixed number of variables. Mathematics of Operations 8, 538–548 (1983)
[8] Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics and Its Applications, vol. 31. Oxford University Press, Oxford (2006)
[9] Woeginger, G.J.: Exact algorithms for NP-hard problems: A survey. In: Jünger, M., Reinelt, G., Rinaldi, G. (eds.) Combinatorial Optimization - Eureka, You Shrink! LNCS, vol. 2570, pp. 185–208. Springer, Heidelberg (2003)

Maximum Motif Problem in Vertex-Colored Graphs[⋆]

Riccardo Dondi[1], Guillaume Fertin[2], and Stéphane Vialette[3]

[1] Dipartimento di Scienze dei Linguaggi, della Comunicazione e degli Studi Culturali
Università degli Studi di Bergamo, Piazza Vecchia 8, 24129 Bergamo - Italy
riccardo.dondi@unimib.it
[2] Laboratoire d'Informatique de Nantes-Atlantique (LINA), UMR CNRS 6241
Université de Nantes, 2 rue de la Houssinière, 44322 Nantes Cedex 3 - France
guillaume.fertin@univ-nantes.fr
[3] IGM-LabInfo, CNRS UMR 8049, Université Paris-Est,
5 Bd Descartes 77454 Marne-la-Vallée, France
vialette@univ-mlv.fr

Abstract. Searching for motifs in graphs has become a crucial problem in the analysis of biological networks. In this context, different graph motif problems have been considered [13,7,5]. Pursuing a line of research pioneered by Lacroix *et al.* [13], we introduce in this paper a new graph motif problem: given a vertex colored graph G and a motif $\mathcal{M}$, where a motif is a multiset of colors, find a maximum cardinality submotif $\mathcal{M}' \subseteq \mathcal{M}$ that occurs as a connected motif in G. We prove that the problem is **APX**-hard even in the case where the target graph is a tree of maximum degree 3, the motif is actually a set and each color occurs at most twice in the tree. Next, we strengthen this result by proving that the problem is not approximable within factor $2^{\log^\delta n}$, for any constant $\delta < 1$, unless $\mathbf{NP} \subseteq \mathbf{DTIME}(2^{\text{poly}\log n})$. We complement these results by presenting two fixed-parameter algorithms for the problem, where the parameter is the size of the solution. Finally, we give exact fast exponential-time algorithms for the problem.

1 Introduction

Searching for motifs in graphs has become a crucial problem in the analysis of biological networks (*e.g.* protein-protein interaction, regulatory and metabolic networks). Roughly speaking, there exist two different views of graph motifs. Topological motifs (patterns occurring in the network) are the classical view [11,17,18,16,12] and computationally reduce to graph isomorphism, in the broad meaning of that term. These motifs have recently been identified as basic modules of molecular information processing. By way of contrast, functional motifs, introduced recently by Lacroix *et al.* [13], do not rely on the key concept of topology conservation but focus on connectedness of the network vertices sought.

[⋆] Supported by the Italian-French PAI Galileo Project 08484VH.

This latter approach has been considered in subsequent papers [7,5,2,3]. Formally, searching for a functional motif reduces to the following graph problem (referred hereafter as GRAPH MOTIF) [13]: Given a target vertex-colored graph $G = (V, E)$ and a multiset of colors $\mathcal{M}$ of size k, find a subset $V' \subseteq V$, $|V'| = k$ $(= |\mathcal{M}|)$ such that (i) the vertex induced subgraph $G[V']$ is connected and (ii) there exists a color-preserving bijective mapping from $\mathcal{M}$ to V'.

GRAPH MOTIF is **NP**-complete even if G is a tree with maximum degree 3 and $\mathcal{M}$ is actually a set [7]. **NP**-completeness has also been shown in case G is a bipartite graph with maximum degree 4 and $\mathcal{M}$ is built over two colors only [7]. The seemingly intractability of GRAPH MOTIF has naturally led to parameterized complexity considerations [6]. GRAPH MOTIF can be solved in $\mathcal{O}(4.32^k\, k^2\, m)$ randomized time [2], where m is the number of edges in G, and in $\mathcal{O}(n^{2c\omega+2})$ time [7], where ω is the tree-width of G and c is the number of distinct colors in $\mathcal{M}$. When the number of distinct colors in the motif is taken as a parameter, GRAPH MOTIF is, however, $\mathbf{W[1]}$-hard even in case G is a tree.

Aiming at accurate models, several variants of GRAPH MOTIF have been considered. Dondi *et al.* [5] introduced the problem of minimizing the number of connected components in $G[V']$, *i.e.*, finding an occurrence of $\mathcal{M}$ in G that results in as few connected components as possible. This problem was referred as MIN-CC. It turns out that MIN-CC is **APX**-hard even in the extremal case where the motif is a set and the target graph is a path and is not approximable within ratio $c \log n$ for some constant $c > 0$, where n is the order of the target graph. From a parameterized point of view, MIN-CC is fixed-parameter tractable when the parameter is the size of the motif but becomes $\mathbf{W[2]}$-hard when the parameter is the number of connected components in the occurrence of the motif (the problem is, however, only known to be $\mathbf{W[1]}$-hard for paths [2]). Betzler *et al.* [2] replaced connectedness demand by more robust requirements, and proved the problem of finding a biconnected occurrence of $\mathcal{M}$ in G to be $\mathbf{W[1]}$-complete when the parameter is the size of the motif. This result is important as it sheds light on the fact that a seemingly small step towards motif topology results in parameterized intractability. In this paper, we consider the MAXIMUM MOTIF problem which is a natural dual variant of GRAPH MOTIF. This problem is concerned with finding a maximum cardinality submotif $\mathcal{M}' \subseteq \mathcal{M}$ that occurs as a connected motif in G. Notice that the problem is an optimization problem whereas GRAPH MOTIF is a pure decision problem.

This paper is organized as follows. We recall basic definitions in Section 2. In Section 3, we present inapproximability results for MAXIMUM MOTIF. In Section 4, we present two exact exponential algorithms for MAXIMUM MOTIF, when the target graph is a tree. In Section 5, we give two fixed-parameter algorithms, parameterized by the size of the solution, when the target graph is a tree and when it is a general graph. Due to space constraints, some proofs are omitted.

2 Preliminaries

We assume readers have basic knowledge about graph theory [4] and we shall only recall basic notations. Let $G = (V, E)$ be a graph. For any $V' \subseteq V$, we denote by $G[V']$ the *subgraph of G induced by V'*, that is $G[V'] = (V', E')$ and $\{u, v\} \in E'$ iff $u, v \in V'$ and $\{u, v\} \in \mathbf{E}(G)$. Let $v \in V$, we denote by $N(v)$, the set of vertices $u \in V$ such that $\{u, v\} \in E$. Let $V' \subseteq V$; we denote by $N(V')$ the set of vertices $u \in (V \setminus V')$ such that $\{u, v\} \in E$, for some $v \in V'$. A *coloring* of G is a mapping $\lambda : V \to \mathcal{C}$, where $\mathcal{C}$ is a set of colors. For any subset V' of V, we let $\mathcal{C}(V')$ stand for the multiset of colors assigned to the vertices in V'. A motif $\mathcal{M}$ is a multiset of colors built over a set of colors $\mathcal{C}$. In case $\mathcal{M}$ is actually a set, we call it a *colorful motif.* An *occurrence* of $\mathcal{M}$ in G is a subset $V' \subseteq V$ such that (i) $G[V']$ is connected, and (ii) $\mathcal{C}(V') = \mathcal{M}$. A tree where a root has been specified is called a *rooted tree.* In a rooted tree with root r, for every non-root node x, let e_x be the unique edge incident to x that lies on the path from x to r. Then e_x can be thought of as connecting each node x to its *parent.* Two vertices with the same parent are said to be *siblings.* Rooted trees can also be considered as directed in the sense that all edges connect parents to their *children.* Given this parent-child relationship, a *descendant* of a node in a directed tree is defined as any other node reachable from that node.

We can now define the Maximum Motif problem we are interested in. Maximum Motif asks for a connected component $G' = (V', E')$ of maximum cardinality in G such that $\mathcal{C}(V') \subseteq \mathcal{M}$ (taking the number of occurrences of each color into account).

Maximum Motif

- **Input** : A target vertex colored graph G and a colored motif $\mathcal{M}$.
- **Output** : A maximum cardinality connected component $G' = (V', E')$ of G such that $\mathcal{C}(V') \subseteq \mathcal{M}$.

Intuitively, Maximum Motif thus asks for the largest submotif $\mathcal{M}' \subseteq \mathcal{M}$ that occurs in G (as a connected component). Being a mere restriction of Graph Motif, Maximum Motif is **NP**-complete as well [13].

3 Hardness of Approximation

We prove **APX**-hardness of Maximum Motif. Recall that, given a graph $G = (V, E)$, the maximum independent set problem (Independent Set) seeks for a maximum cardinality subset $V' \subseteq V$ such that no two vertices in V' are joined by an edge. Independent Set is known to be **APX**-hard even when restricted to cubic graphs [15].

Proposition 1. Maximum Motif *is* **APX**-*hard even if the motif is colorful and the target graph is a tree with maximum degree* 3*.*

Proof. The proof is by reduction from Independent Set for cubic graphs. Let $G = (V, E)$ be an instance of Independent Set for cubic graphs. Write

$V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$. For each $v_i \in V$, let us denote by $E(v_i)$ the three edges of E that are incident to v_i. Furthermore, denote by $e(v_i, j)$ the j-th edge of $E(v_i)$, $1 \leq j \leq 3$, where the order is arbitrary. We show how to construct the corresponding instance of Maximum Motif. This instance consists in a vertex-colored tree $T = (V_T, E_T)$ of maximum degree 3 and a colorful motif $\mathcal{M}$. The tree T is defined as follows: $V_T = \{a_i, b_i, x_{i,I}, x_{i,C}, l_i : 1 \leq i \leq n\} \cup \{d_{i,j}, f_{i,j}, e_{i,j} : 1 \leq i \leq n \ \wedge \ 1 \leq j \leq 3\}$ and $E_T = \{\{a_i, b_i\}, \{b_i, x_{i,I}\}, \{b_i, x_{i,C}\}, \{x_{i,C}, d_{i,1}\}, \{x_{i,I}, f_{i,1}\} : 1 \leq i \leq n\} \cup \{\{a_i, a_{i+1}\} : 1 \leq i < n\} \cup \{\{d_{i,j}, d_{i,j+1}\}, \{f_{i,j}, f_{i,j+1}\} : 1 \leq i \leq n \ \wedge \ 1 \leq j < 3\} \cup \{\{d_{i,j}, e_{i,j}\} : 1 \leq i \leq n \ \wedge \ 1 \leq j \leq 3\}\{\{f_{i,3}, l_i\} : 1 \leq i \leq n\}$. Refer to Figure 1 for a schematic representation of the tree T. Vertex a_i, $1 \leq i \leq n$, is colored $c(a_i)$, vertex b_i, $1 \leq i \leq n$, is colored $c(b_i)$, the two vertices $x_{i,C}$ and $x_{i,I}$, $1 \leq i \leq n$, are colored $c(x_i)$, vertex l_i, $1 \leq i \leq n$, is colored $c(l_i)$, the two vertices $d_{i,j}$ and $f_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq 3$, are colored $c(i, j)$, and vertex $e_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq 3$, is colored $c(e_k)$, where $e_k = e(v_i, j)$. Write $\mathcal{C}$ for the set of all colors that occur in T (notice that each color in $\mathcal{C}$ occurs at most twice in T). The motif $\mathcal{M}$ is defined by $\mathcal{M} = \mathcal{C}$, and is hence colorful.

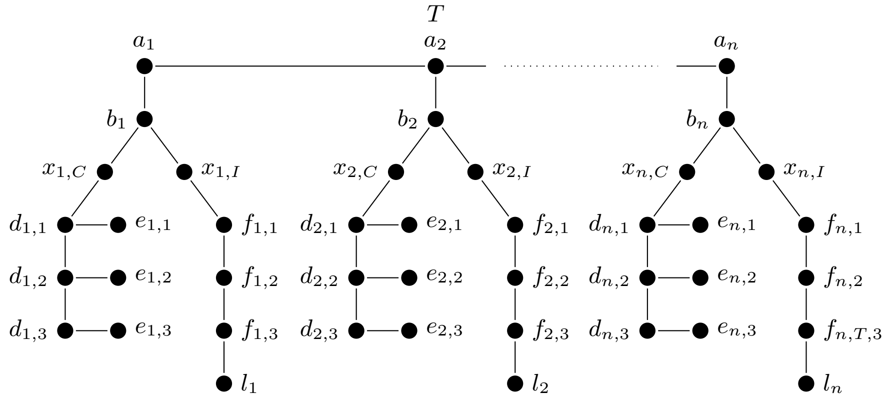


Fig. 1. Schematic representation of the tree T described in the proof of Proposition 1

Suppose there exists an independent set V' of size k in G. For each $e = \{v_i, v_j\} \in E$, define $\min(e)$ to be

$$\min(e) = \begin{cases} v_i & \text{if } (v_j \in V') \ \vee \ (v_i \notin V' \ \wedge \ v_j \notin V' \ \wedge \ i < j), \\ v_j & \text{otherwise.} \end{cases}$$

Consider the subset $V'_T \subseteq V_T$ defined by $V'_T = \{a_i, b_i : 1 \leq i \leq n\} \cup \{x_{i,I}, f_{i,1}, f_{i,2}, f_{i,3}, l_i : v_i \in V'\} \cup \{x_{i,C}, d_{i,1}, d_{i,2}, d_{i,3} : v_i \notin V'\} \cup \{e_{i,j} : e \in E \ \wedge \ \min(e) = e(v_i, j)\}$. Observe that V'_T induces a connected component in T. Furthermore, $\mathcal{C}(V'_T) = \mathcal{M}' \subseteq \mathcal{M}$, contains all colors from $\mathcal{M}$ except those $c(l_i)$ with $v_i \notin V'$.

Conversely, suppose that there exists a motif $\mathcal{M}' \subset \mathcal{M}$, $|\mathcal{M}'| \geq 7$, that occurs in T. Fix one occurrence of $\mathcal{M}'$ in T and write $V'_T \subseteq V_T$ for the vertices of T involved in this occurrence. Without loss of generality, suppose that T' is maximal for inclusion (adding any adjacent vertex to T' results in a subtree that is not an occurrence of a submotif of $\mathcal{M}$). Observe first that $a_i, b_i \in V'_T$, $1 \leq i \leq n$, since adding any of these missing vertices would result in a larger connected component T'' of T, such that $\mathcal{C}(T'') \subseteq \mathcal{M}$, thereby contradicting the maximality of T'. Then it follows that $c(a_i)$, $c(b_i) \in \mathcal{M}'$, $1 \leq i \leq n$. Moreover, since $\mathcal{M}$ is colorful, V'_T contains at most one of $x_{i,C}$ and $x_{i,I}$, $1 \leq i \leq n$; they indeed both have the same color. Therefore, by maximality of T', V'_T contains exactly one of $x_{i,C}$ and $x_{i,I}$, $1 \leq i \leq n$, and hence $\mathcal{M}'$ contains color $c(x_i)$, $1 \leq i \leq n$. Pursuing our maximality argument, if $x_{i,C} \in V'_T$ then V'_T also contains the three vertices $d_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq 3$, and if $x_{i,I} \in V'_T$ then V'_T also contains the three vertices $f_{i,j}$, $1 \leq i \leq n$ and $1 \leq j \leq 3$. Therefore, $\mathcal{M}'$ contains colors $c(i,j)$, $1 \leq i \leq n$ and $1 \leq j \leq 3$. In case $x_{i,I}, f_{i,1}, f_{i,2}, f_{i,3} \in V'_T$, $1 \leq i \leq n$, $l_i \in V'_T$, and hence $\mathcal{M}'$ contains in addition color $c(l_i)$, $1 \leq i \leq n$. We now claim that we may assume that $c(e) \in \mathcal{M}'$ for all $e \in E$, *i.e.*, submotif $\mathcal{M}'$ contains the color associated with each edge of G. Indeed, suppose that for some color $c(e) \in \mathcal{M}$, say $e = \{v_i, v_j\}$, T' has no vertex colored $c(e)$, *i.e.*, $c(e) \notin \mathcal{M}'$. Then, by maximality of T' (and $\mathcal{M}'$), it follows that $\{x_{i,I}, f_{i,1}, f_{i,2}, f_{i,3}, l_i\} \subseteq V'_T$ and $\{x_{j,I}, f_{j,1}, f_{j,2}, f_{j,3}, l_j\} \subseteq V'_T$, and hence that $\{x_{i,C}, d_{i,1}, d_{i,2}, d_{i,3}\} \cap V'_T = \emptyset$ and $\{x_{j,C}, d_{j,1}, d_{j,2}, d_{j,3}\} \cap V'_T = \emptyset$. Therefore, $V''_T = (V'_T - \{x_{i,I}, f_{i,1}, f_{i,2}, f_{i,3}, l_i\}) \cup \{x_{i,C}, d_{i,1}, d_{i,2}, d_{i,3}\} \cup e_{i,p}$, with $c(e_{i,p}) = c(e)$, induces a subtree in T, and this subtree is an occurrence of $\mathcal{M}'' = (\mathcal{M}' - \{c(l_i)\}) \cup \{c(e)\}$. Applying the above procedure will eventually result in a submotif that contains the color associated with each edge of G. Then it follows that $\{v_i : x_{i,C} \in V'_T\}$ is a vertex cover of G, and hence $\{v_i : x_{i,I} \in V'_T\}$ is an independent set in G.

We have thus shown that there is an independent set of size k in G if and only if there exists a submotif of size $6n + m + k$ that occurs in T. But G is a cubic graph, and hence $k \geq \frac{n}{4}$ and $m = \frac{3}{2}n$. Then it follows that the described reduction is indeed an L-reduction [15] from Independent Set for cubic graphs to Maximum Motif for trees, which proves the proposition. □

We now strengthen the inapproximability of Maximum Motif for trees and colorful motifs. More precisely, we show that, for any constant $\delta < 1$, Maximum Motif cannot be approximated within factor $2^{\log^{\delta} n}$ in polynomial-time unless $\mathbf{NP} \subseteq \mathbf{DTIME}[2^{\text{poly} \log n}]$. The proof is by the *self-improvement* technique (see for example [8,9,10]). For the sake of clarity, let us introduce Maximum Level Motif which is the restriction of Maximum Motif to colorful motifs and rooted trees in which two vertices can have the same color only if they are at the same level (i.e., at the same distance to the root) in the target tree. It is easily seen that Proposition 1 can be modified to prove the following result.

Proposition 2. Maximum Level Motif *is* **APX**-*hard.*

The following easy lemma will prove useful in the sequel.

Lemma 1. *Let $I = (T, \mathcal{M})$ be an instance of* Maximum Level Motif *and T' be a solution for instance I. One can compute in polynomial-time a solution T'' for I, such that (i) $|T''| \geq |T'|$ and (ii) T'' contains the root of T.*

Proof. Let $T' = (V', E')$ be a solution of Maximum Level Motif for instance I, and assume that T' does not contain the root r of T. Notice that T' must be a rooted subtree of T, and let $y \in V'$ be the root of T'. Now consider the unique path $P = (r, x'_1, \ldots, x'_p = y)$, from the root r to y. Two vertices x'_i and x'_j of P, $1 \leq i \neq j \leq p$, have distinct colors, since they belong to different levels of T. Moreover, each vertex x'_i, with $1 \leq i \leq p-1$, has a distinct color from each vertex $v \in V'$, since vertices x'_i and v belong to different levels of T. Define T'' as the subtree of T induced by the set of vertices $V'' = V' \cup \bigcup_{i=1}^{p-1} x_i$. Notice that T'' contains the root r of T, and by construction $|V''| \geq |V'|$. □

Aiming at applying the self-improvement technique we need to precisely define the product of two instances I_1 and I_2 of Maximum Level Motif. Let $I_1 = (T_1, \mathcal{M}_1)$ and $I_2 = (T_2, \mathcal{M}_2)$ be two instances of Maximum Level Motif, where $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are vertex-colored trees rooted at r_1 and r_2, respectively. The product $I_1 \times I_2$ is defined to be the instance $(T_{1,2}, \mathcal{M}_{1,2})$ where $T_{1,2} = (V_{1,2}, E_{1,2})$ is a rooted tree defined by $V_{1,2} = \{v_i(v_j) : v_i \in V_1 \wedge v_j \in V_2\}$ and $E_{1,2} = \{\{v_i(v_{j,1}), v_i(v_{j,2})\} : \{v_{j,1}, v_{j,2}\} \in E_2 \wedge v_i \in V_1\} \cup \{\{v_i(r_2), v_j(r_2)\} : \{v_i, v_j\} \in E_1\}$, and $\mathcal{M}_{1,2}$ is a motif defined by $\mathcal{M}_{1,2} = \{c_1(c_2) : c_1 \in \mathcal{M}_1 \wedge c_2 \in \mathcal{M}_2\}$. The tree $T_{1,2}$ is rooted at vertex $r_1(r_2)$. Informally, $T_{1,2}$ is obtained by replacing each vertex $v_i \in V_1$ by a copy of T_2, connecting these copies through their roots. As for the color of each vertex of $T_{1,2}$, if $v_i \in V_i$ is colored c_i and $v_j \in V_j$ is colored c_j then vertex $v_i(v_j) \in T_{1,2}$ is colored $c_i(c_j)$. Denote by $v_i[T_2]$ the subtree of $T_{1,2}$ isomorphic to T_2 rooted at $v_i(r_2)$. Write $V_{1,2,r} = \{v_i(r_2) : v_i \in V_1\}$. Observe that, by construction, the subtree of $T_{1,2}$ induced by $V_{1,2,r}$ is isomorphic to T_1.

Lemma 2. *Let $I_1 = (T_1, \mathcal{M}_1)$ and $I_2 = (T_2, \mathcal{M}_2)$ be two instances of* Maximum Level Motif*. Then $I_1 \times I_2$ is an instance of* Maximum Level Motif*.*

Proof. Write $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ and assume that T_1 and T_2 are rooted at r_1 and r_2, respectively. Let $I_1 \times I_2 = (T_{1,2}, \mathcal{M}_{1,2})$ and write $T_{1,2} = (T_{1,2}, E_{1,2})$. First, we show that $T_{1,2}$ is a rooted tree. Indeed, $T_{1,2}[V_{1,2,r}]$ is isomorphic to T_1 and each vertex in $V_{1,2} - V_{1,2,r}$ belongs to a subtree rooted at some $v_i(r_2) \in V_{1,2,r}$. Furthermore, $T_{1,2}$ is rooted by definition.

Now, we show that two vertices of $T_{1,2}$ have the same color only if they are at the same level in $T_{1,2}$. Let $u_1(u_2)$ and $v_1(v_2)$ be two vertices of $T_{1,2}$ such that $c(u_1(u_2)) = c(v_1(v_2)) = c_a(c_b)$. If $u_1 = v_1$ we are done so that we may now assume $u_1 \neq v_1$. Therefore, we must have $c(u_1) = c(v_1) = c_a$. Furthermore, observe that, by construction, all vertices in $u_1[T_2]$ and $v_1[T_2]$ are colored $c_a(c_x)$ for some color $c_x \in \mathcal{M}$. Consider the subtree $T_{1,2}[V_{1,2,r}]$ induced by $V_{1,2,r}$. Since $T_{1,2}[V_{1,2,r}]$ is isomorphic to T_1, each vertex $x_i(r_2) \in V_{1,2,r}$ has color $c(x_i)(c(r_2))$.

Now, since all vertices of $u_1[T_2]$ and $v_1[T_2]$ are colored $c_a(c_x)$, it follows that the root $x_i(r_2)$ of $u_1[T_2]$ and the root $x_j(r_2)$ of $v_1[T_2]$ have the same color $c_a(c(r_2))$. Then it follows that $x_i(r_2)$ and $x_j(r_2)$ must be at the same level l_1 of $T_{1,2}$, since they both belong to $V_{1,2,r}$ and $T_{1,2}[V_{1,2,r}]$ is isomorphic to T_1, where x_i and x_j must be both at level l_1.

Now, consider the subtrees $u_1[T_2]$ and $v_1[T_2]$ isomorphic to T_2. Recall, that vertices $u_1(u_2)$ and $v_1(v_2)$ of $T_{1,2}$ are both colored $c_a(c_b)$. As previously observed, all vertices $u_1(u_j)$ in $u_1[T_2]$ and $v_1[T_2]$ are associated with colors $c_a(c(u_j))$ for some $u_j \in V_2$. Since $I_2 = (T_2, \mathcal{M}_2)$ is an instance of Maximum Level Motif, vertices u_2 and v_2 must be at the same level l_2 in T_2 since $c(u_2) = c(v_2) = c_b$. Then, since $u_1[T_2]$ and $v_1[T_2]$ are both isomorphic to T_2, $u_1(u_2)$ and $v_1(v_2)$ are both at level l_2 in $u_1[T_2]$ and $v_1[T_2]$, respectively. It follows that both $u_1(u_2)$ and $v_1(v_2)$ are at level $l_1 + l_2$ in $T_{1,2}$.

Finally, let us consider the motif $\mathcal{M}_{1,2}$. By construction, $\mathcal{M}_{1,2}$ is a set, hence it is colorful. □

For any instance I of Maximum Level Motif, write $I^1 = I$ and $I^k = I \times I^{k-1}$ for all $k \geq 2$. According to Lemma 2, it follows by induction that I^k, $k \geq 1$, is an instance of Maximum Level Motif.

Lemma 3. *Let $I = (T, \mathcal{M})$ be an instance of* Maximum Level Motif *and let T_S be a solution for I. Then there exists a solution T_{S^k} for instance I^k such that $|T_{S^k}| \geq |T_S|^k$, for all $k \geq 1$.*

Proof. We prove the lemma by induction on k. The result is certainly valid for $k = 1$. Let $k \geq 2$ and assume that the lemma holds for all $1 \leq k' \leq k-1$. Let $T_S = (V_{T_S}, E_{T_S})$ be a solution of Maximum Level Motif for instance I, with $V_{T_S} = \{v_1, v_2, \ldots, v_z\}$. Observe that T_S is a subtree of T and that all vertices in V_{T_S} have distinct colors since $\mathcal{M}$ is colorful. By Lemma 1, we can assume that the root r of T is part of V_{T_S}. We now construct a solution T_{S^k} for instance I^k as follows.

First, consider the subtree of T^k which consists of the set $V_{T_S,r'}$ of vertices $v_1(r'), v_2(r'), \ldots, v_z(r')$, where each $v_i(r')$ is the root of a subtree of T^k isomorphic to T^{k-1}. Observe that, by construction, the set of vertices $V_{T_S,r'}$ induces a subtree $T^k[V_{T_S,r'}]$ of T^k. Since vertices $v_1, v_2, \ldots, v_z$ all have distinct colors in T, then it follows that $v_1(r'), v_2(r'), \ldots, v_z(r')$ have distinct colors as well. Let $v_i[T^{k-1}]$ and $v_j[T^{k-1}]$, $1 \leq i < j \leq z$, be two subtrees of T isomorphic to T^{k-1} rooted at $v_i(r')$ and $v_j(r')$, respectively. Observe that any two vertices $x \in v_i[T^{k-1}]$ and $y \in v_j[T^{k-1}]$ cannot have the same color, since $c(v_i) \neq c(v_j)$. Now, consider a subtree rooted at $v_i(r')$, with $1 \leq i \leq z$. By induction hypothesis, there is a solution $T_{S^{k-1}}$ of Maximum Level Motif over instance $I^{k-1} = (T^{k-1}, \mathcal{M}^{k-1})$, such that $|T_{S^{k-1}}| \geq |T_S|^{k-1}$. Notice that, by Lemma 1, we can assume that $T_{S^{k-1}}$ contains the root of T^{k-1}. Now we build solution T_{S^k}, by adding, for each $v_i(r')$, $1 \leq i \leq z$, a subtree of $v_i[T^{k-1}]$ isomorphic to $T_{S^{k-1}}$. Since T_S^k consists of $|T_S|$ such subtrees, it follows immediately that the inequality holds.

Finally, notice that the solution we have built is a feasible solution for Maximum Level Motif for instance I^k. First, T_S^k is connected by construction. Furthermore, each vertex of T_S^k has a distinct color. Indeed, we have shown that this holds for any two vertices that are not in the same subtree $v_i[T^{k-1}]$. By induction hypothesis, since $T_{S^{k-1}}$ is a feasible solution of Maximum Level Motif over instance I^{k-1}, it follows that two vertices that belong to the same subtree $v_i[T^{k-1}]$ must have distinct colors. $\square$

Lemma 4. *Let T_{S^k} be a solution of* Maximum Level Motif *for instance $I^k = (T^k, \mathcal{M}^k)$. Then, one can compute in polynomial-time a solution T_S for instance I such that $|T_S|^k \geq |T_{S^k}|$.*

Proof. We prove the lemma by induction on k. The result is certainly valid for $k = 1$. Let $k \geq 2$ and assume that the lemma holds for each $1 \leq k' \leq k-1$. Let $T_{S^k} = (V_{S^k}), E_{S^k}$ be a solution for Maximum Level Motif over instance I^k. According to Lemma 1, there is no loss of generality in assuming that the root of T^k is part of V_{S^k}. Then it follows that V_{S^k} contains vertices $x_1, \dots x_p$ of T^k, with $p \leq |T|$, so that at least one vertex in subtree $x_i[T^{k-1}]$ isomorphic to T^{k-1} belongs to T_{S^k}. For each x_i, $1 \leq i \leq p$, denote by $x_i[T_S^{k-1}]$ the subtree of $x_i[T^{k-1}]$ which is part of T_{S^k}. Let $x_{\max}[T_S^{k-1}]$ be a subtree of maximum size among the subtrees $x_i[T_S^{k-1}]$, $1 \leq i \leq p$. Let T_S^{k-1} be a subtree of T^{k-1} isomorphic to $x_{\max}[T_S^{k-1}]$. Notice that T_S^{k-1} is a solution of Maximum Level Motif over instance I^{k-1}. By induction hypothesis, we can compute in polynomial time a solution $T_{S'}$ over instance I such that $|T_{S'}|^{k-1} \geq |x_{\max}[T_S^{k-1}]|$. Denote now by T_p the subtree of T_{S^k} induced by $\{x_1 \dots x_p\}$. Now $|T_{S^k}| \leq |T_p||x_{\max}[T_S^{k-1}]| \leq |T_p||T_{S'}|^{k-1}$. If $|T_{S'}| \geq |T_p|$, then $T_S = T'_S$ and the lemma holds, since $|T_{S'}||T_{S'}|^{k-1} \geq |T_p||T_{S'}|^{k-1}| \geq |T_{S^k}|$. Otherwise, if $|T_{S'}| < |T_p|$, let T_S be the subtree of T isomorphic to T_p. It follows that $|T_p||T_p|^{k-1} > |T_p||T_{S'}|^{k-1} \geq |T_{S^k}|$.

Observe that T_S is a feasible solution of Maximum Level Motif over instance I. In the former case, when T_S is equal to $T_{S'}$, T_S is feasible by induction hypothesis. Consider the latter case, when T_S is equal to T_p. Let $x_1, \dots x_p$ be the vertices of T_p. Vertex x_i of T_p, $1 \leq i \leq p$, is associated to color $c_i(c(r), c(r), \dots, c(r))$, where $c(r)$ is the color associated to the root of T and $c_i \in \mathcal{M}$. Observe that, since $\mathcal{M}^k$ is colorful, $c_i \neq c_j$, when $i \neq j$, hence the vertices of T_S all have distinct colors. $\square$

We are now in position to state the main results of this section.

Theorem 1. *For any constant $\delta < 1$,* Maximum Level Motif *cannot be approximated within ratio $2^{\log^\delta n}$ in polynomial-time unless* **NP** $\subseteq$ **DTIME**$[2^{\text{poly} \log n}]$.

Proof. Assume that there exists a constant $\delta < 1$ such that Maximum Level Motif can be approximated within ratio $2^{\log^\delta n}$ in $\mathcal{O}(n^c)$ time, for some constant c. For any fixed $\varepsilon > 0$, let $k = \lceil (\frac{\log^{\delta} n}{\log(1+\varepsilon)})^{\frac{1}{(1-\delta)}} \rceil$. Given an instance I of Maximum Level Motif of size n, let I^k be the instance obtained by applying the product k times. Now, since the problem can be approximated within ratio $2^{\log^{\delta} n}$ in $\mathcal{O}(n^c)$ time, it follows that there is an algorithm for Maximum Level Motif for instance I^k with performance ratio $2^{\log^{\delta} n^k}$ that runs in $\mathcal{O}(n^{ck}) = \mathcal{O}(2^{\mathrm{poly}\log n})$ time. But, according to Lemmas 3 and 4, there is an algorithm for instance I with performance ratio $(2^{\log^{\delta} n^k})^{1/k} \leq (1+\varepsilon)$, and hence we have designed a PTAS algorithm for Maximum Level Motif. The result now follows from Proposition 2. □

Notice that Maximum Level Motif is a special case of Maximum Motif, and hence Theorem 1 holds for Maximum Motif.

Substituting the complexity hypothesis $\mathbf{NP} \subseteq \mathbf{DTIME}[2^{\mathrm{poly}\log n}]$ by the classical $\mathbf{P} = \mathbf{NP}$ yields the following result (proof - similar to that of Theorem 1 - omitted): no polynomial-time algorithm achieves a constant approximation ratio for Maximum Level Motif (*i.e.*, Maximum Level Motif is not in **APX**), unless $\mathbf{P} = \mathbf{NP}$. The same result holds also for the Maximum Motif problem.

4 Exponential-Time Algorithms

We give here two exact branch-and-bound algorithms for Maximum Motif in case the target graph is a tree. Let $I = (T, \mathcal{M})$ be an instance of Maximum Motif problem, where the target graph is a tree $T = (V, E)$.

Lemma 5. Maximum Motif *for trees of size n can be solved in $\mathcal{O}(1.62^n \operatorname{poly}(n))$ time. In case the motif is colorful, the time complexity reduces to $\mathcal{O}(1.33^n \operatorname{poly}(n))$.*

We briefly present the main ideas of the proof. First, both algorithms choose a vertex $r \in V$ (we assume w.l.o.g. that r is part of the optimal solution), and the tree T is rooted at r. Both algorithms rely on the fact that, once we have computed a set of vertices $V' \subseteq V$ that are part of the optimal solution, we can compute in polynomial time the maximum cardinality subset $L' \subseteq N(V')$, such that $\mathcal{C}(V') \cup \mathcal{C}(L') \subseteq \mathcal{M}$. Hence, we can assume that a branching occurs only at an internal vertex.

The first algorithm considers a candidate internal vertex v_x and branches in two sub-cases associated with v_x: (1) v_x is added to the solution, or (2) v_x is not added to the solution, and the subtree rooted at v_x is removed.

When $\mathcal{M}$ is colorful, we can assume that there exist two vertices that have the same color c. Indeed, if v_x is the only vertex colored $c(v_x)$, then the algorithm never branches on v_x. The algorithm branches in two sub-cases associated with vertex v_x: (1) v_x is added to the solution, and, for each $v_y \in V'$ colored $c(v_x)$, the subtree rooted at vertex v_y is removed, or (2) v_x is not added to the solution T_S, and then the subtree rooted at v_x is removed.

5 Parameterized Complexity

Fixed-parameter tractability plays a central role in parameterized complexity [6,14]. In this section we present two fixed-parameter tractable algorithms for Maximum Motif. We first describe the perfect family of hash functions used in both algorithms. Newt, we give an FPT algorithm in case the target graph is a tree and finally present a (slower) algorithm for the general case.

Consider an instance $I = (G, \mathcal{M})$ of Maximum Motif, where $G = (V, E)$ is a graph and $\mathcal{M}$ is a multiset of colors. For a color c_i of $\mathcal{M}$ and a subset $V' \subseteq V$, we denote by $m_{\mathcal{M}}(c_i)$ the number of occurrences of c_i in $\mathcal{M}$ and by $m_{V'}(c_i)$ the number of vertices in V' colored c_i. In the sequel, we assume that $m_{\mathcal{M}}(c_i) \leq m_V(c_i)$ since an occurrence of $\mathcal{M}$ in G has at most $\min\{m_{\mathcal{M}}(c_i), m_V(c_i)\}$ occurrences of color c_i. For a subset of vertices $V' \subseteq V$ and a submotif $\mathcal{M}' \subseteq \mathcal{M}$, we say that V' *violates* $\mathcal{M}'$ if $m_{\mathcal{M}'}(c_i) < m_{V'}(c_i)$ for some $c_i \in \mathcal{M}$.

Both algorithms are based on the color-coding technique [1]. We recall the basic definition of perfect hash functions. For a set S, a family F of functions from S to $\{1, 2, \ldots, k\}$ is *perfect* if for any $S' \subseteq S$ of size k, there exists an injective function $f \in F$ from S' to $\{1, 2, \ldots, k\}$. In the sequel, k denotes the size of a solution for Maximum Motif. Consider a family H of perfect hash functions from $\mathcal{M}$ to the set $\{1_H, 2_H, \ldots, k_H\}$ (we use the subscript H to emphasis that this set is related to the family H). Let $\mathcal{M}'$ be a submotif of size k and let $G' = (V', E')$ be the occurrence of $\mathcal{M}'$ in G. Since H is perfect, there exists an injective function $h \in H$ that assigns to each occurrence of a color in $\mathcal{M}'$ a distinct label in $\{1_H, 2_H, \ldots, k_H\}$.

Fix some function $h \in H$. For any $c_i \in \mathcal{M}$, denote by $S_H(c_i) \subseteq \{1_H, 2_H, \ldots, k_H\}$ the set of labels associated with occurrences of color c_i by function h. Furthermore, we associate with each vertex v colored c_i the set of labels $S_H(v) = S_H(c_i)$. Let $V' \subseteq V$, $L_H \subseteq \{1_H, \ldots, k_H\}$, then $\mathcal{C}(S_H, V', L_H)$ is defined as the family of sets $S_H(v) \cap L_H$, with $v \in V'$. Notice that $\mathcal{C}(S_H, V', L_H)$ may contain more occurrences of the same set of labels. For example, if $v_1, v_2 \in V'$ and $c(v_1) = c(v_2)$, then $(S_H(v_1) \cap L_H) = (S_H(v_2) \cap L_H)$. In case $L_H = \{1_H, \ldots, k_H\}$, we abbreviate $\mathcal{C}(S_H, V', L_H)$ by $\mathcal{C}(S_H, V')$.

Definition 1. *Let $\mathcal{C}(S_H, V', L_H)$ be a family of sets $S_H(v)$ with $v \in V'$ and $L_H \subseteq \{1_H, \ldots, k_H\}$, then $\mathcal{C}(S_H, V', L_H)$ is* feasible *if and only if there exists an injective function p from the sets of $\mathcal{C}(S_H, V', L_H)$ to L_H, so that, for each $S_H(v) \in \mathcal{C}(S_H, V', L_H)$, $p(S_H(v))$ is a label of $S_H(v) \cap L_H$.*

Consider now a family $\mathcal{C}(S_H, V')$ of sets associated with V'. Let c_i be a color of $\mathcal{M}$, then by construction $|S_H(c_i)| \leq m_{\mathcal{M}}(c_i)$. Hence, if $\mathcal{C}(S_H, V')$ is feasible, then V' does not violate $\mathcal{M}$.

We now present an FPT algorithm for the case the target graph is a tree $T = (V, E)$. Let $r \in V$, and we want to compute a solution $T' = (V', E')$ of Maximum Motif, so that $|V'| = k$ and $r \in V'$ (we run the algorithm for each $r \in V$.) Define r as the root of T and, for each internal vertex v of V, define a left-to-right ordering on the children of v. Assume that r is colored $c(r)$. Observe that, since r must belong to T', we can safely remove an occurrence of color $c(r)$

from $\mathcal{M}$. Furthermore, we assume that function h assigns to this occurrence of $c(r)$ label 1_H and that $S_H(r) = \{1_H\}$. Observe that there is no other vertex $u \in V - \{r\}$, so that $S_H(u)$ contains 1_H. We can now give the definition of the rightmost vertex of a subtree T' of T.

Definition 2. *Let $T' = (V', E')$ be a subtree of T. A vertex $v \in V'$ is defined to be the rightmost vertex of T' if and only if (i) v has no children in V' and (ii) for each vertex $u \in V'$ on the path from r to v, V' does not contains the right sibling of u.*

Now, consider a vertex $v \in V$ and a subset L_H of labels in $\{1_H, \dots, k_H\}$. Define $P_r[v, L_H]$ as follows:

$$P_r[v, L_H] = \begin{cases} 1 & \text{if there exists a subtree } T' = (V', E') \text{ of } T \text{ with } r \in V' \text{ and with} \\ & \text{rightmost vertex } v \text{ and such that } \mathcal{C}(S_H, V', L_H) \text{ is feasible,} \\ 0 & \text{otherwise.} \end{cases}$$

The recurrence to compute $P_r[v, L_H]$ is as follows.

$$P_r[v, L_H] = \bigvee_{u, L'_H} P_r[u, L'_H], \tag{1}$$

where u is either a descendant of a left sibling of v or the parent of v, and $L'_H = L_H - \{i_H\}$, for some $i_H \in S_h(v) \cap L_H$. Notice that $P_r[v, \{1_H\}] = 0$, for each $v \in V - \{r\}$, $P_r[r, \{1_H\}] = 1$, and that $P_r[r, \{i_H\}] = 0$ for each $i_H \in \{2_H, \dots, k_H\}$.

Lemma 6. *Given a labelling h of the motif $\mathcal{M}$, we can compute in $\mathcal{O}(n^2 2^k)$ time if there is a subtree T' of T of size k that matches a submotif $\mathcal{M}'$ of $\mathcal{M}$.*

Proof. We have to show that $P_r[v, L_H] = 1$ if and only if there exists a subtree $T' = (V', E')$ of T having root r, which is an occurrence of a submotif $\mathcal{M}'$ of $\mathcal{M}$ of size $|L_H|$. Since T' must contain r, we assume that L_H contains 1_H.

First, consider a subtree $T' = (V', E')$ with root r. Let v be the rightmost vertex of T'. From the definition of rightmost vertex, it follows that there is no child of v in V' and that there is no vertex in T' which is a right sibling of a vertex on the path from r to v. Denote by $T'' = (V'', E'')$ the tree obtained from T' by removing v. Let u be the rightmost vertex of T''. By definition of rightmost vertex, u is either the parent of vertex v, denoted by $p(v)$, or a descendant of a child v' of $p(v)$ in T'' (with v' a left sibling of v in T' by definition of rightmost vertex).

Let $\mathcal{M}' = \mathcal{C}(V')$ be the multiset of colors associated with the vertices of T'. Consider $\mathcal{C}(S_H, V', L_H)$, the collection of sets of labels in L_H assigned to V'. Notice that $\mathcal{C}(S_H, V', L_H)$ is feasible, as T' is a solution of Maximum Motif problem. It follows that there is an injective function p that assigns to each set $S_H(u)$, with $u \in V'$, a label i_H in $S_H(u)$. But then, function p assigns label $i_H \in S_H(v)$ to the set $S_H(v)$. It follows that the family of sets $S_H(u)$ with

$u \in (V' - \{v\})$ must be feasible when p assigns a label in set $\{1_H, \ldots, k_H\} - \{i_H\}$ to each set $S_H(u)$, with $u \in (V(T') - \{v\})$. Hence $P_r[v, L_H] = 1$.

Assume now that $P_r[v, L_H] = 1$. We will prove the results by induction. Since $P_r[v, L_H] = 1$, by Recurrence (1) it follows that there must exists a vertex $u \in V'$ and a label $i_H \in S_H(v)$, so that $P_r[u, L_H - \{i_H\}] = 1$. By induction hypothesis, it follows that there is a subtree of $T'' = (V'', E'')$ of T having root r, so that T'' has size $|L_H| - 1$, u is the rightmost vertex of T'' and $\mathcal{C}(S_H, V'', L_H - \{i_H\})$ is feasible. Hence, by construction, also $\mathcal{C}(S_H, V', L_H)$ is feasible. We will show that v is adjacent to a vertex of T''. By definition of rightmost vertex, u is either the parent of vertex v, denoted by $p(v)$, or a descendant of a child v' of $p(v)$ in T'' (with v' a left sibling of v in T'). In the former case clearly u and v are adjacent. In the latter case, that is u is not $p(v)$, since T'' must be rooted at r, $p(v)$ belongs to T'', hence v is adjacent to a vertex of T''.

Observe that, if $P[v, \{1_H, \ldots, k_H\}] = 1$, it follows that there is a subtree $T' = (V', E')$ containing the root of T, so that each $\mathcal{C}(V')$ is assigned a distinct label in $\{1_H, \ldots, k_H\}$. By construction V' does not violate $\mathcal{M}$, hence $\mathcal{C}(V')$ is a submotif of $\mathcal{M}$ of size k.

Now, we consider the time complexity of the algorithm. Observe that there exist $\mathcal{O}(n2^k)$ values of the form $P[v, K']$, with $v \in V(T)$ and $L'_H \subseteq \{1_H, \ldots, k_H\}$. Now, in order to compute value $P[v, K']$, we have to check at most $\mathcal{O}(nk)$ other values $P[u, K'']$. Hence the time complexity is $\mathcal{O}(n^2 k 2^k)$. $\square$

Observe that we have to choose $\mathcal{O}(n)$ possible roots. Furthermore, since the family of perfect hash functions has size $\mathcal{O}(\log n)\, 2^{\mathcal{O}(k)}$, it follows that the algorithm is $\mathcal{O}(k 2^k n^3 \log n)\, 2^{\mathcal{O}(k)}$ time.

Next we describe a parameterized algorithm when the instance of Maximum Motif consists in a graph $G = (V, E)$ and a motif $\mathcal{M}$. The algorithm for this case consists in combining two perfect families of hash functions, and then applying a strategy similar to that presented in [7,5].

Consider two different perfect families of hash functions: a family H from $\mathcal{M}$ to $\{1_H, \ldots, k_H\}$, as we have previously introduced in this section, and a family F from the set V to $\{1_F, \ldots, k_F\}$. By the property of the family of perfect hash functions, we know that there is a function $f \in F$ such that the vertices of G that belong to a solution of size k are associated with distinct labels of $\{1_F, \ldots, k_F\}$. Similarly, we know that there is a function $h \in H$ such that the occurrences of colors of $\mathcal{M}$ that belong to an optimal solution, are associated with different labels of $\{1_H, \ldots, k_H\}$. Observe that each family of perfect hash functions consists of $\mathcal{O}(\log n)\, 2^{\mathcal{O}(k)}$ functions. Hence, we can combine all the possible pairs (f, h) of functions, with $f \in F$ and $h \in H$, in $\mathcal{O}(\log^2 n)\, 4^{\mathcal{O}(k)}$ time.

Recall that, for each color $c_i \in \mathcal{M}$, $S_H(c_i)$ denotes the set of labels associated with occurrences of color c_i by function h, and that, given v is colored c_i, $S_H(v) = S(c_i)$. Now, for each $v \in V$ and for each subset $L \subseteq \{1_F, \ldots, k_F\}$, define $M_L(v)$ as the family of all sets of labels $H' \subseteq \{1_H, \ldots, k_H\}$ so that there exists an occurrence V', with $v \in V'$, where the set of labels in $\{1_F, \ldots, k_F\}$ that f assigns to V' is exactly L and such that $\mathcal{C}(S_H, V', H')$ is feasible. Now, we present a method called *Batch procedure* for computing $M_L(v)$, similar to that introduced in [7,5]. Assume that we have computed the family of sets $M_{L'}(v)$, with $L' \subseteq L \setminus f(v)$, we apply the following procedure.

Batch Procedure(L, v):

- Define C_H to be the family of all pairs (H', L') such that $H' \subseteq \{1_H, \ldots, k_H\} - \{i_H\}$ for some $i_H \in S_H(v_i)$, $L' \subseteq L \setminus \{f(v)\}$, and $H' \in M_{L'}(u)$ for some $u \in N(v)$.
- Run through all pairs of (H', L'), (H'', L'') in C_H and determine whether $H' \cap H'' = \emptyset$ and $H' \cup H'' \subseteq \{1_H, \ldots, k_H\} - \{i_H\}$, for some $i_H \in S_H(v_i)$, and whether $L' \cap L'' = \emptyset$. If there is such a pair, add $(H' \cup H'', L' \cup L'')$ to C_H and repeat this step. Otherwise, continue to the next step.
- Set $M_L(v)$ to be all the sets of labels $H' \cup \{i_H\}$, where $i_H \in S_H(v_i) - H'$, $(H', L') \in C_H$ and $L' = L \subseteq -\{f(v)\}$.

Lemma 7. *Given a vertex $v \in V$ and $L \subseteq \{1_F, \ldots, k_F\}$, the batch procedure computes correctly $M_L(v)$, assuming $M_{L'}(u)$ is given for each u adjacent to v and for each $L' \subseteq L \setminus \{f(v)\}$.*

Notice that function h assigns a distinct label in $\{1_H, \ldots, k_H\}$ to each occurrence of a color in a submotif $\mathcal{M}'$, with $|\mathcal{M}'| = k$. Consider $M_L(v) = \{1_H, 2_H, \ldots, k_H\}$ with $L = \{1_F, 2_F, \ldots, k_F\}$. The set of vertices in V' associated with labels $\{1_F, 2_F \ldots, k_F\}$ are then associated with colors having labels in $\{1_H, 2_H \ldots, k_H\}$. Hence, $C(V')$ does not violate $\mathcal{M}$.

Lemma 8. *Given labeling functions $h : \mathcal{M} \rightarrow \{1, \ldots, k\}$ and $f : V \rightarrow \{1, \ldots, k\}$, the batch procedure determines in $\mathcal{O}(2^{5k}kn^2)$ time whether there exists a solution of* Maximum Motif *of size k.*

Proof. First, we will show that a set $M_L(v)$ is computed by batch procedure in $\mathcal{O}(2^{4k}kn)$ time. The first step of batch procedure searches at most $2^k n$ families of subsets H' of labels in $\{1_H, \ldots, k_H\}$, for each $i_H \in S_H(v)$. Notice that $|S_H(v)| \leq k$. Each family consists of at most 2^k sets. Hence, the first step requires $\mathcal{O}(2^{2k}kn)$.

For the second step of the batch procedure, observe that there are at most 2^{2k} set of label-subset pairs H' and L', so the second step is repeated 2^{2k} times. Each iteration of this step can be computed in $\mathcal{O}(2^k n)$ time, hence the second step require $\mathcal{O}(2^{4k}kn)$ time. Accounting also for the third step, the overall time complexity for of one invocation of the batch procedure is $\mathcal{O}(2^{4k}k + 2^{2k}kn) = \mathcal{O}(2^{4k}kn)$.

According to Lemma 7, the batch procedure must be invoked at most $2^k n$ times in order to obtain $\mathcal{M}_L(v)$ for every $v \in V$ and every label subset $L' \subseteq \{1_F, \ldots, k_F\}$, hence the overall time complexity is $\mathcal{O}(2^{5k}kn^2)$. □

Since each perfect family of hash functions has size $\mathcal{O}(\log n)\, 2^{\mathcal{O}(k)}$, the overall time complexity of the algorithm is $\mathcal{O}(2^{5k} kn^2 \log^2 n)\, 4^{\mathcal{O}(k)}$.

References

1. Alon, N., Yuster, R., Zwick, U.: Color coding. Journal of the ACM 42(4), 844–856 (1995)
2. Betzler, N., Fellows, M.R., Komusiewicz, C., Niedermeier, R.: Parameterized algorithms and hardness results for some graph motif problems. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 31–43. Springer, Heidelberg (2008)
3. Bruckner, S., Hüffner, F., Karp, R.M., Shamir, R., Sharan, R.: Topology-free querying of protein interaction networks. In: Batzoglou, S. (ed.) Proc. 13th Annual International Conference on Computational Molecular Biology (RECOMB 2009). LNCS, vol. 5541, pp. 74–89. Springer, Heidelberg (2009)
4. Diestel, R.: Graph theory, 2nd edn. Graduate texts in Mathematics, vol. 173. Springer, Heidelberg (2000)
5. Dondi, R., Fertin, G., Vialette, S.: Weak pattern matching in colored graphs: Minimizing the number of connected components. In: Proc. 10th Italian Conference on Theoretical Computer Science (ICTCS), Roma, Italy, pp. 27–38. World Scientific, Singapore (2007)
6. Downey, R., Fellows, M.: Parameterized complexity. Springer, Heidelberg (1999)
7. Fellows, M., Fertin, G., Hermelin, D., Vialette, S.: Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 340–351. Springer, Heidelberg (2007)
8. Hein, J., Jiang, T., Wang, L., Zhang, K.: On the complexity of comparing evolutionary trees. Discrete Applied Mathematics 71, 153–169 (1996)
9. Jiang, T., Li, M.: On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 24, 1122–1139 (1995)
10. Karger, D., Motwani, R., Ramkumar, G.D.S.: On approximating the longest path in a graph. SIAM Journal on Computing 24, 1122–1139 (1995)
11. Kelley, B.P., Sharan, R., Karp, R.M., Sittler, T., Root, D.E., Stockwell, B.R., Ideker, T.: Conserved pathways within bacteria and yeast as revealed by global protein network alignment. Proceedings of the National Academy of Sciences 100(20), 11394–11399 (2003)
12. Koyutürk, M., Grama, A., Szpankowski, W.: Pairwise local alignment of protein interaction networks guided by models of evolution. In: Miyano, S., Mesirov, J., Kasif, S., Istrail, S., Pevzner, P.A., Waterman, M. (eds.) RECOMB 2005. LNCS (LNBI), vol. 3500, pp. 48–65. Springer, Heidelberg (2005)
13. Lacroix, V., Fernandes, C.G., Sagot, M.-F.: Motif search in graphs: application to metabolic networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) 3(4), 360–368 (2006)
14. Niedermeier, R.: Invitation to fixed parameter algorithms. Lecture Series in Mathematics and Its Applications. Oxford University Press, Oxford (2006)
15. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation and complexity classes. Journal of Computer and System Sciences 43, 425–440 (1991)

16. Scott, J., Ideker, T., Karp, R.M., Sharan, R.: Efficient algorithms for detecting signaling pathways in protein interaction networks. Journal of Computational Biology 13, 133–144 (2006)
17. Sharan, R., Ideker, T., Kelley, B., Shamir, R., Karp, R.M.: Identification of protein complexes by comparative analysis of yeast and bacterial protein interaction data. In: Proc. 8th annual international conference on Computational molecular biology (RECOMB), San Diego, California, USA, pp. 282–289. ACM Press, New York (2004)
18. Sharan, R., Suthram, S., Kelley, R.M., Kuhn, T., McCuine, S., Uetz, P., Sittler, T., Karp, R.M., Ideker, T.: Conserved patterns of protein interaction in multiple species. Proc. Natl Acad. Sci. USA 102(6), 1974–1979 (2005)

Fast RNA Structure Alignment for Crossing Input Structures

Rolf Backofen[1], Gad M. Landau[2], Mathias Möhl[3], Dekel Tsur[4], and Oren Weimann[5]

[1] Bioinformatics, Institute of Computer Science, Albert-Ludwigs-Universität, Freiburg, Germany
backofen@informatik.uni-freiburg.de

[2] Department of Computer Science, Haifa University, Haifa 31905, Israel
landau@cs.haifa.ac.il
Department of Computer and Information Science, Polytechnic Institute of NYU, Six MetroTech Center, Brooklyn, NY 11201-3840

[3] Programming Systems Lab, Saarland University, Saarbrücken, Germany
mmohl@ps.uni-sb.de

[4] Ben-Gurion University, Beer-Sheva, Israel
dekelts@cs.bgu.ac.il

[5] Massachusetts Institute of Technology, Cambridge, MA 02139, USA
oweimann@mit.edu

Abstract. The complexity of pairwise RNA structure alignment depends on the structural restrictions assumed for both the input structures and the computed consensus structure. For arbitrarily crossing input and consensus structures, the problem is NP-hard. For non-crossing consensus structures, Jiang et al's algorithm [1] computes the alignment in $O(n^2m^2)$ time where n and m denote the lengths of the two input sequences. If also the input structures are non-crossing, the problem corresponds to tree editing which can be solved in $O(m^2n(1 + \log \frac{n}{m}))$ time [2]. We present a new algorithm that solves the problem for d-crossing structures in $O(dm^2n \log n)$ time, where d is a parameter that is one for non-crossing structures, bounded by n for crossing structures, and much smaller than n on most practical examples. Crossing input structures allow for applications where the input is not a fixed structure but is given as base-pair probability matrices.

Keywords: RNA, sequence structure alignment, simultaneous alignment and folding.

1 Introduction

With the recent focus on non-protein-coding RNA (ncRNA) genes, interest in detecting novel ncRNAs has rapidly emerged. A recent screen on ncRNAs has detected more than 30000 putative ncRNAs in human genome [3], most of them with unknown function. Since the structure of RNA is evolutionarily more conserved than its sequence, predicting the RNA's secondary structure is the most important step towards its functional analysis [4].

The secondary structure of an RNA molecule can be calculated from its nucleotide sequence by determining a folding with minimal free energy [5,6,7,8,9]. Albeit this so-named thermodynamic approach is a success story in the analysis of RNA, it is known that predicting the secondary structure from a single sequence is error-prone, where the best available approaches can correctly predict only up to 73% of the base-pairs [10]. This situation can be improved by taking phylogenetic information into account, i.e., by predicting a common consensus structure from a whole set of evolutionary related RNA sequences.

There are several approaches for this problem (see [11] for an overview) which increase both in computational complexity as well as in the average quality. The simplest and fastest approach is to align the RNA sequences using a multiple sequence alignment, and then to fold the complete alignment using approaches like RNAalifold [12] and Petfold [13]. This has time complexity of $O(k^2n^2)$ for the pairwise alignment, and $O(n^3)$ for the final folding, which has to be applied only once on the complete alignment, where k is the number of sequences.

The second approach is to predict for all k sequences the minimum free-energy structure (with complexity $O(n^3)$), and then to perform progressive sequence-structure alignment whose complexity is dominated by the pairwise alignment steps. For a long time, the best complexity known for the pairwise alignment step was $O(n^3 \log(n))$ as given by the seminal work of Klein [14]. Just recently this has been improved to $O(n^3)$ [2]. However, this approach crucially depends on the quality of the initial structure prediction, which is error-prone.

Hence, the gold standard are Sankoff-like approaches [15,16,17,18,19] which *simultaneously* align and fold the sequences. However, as stated in [11], the Sankoff-approach requires "extreme amounts of memory and space" with a space complexity of $O(n^4)$ and a time complexity of $O(n^6)$. In [19], we improved this complexity to $O(n^4)$ time by aligning base-pair probability matrices. Basically, one is given two sets of weighted base-pairs that are possibly crossing, and the goal is to find the best common *nested* consensus structure for both sets, taking both base-pair weights and the associated RNA sequences into account.

In this work, we want to shorten the gap between sequence-structure alignment methods (with a complexity of $O(n^3)$), and the Sankoff-like approaches (with a complexity of $O(n^4)$ for alignment of base-pair probability matrices) for a practical application scenario. Basically, sequence-structure alignment approaches use exactly *one* structure per sequence as an input, whereas Sankoff-like approaches use *all* possible structures as an input. However, in many practical cases, one has a mixture of both, namely a main structure that allows for a small deviation. As shown in the example in Fig. 1, the alternative structures together form a crossing input structure, where the offset between crossing arcs is small. In this paper, we introduce a measurement for this deviation (d-crossing), and introduce an efficient algorithm with complexity $O(n^3 \log(n))$ given that the deviation is small (i.e., that the input base-pair probability matrix is d-crossing for a small constant d). Note that the crossing structure in Fig. 1 forms a two-page embedding (or is 2-colorable, as it is called in [20]), but our approach is not restricted to this class of structures.

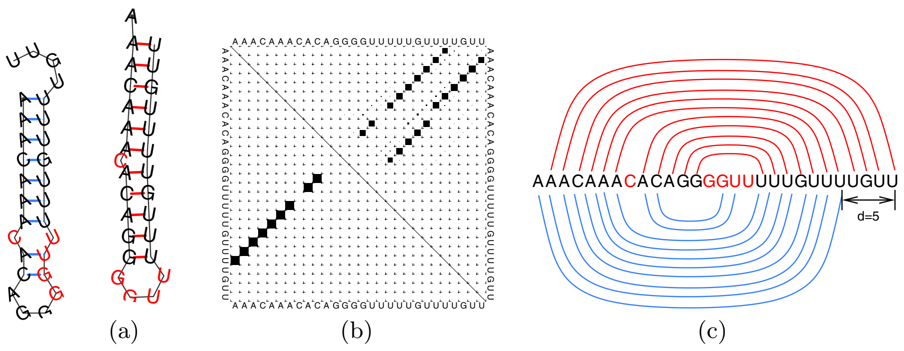


Fig. 1. (a) Two structures for the sequence AAACAAACACAGGGGUUUUUGUUUUGUU with similar free energy. The stem in the second sequence is shifted by 5 nucleotides. (b) associated base-pair probability matrix (upper triangle) and minimum free energy structure (lower triangle). The shifted stem is indicated by two parallel diagonals, a pattern often seen in RNA-structures. (c) both nested structures together form a crossing input. The outermost arcs of both structures are d-crossing for $d = 5$.

The fast available sequence structure alignment methods for non crossing input structures as in Klein [14] (a formal definition of non-crossing is given in Section 2) rely on a heavy path decomposition which was so far only available for tree-like structures. Our approach generalizes this to d-crossing structures.

2 Preliminaries

An *arc-annotated sequence* is a pair (S, P), where S is a string over the set of bases $\{A, U, C, G\}$ and P is a set of arcs (l, r) with $1 \leq l < r \leq |S|$ representing bonds between bases. We allow more than one arc to be adjacent to one base, but require that $|P| \in O(|S|)$, that is, on average each base is adjacent to only a constant number of arcs. We denote the i-th symbol of S by $S[i]$ and the substring from symbol i to symbol j with $S[i \ldots j]$. For an arc $p = (l, r)$, we denote its left end l and right end r by p^{L} and p^{R}, respectively. The *span* of p is defined as $\mathrm{span}(p) = p^{\mathrm{R}} - p^{\mathrm{L}} + 1$.

Two arcs p_1 and p_2 in an arc-annotated sequence (S, P) are *crossing* if $p_1^{\mathrm{L}} \leq p_2^{\mathrm{L}} \leq p_1^{\mathrm{R}} \leq p_2^{\mathrm{R}}$ or $p_2^{\mathrm{L}} \leq p_1^{\mathrm{L}} \leq p_2^{\mathrm{R}} \leq p_1^{\mathrm{R}}$. Two crossing arcs p_1 and p_2 are *d-crossing* if $|p_1^{\mathrm{L}} - p_2^{\mathrm{L}}| < d$ and $|p_1^{\mathrm{R}} - p_2^{\mathrm{R}}| < d$. An arc p_1 is *nested* in an arc p_2 if $p_2^{\mathrm{L}} < p_1^{\mathrm{L}} < p_1^{\mathrm{R}} < p_2^{\mathrm{R}}$. An arc p_1 *precedes* an arc p_2 if $p_1^{\mathrm{R}} < p_2^{\mathrm{L}}$. For every two arcs, either the two arcs are crossing, one of the arc is nested in the other, or one of the arc precedes the other. An arc-annotated sequence (S, P) containing crossing arcs is called *crossing*, otherwise *non-crossing* or *nested*. A *d-crossing sequence* is a crossing sequence in which every two crossing arcs are d-crossing.

3 Problem Definition

An *alignment* A of two arc-annotated sequences (S_1, P_1) and (S_2, P_2) is a set $A = A_{\text{match}} \uplus A_{\text{gap}}$. The set $A_{\text{match}} \subseteq [1,n] \times [1,m]$ of *match edges* satisfies that for all $(i,j),(i',j') \in A$, (1) $i > i'$ implies $j > j'$, and (2) $i = i'$ if and only if $j = j'$. Given A_{match}, the set of *gap edges* is implied as $A_{\text{gap}} := \{\,(i,-) | \, i \in [1,n] \wedge \nexists j.(i,j) \in A_{\text{match}}\,\} \cup \{\,(-,j) | \, j \in [1,m] \wedge \nexists i.(i,j) \in A_{\text{match}}\,\}$. A *consensus structure* for an alignment A is a matching $P \subseteq P_1 \times P_2$ that satisfies $(p_1, p_2) \in P \Rightarrow (p_1^{\mathrm{L}}, p_2^{\mathrm{L}}) \in A \wedge (p_1^{\mathrm{R}}, p_2^{\mathrm{R}}) \in A$. We require a consensus structure to be non-crossing (formally $\{(p_1,p_2),(p_1',p_2')\} \subseteq P \Rightarrow p_1$ and p_1' do not cross) and such that each base is adjacent to at most one arc (i.e. $\{(p_1,p_2),(p_1',p_2')\} \subseteq P \Rightarrow (p_1^{\mathrm{L}} = p_1'^{\mathrm{L}} \Leftrightarrow p_1^{\mathrm{R}} = p_1'^{\mathrm{R}})$).

Each alignment together with some consensus structure has an associated cost based on functions $\gamma_1 \in [1,n] \to \mathbb{N}$, $\gamma_2 \in [1,m] \to \mathbb{N}$, $\beta \in [1,n] \times [1,m] \to \mathbb{N}$, and $\alpha \in ([1,n])^2 \times ([1,m])^2 \to \mathbb{N}$. $\gamma_k(i)$ denotes the cost to align position i of sequence k to a gap, $\beta(i,j)$ the cost for a base match, i.e. cost to align position i of the first sequence to position j of the second sequence, provided arcs adjacent to i and j are not contained in the consensus structure, and $\alpha(p_a, p_b)$ denotes the cost to match arcs p_a, p_b in the consensus structure. The cost of an alignment A with consensus structure P, denoted $C_P(A)$, is

$$\sum_{(i,-)\in A} \gamma_1(i) + \sum_{(-,j)\in A} \gamma_2(j) + \sum_{((i,j),(i',j'))\in P} \alpha((i,j),(i',j')) + \sum_{(i,j)\in A'} \beta(i,j),$$

where A' is the set of all edges $(i,j) \in A$ such that there is no edge $(i',j') \in A$ for which $((i,j),(i',j')) \in P$ or $((i',j'),(i,j)) \in P$. Note that this scoring scheme can easily be instantiated with the edit distance scoring scheme of Jiang et al [1] if each base is adjacent to at most one arc. For this case we set $\gamma_1(i) = w_d + \psi_1(i)(\frac{w_r}{2} - w_d)$, $\gamma_2(j) = w_d + \psi_2(j)(\frac{w_r}{2} - w_d)$, $\beta(i,j) = \chi(i,j)w_m + (\psi_1(i) + \psi_2(j))\frac{w_b}{2}$, and $\alpha((i,j),(i',j')) = (\chi(i,j) + \chi(i',j'))\frac{w_{am}}{2}$ where ψ_1, ψ_2, χ, w_d, w_r, w_m, w_b, and w_{am} are defined as in [1]. However, we formulate the algorithm with the more general scoring scheme, since $\alpha((i,j),(i',j'))$ can be used to encode base pair weights which is more suitable in the presence of several adjacent arcs per base that represent alternative structures.

The *RNA structure alignment problem* is given two arc-annotated sequences (S_1, P_1) and (S_2, P_2), to find an alignment A and a consensus structure P such that $C_P(A)$ is minimal. For the remainder of this paper we fix two arc-annotated sequences (S_1, P_1) and (S_2, P_2) with $|S_1| = n$, $|S_2| = m$, $|P_1| \in O(n)$ and $|P_2| \in O(m)$ and assume that (S_1, P_1) is $d-$crossing. We assume w.l.o.g. that P_1 contains an arc $(1,n)$.

Arc annotated sequences are often classified as PLAIN, NEST, CROSS or UNLIM, as originally proposed in [21]. We solve for our scoring scheme the edit problem for a class that fully contains EDIT(NEST,NEST) and partially contains EDIT(UNLIM,UNLIM) (namely those instances where one structure is $d-$crossing and where on average each base is adjacent to only a constant number of arcs).

4 The Algorithm Recursion

The algorithm consists of two stages. The first stage computes the optimal costs to align certain fragments that are required for the second stage.

4.1 Stage 1

In the first stage, we compute a table M analogously to the recursion of Jiang et al. [1]. The entry $M[i, i', j, j']$ represents the minimal cost of an alignment between $(S_1[i \dots i'], P_1 \cap [i, i']^2)$ and $(S_2[j \dots j'], P_2 \cap [j, j']^2)$.

The base cases where $i' = i - 1$ and $j' = j - 1$ are initialized with $M[i, i - 1, j, j - 1] = 0$, the other entries are computed recursively as defined in Fig. 2. In the recursive computation, cases that rely on invalid items (i.e. where any of i, i', j, j' are not within their allowed range) are implicitly skipped. While Jiang et al's algorithm computes the entire alignment based on this recursion, we only compute entries of M for short fragments of the first sequence that have a length of at most $2d + 2$, i.e. for $1 \leq i \leq n$, $i - 1 \leq i' \leq \min(i + 2d + 1, n)$, $1 \leq j \leq m$, and $j - 1 \leq j' \leq m$.

$$M[i, i', j, j'] = \min \begin{cases} M[i, i' - 1, j, j'] + \gamma_1(i') & \text{I} \\ M[i, i', j, j' - 1] + \gamma_2(j') & \text{II} \\ M[i, i' - 1, j, j' - 1] + \beta(i', j') & \text{III} \\ \text{for all } p_1 = (i_0, i') \in P_1,\ p_2 = (j_0, j') \in P_2 \text{ with } i \leq i_0,\ j \leq j_0 & \text{IV} \\ \quad M[i, i_0 - 1, j, j_0 - 1] + M[i_0 + 1, i' - 1, j_0 + 1, j' - 1] + \alpha(p_1, p_2) & \end{cases}$$

Fig. 2. Recursion for the table M

4.2 Stage 2

For non-crossing input structures, the correspondence of these structures to trees allows for alignment methods that are asymptotically faster than the recursion used in the first stage [14,2]. In our approach we apply a similar technique, but since our input structures do not correspond to trees, we select a subset $P_T \subseteq P_1$ of the arcs.

The arcs in P_T do not cross and at most one of them is adjacent to each base. Hence, the arcs in P_T form a tree structure that guides the recursive decomposition during the computation of the alignment.

Construction of $\boldsymbol{P_T}$. Define the *inner d-range* of p as $I_d(p) = [p^{\mathrm{L}} + 1, p^{\mathrm{L}} + d - 1] \times [p^{\mathrm{R}} - d + 1, p^{\mathrm{R}} - 1]$. For a set of arcs $P \subseteq P_1$, the set $\mathrm{tree}\,(P)$ is defined recursively as follows. If $P = \emptyset$ or all arcs in P have span at most $2d$ then $\mathrm{tree}\,(P) = \emptyset$. Otherwise, let p be some arc in P with maximum span (ties are broken arbitrarily), and

$$\begin{aligned}\text{tree}\,(P) = &\{p\} \cup \text{tree}\left(P \cap [1, p^{\text{L}}-1]^2\right) \cup \text{tree}\left(P \cap [p^{\text{R}}+1, n]^2\right) \cup \\ &\text{tree}\left((P \cap [p^{\text{L}}+1, p^{\text{R}}-1]^2) \setminus I_d(p)\right).\end{aligned}$$

Lemma 1. *Every arc in P crosses at most one arc in* tree (P)*.*

Proof. Let p_1 and p_2 be two arcs in tree (P), and assume w.l.o.g. that $p_1^{\text{L}} < p_2^{\text{L}}$. We have that either p_2 is nested in p_1 or p_1 precedes p_2.

If p_2 is nested in p_1 then by the definition of tree (P), either $p_2^{\text{L}} - p_1^{\text{L}} \geq d$ or $p_1^{\text{R}} - p_2^{\text{R}} \geq d$. Suppose w.l.o.g. that $p_2^{\text{L}} - p_1^{\text{L}} \geq d$. Let p be an arc that crosses p_1. If $p^{\text{L}} \leq p_1^{\text{L}}$ then $|p^{\text{L}} - p_2^{\text{L}}| \geq p_2^{\text{L}} - p_1^{\text{L}} \geq d$, so p does not cross p_2. If $p^{\text{L}} > p_1^{\text{L}}$ then $p^{\text{L}} \leq p_1^{\text{L}} + d - 1 < p_2^{\text{L}}$ and $p^{\text{R}} \geq p_1^{\text{R}} > p_2^{\text{R}}$. Therefore, p_2 is nested in p, and in particular, p does not cross p_2.

If p_1 precedes p_2 then $p_2^{\text{L}} > p_1^{\text{R}} = p_1^{\text{L}} + \text{span}(p_1) - 1 \geq p_1^{\text{L}} + 2d$. Therefore, for every arc p, either $|p^{\text{L}} - p_1^{\text{L}}| \geq d$, or $|p^{\text{L}} - p_2^{\text{L}}| \geq d$. We conclude that p cannot cross both p_1 and p_2. □

Lemma 2. *An arc $p \in P$ satisfies $p \in I_d(p')$ for at most one arc $p' \in$* tree (P)*. If p does not cross an arc in* tree (P) *then $p \in I_d(p')$ for a unique arc $p' \in$* tree (P)*.*

Proof. To prove the first part of the lemma, let p_1 and p_2 be two arcs in tree (P) with $p_1^{\text{L}} < p_2^{\text{L}}$. Either p_2 is nested in p_1 or p_1 precedes p_2. If p_2 is nested in p_1 then either $p_2^{\text{L}} - p_1^{\text{L}} \geq d$ or $p_1^{\text{R}} - p_2^{\text{R}} \geq d$. In the former case, the intervals $[p_1^{\text{L}}+1, p_1^{\text{L}}+d-1]$ and $[p_2^{\text{L}}+1, p_2^{\text{L}}+d-1]$ are disjoints, and therefore $I_d(p_1) \cap I_d(p_2) = \phi$. Similarly, $I_d(p_1) \cap I_d(p_2) = \phi$ when $p_1^{\text{R}} - p_2^{\text{R}} \geq d$ or when p_1 precedes p_2. Thus, p cannot be both in $I_d(p_1)$ and $I_d(p_2)$.

We prove the second part of the lemma using induction on $|P|$. Let $P \subseteq P_1$ be a nonempty set of arcs, and let p be some arc in P that does not cross an arc in tree (P). Let p' be the maximum span arc in P that is chosen when computing tree (P). Recall that tree $(P) = \{p'\} \cup \text{tree}\left(P^1\right) \cup \text{tree}\left(P^2\right) \cup \text{tree}\left(P^3\right)$ where $P^1 = P \cap [1, p'^{\text{L}}-1]^2$, $P^2 = P \cap [p'^{\text{R}}+1, n]^2$, and $P^3 = (P \cap [p'^{\text{L}}+1, p'^{\text{R}}-1]^2) \setminus I_d(p')$. If $p \in I_d(p')$ we are done. Otherwise, since p does not cross p' and $p \notin I_d(p')$, we have that p is in some set P^i. Since $|P^i| < |P|$, by the induction hypothesis there is an arc $p'' \in \text{tree}\left(P^i\right)$ such that $p \in I_d(p'')$. □

We define $P_T = \text{tree}\,(P_1)$, and we call the arcs in P_T *tree arcs*. For every $p \in P_1$ we define $T(p)$ to be the unique tree arc p' such that p crosses p', if such arc exists. Otherwise, $T(p)$ is the unique tree arc p' such that $p \in I_d(p')$.

Lemma 3. *For every $p \in P_1$, $|p^L - T(p)^L| < d$ and $|p^R - T(p)^R| < d$.*

Proof. If p crosses $T(p)$ then the inequalities of the lemma are satisfied since (S_1, P_1) is d-crossing. Otherwise, from Lemma 2, $p \in I_d(T(p))$, and the inequalities of the lemma are satisfied by the definition of $I_d(\cdot)$. □

Lemma 4. *Let $p \in P_1$ and let $p' \in P_T$ such that $p' \neq p$ and p' is nested in $T(p)$. Then, p' is nested in p.*

Proof. Let p and p' be two arcs satisfying the conditions of the lemma. From the definition of $T(\cdot)$, p cannot cross p'. Moreover, from Lemma 3 and the fact that $\mathrm{span}(p) > 2d$, p' cannot precede p, or vice versa. □

For every tree arc $p \in P_T$ we select a tree arc denoted $\mathrm{hchild}(p)$ such that $\mathrm{hchild}(p)$ is nested in p and $\mathrm{span}(\mathrm{hchild}(p))$ is maximum (if there is such an arc). For $p \in P_T$ and $p \neq (1, n)$, define $\mathrm{parent}(p)$ to be the minimum span tree arc that p is nested in. We define $\mathrm{parent}((1, n)) = (1, n)$.

Recursion. For each $p \in P_T$ we build two tables L^p and R^p. Intuitively, one obtains the optimal alignments of the area below p or any arc crossing p by first extending the optimal alignments of $\mathrm{hchild}(p)$ or any arc crossing $\mathrm{hchild}(p)$ to the left (with L^p) and then to the right (with R^p). We compute the tables in an order such that for each p, L^p is computed before R^p and such that the tables of all $p' \in P_T$ that are nested in p are computed before the tables of p.

The table entries $L^p[i, i', j, j']$ and $R^p[i, i', j, j']$ have the same semantics as $M[i, i', j, j']$ and only differ in the domains of the indices i, i', j, j' and the recursions according to which they are computed. Let us first assume that $\mathrm{hchild}(p)$ is defined for p. Then, $L^p[i, i', j, j']$ is defined for

$$\begin{aligned}
\max(p^{\mathrm{L}} - d, \mathrm{parent}(p)^{\mathrm{L}}) \leq i &\leq \mathrm{hchild}(p)^{\mathrm{L}} \\
\mathrm{hchild}(p)^{\mathrm{R}} \leq i' &\leq \min(\mathrm{hchild}(p)^{\mathrm{R}} + d, p^{\mathrm{R}}) \\
1 \leq j &\leq m \\
j - 1 \leq j' &\leq m.
\end{aligned}$$

and for $R^p[i, i', j, j']$ the domains of j and j' are the same, but i and i' must satisfy

$$\begin{aligned}
\max(p^{\mathrm{L}} - d, \mathrm{parent}(p)^{\mathrm{L}}) \leq i &\leq \min(p^{\mathrm{L}} + d, \mathrm{hchild}(p)^{\mathrm{L}}) \\
\mathrm{hchild}(p)^{\mathrm{R}} \leq i' &\leq \min(p^{\mathrm{R}} + d, \mathrm{parent}(p)^{\mathrm{R}}).
\end{aligned}$$

If $\mathrm{hchild}(p)$ is not defined for p, no L^p table is computed and the R^p tables contain entries for

$$\begin{aligned}
\max(p^{\mathrm{L}} - d, \mathrm{parent}(p)^{\mathrm{L}}) \leq i &\leq p^{\mathrm{L}} + d \\
p^{\mathrm{L}} + d \leq i' &\leq \min(p^{\mathrm{R}} + d, \mathrm{parent}(p)^{\mathrm{R}})
\end{aligned}$$

and j, j' restricted as in the table R^p in the case where $\mathrm{hchild}(p)$ is defined. The domains of i and i' for the different cases are visualized in Fig. 3.

Computation of L^p. All entries $L^p[i, i', j, j]$ with $i \geq \max(\mathrm{hchild}(p)^{\mathrm{L}} - d, p^{\mathrm{L}})$ are initialized as $L^p[i, i', j, j'] = R^{\mathrm{hchild}(p)}[i, i', j, j']$. All other entries are computed according to the recursion shown in Fig. 5. Again cases relying on invalid items are implicitly skipped. The last three cases of the recursion are visualized in Fig. 4.

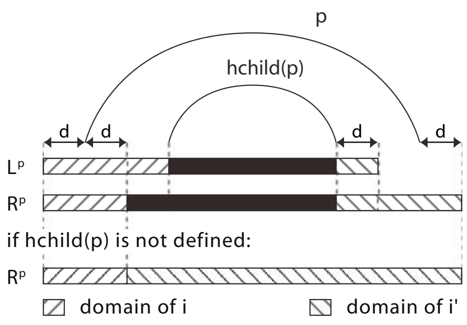


Fig. 3. Visualization of the domains for the different tables

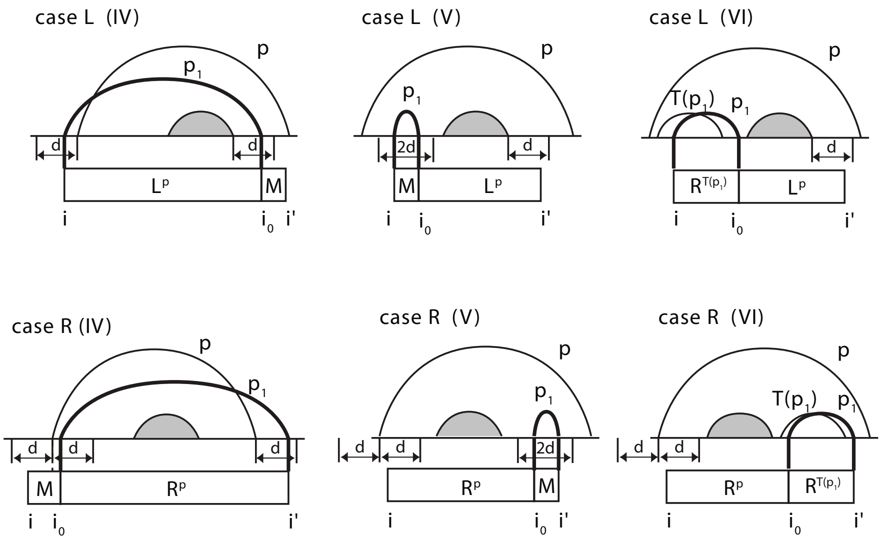


Fig. 4. Visualization of the recursion cases. The arc bounding the gray area denotes hchild(p)

Computation of R^p. The computation of the R^p tables is similar to the computation of the L^p tables, only that the fragments are extended to the right instead of to the left. If hchild(p) is defined, we initialize all entries with $i' \leq \min(\text{hchild}(p)^{\text{R}} + d, p^{\text{R}})$ as $R^p[i, i', j, j'] = L^p[i, i', j, j']$. All other items are computed according to the recursion shown in Fig. 6. If hchild(p) is not defined, we initialize all items with $i' = p^{\text{L}} + d$ as $R^p[i, i', j, j'] = M[i, i', j, j']$. The recursion for R^p in this case includes lines I, II, III, and V from Fig. 6.

Once the tables are computed, the actual alignment can be constructed using the usual backtrace technique.

$$L^p[i,i',j,j'] =$$
$$\min \begin{cases} L^p[i+1,i',j,j'] + \gamma_1(i) & \text{I} \\ L^p[i,i',j+1,j'] + \gamma_2(j) & \text{II} \\ L^p[i+1,i',j+1,j'] + \beta(i,j) & \text{III} \\ \text{for all } p_1 = (i,i_0) \in P_1,\ p_2 = (j,j_0) \in P_2 \text{ with } i_0 \le i',\ j_0 \le j', & \\ \text{and hchild}(p) \text{ is nested in } p_1 & \text{IV} \\ \quad L^p[i+1,i_0-1,j+1,j_0-1] + M[i_0+1,i',j_0+1,j'] + \alpha(p_1,p_2) & \\ \text{for all } p_1 = (i,i_0) \in P_1,\ p_2 = (j,j_0) \in P_2 \text{ with } i_0 \le i',\ j_0 \le j', & \\ \text{hchild}(p) \text{ is not nested in } p_1, \text{ and span}(p_1) \le 2d & \text{V} \\ \quad M[i+1,i_0-1,j+1,j_0-1] + L^p[i_0+1,i',j_0+1,j'] + \alpha(p_1,p_2) & \\ \text{for all } p_1 = (i,i_0) \in P_1,\ p_2 = (j,j_0) \in P_2 \text{ with } i_0 \le i',\ j_0 \le j', & \\ \text{hchild}(p) \text{ is not nested in } p_1, \text{ and span}(p_1) > 2d & \text{VI} \\ \quad R^{T(p_1)}[i+1,i_0-1,j+1,j_0-1] + L^p[i_0+1,i',j_0+1,j'] + \alpha(p_1,p_2) & \end{cases}$$

Fig. 5. The recursions for the table L^p

$$R^p[i,i',j,j'] =$$
$$\min \begin{cases} R^p[i,i'-1,j,j'] + \gamma_1(i) & \text{I} \\ R^p[i,i',j,j'-1] + \gamma_2(j) & \text{II} \\ R^p[i,i'-1,j,j'-1] + \beta(i,j) & \text{III} \\ \text{for all } p_1 = (i_0,i') \in P_1,\ p_2 = (j_0,j') \in P_2 \text{ with } i \le i_0,\ j \le j_0, & \\ \text{and hchild}(p) \text{ is nested in } p_1 & \text{IV} \\ \quad M[i,i_0-1,j,j_0-1] + R^p[i_0+1,i'-1,j_0+1,j'-1] + \alpha(p_1,p_2) & \\ \text{for all } p_1 = (i_0,i') \in P_1,\ p_2 = (j_0,j') \in P_2 \text{ with } i \le i_0,\ j \le j_0, & \\ \text{hchild}(p) \text{ is not nested in } p_1, \text{ and span}(p_1) \le 2d & \text{V} \\ \quad R^p[i,i_0-1,j,j_0-1] + M[i_0+1,i'-1,j_0+1,j'-1] + \alpha(p_1,p_2) & \\ \text{for all } p_1 = (i_0,i') \in P_1,\ p_2 = (j_0,j') \in P_2 \text{ with } i \le i_0,\ j \le j_0, & \\ \text{hchild}(p) \text{ is not nested in } p_1, \text{ and span}(p_1) > 2d & \text{VI} \\ \quad R^p[i,i_0-1,j,j_0-1] + R^{T(p_1)}[i_0+1,i'-1,j_0+1,j'-1] + \alpha(p_1,p_2) & \end{cases}$$

Fig. 6. The recursions for the table R^p

4.3 Correctness

Let (A, P) be an optimal alignment and consensus structure for the fragments corresponding to some table entry $M[i,i',j,j']$, $L^p[i',i,j',j]$, or $R^p[i,i',j,j']$ (note the swapped indices in the entry of L^p). In all recursions, lines I and II cover the cases where A aligns i' or j' to a gap. Line III covers the cases where $(i',j') \in A$ and no arcs of P are adjacent to i' or j'. Furthermore i' and j' can never be adjacent to arcs of the consensus structure whose other end is outside of the current fragment (due to the semantics of the table entries). Hence, the case that remains is where i' and j' are one end of some arc of the consensus structure whose other end is also contained in the current fragment. In the recursion for M, this case is covered in line IV, and in the recursions for L and R this case is further decomposed into subcases corresponding to lines IV to VI. In all those cases, the fragment is decomposed in the arc match (p_1, p_2), the fragment below the arc match and the fragment before it (or behind it, in the case of the table L). This decomposition is correct since the consensus structure is nested and hence cannot contain other arc pairs whose arcs cross p_1 and p_2 to connect the fragments before and below (p_1, p_2). It remains to show that in each case the table entries we recursively descend to exist.

Fix an arc $p \in P_T$ for which $\text{hchild}(p)$ is defined (the case where $\text{hchild}(p)$ is not defined is similar). Let $p_1 = (i_0, i')$ be an arc considered in lines IV to VI of the recursion for R^p.

Lemma 5. *p_1 does not cross* $\text{hchild}(p)$*.*

Proof. Since the case $i' \leq \min(\text{hchild}(p)^{\mathrm{R}} + d, p^{\mathrm{R}})$ is handled by the initialization of R^p, we have $i' > \min(\text{hchild}(p)^{\mathrm{R}} + d, p^{\mathrm{R}})$. Therefore, either $i' > \text{hchild}(p)^{\mathrm{R}} + d$ or $i' > p^{\mathrm{R}}$. In the former case we have from the assumption that (S_1, P_1) is d-crossing that p_1 does not cross $\text{hchild}(p)$. In the latter case we also have that p_1 does not cross $\text{hchild}(p)$ since otherwise, p_1 would also cross p, contradicting Lemma 1. □

By Lemma 5, either $\text{hchild}(p)$ is nested in p_1 or $\text{hchild}(p)$ precedes p_1. The case where $\text{hchild}(p)$ is nested in p_1 is handled in line VI of the recursion. In this case we have that either $T(p_1) = p$ or p is nested in p_1. In both cases we have that $i_0 \leq p^{\mathrm{L}} + d - 1$ (due to Lemma 3). From this inequality we obtain that $(i_0 - 1) - i = (i_0 - p^{\mathrm{L}}) + (p^{\mathrm{L}} - i) - 1 \leq 2d - 2$, so the entry $M[i, i_0 - 1, j, j_0 - 1]$ exists. Moreover, from the inequality $i_0 \leq p^{\mathrm{L}} + d - 1$ and the assumption that $\text{hchild}(p)$ is nested in p_1 we obtain that the entry $R^p[i_0 + 1, i' - 1, j_0 + 1, j' - 1]$ exists.

Now consider the case where $\text{hchild}(p)$ precedes p_1 which is handled in lines V and VI of the recursion. In both lines, the common entry $R^p[i, i_0 - 1, j, j_0 - 1]$ exists.

If $\text{span}(p_1) \leq 2d$ then the entry $M[i_0 + 1, i' - 1, j_0 + 1, j' - 1]$ exists since $(i' - 1) - (i_0 + 1) = \text{span}(p_1) - 3 \leq 2d - 3$. If $\text{span}(p_1) > 2d$ then we need to show that the entry $R^{T(p_1)}[i_0, i', j_0, j']$ exists. We have that $p_1^{\mathrm{R}} - p^{\mathrm{R}} > \text{span}(\text{hchild}(p)) > 2d$, and therefore p_1 does not cross p and $p_1 \notin I_d(p)$. It follows that $T(p_1) \neq p$. Therefore, $T(p_1)$ is nested in p, so the table $R^{T(p_1)}$ was already filled by the algorithm when the table R^p is filled. From Lemma 3 and Lemma 4 we conclude that the entry $R^{T(p_1)}[i_0, i', j_0, j']$ exists. The correctness arguments for the recursion for L^p are analogous.

4.4 Time Complexity

Let $d_k^R(i)$ (resp., $d_k^L(i)$) denote the number of arcs p in P_k with $p^{\mathrm{R}} = i$ (resp., $p^{\mathrm{L}} = i$). Let $d_k(i) = d_k^R(i) + d_k^L(i) + 1$. In stage 1, the time complexity for computing an entry $M[i, i', j, j']$ is $O((1 + d_1^R(i'))(1 + d_2^R(j'))) = O(d_1(i')d_2(j'))$. For fixed i' and j', the number of entries of the form $M[i, i', j, j']$ that are computed by the algorithm is $O(dm)$. Therefore, the time complexity of stage 1 is $O\left(\sum_{i'=1}^{n}\sum_{j'=1}^{m} dm \cdot d_1(i')d_2(j')\right) = O\left(dm \sum_{i'=1}^{n} d_1(i') \sum_{j'=1}^{m} d_2(j')\right) = O(dnm^2)$.

For $p \in P_T$, the time complexity of computing an entry $L^p[i, i', j, j']$ is $O((1 + d_1^L(i))(1 + d_2^L(j))) = O(d_1(i)d_2(j))$, and the time complexity of computing an entry $R^p[i, i', j, j']$ is $O((1 + d_1^R(i'))(1 + d_2^R(j'))) = O(d_1(i')d_2(j'))$. Consider some fixed arc p and fixed indices i and j. Let $c_{i,j}^p$ denote the number of computed entries of the form $L^p[i, i', j, j']$ or $R^p[i', i, j', j]$. Then stage 2 requires $O\left(\sum_{p \in P_T}\sum_{i=1}^{n}\sum_{j=1}^{m} c_{i,j}^p \cdot d_1(i)d_2(j)\right)$ time.

For every $p \in P_T$, $c_{i,j}^p \in O(dm)$ for all i and j. Assuming i and j are fixed, we now count the number of arcs $p \in P_T$ for which $c_{i,j}^p > 0$. Let p_0 be the minimum span tree arc such that $i \in [p_0^{\mathrm{L}}, p_0^{\mathrm{R}}]$. If p is a tree arc with $c_{i,j}^p > 0$ then p satisfies one of the following:

1. p is nested in p_0, $p^{\mathrm{R}} < i$, and p^{R} is maximal among all tree arcs that satisfy the previous two conditions.
2. p is nested in p_0, $p^{\mathrm{L}} > i$, and p^{L} is minimal among all tree arcs that satisfy the previous two conditions.
3. p_0 is nested in p and $i \notin [\mathrm{hchild}(p)^{\mathrm{L}}, \mathrm{hchild}(p)^{\mathrm{R}}]$.

There are at most two arcs of types 1 and 2 above. Let $p_0, p_1, \ldots, p_k$ be all the tree arcs of the third type, such that p_i is nested in p_{i+1} for all i. Since $\mathrm{span}(p_i) \le \mathrm{span}(\mathrm{hchild}(p_{i+1}))$, we have $\mathrm{span}(p_{i+1}) > 2 \cdot \mathrm{span}(p_i)$ for all i and therefore $k < \log_2 n$. Thus, the time complexity of stage 2 is $O\left(\sum_{i=1}^{n}\sum_{j=1}^{m} dm \log n \cdot d_1(i)d_2(j)\right) = O(dm^2 n \log n)$.

5 Conclusion

We presented an algorithm that computes the optimal sequence structure alignment for a nested consensus structure and crossing input structures. In practice, crossing input structures can be used to represent several suboptimal structures simultaneously, from which the alignment effectively selects the most appropriate one. On the theoretical side, we generalized the optimizations developed by Klein [14] to crossing input structures. In future work, we will try to incorporate also the space optimization and the optimization of Demaine et al [2].

Acknowledgements. This work is partially funded by DFG grant BA 2168/3-1, the Israel Science Foundation grant 35/05 and the Israel-Korea Scientific Research Cooperation.

References

1. Jiang, T., Lin, G., Ma, B., Zhang, K.: A general edit distance between RNA structures. J. Comput. Biol. 9(2), 371–388 (2002)
2. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 146–157. Springer, Heidelberg (2007)
3. Washietl, S., Hofacker, I.L., Lukasser, M., Huttenhofer, A., Stadler, P.F.: Mapping of conserved RNA secondary structures predicts thousands of functional noncoding RNAs in the human genome. Nat. Biotechnol. 23(11), 1383–1390 (2005)
4. Consortium, A.F.B., Backofen, R., Bernhart, S.H., Flamm, C., Fried, C., Fritzsch, G., Hackermuller, J., Hertel, J., Hofacker, I.L., Missal, K., Mosig, A., Prohaska, S.J., Rose, D., Stadler, P.F., Tanzer, A., Washietl, S., Will, S.: RNAs everywhere: genome-wide annotation of structured RNAs. J. Exp. Zoolog B Mol. Dev. Evol. 308(1), 1–25 (2007)
5. Waterman, M., Smith, T.: RNA secondary structure: a complete mathematical analysis. Math. Biosci. 42, 257–266 (1978)
6. Nussinov, R., Jacobson, A.: Fast algorithm for predicting the secondary structure of single-stranded RNA. Proc. Natl. Acad. Sci. 77(11), 6309–6313 (1980)
7. Zuker, M., Stiegler, P.: Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. Nucleic Acids Research 9(1), 133–148 (1981)
8. Akutsu, T.: Approximation and exact algorithms for RNA secondary structure prediction and recognition of stochastic context-free languages. Journal of Combinatorial Optimization 3, 321–336 (1999)
9. Wexler, Y., Zilberstein, C., Ziv-Ukelson, M.: A study of accessible motifs and RNA folding complexity. Journal of Computational Biology 14(6), 856–872 (2007)
10. Do, C.B., Woods, D.A., Batzoglou, S.: CONTRAfold: RNA secondary structure prediction without physics-based models. Bioinformatics 22(14), e90–e98 (2006)
11. Gardner, P.P., Giegerich, R.: A comprehensive comparison of comparative RNA structure prediction approaches. BMC Bioinformatics 5, 140 (2004)
12. Hofacker, I.L., Fekete, M., Stadler, P.F.: Secondary structure prediction for aligned RNA sequences. Journal of Molecular Biology 319(5), 1059–1066 (2002)
13. Seemann, S.E., Gorodkin, J., Backofen, R.: Unifying evolutionary and thermodynamic information for RNA folding of multiple alignments. Nucleic Acids Research (2008)
14. Klein, P.: Computing the edit-distance between unrooted ordered trees. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 91–102. Springer, Heidelberg (1998)
15. Sankoff, D.: Simultaneous solution of the RNA folding, alignment and protosequence problems. SIAM J. Appl. Math. 45(5), 810–825 (1985)
16. Mathews, D.H., Turner, D.H.: Dynalign: an algorithm for finding the secondary structure common to two RNA sequences. Journal of Molecular Biology 317(2), 191–203 (2002)
17. Havgaard, J.H., Lyngso, R.B., Stormo, G.D., Gorodkin, J.: Pairwise local structural alignment of RNA sequences with sequence similarity less than 40. Bioinformatics 21(9), 1815–1824 (2005)

18. Ziv-Ukelson, M.I., Gat-Viks, Y.W., Shamir, R.: A faster algorithm for RNA co-folding. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 174–185. Springer, Heidelberg (2008)
19. Will, S., Reiche, K., Hofacker, I.L., Stadler, P.F., Backofen, R.: Inferring non-coding RNA families and classes by means of genome-scale structure-based clustering. PLOS Computational Biology 3(4), e65 (2007)
20. Evans, P.A.: Finding common rna pseudoknot structures in polynomial time. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 223–232. Springer, Heidelberg (2006)
21. Evans, P.A.: Algorithms and Complexity for Annotated Sequence Analysis. PhD thesis, University of Alberta (1999)

Sparse RNA Folding: Time and Space Efficient Algorithms

Rolf Backofen[1], Dekel Tsur[2], Shay Zakov[2,⋆], and Michal Ziv-Ukelson[2]

[1] Albert Ludwigs University, Freiburg, Germany
backofen@informatik.uni-freiburg.de
[2] Department of Computer Science, Ben-Gurion University of the Negev, Israel
{dekelts,zakovs,michaluz}@cs.bgu.ac.il

Abstract. The classical algorithm for *RNA single strand folding* requires $O(nZ)$ time and $O(n^2)$ space, where n denotes the length of the input sequence and Z is a sparsity parameter that satisfies $n \leq Z \leq n^2$. We show how to reduce the space complexity of this algorithm. The space reduction is based on the observation that some solutions for subproblems are not examined after a certain stage of the algorithm, and may be discarded from memory. This yields an $O(nZ)$ time and $O(Z)$ space algorithm, that outputs both the cardinality of the optimal folding as well as a corresponding secondary structure. The space-efficient approach also extends to the related *RNA simultaneous alignment with folding* problem, and can be applied to reduce the space complexity of the fastest algorithm for this problem from $O(n^2m^2)$ down to $O(nm^2 + \tilde{Z})$, where n and m denote the lengths of the input sequences to be aligned, and $\tilde{Z}$ is a sparsity parameter that satisfies $nm \leq \tilde{Z} \leq n^2m^2$.

In addition, we also show how to speed up the base-pairing maximization variant of RNA single strand folding. The speed up is achieved by combining two independent existing techniques, which restrict the number of expressions that need to be examined in bottleneck computations of these algorithms. This yields an $O(LZ)$ time and $O(Z)$ space algorithm, where L denotes the maximum cardinality of a folding of the input sequence.

Additional online supporting material may be found at:
http://www.cs.bgu.ac.il/ zakovs/RNAfold/
CPM09_supporting_material.pdf

1 Introduction

The structure of RNA is evolutionarily more conserved than its sequence and is thus key to its functional analysis [1]. Unfortunately, although massive amounts of sequence data are continuously generated, the number of known RNA structures is still very limited since experimental methods, such as NMR and Crystallography, require expertise and long experimental time. Therefore, computational methods for predicting RNA structures are of great value [2,3,4].

⋆ Corresponding author.

RNA is typically produced as a single stranded molecule, which then folds upon itself to form a number of short base-paired stems. This base-paired structure is called the *secondary structure* of the RNA. The secondary structure almost always does not contain pseudoknots (i.e. crossing base pairs). Under the assumption that the structure does not contain pseudoknots, a model was proposed by Tinoco et al. [5] to calculate the stability (in terms of free energy) of a folded RNA molecule by summing all contributions from the stabilizing, consecutive base pairs and from the loop-destabilizing terms in the secondary structure. Based on this model, dynamic programming algorithms were suggested for computing the most stable structures [6,7,8,9,10], applying various scoring criteria such as the maximal number of base pairs [7] or the minimal free energy [8]. This optimization problem is formally denoted *RNA single strand folding*, and the time and space complexities of the classical algorithms for solving it are $O(n^3)$ and $O(n^2)$, respectively, where n denotes the length of the input RNA sequence. Recently, these were sped up to yield $O(nZ)$ time and $O(n^2)$ space [10] algorithms, where Z is a sparsity parameter that satisfies $n \leq Z \leq n^2$. We note that these algorithms are practical in the sense that the hidden constants are small. On a more theoretical front, Akutsu suggested an $O(D(n))$ algorithm for this problem [9], where $D(n)$ is the time for computing the distance product of two $n \times n$ matrices. The best current bound on $D(n)$ is $O(n^3 \log^3 \log n / \log^2 n)$ [11].

Another approach to RNA folding is the *simultaneous alignment with folding* (SAF for short) [12,13,14,15,16]. This approach consists of finding an optimal alignment between a set of RNA sequences, where an alignment score is evaluated with respect to some common folding of the input sequences. However, as stated in [17], even for the simple case where the input consists of only two sequences, this approach requires "extreme amounts of memory and space" with complexity of $O(n^2m^2)$ space and $O(n^3m^3)$ time, where n and m are the lengths of the input RNA sequences to be aligned. Thus, most existing practical implementations of this algorithm [13,14,16] use restricted versions of the original problem. Since these restrictions introduce another source of error, it is of utmost practical importance to the research on RNA to improve both the space and time complexities of the full version of SAF. A first non-heuristic speedup, which does not sacrifice the optimality of results, was recently described in [15]. This work extends the approach of [10] and yields an $O(nm\tilde{Z})$ time and $O(n^2m^2)$ space algorithm for the SAF problem, where $\tilde{Z}$ is a sparsity parameter that satisfies $nm \leq \tilde{Z} \leq n^2m^2$. However, experimental analysis of this algorithm indicates that the high memory requirements pose a major bottleneck in practice, both in constraining the lengths of the input sequences, as well as in exhausting the benchmark machine's memory, which in turn results in a page-fault slowdown.

Our contribution

(1.) Reducing the space requirements of RNA folding problems. In this work we focus on improving the space complexity of the base-pairing maximization variant of the RNA single strand folding problem [6,7,9]. The space requirement reduction is based on the observation that some solutions for subproblems are

Table 1. Time and space complexities of RNA folding algorithms

	Previous results		New results	
	Time	Space	Time	Space
Single strand base-paring maximization	$O(n^3)$[7] $O(nZ)$[10] $O(D(n))$[9]	$O(n^2)$	$O(LZ)$	$O(Z)$
Single strand energy minimization	$O(n^3)$[8] $O(nZ)$[10]	$O(n^2)$	$O(nZ)$	$O(Z)$
Simultaneous alignment with folding	$O(n^3m^3)$[12] $O(nm\tilde{Z})$ [15]	$O(n^2m^2)$	$O(nm\tilde{Z})$	$O(nm^2+\tilde{Z})$

not examined after a certain stage of the algorithms, and may be discarded from memory. This yields an $O(nZ)$ time and $O(Z)$ space algorithm for this problem. In addition to the optimal folding cardinality computation, we show a trace-back procedure which outputs a corresponding secondary structure. Note that it is an interesting challenge on its own to recover an optimal folding within the time and space complexity bounds of the space-reduced algorithm, since due to the sparse representation only partial information is kept. The presented strategy may also be extended to the score computation of a family of RNA folding algorithms, which includes algorithms for the energy minimization variant of the single strand folding problem [8,10] (improving the space complexity from $O(n^2)$ to $O(Z)$), as well as algorithms for SAF [12,15] (improving the space complexity from $O(n^2m^2)$ to $O(nm^2+\tilde{Z})$).

(2.) A sparse RNA single strand folding algorithm. We also describe a fast algorithm for the base pairing maximization variant of RNA single strand folding that exploits an additional sparsity parameter, based on the cardinality of the optimal folding. This is achieved by combining two independent techniques, which were previously used to reduce the number of sub-instance pairs that need to be considered by the algorithm. This combination yields the simultaneous exploitation of two key properties emerging from the formal definitions of these folding problems: the triangle inequality property, previously exploited in [10] and [15], as well as the monotonicity and unit-step properties, previously utilized in [18] for a related problem. The result is an $O(LZ)$ time and $O(Z)$ space algorithm, where L denotes the maximum cardinality of a folding of the input sequence and $n \le Z \le n(L+1)$.

We note that the algorithms described here are practical in the sense that the hidden constants are small. In the context of practical contribution, we also point out that our space complexity improvements are more significant than the time complexity improvements, since while the expected value of L is $\Theta(n)$ (assuming uniform character distribution), both Z and $\tilde{Z}$ were experimentally shown to be significantly less than n^2 and n^2m^2, respectively [10,15]. Furthermore, reducing the space complexity of the SAF problem is a key result in practice, as in the previous results the space complexity was typically the computational bottleneck [17,15].

Due to space constrains, figures, pseudocode and some omitted proofs are differed to an online supporting material document at `http://www.cs.bgu.ac.il/~zakovs/RNAfold/CPM09_supporting_material.pdf`.

2 Preliminaries

An RNA sequence is a sequence over the alphabet $\{A, C, G, U\}$. Each letter in an RNA sequence is also called a *base*. The bases A and U are called *complementary bases*, and so are the bases C and G[1]. For a base $\sigma \in \{A, C, G, U\}$, denote by $\overline{\sigma}$ the complementary base of σ. Fix henceforth an RNA sequence $S = s_1 s_2 \cdots s_n$. Denote by $S_{i,j}$ the subsequence $s_i \cdots s_j$ of S, where $S_{i,i-1}$ is defined to be an empty sequence.

Definition 1. *A* folding *F of a subsequence $S_{i,j}$ is a set of index pairs that satisfies the following:*
1. *For every $(k, l) \in F$, $i \leq k < l \leq j$, and $s_l = \overline{s_k}$.*
2. *There are no $(k, l), (k', l') \in F$, such that $k \leq k' \leq l \leq l'$.*

A pair $(k, l) \in F$ is called a *base-pair*. Say that index k is *paired* in a folding F if k appears in a base-pair in F, otherwise k is *unpaired* in F. Call an index q a *branch point* with respect to F if for all $(k, l) \in F$, either $l < q$ or $k \geq q$. We distinguish between two kinds of foldings of $S_{i,j}$: *co-terminus foldings* are foldings that include the base-pair (i, j), and *partitionable foldings* are those who do not include the base-pair (i, j). Note that for $j > i$, F is partitionable if and only if F has a branch point $i < q \leq j$. Denote by $|F|$ the *size* of a folding F, i.e. the number of base-pairs in F. The *single strand base-pairing maximization problem* was first addressed in [7]. The formal problem definition is given below.

Problem 1. Compute the maximum size of a folding of the instance sequence S.

Definition 2. *For a subsequence $S_{i,j}$, denote:*
1. *$L(i, j)$ is the maximum size of a folding of $S_{i,j}$.*
2. *$L^p(i, j)$ is the maximum size of a partitionable folding of $S_{i,j}$.*
3. *$L^c(i, j)$ is the maximum size of a co-terminus folding of $S_{i,j}$, or $-\infty$ if there is no such folding (if $j \leq i$ or $s_j \neq \overline{s_i}$).*

Call a folding F of $S_{i,j}$ for which $|F| = L(i, j)$, an *optimal* folding of $S_{i,j}$. In the rest of this paper, we use L instead of $L(1, n)$ whenever the context is clear.

3 RNA Folding via Base-Pairing Maximization

In this section we describe a recursive solution for the single strand base-pairing maximization problem, and present a technique for reducing its space complexity.

[1] For the sake of clarity, we disregard the possible "wobble" pairing between G and U. All presented results may be easily extended to include $G-U$ pairing as well.

This technique also extends to the single-strand RNA folding algorithms that are based on a thermodynamic model [2,3,4]. In addition, we suggest how to extend the space-reduction technique and apply it to the SAF problem [12,15].

3.1 A Recursive Solution

For a subsequence $S_{i,j}$ such that $j \leq i$, the only possible folding is the empty folding, and therefore $L(i,j) = 0$. The following equations show how to recursively compute $L(i,j)$ when $j > i$:

$$L(i,j) = \max\{L^p(i,j), L^c(i,j)\}. \tag{3.1}$$

$$L^c(i,j) = \begin{cases} L(i+1, j-1) + 1, & s_j = \overline{s_i}, \\ -\infty, & s_j \neq \overline{s_i}. \end{cases} \tag{3.2}$$

$$L^p(i,j) = \max_{i<q\leq j}\{L(i,q-1) + L(q,j)\}. \tag{3.3}$$

Note that the time complexity bottleneck in algorithms which implement the recursive computation of Equations 3.1 to 3.3 is due to the consideration of $O(n)$ branch points q in the computation of $L^p(i,j)$, according to Equation 3.3. In the rest of this section, as well as in Section 4, we describe techniques that reduce the number of branch points that need to be examined in this computation, and thus improve the time complexity of such algorithms. Due to Equations 3.1 and 3.3, the following (inverse) triangle inequality is sustained in the base-paring maximization problem:

Observation 1 (triangle inequality). *For every subsequence $S_{i,j}$ and for every $i < q \leq j$, $L(i,j) \geq L(i,q-1) + L(q,j)$.*

Based on the triangle inequality, Wexler et al. [10] observed that it is sufficient to examine only a subset of the branch points in order to compute $L^p(i,j)$. We present here a slightly different notation for the same concept.

Definition 3 (OCT). *A subsequence $S_{i,j}$ is* optimally co-terminus (OCT) *if $i = j$, or if every optimal folding of $S_{i,j}$ is co-terminus (that is, if $L(i,j) = L^c(i,j) > L^p(i,j)$).*

Call an index q for which $L^p(i,j) = L(i,q-1) + L(q,j)$ an *optimal branch point* with respect to $S_{i,j}$.

Lemma 1 (Wexler et al. [10]). *For every subsequence $S_{i,j}$, there is an optimal branch point q with respect to $S_{i,j}$ such that $S_{q,j}$ is an OCT.*

Define the following subset of branch points with respect to $S_{i,j}$:

$$Q_{i,j} = \{i < q \leq j : S_{q,j} \text{ is an OCT}\}.$$

The following equation restates Equation 3.3, based on Lemma 1, by restricting the branch points considered by the maximization term to those in $Q_{i,j}$.

$$L^p(i,j) = \max_{q\in Q_{i,j}}\{L(i,q-1) + L(q,j)\}. \tag{3.4}$$

We define the following sparsity measure of RNA sequences.

Definition 4. *For a subsequence $S_{i,j}$, $Z(i,j)$ is the number of subsequences of $S_{i,j}$ which are OCTs.*

In the rest of this paper, we use Z instead of $Z(1,n)$ whenever the context is clear. In the sparse case, only a small portion of the $O(n^2)$ subsequences of S are OCTs. In Section 4.1 we show that, in the base pairing maximization variant of the problem, Z is bounded by $n(L+1)$. For the minimum free energy problem variant, an estimation of the expected value of a parameter related to Z, based on a probabilistic model for polymer folding and measured by simulations, which shows that that Z is significantly smaller than $O(n^2)$, can be found in [10].

Previous algorithms for the base-pairing maximization problem were presented by Nussinov and Jacobson [7] and by Wexler et al. [10][2]. Both algorithms are dynamic programming algorithms that perform a bottom-up computation of the recurrence described in this section, where the Nussinov-Jacobson algorithm uses Equation 3.3 for the computation of $L^p(i,j)$, and the Wexler et al. algorithm improves it by using Equation 3.4. These algorithms compute the upper triangle of a table $M_{n \times n}$, where each cell $M[i,j]$ stores the value $L(i,j)$. The entries of M are traversed in an order which guarantees that all values that are needed for the computation of $M[i,j] = L(i,j)$, according to the recurrence formula, are computed and stored in M prior to the computation of $M[i,j]$. Upon termination, $M[1,n]$ holds the value L. The time complexity of the algorithm by Nussinov and Jacobson is $O(n^3)$, whereas that of the algorithm by Wexler et al. is $O(nZ)$. Both algorithms use $O(n^2)$ space.

3.2 A Space Efficient Algorithm

Our space reduction strategy is based on the observation that some of the values stored by the algorithm of Wexler et al. [10] are not necessary throughout the complete run of the algorithm. In the following lemma we characterize the values that need to be maintained in memory for the computation of $L(i,j)$.

Lemma 2. *For a subsequence $S_{i,j}$, it is possible to compute $L(i,j)$ by examining only those values $L(a,b)$, where $i \leq a < b \leq j$ and $b-a < j-i$, which sustain that either $a = i$, $a = i+1$, or $S_{a,b}$ is an OCT.*

Proof. Immediate from Equations 3.1, 3.2 and 3.4. □

Consider a dynamic programming algorithm which fills the table M by traversing its entries row by row from bottom to top, and each row from left to right. Lemma 2 implies that at the stage where $M[i,j]$ is computed, it is sufficient to keep only the values in the currently computed i-th row, the values in the recently computed $(i+1)$-th row, and values in entries which correspond to

[2] [10] deals with the more realistic *energy minimization* variant of the problem. For clarity, we project their notions on the simpler *base-paring maximization* variant discussed here.

OCT subsequences of S. Thus, there is no need to maintain the complete table M in memory, rather, at each stage, entries which are guarantied not to be further examined by the algorithm may be discarded. This yields a total space complexity of $O(n+Z) = O(Z)$. Note that the computation of each entry $M[i,j]$ requires $O(|Q_{i,j}|)$ operations, due to the consideration of the branch point set $Q_{i,j}$ (these sets are maintained as lists in order to allow an efficient traversal, as explained in [10]). Since $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|Q_{i,j}| \le \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|Q_{1,j}| \le \sum_{i=1}^{n-1} Z < nZ$, the running time of the algorithm is $O(nZ)$. Fig. 2 and Alg. 1 in the online supporting material illustrate and give the pseudo code of the above described algorithm. Its time and space complexities are summarized in the following lemma.

Lemma 3. *Given an RNA sequence S of length n, there is an algorithm which computes $L(1,n)$ in $O(nZ)$ time and $O(Z)$ space.*

3.3 Folding Reconstruction

In addition to computing the optimal folding score of a given RNA sequence, it is often of interest to report at least one optimal folding. Some well known standard techniques for reporting an optimal folding apply trace-back procedures over the folding score matrix M, in $O(n^2)$ time [19]. In this section we show how to reconstruct one optimal folding, without exceeding the time and space complexities of our folding algorithm. Note that this is a challenging task, as the classical trace-back algorithm requires the availability of the full table M, while our algorithm stores only partial information.

Assume that the full table M is given, with annotated OCT subsequences. The basic recursive folding reconstruction algorithm [19] could be modified as follows to utilize the OCT subsequences:

1. For $j \le i$, the only (optimal) folding of $S_{i,j}$ is the empty folding, and the algorithm halts without reporting any base-pair.
2. For $j > i$, if $S_{i,j}$ is an OCT, the algorithm reports the pair (i,j) and is called recursively on the subsequence $S_{i+1,j-1}$.
3. Otherwise, $S_{i,j}$ is partitionable, and therefore the algorithm finds an index $q \in Q_{i,j}$ for which $M[i,j] = M[i,q-1] + M[q,j]$ and then continues by computing an optimal folding of $S_{i,q-1}$ and of $S_{q,j}$. An optimal folding of $S_{i,q-1}$ is obtained by calling the algorithm recursively with the sub-instance $S_{i,q-1}$. As for computing an optimal folding of $S_{q,j}$, note that $S_{q,j}$ is an OCT, and consider the two cases, where either $q = j$ or $q < j$. If $q = j$, then there is no need for another recursive call. Otherwise $q < j$, and an optimal folding of $S_{q,j}$ is obtained by first reporting the base-pair (q,j) and then calling the algorithm recursively with the sub-instance $S_{q+1,j-1}$.

Time complexity analysis of the trace-back algorithm on the full table M. When calling the above algorithm to compute the folding traceback of $S_{i,j}$, recursive calls with three different subsequences could be initiated at the top level:

$S_{i-1,j-1}$, $S_{q+1,j-1}$ and $S_{i,q-1}$, thus index j is eliminated from further consideration as an end index. Therefore, each recursive call is performed with a different end index j, and altogether there are at most n recursive calls in the whole computation. For a recursive call in which the end index is j, at most $O\left(|Q_{1,j}|\right)$ operations are preformed in finding a $q \in Q_{i,j}$ for which $M[i,j] = M[i,q-1]+M[q,j]$. Since $\sum\limits_{1 \leq j \leq n} |Q_{1,j}| \leq Z$, the total running time is $O(Z)$.

We next turn to address the challenge of reconstructing an optimal folding from the sparse table M computed in Section 3.2. The above described algorithm cannot be applied directly in this case, due to the fact that when the algorithm needs to find $q \in Q_{i,j}$ for which $M[i,j] = M[i,q-1]+M[q,j]$, the values $M[i,j]$ and $M[i,q-1]$ may have been discarded from memory (while $M[q,j]$ is maintained in memory since $S_{q,j}$ is an OCT). In order to overcome this difficulty we adopt a similar approach as of the algorithm of Hirschberg [20], namely performing on-demand value re-computations of discarded entries. Thus, it remains to show how to recover such deleted entries.

Lemma 4. *Given the sparse table M that contains folding scores for OCT subsequences, there is an algorithm which recovers the set of entries $M[i,i+1]$, $M[i,i+2], \ldots, M[i,j]$, for a pair of given indices i and j, in $O(Z)$ time.*

Proof. The entries of the form $M[i,j']$ which have been discarded from memory correspond to partitionable subsequences, where $L\left(i,j'\right) = L^p\left(i,j'\right)$, and thus may be recomputed based solely on Equation 3.4. Observe that this computation examines only entries of the form $M[i,q]$ for $q < j'$, and $M[q,j']$ for OCT subsequences $S_{q,j'}$. Re-computing the entries of the ith row from left to right guaranties that upon computing $M[i,j']$, all necessary values for the computation of $L^p\left(i,j'\right)$ are already stored in M. For each $i < j' \leq j$, there are $O(|Q_{1,j'}|)$ operations performed along this computation, due to the consideration of branch points in the set $Q_{i,j'}$. As before, summing this expression over all $i < j' \leq j$ accumulates to $O(Z)$. □

We next show that, throughout the full run of the algorithm, the process of restoring row entries is applied to $O(L)$ distinct start indices. Consider the case where the trace-back algorithm is applied on $S_{i,j}$ and assume that the set of entries $M[i,i+1], M[i,i+2], \ldots, M[i,j]$ was already previously restored. Note that a recursive call with a sub-instance of the form $S_{i,q-1}$ does not require the restoration of the entries $M[i,i+1], M[i,i+2], \ldots, M[i,q-1]$, as (by the assumption) they have already been restored and are maintained in M. The other two possible recursive calls with sub-instances of the form $S_{i+1,j-1}$ or $S_{q+1,j-1}$, do require re-computation of entries in M (in rows $i+1$ or $q+1$, correspondingly). However, observe that each call of the latter kind is preceded by a detection of a base-pair. Since throughout the full run of the algorithm only L base pairs are detected, we get that the row entry recovery only needs to be executed L times (in addition to the recovery of $M[1,1], M[1,2], \ldots, M[1,n]$ during initialization). Thus, according to Lemma 4, the entry value recovery contributes an additional $O(LZ)$ factor to the total time complexity of the trace-back algorithm.

Furthermore, note that upon performing such a re-computation of an entry set, there is no need to further maintain the values in $M[i, i+1], M[i, i+2], \ldots, M[i, j]$ in the case where $S_{i,j}$ is co-terminus, nor to keep the values in $M[i, q], M[i, q+1], \ldots, M[i, j]$ in the case where $S_{i,j}$ is partitionable. This allows to discard these values from memory before the re-computation of the entry set for the corresponding sub-instance, guaranteeing that at each stage, at most n recovered entries are maintained in the sparse table M, in addition to the already existing OCT corresponding entries. Therefore, the space complexity of the trace-back algorithm remains $O(Z + n) = O(Z)$.

Alg. 2 in the online supporting material implements the efficient trace-back scheme.

Lemma 5. *There is an algorithm which, given the sparse table M that contains folding scores for all OCT subsequence of S, computes an optimal folding of S in $O(LZ)$ time and $O(Z)$ space.*

3.4 Extending the Space Reduction to Simultaneous Alignment with Folding

The goal of the SAF problem is to find a multiple sequence alignment and a common folding of the aligned sequences, which optimizes some score function. For simplicity, we assume the problem instance consists of two sequences. Similarly to single RNA strand folding algorithms, the basic dynamic programming algorithm for the SAF problem [12] computes the scores for all sub-instances of its input instance, and then combines these values to resolve the score of the full input instance. Given an instance of the problem - a pair of RNA sequences S and T, the algorithm maintains the scores of sub-instances $(S_{i,j}, T_{i',j'})$ in a four-dimensional table N (see Fig. 3). For $|S| = n$ and $|T| = m$, we depict N as an $n \times n$ "super table", in which each entry $N_{i,j}$ corresponds to an internal table of size $m \times m$, where the combined alignment-with-folding score of the sub-instance $(S_{i,j}, T_{i',j'})$ is stored in the entry $N_{i,j}[i', j']$. The time-complexity of the basic dynamic programming algorithm for the SAF problem is dictated by the need to compute all $O(n^2m^2)$ sub-instances, where each such computation involves the consideration of a set of $O(nm)$ competing branch point index pairs (*i.e.* all (q, q') such that $i < q \leq j$ and $i' < q' \leq j'$). This yields a total time complexity of $O(n^3m^3)$.

Recently, [15] extended the approach of [10] and applied it to speed up SAF by reducing the number of branch points that need to be considered in the main recursion for the SAF score computation. Similarly to the concept of OCT sequences, it is possible to define *OCT-aligned* sequence pairs, where the pair $(S_{i,j}, T_{i',j'})$ is OCT-aligned if, in every optimal alignment-with-folding of $(S_{i,j}, T_{i',j'})$ the bases s_i and $t_{i'}$ are aligned to each other, the bases s_j and $t_{j'}$ are aligned to each other, and the common folding is co-terminus. Using this formulation to describe the results of [15], it was shown that it is sufficient to examine branch point pairs (q, q') such that the sequences $S_{q,j}$ and $T_{q',j'}$ are

OCT-aligned, thus reducing the number of examined branch points and improving the running time of the algorithm. This extension yields an $O(nm\tilde{Z})$ time and $O(n^2m^2)$ space algorithm for the SAF problem, where $\tilde{Z}$ is the number of OCT-aligned sub-instances, and $nm \leq \tilde{Z} \leq n^2m^2$ (in practice, $\tilde{Z}$ is expected to be significantly smaller than $O(n^2m^2)$ [15]).

Applying an observation similar to Observation 2, an algorithm is suggested here which, upon the computation of entry $N_{i,j}[i', j']$, queries only those entries which correspond to OCT-aligned sub-instances, in addition to entries in rows i and $i+1$ of the "super table" N. The space complexity of SAF is thus reduced to $O(nm^2 + \tilde{Z})$ (w.l.o.g. $m \leq n$). In the extended version of this paper we describe in detail how to extend the space-reduction technique described in Section 3.2 to the four-dimensional matrix computed by the SAF algorithm [15]. An intuitive explanation can be found in Fig. 3.

Lemma 6. *There is an algorithm that computes the simultaneous alignment with folding of two RNA sequences S and T in $O(nm\tilde{Z})$ time and $O(nm^2 + \tilde{Z})$ space, where $n = |S|$, $m = |T|$, and w.l.o.g. $m \leq n$.*

4 Utilizing Step Characterization

In this section we take advantage of a step characterization of the single strand base-pairing maximization problem in order to improve the running time of the algorithms which compute it. Based on this approach, in Section 4.1 we describe an improvement to Alg. 1 which reduces its running time from $O(nZ)$ to $O\left(n^2 + LZ\right)$, and then in Section 4.2 we further reduce it to $O(LZ)$. Both algorithms have the same space complexity as Alg. 1, which is $O(Z)$.

Let $S_{i,j}$ be a subsequence of S. For $L' = L(i+1, j)$ or $L' = L(i, j-1)$, it is straightforward to show that $L' \leq L(i,j) \leq L' + 1$. Therefore, we get the following observation:

Observation 2. *For every $1 \leq k \leq n$, the sequence $L(k,k), L(k,k+1), \ldots, L(k,n)$, as well as the sequence $L(k,k), L(k-1,k), \ldots, L(1,k)$ are monotonically non-decreasing with unit steps in the range $0 - L$.*

The above observation implies a bound on Z, as follows.

Lemma 7. *The value of Z satisfies $n \leq Z \leq n(L+1)$.*

Proof. Every OCT subsequence $S_{i,j}$ satisfies that either $i = j$, or $L(i,j) > L(i+1,j)$. Hence, according to Observation 2, there are at most $L+1$ OCT subsequences that end with a given index j, and at therefore there are most $n(L+1)$ OCT subsequences of S. □

4.1 An $O(n^2 + LZ)$ Algorithm

Similarly to the previously presented technique for restricting the set of examined branch points in the computation of $L^p(i,j)$, we next show another dominance relation which can be utilized to further constrain the set of branch points examined in Equation 3.3.

Definition 5 (step sequence). *Call a subsequence $S_{i,j}$ a* step sequence *if in every optimal folding of $S_{i,j}$ the base i is paired.*

Observe that $S_{i,j}$ is a step sequence if and only if $q = i+1$ is not a branch point in any of the optimal foldings of $S_{i,j}$, i.e. $L(i,j) > L(i,i)+L(i+1,j) = L(i+1,j)$ (hence the term “step”). Also note that any OCT subsequence of length greater than 1 is a step sequence, though the opposite is not necessarily true. In the following Lemma we further restrict the branch points which need to be examined in a recursive computation of $L^p(i,j)$.

Lemma 8. *For any subsequence $S_{i,j}$ such that $j > i$, there is an optimal branch point q with respect to $S_{i,j}$ such that either $q = i+1$, or $S_{i,q-1}$ is a step sequence and $S_{q,j}$ is an OCT.*

Proof. If $q = i+1$ is an optimal branch point with respect to S, the lemma holds. Otherwise, $L^p(i,j) > L(i,i)+L(i+1,j) = L(i+1,j)$. According to Lemma 1, there is an optimal branch point $i+1 < q \leq j$ such that $S_{q,j}$ is an OCT. Therefore, $L(i,q-1)+L(q,j) = L^p(i,j) > L(i+1,j) \overset{\text{Obs. 1}}{\geq} L(i+1,q-1)+L(q,j)$. It follows that $L(i,q-1) > L(i+1,q-1)$, hence $S_{i,q-1}$ is a step sequence. □

Define the following subset of branch points with respect to $S_{i,j}$:

$$P_{i,j} = \{i+1\} \cup \{i+1 < q \leq j : S_{i,q-1} \text{ is a step sequence and } S_{q,j} \text{ is an OCT}\}.$$

The following equation restates Equation 3.4, based on Lemma 8.

$$L^p(i,j) = \max_{q \in P_{i,j}} \{L(i,q-1) + L(q,j)\}. \quad (4.1)$$

We next show a bottom-up algorithm that computes L according to Equations 3.1, 3.2, and 4.1. The presented algorithm is similar to Alg. 1, where a forward dynamic programming technique is applied in order to efficiently compute $L^p(i,j)$ (forward dynamic programming was also applied by Jansson et al. [18] to a related problem).

The new algorithm also scans and computes the entries of M in decreasing row index and increasing column index. It maintains the following invariant: upon reaching entry $M[i,j]$, the entry contains the value $L^p(i,j)$. Before computing row i in M, the entries $M[i,i-1]$ and $M[i,i]$ are initialized with zeros, and all entries $M[i,j]$ for $i < j \leq n$ are initialized with the corresponding values $M[i+1,j]$. This initialization is equivalent to examining the branch point $q = i+1$ in the computation of $L^p(i,j)$ according to equation 4.1 for all $j > i$ (the branching at $q = i+1$ is handled separately from other branch points in $P_{i,j}$ since it does not follow the step sequence-prefix-OCT-suffix rule as the rest of the group). Note that in this stage the invariant is sustained for the first entry in the row which is traversed by the algorithm - $M[i,i+1]$, since $P_{i,i+1} = \{i+1\}$.

Based on the invariant, upon reaching $M[i,j]$, the entry contains the value $L^p(i,j)$, and the value $L(i,j)$ can be computed by resolving the maximum between the current entry value and the value of $L^c(i,j)$, which is obtained from Equation 3.2. If $L^c(i,j) > L^p(i,j)$, $S_{i,j}$ is classified as an OCT. Then, if $M[i,j] > M[i+1,j]$, $S_{i,j}$ is classified as a step sequence, and the branch point $q = j+1$ is considered and forward-reflected to the computation of $L^p(i,j')$, for all $j' > j$ such that $S_{j+1,j'}$ is an OCT, by updating the value of $M[i,j']$ to be the maximum among its current value and that of $M[i,j] + M[j+1,j']$, thus accumulating the maximum according to Equation 4.1, and guaranteeing the maintenance of the invariant.

Alg. 5 in the online supporting material implements the forward dynamic programming approach described above, combined with the space-efficient approach described in Section 3.2. An illustration of its run is given in Fig. 4. The speedup obtained by this algorithm is due to the fact that branch points are examined by Equation 4.1 only if both the sequence prefix before the branch point is a step-sequence and its suffix, as from the branch point on, is an OCT. Note that, for each one of the Z OCT subsequences $S_{q,j}$ which are examined as suffices by Equation 4.1, Observation 2 shows that there are at most L sequences $S_{i,q-1}$ which may be corresponding step-sequence prefixes, and thus the total run-time contribution due to computation of values of the form $L^p(i,j)$ is $O(LZ)$. Since the table M has $O(n^2)$ entries, where for each entry $O(1)$ operations are performed in addition to the operations involved in the computations of $L^p(i,j)$, the total running time is $O(n^2 + LZ)$. The space complexity remains $O(Z)$, as the space complexity of Alg. 1.

Lemma 9. *Given an RNA sequence S of length n, there is an algorithm which computes $L(1,n)$ in $O(n^2 + LZ)$ time and $O(Z)$ space.*

4.2 An $O(LZ)$ Algorithm

In this section we further reduce the running time of the folding algorithm from $O(n^2 + LZ)$ to $O(LZ)$. We do so by applying a *step encoding* [21] to M, representing each of its rows by its $O(L)$ steps (see Fig. 5). Hence, in what follows we give corresponding step-encoding formulations, where a typical instance is composed of a suffix $S_{i,n}$ of S and a folding cardinality x, which will be denoted by the pair $(S_{i,n}, x)$. The goal is to compute the minimum index $i-1 \leq j \leq n$ such that there is a folding of $S_{i,j}$ whose cardinality is x. The next definition gives the step-encoding equivalents of the entities $L(i,j)$, $L^p(i,j)$, and $L^c(i,j)$.

Definition 6. *For $1 \leq i \leq n$, $1 \leq x$, and $\alpha \in \{\epsilon, p, c\}$ (where ϵ denotes the empty word), define $\beta^\alpha(i,x)$ to be the minimum index j such that $L^\alpha(i,j) \geq x$, or ∞ if there is no such j.*

Note the relation between the step-encoding formulation and the standard formulation, where $L(i,j)$ is the maximum x such that $\beta(i,x) \leq j$. Say that a sub-instance $(S_{i,n}, x)$ is a *β-OCT* if $\beta(i,x) = \beta^c(i,x) < \beta^p(i,x)$. The set $Y_{i,x}$ is the step-encoding equivalent of $P_{i,j}$:

$$Y_{i,x} = \{i+1\} \cup \left\{ i+1 < q \leq \beta(i+1, x-1) : \begin{array}{c} S_{i,q-1} \text{ is a step sequence, and} \\ (S_{q,n}, x - L(i,q-1)) \text{ is a } \beta\text{-OCT} \end{array} \right\}.$$

The following auxiliary function will be used in the computation of $\beta^c(i,x)$.

Definition 7. *For $\sigma \in \{A, C, G, U\}$ and $1 \le r \le n$, define $next(r, \sigma)$ to be the minimum index $r' > r$ such that $s_{r'} = \sigma$, or ∞ if there is no such index r'.*

We now convert Equations 3.1, 3.2 and 4.1 to their equivalent forms in the step encoding. For all $1 \le i \le n$ and $1 \le x$:

$$\beta(i,x) = \min\{\beta^c(i,x), \beta^p(i,x)\}. \tag{4.2}$$

$$\beta^c(i,x) = next(\beta(i+1, x-1), \overline{s_i}). \tag{4.3}$$

$$\beta^p(i,x) = \min\left\{\min_{q \in Y_{i,x}} \{\beta(q, x - L(i, q-1))\}, \beta^c(i,x) + 1\right\}. \tag{4.4}$$

Formal proofs of the correctness of Equations 4.2 to 4.4, as well as the pseudocode of an algorithm that implements them, are included in the online supporting material. This algorithm, denoted Alg. 6, adopts a forward dynamic programming approach, similarly to that of Alg. 5. This allows for efficient computation of Equation 4.4, where the number of sub-instances, as well as the dimensions of the data structure that stores solutions for these sub-instances, is $O(Ln)$ (instead of $O(n^2)$).

Lemma 10. *Given an RNA sequence S of length n, there is an algorithm that computes $L(1,n)$ in $O(LZ)$ time and $O(Z)$ space.*

Acknowledgments. The work of Shay Zakov and Michal Ziv-Ukelson was partially supported by the Frankel Center for Computer Science at Ben Gurion University of the Negev. Rolf Backofen received funding from the German Research Foundation (DFG grant BA 2168/2-1 SPP 1258), and from the German Federal Ministry of Education and Research (BMBF grant 0313921 FRISYS).

References

1. Consortium, A.F.B., Backofen, R., Bernhart, S.H., Flamm, C., Fried, C., Fritzsch, G., Hackermuller, J., Hertel, J., Hofacker, I.L., Missal, K., Mosig, A., Prohaska, S.J., Rose, D., Stadler, P.F., Tanzer, A., Washietl, S., Will, S.: RNAs everywhere: genome-wide annotation of structured RNAs. Journal of Experimental Zoology Part B: Molecular and Developmental Evolution 308(1), 1–25 (2007)
2. Zuker, M.: Mfold web server for nucleic acid folding and hybridization prediction. Nucleic Acids Research (13), 3406–3415 (2003)
3. Hofacker, I.L.: Vienna RNA secondary structure server. Nucleic Acids Research (13), 3429–3431 (2003)
4. Zuker, M.: Computer prediction of RNA structure. Methods Enzymol. 180, 262–288 (1989)
5. Tinoco, I., Borer, P., Dengler, B., Levine, M., Uhlenbeck, O., Crothers, D., Gralla, J.: Improved estimation of secondary structure in ribonucleic acids. Nature New Biology 246, 40–41 (1973)

6. Waterman, M., Smith, T.: RNA secondary structure: a complete mathematical analysis. Mathematical Biosciences 42, 257–266 (1978)
7. Nussinov, R., Jacobson, A.B.: Fast algorithm for predicting the secondary structure of single-stranded RNA. PNAS 77(11), 6309–6313 (1980)
8. Zuker, M., Stiegler, P.: Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. Nucleic Acids Research 9(1), 133–148 (1981)
9. Akutsu, T.: Approximation and exact algorithms for RNA secondary structure prediction and recognition of stochastic context-free languages. Journal of Combinatorial Optimization 3, 321–336 (1999)
10. Wexler, Y., Zilberstein, C., Ziv-Ukelson, M.: A study of accessible motifs and RNA folding complexity. Journal of Computational Biology 14(6), 856–872 (2007)
11. Chan, T.M.: More algorithms for all-pairs shortest paths in weighted graphs. In: Proc. 39th Symposium on the Theory of Computing (STOC), pp. 590–598 (2007)
12. Sankoff, D.: Simultaneous solution of the RNA folding, alignment and protosequence problems. SIAM Journal on Applied Mathematics 45(5), 810–825 (1985)
13. Mathews, D.H., Turner, D.H.: Dynalign: an algorithm for finding the secondary structure common to two RNA sequences. Journal of Molecular Biology 317(2), 191–203 (2002)
14. Havgaard, J., Lyngso, R., Stormo, G., Gorodkin, J.: Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%. Bioinformatics 21(9), 1815–1824 (2005)
15. Ziv-Ukelson, M., Gat-Viks, I., Wexler, Y., Shamir, R.: A faster algorithm for RNA co-folding, pp. 174–185 (2008)
16. Will, S., Reiche, K., Hofacker, I.L., Stadler, P.F., Backofen, R.: Inferring non-coding RNA families and classes by means of genome-scale structure-based clustering. PLOS Computational Biology 3(4), e65 (2007)
17. Gardner, P.P., Giegerich, R.: A comprehensive comparison of comparative RNA structure prediction approaches. BMC Bioinformatics 5, 140 (2004)
18. Jansson, J., Ng, S.K., Sung, W.K., Willy, H.: A faster and more space-efficient algorithm for inferring arc-annotations of RNA sequences through alignment. Algorithmica 46(2), 223–245 (2006)
19. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological sequence analysis: Probabilistic models of proteins and nucleic acids. Cambridge University Press, Cambridge (1998)
20. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Communications of the ACM 18(6), 341–343 (1975)
21. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. JACM 24, 664–675 (1977)

Multiple Alignment of Biological Networks: A Flexible Approach

Yves-Pol Deniélou[1], Frédéric Boyer[2], Alain Viari[1], and Marie-France Sagot[1]

[1] INRIA Grenoble-Rhône-Alpes, projet BAMBOO, 655 avenue de l'Europe, 38334 Montbonnot Cedex, France
{yves-pol.denielou,marie-france.sagot,alain.viari}@inrialpes.fr
[2] CEA, iRTSV, Laboratoire Biologie, Informatique et Mathématiques, F-38054 Grenoble, France
frederic.boyer@cea.fr

Abstract. Recent experimental progress is once again producing a huge quantity of data in various areas of biology, in particular on protein interactions. In order to extract meaningful information from this data, researchers typically use a graph representation to which they apply network alignment tools. Because of the combinatorial difficulty of the network alignment problem, most of the algorithms developed so far are heuristics, and the exact ones are of no use in practice on large numbers of networks. In this paper, we propose a unified scheme on the question of network alignment and we present a new algorithm, C3Part-M, based on the work by Boyer *et al.* [2], that is much more efficient than the original one in the case of multiple networks. We compare it as concerns protein-protein interaction networks to a recently proposed alignment tool, NetworkBLAST-M [10], and show that we recover similar results, while using a different but exact approach.

Keywords: multiple graph alignment, biological network comparison, protein-protein interactions.

1 Introduction

The recent advances in high-throughput experiments have fueled the research for an automated characterisation of biological networks such as protein-protein interactions (PPI) networks [21], metabolic pathways [13], or gene regulation networks [1]. A central question to the analysis of this kind of data is to align the networks in order to extract conserved subnetworks across multiple species or data sources. This information is of primary importance to study both the evolution and function of the proteins.

The network alignment question shares some similarities with the classical case of sequence alignment. Indeed, network alignment approaches can be local [2, 7, 10, 11, 12, 16] or global [6, 17, 18], pairwise [11, 12, 17, 20] or multiple [6, 7, 10, 16, 18]. In addition, just like with sequence symbols, one must also address the question of defining a similarity between the vertices of different networks.

However, a specific question that arises only in network alignments is to define which topological similarity to enforce on the aligned subnetworks.

A key concept, used in most of the studies [2,11,12,15,16], is that of a merged representation of the networks, which is called network alignment graph [16] or correspondence multigraph [2] in the literature. Given n networks $G_1, G_2, \dots G_n$, with $G_i = (V_i, E_i)$, and a correspondence relation S between the vertices of $\bigcup_i V_i$, the idea is to select n-tuples (alternatively called n-spines [10]) of vertices, one from each V_i, and to connect them with edges taken from the original E_i.

As we shall see later, an explicit construction of this representation is not always needed (nor desirable) but it turns out to be very useful to understand the differences between the algorithms or the biological questions to be solved. This can be summarised in three main questions:

– how to select the n-tuples (i.e. how to aggregate the vertices from the different sets V_i using S);
– which edges (from the different sets E_i) to keep;
– which topological condition(s) should be satisfied by the solution subgraphs.

Of course, the algorithmic difficulty of a network alignment greatly depends upon the answers to these questions and in most cases one has to resort to heuristics. In addition, several authors introduced scoring models at different levels of such heuristics. This will not be discussed here and we instead treat the question from

Table 1. Different methods for PPI network alignment and the corresponding assumptions. The *n-tuples* column gives the scheme used to gather homologuous proteins according to similarity edges. CC stands for connected components. The *edges* column details which edges from the PPI networks are taken into account. "conserved" means that only edges that appear in every single network are conserved. The *topology* column tells which kind of subgraphs the method is recovering, for instance the NetworkBLAST method recovers clusters and paths of the network alignment graph, whereas NetwokBLAST-M maximises a sum of scores, one for each network, ensuring the subgraphs are dense on each network.

Method	n-tuples construction	edges construction	topology	reference
PathBLAST	pairs (n=2)	conserved	paths	[11]
NetworkBLAST	CC ($n \leq 3$)	conserved	clusters / paths	[16]
NetworkBLAST-M	paths	all edges	dense clusters on each network	[10]
GRAEMLIN 1.0	non-overlap- -ping CC	conserved	user-defined	[7]
CAPPI	non-overlap- -ping CC	all edges > threshold	CC	[5]
HopeMap	pairs	conserved	CC	[20]
MaWISH	pairs	conserved	max-weight subgraph	[12]
PHUNKEE	pairs	all edges	max shared-edges ratio	[3]
C3PART	cliques or stars	all edges	common CC	[2]

a purely combinatorial point of view. We also focus on the more general question of a multiple alignment.

Table 1 summarises some important cases found in the literature.

For example, in the study by Sharan *et al.* [16] , the n-tuples are connected components for the correspondence relation S, the edges are those from E_i conserved in all the networks, and the enforced topologies are "dense" clusters, whereas in Boyer *et al.* [2] the n-tuples are cliques of S, all the edges in all the E_i's are kept (therefore giving rise to a multigraph instead of a graph) and the topological condition is simply the connectivity.

It is important to note that when the correspondence relation S is not one-to-one, the size of the network alignment graph may grow exponentially with both the number of vertices (n-tuples) and of edges, making it difficult to handle more than three networks. A general question thus appears: how to avoid the explicit construction of the network alignment graph ?

Several methods have been proposed to address this difficulty, among them the Graemlin algorithm [7] which avoids this construction by using a progressive alignment approach, and NetworkBLAST-M [10] which builds a set of n-tuple seeds with maximum score and then extends them greedily.

In this paper, we propose a non-heuristic approach to avoid the explicit construction of the network alignment graph and yet recover a pertinent and well-defined alignment. This work is based on the general framework proposed by Boyer *et al.* [2], which uses a correspondence multigraph formalism to extract connected components conserved in multiple networks. Our algorithm is an improvement of Boyer's original algorithm, C3Part, in order to avoid the explicit construction of the network alignment multigraph, and thus to be able to deal with a greater number of networks or a more degenerate correspondence relation across the networks. We illustrate this approach on the example of PPI networks, but all algorithmic concepts presented here can be applied on other kind of data as well.

The paper is organised as follows: Sections 2.1 to 2.3 define the layered data graph and the network alignment multigraph. Section 2.4 gives a brief overview of the C3Part algorithm and its limits. Section 2.5 presents our improvement of this algorithm, that builds on the fly the parts of the network alignment multigraph that are really needed for the alignment. And finally, Section 3 provides some results in the case of PPI networks, allowing us to compare the efficiency of our method with other similar approaches.

2 Methods

2.1 Layered Data Graph

The layered data graph (also called layered alignment graph in [10]) provides the simplest representation of the data at hand.

Definition 1. (adapted from [10]) *Given a set of n networks $G_i = (V_i, E_i)$, $i \in [|1, n|]$ (hereafter called* primary networks*) and a correspondence relation*

S between the elements of distinct sets V_i, the layered data graph *is the graph $D = (V, E)$ with*

- $V = \bigcup_i V_i$
- $E = (\bigcup_i E_i) \cup \{(u, v) \in V_i \times V_{j \neq i} / u\ S\ v\}$

Observe that there are two kinds of edges in E: one corresponds to the original sets E_i (hereafter called *intra-layer edges*) and the other one connects vertices from different layers (hereafter called *inter-layer edges*) (see Figure 1).

2.2 Correspondence n-Way, and n-Tuples

As mentioned before, for $n > 2$ networks, one has to define formerly how the layered data graph vertices are aggregated to form n-tuples.

Definition 2. *An n*-way correspondence *between elements of $V_1, V_2, \ldots V_n$ is defined as a restriction $\mathcal{R}$ of the cartesian product, denoted by $\mathcal{R}(V_1 \times V_2 \times \ldots \times V_n)$.*

There are three main interesting practical cases of such an aggregation.

1. The clique aggregator :
   $(v_1, \ldots v_n) \in \mathcal{R}(V_1 \times V_2 \times \ldots \times V_n) \Leftrightarrow \forall i, j \in [|1, n|],\ v_i\ S\ v_j$
   ie we require that all elements of the tuple are pairwise related.
2. The centered-star aggregator :
   $(v_1, \ldots v_n) \in \mathcal{R}(V_1 \times V_2 \times \ldots \times V_n) \Leftrightarrow \exists i \in [|1, n|] / \forall j \in [|1, n|],\ j \neq i,\ v_i\ S\ v_j$
   ie we require a "star" topology of the relations between elements of the tuple.
3. The connected component aggregator :
   $(v_1, \ldots v_n) \in \mathcal{R}(V_1 \times V_2 \times \ldots \times V_n) \Leftrightarrow (v_1, \ldots v_n)$ is a connected component of S.
   A particular and important case of connected component aggregator is the path aggregator [16] which requires the vertices to be connected by a path. Among the path aggregators, the tree-guided path aggregator [10] requires the path to be compatible with a given phylogenetic tree.

2.3 Network Alignment MultiGraph

A network alignment multigraph is a graph data structure that summarises both the n-way correspondence and the connectivity in the primary networks.

Definition 3. *The* network alignment multigraph *is the multigraph $M = (V, E'_1, \ldots, E'_n)$ such that:*

- $V = \mathcal{R}(V_1 \times V_2 \times \ldots \times V_n)$
- $\forall u = (u_1, u_2, \ldots, u_n) \in V$ *and* $v = (v_1, v_2, \ldots, v_n) \in V$,
  $(u, v) \in E'_i \Leftrightarrow (u_i, v_i) \in E_i \vee (v_i = u_i)$

In other words, the vertices of the multigraph are n-tuples and there is an edge between two vertices if the i^{th} elements (vertices) of the tuples are connected in the primary network G_i, for all $i \in [|1, n|]$. In the following, we refer to such an edge as an *edge of colour i*.

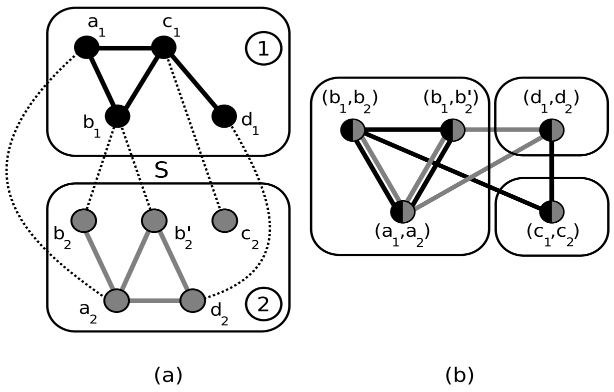


Fig. 1. Example of a layered data graph (a) and the corresponding network alignment multigraph (b) with 3 connectons. The correspondence relation S is represented by dotted lines. The restriction R is defined by the couples of vertices linked by S edges. Observe that the whole multigraph is not a connecton since (c_1, c_2) is not reachable by both colours. Also observe that connectons do not correspond to the intersection of the connected components.

2.4 Defining Connectons in the Network Alignment Multigraph

With this definition of the network alignment multigraph at hand, there are several possible definitions of the property we want to look for (see Table 1), and therefore of the conserved subgraphs we want to recover. Here we choose the definition formalised by Boyer *et al.* [2] which presents several advantages that will be discussed later.

Definition 4. *A* connecton *is a maximal set of vertices in the network alignment multigraph which are connected components for each* E'_i, $i \in [|1, n|]$.

An example of connecton is given in Figure 1. An important property is that the sets of connectons form a partition of the vertices of the multigraph. This property, which is usually not satisfied with other definitions (e.g. dense clusters [10]), allows the use of exact algorithms instead of heuristics to enumerate them. Moreover, the connecton condition is weaker than would be a condition on dense clusters, and, provided that connectons are not too numerous, they can be further post-processed to satisfy a stronger constraint. Also observe that to speed up the algorithm, we can restrict ourselves in the definition of connectons to connected components above a fixed size.

The algorithm used by Boyer *et al.* [2] to find connectons in the network alignment multigraph computes iteratively the partition, starting with a single class containing all the vertices and refining the partition at each step. The refinement procedure used in C3Part works as follows:

1. start with one single class containing all the vertices in the network alignment multigraph;

2. compute $\bigcap_i (CC_i)$, the intersection of all connected components on all colours, this gives rise to a new partition where each class is a potential connecton;
3. iterate -2- on each class until the partition does not change.

The worst-case time complexity is $\mathcal{O}((N+M) \times N)$ where N is the number of vertices and M the number of edges in the multigraph. This comes from the fact that each iteration requires $\mathcal{O}(N+M)$ operations to compute the connected components and that, in the worst-case one will have to perform $\mathcal{O}(N)$ iterations (this corresponds to the case where there are $\mathcal{O}(N)$ classes at the end and each class has been extracted at each iteration).

This complexity can actually be improved. In 2003, Gai *et al.* [8] proposed a more sophisticated algorithm combining the dynamic maintenance of a spanning forest together with an Hopcroft-like partitioning approach. This algorithm achieves an $\mathcal{O}((N + M \times logN) \times logN)$ complexity. Furthermore, in the case of interval graphs, the complexity reduces to $\mathcal{O}((N+M) \times logN)$ [9]. This latter particular case is important to handle questions related to chromosomal syntenies [14] where the primary networks are interval graphs.

In practice, neither C3Part nor those algorithms are usable on large numbers of networks, since their prerequisite is the construction of the network alignment multigraph, whose size is exponential with the number of networks when the correspondence relation S is not a one-to-one correspondence.

The objective of this work is to apply the same approach but to avoid the initial construction of the multigraph. The general idea is to build the multigraph on the fly, starting with a connected component on the first primary graph, expanding it on the second one, then splitting it on those two colours, and expanding recursively the results on the third graph, etc.

2.5 On the Fly Construction of the Network Alignment Multigraph

Although the original C3Part algorithm adopted a Breadth-First-Search presentation, is is easy to transform it into a Depth-First-Search by observing that the split of a class can actually be conducted on each colour in turn. This is made possible by the fact that if two vertices are disconnected by two or more colours, only one is actually needed to split the class. With a DFS approach, all classes are therefore refined independently. The next observation is that, since colours are now considered in turn, when we split a class on a colour i, we may not need the information about the $(n-i)$ remaining colours that will be used later on. This therefore makes it possible to add the new colours only when necessary.

The new algorithm, C3Part-M, will use two different operations.

– $SPLIT_{1-i}$ that splits a class on colours 1 to i;
– $EXPAND_{i+1}$ that adds the $(i+1)^{th}$ colour to the current network alignment multigraph.

A *stable class* is a class C such that $SPLIT_{1-n}(C) = C$ and is therefore a connecton.

The $SPLIT_{1-i}$ operation computes the connected components on each colour in turn. If, for a colour, the class is split, then it returns the split parts.

When a class C is such that $SPLIT_{1-i}(C) = C$ then it is stable for colours 1 to i and needs expansion to the $(i+1)^{th}$ colour.

The pseudocode of the algorithm is given hereafter.

Algorithm: C3Part-M.
Input: *Set_of_vertices class /* class to refine: initialised with all vertices from the first network */*
Colour_Index i / current colour index: initialised to 1 */*
Variables: *Partition_of_vertices split*

```
(1)   begin
(2)       split ← $SPLIT_{1\to i}$(class);
(3)       if ($|split| \neq 1$) then
(4)           for $s \in split$ do
(5)               C3Part-M($s, i$)
(6)           end for
(7)       else if ($i \neq maxcolour$) then
(8)           newclass ← $EXPAND$(class, $i+1$);
(9)           C3Part-M(newclass, $i+1$);
(10)      else
(11)          /* class is stable */
(12)          PRINT(class)
(13)      end if
(14)  end
```

The expansion is done by the $EXPAND$ operation that works in two steps. Starting from the current class C (i.e. a reduced multigraph defined on colours 1 to i), we:

1. expand each vertex $v = (v_1, v_2, \ldots v_i)$ of size i to new vertices of size $i+1$ ($EXPAND\_VERTEX$);
2. add edges between these new vertices ($EXPAND\_EDGES$).

$EXPAND\_VERTEX$ works as follows. For each vertex $v = (v_1, v_2, \ldots v_i)$ in the current class C, we first collect all vertices v_{i+1} of the $(i+1)^{th}$ primary graph such that $(v_1, v_2, \ldots, v_i, v_{i+1}) \in \mathcal{R}(V_1 \times V_2 \times \ldots \times V_i \times V_{i+1})$. These vertices are called the *terminals* of v. Then for each of these terminals, a new vertex $v' = (v_1, v_2, \ldots v_{i+1})$ is created. This new vertex v' is hereafter called a *son* of v and v is called its *father*.

Observe that how all terminals of a given vertex v are collected depends upon the chosen aggregator. For the clique aggregator (resp. star aggregator), this is a simple task since, by construction $v = (v_1, v_2, \ldots v_i)$ is already a clique (resp. star) of S, then one has just to collect the terminals v_{i+1} that are S-connected to each v_i (resp. to the center star). For the connected component aggregator, the task is more demanding since vertices v_{i+1} may form a connected component with $v = (v_1, v_2, \dots v_i)$ only once all colours have been considered (i.e. colours greater than $i + 1$). The complexity therefore strongly depends upon the degeneracy of S.

Once all the new vertices have been added, one should add the new edges as well ($EXPAND\_EDGES$). There are actually three kinds of such edges, connecting two new vertices $(u_1, u_2, \dots u_i, u_{i+1})$ and $(v_1, v_2, \dots v_i, v_{i+1})$. The first kind is simply the edges already existing in the class C, i.e. connecting fathers, that should be restored with their previous colours. The second kind is edges connecting sons of the same father. Those edges have all colours from 1 to i since the father is the same ($(u_1, u_2, \dots u_i) = (v_1, v_2, \dots v_i)$). Finally, one has to add new edges of colour $i + 1$ corresponding to the terminals u_{i+1} and v_{i+1}, i.e. connecting all new vertices such that $(u_{i+1}, v_{i+1}) \in E_i$ (or $u_{i+1} = v_{i+1}$).

The worst case complexity of this new algorithm (C3Part-M) is the same as the one of C3Part. It corresponds to the case where there is no split on the first $n-1$ colours, thus giving rise to the full alignment multigraph that is eventually splitted on the last n^{th} colour. For all practical cases we have tested so far, this never happens and the new version turns out to be several order of magnitude faster than the previous one (see [4] for an example on syntenies). Moreover, one can observe that since the result does not depend upon the order in which the colours are chosen, it is possible to avoid this worst case either by choosing an order more likely to be favourable at the beginning or, better, by reordering the colours dynamically during the recursive split. Several heuristic optimisations have been tested but will not be further described in this extended abstract.

3 Results and Discussion

In order to evaluate the new algorithm, we selected the benchmark set of 10 microbial PPI networks used in previous similar studies [7,10]. Those networks are not experimental but were actually produced by an inference algorithm (called SRINI) described in [19]. Briefly, SRINI generates a probabilistic interaction network by integrating several sources of information such as co-expression, co-evolution or chromosomal co-location. Unlike experimental PPIs, the output of SRINI is a complete graph where each pair of vertices is labelled by an interaction probability. In order to be used in any algorithm, these networks should therefore be thresholded.

As for the correspondence relation S, we used the same data as in [7, 10], *i.e.* a sequence similarity relation determined by BlastP. We selected a Blast threshold of 10^{-10} and limited the number of hits per protein to 5. The restriction R was defined by cliques of S (clique aggregator *i.e.* all proteins are similar one to the other).

The sizes of the obtained layered data graphs are given in Table 2 for different numbers of selected species (3, 5, 7 and 10; the selected species are the same as in [10]).

The first observation we can make is that C3Part-M was able to cope with these large networks whereas the explicit computation of the alignment multigraph is intractable by C3Part (and NetworkBLAST) for more than 3 species.

Table 2. Comparison of C3Part-M and NetworkBlast-M on 10 microbial species. NetworkBlast-M running times are given both for the relaxed mode (1) and for the tree-guided-path mode (2) (comparisons of other results are given for the relaxed mode). *#Spine* corresponds to the number of different n-tuples found by the various algorithms. *#Prot* corresponds to the number of different proteins involved in these n-tuples.

n	PPI Thresh.	# Prot	#Edges Simil.	#Edges PPI	Time (s) NBM (1)	Time (s) NBM (2)	Time (s) C3P-M	#Spine NBM	#Spine C3P-M	#Spine Common	#Prot NBM	#Prot C3P-M	#Prot Common
	0.7	3751	1526	7450	1	1	4	91	194	76(84%)	235	339	207(88%)
3	0.5	5322	2739	18511	2	2	6	169	457	128(76%)	413	628	354(86%)
	0.3	7826	4606	66484	9	5	12	291	972	195(67%)	679	1128	557(82%)
	0.7	5910	4444	13061	12	2	6	82	261	58(71%)	337	370	266(79%)
5	0.5	8169	7422	38645	34	3	9	168	566	73(43%)	599	656	390(65%)
	0.3	11404	11805	134437	175	12	19	201	1432	104(52%)	794	1231	562(71%)
	0.7	8416	8721	18707	166	2	7	31	190	19(61%)	206	252	156(76%)
7	0.5	12354	16897	60517	478	5	12	119	531	54(45%)	602	640	394(65%)
	0.3	16579	26452	211368	2832	17	38	193	2500	77(40%)	1002	1284	601(60%)
	0.7	15411	21995	47340	9038	3	60	24	853	18(75%)	240	292	210(88%)
10	0.5	23370	50758	173881	39644	10	74	111	2062	45(41%)	798	729	472(59%)
	0.3	30534	74126	616307	N/A	33	1143	N/A	13250	N/A	N/A	1613	N/A

We then compared our results with those obtained using NetworkBlast-M [10], that is the multiple version of the NetworkBlast algorithm [16]. As mentioned before, NetworkBlast-M is a heuristic algorithm that relies on different assumptions concerning the construction of the vertices and edges of the alignement multigraph and on the required topology of the subnetworks (see Table 2). We ran NetworkBlast-M both in “relaxed” mode (where n-spines are paths) and in the tree-guided-path mode that was shown to be much quicker [10]. The running time of C3Part-M compares well with the tree-guided-path mode and could in most case retrieve the connectons within a few minutes.

A more interesting comparison with NetworkBlast-M is in terms of the results found. Since the assumptions are not the same, we cannot compare directly the complexes found. We therefore decided to compare the results in terms of the different constructed n-tuples and different proteins involved. Interestingly, despite their different asumptions, it turns out that the two algorithms recover essentially the same objects, both in terms of spines and proteins. For 3 species, this was somehow expected since the clique and the path conditions are quite similar, and up to 84% of the spines and 88% of the proteins reported by NetworkBlast-M were also found by C3Part-M. Of course, this overlap decreases with the number of species but remains remarkably high even for 10 species. An explanation for this comes from the observation done by Kalaev *et al.* [10] that most of the recovered spines actually are cliques.

4 Conclusion

We addressed the problem of multiple network alignment with an exact and generic approach based on the work by Boyer *et al.* [2]. By avoiding the explicit construction of the network alignment multigraph, we were able to deal with a large number of networks.

A comparison of our algorithm to NetworkBlast-M [10] led to very similar results, with reasonable execution times. Furthermore, our definition of connectons as subgraphs corresponding to connected components on each network allows us to cleanly separate the heuristic choices of biologically-relevant scoring functions from the alignment procedure itself. Our approach can consequently be used as a pre-filter to other more specialised tools.

Many important challenges remain. For instance the introduction of weaker aggregators would help to recover conserved subgraphs in the case of missing proteins. One idea would be to introduce a species quorum *i.e.* to look for spines not necessarily containing all the species.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments and suggestions.

References

1. Babu, M.M., Luscombe, N.M., Aravind, L., Gerstein, M., Teichmann, S.A.: Structure and evolution of transcriptional regulatory networks. Curr. Opin. Struct. Biol. 14(3), 283–291 (2004)
2. Boyer, F., Morgat, A., Labarre, L., Pothier, J., Viari, A.: Syntons, metabolons and interactons: an exact graph-theoretical approach for exploring neighbourhood between genomic and functional data. Bioinformatics 21(23), 4209–4215 (2005)
3. Cootes, A.P., Muggleton, S.H., Sternberg, M.J.: The identification of similarities between biological networks: Application to the metabolome and interactome. Journal of Molecular Biology 369(4), 1126–1139 (2007)
4. Denielou, Y.-P., Boyer, F., Sagot, M.-F., Viari, A.: Recovering isofunctional genes: a synteny-based approach. In: JOBIM, pp. 11–16 (2008)
5. Dutkowsky, J., Tiuryn, J.: Identification of functional modules from conserved ancestral protein protein interactions. Bioinformatics 23(13) (2007)
6. Flannick, J.A., Novak, A.F., Do, C.B., Srinivasan, B.S., Batzoglou, S.: Automatic parameter learning for multiple network alignment. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 214–231. Springer, Heidelberg (2008)
7. Flannick, J., Novak, A., Srinivasan, B.S., McAdams, H.H., Batzoglou, S.: Græmlin: general and robust alignment of multiple large interaction networks. Genome Res. 16(9), 1169–1181 (2006)
8. Gai, A.T., Habib, M., Paul, C., Raffinot, M.: Identifying Common Connected Components of Graphs. Technical report, LIRMM (2003)
9. Habib, M., Paul, C., Raffinot, M.: Maximal Common Connected Sets of Interval Graphs. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 347–358. Springer, Heidelberg (2004)
10. Kalaev, M., Bafna, V., Sharan, R.: Fast and accurate alignment of multiple protein networks. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 246–256. Springer, Heidelberg (2008)

11. Kelley, B.P., Sharan, R., Karp, R.M., Sittler, T., Root, D.E., Stockwell, B.R., Ideker, T.: Conserved pathways within bacteria and yeast as revealed by global protein network alignment. Proc. Natl. Acad. Sci. USA 100(20), 11394–11399 (2003)
12. Koyutürk, M., Kim, Y., Topkara, U., Subramaniam, S., Szpankowski, W., Grama, A.: Pairwise alignment of protein interaction networks. J. Comput. Biol. 13(2), 182–199 (2006)
13. Papin, J.A., Price, N.D., Wiback, S.J., Fell, D.A., Palsson, B.O.: Metabolic pathways in the post-genome era. Trends Biochem. Sci. 28(5), 250–258 (2003)
14. Pasek, S., Bergeron, A., Risler, J.L., Louis, A., Ollivier, E., Raffinot, M.: Identification of genomic features using microsyntenies of domains: Domain teams. Genome Res (2005)
15. Sharan, R., Ideker, T., Kelley, B., Shamir, R., Karp, R.M.: Identification of protein complexes by comparative analysis of yeast and bacterial protein interaction data. J. Comput. Biol. 12(6), 835–846 (2005)
16. Sharan, R., Suthram, S., Kelley, R.M., Kuhn, T., McCuine, S., Uetz, P., Sittler, T., Karp, R.M., Ideker, T.: From the cover: Conserved patterns of protein interaction in multiple species. Proc. Natl. Acad. Sci. USA 102(6), 1974–1979 (2005)
17. Singh, R., Xu, J., Berger, B.: Pairwise global alignment of protein interaction networks by matching neighborhood topology. In: Speed, T., Huang, H. (eds.) RECOMB 2007. LNCS (LNBI), vol. 4453, pp. 16–31. Springer, Heidelberg (2007)
18. Singh, R., Xu, J., Berger, B.: Global alignment of multiple protein interaction networks with application to functional orthology detection. Proceedings of the National Academy of Sciences 105(35), 12763–12768 (2008)
19. Srinivasan, B.S., Novak, A.F., Flannick, J.A., Batzoglou, S., McAdams, H.H.: Integrated protein interaction networks for 11 microbes. In: Apostolico, A., Guerra, C., Istrail, S., Pevzner, P.A., Waterman, M. (eds.) RECOMB 2006. LNCS (LNBI), vol. 3909, pp. 1–14. Springer, Heidelberg (2006)
20. Tian, W., Samatova, N.F.: Pairwise alignment of interaction networks by fast identification of maximal conserved patterns. In: PSB 2009, pp. 99–110 (2009)
21. Tucker, C.L., Gera, J.F., Uetz, P.: Towards an understanding of complex protein networks. Trends in Cell Biology 11(3), 102–106 (2001)

Graph Mining: Patterns, Generators and Tools

Christos Faloutsos

Carnegie Mellon University
christos@cs.cmu.edu

Abstract. How do graphs look like? How do they evolve over time? How can we generate realistic-looking graphs? We review some static and temporal 'laws', and we describe the "Kronecker" graph generator, which naturally matches all of the known properties of real graphs. Moreover, we present tools for discovering anomalies and patterns in two types of graphs, static and time-evolving. For the former, we present the 'CenterPiece' subgraphs (CePS), which expects q query nodes (eg., suspicious people) and finds the node that is best connected to all q of them (eg., the master mind of a criminal group). We also show how to compute CenterPiece subgraphs efficiently. For the time evolving graphs, we present tensor-based methods, and apply them on real data, like the DBLP author-paper dataset, where they are able to find natural research communities, and track their evolution.

Finally, we also briefly mention some results on influence and virus propagation on real graphs, as well as on the emerging map/reduce approach and its impact on large graph mining.

Level-k Phylogenetic Networks Are Constructable from a Dense Triplet Set in Polynomial Time

Thu-Hien To and Michel Habib

LIAFA, CNRS and University Paris Diderot - Paris 7
thu-hien.to@liafa.jussieu.fr, habib@liafa.jussieu.fr

Abstract. For a given dense triplet set $\mathcal{T}$, there exist two natural questions [7]: Does there exist any phylogenetic network consistent with $\mathcal{T}$? In case such networks exist, can we find an effective algorithm to construct one? For cases of networks of levels $k = 0$, 1 or 2, these questions were answered in [1,6,7,8,10] with effective polynomial algorithms. For higher levels k, partial answers were recently obtained in [11] with an $O(|\mathcal{T}|^{k+1})$ time algorithm for simple networks. In this paper, we give a complete answer to the general case, solving a problem proposed in [7]. The main idea of our proof is to use a special property of SN-sets in a level-k network. As a consequence, for any fixed k, we can also find a level-k network with the minimum number of reticulations, if one exists, in polynomial time.

1 Introduction

The goal of phylogenetics is to reconstruct plausible evolutionary histories from biological data of currently living species. Normally, the standard model to describe the derivation is a tree in which each leaf is labeled by a species, and in which each node having descendants represents the most recent common ancestor of its descendants. However in reality, if we count on the hybridizations, recombinations and lateral gene transfer events, the model will be a network in which we allow the fact that a species can have more than one parent. Such a node is called an hybrid node or a reticulation. To study general phylogenetic networks, a way to classify them using levels has been introduced in [3], based on the number of reticulations in its biconnected components. Under this classification, a phylogenetic tree is considered as a level-0 phylogenetic network. This view gives an approach to analyze networks by decomposing them into several sub-networks having the same level. Besides the level, the most basic description of a phylogenetic evolution is a triplet which gives us the information on the relation of 3 species: which 2 species are closer than the last. Therefore, a fundamental problem is to construct a phylogenetic network consistent with a set of triplets. If there is no constraint on the triplet set, the problem is NP-hard with networks of levels higher than 0 [6,10,12]. However if we impose the density on the triplet set, that is if we require that there is at least one triplet for each three species, then the species set has a better structure. Using this structure, [6,7,8] constructed a level-1 network, if one exists, in polynomial time, and [10] extended the result to level-2 networks. The following question was first asked in [7]: Does the problem remain polynomial for level-k network with any fixed k? We present here an affirmative answer to this question. The algorithmic paradigm that we use is closed to those used for the one of level-1 and level-2 networks. However, we prove a new property of level-k networks which allows us to bound the number of all possible solutions. As a consequence, for any fixed k, we can also find a level-k network with the minimum number of reticulations, if one exists, in polynomial time.

Related works. Aho, Sagiv, Szymanski, and Ullman [1] presented an $O(|\mathcal{T}|.n)$-time algorithm for determining whether a given set $\mathcal{T}$ of triplets on n leaves is consistent with some rooted, distinctly leaf-labeled tree, i.e. a level-0 network, and if so, returning such a tree. Later, improvements were given in [4,5]. But the problem has been proved to be NP-hard for all other levels [6,10,12]. Similarly the problem of finding a network consistent with the maximum number of triplets is also NP-hard for all levels [6,12]. The approximation problem which gives a factor on the number of triplets that we can construct a network consistent with, is also studied in [2] for level-0, level-1, and level-2 networks.

Concerning the particular case of dense triplet sets, there are following results. For level-1, [6,7] give an $O(|\mathcal{T}|)$-time algorithm to construct a consistent network, and [11] gives an $O(n^5)$-time algorithm to construct the consistent one with the minimum number of reticulations. For level-2, [10] gives an $O(|\mathcal{T}|^{\frac{8}{3}})$-time algorithm to construct a consistent network, and [11] presents an $O(n^9)$-time algorithm to construct the consistent one with the minimum number of reticulations. For level-k networks with any fixed k, there is only a result constructing all simple consistent networks with an $O(|\mathcal{T}|^{k+1})$-time algorithm [11]. The problem of finding a network consistent with the maximum number of triplets is also NP-hard for all levels in this case [12]. However, it is still unknown if we can find a consistent network with the minimum level in polynomial time.

There are also studies on the version of extremely dense triplet sets, that is when $\mathcal{T}$ is considered to contain all triplets of a network. In this case, an $O(|\mathcal{T}|^{k+1})$ time algorithm was given in [11] for level-k networks. But even in this case, the problem of minimizing the level of consistent networks is still open.

2 Notations

Let $\mathcal{L}$ be a set of n species. A *phylogenetic network* N on $\mathcal{L}$ is a connected, directed, acyclic graph which has:

- a unique vertex of indegree 0 and outdegree 2 (root).
- vertices of indegree 1 and outdegree 2 (speciation vertices).
- vertices of indegree 2 and outdegree 1 (hybrid vertices or reticulation vertices).

- n vertices labeled distinctly by $\mathcal{L}$ of indegree 1 and outdegree 0 (leaves). So $\mathcal{L}$ is also called the leaf set.

We denote $u \rightsquigarrow v$ if there is a path in N from u to v (u and v may be the same vertex).

A simple (without multiple edges) undirected graph $G = (X, E)$ is *biconnected* iff for $\forall x, y \in X$, there exist two vertex-disjoint paths in G from x to y. A *biconnected component* of a graph is a maximal biconnected subgraph, by convention a graph reduced to one vertex is considered biconnected. Therefore one can consider the decomposition of G into its biconnected components. Let $\mathcal{U}(N)$ be the underlying undirected graph of N, obtained by replacing each directed edge of N by an undirected edge. Let us denote by *cut-arc* an arc in N whose removal disconnects N. A cut-arc $a = (u, v)$ is *highest* if there is no cut-arc $a' = (u', v')$ such that $v' \rightsquigarrow u$. A highest cut-arc is always a cut-arc starting from a vertex of the biconnected component which contains the root.

A phylogenetic network is *simple* if it has only one non trivial biconnected component which is the one containing the root, and every its cut-arc, which is also a highest cut-arc, connects a vertex of this biconnected component to a leaf. A network N is called of *level-k* if each biconnected component of $\mathcal{U}(N)$ contains at most k hybrid vertices.

A *triplet* $x|yz$ is a rooted binary tree on the leaves x, y and z such that x and the parent of y and z are children of the root. A set $\mathcal{T}$ of triplets is *dense* if for any set $\{x, y, z\} \subseteq \mathcal{L}$, at least one triplet on these three leaves belongs to $\mathcal{T}$. A triplet $x|yz$ is *consistent* with a network N if N contains two vertices $u \neq v$ and pairwise internally vertex-disjoint paths $u \rightsquigarrow x$, $u \rightsquigarrow v$, $v \rightsquigarrow y$, and $v \rightsquigarrow z$.

Let $\mathcal{P}$ be a partition of the leaf set $\mathcal{L}$: $\mathcal{P} = \{P_1, \ldots, P_q\}$. We denote by $\mathcal{T}\nabla\mathcal{P}$ the induced triplet set $P_iP_j|P_k$ such that there exist $x \in P_i, y \in P_j, z \in P_k$ with $xy|z \in \mathcal{T}$ and i, j and k are distinct.

Let L be a subset of the leaf set $\mathcal{L}$. We denote the restriction of $\mathcal{T}$ to L by $\mathcal{T}|L = \{x|yz \in \mathcal{T}$ such that $x, y, z \in L\}$.

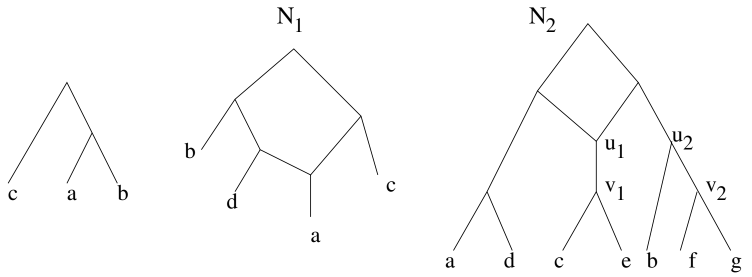


Fig. 1. The triplet $c|ab$ is consistent with N_1, but not with N_2. N_1 is a simple level-1 network, N_2 is also a level-1 network but not simple. In N_2, (u_1, v_1) is a highest cut-arc, (u_2, v_2) is also a cut-arc but not highest. Note that, as with all figures in this paper, all arcs are directed downwards, away from the root.

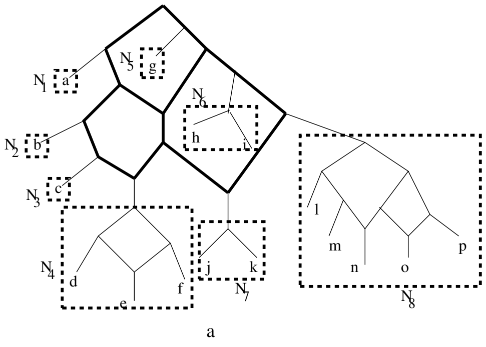
3 Construction of a Level-k Phylogenetic Network from a Dense Triplet Set

In this section we show that, for any fixed positive integer k, it is possible to construct in polynomial time a level-k phylogenetic network from a dense triplet set, if such a network exists. Let us start with some properties of level-k networks.

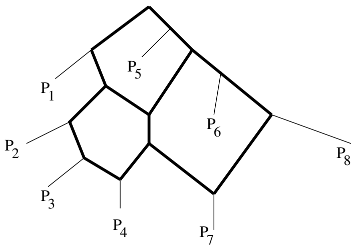
Let N be a level-k network. Then:

i) We can decompose N into a finite number of sub-networks as follows (see figure 2(a)): let C be the biconnected component which contains the root. Each highest cut-arc of N connects a vertex of C to a sub-network. Theses sub-networks, which are denoted by $N_1, \ldots, N_m$, are all level-k and pairwise vertex-disjoint.

ii) For any $j = 1, \ldots, m$, let P_j be the leaf set of N_j. So $\mathcal{P} = \{P_1, \ldots, P_m\}$ is a partition of $\mathcal{L}$. We call each part P_i of the partition a *leaf set below a highest cut-arc*. By replacing each N_i by a representing leaf, also called P_i, we obtain a simple level-k network N_s (see figure 2(b)). Assuming that for any j, N_j is consistent with $\mathcal{T}|P_j$ (the restriction of $\mathcal{T}$ to P_j). So, N is consistent with $\mathcal{T}$ if and only if N_s is consistent with $\mathcal{T}\nabla\mathcal{P}$.



(a) Decompose a network: the biconnected component C is in bold, each sub-network N_j is framed by a dotted bold rectangle, each highest cut-arc connects a vertex of C to a sub-network.



(b) Reduce to a simple network: each leaf P_j represents the leaf set of the sub-network N_j.

Fig. 2. Decompose a level-k network into several level-k sub-networks and reduce it to a simple level-k network

Using these properties, the following algorithm was introduced in [6,7,8], to construct a network. This algorithm enumerates all the possible decompositions: i.e. it considers how the leaf set can be partitioned below the highest cut-arcs and computes a simple network on each part of the partition (i.e. leaves correspond to parts of the partition). Then it recurses and for each part of the partition it computes a consistent sub-network using the same method. [11] propose a method to construct all simple level-k networks consistent with a dense triplet set $\mathcal{T}$ in $O(|\mathcal{T}|^{k+1})$ time. So it remains to know the number of possible partitions of the leaf set $\mathcal{L}$ below the highest cut-arcs. We will show in the remaining of this section that, for any fixed k, the number of the possible partitions can be bound by a polynomial in n.

We know that each part of the partition is a leaf set below a highest cut-arc. Hence, the question is answered by exploring the leaf sets below cut-arcs. Remark that if A is a leaf set below a cut-arc, then for any $z \in \mathcal{L} \backslash A$, $x, y \in A$, the only triplet on $\{x, y, z\}$ that can be consistent with the network is $z|xy$. Based on this property, we define a family of leaf sets, called **CA-sets**, for CutArc-sets, as follows.

Definition 1. *Let $A \subseteq \mathcal{L}$. We say that A is a CA-set if either it is a singleton or the whole $\mathcal{L}$, or if it satisfies the following property: For any $z \in \mathcal{L} \backslash A$, $x, y \in A$, the only triplet on $\{x, y, z\}$ in $\mathcal{T}$, if there is any, is $z|xy$.*

As remarked, the set of all leaves below a cut-arc is a CA-set, but the converse claim is not always true. Let us recall that [7] presented a variation of these CA-sets, namely the notion of SN-set. A *SN-set* is defined by a closure operation as follows. Let A be a subset of $\mathcal{L}$, the SN-set of A, denoted $SN(A)$, is the set recursively defined as $SN(A \cup \{z\})$ if there exists some $z \in \mathcal{L} \backslash A$ and $x, y \in A$ such that $x|yz \in \mathcal{T}$, and as A otherwise. The following lemma states the equivalence between these two definitions.

Lemma 1. *CA-sets = SN-sets.*

(i) For any $A \subseteq \mathcal{L}$, $SN(A)$ is a CA-set.

(ii) For any CA-set A, there exists B, a subset of $\mathcal{L}$, such that $SN(B) = A$.

Proof. All claims are obviously true for singletons. Let us now consider only the non trivial sets.

(i) For any non trivial $A \subseteq \mathcal{L}$, $\forall z \in \mathcal{L} \backslash SN(A)$, $\forall x, y \in SN(A)$, neither $x|yz$ nor $y|xz$ is in $\mathcal{T}$ because if one of them is, following the definition of SN-set, $SN(A)$ will be $SN(A \cup \{z\})$, and will contain z. So, the only triplet on $\{x, y, z\}$ in $\mathcal{T}$, if there is any, is $z|xy$. In other words, $SN(A)$ is a CA-set, according to the definition 1.

(ii) For any CA-set A, there can exist several B such that $SN(B) = A$. We take, for example, B equals to A. We have to show that $SN(A) = A$. Indeed, as A is a CA-set, there does not exist any $z \in \mathcal{L} \backslash A$ and $x, y \in A$ such that $x|yz \in \mathcal{T}$. It means that $SN(A)$ is exactly A, according to the recursive definition of SN-set. □

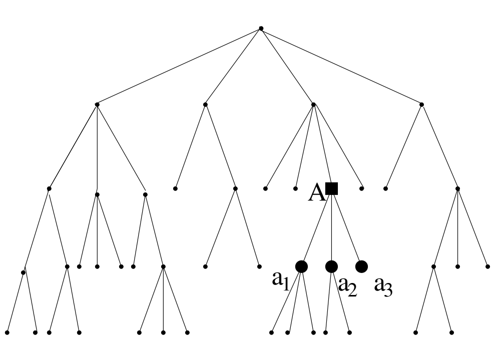
Therefore, the family of SN-sets is exactly the family of CA-sets and we will stick to the notation of SN-set for any CA-set determined by the definition 1.

It was proved in [7] that if $\mathcal{T}$ is dense, then the collection of its SN-sets is a laminar family [9]. It means that two SN-sets are either disjoint or one of them contains the other, hence the family is tree structured under inclusion. So all SN-sets are representable by a tree, called SN-tree. Each node of the SN-tree corresponds to a SN-set. The root corresponds to $\mathcal{L}$, and the leaves correspond to the singletons. The SN-tree can be calculated in $O(n^3)$ time [6]. In the remaining of this section, $\mathcal{T}$ is always a dense triplet set.

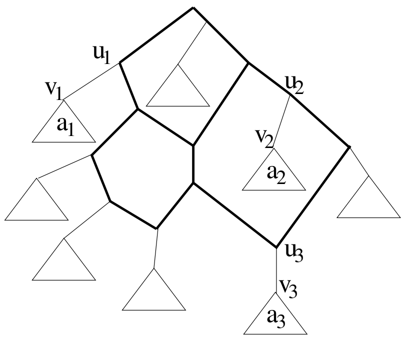
Let A, a be two SN-sets. We say that a is a **child** of A if in the SN-tree, the vertex which represents a is a child of the vertex which represents A.

The next notion is used for a SN-set of $\mathcal{T}$, and is related to a network which is consistent with $\mathcal{T}$.

Definition 2. *Let A be a SN-set of $\mathcal{T}$, and let N be a network consistent with $\mathcal{T}$. We say that A is* **split** *in N if its children are hung below different highest cut-arcs of N (see figure 4).*



(a) The SN-tree of $\mathcal{T}$



(b) a network N consistent with $\mathcal{T}$ in which A is split.

Fig. 3. The SN-set A of $\mathcal{T}$ is split in N, a consistent network of $\mathcal{T}$. Each SN-set a_i, child of A, is hung below a highest cut-arc (u_i, v_i) of N.

A SN-set can be split in some network but not-split in another network both consistent with the same triplet set. So when we say that A is split, we have to be precise in which network. A leaf of a SN-tree, which is always a singleton, is not split in any network because it does not have any children. In every level-1 network, there is only one SN-set which is split: the total leaf set $\mathcal{L}$. Indeed, each leaf set below a highest cut-arc of a level-1 network is a maximal SN-set [6,8], i.e a child of $\mathcal{L}$ as defined in [8].

It should be noticed that the meaning of *split* defined by definition 2 is more restricted than the usual meaning of the word "split". Naturally, we say that A is split if it is split into several disjoint subsets, and each one is hung below a different cut-arc. However, by definition 2, in order that A is *split*, we demand two further conditions: each subset must be a *child* of A, and it must be hung below a *highest* cut-arc. So, for the example in the figure 3, if a_1, a_2, a_3 are hung below three different cut-arcs but one of them is not highest, we will not say that A is *split*. Another example is the one in the figure 4, the root does not represent a *split* SN-set because there are subsets below highest cut-arcs which are its grandchildren or grand-grandchildren, although it is really split into several subsets by normal sense. Hence, if A is *split* by definition 2, none of its descendants can be *split*.

Therefore *with our definition 2, the set of all SN-sets which are split in a network can represent the partition of the leaf set below the highest cut-arcs of* *this network.* Let us see an example of a SN-tree in the figure 4, the black square nodes represent the SN-sets split in a certain network which is not stated here, we have three such sets. The children of these three sets, together with the maximal SN-sets that do not contain any of these three sets are the SN-sets below the highest cut-arcs. In the figure 4, these sets are marked by the black round nodes, they create a partition of the leaf set.

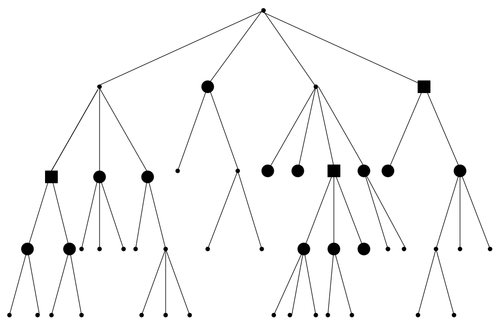


Fig. 4. The black square nodes represent the split SN-sets. The black round nodes represent the SN-sets below the highest cut-arcs.

Next, we will define a new **function** f on a network N consistent with $\mathcal{T}$. This function gives a relation between a SN-set of $\mathcal{T}$ which is split in a network and a set of hybrid vertices of this network. Let N be a level-k network consistent with $\mathcal{T}$. Denote by N_S the simple network of N, by H the set of the hybrid vertices of N_S, so $|H| \leq k$, and by $\mathcal{A}$ the set of all SN-sets split in N. In the figure 4, $\mathcal{A}$ is the set of SN-sets corresponding to the square nodes. Let $\mathcal{H}$ be the set of all subsets of H. We define a function f from $\mathcal{A}$ to $\mathcal{H}$ as follows.

Definition 3. *Given $A \in \mathcal{A}$ and $a_1, \dots a_m$ the children of A in the SN-tree. As A is split in N, each a_i is hung below a highest cut-arc (u_i, v_i) of N. We define:*
$f(A) = \{h \in H | \exists i$ so that $u_i \rightsquigarrow h$ and the path from u_i to h does not contain any internal hybrid vertex (if u_i is a hybrid vertex, then $h = u_i$)$\}$ (see figure 5).

Lemma 2. *The function f has the following properties:*
(i) $\forall A \in \mathcal{A}, f(A) \neq \emptyset$.
(ii) $\forall h \in H$, there are at most three pairwise disjoint sets of $\mathcal{A}$ so that their image by f contains h.

Proof. (i) For any $A \in \mathcal{A}$, we will prove that $\forall u_i, \exists h \in H$ such that $u_i \rightsquigarrow h$ and the path from u_i to h does not contain any internal hybrid vertex. This fact implies $f(A) \neq \emptyset$.

Indeed, if u_i is a hybrid vertex, then we have $h = u_i$.

If u_i is not a hybrid vertex, then there are two arcs starting from u_i: one is (u_i, v_i) and let the second be (u_i, v_i'). Assuming that there is no hybrid vertex of N_S that is reachable from u_i, so both (u_i, v_i) and (u_i, v_i') are cut-arcs. We infer

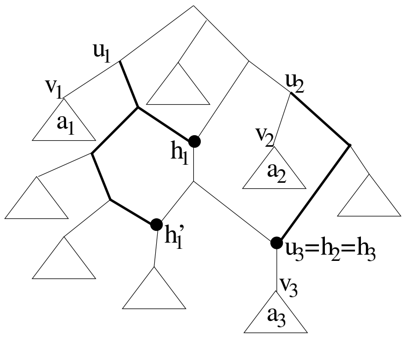


Fig. 5. $f(A) = \{h_1, h'_1, u_3\}$ where A is the split SN-set which has three children a_1, a_2, a_3. The paths, which are in bold, $u_1 \rightsquigarrow h_1$, $u_1 \rightsquigarrow h'_1$, and $u_2 \rightsquigarrow u_3$ do not contain any internal hybrid vertex.

that the arc coming to u_i is also a cut-arc, so (u_i, v_i) is not a highest cut-arc, a contradiction.

(ii) Assuming that there are four pairwise disjoint sets $A_1, A_2, A_3, A_4 \in \mathcal{A}$ so that $\exists h \in f(A_1) \cap f(A_2) \cap f(A_3) \cap f(A_4)$. According to the definition of f, $\forall i \in \{1, 2, 3, 4\}$, there is at least a child a_i of A_i such that no internal vertex of the path from u_i to h is hybrid.

As the four a_i's are distinct, there is at most one u_i that can be equal to h. Hence, there are at least three u'_is that are strictly above h. As h has only two parents, and the path from u_i to h does not contain any internal hybrid vertex, so there exist $i1, i2 \in \{1, 2, 3, 4\}$ so that u_{i2} is placed on the path from u_{i1} to h.

The following proof is illustrated by the figure 6.

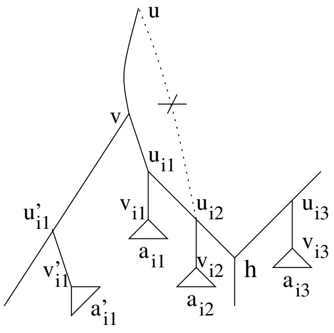


Fig. 6. The triplets $a_{i2}|a_{i1}a'_{i1}$ can not be consistent with the network: all paths from u to a_{i2} have to pass u_{i1}

For convenience, we use the notation $a_i|a_ja_k$ for the set of all triplets $x|yz$ where $x \in a_i, y \in a_j$ and $z \in a_k$.

Let a'_{i1} be another child of A_{i1}. As A_{i1} is a SN-set, and a_{i2} is not included in A_{i1}, by definition of SN-set, $a_{i2}|a_{i1}a'_{i1} \subset \mathcal{T}$. By definition of consistence, there exist two vertices $u \neq v$ and pairwise internally vertex-disjoint paths $u \rightsquigarrow a_{i2}, u \rightsquigarrow v, v \rightsquigarrow a_{i1}, v \rightsquigarrow a'_{i1}$. We remark that all paths starting from a vertex above u_{i1} that come to a_{i2} have to pass u_{i1} because there is no hybrid vertex on the path from u_{i1} to u_{i2}. As u is a vertex above u_{i1}, the path from u to a_{i2} has to pass u_{i1}. The path from v to a_{i1} has to pass u_{i1} too. So the two paths $u \rightsquigarrow a_{i2}$ and $v \rightsquigarrow a_{i1}$ are not internally vertex-disjoint, a contradiction. □

With the precedent lemma, we have the following lemma which allows us to bound the number of SN-sets which are split in a level-k network.

Lemma 3. *(Fundamental) Let $\mathcal{T}$ be a dense triplet set. For any level-k network N consistent with $\mathcal{T}$, if $\mathcal{A}$ is the set of all SN-sets of $\mathcal{T}$ which are split in N, then $|\mathcal{A}| \leq 3k$.*

Proof. Firstly, we observe that all elements of $\mathcal{A}$ are pairwise disjoint. Actually, for any two SN-sets, they are either disjoint or included one in another. However, if a SN-set A is split in a network, its children are not. It means that if A is in $\mathcal{A}$, its subsets are not. Then, the two sets of $\mathcal{A}$ can not be included one in another. They are disjoint.

Let $H' \subseteq H$ be the image of f. For any $A \in \mathcal{A}$, and $h \in H'$, we say that A corresponds to h, and h corresponds to A if $f(A)$ contains h. We infer from the lemma 2 that each element of $\mathcal{A}$ corresponds to at least one element of H', and each element of H' corresponds to at most three elements of $\mathcal{A}$. So $|\mathcal{A}| \leq 3|H'| \leq 3|H| \leq 3k$. □

Moreover, if we are only interested in finding a certain consistent network, we can have a better bound using the following lemma. The idea is to modify an arbitrary consistent network in such a way that does not increase the level, and such that the resulted network is still consistent with $\mathcal{T}$ and has a particular property. However the class of modified networks may not contain the one with the minimum number of reticulations.

Lemma 4. *Let $\mathcal{T}$ be a dense triplet set, let N be a level-k network consistent with $\mathcal{T}$. If the simple network of N has level greater than 1, then there exists another level-k network N' consistent with $\mathcal{T}$ such that: for any SN-set A which is split in N', $|f(A)| \geq 2$.*

Proof. Assuming that there exists a SN-split A of N such that $|f(A)| = 1$. Denote $f(A) = \{h\}$. In N_S, there are 2 paths leading to h. So there are 2 cases that can happen.

In the first case (figure 7(a)), u_i's are all placed on one path leading to h, for example on the left one. Let u_1 be the highest and u_f be the lowest vertex on all u_i. There are two possible positions for u_f: either it is right above h, i.e (u_f, h) is an arc, or it is equal to h. Let G_A be the sub-network of N on A.The network N' is obtained from N by the following modifications: delete all vertices u_i's, highest cut-arcs connecting with u_i's and sub-networks on a_i's; at the position of u_f, add a new arc which connects to G_A at u_1 (figure 7(b)).

In the second case (figure 7(c)), u_i's are placed on the two paths leading to h. We remark that the leaf set below h has to be also a child of A. Let G_A be the sub-network of N on A, and let G'_A be the network obtained from G_A by gluing the top of the two branches of G_A into one vertex u. The network N' is obtained from N by the following modifications: delete all vertices u_i's, highest cut-arcs connecting with u_i's and sub-networks on a_i's; at the position of h, add a new arc which connects to G'_A at u (figure 7(d)).

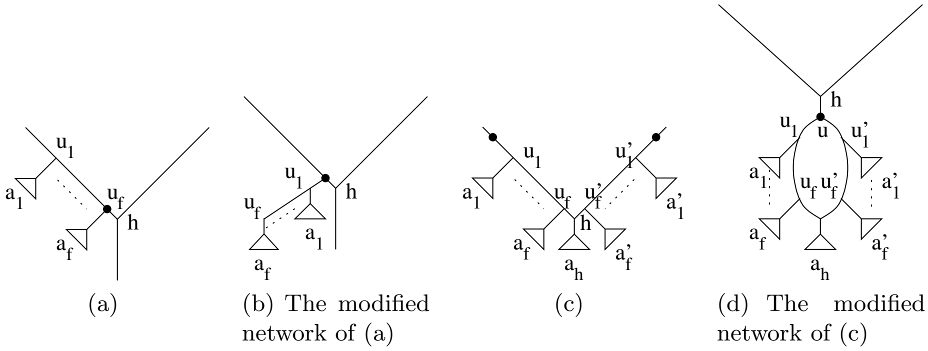


Fig. 7. The modified networks are level-k, still consistent with $\mathcal{T}$, and have all leaves of A hung below a highest cut-arc

In both two cases, we can verify that the modifications do not increase the level of the network, the new network is still consistent with all triplets of $\mathcal{T}$, and A is no longer split in the new network: it is now hung below a highest cut-arc. The fact that the simple network of N has level greater than 1 assures that the new network does not contain any two parallel arcs with the same extremities.

By modifying the network for any split SN-set whose image by f contains only one element, we obtain finally a network in which there is no longer such split SN-set. In addition, the lemma 2 says that the image by f of any split SN-sets is not empty. Then we have a new network in which $|f(A)| \geq 2$ for any split SN-set A. $\square$

Lemma 5. *Let $\mathcal{T}$ be a dense triplet set. If $\mathcal{T}$ is consistent with a level-k network N, then there exists a level-k network N' consistent with $\mathcal{T}$ which satisfies: let $\mathcal{A}$ be the set of SN-sets of $\mathcal{T}$ which are split in N', then $|\mathcal{A}| \leq \lfloor \frac{3k}{2} \rfloor$.*

Proof. If the simple network of N is of level-1, we choose $N' = N$. As comment after the definition 2, in this case, there is only one split SN-set: $\mathcal{L}$. Hence $|\mathcal{A}| = 1 \leq \lfloor \frac{3k}{2} \rfloor$, and the lemma is obviously true in this case.

Otherwise, by lemma 4, there exists a level-k network N' consistent with $\mathcal{T}$ which satisfies: for any SN-set A split in N', $|f(A)| \geq 2$. Let $H' \subseteq H$ be the image of f. For any $A \in \mathcal{A}$, and $h \in H'$, we say that A corresponds to h, and h corresponds to A if $f(A)$ contains h. So each element of $\mathcal{A}$ corresponds to at least two elements of H' (lemma 4), and each element of H' corresponds to at most three elements of $\mathcal{A}$ (lemma 2). Then $|\mathcal{A}| \leq \lfloor \frac{3}{2}|H'| \rfloor \leq \lfloor \frac{3}{2}|H| \rfloor \leq \lfloor \frac{3k}{2} \rfloor$. □

Theorem 1. *Given a dense triplet set $\mathcal{T}$, and a fixed positive integer k, it is possible to construct a level-k network consistent with $\mathcal{T}$, if one exists, in $O(|\mathcal{T}|^{k+1} n^{\lfloor \frac{3k}{2} \rfloor + 1})$ time.*

Proof. The algorithm, which is described in the algorithm 1, constructs from each SN-set A, in small-big order, a level-k sub-network consistent with $\mathcal{T}|A$, if there is any. So the leaf set of such sub-network corresponds to a SN-set. And the final constructed sub-network whose the leaf set is the biggest SN-set, i.e $\mathcal{L}$, is the wanted one.

Now, we will show how to construct a sub-network on a SN-set A, knowing a sub-network on each descendant of A. By the lemma 5, each sub-network must have $\leq \lfloor \frac{3k}{2} \rfloor$ split SN-sets. Each split SN-set is a descendant of A, and A has totally $O(n)$ non-singleton descendants. So there are $O(n^{\lfloor \frac{3k}{2} \rfloor})$ possibilities of $\mathcal{A}$, the set of all SN-sets split in the sub-network. For each choice of $\mathcal{A}$, we calculate the corresponding partition $\mathcal{P}$, and then $(\mathcal{T}|A)\nabla\mathcal{P}$. After that, we search for a simple network consistent with $(\mathcal{T}|A)\nabla\mathcal{P}$. Theorem 3 in [11] says that it is possible to construct all simple level-k networks consistent with a dense triplet set $\mathcal{T}$ in $O(|\mathcal{T}|^{k+1})$ time. Remind that each leaf of the simple network corresponds to a part of the partition $\mathcal{P}$, which is also a SN-set descendant of A. Therefore, to obtain the sub-network on A, we replace each leaf of the simple network by the sub-network on the corresponding descendant of A, which is already found before. If for all choices of $\mathcal{A}$, there is always no simple network consistent with $(\mathcal{T}|A)\nabla\mathcal{P}$, then there is not any consistent sub-network on A. In this case we can conclude immediately that there is no network consistent with $\mathcal{T}$, and the algorithm returns *null*. Indeed, if there exists a network consistent with $\mathcal{T}$, then for any $A \subseteq \mathcal{L}$ there exists also a network consistent with $\mathcal{T}|A$.

So, it takes totally $O(|\mathcal{T}|^{k+1} n^{\lfloor \frac{3k}{2} \rfloor})$ time to find a sub-network on A, knowing a sub-network on each descendant of A.

As there are only $O(n)$ non-singleton SN-sets, and we retain only one sub-network on each SN-sets, we have to construct only $O(n)$ sub-networks. Therefore, the complexity will be multiplied by n. The construction of SN-tree takes $O(n^3)$, all other operations take a negligible time compared with the time to find all possible decompositions. So, the total complexity is $O(|\mathcal{T}|^{k+1} n^{\lfloor \frac{3k}{2} \rfloor + 1})$. □

As a consequence, using a recursive property of the network with minimum number of hybrid vertices, we can solve the problem of finding the consistent level-k network, for any fixed k, which minimizes the number of hybrid vertices in polynomial time. The only difference from the algorithm 1 is that, in this algorithm, each sub-network constructed on a SN-set A is always the consistent one with the minimum number of reticulations among those who are consistent with $\mathcal{T}|A$. So, instead of searching a certain simple network consistent with $(\mathcal{T}|A)\nabla\mathcal{P}$, we search for the one such that the corresponding sub-network minimizes the number of reticulations. As we can find all simple network in polynomial time, the complexity remains polynomial.

Algorithm 1. Level-k network

```
Require: A dense triplet set $\mathcal{T}$ and a fixed positive integer $k$.
Ensure: A level-$k$ network consistent with $\mathcal{T}$, if one exists; otherwise, null.
  Let $R$ be the SN-tree of $\mathcal{T}$.
  For every singleton $u$ of $\mathcal{L}$, define $N_u$ to be the network containing only one leaf $u$.
  for (each non-singleton SN-set $A$ of $R$, in bottom-up order) do
    Let $\mathcal{T}' = \mathcal{T}|A$.
    $found = false$;
    for ($i = 1$; $i \leq \frac{3k}{2}$ and $!(found)$;$i++$) do
      for (each $i$ disjoint non-singleton SN-sets, descendants of $A$) do
        Consider these $i$ SN-sets as the split SN-sets, to calculate the partition $\mathcal{P}$ of
        the leaf set $A$. Then, calculate $\mathcal{T}'\nabla\mathcal{P}$.
        if (there exists a simple network $Ns_A$ consistent with $\mathcal{T}'\nabla\mathcal{P}$) then
          $N_A$ is obtained by replacing each leaf of $Ns_A$ by the sub-network already
          found on the corresponding descendant of $A$.
          $found = true$; break.
        end if
      end for
    end for
    if $found = false$ then
      return null.
    end if
  end for
  return $N_{\mathcal{L}}$.
```

Theorem 2. *Given a dense triplet set $\mathcal{T}$, and a fixed positive integer k, it is possible to construct a level-k network consistent with $\mathcal{T}$ which minimizes the number of hybrid vertices, if one exists, in $O(|\mathcal{T}|^{k+1}n^{3k+1})$ time.*

Proof. Let N be a level-k network consistent with $\mathcal{T}$, let $N_1, \ldots, N_m$ be the sub-networks of N below the highest cut-arcs of N, and let P_i be the leaf set of N_i. The number of hybrid vertices of N is equal to the sum of the number of hybrid vertices of each N_i and the number of hybrid vertices of the simple network of N. So if N is the network that minimizes the minimum number of hybrid vertices, then each N_i has to be also the network which minimizes the number of hybrid vertices among those who are consistent with $\mathcal{T}|P_i$. This property allows us to have a recursive construction as the algorithm 1. Indeed, in the algorithm 1, when we construct a sub-network on a SN-set A, instead of taking any simple network consistent with $(\mathcal{T}|A)\nabla\mathcal{P}$, we take the one such that the corresponding network minimizes the number of hybrid vertices. The construction, which is described in the algorithm 2, stays in polynomial time because we can find all simple level-k networks in $O(|T|^{k+1})$ time, and all possible partitions of the leaf set in $O(n^{3k})$ time. Finally, the construction of $O(n)$ sub-networks on $O(n)$ SN-sets makes the total complexity $O(|\mathcal{T}|^{k+1}n^{3k+1})$. □

Algorithm 2. Level-k network with the minimum number of hybrid vertices

```
Require: A dense triplet set $\mathcal{T}$ and a fixed positive integer $k$.
Ensure: A level-$k$ network consistent with $\mathcal{T}$ that minimizes the number of hybrid
  vertices, if one exists; $null$ otherwise.
  Calculate the SN-tree of $\mathcal{T}$.
  For every singleton $u$ of $\mathcal{L}$, define $N_{u_{min}}$ to be the network containing only one leaf
  $u$.
  for (each non-singleton SN-set $A$ of $\mathcal{T}$, in small-big order) do
    Let $\mathcal{T}' = \mathcal{T}|A$.
    $N_{A_{min}} = null; min = n$.
    for ($i = 1$; $i \leq 3k$; $i++$) do
      for (each $i$ disjoint non-singleton SN-sets descendants of $A$) do
        Consider these $i$ SN-sets as the split SN-sets, to calculate the partition $\mathcal{P}$ of
        the leaf set $A$. Then, calculate $\mathcal{T}'\nabla\mathcal{P}$.
        for (each simple network $Ns_A$ consistent with $\mathcal{T}'\nabla\mathcal{P}$) do
          $N_A$ is obtained by replacing each leaf of $Ns_A$ by the sub-network already
          found on the corresponding descendant of $A$.
          $m$ = the number of hybrid vertices of $N_A$.
          if ($m < min$) then
            $min = m; N_{A_{min}} = N_A$.
          end if
        end for
      end for
    end for
    if ($N_{A_{min}} = null$) then
      return $null$;
    end if
  end for
  return $N_{\mathcal{L}_{min}}$.
```

4 Conclusion and Perspectives

To any triplet set $\mathcal{T}$ we can define its *treerank*$(\mathcal{T})$ as the minimum k for which there exists a level-k network which represents $\mathcal{T}$. This measures the distance from $\mathcal{T}$ to a tree in term of number of hybrid vertices. We proved in the previous section that for dense triplets, and for any fixed k, checking if $treerank(\mathcal{T}) \leq k$ can be done in polynomial time. Therefore this new parameter is analogous to treewidth for graphs and we conjecture that its computation is also NP-hard for dense triplet set or extremely dense triplet set. However, compared with the complexity of the existing efficient algorithms for the cases $k = 0, 1, 2$, the results given here could be improved for level-k networks. Another interesting question is under which conditions on the triplet set $\mathcal{T}$ there is only one network consistent with $\mathcal{T}$. It would be interesting to know whether the condition of density on the triplet set can be relaxed so that there is still a polynomial algorithm to construct a consistent level-k network, if there any, with any fixed k.

Thanks. We would like to thank Philippe Gambette for many useful references and comments, and also for his practical web site *Who is Who in Phylogenetic Networks* which gave us an easy access to the current literature on phylogenetic networks.

References

1. Aho, A.V., Sagiv, Y., Szymanski, T.G., Ullman, J.D.: Inferring a Tree from Lowest Common Ancestors with an Application to the Optimization of Relational Expressions. SIAM Journal on Computing 10(3), 405–421 (1981)
2. Byrka, J., Gawrychowski, P., Huber, K.T., Kelk, S.: Worst-case optimal approximation algorithms for maximizing triplet consistency within phylogenetic networks (2008) arXiv:0710.3258v3 [q-bio.PE]
3. Choy, C., Jansson, J., Sadakane, K., Sung, W.-K.: Computing the Maximum Agreement of Phylogenetic Networks. Theoretical Computer Science 355(1), 93–107 (2005)
4. Gasieniec, L., Jansson, J., Lingas, A., Ostlin, A.: Inferring ordered trees from local constraints. In: CATS 1998, vol. 20(3), pp. 67–79 (1998)
5. Henzinger, M.R., King, V., Warnow, T.: Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. Algorithmica 24(1), 1–13 (1999)
6. Jansson, J., Nguyen, N.B., Sung, W.-K.: Algorithms for Combining Rooted Triplets into a Galled Phylogenetic Network. SIAM Journal on Computing 35(5), 1098–1121 (2006)
7. Jansson, J., Sung, W.-K.: Inferring a Level-1 Phylogenetic Network from a Dense Set of Rooted Triplets. In: Chwa, K.-Y., Munro, J.I.J. (eds.) COCOON 2004. LNCS, vol. 3106, pp. 462–471. Springer, Heidelberg (2004)
8. Jansson, J., Sung, W.-K.: Inferring a Level-1 Phylogenetic Network from a Dense Set of Rooted Triplets. Theoretical Computer Science 361(1), 60–68 (2006)
9. Schrijver, A.: Combinatorial Optimization - Polyhedra and Efficiency. Springer, Heidelberg (2003)
10. van Iersel, L., Keijsper, J., Kelk, S., Stougie, L., Hagen, F., Boekhout, T.: Constructing level-2 phylogenetic networks from triplets. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 450–462. Springer, Heidelberg (2008)
11. van Iersel, L., Kelk, S.: Constructing the Simplest Possible Phylogenetic Network from Triplets. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 472–483. Springer, Heidelberg (2008)
12. van Iersel, L., Kelk, S., Mnich, M.: Uniqueness, Intractability and Exact Algorithms: Reflections on level-k Phylogenetic Networks. JBCB (2009)

The Structure of Level-k Phylogenetic Networks

Philippe Gambette, Vincent Berry, and Christophe Paul

Département informatique, L.I.R.M.M., C.N.R.S. - Université Montpellier II
{gambette,vberry,paul}@lirmm.fr

Abstract. Evolution is usually described as a phylogenetic tree, but due to some exchange of genetic material, it can be represented as a *phylogenetic network* which has an underlying tree structure. The notion of *level* was recently introduced as a parameter on realistic kinds of phylogenetic networks to express their complexity and tree-likeness. We study the structure of level-k networks, and how they can be decomposed into level-k generators. We also provide a polynomial time algorithm which takes as input the set of level-k generators and builds the set of level-$(k+1)$ generators. Finally, with a simulation study, we evaluate the proportion of level-k phylogenetic networks among networks generated according to the coalescent model with recombination.

1 Introduction

Networks have been introduced in phylogenetics to generalize the tree model of evolution which can only represent *speciation* events. In a phylogenetic network, additional branches join vertices already connected by a path, hence defining *reticulations*. This enables to represent hybridization [13, 24], recombination [15, 34] or lateral gene transfer events [14, 26]. Phylogenetic networks are a very active field of computational molecular biology and a number of algorithms have been developed recently to reconstruct such objects or parts thereof from various kinds of input: sequences, splits, distances, quartets, rooted or unrooted trees, or networks (see [16, 9] for a comprehensive list of papers).

The fact that networks are generally hard to handle gave rise to many different restrictions on their structure in order to get tractable algorithms. These restrictions are mostly described in terms of combinatorial patterns allowed or forbidden in the various restrictions. We examine here the broad class of networks called *explicit* networks or *reticulate* networks, in which reticulations are interpreted as precise biological events. In this context, a network is a rooted directed acyclic graph whose vertices have degree at most 3 – speciation vertices have indegree 1 and outdegree 2 and reticulation vertices have indegree 2 and outdegree 1. To cover all such explicit phylogenetic networks, the *level-k* hierarchy was introduced in [5]. In this setting, a phylogenetic network is viewed as a blobbed-tree [11], that is a network with tree-like parts and non reticulate ones called *blobs*. The *level* of a network reflects the complexity of its blobs: it is defined as the maximum number of reticulations inside a blob of the network.

Level-1 networks correspond to a class of explicit networks, first studied in 1998 [29] and later named *galled trees* [34, 12], for which many polynomial algorithms have been found [12, 4, 30, 20, 21, 6]. The level-k hierarchy can be seen as a promising framework to generalize these algorithms to all explicit phylogenetic networks.

Although level-k networks have recently attracted a lot of attention in the context of reconstruction from triplets [3, 17, 18, 19, 33] or maximum agreement subnetwork [5], their combinatorial structure has not yet been studied in detail. A notable exception is the work of [17], who introduced combinatorial patterns called *level-k generators* from which *simple* level-k networks [17] can be characterized. Yet, complete lists of generators were not easy to obtain for the first levels of the hierarchy: level-2 generators were only obtained by a case analysis, while the 65 level-3 generators were obtained by a brute force algorithm [22].

In this paper, we generalize these results. In Section 2, we give explicit rules to build, for all k, all level-$(k+1)$ generators from level-k generators. On this basis, we provide an algorithm that builds level-$(k+1)$ generators in time that is polynomial in the number of level-k generators. We use this algorithm to compute the 1993 level-4 generators. These generators can be downloaded as supplementary material from `http://www.lirmm.fr/~gambette/ProgGenerators.php`. We also provide lower and upper bounds on the number of level-k generators. Section 3 focuses on the structure of level-k networks. We show how they decompose into level-k generators. Finally, in Section 4, we consider the relevance of networks with a small level in the context of the coalescent model with recombination. For this purpose, we measure the proportion of level-k phylogenetic networks among networks generated according to this model.

2 Construction of Level-k Generators

2.1 Definitions

A *phylogenetic tree* is a rooted binary tree with directed arcs and distinctly labeled leaves. A *phylogenetic network* is a generalization of a phylogenetic tree, defined as a directed acyclic graph in which exactly one vertex has indegree 0 and outdegree 2 (the root) and all other vertices have either indegree 1 and outdegree 2 (*split vertices*), indegree 2 and outdegree ≤ 1 (*hybrid vertices*) or indegree 1 and outdegree 0 (*leaves*). The leaves have distinct labels. Note that in this graph, we allow multiple arcs, as is shown by the blob containing r_1 in Fig. 1. Choosing whether to allow this configuration (an "empty" cycle in the network) in the definition of a phylogenetic network is just a technical point (here we allow it to be able, later, to define level-k generators as level-k phylogenetic networks).

A directed acyclic graph is *biconnected* if it contains no vertex whose removal disconnects the graph. A *biconnected component*, or *blob*, of a pylogenetic network, is a maximal biconnected subgraph. An arc is a *cut-arc* if removing it disconnects the graph. For any arc (u, v) of a phylogenetic network N, u is a parent of v, and v a child of u. We say that u is *over* v, or v is *under* u in N, if N contains a directed path from u to v.

A phylogenetic network is called a *level-k phylogenetic network* [5] (or just *level-k network*) if each biconnected component contains at most k hybrid vertices. A level-k network which is not a level-$(k-1)$ network is called a *strict level-k phylogenetic network*. A level-0 phylogenetic network is a phylogenetic tree, and a level-1 network is commonly called a *galled tree*. Many hard problems can be solved in polynomial time on these classes of networks. However, these networks only cover part of the practical networks – see section 4, which motivates the study of upper levels.

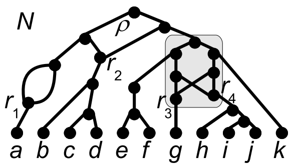


Fig. 1. A level-2 network N with root ρ and leaf set $\{a, b, c, d, e, f, g, h, i, j, k\}$. All unlabeled vertices are split vertices. The gray area is a biconnected component with two hybrid vertices, namely r_3 and r_4. The arc from r_2 to its child is a cut-arc. All arcs are directed downward but orientation is not displayed for the sake of readability, as in the next figures.

Definition 2.1 ([17]). *A* level-k generator *(see Fig. 2) is a biconnected strict level-k network. Vertices of outdegree 0 and arcs of a level-k generator are called its* sides*, they are* empty *if no subtree is hanging from them. We call S_k the set of generators of level at most k, and S_k^* the set of level-k generators.*

Phylogenetic networks have been defined above such that a level-k generator is a level-k phylogenetic network (contrary to [17] we allow phylogenetic networks to contain hybrid vertices of outdegree 0). In particular, level-k generators and level-k networks are not allowed to contain vertices whose indegree and outdegree both equal 1.

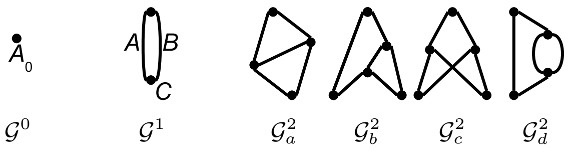


Fig. 2. Level-0 generator $\mathcal{G}^0$, level-1 generator $\mathcal{G}^1$, and level-2 generators: $\mathcal{G}^2_a$, $\mathcal{G}^2_b$, $\mathcal{G}^2_c$ and $\mathcal{G}^2_d$

2.2 Construction Rules

The level-0, respectively level-1, generator is called $\mathcal{G}^0$, respectively $\mathcal{G}^1$). In [17], the level-2 generators are found by a case analysis which can also be applied to compute the 65 level-3 generators [22]. Here we provide rules to compute all level-$(k+1)$ generators from level-k generators.

Definition 2.2. *Let N be is a level-k generator. We define the following partial order $\succ_N$ on its sides: for two sides X and Y of N, $Y \succ_N X$ if the source of arc Y (or Y itself, if Y is a vertex) can be reached from the target of arc X (or X itself, if X is a vertex).*

The network $R1(N, X, Y)$ is obtained by choosing two sides X and Y of N, such that if $X = Y$ then X is not a hybrid vertex, and hanging a new hybrid vertex under X and Y (see Fig. 3).

The network $R2(N, X, Y)$ is obtained by choosing a side X of N and an arc $Y \not\succ_N X$ of N, and putting an arc from X to Y, which creates a new hybrid vertex "inside" arc Y.

Note that sides X and Y have a symmetric role for rule $R1$ but not for rule $R2$. When we build $R1(N, X, Y)$ from N, we say that we apply rule $R1$ on X and Y (and the same for $R2$). Note also that in the definition of $R1(N, X, X)$, we only allow X to be an arc, or, in the particular case of $N = \mathcal{G}^0$, to be its only node.

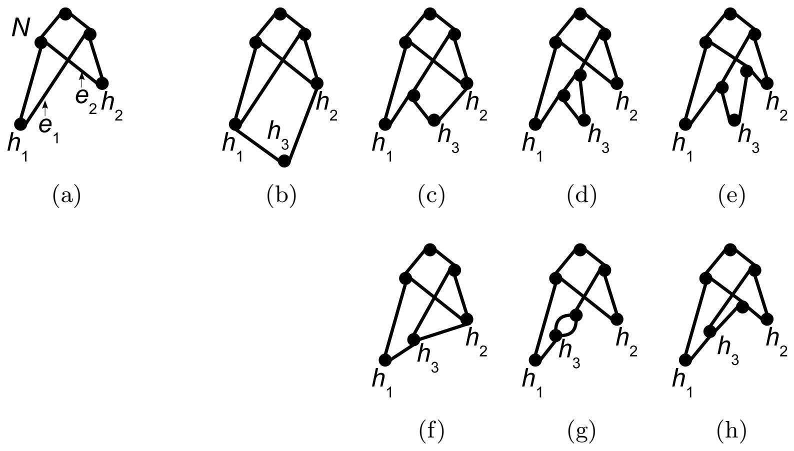


Fig. 3. Results of applying rules $R1$ and $R2$ on a level-2 generator N (a) depending on the type of side (arc or hybrid vertex) where it is applied: $R1(N, h_1, h_2)$ (b), $R1(N, e_1, h_2)$ (c), $R1(N, e_1, e_1)$ (d), $R1(N, e_1, e_2)$ (e), $R2(N, h_2, e_1)$ (f), $R2(N, e_1, e_1)$ (g), $R2(N, e_2, e_1)$ (h). In each case, a new hybrid node, h_3 is created.

Proposition 2.1. *Let N be a level-k generator and X and Y two sides of N such that $N_1 = R1(N, X, Y)$, resp. $N_2 = R2(N, X, Y)$, is well-defined. Then N_1, resp. N_2, is a level-$(k + 1)$ generator.*

Sketch of proof. It is easy to check that applying rule $R1$ or $R2$ adds, in all cases, exactly one reticulation node, and indeed provides a level-$(k + 1)$ generator. □
We have seen in Proposition 2.1 that we can build level-$(k + 1)$ generators from level-k generators, it remains to be proved that any level-$(k + 1)$ generator can be obtained in this way.

Proposition 2.2. *For any level-$(k+1)$ generator N, there exists a level-k generator N', and some sides X and Y of N' such that $N = R1(N', X, Y)$ or $N = R2(N', X, Y)$.*

Sketch of proof. The proof works by "reversing the rules" and finding an appropriate target vertex to remove by the reversed rule. □

2.3 Bounding the Number of Generators

The rules we have defined can be used to obtain lower and upper bounds on the number of level-k generators.

Proposition 2.3. *For $k \geq 1$, a level-k generator has at most $3k-1$ vertices and $4k-2$ arcs.*

Proof. The unique level-1 generator has two vertices and two arcs. By Proposition 2.2, each level-k generator is obtained by applying rule $R1$ or $R2$ to a level-$(k-1)$ generator, hence by k applications of rules $R1$ or $R2$. We then notice that each application of rule $R1$ or $R2$ just adds at most three vertices and four arcs. The bounds are reached when $R2$ is repeatedly applied on two different arcs as in Fig. 3(e). □

Proposition 2.4. *The number g_k of level-k generators is at least 2^{k-1}.*

Proof. The property is true for $k = 0$, so we fix $k \geq 1$. We define an injection G_k between the set of integers $[0..2^{k-1}-1]$ and a set of level-k generators. The generator $G_k(a)$ is build from the binary representation of a using only rule $R1$. The construction process is illustrated in Fig. 4. Let $a = \sum_{i=0}^{k-2} a_i 2^i \in [0..2^{k-1}-1]$ such that $a_i \in \{0,1\}$. We start with the level-1 generator $\mathcal{G}^1$, then for i from 0 to $k-2$:

- let h_i be the lowest hybrid vertex of the currently built generator G.
- let e_i be the edge from the highest parent of h_i (a simple proof by induction shows that there always exists one parent of h_i under the other).
- change G into $R1(G, e_i, h_i)$ if a_i=1, into $R1(G, e_i, e_i)$ if $a_i = 0$.

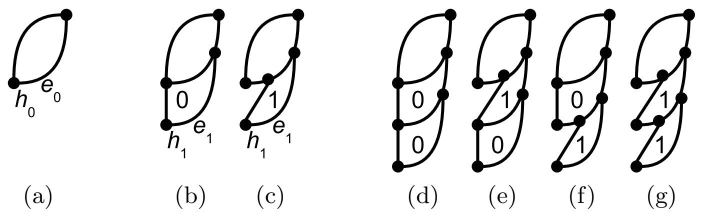


Fig. 4. Construction of 2^{k-1} non-isomorphic level-k generators : we start from generator $\mathcal{G}^1$ (a) and apply $R1(\mathcal{G}^1, e_0, h_0)$ to get $G_2(0)$ (b), $G_2(1) = R1(\mathcal{G}^1, e_0, e_0)$ (c), $G_3(0) = R1(G_2(0), e_1, h_1)$ (d), $G_3(1) = R1(G_2(1), e_1, h_1)$ (e), $G_3(2) = R1(G_2(0), e_1, e_1)$ (f), $G_3(3) = R1(G_2(1), e_1, e_1)$ (g)

This way, we get for $G_k(a)$ a digraph whose structure is a chain of cycles which encodes the binary representation of a. Proposition 2.1 ensures that $G_k(a)$ is a level-k generator. Thus, for each k, we can build a set $\{G_k(a), a \in [0..2^{k-1}-1]\}$ of 2^{k-1} level-k generators. These generators are obviously non isomorphic, since they are each composed by a specific chain of two kinds of cycles. □

Proposition 2.5. *The number g_k of level-k generators is lower than $k!^2 50^k$.*

Proof. Proposition 2.3 ensures that the number of arcs of a level-k generator is less than $4k$, and its number of hybrid vertices is k, so its number of sides is less than $5k$. When applying the k^{th} rule $R1$ or $R2$, we choose a pair of sides, that is hybrid vertices or arcs, so there are less than $(5k)^2$ possibilities. Thus $g_{k+1} \leq 2(5k)^2 g_k < 50k^2 g_k$, so finally $g_k < k!^2 50^k$. □

Note that although these bounds are not tight, they give useful information on level-k generators. The lower bound shows that there is an exponential number of level-k generators, which implies, by the decomposition Theorem of Section 3, a great complexity inside the blobs of a network of high level. The upper bound for g_{k+1} from g_k and the fact that $g_3 = 65$ [22] shows that it seems realistic to generate automatically level-4 and 5 generators at least.

2.4 The Generator Construction Algorithm

We now study how to use rules $R1$ and $R2$ in practice to build level-$(k+1)$ generators knowing the set of level-k generators. Note that different sequences of rules may produce isomorphic level-k generators. Hence, isomorphic level-k generators have to be removed in the process.

Theorem 2.1. *There exists a polynomial algorithm which takes as input the set S_k^* of all level-k generators and outputs the set of all level-$(k+1)$ generators.*

Sketch of proof. The algorithm, `BuildGenerators`, detailed below, works by simply trying to apply rules $R1$ and $R2$ on any generator in S_k^*, then removing the isomorphic ones. To prove the polynomial complexity, the main point is the fact that the isomorphism test which is Graph Isomorphism-complete on general digraphs [35], can be done, in our case, in polynomial time [25, 27] in the size of the graph which is polynomial in $|S_k^*|$ by propositions 2.3 and 2.4. □

Though graph isomorphism is decidable in polynomial time for graphs of bounded maximum degree, there exists no implementation of this algorithm, which seems difficult to use in practice [23]. Instead, to actually build all level-4 generators from the 65 level-3 generators, we used an exponential time backtracking algorithm which tests isomorphism by trying to identify corresponding vertices by going through both input graphs at the same time. Among the 8501 level-4 generators built by applying rule $R1$ or $R2$, a total of 1993 are non-isomorphic. The list of these generators, the program to build them, its source and implementation notes are available at `http://www.lirmm.fr/~gambette/ProgGenerators.php`. Note that the sequence 1,4,65,1993 is not present in the On-Line Encyclopedia of Integer Sequences [31].

Algorithm 1. *BuildGenerators* builds the set S of level-$(k+1)$ generators from the set S_k^* of level-k generators in polynomial time

```
BuildGenerators($S_k^*$: set of level-$k$ generators)
$S \leftarrow \emptyset$
forall level-k generators $g$ in $S_k^*$ do
    forall pairs $(X, Y)$ of sides of $g$ do
        if rule $R1$ can be applied on sides $X$ and $Y$ then
            $g' \leftarrow R1(g, X, Y)$
            forall level-$(k+1)$ generators $h$ in $S$ do
                if $g'$ is not isomorphic to $h$ then $S \leftarrow S \cup \{g'\}$
        if rule $R2$ can be applied on sides $X$ and $Y$ then
            $g' \leftarrow R2(g, X, Y)$
            forall level-$(k+1)$ generators $h$ in $S$ do
                if $g'$ is not isomorphic to $h$ then $S \leftarrow S \cup \{g'\}$
return $S$
```

3 Generating Level-*k* Phylogenetic Networks

The concept of generator was introduced in [17] to build restrictions of level-k phylogenetic networks, called *simple*, which contain no cut-arc except the *trivial* ones leading to leaves. We give an explicit composition theorem which shows how generators can be used to build any level-k network, and exhibits the link with the blobbed-tree structure of phylogenetic networks.

Definition 3.1. *Given a set S_k of generators of level at most k, and a phylogenetic network N, we define the following rules, illustrated in Fig. 5:*

- *$MergeRoot_k(G_0, G_1)$ is obtained by hanging generators G_0 and $G_1 \in S_k$ under a root.*
- *$Attach_k(v, G, N)$ is the network obtained by adding an arc from hybrid vertex $v \in N$ of outdegree 0 to a copy of a generator $G \in S_k$.*
- *$Attach_k(a, G, N)$ is the network obtained by subdividing arc a (i.e. adding a vertex of indegree 1 and outdegree 1 inside a) and adding an arc from the created vertex to a copy of $G \in S_k$.*

Note that rule $MergeRoot_k$ can be used only once, and that it is used for level-k networks that are disconnected when removing their root.

Theorem 3.1. *N is a level-k network iff there exists a sequence of $r \in \mathbb{N}$ locations (arcs or hybrid vertices) $(\ell_j)_{j\in[1,r]}$ and a sequence of generators $(G_j)_{j\in[0,r]}$ in S_k, such that:*

$$\begin{aligned} N = {} & Attach_k(\ell_r, G_r, Attach_k(\ldots \\ & Attach_k(\ell_2, G_2, Attach_k(\ell_1, G_1, G_0))\ldots)), \\ \text{or } N = {} & Attach_k(\ell_r, G_r, Attach_k(\ldots \\ & Attach_k(\ell_2, G_2, MergeRoot_k(G_1, G_0))\ldots)). \end{aligned}$$

Sketch of proof. *The proof works by induction.* □

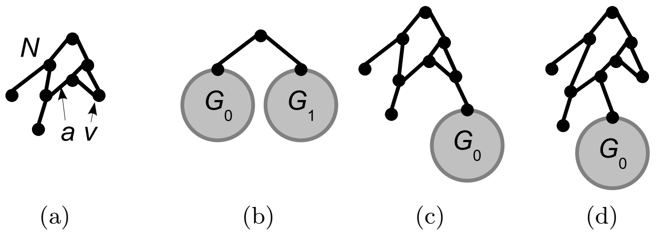


Fig. 5. Rules for building a level-k network from generators of level at most k: a phylogenetic network N (a); the network obtained by applying $MergeRoot_k(G_0, G_1)$ (b), $Attach_k(v, G_0, N)$ (c), and $Attach_k(a, G_0, N)$ (d)

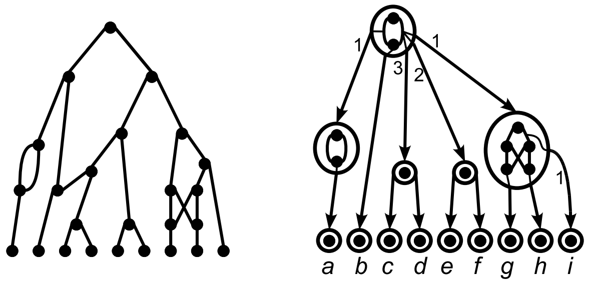


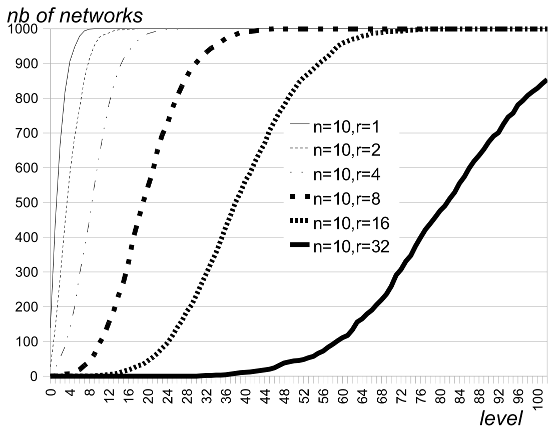
Fig. 6. A level-2 phylogenetic network and its canonical decomposition tree: each node of the tree contains a generator of level $\leq k$; each arc of the tree is linked to a side of the generator at the source node, and labeled by an integer showing in which order it is attached to the side, if the side is an arc

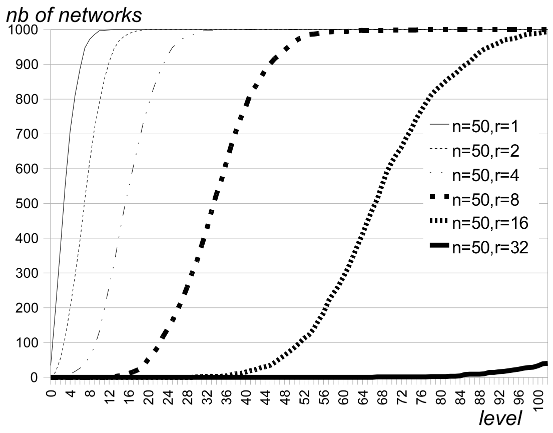
Theorem 3.1 characterizes level-k networks by a sequence of rules on a finite set of generators. In this form, the characterization does not yield canonicity: two different sequences of rule applications may lead to the same phylogenetic network (typically, by just changing the order in which rules are applied).

Table 1. Number of simulated networks falling in each class as a function of the recombination rate r = 0, 1, 2, 4, 8, 16, and 32 for sample size n = 10 or n = 50

n	r	Tree	Level-1	Level-2	Level-3	Level-4	Level-5
10	0	1000	1000	1000	1000	1000	1000
10	1	139	440	667	818	906	948
10	2	27	137	281	440	582	691
10	4	1	21	53	85	136	201
10	8	0	1	1	6	7	12
10	16, 32	0	0	0	0	0	0
50	0	1000	1000	1000	1000	1000	1000
50	1	34	198	373	557	709	811
50	2	0	15	54	117	200	292
50	4	0	1	1	2	9	17
50	8, 16, 32	0	0	0	0	0	0

(a)





(b)

Fig. 7. Level-k phylogenetic networks and the coalescent model with recombination: for recombination rates r =1, 2, 4, 8, 16, 32, the number of phylogenetic networks of level-k is shown, for simulations on 10 leaves (a) or 50 leaves (b)

However, this characterization is deeply based on a canonical tree decomposition of level-k networks which by lack of space cannot be detailed here, but is illustrated in Figure 6. It enters the framework of graph grammars [7, 10]. Such a canonical representation is a first step towards counting or efficient exhaustive generation of level-k phylogenetic networks, which would extend currently known results on the number of unicyclic networks and galled trees [32].

4 Level-*k* Networks and the Coalescent Model with Recombination

In [2], Arenas *et al* conducted a simulation study to generate a number of realistic phylogenetic networks, according to the coalescent model with

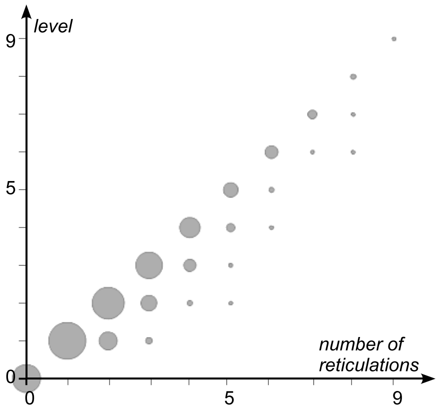


Fig. 8. Number of reticulations and level of the simulated networks for $n = 10$ and $r = 1$. The size of the dot at position (x, y) reflects the number of strict level-x networks with y hybrid vertices.

recombination, and measure the proportion of these networks contained in different subclasses of phylogenetic networks, among which trees and galled trees, i.e. level-0 and level-1 phylogenetic networks. We extend their study by computing the level of a sample of phylogenetic networks generated by the program Recodon [1]. The Java implementation of a simple biconnected component decomposition algorithm to compute the level is also available at `http://www.lirmm.fr/~gambette/ProgGenerators.php`.

For small levels, the results we obtained are shown in Table 1, and an insight on upper levels is given in Fig. 7. We observe that phylogenetic networks with a small level, like restricted phylogenetic networks formerly studied (*regular*, *tree-sibling* and *tree-child*, see [2]), cover a small portion of the networks corresponding to the coalescent model with high recombination rates. Still, the proportion of level-2 phylogenetic networks for 10 leaves is greater than the proportion of tree-child networks, but to get similar proportions on 50 leaves we have to consider level-3 networks.

In fact, our results show that level-k phylogenetic networks do not have a blobbed-tree structure in the context of the coalescent model. Instead, most of the simulated networks have all their hybrid vertices inside one same blob. This phenomenon even appears with a small recombination rate, as shown in Fig. 8. Thus, for this context, new structures and algorithmic techniques have to be found.

The coalescent model is not suitable to describe all cases of reticulate evolution. For example, a simple model of horizontal gene transfer based on inserting transfer events according to a Poisson distribution, respecting time constraints was given in [8]. The use of phylogenetic networks of bounded level may be more appropriate for this model, or others [28].

Acknowledgments

We thank the French ANR projects ANR-08-EMER-011-01 (PhylARIANE) and ANR-06-BLAN-0148-01 (GRAAL) for support. We thank Steven Kelk, Leo van Iersel, Katharina Huber and Matthias Mnich for their comments on earlier versions of this paper. We thank Michael Rao for his help on the isomorphism test for level-k generators, and Gabriel Valiente and Miguel Arenas for sharing their simulated data on the coalescent model.

References

1. Arenas, M., Posada, D.: Recodon: Coalescent Simulation of Coding DNA Sequences with Recombination, Migration and Demography. BMC Bioinformatics 8, 458 (2007)
2. Arenas, M., Valiente, G., Posada, D.: Characterization of Reticulate Networks based on the Coalescent with Recombination. Molecular Biology and Evolution 25(12), 2517–2520 (2008)
3. Byrka, J., Gawrychowski, P., Huber, K.T., Kelk, S.: Worst-case Optimal Approximation Algorithms for Maximizing Triplet Consistency within Phylogenetic Networks. To appear in Journal of Discrete Algorithms (2009)
4. Chan, H.-L., Jansson, J., Lam, T.-W., Yiu, S.-M.: Reconstructing an Ultrametric Galled Phylogenetic Network from a Distance Matrix. Journal of Bioinformatics and Computational Biology 4(4), 807–832 (2006)
5. Choy, C., Jansson, J., Sadakane, K., Sung, W.-K.: Computing the Maximum Agreement of Phylogenetic Networks. Theor. Comput. Sci. 335(1), 93–107 (2005)
6. Cardona, G., Rosselló, F., Valiente, G.: Comparison of Tree-Child phylogenetic networks. To appear in IEEE/ACM Trans. on Comp. Biol. and Bioinf. (2009)
7. Engelfriet, J., van Oostrom, V.: Logical Description of Contex-Free Graph Languages. J. Comput. Syst. Sci. 55(3), 489–503 (1997)
8. Galtier, N.: A Model of Horizontal Gene Transfer and the Bacterial Phylogeny Problem. Systematic Biology 56, 633–642 (2007)
9. Gambette, P.: Who is Who in Phylogenetic Networks: Articles, Authors and Programs, `http://www.lirmm.fr/~gambette/PhylogeneticNetworks`
10. Gioan, E., Paul, C.: Split Decomposition and Graph-Labelled Trees: Characterizations and Fully-Dynamic Algorithms for Totally Decomposable Graphs (submitted, 2009)
11. Gusfield, D., Bansal, V.: A Fundamental Decomposition Theory for Phylogenetic Networks and Incompatible Characters. In: Miyano, S., Mesirov, J., Kasif, S., Istrail, S., Pevzner, P.A., Waterman, M. (eds.) RECOMB 2005. LNCS (LNBI), vol. 3500, pp. 217–232. Springer, Heidelberg (2005)
12. Gusfield, D., Eddhu, S., Langley, C.: Efficient Reconstruction of Phylogenetic Networks with Constrained Recombination. In: IEEE Computational Systems Bioinformatics Conference (CSB 2003), pp. 363–374 (2003)
13. Grant, V.: Plant Speciation, pp. 300–320. Columbia University Press (1971)
14. Hallett, M., Lagergren, J.: Efficient Algorithms for Lateral Gene Transfers Problems. In: International Conference on Research in Computational Molecular Biology (RECOMB 2001), pp. 141–148 (2001)
15. Hudson, R.R.: Properties of the Neutral Allele Model with Intragenic Recombination. Theoretical Population Biology 23, 183–201 (1983)

16. Huson, D.H.: Split Networks and Reticulate Networks. In: Gascuel, O., Steel, M. (eds.) Reconstructing Evolution, pp. 247–276. Oxford University Press, Oxford (2007)
17. van Iersel, L., Keijsper, J., Kelk, S., Stougie, L., Hagen, F., Boekhout, T.: Constructing Level-2 Phylogenetic Networks from Triplets. In: Vingron, M., Wong, L. (eds.) RECOMB 2008. LNCS (LNBI), vol. 4955, pp. 450–462. Springer, Heidelberg (2008)
18. van Iersel, L., Kelk, S.: Constructing the Simplest Possible Phylogenetic Network from Triplets. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 472–483. Springer, Heidelberg (2008)
19. van Iersel, L., Kelk, S., Mnich, M.: Uniqueness, Intractability and Exact Algorithms: Reflections on Level-k Phylogenetic Network. To appear in Journal of Bioinformatics and Computational Biology (2009)
20. Jansson, J., Sung, W.-K.: Inferring a Level-1 Phylogenetic Network from a Dense Set of Rooted Triplets. Theor. Comput. Sci. 363(1), 60–68 (2006)
21. Kanj, I.A., Nakhleh, L., Than, C., Xia, G.: Seeing the Trees and Their Branches in the Network is Hard. Theor. Comput. Sci. 401, 153–164 (2008)
22. Kelk, S., `http://homepages.cwi.nl/~kelk/lev3gen/`
23. Kaibel, V., Schwartz, A.: On the Complexity of Polytope Isomorphism Problems. Graphs and Combinatorics 19(2), 215–230 (2003)
24. Linder, C.R., Rieseberg, L.H.: Reconstructing Patterns of Reticulate Evolution in Plants. American Journal of Botany 91(10), 1700–1708 (2004)
25. Luks, E.M.: Isomorphism of Graphs of Bounded Valence Can be Tested in Polynomial Time. Journal of Computer and System Sciences 25(1), 42–65 (1982)
26. MacLeod, D., Charlebois, R.L., Doolittle, W.F., Bapteste, E.: Deduction of Probable Events of Lateral Gene Transfer through Comparison of Phylogenetic Trees by Recursive Consolidation and Rearrangement. BMC Evol. Biol. 5, 27 (2005)
27. Miller, G.L.: Graph Isomorphism, General Remarks. In: ACM Symposium on Theory of Computing (STOC 1977), pp. 143–150 (1977)
28. Morin, M.M., Moret, B.M.E.: NetGen: Generating Phylogenetic Networks with Diploid Hybrids. Bioinformatics 22(15), 1921–1923 (2006)
29. Ma, B., Wang, L., Li, M.: Fixed Topology Alignment with Recombination. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 174–188. Springer, Heidelberg (1998)
30. Nakhleh, L., Warnow, T., Linder, C.R., St. John, K.: Reconstructing Reticulate Evolution in Species - Theory and Practice. Journal of Computational Biology 12(6), 796–811 (2005)
31. Sloane, N.J.A.: The On-Line Encyclopedia of Integer Sequences. Published electronically, `http://www.research.att.com/~njas/sequences/`
32. Semple, C., Steel, M.: Unicyclic Networks: Compatibility and Enumeration. IEEE/ACM Trans. on Comp. Biol. and Bioinf. 3, 398–401 (2004)
33. To, T.-H., Habib, M.: Level-k Phylogenetic Network Are Constructable from a Dense Triplet Set in Polynomial Time. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 275–288. Springer, Heidelberg (2009)
34. Wang, L., Zhang, K., Zhang, L.: Perfect phylogenetic networks with recombination. In: ACM Symposium on Applied Computing (SAC 2001), pp. 46–50 (2001)
35. Zemlyachenko, V.N., Korneenko, N.M., Tyshkevich, R.I.: Graph Isomorphism Problem. Journal of Mathematical Sciences 29(4), 1426–1481 (1985)

Finding All Sorting Tandem Duplication Random Loss Operations

Matthias Bernt[1], Ming-Chiang Chen[2], Daniel Merkle[3,⋆], Hung-Lung Wang[2], Kun-Mao Chao[2], and Martin Middendorf[1]

[1] Parallel Computing and Complex Systems Group, Department of Computer Science, University of Leipzig, Germany
{bernt,middendorf}@informatik.uni-leipzig.de
[2] Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
{d93003,r92085,kmchao}@csie.ntu.edu.tw
[3] Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
daniel@imada.sdu.dk

Abstract. A tandem duplication random loss (TDRL) operation duplicates a contiguous segment of genes, followed by the loss of one copy of each of the duplicated genes. Although the importance of this operation is founded by several recent biological studies, it has been investigated only rarely from a theoretical point of view. Of particular interest are sorting TDRLs which are TDRLs that, when applied to a permutation representing a genome, reduce the distance towards another given permutation. The identification of sorting genome rearrangement operations in general is a key ingredient of many algorithms for reconstructing the evolutionary history of a set of species. In this paper we present methods to compute all sorting TDRLs for two given gene orders. In addition, a closed formula for the number of sorting TDRLs is derived and further properties of sorting TDRLs are investigated. It is also shown that the theoretical findings are useful for identifying unique sorting TDRL scenarios for mitochondrial gene orders.

1 Introduction

Genomic rearrangement operations are useful to infer the phylogenetic relationship of gene orders representing species. Especially the mitochondrial gene orders became a fruitful source for such investigations, as the number of genes on mitochondrial genomes is small and for more than 1000 species the mitochondrial

⋆ This work was funded within the DAAD/NSC Joint Research Collaboration programme by grant D/05/06868 from the German Academic Exchange Service (DAAD) and NSC grants 95-2221-E-002-126-MY3 and 97-2221-E-002-097-MY3 from the National Science Council, Taiwan. This work was supported by the German Research Foundation (DFG) through the project "Deep Metazoan Phylogeny" within SPP 1174.

gene order is known. The most often studied rearrangement operations are inversions and transpositions [2,9]. Especially from a theoretical point of view these operations have been investigated thoroughly. In recent biological studies it has been shown that the so called tandem duplication random loss (TDRL) operation is a genomic rearrangement operation that can be found several times in the mitochondrial gene order evolution, e.g., in millipedes [11] and eels [10]. A TDRL duplicates a contiguous segment of genes, followed by the loss of one copy of each of the duplicated genes. This operation is considered by some authors as "being the most important rearrangement operation in vertebrate" mitochondrial genomes ([12] and furthermore [4,10]). Although there are some studies that use rearrangement models which are based on inversions, translocations, chromosome fissions and chromosome fusions, that also try to include gene duplications and losses (e.g. [8,15,16]), the properties of TDRLs have only rarely been investigated.

In [7] TDRL operations were studied formally for the first time and a distance measure between gene orders that is based on TDRLs has been defined. A radix sort inspired algorithm to compute a sequence of TDRLs between gene orders that realizes the minimum distance has been presented. Furthermore the asymmetry of TDRLs and the corresponding distance measure was pointed out. Additionally the TDRL median problem, i.e. finding a gene order which has a minimum TDRL distance to two input gene orders, was studied. In [5] a variant considering TDRLs of limited size is studied.

For phylogenetic inference it is important to identify so-called sorting genomic rearrangement operations, i.e., operations that reduce the distance towards a given gene order, when applied to another gene order. One reason is that rearrangement scenarios for two gene orders with a minimum number of operations satisfy the minimum parsimony principle and might therefore be considered more likely than non sorting operations. Hence, in order to find a realistic scenario for describing the rearrangement relation between two gene orders it is helpful to know all possible sorting operations. The use of all equally good solutions in algorithms has often a positive effect on the solution quality, as already shown in the context of genome rearrangements [3]. In this case it is possible to select a sorting operation that satisfies certain properties, e.g. smallest number of involved genes. Note that such problems have been investigated for example in [14,1,6] in the context of inversions. But to the best of our knowledge for TDRLs no method for computing all sorting TDRLs between two gene orders has been published.

Here we present methods to enumerate all sorting TDRLs for two given gene orders, derive closed formulas to compute the number of sorting TDRLs, and derive some theoretical properties of sorting TDRLs. For the example of mitochondrial gene order analysis we show the relevance of our results[1].

[1] The proofs of all lemmas, propositions, and theorems are given in the Appendix. Due to space limitations, the Appendix can not be included in the proceedings and is given for reviewing purposes only.

2 Basic Definitions

A *permutation of size* n is a permutation of the elements in $\{1, 2, \ldots, n\}$. π_i is the element of π at the i-th position. The *inverse permutation* π^{-1} is defined such that π_e^{-1} is the index of element e in π, i.e. $\pi_e^{-1} = i$ iff $\pi_i = e$. An *interval* of a permutation π is a set of consecutive elements of the permutation π.

A *binary string* s is a string over an alphabet Σ of size two. A binary string is defined by specifying the elements for the positions. A pair of consecutive elements (s_i, s_{i+1}) is called a *transition* at position i if $s_i \neq s_{i+1}$. A transition is an *xy-transition* with $x, y \in \Sigma$ if $s_i = x$ and $s_{i+1} = y$.

A *tandem duplication random loss* (TDRL) rearrangement τ transforms a permutation (genome) by a tandem duplication of an interval of the permutation and subsequent random loss of one of the copies of the duplicated elements (genes). A TDRL is regarded as an atomic operation, i.e., the tandem duplication and the random loss of the copied genes are not separable. The set of elements of the permutations before and after the application of a TDRL are equal. Formally, a TDRL applied to a permutation π is defined as $\tau(F, S)$, where F specifies the set of elements which are kept in the first copy and S defines the set of elements kept in the second copy. If the context is clear we simply write τ for $\tau(F, S)$. The following two conditions must hold for a TDRL $\tau(F, S)$: i) $F \cup S$ is an interval in π, and ii) $F \cap S = \emptyset$. Let for example $\pi = (3\ 7\ 1\ 5\ 8\ 2\ 6\ 4)$ be a permutation and let $\tau(\{3, 7, 5, 2\}, \{1, 8, 6, 4\})$ be a TDRL applied to π, then $\pi \circ \tau(F, S) = (3\ 7\ 5\ 2\ 1\ 8\ 6\ 4)$.

W.l.o.g. we assume in this paper $F \cup S = \{1, \ldots, n\}$. TDRLs duplicating only a subset of the genes can obviously be represented by a TDRL duplicating all genes. Two TDRLs leading to the same permutation are considered to be identical. To identify if an element is kept in the first or in the second copy, we define a function $C_\tau : \{1, \ldots, n\} \mapsto \{1, 2\}$ with $C_\tau(e) = 1$ if $e \in F$ and $C_\tau(e) = 2$ for $e \in S$. Using the definition of TDRLs we can immediately define a distance measure as follows. Let π_1 and π_2 be two permutations. The minimum number of TDRLs needed to transform π_1 into π_2 is called the *(TDRL)-distance* of π_1 and π_2, denoted as $d(\pi_1, \pi_2)$. The set $\mathcal{S}(\pi_1, \pi_2)$ of all sorting TDRLs is the set of all TDRLs for which the distance from π_1 towards π_2 is decreased, i.e. $\mathcal{S}(\pi_1, \pi_2) := \{\tau | d(\pi_1 \circ \tau, \pi_2) < d(\pi_1, \pi_2)\}$.

3 TDRLs and Chains

In this section the notion of chains of elements in a permutation is introduced. Chains serve as the key ingredient to compute all sorting TDRLs in this paper.

3.1 Definition and Computation of Chains

The sorting problem is the problem of transforming a permutation π_1 into a permutation π_2 with as few rearrangement operations as possible. W.l.o.g. the sorting problem is typically considered for π_2 being the identity permutation

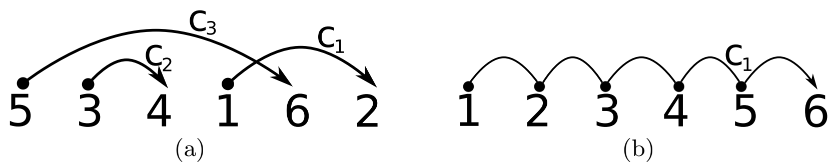


Fig. 1. a) The 3 chains of permutation $\pi = (5\ 3\ 4\ 1\ 6\ 2)$: $c_1 = (1, 2)$, $c_2 = (3, 4)$, and $c_3 = (5, 6)$; b) chain for the identity permutation

denoted by ι (an exception is [7] where $\pi_1 = \iota$ was used). With the notion of chains of elements in a permutation the sorting by TDRLs can be described as transforming a given permutation into the identity permutation.

Definition 1. *A* chain *of a permutation π is a list $(e_1, \ldots, e_k)$ of elements of π with maximal length, where either $k = 1$ or $e_i + 1 = e_{i+1}$ and $\pi^{-1}_{e_i} < \pi^{-1}_{e_{i+1}}$ holds for all $i \in [1 : k - 1]$, $k > 1$.*

A chain connects an element e with element $e + 1$ iff $e + 1$ is located to the right of e in π. Obviously, each element of π belongs to exactly one chain. An example of how a permutation is divided into its chains is depicted in Figure 1(a). We say *a chain is included* in a set of elements (e.g., in a set F of elements kept in the first copy of a TDRL), if all elements of the chain are included in this set of elements.

Let $\rho(\pi)$ be the number of chains of permutation π. We define an indexing scheme for the chains of a permutation in a straightforward manner: Let c and c' be two chains of a permutation π. We define a strict total order for chains as follows: $c < c'$ iff $\forall e \in c, \forall e' \in c' : e < e'$. Let $c_1 < \ldots < c_{\rho(\pi)}$ be the total order of all chains of permutation π. Then c_i is said to be the i-th chain of π. Furthermore we define a function $C_\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, \rho(\pi)\}$ for which the index of the chain can be determined for an element in π, i.e. $C_\pi(e) = i$ iff e is an element of chain c_i. The straightforward algorithm for computing the chains of a permutation has the overall runtime $O(n)$.

3.2 Basic Properties of Chains

As noted above, we can assume w.l.o.g. that sorting is done towards the identity permutation ι. Then sorting by TDRLs corresponds to merging of chains in order to obtain a permutation with only one chain, which is the identity permutation (see Figure 1(b)). Note that the identity permutation is the only permutation which has only one chain. A TDRL moves the elements of the first copy to the left and the elements of the second copy to the right, such that the order of elements kept in the same copy is not changed. Formally, this is stated in the following proposition. The very easy proof is omitted.

Proposition 1. *Let $\tau(F, S)$ be a TDRL, let π be a permutation, and let e_1 and e_2 be two elements of π.*

- *If* $e_1 \in F$ *and* $e_2 \in S$*, then* $(\pi \circ \tau)^{-1}_{e_1} < (\pi \circ \tau)^{-1}_{e_2}$.
- *If* $C_\tau(e_1) = C_\tau(e_2)$ *and* $\pi^{-1}_{e_1} < \pi^{-1}_{e_2}$*, then* $(\pi \circ \tau)^{-1}_{e_1} < (\pi \circ \tau)^{-1}_{e_2}$.

Changes in the order of the elements of a permutation due to a TDRL have implications on the chains, some chains might be split whereas others might be merged. This is formally described in the following two propositions in more detail.

Proposition 2. *Let* π *be a permutation and* e_1 *and* e_2 *be two successive elements (i.e.* $e_2 = e_1 + 1$*) in the same chain* c *of* π*. Let* $\tau(F, S)$ *be a TDRL applied to* π*. Chain* c *is split into a chain ending with* e_1 *and another starting with* e_2 *in* $\pi \circ \tau$ *iff* $e_1 \in S$ *and* $e_2 \in F$*.*

Proposition 3. *Let* π *be a permutation and* e_1 *and* e_2 *be two successive elements (i.e.* $e_2 = e_1 + 1$*) of* π *which are in different chains. Let* $\tau(F, S)$ *be a TDRL applied to* π*. Elements* e_1 *and* e_2 *are in the same chain in* $\pi \circ \tau$ *iff* $e_1 \in F$ *and* $e_2 \in S$*.*

Summarizing propositions 2 and 3 a single TDRL operation can merge chains and split others at the same time depending on which elements are selected in the first copy and which are kept in the second copy.

3.3 TDRL Distance

In [7] it was shown that the TDRL distance $d(\iota, \pi) = \lceil \log_2(\varrho(\pi)) \rceil$, with $\varrho(\pi)$ being the number of maximal increasing substrings in π, i.e., a substring of maximal length with increasing consecutive elements (In [7] a permutation π is seen as a string with elements of the permutation as letters). The following proposition clarifies the relation of maximal increasing substrings and chains.

Proposition 4. *Let* π *be a permutation of length* n*.* $s = (\pi_i \ \pi_{i+1} \ \dots \ \pi_k)$ *is a maximal increasing substring of* π *iff* $c = (i, i+1, \dots, k)$ *is a chain in* π^{-1}*.*

By simply renaming all elements in π it is clear that $d(\iota, \pi) = d(\pi^{-1}, \iota)$. Using the fact that there is a one to one correspondence between maximal increasing substrings in π and chains in π^{-1} as shown in Proposition 4 it follows that the TDRL distance can be computed by $d(\pi, \iota) = \lceil \log_2(\rho(\pi)) \rceil$, where $\rho(\pi)$ is the number of chains in π. Note that the TDRL distance is not symmetric.

4 All Sorting TDRLs

In this section we show how to compute all sorting TDRLs. First, we discuss a restricted case where only TDRLs are considered for which any existing chain in a permutation is completely included either in the first or in the second copy of the TDRL. Such TDRLs will be referred to as *restricted TDRLs*. Interestingly, it can be shown that the number of restricted TDRLs which is needed to sort a permutation is the same as in the general case, i.e., the minimum distance of two permutations is not changed by the restriction. Nevertheless, the set of sorting TDRLs is usually smaller for the restricted case. Note that the sorting TDRL scenarios infered by the algorithm of [7] consist of resticted TDRLs only.

4.1 The Restricted Case

The following two propositions are used to show that the distance of two permutations is unchanged when only restricted TDRLs are allowed.

Proposition 5. *Let $\tau(F,S)$ be a TDRL applied to π and let c_i and c_j be chains of π. $\tau(F,S)$ merges c_i and c_j iff $i+1=j$ and $c_i \in F$ and $c_j \in S$.*

While the Proposition 5 is valid for restricted and unrestricted TDRLs, the following proposition is only valid for restricted TDRLs.

Proposition 6. *Three chains c_i, c_{i+1}, and c_{i+2} can not be merged with one restricted TDRL.*

Using Propositions 5 and 6 the following theorem shows that the restricted TDRL distance is identical to the general TDRL distance.

Theorem 1. *The restricted TDRL distance $d(\pi,\iota) = \lceil \log_2(\rho(\pi)) \rceil$ for a permutation π.*

A sorting restricted TDRL has to reduce the distance by one. Thus the number of chains $\rho(\pi)$ has to be reduced at least to the next smaller value, that is a power of 2. Formally, the number of chains has to be reduced at least by $\rho(\pi) - 2^{\lceil \log_2(\rho(\pi)) \rceil - 1}$. The maximal reduction of the number of chains is $\lfloor \frac{\rho(\pi)}{2} \rfloor$, as chains can be merged only pairwise.

The problem of computing the number of sorting restricted TDRL for a permutation π (toward ι) can be rephrased as a combinatorial problem of binary strings over the alphabet $\Sigma = \{1,2\}$. Let $\tau(F,S)$ be a restricted TDRL for a permutation π. Let $t = t_1 \dots t_{\rho(\pi)}$ be a binary string of length $\rho(\pi)$ with $t_i = 1$ if $c_i \in F$ and $t_i = 2$ if $c_i \in S$. That is, the string t corresponds to a TDRL, such that t_i indicates if the chain c_i is kept in the first or in the second copy of τ. The number of sorting restricted TDRLs for a permutation π corresponds to the number of binary strings for which the number of 12-transitions is at least k with $k = \rho(\pi) - 2^{\lceil \log_2(\rho(\pi)) \rceil - 1}$. This is because the number of 12-transitions in t is identical to the chain decrementation due to τ. The following lemma is needed to derive the number of sorting restricted TDRL.

Lemma 1. *The number of binary strings $t = t_1 \dots t_n$ of length n over alphabet $\Sigma = \{1,2\}$ which have k 12-transitions is*

$$\binom{n+1}{2k+1}$$

A restricted TDRL τ for a permutation π is sorting when the reduction of the number of chains due to τ is between $\rho(\pi) - 2^{\lceil \log_2(\rho(\pi)) \rceil - 1}$ and $\lfloor \frac{\rho(\pi)}{2} \rfloor$. In Corollary 1 the number of sorting restricted TDRL is given.

Corollary 1. *For a permutation π with $\rho(\pi)$ chains there are*

$$\sum_{i=\rho(\pi)-2^{\lceil \log_2(\rho(\pi)) \rceil -1}}^{\lfloor \frac{\rho(\pi)}{2} \rfloor} \binom{\rho(\pi)+1}{2i+1} = \sum_{i=0}^{2^{\lceil \log_2(\rho(\pi)) \rceil} - \rho(\pi)} \binom{\rho(\pi)}{i}$$

sorting restricted TDRLs.

Corollary 1 shows that the number of sorting TDRLs corresponds to a prefix sum in the $\rho(\pi)$-th row of the Pascal's triangle. Note that this sum does only depend on the number of chains and not on the length of the chains, their position in the permutation, or the length of the permutation. An algorithm for enumerating all sorting restricted TDRL can be obtained by enumerating all strings t with a number of 12-transitions as given in the sum of Corollary 1 and inferring the corresponding TDRLs.

Interestingly, if the number of chains $\rho(\pi)$ is equal to a power of 2, then there exist only one sorting restricted TDRL. Also there is only one possible sorting sequence towards ι, as after applying the TDRL the number of chains is halved and is therefore still a power of 2. If the number of chains is $\rho(\pi) = 2^k - 1$ for some k, then there exist several sorting restricted TDRLs, but the subsequent TDRLs towards ι are predetermined as $\lceil (2^k - 1)/2 \rceil = 2^{k-1}$. Assume that a variable is initialized by an integer $\rho > 1$ and let an operation reduce the value of the variable from ρ to ρ' with $\lceil \rho/2 \rceil \leq \rho' \leq 2^{\log_2(\rho)-1}$ (Observe, this is what a TDRL does with the number of existing chains). Then the operation has to be repeated $\lceil \log_2(2^{\lceil \log_2(\rho) \rceil} - \rho + 1) \rceil$ times, such that the resulting number is a power of 2. The easy but technical proof for this is omitted. The following Corollary follows.

Corollary 2. *Let π be a permutation with $\rho(\pi)$ chains. After the application of $\lceil \log_2(2^{\lceil \log_2(\rho(\pi)) \rceil} - \rho(\pi) + 1) \rceil$ sorting restricted TDRLs, the number of remaining chains is a power of 2, and therefore the remaining sorting sequence towards ι has no alternatives.*

4.2 The General Case

For the general case of computing all sorting TDRLs the problem is formulated as a problem of finding binary strings with certain properties and some definitions are needed. Let π be a permutation of length n. Let $\tau(F, S)$ be a TDRL applied to π. Let $p = p_1 \dots p_{n-1}$ be a binary string of length $n-1$ over the alphabet $\{\phi, \theta\}$ defined by the chains of π as follows. If $C_\pi(e) = C_\pi(e+1)$ (i.e., if element e and $e+1$ are in the same chain), then $p_e = \theta$ otherwise $p_e = \phi$, $e \in [1 : n-1]$. String p is called *transition string* of π. Let $u = u_1 \dots u_n$ be a binary string of length n over the alphabet $\{1, 2\}$ defined by a TDRL $\tau(F, S)$ as follows. If $e \in F$ then $u_e = 1$, otherwise $u_e = 2$, i.e., the string u corresponds to a TDRL, such that u_e indicates if the corresponding element e of π is kept in the first or in the second copy of τ.

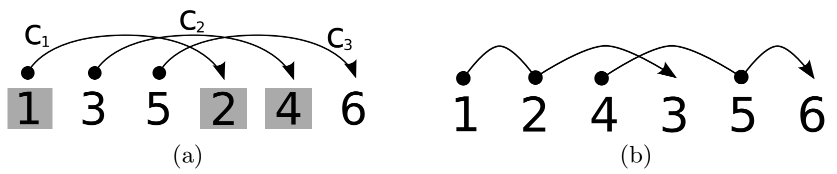


Fig. 2. a) Permutation π having three chains c_1, c_2, c_3; the corresponding string $p = \theta\phi\theta\phi\theta$, e.g. $p_3 = \theta$ as $C_\pi(3) = C_\pi(4) = 2$; one of the sorting TDRLs is $\tau(\{1,2,4\},\{3,5,6\})$ where the elements from F are boxed and elements from S not; the corresponding $u = 112122$; b) Permutation $\pi \circ \tau$: elements 2 and 3 are connected due to τ, as $u_2 = 1$, $u_3 = 2$, and $p_2 = \phi$ induce a 12_ϕ-transition. Furthermore τ connects elements 4 with 5 and destroys chain $c_2 = (3,4)$.

The effects of a TDRL $\tau(F,S)$ on the chains of a permutation π can be defined by the corresponding strings u and p. If $u_e = 2$, $u_{e+1} = 1$, and $p_e = \theta$ (denoted as a 21_θ-transition), then the chain with index $C_\pi(e)$ is split after element e and the number of chains is increased by one. If $u_e = 1$, $u_{e+1} = 2$, and $p_e = \phi$ (12_ϕ-transition) then the number of chains is decreased by one as the element e gets connected to element $e+1$. See Figure 2 for an illustration.

Let Φ be the number of 12_ϕ-transitions in u, and let Θ be the number of 21_θ-transitions in u. Applying a TDRL τ (that corresponds to a binary string u) to a permutation π with transition string p, reduces the number of chains by Φ and increases it by Θ. Therefore the overall reduction in the number of chains due to τ is $k = \Phi - \Theta$. The computation of the number of sorting TDRLs for a given permutation π is equivalent to the computation of the number of strings u (corresponding to a TDRL τ) such that k is between $\rho(\pi) - 2^{\lceil \log_2(\rho(\pi)) \rceil - 1}$ and $\lfloor \frac{\rho(\pi)}{2} \rfloor$.

Let a transition string p be given. Let a *k-sorting string* be a string u, for which the number of 12_ϕ-transitions is Φ, the number of 21_θ-transitions is Θ, and $k = \Phi - \Theta$. The number of TDRLs for a permutation π, that reduce the number of chains by k, is equivalent to the number of k-sorting strings u with p being the transition string of π.

For the computation of the number of sorting TDRLs we apply a dynamic programming scheme as follows. Let $p = p_1 \dots p_{n-1}$ be the transition string for a permutation π of length n. Let $a^x_{j,k}$ be the number of k-sorting strings u of length j ending with $x \in \{1,2\}$. All values of the dynamic programming matrix are initialized with 0 except for $a^1_{1,0} = 1$ and $a^2_{1,0} = 1$. With the following recursion $a^x_{n,k}$ can be computed.

$$a^1_{j+1,k} = \begin{cases} a^1_{j,k} + a^2_{j,k+1} & \text{if } p_j = \theta \\ a^1_{j,k} + a^2_{j,k} & \text{else} \end{cases} \quad (1) \qquad a^2_{j+1,k} = \begin{cases} a^1_{j,k-1} + a^2_{j,k} & \text{if } p_j = \phi \\ a^1_{j,k} + a^2_{j,k} & \text{else} \end{cases} \quad (2)$$

Note that $a^x_{j,k}$ should be interpreted as the number of sorting TDRLs only if $j = n$ because strings u of size $j < n$ do not define valid TDRLs. An illustration of

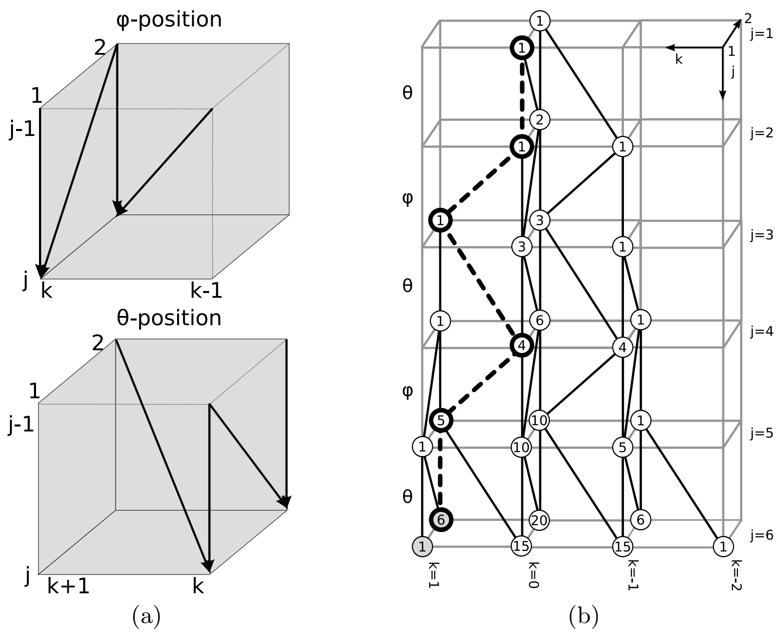


Fig. 3. a) Illustration of the recursion in the dynamic programming matrix for $p_j = \phi$ (top) and $p_{j-1} = \theta$ (bottom); arrows indicate which values are used to compute the sums of the recursion; b) the dynamic programming matrix for the example in Figure 2; values in circles correspond to $a^x_{j,k}$; the sum of all values in grey filled circles corresponds to the number of all sorting TDRLs; the dashed path corresponds to the TDRL $\tau(\{1,2,4\},\{3,5,6\})$.

the recursion for the dynamic programming is given in Figure 3(a). The reduction of the number of chains due to τ has to be between $\rho(\pi) - 2^{\lceil \log_2(\rho(\pi)) \rceil - 1}$ and $\lfloor \frac{\rho(\pi)}{2} \rfloor$ in order to be a sorting TDRL. Thus, the sum

$$\sum_{i=\rho(\pi)-2^{\lceil \log_2(\rho(\pi))\rceil-1}}^{\lfloor \frac{\rho(\pi)}{2} \rfloor} a^1_{n,i} + a^2_{n,i}$$

gives the number of sorting TDRLs. Figure 3(b) shows the dynamic programming matrix for the example as given in Figure 2. Interestingly, this sum can again be reduced to a closed formula, where the number of sorting TDRLs corresponds to the prefix sum of the Pascal's triangle. This is formalized in the following Theorem.

Theorem 2. *For a permutation π of length n with $\rho(\pi)$ chains the number of sorting TDRLs is*

$$\sum_{i=0}^{2^{\lceil \log_2(\rho(\pi))\rceil}-\rho(\pi)} \binom{n}{i}$$

Similar as for sorting restricted TDRLs the number of sorting TDRLs is neither dependent on the length of the chains nor on their position in the permutation. But in contrast to the case of sorting restricted TDRLs the number depends on the length of the input permutation. In the case of $\rho(\pi)$ being a power of 2, there exists only one possible sorting sequence for π towards ι. Similar as in the restricted case an algorithm for enumerating all sorting TDRLs can be inferred directly by an enumeration of the corresponding strings u.

5 Experiments

In this section we show the relevance of our theoretical findings for the analysis of mitochondrial gene orders. First, we investigate the number of sorting TDRLs for the restricted and the general case. Then, we apply random TDRLs to the identity permutation and present the number of resulting chains in order to estimate how likely certain numbers of chains are. The result is then used to support scenarios of a sequence of TDRLs. Such a very likely scenario is presented for mitogenomes.

In Table 1 the number of sorting TDRLs is given for permutations of length n having $|\rho(\pi)| \in \{2, \ldots, 16\}$ chains. Note that it is not relevant if the permutations

Table 1. Number of sorting TDRLs for different number of chains $\rho(\pi) \in \{2, \ldots, 16\}$; $|\mathcal{S}^r|$: restricted case; $|\mathcal{S}^g_n|$: general case with permutation length $n \in \{10, 37\}$

$\rho(\pi)$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\lvert\mathcal{S}^r\rvert$	1	4	1	26	22	8	1	502	848	1,024	794	378	106	16	1
$\lvert\mathcal{S}^g_{10}\rvert$	1	38	1	8474	704	38	1	13,130,672	2,835,200						
$\lvert\mathcal{S}^g_{37}\rvert$	1	101	1	166,751	5,051	101	1	814,947,389	555,600,013	79,375,496	4,087,976	166,751	5,051	101	1

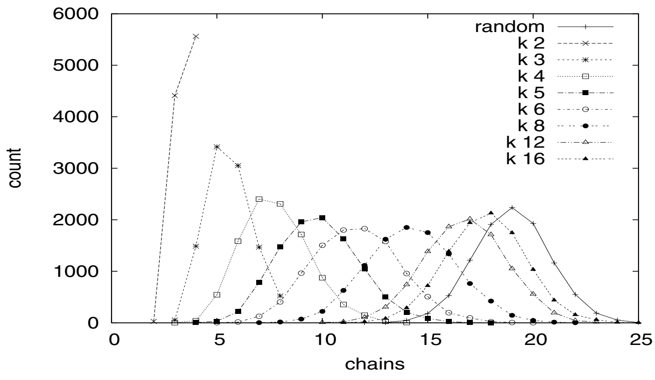


Fig. 4. Number of chains of 10,000 permutations Π_k for each $k \in \{2, 3, 4, 5, 6, 8, 12, 16\}$; k is the number of TDRLs applied to the identity permutation to obtain a permutation $\pi \in \Pi_k$; for the line denoted with random each π is a random permutation

are mitogenomes or random permutations, as only the number of chains and the length of the permutation are relevant. Let $\mathcal{S}^r$ be the number of sorting TDRLs in the restricted case and let $\mathcal{S}^g_n$ be the number of sorting TDRLs in the general case. Recall that the number of sorting TDRLs is dependent on n only in the general case. For the general case we analyze the cases $n = 10$ and $n = 37$, where the latter is the length of mitochondrial gene orders. While for some values of $|\rho(\pi)|$ the number of TDRLs is immense (e.g., more than 800 millions in the general case with $n = 37$ and $|\rho(\pi)| = 9$) it is 1 if the number of chains is a power of 2. This fact can be used to identify sequences of TDRLs for which no alternative sorting scenario exist. Figure 4 shows the results for the case that $k \in \{2, 3, 4, 5, 6, 8, 12, 16\}$ TDRLs have been applied to the identity permutation of length 37 resulting in permutation π. For each k this has been repeated 10,000 times resulting in a set of permutations Π_k. The histograms for the number of resulting chains for Π_k are depicted in Figure 4. Additionally the case when π is chosen to be a random permutation is shown. If, for example, the number of chains for a pair of permutations of length 37 is 4, it is very likely that this is due to 2 unique TDRL operations.

Based on these results all pairs of existing complete mitochondrial gene orders from the NCBI database have been analyzed and sequences of TDRLs have been identified that can be considered as biologically likely. An example which shows the potential of our results is given in Figure 5 for species *Salvelinus fontinalis* (which has the typical vertebrate gene order) and *Porichthys myriaster*. The TDRL-scenario is supported by the following observations. Scenarios considering other rearrangements are much longer, e.g., the reversal distance is 15 and the transposition distance is at least 7 (computed with the lower bound given in [2]). The given scenario is the only parsimonious sorting scenario based on TDRLs. That is i) the TDRL scenario in the given direction is unique, ii) the TDRL distance in the other direction of length 3 is not parsimonious, and iii) there exists no ancestral permutation of *S. fontinalis* and *P. myriaster*, such that the two genomes can be reached with two or less TDRLs (this was checked by a computationally expensive brute force algorithm). Due to the histograms as presented in Figure 4, it is clear that the 4 chains indeed occurred very likely due to 2 TDRLs. Furthermore, according to [13] duplications and losses are supported by fragments of nucleotide sequences in the mitogenomes. Summarizing, there is very strong support that the *S. fontinalis* gene order is the ancestral gene one of *S. fontinalis* and *P. myriaster*, and there are strong indications that the 2 presented TDRLs have changed the gene order. Note that the genomic

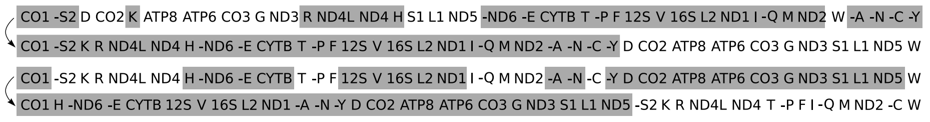

CO1 -S2 D CO2 K ATP8 ATP6 CO3 G ND3 R ND4L ND4 H S1 L1 ND5 -ND6 -E CYTB T -P F 12S V 16S L2 ND1 I -Q M ND2 W -A -N -C -Y
CO1 -S2 K R ND4L ND4 H -ND6 -E CYTB T -P F 12S V 16S L2 ND1 I -Q M ND2 -A -N -C -Y D CO2 ATP8 ATP6 CO3 G ND3 S1 L1 ND5 W
CO1 -S2 K R ND4L ND4 H -ND6 -E CYTB T -P F 12S V 16S L2 ND1 I -Q M ND2 -A -N -C -Y D CO2 ATP8 ATP6 CO3 G ND3 S1 L1 ND5 W
CO1 H -ND6 -E CYTB 12S V 16S L2 ND1 -A -N -Y D CO2 ATP8 ATP6 CO3 G ND3 S1 L1 ND5 -S2 K R ND4L ND4 T -P F I -Q M ND2 -C W

Fig. 5. The unique TDRL scenario from *Salvelinus fontinalis* (top row) to *Porichthys myriaster* (bottom row); genes kept in the first copy are boxed

rearrangement events for the gene orders of this example have not yet been described in the literature. An evaluation of TDRL scenarios for a larger number of mitochondrial genomes will be presented elsewhere.

6 Conclusion

Tandem duplication random loss (TDRL) events are important gene order rearrangement operation especially in mitochondrial gene orders. Methods were presented to derive sorting TDRLs, i.e., TDRLs that cause a permutation to become closer to another given permutation. An interesting restricted case of the problem has been analyzed which leads also to an analysis of the general case. A closed formula for the number of sorting (restricted) TDRLs has been presented. Algorithms for enumerating all sorting TDRLs have been obtained by an enumeration of binary strings with certain properties. We have shown that the theoretical findings are relevant when identifying sequences of TDRLs for real biological data, e.g., mitochondrial gene orders.

References

1. Ajana, Y., Lefebvre, J.-F., Tillier, E.R.M., El-Mabrouk, N.: Exploring the set of all minimal sequences of reversals an application to test the replication-directed reversal hypothesis. In: Guigó, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 300–315. Springer, Heidelberg (2002)
2. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM Journal on Discrete Mathematics 11, 224–240 (1998)
3. Bernt, M., Merkle, D., Middendorf, M.: Using median sets for inferring phylogenetic trees. Bioinformatics 23(2), 129–135 (2007)
4. Boore, J.L.: The duplication/random loss model for gene rearrangement exemplified by mitochondrial genomes of deuterostome animals. In: Comparative genomics. Computational biology series, vol. 1, pp. 133–147. Kluwer, Dordrecht (2000)
5. Bouvel, M., Rossin, D.: A variant of the tandem duplication - random loss model of genome rearrangement. Theoretical Computer Science 410(8-10), 847–858 (2009)
6. Braga, M., Sagot, M.-F., Scornavacca, C., Tannier, E.: Exploring the solution space of sorting by reversals, with experiments and an application to evolution. IEEE/ACM Transactions on Computational Biology and Bioinformatics 5, 348–356 (2008)
7. Chaudhuri, K., Chen, K., Mihaescu, R., Rao, S.: On the tandem duplication-random loss model of genome rearrangement. In: SODA, pp. 564–570 (2006)
8. El-Mabrouk, N.: Genome rearrangement by reversals and insertions/deletions of contiguous segments. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 222–234. Springer, Heidelberg (2000)
9. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. In: ACM Symposium on Theory of Computing, pp. 178–189 (1995)
10. Inoue, J.G., Miya, M., Tsukamoto, K., Nishida, M.: Evolution of the deep-sea gulper eel mitochondrial genomes: Large-scale gene rearrangements originated within the eels. Mol. Biol. Evol. 20, 1917–1924 (2003)

11. Lavrov, D.V., Boore, J.L., Brown, W.M.: Complete mtdna sequences of two millipedes suggest a new model for mitochondrial gene rearrangements: Duplication and nonrandom loss. Mol. Biol. Evol. 19, 163–169 (2002)
12. Mauro, D.S., Gower, D.J., Zardoya, R., Wilkinson, M.: A hotspot of gene order rearrangement by tandem duplication and random loss in the vertebrate mitochondrial genome. Mol. Biol. Evol. 23, 227–234 (2006)
13. Miya, M., Satoh, T.P., Nishida, M.: The phylogenetic position of toadfishes (order batrachoidiformes) in the higher ray-finned fish as inferred from partitioned bayesian analysis of 102 whole mitochondrial genome sequences. Biol. J. Linn. Soc. Lond. 85, 289–306 (2005)
14. Siepel, A.C.: An algorithm to enumerate sorting reversals for signed permutations. Journal of Computational Biology 10(3-4), 575–597 (2003)
15. Swenson, K.M., Marron, M., Earnest-DeYoung, J.V., Moret, B.M.E.: Approximating the true evolutionary distance between two genomes. ACM Journal on Experimental Algorithmics 12(3.5) (2008)
16. Tang, J., Moret, B.M.E., Cui, L., de Pamphilis, C.W.: Phylogenetic reconstruction from arbitrary gene-order data. In: Proc. 4th IEEE Conf. on Bioinformatics and Bioengineering BIBE 2004, pp. 592–599 (2004)

Average-Case Analysis of Perfect Sorting by Reversals

Mathilde Bouvel[1,⋆], Cedric Chauve[2], Marni Mishna[2], and Dominique Rossin[1,⋆]

[1] CNRS, Université Paris Diderot, LIAFA, Paris, France
{mbouvel,rossin}@liafa.jussieu.fr
[2] Department of Mathematics, Simon Fraser University, Burnaby (BC), Canada
{cedric.chauve,marni.mishna}@sfu.ca

Abstract. A sequence of reversals that takes a signed permutation to the identity is perfect if it preserves all common intervals between the permutation and the identity. The problem of computing a parsimonious perfect sequence of reversals is believed to be NP-hard, as the more general problem of sorting a signed permutation by reversals while preserving a given subset of common intervals is NP-hard. The only published algorithms that compute a parsimonious perfect reversals sequence have an exponential time complexity. Here we show that, despite this worst-case analysis, with probability one, sorting can be done in polynomial time. Further, we find asymptotic expressions for the average length and number of reversals in commuting permutations, an interesting sub-class of signed permutations.

1 Introduction

The sorting of signed permutations by reversals is a simple combinatorial problem with a direct application in genome arrangement studies. Different sorting scenarios provide estimates for evolutionary distance and can help explain the differences in gene orders between two species (see [9] for example). Initially, the shortest (parsimonious) sequences of reversals were sought, and polynomial time algorithms to find such sequences were described [14,8,20]. Recently, biologically motivated refinements have been considered, specifically accounting for groups of genes that are co-localized with the different homologous genes (genes having a single common ancestor) in the genomes of different species. It is then likely that such groups of genes were contiguous in the common ancestral genome, and were not disrupted during evolution, hence, we expect them to appear together at every step of the evolution. In terms of our combinatorial model, a group of co-localized genes is modeled by a *common interval*, and the prefectness condition implies that common intervals are preserved by reversals. This constraint leads to the following algorithmic problem:

> What is the smallest number of reversals required to sort a signed permutation into the identity permutation without breaking any common interval?

⋆ Supported by ANR project GAMMA BLAN07-2_195422.

These scenarios are called *perfect* [12]. Because of the additional constraint, it is possible that the shortest perfect sorting scenario is longer that the shortest scenario.

It is known that the refined problem of preserving a given subset of all common intervals is NP-hard [12]. However, several authors have described classes of instances which can be solved in polynomial time [3,4,11], and fixed parameter algorithms exist [4,5]. For example, *commuting permutations* form a class of instances such that the property of a reversal scenario being perfect is preserved even when the sequence of reversals is reordered arbitrarily. A central concept in the theory of perfect sorting by reversals is the "strong interval tree" associated to a permutation [4].

Recently, several works have investigated expected properties of combinatorial objects related to genomic distance computation, such as the breakpoint graph [24,22,23,19]. We explore this route here, but focusing on the strong interval tree, to conduct an average-case analysis of perfect sorting by reversals. First, in Section 3, we prove that for large enough n, with probability 1, computing a perfect reversal sorting scenario for signed permutations can be done in time polynomial in n, despite the fact that this is NP-hard. Secondly, in Section 4, we show that, in a parsimonious perfect scenario for a commuting permutation of length n, the average number of reversals is asymptotically $1.2n$, and the average length of a reversal is $1.02\sqrt{n}$. We conclude by describing future research avenues, both theoretical and applied.

2 Preliminaries

We first summarize the combinatorial and algorithmic frameworks for perfect sorting by reversals. For a more detailed treatment, we refer to [4].

Permutations, reversals, common intervals and perfect scenarios. A *signed permutation* on $[n]$ is a permutation on the set of integers $[n] = \{1, 2, \ldots, n\}$ in which each element has a sign, positive or negative. Negative integers are represented by placing a bar over them. We denote by Id_n (resp. $\overline{Id_n}$) the identity (resp. reversed identity) permutation, $(1\ 2 \ldots n)$ (resp. $(\overline{n} \ldots \overline{2}\ \overline{1})$). When the number n of elements is clear from the context, we will simply write Id or $\overline{Id}$.

An *interval* I of a signed permutation σ on $[n]$ is a segment of adjacent elements of σ. The *content* of I is the subset of $[n]$ defined by the absolute values of the elements of I. Given σ, an interval is defined by its content and from now, when the context is unambiguous, we identify an interval with its content.

The *reversal* of an interval of a signed permutation reverses the order of the elements of the interval, while changing their signs. If σ is a permutation, we denote by $\overline{\sigma}$ the permutation obtained by reversing the complete permutation σ. A *scenario* for σ is a sequence of reversals that transforms σ into Id_n or $\overline{Id_n}$. The *length* of such a scenario is the number of reversals it contains. The length of a reversal is the number of elements in the interval that is reversed.

Two distinct intervals I and J *commute* if their contents trivially intersect, that is either $I \subset J$, or $J \subset I$, or $I \cap J = \emptyset$. If intervals I and J do not commute, they *overlap*. A *common interval* of a permutation σ on $[n]$ is a subset of $[n]$ that is an interval in both σ and the identity permutation Id_n. The singletons and the set $\{1, 2, \ldots, n\}$ are always common intervals called *trivial common intervals*.

A scenario S for σ is called a *perfect scenario* if every reversal of S commutes with every common interval of σ. A perfect scenario of minimal length is called a *parsimonious perfect scenario*.

A permutation σ is said to be *commuting* if there exists a perfect scenario for σ such that for every pair of reversals of this scenario, the corresponding intervals commute. In such a case, this property holds for every perfect scenario for σ [4].

The strong interval tree. A common interval I of a permutation σ is a *strong interval* of σ if it commutes with every other common interval of σ.

The inclusion order of the set of strong intervals defines an n-leaf tree, denoted by $T_{\mathcal{S}}(\sigma)$, whose leaves are the singletons, and whose root is the interval containing all elements of the permutation. The strong interval tree of σ can be computed in linear time and space (see [7] for example). We call the tree $T_{\mathcal{S}}(\sigma)$ the *strong interval tree* of σ, and we identify a vertex of $T_{\mathcal{S}}(\sigma)$ with the strong interval it represents. In a more combinatorial context, this tree is also called *substitution decomposition tree* [1]. If σ is a signed permutation, the sign of every element of σ is given to the corresponding leaf in $T_{\mathcal{S}}(\sigma)$. (See Fig. 1.)

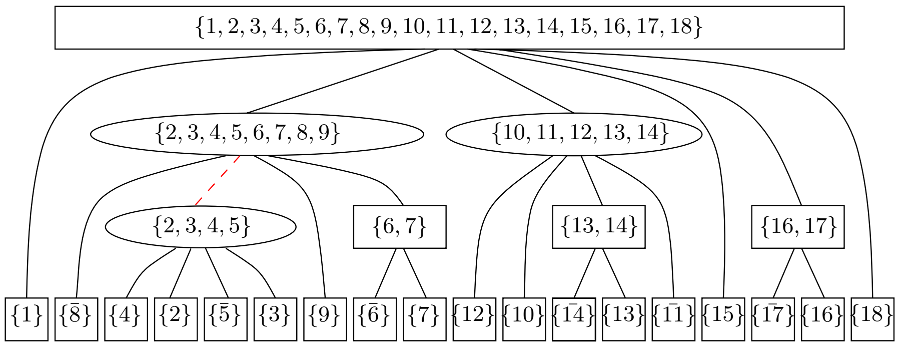


Fig. 1. The strong interval tree $T_{\mathcal{S}}(\sigma)$ of the permutation $\sigma = (1\ \bar{8}\ 4\ 2\ \bar{5}\ 3\ 9\ \bar{6}\ 7\ 12\ 10\ \bar{14}\ 13\ \bar{11}\ 15\ \bar{17}\ 16\ 18)$. Prime and linear vertices are distinguished by their shape. There are three non-trivial linear vertices, the rectangular vertices, and three prime vertices, the round vertices. The root and the vertex $\{6, 7\}$ are increasing linear vertices, while the linear vertices $\{16, 17\}$ and $\{13, 14\}$ are decreasing.

Let I be a strong interval of σ and $\mathcal{I} = (I_1, \ldots, I_k)$ the unique partition of the elements of I into maximal strong intervals, from left to right. The *quotient permutation* of I, denoted σ_I, is the permutation of size k defined as follows: $\sigma_I(i)$ is smaller than $\sigma_I(j)$ in σ_I if any element of I_i is smaller (in absolute value if σ

is a signed permutation) than any element of I_j. The vertex I, or equivalently the strong interval I of σ, is either: *increasing linear*, if σ_I is the identity permutation, or *decreasing linear*, if σ_I is the reversed identity permutation, or *prime*, otherwise. For exposition purposes we consider that an increasing vertex is positive and a decreasing vertex is negative. The strong interval tree as computed in the algorithm of [7] contains the nature – increasing/decreasing linear or prime – of each vertex. It can easily be adapted to compute also in linear time the quotient permutation associated to each strong interval.

For a vertex I of $T_S(\sigma)$, we denote by $L(I)$ the set of elements of σ that label the leaves of the subtree of $T_S(\sigma)$ rooted at I.

The strong interval tree as a guide for perfect sorting by reversals. We describe now important properties, related to the strong interval tree, of the algorithm described in [4] for perfect sorting by reversals a signed permutation. Let σ be a signed permutation of size n and $T_S(\sigma)$ its strong interval tree, having m internal vertices, called $I_1, \ldots, I_m$, including p prime vertices:

Theorem 1. [4]

1. *The algorithm described in [4] can compute a parsimonious perfect scenario for σ in worst-case time $O(2^p n\sqrt{n\log(n)})$.*
2. *σ is a commuting permutation if and only if $p = 0$.*
3. *If σ is a commuting permutation, then every perfect scenario has for reversals set the set $\{L(I_j) | I_j$ has a sign different from its parent in $T_{\mathcal{S}}(\sigma)\}$.*

Remark 1. The strong interval tree of an unsigned permutation is equivalent to the modular decomposition tree of the corresponding labeled permutation graph (see [4] for example). Also commuting permutations have been investigated, in connection with permutation patterns, under the name of *separable* permutations [15].

3 On the Number of Prime Vertices

Motivated by the average-time complexity of the algorithm described in [4] for computing a parsimonious perfect scenario, we first investigate the average shape of a strong interval tree of a permutation of size n. Such a tree is characterized by the shape of the tree along with the quotient permutations labeling internal vertices. For prime vertices, those quotient permutations correspond to *simple permutations* as defined in [2]. We first concentrate on enumerative results on simple permutations. Next, we derive from them enumerative consequences on the number of permutations whose strong interval tree has a given shape. Exhibiting a family of shapes with only one prime vertex, we can prove that nearly all permutations have a strong interval tree of this special shape.

3.1 Combinatorial Preliminaries: Strong Interval Trees and Simple Permutations

Let $T_{\mathcal{S}}(\sigma)$ be the strong interval tree of a permutation σ of length n. From a combinatorial point of view it is simply a plane tree (the children of a vertex are totally ordered) with n leaves and its internal vertices labeled by their quotient permutation: an internal vertex having k children can be labeled either by the permutation $(1\,2\,\ldots\,k)$ (increasing linear vertex), the permutation $(k\,k{-}1\,\ldots\,1)$ (decreasing linear vertex) or a permutation of length k whose only common intervals are trivial (prime vertex). Due to the fact that $T_{\mathcal{S}}(\sigma)$ represents the common intervals between σ and the identity permutation, it has two important properties.

Property 1. *1. No edge can be incident to two increasing or two decreasing linear vertices.*
2. The labeling of the leaves by the integers $\{1,\ldots,n\}$ is implicitly defined by the permutations labeling the internal vertices.

Permutations whose common intervals are trivial are called *simple permutations.* The shortest simple permutations are of length 4 and are $(3\,1\,4\,2)$ and $(2\,4\,1\,3)$. The enumeration of simple permutations was investigated in [2]. The authors prove that this enumerative sequence is not P-recursive and there is no known closed formula for the number of simple permutations of a given size. However, it was shown in [2] that an asymptotic equivalent for the number s_n of simple permutations of size n is

$$s_n = \frac{n!}{e^2}\left(1 - \frac{4}{n} + \frac{2}{n(n-1)} + \mathcal{O}\left(\frac{1}{n^3}\right)\right) \text{ when } n \to \infty. \qquad (1)$$

3.2 Average Shape of Strong Interval Trees

A *twin* in a strong interval tree is a vertex of degree 2 such that each of its two children is a leaf. A twin is then a linear vertex. The following result, that applies both to signed permutations and unsigned permutations, is the main result of this section.

Theorem 2. *Asymptotically, with probability 1, a random permutation σ of size n has a strong interval tree such that the root is a prime vertex and every child of the root is either a leaf or a twin. Moreover the probability that $T_{\mathcal{S}}(\sigma)$ has such a shape with exactly k twins is $\frac{2^k}{e^2 k!}$.*

The proof follows from Lemma 1 and Equation (1).

Lemma 1. *If $p'_{n,k}$ denotes the number of permutations of length n which contain a common interval I of length k then for any fixed positive integer c:*

$$\sum_{k=c+2}^{n-c} \frac{p'_{n,k}}{n!} = O(n^{-c})$$

Proof. The proof is very similar to Lemma 7 in [2]. We have $p'_{n,k} \le (n-k+1)k!(n-k+1)!$. Indeed, the right hand side counts the number of permutations of $\{1\ldots k\}$ corresponding to I $(k!)$, the possible values of the minimal element of I $(n-k+1)$ and the structure of the rest of the permutation with one more element which marks the insertion of I $((n-k+1)!)$. Only the extremal terms of the sum can have magnitude $\mathcal{O}(n^{-c})$ and the remaining terms have magnitude $\mathcal{O}(n^{-c-1})$. Since there are fewer than n terms the result of Lemma 1 follows.

Proof (Proof of Theorem 2). Lemma 1 with $c = 1$ gives that the proportion of non-simple permutations with common intervals of size greater than or equal to 3 is $O(n^{-1})$. But permutations whose common intervals are only of size $1, 2$ or n are exactly permutations whose strong interval tree has a prime root and every child is either a leaf or a twin.

Then the number of permutations whose strong interval tree has a prime root with k twins is $s_{n-k}\binom{n-k}{k}2^k$. From Equation (1) the asymptotics for this number is $\frac{n!2^k}{e^2 k!}$, proving Theorem 2. □

3.3 Average Time Complexity of Perfect Sorting by Reversals

Corollary 1. *The algorithm described in [4] for computing a parsimonious perfect scenario for a random permutation runs in polynomial time with probability* 1 *as* $n \to \infty$.

Proof. Direct consequence of point 1 in Theorem 1 and of Theorem 2, applied on signed permutations. □

This result however does not imply that the average complexity of this algorithm is polynomial, as the average time complexity is the sum of the complexity on all instances of size n divided by the number of instances. Formally, to assess the average time complexity, we need to prove that as n grows, the ratio

$$p_n = \frac{\sum_p 2^p T_{n,p}}{T_n}$$

is bounded by a polynomial in n, where T_n is the number of strong interval trees with n leaves and $T_{n,p}$ the number of such trees with p prime vertices. The factor 2^p comes from the complexity given in Theorem 1.

Let $T(x,y)$ be the bivariate generating function $T(x,y) = \sum_{k,n} T_{n,p}x^n y^p$ Then $p_n = \frac{[x^n]T(x,2)}{[x^n]T(x,1)}$. Let moreover $P(x)$ be the generating function of simple permutations $P(x) = \sum_{n\geq 0} s_n x^n$ (whose first terms can be obtained from entry A111111 in [18]). Using the specification for strong interval trees given in Section 3.1 and techniques described in [13] for example, it is immediate that $T(x,y)$ satisfies the following system of functional equations:

$$\begin{cases} T(x,y) = x + yP(T(x,y)) + 2\frac{B(x,y)^2}{1-B(x,y)} \\ B(x,y) = x + yP(T(x,y)) + \frac{B(x,y)^2}{1-B(x,y)} \end{cases}$$

By iterating these equations, we computed the 25 first values of p_n (Fig. 2) that suggest that p_n is even bounded by a constant close to 2 and lead us to Conjecture 1.

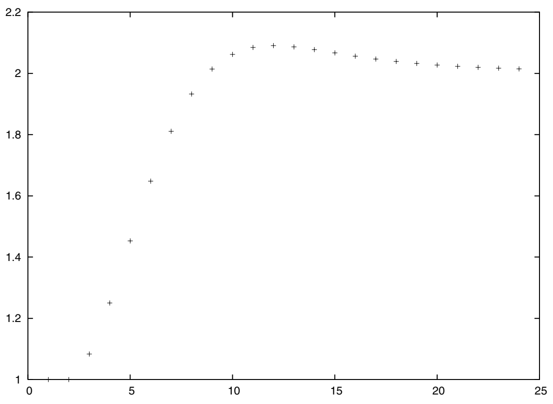


Fig. 2. p_n, up to $n = 25$

Conjecture 1. *The average-time complexity of the algorithm described in [4] for computing a parsimonious perfect scenario is polynomial, bounded by* $\mathcal{O}(n\sqrt{n \log n})$.

4 Average-Case Properties of Commuting Permutations

We now study the family of commuting (signed) permutations and more precisely the average number of reversals in a parsimonious perfect scenario for a commuting permutation and the average length of a reversal of such a scenario. These questions are motivated by two problems. First, from a more theoretical point of view, understanding strong interval trees with no prime node is a first step towards more general results on strong interval trees with few prime nodes, that are common with real data. Second, from an applied point of view, in the strong interval trees computed from real data, it is common to find large genome segments corresponding to a subtree that contains only linear nodes. Hence, the results of this section can be applied to such subtrees to detect genome segments with non-random evolution scenarios. Also the results in this section

Let σ be a commuting permutation of size n, i.e. a signed permutation whose strong interval tree $T_{\mathcal{S}}(\sigma)$ has no prime vertex. It follows from the combinatorial specification of strong interval trees given in Section 3.1 that $T_{\mathcal{S}}(\sigma)$ is simply a plane tree with internal vertices having at least two children and a sign on the root (from Property 1, that defines implicitly the signs of the other internal vertices, and the labels $\{1 \dots n\}$ of the leaves). These trees are then Schröder trees (entry A001003 in the On-Line Encyclopedia of Integer Sequences [18]) with a sign on the root.

Theorem 3. *The average length of a parsimonious perfect scenario for a commuting permutation of length n is asymptotically*

$$\frac{1+\sqrt{2}}{2}n \simeq 1.2n.$$

Proof. We now sketch the main steps of the proof. From the previous section and points 2 and 3 in Theorem 1, the problem of computing the expected number of reversals of a parsimonious perfect scenario reduces to computing the expected number of internal vertices of $T_{\mathcal{S}}(\sigma)$ other than the root (because, from Property 1.1, two adjacent linear vertices cannot have the same sign) and the expected number of leaves whose sign in σ differs from the sign of its parent in $T_{\mathcal{S}}(\sigma)$.

The expected number of leaves whose sign in σ is different from its parent in $T_{\mathcal{S}}(\sigma)$ is obviously $n/2$, as the sign of the leaf and of its parent are independent.

To compute the average number of internal vertices in a Schröder tree, we use symbolic methods as defined in [13]. Let us define the bivariate generating function $S(x,y) = \sum_{k,n} S_{n,k}x^n y^k$ where $S_{n,k}$ denotes the number of Schröder trees with n leaves and k internal vertices. The average number of internal vertices in a Schröder tree with n leaves is

$$\frac{\sum_k kS_{n,k}}{\sum_k S_{n,k}} = \frac{[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1}}{[x^n]S(x,1)}.$$

A Schröder tree can be recursively described as a single leaf, or a root having at least two children, which are again Schröder trees. Consequently, $S(x,y)$ satisfies the equation

$$S(x,y) = x + y\frac{S(x,y)^2}{1-S(x,y)},$$

and solving this equation gives

$$S(x,y) = \frac{(x+1) - \sqrt{(x+1)^2 - 4x(y+1)}}{2(y+1)}. \tag{2}$$

The number $[x^n]S(x,1)$ of Schröder trees ([18, entry A001003]) is asymptotically equivalent to

$$\frac{\sqrt{3\sqrt{2}-4}}{4}(3+2\sqrt{2})^n\frac{1}{\sqrt{\pi n^3}}.$$

From Equation (2) we obtain an equivalent of the coefficients $[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1}$ when $n \to \infty$:

$$[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1} \sim \frac{3-2\sqrt{2}}{4\sqrt{3\sqrt{2}-4}}(3+2\sqrt{2})^n\frac{1}{\sqrt{\pi n}}.$$

An equivalent of the average number of internal vertices in a Schröder tree with n leaves is now easily derived as

$$\frac{[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1}}{[x^n]S(x,1)} \sim \frac{3-2\sqrt{2}}{3\sqrt{2}-4}n \sim \frac{n}{\sqrt{2}}.$$

Combined with the average number $n/2$ of leaves whose sign is different from its parent in $T_{\mathcal{S}}(\sigma)$, and correcting for having counted the root in the internal vertices (by substracting 1, which does not count asymptotically), this leads to Theorem 3. □

Remark 2. It is interesting to note the large representation of reversals of length 1, that composes almost half of the expected reversals. A similar property was observed in [17] on datasets of bacterial genomes.

Theorem 4. *The average length of a reversal in a parsimonious perfect scenario for a commuting permutation of length n is asymptotically*

$$\frac{2^{7/4}\sqrt{3-2\sqrt{2}}}{1+\sqrt{2}}\sqrt{\pi n} \simeq 1.02\sqrt{n}$$

Proof. We want to compute the ratio between the average sum of the lengths of the reversals of a parsimonious perfect scenario for a commuting permutation and the average length of such a scenario. The later was obtained above (Theorem 3), and we concentrate on the former.

A reversal defined by a vertex x of the strong interval tree $T_{\mathcal{S}}(\sigma)$ is of length $L(x)$ (it reverses the segment of the signed permutation that contains the leaves of the subtree rooted at x, see [4]). We first focus on the average value of the sum of the sizes of all subtrees in a Schröder tree. For simplicity in the computation, we will also count the whole tree and the leaves as subtrees (obviously of size 1), which will give the same quantity we want to compute, up to subtracting $3/2 \cdot n$ to the final result. We first define the bivariate generating function (that we call again S, but which is slightly different)

$$S(x, y) = \sum_{k,n} S_{n,k} x^n y^k$$

where $S_{n,k}$ denotes the number of Schröder trees with n leaves and sizes of subtrees (including leaves and the whole tree) that sum to k. The average value of the sum of the sizes of every subtree in a Schröder tree with n leaves is

$$\frac{\sum_k kS_{n,k}}{\sum_k S_{n,k}} = \frac{[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1}}{[x^n]S(x,1)}.$$

$S(x, y)$ satisfies the functional equation

$$S(x, y) = xy + \frac{S(xy, y)^2}{1 - S(xy, y)}. \tag{3}$$

which leads to

$$\frac{\partial S(x,y)}{\partial y}|_{y=1} = \frac{x}{(1-C)^2}, \text{ where } C = \frac{2S(x,1) - S(x,1)^2}{(1-S(x,1))^2}$$

and then to

$$[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1} \sim \frac{3-2\sqrt{2}}{2}(3+2\sqrt{2})^n$$

An equivalent of the average value of the sum of the sizes of all subtrees in a Schröder tree with n leaves is now easily derived as

$$\frac{[x^n]\frac{\partial S(x,y)}{\partial y}|_{y=1}}{[x^n]S(x,1)} \sim 2^{3/4}\sqrt{3-2\sqrt{2}}\,\sqrt{\pi n^3}.$$

The above result does take into account the whole tree and all leaves, that should not be counted but these terms are negligible asymptotically. Hence, the average sum of the lengths of the reversals of a parsimonious perfect scenario for a commuting permutation of size n is asymptotically

$$2^{3/4}\sqrt{3-2\sqrt{2}}\,\sqrt{\pi n^3}.$$

Dividing by the average number of reversals of such a scenario (Theorem 3), we obtain Theorem 4. □

5 Conclusion and Perspectives

We showed that perfect sorting by reversals, although an intractable problem, is very likely to be solved in polynomial time for random signed permutations. This result relies on a study of the shape of a random strong interval tree that shows that asymptotically such trees are mostly composed of a large prime vertex at the root and small subtrees. As the strong interval tree of a permutation is equivalent to the modular decomposition tree of the corresponding labeled permutation graph [4], this result agrees with the general belief that the modular decomposition tree of a random graph has a large prime root. We were also able to give precise asymptotic results for the expected lengths of a parsimonious perfect scenario and of a reversal of such a scenario for random commuting permutations.

Our research leaves open several problems. The most natural theoretical problem is to prove that computing a parsimonious perfect scenario can be done in polynomial time on the average. It would also be interesting to see if our approach can be extended to the perfect rearrangement problem for the Double-Cut-and-Join model that has been introduced recently [6] and has the intriguing property that instances that were hard to solve for reversals can be solved in polynomial time in the DCJ context and conversely. From a more applied point of view, our results can be applied to the general problem of detecting genome segments that evolve under non-random evolutionary pressure, or more generally whose evolution differs from what would be expected for random permutations. This could be done by detecting subtrees of strong interval trees obtained from real data whose properties differ from the properties of random subtrees. It then would be useful to extend them to more general trees, such as trees with small number of prime nodes. Finally, we plan to apply our techniques to find precise properties of random PQ-trees. PQ-trees lie between strong interval trees with no prime nodes and unrestricted strong interval trees. They have been widely used in genomics, for physical mapping [21], comparative genomics [16] and paleogenomics [10], but very little is known about their random properties [21].

References

1. Albert, M.H., Atkinson, M.D.: Simple permutations and pattern restricted permutations. Discrete Math. 300, 1–15 (2005)
2. Albert, M., Atkinson, M., Klazar, M.: The enumeration of simple permutations. J. Integer Seq. 4, 03.4.4 (2003)
3. Bérard, S., Bergeron, A., Chauve, C.: Conservation of combinatorial structures in evolution scenarios. In: Lagergren, J. (ed.) RECOMB-WS 2004. LNCS (LNBI), vol. 3388, pp. 1–14. Springer, Heidelberg (2005)
4. Bérard, S., Bergeron, A., Chauve, C., Paul, C.: Perfect sorting by reversals is not always difficult. IEEE/ACM Trans. Comput. Biol. Bioinform. 4, 4–16 (2007)
5. Bérard, S., Chauve, C., Paul, C.: A more efficient algorithm for perfect sorting by reversals. Inform. Proc. Letters 106, 90–95 (2008)
6. Bérard, S., Chauve, C., Paul, C., Tannier, E.: Perfect DCJ rearrangement. In: Nelson, C.E., Vialette, S. (eds.) RECOMB-CG 2008. LNCS (LNBI), vol. 5267, pp. 158–169. Springer, Heidelberg (2008)
7. Bergeron, A., Chauve, C., de Montgolfier, F., Raffinot, M.: Computing common intervals of k permutations, with applications to modular decomposition of graphs. SIAM J. Discrete Math. 22, 1022–1039 (2008)
8. Bergeron, A., Mixtacki, J., Stoye, J.: The inversion distance problem. In: Mathematics of Evolution and Phylogeny. Oxford University Press, Oxford (2005)
9. Bourque, G., Pevzner, P.: Genome-scale evolution: reconstructing gene orders in the ancestral species. Genome Res. 12, 26–36 (2002)
10. Chauve, C., Tannier, E.: A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian. PLoS Comput. Biol. 4, e1000234 (2008)
11. Diekmann, Y., Sagot, M.-F., Tannier, E.: Evolution under reversals: Parsimony and conservation of common intervals. IEEE/ACM Trans. Comput. Biol. Bioinform. 4, 301–309 (2007)
12. Figeac, M., Varré, J.-S.: Sorting by reversals with common intervals. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 26–37. Springer, Heidelberg (2004)
13. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, Cambridge (2008)
14. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. J. ACM 46, 1–27 (1999)
15. Ibarra, L.: Finding pattern matchings for permutations. Inform. Proc. Letters 61, 293–295 (1997)
16. Landau, G.M., Parida, L., Weimann, O.: Gene proximity analysis across whole genomes via PQ trees. J. Comput. Biol. 12, 1289–1306 (2005)
17. Lefebvre, J.-F., El-Mabrouk, N., Tillier, E.R.M., Sankoff, D.: Detection and validation of single gene inversions. Bioinformatics, i190–i196 (2003)

18. Sloane, N.J.A.: The on-line encyclopedia of integer sequences (2007), Published electronically, http://www.research.att.com/~njas/sequences/
19. Swenson, K., Lin, Y., Rajan, V., Moret, B.: Hurdles hardly have to be heeded. In: Nelson, C.E., Vialette, S. (eds.) RECOMB-CG 2008. LNCS (LNBI), vol. 5267, pp. 241–251. Springer, Heidelberg (2008)
20. Tannier, E., Bergeron, A., Sagot, M.-F.: Advances on sorting by reversals. Discrete Appl. Math. 155, 881–888 (2007)
21. Wilson, D.B., Greenberg, D.S., Phillips, C.A.: Beyond islands: runs in clone-probe matrices. In: RECOMB 1997, pp. 320–329. ACM Press, New York (1997)
22. Xu, W.: The distribution of distances between randomly constructed genomes: Generating function, expectation, variance and limits. J. Bioinform. Comput. Biol. 6, 23–36 (2008)
23. Xu, W., Benoît, A., Sankoff, D.: Poisson adjacency distributions in genome comparison: multichromosomal, circular, signed and unsigned cases. Bioinformatics 24, i146–i152 (2008)
24. Xu, W., Zheng, C., Sankoff, D.: Paths and cycles in breakpoint graph of random multichromosomal genomes. J. Comput. Biol. 14, 423–435 (2007)

Statistical Properties of Factor Oracles

Jérémie Bourdon[1,2] and Irena Rusu[1]

[1] LINA, CNRS UMR 6421 and University of Nantes, France
[2] IRISA INRIA Rennes-Bretagne Atlantique
(Jeremie.Bourdon,Irena.Rusu)@univ-nantes.fr

Abstract. Factor and suffix oracles have been introduced in [1] in order to provide an economic and efficient solution for storing all the factors and suffixes respectively of a given text. Whereas good estimations exist for the size of the factor/suffix oracle in the worst case, no average-case analysis has been done until now. In this paper, we give an estimation of the average size for the factor/suffix oracle of an n-length text when the alphabet size is 2 and under a Bernoulli distribution model with parameter 1/2. To reach this goal, a new oracle is defined, which shares many of the properties of a factor/suffix oracle but is easier to study and provides an upper bound of the average size we are interested in. Our study introduces tools that could be further used in other average-case analysis on factor/suffix oracles, for instance when the alphabet size is arbitrary.

Keywords: indexing structure, average-case analysis, factor recognition, suffix recognition.

1 Introduction

Finding a given pattern inside a given text is a classical problem (the *pattern matching* problem) for which many solutions have been proposed until now. A very important class of solutions relies on the use of indexing structures, *i.e.* data structures that allow to store the text, to have a fast access to it and to quickly execute certain operations on data. Suffix arrays, suffix automata, suffix trees are classical structures which can be implemented in linear time with respect to the text size.

Still, these structures require a too important (although linear) amount of space. Several techniques for reducing the memory space needed by index implementation were developed (see [4] for a survey). *Language approximation* is one of these techniques, and factor/suffix oracles (introduced in [1]) are one way to illustrate it. Whereas suffix arrays, suffix automata and suffix trees owe their efficacity to their perfect accuracy when answering to the question “Is the word w a suffix (or a factor) of the stored text?”, the factor/suffix oracles are only accurate when they provide the negative answer. The language each of them recognizes is larger or equal to the set of factors/suffixes (respectively) of the text, but their size is very small. The words accepted by a factor/suffix oracle which are not factors/suffixes of the stored text will be termed *by-products*.

A simple, space economical and linear on-line algorithm to build oracles is given in [1], together with some applications to pattern matching. Other applications to pattern matching, finding maximal repeats and text compression can be found in [8], [9], [10] and [11]. A linear compression algorithm, improving the previous quadratic algorithms proposed in [2] and [3], to transform a suffix tree into an oracle can be found in [16]. Another algorithm, based on Ukkonen's algorithm to build a suffix tree, is given in [5].

Two ideas come easily out from these applications. On the one hand, oracles should be reasonably envisaged when one has to deal with a text mining problem. On the other hand, evaluating precisely the performances of an application that uses oracles is a hard task, especially in the average case. Although theoretical studies have been performed for the maximum number of transitions [1] and the maximum number of by-products [12] for the oracles of an n-length text, no theoretical study exists in the average case (An experimental study was realized in [12] for the number of by-products). As a consequence, no theoretical average-case running-time or memory space analysis exists for any algorithm based on oracles. Moreover, experimentally supported conjectures are still open. This is the case, for instance, for the conjecture claiming that the BOM pattern matching algorithm presented in [1] is optimal in the average.

In this paper, we estimate the average number of transitions (*i.e.* the average space occupancy) of the factor/suffix oracle of an n-length text, when the alphabet size is 2 and under a Bernoulli distribution model with parameter 1/2. In this way, we answer another one of the questions raised in the seminal paper [1] (and raised again in [5]). The first of these questions, concerning the characterization of the language recognized by the factor/suffix oracle, was answered in [13].

The paper is organized as follows. In Section 2 we define the factor/suffix min-oracle (which is the classical factor/suffix oracle) and present its main properties. In Section 3, the factor/suffix short-oracle is introduced and is briefly compared to the factor/suffix min-oracle. In section 4, we investigate local properties of the min- and short-oracles and deduce probabilistic results, that we use in Section 5 to estimate the average space occupancy of a short-oracle, and thus of a min-oracle. Section 6 is the conclusion.

2 Factor and Suffix (min-)Oracles

Let $w = w_1 w_2 \dots w_n$ be a sequence of length $|w| = n$ on a finite alphabet Σ. Given integers i, j, $1 \leq i, j \leq n$, we denote $w[i \dots j] = w_i w_{i+1} \dots w_j$ and we call this word a *factor* of w (notice that when $j < i$ the resulting factor is by convention the empty word ε). A *suffix* of w is a factor of w one of whose occurrences ends in position n. The i-th suffix of w, denoted $\mathit{Suff}_w(i)$, is the suffix $w[i \dots n]$ and has length $n + 1 - i$. A *prefix* of w is a factor of w one of whose occurrences starts in position 1. The i-th prefix of w, denoted $\mathit{Pref}_w(i)$, is the prefix $w[1 \dots i]$. By convention, the empty word ε is both a suffix and a prefix of w. Say that a suffix of w is *maximal* if it is not identical to w and it is not the prefix of another suffix of w. Say that a suffix of w is *repeated* if it is a

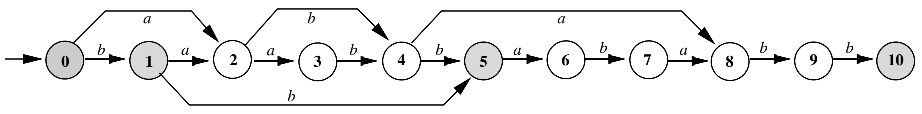


Fig. 1. *The suffix* min*-oracle Omin*(w) *for* $w = baabbababb$*. The final states are grey.*

factor of $w[1 \ldots n-1]$, and *non-repeated* in the contrary case. It is easy to see that a maximal suffix is always a non-repeated suffix, whereas the viceversa is true only for non-repeated *proper* suffixes, *i.e.* distinct from w.

The *factor/suffix oracle* of w is a deterministic automaton which has $n+1$ states denoted $0, 1, 2, \ldots, n$, one *internal transition* $(i, w_{i+1}, i+1)$ for each state i except n, and at most $n-1$ *external transitions* denoted (i, w_j, j), for some pairs i, j with $i+1 < j$. Consequently, the factor/suffix oracle of w is *homogeneous*, that is, all the transitions incoming to a given state have the same label. Each state is final in the factor oracle, while only the states ending the spelling of a suffix of w (including the empty one) are final in the suffix oracle (see Figure 1 for the suffix oracle of $w = baabbababb$).

The factor/suffix oracle was introduced in [1] and can be built using an on-line linear algorithm. The algorithm **Build_Oracle** we give here (also proposed in [1]) is quadratic, but more intuitive. In the algorithm, $Omin(w)$ denotes indifferently the factor or suffix oracle.

Figure 1 shows that the factor/suffix oracle can accept words that are not factors/suffixes, *e.g.* *baabb* which is not a suffix of $w = baabbababb$ but is accepted in the final state 4 of its suffix oracle. These words are called *by-products*.

Algorithm Build_Oracle [1]
Input: Sequence w.
Output: $Omin(w)$.

```
 1. for $i$ from 0 to $n$ do
 2.    create a new state $i$;
 3. for $i$ from 0 to $n-1$ do
 4.    build a new transition from $i$ to $i+1$ by $w_{i+1}$;
 5. for $i$ from 0 to $n-1$ do
 6.    let $x$ be a minimum length word whose reading ends in state $i$;
 7.    for all $\gamma \in \Sigma$, $\gamma \neq w_{i+1}$ do
 8.      if $x\gamma$ is a factor of $w' = w[i-|x|+1 \ldots n]$ then
 9.        let $j$ be the end position of the first occurrence of $x\gamma$ in $w'$;
10.        build a transition from $i$ to $j$ by $\gamma$
11.      endif
12.    endfor
13. endfor
```

Several important results on oracles have been proved in [1]. Here are the ones which will be needed in the rest of the paper. We denote $poccur(v, w)$ the ending position of the first occurrence of v in w, for each factor v of w.

Lemma 1. [1] *Let w be a word of length n on the alphabet Σ. Then we have:*

(i) *For each state i of $Omin(w)$, there is a unique minimum length word accepted in i, that we note $min_w(i)$.*
(ii) *For each state i of $Omin(w)$, we have $i = poccur(min_w(i), w)$. In addition, $min_w(i)$ is a suffix of every other word accepted in state i.*
(iii) *If $i < j$ are two states of $Omin(w)$ and $\gamma \in \Sigma$, then there exists a transition (i, γ, j) in $Omin(w)$ if and only if we have $j = poccur(min_w(i)\gamma, w)$.*
(iv) *Each factor v of w is recognized by $Omin(w)$ in a state j such that $j \leq poccur(v, w)$.*

For a word u on Σ, let $min(u) = min_u(|u|)$ and notice that if we denote $u = Pref_i(w)$, then $min(u) = min_w(i)$ and all the properties in Lemma 1 may be formulated using $min(u)$ instead of $min_w(i)$.

Remark 1. The algorithm Build_Oracle may be seen as a generic algorithm where the function used to define the word x in step 6 acts as a generator of external transitions. From this perspective, the factor/suffix oracle is the automaton defined by this generic algorithm using the precise function $min()$ as a generator. This is why, in the rest of the paper, the factor/suffix oracle will be called the **factor/suffix min-oracle** (or simply the **min-oracle**) and will be denoted (as we already did) $Omin(w)$.

The best (to the date) estimation of the maximum number of external transitions in a min-oracle was proved in [16].

Lemma 2. [16] *The number of external transitions $ETmin(w)$ of the oracle $Omin(w)$ is upper bounded by the number of maximal suffixes of w.*

3 Factor and Suffix short-Oracles

Provided a word u on Σ, denote $short(u)$ the shortest non-repeated suffix of u (by convention, $short(\varepsilon) = \varepsilon$). Then, consider the generic algorithm Build_Oracle in which the generator is now the fonction $short()$. Or, equivalently, step 6 now reads $x = short(Pref_i(w))$, instead of the affectation $x = min(Pref_i(w))$ performed to obtain $Omin(w)$. The resulting homogeneous automaton is denoted $Oshort(w)$ and is called the **short-oracle** of w. Its factor and suffix versions are obtained as for the min-oracle.

Remark 2. For some sequences w, $Omin(w)$ and $Oshort(w)$ are identical, but this is not always the case, since $short(u)$ and $min(u)$ may be different, as is the case for $u = baabbab$: $short(u) = bab$ and $min(u) = bbab$. Then $Oshort$ $(baabbababb)$ has one external transition labeled b leaving state 7 (see Figure 3) because of the occurrence of $babb$ ending in state 10. In opposition, $Omin(baabbababb)$ has no such transition since $bbabb$ has no occurrence ending in a state $j > 7$.

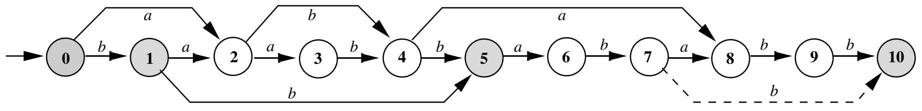


Fig. 2. The suffix short-oracle $Oshort(w)$ for $w = baabbababb$. The final states are grey. The supplementary transition with respect to $Omin(w)$ is dotted.

Although possibly different, the min- and short-oracles share many good properties, as shown by the following claim, very close to Lemma 1.

Claim 1. *Let w be a word of length n on the alphabet Σ. Then we have:*

(i) *For each word u, there is a unique shortest non-repeated suffix of u. Consequently $short(u)$ is well-defined.*
(ii) *For each state i of $Oshort(w)$, we have $i = poccur(short(u), w)$ where $u =$Pref$_i(w)$. In addition, $short(u)$ is a suffix of every other word accepted in state i.*
(iii) *If $i < j$ are two states of $Oshort(w)$ and $\gamma \in \Sigma$, then there exists a transition (i, γ, j) in $Oshort(w)$ if and only if we have $j = poccur(short(u)\gamma, w)$, where $u =$Pref$_i(w)$.*
(iv) *Each factor v of w is recognized by $Oshort(w)$ in a state j such that $j \leq poccur(v, w)$.*

It is worth noticing here that, although the external transitions of the min- and short-oracles are built according to similar rules and satisfy similar properties (items (iii) in Lemma 1 and Claim 1), it is however much easier to find $short(u)$ than $min(u)$. Indeed, $short(u)$ is simply obtained by considering every suffix of u and testing whether it occurs elsewhere in u, whereas finding $min(u)$ needs to build the min-oracle. As a consequence, it is much easier as well to estimate the number of external transitions in $Oshort(w)$ than in $Omin(w)$. This is why the following result is essential.

Claim 2. *Let w be a sequence and let $ETmin(w)$, $ETshort(w)$ be the number of external transitions in $Omin(w)$ and $Oshort(w)$ respectively. Then we have $ETmin(w) \leq ETshort(w)$.*

4 Probabilities that an External Transition Exists for Binary Alphabets

We now focus on random binary sequences issued from an unbiased Bernoulli model $\mathbb{B}$, in which a sequence w on $\Sigma = \{a, b\}$ is produced with probability $p_w = 1/2^{|w|}$. We denote by $\mathbb{B}_n$ the restriction of $\mathbb{B}$ to sequences w of length n.

The two parameters below are of great relevance for our study:

- $pmin_{i \to j}$, where $0 \leq i < j \leq n$, is the probability that there exists a transition from state i to state j in $Omin(w)$.
- $pmin_i$, where $0 \leq i < n$, is the probability that an external transition leaving state i exists in $Omin(w)$. Obviously, the equality $pmin_i = \sum_{j=i+2}^{n} pmin_{i \to j}$ holds.

We first provide exact expressions for the probabilities $pmin_{i \to j}$ and $pmin_i$ when $i = 0$ or $i = 1$. In these simple cases, it is possible to characterize precisely the language of sequences whose min-oracle possesses a transition from state i to state j, when two states i and j are given. An exact formula for the expected probability is then derived. In the general case, such a characterization is no longer possible and we use a method based on Guibas-Odlyzko's equations together with a generating functions methodology to obtain the desired probabilities, as well as their equivalents in the short-oracle.

4.1 Languages Viewpoint

Leaving state $i = 0$. First, we study the case of transitions that leave state 0. Let w be a sequence of length n and let j $(1 < j \leq n)$ be an integer. It is obvious that the min-oracle of w possesses a transition from 0 to j if and only if j is the position of first occurrence of a new letter. In the binary case, this means that w is any sequence of one of the languages $a^{j-1}b(a+b)^{n-j}$ or $b^{j-1}a(a+b)^{n-j}$. It is easy to show the following:

Claim 3. *Let j $(1 < j \leq n)$ be an integer. Under the Bernoulli model $\mathbb{B}_n$, we have $pmin_{0 \to j} = \frac{1}{2^{j-1}}$ and $pmin_0 = 1 - \frac{1}{2^{n-1}}$.*

Leaving state $i = 1$. Let j $(3 < j \leq n)$ be an integer. Two cases must be considered with respect to the two first letters of the sequence w.

If they are equal, say aa, then there is a transition from state $i = 1$ to state j if, and only if, j is the position of the first occurrence of b in w. The probability of such an event is $1/2^{j-1}$.

If they are distinct, say ab, then there exists a transition from state $i = 1$ to state j if, and only if, the first occurrence of aa ends at position j. This implies that $j > 4$ and w must belong to one of the two languages $\mathcal{L}_a = ab[(b+ab)^\star \cap (a+b)^{j-4}]aa(a+b)^{n-j}$ and $\mathcal{L}_b = ba[(a+ba)^\star \cap (a+b)^{j-4}]bb(a+b)^{n-j}$. In order to deduce the probability for w to belong to $\mathcal{L}_a$ or $\mathcal{L}_b$, we first give the following result.

Claim 4. *The number of sequences of size $J \geq 0$ of the form $(b+ab)^\star$ equals the $(J+1)$-th Fibonacci number F_{J+1} defined recursively by $F_0 = F_1 = 1$, and for all $h > 1$, $F_h = F_{h-1} + F_{h-2}$.*

Previous lemma together with Binet's formula on Fibonacci numbers ($F_J = (\phi^{J+1} - \overline{\phi}^{J+1})/\sqrt{5}$, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the Golden ratio and $\overline{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$ its conjugate), allows us to prove the following result.

Claim 5. *Let j, $3 < j \leq n$ be an integer. Under the Bernoulli model $\mathbb{B}_n$, we have*

$$pmin_{1\to j} = \frac{F_{j-3}+1}{2^{j-1}} = \frac{1}{2^{j-1}}\left[1+\frac{1}{\sqrt{5}}(\phi^{j-2}-\overline{\phi}^{j-2})\right] \qquad (1)$$

$$pmin_1 = 1 - \frac{F_{n-1}+1}{2^{n-1}}.$$

We now focus on obtaining asymptotic expressions when i is arbitrary, and need to apply a classical study involving generating functions.

4.2 Generating Functions Methodology

This section is devoted to a brief presentation of some essential tools from the generating function theory. The reader can refer to [17] for details and supplementary material. After a general approach using an alphabet with an arbitrary number of symbols that is randomly generated by a Bernoulli probabilistic process, we focus on the simpler case of a binary alphabet whose symbols are produced uniformly at random. In this section, Σ is a finite alphabet, $\Sigma^\star$ is the set of all possible words of any length and Σ^+ is the set of all possible words of any length except the empty word ε. For two sequences x and u in $\Sigma^\star$, the function $occ(x,u)$ counts the number of occurrences of motif x in the text u.

Generating functions are very useful tools to study average-case problems on languages. Let $\mathcal{L}$ be a language. The generating function $L(z)$ associated to language $\mathcal{L}$ is defined by $L(z) = \sum_{u\in\mathcal{L}} p_u z^{|u|}$, where p_u is the probability of word u to be produced. In the sequel, we denote by $[z^k]L(z) = \sum_{u\in\mathcal{L}\cap\Sigma^k} p_u$ the coefficient of z^k in $L(z)$, that equals the probability for a word of length k to belong to $\mathcal{L}$.

Consider the following three sets

$$\begin{aligned}
\mathcal{S}_x &= \{u \in \Sigma^\star, occ(x,u) = 0\},\\
\mathcal{T}_x &= \{u \in \Sigma^\star, u = v\cdot x \text{ and } occ(x,u) = 1\},\\
\mathcal{C}_x &= \{u \in \Sigma^\star, \exists v, v' \in \Sigma^+, v\cdot u = v'\cdot v = x\},
\end{aligned}$$

where $v \cdot u$ denotes the concatenation of the two words u and v in this order. These sets are very classical in the so-called Guibas-Odlyzko [6] methodology. The first one, $\mathcal{S}_x$, is the set of words that do not contain x as a factor. The second one, $\mathcal{T}_x$, is the set of words that contain x only as a suffix. Finally, $\mathcal{C}_x$ is the set of suffixes u of x such that x is a suffix of $x\cdot u$. Set $\mathcal{C}_x$ is commonly called the *autocorrelation set* of x.

In the same vein, we define the correlation set $\mathcal{C}_{x,y}$ between two words x and y by $\mathcal{C}_{x,y} = \{u \in \Sigma^\star, \exists v \in \Sigma^\star, v' \in \Sigma^+, x = v\cdot v' \text{ and } y = v'\cdot u\}$.

Sets $\mathcal{S}_x$, $\mathcal{T}_x$ and $\mathcal{C}_x$ are related by the following equalities

$$\mathcal{S}_x \times \Sigma + \varepsilon = \mathcal{S}_x + \mathcal{T}_x \quad \text{and} \quad \mathcal{S}_x \times x = \mathcal{T}_x \times \mathcal{C}_x.$$

By using decomposition properties of memoryless sources, such algebraic decompositions on sets directly translate into equations involving generating functions. By solving the resulting system of equations, one obtains:

Lemma 3 (Guibas-Odlyzko [6]). *The generating functions, denoted respectively $S_x(z)$, $T_x(z)$ and $C_x(z)$, of the sets $\mathcal{S}_x$, $\mathcal{T}_x$, $\mathcal{C}_x$ satisfy*

$$S_x(z) = \frac{C_x(z)/p_x}{D_x(z)} \quad \textit{and} \quad T_x(z) = \frac{z^{|x|}}{D_x(z)},$$

where p_x is the probability of word x to be produced and $D_x(z) = z^{|x|} + (1-z)C_x(z)/p_x$ is a polynom of degree $|x|$.

Thus $S_x(z)$ and $T_x(z)$ are rational functions whose dominant singularities (*i.e.,* the dominant roots of $D_x(z)$) dictate the main order asymptotic term of $[z^k]S_x(z)$ and $[z^k]T_x(z)$. The following lemma may be found in [15].

Lemma 4 (Szpankowski-Regnier [15]). *The coefficients of $[z^k]$ (with $k > 0$) in $S_x(z)$ and $T_x(z)$ satisfy*

$$[z^k]S_x(z) = K_x\rho_x^{-(k+1)} + O(\mu_x^{-k}) \textit{ and } [z^k]T_x(z) = K'_x\rho_x^{-(k-|x|+1)} + O(\mu_x^{-k}),$$

where ρ_x is the root of $D_x(z)$ of smallest modulus, $K_x = \frac{-C_x(1)}{p_x D'_x(\rho_x)}$, $K'_x = \frac{-1}{D'_x(\rho_x)}$ and μ_x is the second modulus of roots of $D_x(z)$.

As an example, it is easy to get the main order term of $pmin_{1\to j}$. In this case, $pmin_{1\to j}$ and the generating function of $\mathcal{T}_{aa}$ are related by $pmin_{1\to j} = \frac{1}{2^{j-1}} + 2[z^j]T_{aa}(z)$. The denominator $D_{aa}(z) = z^2 + 4(1-z)(1+z/2)$ of $T_{aa}(z)$ possesses $\rho_{aa} = 2/\phi$ as dominant root and $\mu_{aa} = |2/\overline{\phi}| \approx 3.236$. Applying Lemma 4 leads to the expected asymptotic expression of $pmin_{1\to j}$ given in equation (1).

In the case of binary Bernoulli unbiased sources, the root ρ_w can be approximated by a quantity depending only on the word length $|x|$.

Claim 6. *Let x be a binary word of length $k > 1$, $s_k = \rho_{a^k}$ and $r_k = \rho_{a^{k-1}b}$. We have:*

(i) $s_k \le \rho_w \le r_k$;
(ii) $r_{k+1} = s_k$;
(iii) *if* $|x| = k > 2$, $\rho_x = 1 + \frac{1}{2^k} + o(1/2^k)$.

4.3 Probabilities of an External Transition : General Case

Define the two parameters $pshort_{i\to j}$ and $pshort_i$ similarly to $pmin_{i\to j}$ and $pmin_i$, but for $Oshort(w)$.

Now, coming back to the binary case we show how transition probabilities ($pmin_{i\to j}$, $pmin_i$, $pshort_{i\to j}$ and $pshort_i$) can be related to Guibas-Odlyzko languages $\mathcal{S}_x$ and $\mathcal{T}_x$ defined in previous section.

Remark 3. For the sake of simplicity, we deduce in this subsection general expressions *only* for $pmin_{i\to j}$ and $pmin_i$. However, the reader will easily notice that the only property of $Omin(w)$ used in this section is Lemma 1 (iii), and that this property has an equivalent for $Oshort(w)$, namely Claim 1 (iii). Consequently, the reasoning and the results in this part are easily transfered to $Oshort(w)$ (just by replacing $min(u)$ by $short(u)$ appropriately), so as to obtain similar expressions for $pshort_{i\to j}$ and $pshort_i$.

For each letter $m \in \{a, b\}$, notation $\overline{m}$ designates the opposite letter (e.g., $\overline{a} = b$ and $\overline{b} = a$). We now prove the two following claims.

Claim 7. *Let $i < j - 1$. The set $\mathcal{P}_{i\to j,n}$ of all binary words of length n whose oracle possesses an external transition from state i to state j is*

$$\mathcal{P}_{i\to j,n} = \bigcup_{u\in\Sigma^i, m\in\Sigma} u \cdot m \cdot \left(\left(\mathcal{T}_{\min(u)\cdot\overline{m}} \cup \mathcal{C}_{\min(u)\cdot m,\min(u)\cdot\overline{m}}\right) \cap \Sigma^{j-i-1}\right) \cdot \Sigma^{n-j}.$$

In the same vein, it is possible to obtain a similar expression for the transitions leaving a given state.

Claim 8. *The set $\mathcal{P}_{i,n}$ of all binary words of length n whose factor oracle possesses an external transition leaving state i equals*

$$\mathcal{P}_{i,n} = \bigcup\nolimits_{u\in\Sigma^i, m\in\Sigma} u \cdot m \cdot \Big(\big({}^c\mathcal{S}_{\min(u)\cdot\overline{m}} \cup \\ (\mathcal{S}_{\min(u)\cdot\overline{m}} \cap (\mathcal{C}_{\min(u)\cdot m,\min(u)\cdot\overline{m}} \cdot \Sigma^\star))\big) \cap \Sigma^{n-i-1}\Big),$$

where ${}^cX = \Sigma^\star \setminus X$ denotes the complementary set of X.

Formulas for $\boldsymbol{pmin_{i\to j}}$ and $\boldsymbol{pmin_i}$. It is now obvious to derive expressions for $pmin_{i\to j}$ and $pmin_i$ by means of dominant roots of Guibas-Odlyzko's generating functions. Indeed, $pmin_{i\to j} = \sum_{w\in\mathcal{P}_{i\to j,n}} p_w$ and $pmin_i = \sum_{w\in\mathcal{P}_{i,n}} p_w$. Then, Claims 7 and 8 allow to express these probabilities as particular coefficients of generating functions $T_{\min(u)\cdot\overline{m}}(z)$, $S_{\min(u)\cdot\overline{m}}(z)$ and $C_{\min(u)\cdot m,\min(u)\cdot\overline{m}}(z)$. The following claim providing asymptotic approximations for the transition probabilities is a direct consequence of Lemma 4.

Claim 9. *Under $\mathbb{B}_n$, the probabilities that an external transition exists satisfy*

$$pmin_{i\to j} = \frac{1}{2^{i+1}} \sum_{u\cdot m\in\Sigma^{i+1}} K'_{\min(u)\cdot\overline{m}} \rho_{\min(u)\cdot\overline{m}}^{-j+i+|\min(u)|+1} + O(1/2^{j-i}),$$

$$pmin_i = \frac{1}{2^{i+1}} \sum_{u\cdot m\in\Sigma^{i+1}} \left(1 - K_{\min(u)\cdot\overline{m}} \rho_{\min(u)\cdot\overline{m}}^{-n+i}\right) + O(1/2^{n-i-1}),$$

where ρ_x, K_x and K'_x are quantities defined in Lemma 4.

Simpler approximations. The two previous expressions are quite ineffective because they involve sums over all possible words of a given length. Now, we show that it is possible to obtain computable approximation formulas for $pmin_{i \to j}$ and $pmin_i$. The approximation involves the probability distribution of the minimum length words, which is defined as follows. Let $M(u) = |\min(u)|$ be the function that associates with any word u the length of its minimum length word. The restriction of $M(u)$ to $\mathbb{B}_i$ is itself a random variable denoted by M_i. Its probability distribution, called in the sequel *probability distribution of minimum length words* is defined by $\text{Prob}\{M_i = k\} = \sum_{u \in \Sigma^i, M(u)=k} \frac{1}{2^i}$.

Claim 10. *Let* $\text{Prob}\{M_i = k\}$ *be the probability distribution of minimum length words,* $\alpha_k = \frac{1}{2^k}$ *and* $\lambda_k = 1 + \frac{1}{2^k}$*. The transition probabilities* $pmin_{i \to j}$ *and* $pmin_i$ *satisfy*

$$pmin_{i \to j} = \sum_{k=1}^{i} \text{Prob}\{M_i = k\} \, \alpha_{k+1} \lambda_{k+1}^{-j+i+k+1} + O(1/2^{j-i}),$$

$$pmin_i = 1 - \sum_{k=1}^{i} \text{Prob}\{M_i = k\} \, \lambda_{k+1}^{-n+i} + O(1/2^{n-i-1}).$$

Remark 4. According to Remark 3, $pshort_{i \to j}$ and $pshort_i$ satisfy the same equalities as $pmin_{i \to j}$ and $pmin_i$ in Claim 10, up to $\text{Prob}\{M_i = k\}$ which is replaced by $\text{Prob}\{S_i = k\}$, where $S(u) = |\text{short}(u)|$ is the size of the minimum length non repeated suffix of u and S_i its restriction to $\mathbb{B}_i$.

5 Average Space Occupancy

The memory requirement for storing the min-oracle of w is the sum of the number of states of $Omin(w)$ (fixed and equal to $n+1$), the number of internal transitions (fixed and equal to n) and the number of external transitions. As an application of our results, we present now an estimation of the average space occupancy (in terms of external transitions) $\text{E}[ETmin_n]$, where $ETmin(w)$ is the function that counts the number of external transitions of $Omin(w)$ and $ETmin_n$ is its restriction on $\mathbb{B}_n$. This estimation is computable in linear time.

Theorem 1. *Under* $\mathbb{B}_n$ *(the set of random independently and identically distributed binary words of length* n*), the average space occupancy* $\text{E}[ETmin_n]$ *in terms of external transitions of a* min*-oracle for a word of length* n *satisfies*

$$\text{E}[ETmin_n] \leq pmin_0 + pmin_1 + (n-3) - \sum_{k=2}^{n-2} \frac{\gamma_k^{k-1} \lambda_{k+1}^{k-n} - \gamma_k^{n-2} \lambda_{k+1}^{-1}}{1 - \gamma_k \lambda_{k+1}}$$
$$- \sum_{k=2}^{n-2} \frac{\gamma_{k-1}^{k-1} \lambda_{k+1}^{k-n} - \gamma_{k-1}^{n-2} \lambda_{k+1}^{-1}}{1 - \gamma_{k-1} \lambda_{k+1}}$$

with $\gamma_k = 1 - \frac{1}{2^k}$ *and* $\lambda_k = 1 + \frac{1}{2^k}$.

Proof. First notice that the average space occupancy equals the sum of all probabilities of leaving states,

$$\mathrm{E}\left[ETmin_n\right] = \sum_{i=0}^{n-2} pmin_i.$$

It is thus of great interest to obtain a tractable formula for $pmin_i$ and consequently for $\mathrm{Prob}\left\{M_i = k\right\}$, the distribution probability of minimum length words. It is still a challenge to obtain such formulas for min-oracles. Claim 2 proves that the average number $\mathrm{E}\left[ETshort_n\right]$ of external transitions of short-oracles provides an upper bound for the expectation $\mathrm{E}\left[ETmin_n\right]$. Then we concentrate on computing $\mathrm{E}\left[ETshort_n\right] = \sum_{i=0}^{n-2} pshort_i$, where the expression of $pshort_i$ is obtained using Remark 4. We then study the probability distribution of S_i. Then, considering the prefix tree built using all the prefixes of the mirror $w_i \cdots w_1$ of word $w = w_1 \cdots w_i$, $|short(w)| - 1$ exactly equals the insertion depth of the i-th prefix in the tree. In [14], Park et al. study the probability distribution of insertion depth in the case of random words built by an i.i.d. binary source. Applying their results to S_i yields $\mathrm{Prob}\left\{S_i = k\right\} = \gamma_k^{n-1} - \gamma_{k+1}^{n-1}$, with $\gamma_k = 1 - 2^{-k}$. Next, we use exact formulas for $pshort_0 = pmin_0$ and $pshort_1 = pmin_1$ and approximations for other probabilities. Finally, it is possible to invert the double sum $\sum_{i=2}^{n-2} \sum_{k=2}^{i}$ into $\sum_{k=2}^{n-2} \sum_{i=2}^{n-2}$ which involves geometric sums leading to the expected result. □

6 Conclusion

In this paper, we provide precise approximations for the probabilities that an external transition exists in the min- and short-oracles. These approximations allow us to study the average space occupancy of these oracles. The main goal of such results is to allow comparing the factor/suffix oracle with other indexing structures such as suffix trees, whose space occupancy closely depends on the number of its internal edges, that is known to be of order $n/\log 2$ (see [7]). Figure 3 compares our bound on the average number of external transitions to the average number of edges of suffix trees. This latter figure suggests a conjecture of $n/3 + 1$ for the average number of external transitions of short-oracles.

Notice that one of the main open questions arising when studying oracles concerns the number of words, recognized by an oracle, that are not factor or suffixes. Our results should certainly be helpful since the total number of words recognized by a factor oracle expresses as a sum $\sum_{k=0}^{n} N_k$, where N_i is the expected number of words recognized in state i. They satisfy $N_0 = 1$ and for all $0 < j \leq n$, $N_j = \sum_{i=0}^{j} pmin_{i \to j} N_i$. It is still a challenge to solve this latter recurrence. Nevertheless, it is quite easy to design a dynamical programming algorithm yielding an upper bound for the expected number of words recognized by a min-oracle, in the same vein of our bound for the expected number of external transitions.

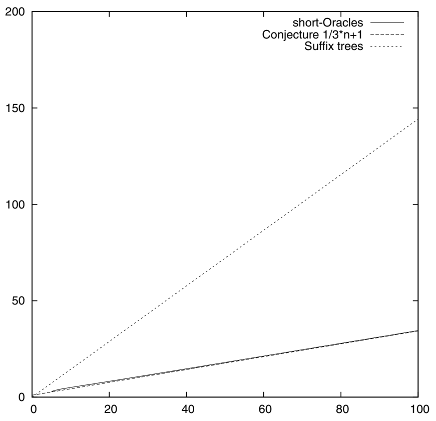


Fig. 3. A comparison of the space occupancy of short-oracles and suffix trees

References

1. Allauzen, C., Crochemore, M., Raffinot, M.: Factor Oracle: A New Structure for Pattern Matching. In: Bartosek, M., Tel, G., Pavelka, J. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 295–310. Springer, Heidelberg (1999)
2. Assayag, G., Dubnov, S.: Using Factor Oracles for Machine Improvisation. Soft Computing 8, 1–7 (2004)
3. Cleophas, L., Zwaan, G., Watson, B.W.: Constructing Factor Oracles. In: Proceedings of the Prague Stringology Conference 2003 (PSC 2003), pp. 37–50 (2003)
4. Crochemore, M.: Reducing space for index implementation. Theoretical Computer Science 292, 185–197 (2003)
5. Crochemore, M., Ilie, L., Seid-Hilmi, E.: The Structure of Factor Oracles. Int. J. Found. Comput. Sci. 18(4), 781–797 (2007)
6. Guibas, L.J., Odlyzko, A.M.: String Overlaps, Pattern Matching, and Nontransitive Games. J. Combin. Theory Ser. A 30(2), 183–208 (1981)
7. Jacquet, P., Szpankowski, W.: Autocorrelation on words and its applications: analysis of suffix trees by string-ruler approach. J. Combin. Theory Ser. A 66(2), 237–269 (1994)
8. Kato, R.: A new full-text search algorithm using factor oracle as index, TR-C185, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan (2003)
9. Kato, R.: Finding maximal repeats with factor oracles, TR-C190, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan (2004)
10. Lecroq, T., Lefebvre, A.: Computing repeated factors with a factor oracle. In: Brankovic, L., Ryan, J. (eds.) Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms, pp. 145–158 (2000)
11. Lecroq, T., Lefebvre, A.: Compror: on-line lossless data compression with a factor oracle. Information Processing Letters 83, 1–6 (2002)
12. Mancheron, A.: Extraction de motifs communs dans un ensemble de séquences, Ph. D. thesis, University of Nantes, France (2006)

13. Mancheron, A., Moan, C.: Combinatorial characterization of the language recognized by factor and suffix oracles. International Journal of Foundations of Computer Science 16(6), 1179–1191 (2005)
14. Park, G., Hwang, H.-K., Nicodème, P., Szpankowski, W.: Profile of Tries. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 1–11. Springer, Heidelberg (2008)
15. Régnier, M., Szpankowski, W.: On Pattern Frequency Occurrences in a Markovian Sequence. Algorithmica 22(4), 631–649 (1998)
16. Rusu, I.: Converting Suffix Trees into Factor/Suffix Oracles. Journal of Discrete Algorithms 6(2), 324–340 (2008)
17. Szpankowski, W.: Average case analysis of algorithms on sequences. Wiley-Interscience Series in Discrete Mathematics and Optimization (2001)
18. Wells, D.: The Penguin Book of Curious and Interesting Mathematics (1997)

Haplotype Inference Constrained by Plausible Haplotype Data

Michael R. Fellows[1], Tzvika Hartman[2,⋆], Danny Hermelin[3,⋆⋆], Gad M. Landau[3,4,⋆⋆⋆], Frances Rosamond[1], and Liat Rozenberg[3]

[1] The University of Newcastle, Callaghan NSW 2308 - Australia
{mike.fellows,frances.rosamond}@cs.newcastle.edu.au
[2] Google, Tel-Aviv
tzvika@google.com
[3] Department of Computer Science, University of Haifa, Haifa - Israel
{danny,liat}@cri.haifa.ac.il, landau@cs.haifa.ac.il
[4] Department of Computer and Information Science, Polytechnic University, New York - USA

Abstract. The *haplotype inference problem* (HIP) asks to find a set of haplotypes which resolve a given set of genotypes. This problem is of enormous importance in many practical fields, such as the investigation of diseases, or other types of genetic mutations. In order to find the haplotypes that are as close as possible to the real set of haplotypes that comprise the genotypes, two models have been suggested which by now have become widely accepted: The *perfect phylogeny* model and the *pure parsimony* model. All known algorithms up till now for the above problem may find haplotypes that are not necessarily plausible, *i.e.* very rare haplotypes or haplotypes that were never observed in the population. In order to overcome this disadvantage we study in this paper, for the first time, a new constrained version of HIP under the above mentioned models. In this new version, a pool of plausible haplotypes $\widetilde{H}$ is given together with the set of genotypes G, and the goal is to find a subset $H \subseteq \widetilde{H}$ that resolves G. For the *constrained perfect phylogeny haplotyping* (CPPH) problem we provide initial insights and polynomial-time algorithms for some restricted cases that help understanding the complexity of that problem. We also prove that the *constrained parsimony haplotyping* (CPH) problem is fixed parameter tractable by providing a parameterized algorithm that applies an interesting dynamic programming technique for solving the problem.

1 Introduction

Genetic information in living organisms is encoded in DNA sequences that are organized into *chromosomes*. Diploid organisms such as humans have two copies

⋆ Work done while at CRI, Haifa University, and Department of Computer Science, Bar-Ilan University.

⋆⋆ Supported by the Adams Fellowship of the Israel Academy of Sciences and Humanities.

⋆⋆⋆ Partially supported by the Israel Science Foundation grant 35/05 and the Israel-Korea Scientific Research Cooperation.

of every chromosome, which are not necessarily identical, with each copy called a *haplotype*. Identifying the common genetic variations that occur in humans are valuable in understanding diseases [1]. The genetic sequences of the population are almost totally identical, except from some bases that differ from one person to another with a frequency of more than some threshold (1% for example). Those differences are the common genetic variations and they are known as *single nucleotide polymorphisms* (SNPs).

The data described in each haplotype may be the full DNA, but it is more common to consider only the data of the SNPs since the other sites are assumed to be identical. A *genotype* is the description of the two copies (haplotypes) together. When the two haplotypes agree, the site in the genotype has the agreed base. Such a site is called a *homozygous* site. When the two haplotypes disagree, the genotype has both bases, yet it does not tell which base occur in which haplotype. This type of site is called *heterozygous*.

Current biological technologies give us an easier and cheaper way to obtain genotype data in comparison to haplotype data. However, the haplotype information is the one of greater use [14]. For this reason, it is necessary to computationally infer the haplotype information from the genotype data. An important biological fact is that almost always there are only two bases at a SNP, which can be marked as 0 and 1. A genotype will have 0 or 1 if the two haplotypes both have 0 or 1 in the same site respectively, or 2 otherwise.

In view of that, a set of genotypes and a set of haplotypes can be represented as matrices. A *genotype matrix* is a matrix over $\{0, 1, 2\}$ where each row is a genotype and each column represents SNP, and a *haplotype matrix* is a matrix over $\{0, 1\}$, where each row is a haplotype and each column represents SNP.

For the rest of the paper, let $g(i)$ represent the data at site i of genotype g, and $h(i)$ the data at site i of haplotype h.

Definition 1 (Resolution). *A pair of haplotypes $\{h, h'\}$ is said to* resolve *g if for each i: $g(i) = h(i)$ where $h(i) = h'(i)$, and $g(i) = 2$ otherwise. We extend this and say that a set of haplotypes H* resolves *a set of genotypes G, if for each $g \in G$, there is a pair $\{h, h'\} \in H$ which resolves g. The pair $\{h, h'\}$ is called a* resolution *of g, and H is* resolution *of G.*

Definition 2 (Haplotype Inference Problem (HIP)). *Given a set of n genotypes G, each of length m, find a resolution of G.*

Note that if a genotype has $d \leq m$ heterozygous sites (sites marked with 2), then the number of possible resolving pairs is 2^{d-1}. The goal is to find the set of pairs which as close as possible to the real set of haplotypes that created the genotype. Currently, there are two models used in practice that give two different biologically motivated heuristics on how to determine this:

1. **Perfect Phylogeny:** The perfect phylogeny model is a coalescent model which assumes no recombination. This means that the history of the haplotypes is represented as a tree where two haplotypes from two individuals have at most one recent common ancestor [14] (see [12,14,19,27] for further

information). Formally, a set of haplotypes (binary sequences) of length m *defines a perfect phylogeny* if the haplotypes appear as labels of a rooted tree which obeys the following properties [8]:

- Each vertex of the tree is labeled by a binary sequence of length m representing a possible haplotype;
- Every edge (u, v) is marked with i, where the base at site i in sequence u is different from the one in sequence v. Every coordinate i labels at most one edge.

A common way of checking whether a set of haplotypes defines a perfect phylogeny is to check whether it obeys the *four gamete test*, *i.e.* the corresponding haplotype matrix does not contain, in any two columns, the *forbidden gamete submatrix*:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}.$$

See Figure 1 for an example of a perfect phylogenetic tree, and the corresponding haplotype matrix. The *Perfect Phylogeny Haplotyping* (PPH) problem is the problem of finding for a given set of genotypes a resolution which defines a perfect phylogeny, if such resolution exists.

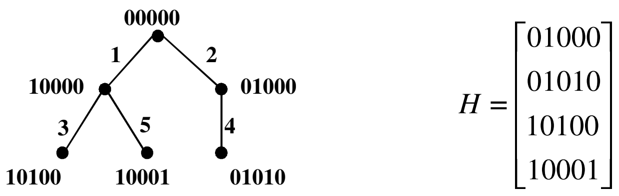


Fig. 1. Example of a perfect phylogenetic tree for the haplotypes $h_1 = (01000), h_2 = (01010), h_3 = (10100), h_4 = (10001)$. H is the haplotype matrix of the above haplotypes, which obeys the four gamete test.

2. **Pure Parsimony:** The pure parsimony model seeks the minimum set of haplotypes that resolves a given set of genotypes. The biological motivation behind this is the statistical observation that the number of distinct haplotypes in the population is vastly small [13,14]. The *Parsimony Haplotyping* (PH) problem is the problem of finding a resolution of smallest size possible for a given set of genotypes.

In [12], Gusfield showed that the PPH problem is solvable in $O(nm\alpha(nm))$ time, where α is the inverse Ackerman function. Gusfield also showed a linear-time algorithm to build, once the first solution is found, a linear-space data structure that represents all PPH solutions. However, his work is based on complex graph-theoretic algorithms which are difficult to implement [14]. In [2,8], algorithms fine-tuned to the actual combinatorial structure of the PPH problem were shown. These algorithms run in $O(nm^2)$ time and are easy to understand and implement. They also give a representation of all PPH solutions. More recent work developed $O(nm)$ time algorithms: In [6], the algorithm is graph-theoretic and uses a directed rooted graph called a "shadow tree", and in [25], the algorithm is based on interdependencies among the pairs of SNPs, and builds a data structure called "FlexTree" to represent all PPH solutions. Other works have researched different variations of PPH [3,5,9,18].

The parsimony haplotyping (PH) problem was first suggested and proved to be NP-hard by Earl Hubell (unpublished). Gusfield formally introduced the problem in [13], and proposed an integer linear programming solution. More integer linear programming solutions following this were proposed in [4,13,16,22]. Approximation algorithms for the problem were presented in [22,23], and in [28], a branch-and-bound algorithm was proposed. Other theoretical results were shown in [20,22,26]. Most notably is the work of Sharan *et. al* [26], who characterized restricted instances of PH under the term $(\alpha, \beta) - bounded$, where α and β stand for the maximum number of heterozygous sites per row and column of the genotype matrix. Sharan *et al.* also showed that the PH problem is *fixed parameter tractable* (see [7] for formal definition) when parameterized by the number of k haplotypes in the resolution of G. Many other works have researched other variations of the haplotyping problem [10,11,15,17,21,24].

All known algorithms up till now for haplotype inference under the perfect phylogeny model find resolutions for a given set of genotypes from the superset of all possible haplotypes (*i.e.* all m-length binary vectors). However, these algorithms may find resolutions that include binary vectors representing haplotypes that do not actually occur in the population, or are otherwise very rare. It is therefore biologically interesting to force the resolving haplotypes to be chosen only from a specific pool that contains only *plausible* haplotypes, *i.e.* haplotypes which have already been observed in relatively high frequencies in previous experiments. This pool can be determined by empirically setting up some statistical threshold, or by any other reasonable method.

In view of all this, we study here for the first time, a new constrained variant of the haplotype inference problem, in which a pool of plausible haplotypes $\widetilde{H}$ is given alongside the set of genotypes G, and the goal is to find a resolution of G which is a subset of $\widetilde{H}$.

Definition 3 (Constrained Haplotype Inference Problem (CHIP)). *Given a set of ℓ distinct genotypes G, each of length m, and a pool of n distinct plausible haplotypes $\widetilde{H}$ for G, each of length m, find a resolution $H \subseteq \widetilde{H}$ of G.*

The constrained perfect phylogeny haplotyping (CPPH) problem and the constrained parsimony haplotyping (CPH) problem are defined accordingly. Note that if $\ell > n(n-1)$ in the above definition, there is no solution automatically, since taking the entire pool of n plausible haplotypes we can resolve at most $n(n-1)$ genotypes. On the other hand, there is no inequality necessarily in the other direction. We therefore assume $\ell \leq n(n-1)$ throughout the paper.

In this paper, we provide an initial insight to determining the complexity of CPPH. We present a framework which helps partially answer this question, and allows polynomial-time solutions for the CPPH (α, β) bounded cases of (*,1), (2,*), (5,2), and (3,3). As is the case for the PH problem [8,26] these cases can be very useful for speeding up implementations of CPPH, but they also give a glimpse into the complexity of the problem. For example, while it was proved in [8] that MPPH and PH for (3,3)-bounded genotype matrices are both APX-hard, we show that CPPH is polynomial-time solvable in this case.

In the second part of the paper we turn to consider CPH. We show that like PH [8], CPH is fixed-parameter tractable when parameterized by the number of haplotypes k in a minimum resolution $H \subseteq \widetilde{H}$ of G. The parameterized algorithm for CPH, is however much more involved than the one for PH, and it applies an interesting dynamic programming technique for solving the problem. Proofs are omitted due to space considerations.

2 Constrained Perfect Phylogeny Haplotyping

In this section we describe polynomial-time algorithms for CPPH with genotype matrices of specific structures. In [26], bounded cases of genotype matrices were introduced in order to explore the complexity of PH. The bounded cases were defined as follows:

Definition 4 ((α,β)-bounded [26]). *A genotype matrix G is* (α,β)-bounded *if it has at most α 2's per row and at most β 2's. α and β might be * which means there is no bound on the number of 2s per row or column, respectively.*

Here we use the same term of (α,β)-bounded to present polynomial-time algorithms for special cases of CPPH. We will present algorithms for the following cases: (*,1), (2,*), (5,2) and (3,3).

We begin with the following lemma which lists ten matrices that we can assume G does not include, since including any one of them implies that all resolutions of G necessarily include the forbidden gamete submatrix. Its proof is left to the reader.

Lemma 1. *If G includes one of the following 2×3 submatrices:*

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 1 & 2 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 2 & 0 \\ 1 & 2 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 0 & 2 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}, \text{ or } \begin{pmatrix} 0 & 2 \\ 1 & 1 \\ 1 & 0 \end{pmatrix},$$

or one of the following 2×2 submatrices: $\left(\begin{smallmatrix} 2 & 0 \\ 2 & 1 \end{smallmatrix}\right)$ *or* $\left(\begin{smallmatrix} 0 & 2 \\ 1 & 2 \end{smallmatrix}\right)$, *then G does not have a perfect phylogenic resolution.*

As in Eskin *et al.* [8], we will be working with pairs of columns in G. Pairs of sites of a genotype can be split into two types, according to the data in those sites. Type I includes the pairs of sites that have only one possible resolution. Those pair of sites are (00), (01), (10), (11), (20), (21), (02) and (12). The resolutions of those sites are described in the following list:

1. (00) $\rightarrow \left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$	2. (01) $\rightarrow \left(\begin{smallmatrix} 0 & 1 \\ 0 & 1 \end{smallmatrix}\right)$	3. (10) $\rightarrow \left(\begin{smallmatrix} 1 & 0 \\ 1 & 0 \end{smallmatrix}\right)$
4. (11) $\rightarrow \left(\begin{smallmatrix} 1 & 1 \\ 1 & 1 \end{smallmatrix}\right)$	5. (20) $\rightarrow \left(\begin{smallmatrix} 0 & 0 \\ 1 & 0 \end{smallmatrix}\right)$	6. (21) $\rightarrow \left(\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right)$
7. (02) $\rightarrow \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$	8. (12) $\rightarrow \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right)$	

Type II includes pairs of sites with (22) (*22-columns*). A 22-columns have two potential resolutions: (22) $\rightarrow \left(\begin{smallmatrix}0&0\\1&1\end{smallmatrix}\right)$, which will be called *equal resolution*, or (22) $\rightarrow \left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right)$, which will be called *unequal resolution*.

Determining whether there is a perfect phylogenic resolution of G boils down to deciding the resolution type, equal or unequal, for any pair of 22-columns. For some 22-columns, the type of the resolution is determined by the given set of genotypes, for others it determined by the given set of haplotypes, and for the rest we need algorithms that will find the proper resolution.

2.1 Preprocessing

We next present a preprocessing stage which is performed before all algorithms regardless of the specific structure of the input sets of genotypes or haplotypes.

A 22-columns ij must be resolved equally if the given set of genotypes G includes at least one of the following submatrices in columns ij: $\left(\begin{smallmatrix}0&0\\1&1\end{smallmatrix}\right), \left(\begin{smallmatrix}2&0\\1&2\end{smallmatrix}\right)$ or $\left(\begin{smallmatrix}2&1\\0&2\end{smallmatrix}\right)$, since any resolution of G must include the combinations "00" and "11" in this case. For the same reason columns ij must be resolved unequally when G includes at least one of the submatrices $\left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right), \left(\begin{smallmatrix}2&0\\0&2\end{smallmatrix}\right)$ or $\left(\begin{smallmatrix}2&1\\1&2\end{smallmatrix}\right)$. We will call this type of constraints on the resolution type *genotypes constraints*. In addition, a 22-columns ij must be resolved equally (unequally) if the haplotypes set includes for some genotype only equal (unequal) resolutions. This type of constraints will be called *haplotypes constraints*.

The preprocessing ensures that haplotype-pairs which violate the above constraints will not be chosen. For each genotype $g_i \in G$, we use $\widetilde{H}(g_i)$ to denote all possible resolutions of g_i in $\widetilde{H}$, *i.e.* $\widetilde{H}(g_i) = \{\{h, h'\} \,|\, h, h' \in \widetilde{H}, h \text{ and } h' \text{ resolve } g_i\}$. The preprocessing step includes the following four steps:

1. Check whether the genotype matrix G can be resolved in a perfect phylogenic way (use any algorithm from [2,6,8,25]). If not, report there is no solution.
2. For each genotype $g_i \in G$, $1 \le i \le \ell$, go over all pairs of haplotypes from $\widetilde{H}$ and compute $\widetilde{H}(g_i)$.
3. For each genotype constraint, delete from the sets $\widetilde{H}(g_1), \ldots, \widetilde{H}(g_\ell)$ all haplotype pairs that violate the constraint, *i.e.* resolve the relevant sites in a different way than the constraint indicates.
4. For each haplotype constraint, delete from the sets $\widetilde{H}(g_1), \ldots, \widetilde{H}(g_\ell)$ all the haplotype pairs that violate it. Note that the deletion of haplotypes may create a new haplotype constraints. Repeat Step 4 until there is no change in the haplotype constraints.

After each step of steps 2 to 4 in the preprocessing stage, if one of the haplotypes sets $\widetilde{H}(g_1), \ldots, \widetilde{H}(g_\ell)$ becomes empty, it means there is no solution and we done. Once the preprocessing is complete, our goal is to find a resolution $H \subseteq \widetilde{H}$ of G, by selecting one pair of haplotypes from each $\widetilde{H}(g)$, $g \in G$. From here on out, we will only be concerned with resolutions of this type. Note that even after the preprocessing stage, not all resolutions of this type will define a perfect phylogeny. This is because we are still left with 22-columns that have yet been resolved, as there might be two different genotypes g and g' which share a common pair of 22-columns ij, and $\widetilde{H}(g)$ and $\widetilde{H}(g')$ includes both resolutions for ij (*i.e.* equally and unequally). Such a pair of resolution is said to be *conflicting*, and more generally, any pair of resolutions $\{h_1, h'_1\}$ and $\{h_2, h'_2\}$ are *conflicting* if $\{h_1, h'_1, h_2, h'_2\}$ does not define a perfect phylogeny. We have the following two important lemmas:

Lemma 2. *After the preprocessing stage, $\{h_1, h'_1\}, \ldots, \{h_r, h'_r\}$ are pairwise non-conflicting resolutions of r genotypes in G iff $H = \bigcup_{1 \le i \le r} \{h_i, h'_i\}$ defines a perfect phylogeny.*

Proof. Let $G' \subseteq G$ denote the subset of r genotypes which H resolves. If H defines a perfect phylogeny, then clearly $\{h_1, h'_1\}, \ldots, \{h_r, h'_r\}$ are pairwise non-conflicting. To prove the other direction of the lemma, it suffices to show that any four haplotypes in H define a perfect phylogeny. Suppose in way of contraction that this is not the case. Then there are four haplotypes h_a,h_b,h_c, and h_d in H, such that $\{h_a, h_b, h_c, h_d\}$ do not define a perfect phylogeny. This means that there is a pair of sites $i, j \in \{1, \ldots, m\}$ such that $\{h_a, h_b, h_c, h_d\}$ will have the forbidden gamete matrix at ij. There are three possible cases:

i) The four haplotypes belong to two different resolutions in $\{h_1, h'_1\}, \ldots, \{h_r, h'_r\}$. But this contradicts the assumption that $\{h_1, h'_1\}, \ldots, \{h_r, h'_r\}$ are pairwise non-conflicting.

ii) The four haplotypes belong to three different resolutions in $\{h_1, h'_1\}, \ldots, \{h_r, h'_r\}$. Then, w.l.o.g., $\{h_a, h_b\}$ is resolution of some genotype $g \in G'$, and ij is a pair of 22-columns in g, since $h_a(i) \neq h_b(i)$ and $h_a(j) \neq h_b(j)$. Suppose w.l.o.g. (the other case is symmetric) that h_a and h_b resolve ij equally, *i.e.* $h_a(i) = h_a(j)$ and $h_b(i) = h_b(j)$, and let $g', g'' \in G'$ be the two genotypes that h_c and h_d resolve. Then it is not hard to verify that G must include one of the following five submatrices at rows $g'g''$ and columns ij:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \begin{pmatrix} 2 & 0 \\ 2 & 1 \end{pmatrix}, \text{ or } \begin{pmatrix} 0 & 2 \\ 1 & 2 \end{pmatrix}.$$

If G includes the last two submatrices, then G does not have a perfect phylogenic resolution in the first place (Lemma 1), and so the preprocessing stage would have reported "no solution". If G includes the first three submatrices, then there is a genotype constraint on ij stating that it must be resolved unequally, and so $\{h_a, h_b\}$ would have been removed from $\widetilde{H}$ at step 3 of the preprocessing stage. In both cases we reach a contradiction.

iii) The four haplotypes belong to four different resolutions in $\{h_1, h'_1\}, \ldots, \{h_r, h'_r\}$. Consider the four genotypes $g_a, g_b, g_c, g_d \in G'$ that h_a,h_b,h_c, and h_d resolve. If ij is a pair of 22-columns in one of these genotypes, then this case is similar to the previous case. Otherwise, it is not difficult to verify that G must include one of the forbidden submatrices of Lemma 1, and so the preprocessing stage would have halted at its first step. Therefore, contradiction.

In all three cases we have reached a contradiction, and so the lemma is proven. □

Lemma 3. *The preprocessing stage takes $O(m^4n^2 + m^2n^4)$ time.*

2.2 The Dependency Graph

This brings us to the notion independency and dependency between genotypes. Loosely speaking, a dependency between two genotypes $g, g' \in G$ arises when the decision on how to resolve g affects the decision on how to resolve g'. This obviously happens when there is a resolution $\{h_1, h_1'\} \in \widetilde{H}(g)$ conflicting with a resolution $\{h_2, h_2'\} \in \widetilde{H}(g')$. In this case we say that g and g' are *directly dependent*. If there is no resolution in $\widetilde{H}(g)$ conflicting with a solution in $\widetilde{H}(g')$, we say that g and g' are *independent*.

We next introduce the *dependency graph* $DG(G)$ of our given set of genotypes G, after they have been preprocessed by the algorithm in the previous section. Later on, we will use the properties of the dependency graph in our polynomial algorithms.

Definition 5 (dependency graph). *The* dependency graph $DG(G)$ *of a set of genotypes G is a graph which has a vertex for each genotype $g \in G$, and edge between vertices representing directly dependent genotypes.*

Lemma 4. *After the preprocessing stage, two genotypes that do not have any pair of 22-columns in common are independent.*

Proof. Consider two genotypes $g, g' \in G$ that don't have any common pair of 22-columns. Suppose by way of contradiction that there is a resolution $\{h_x, h_y\} \in \widetilde{H}(g)$ conflicting with a $\{h_{x'}, h_{y'}\} \in \widetilde{H}(g')$. This means that there is a pair of columns ij of $\{h_x, h_y, h_{x'}, h_{y'}\}$ that has the forbidden gamete matrix. But this can only happen when G includes (in the rows gg', and in columns ij) one of the two 2×2 forbidden submatrices of Lemma 1. $\square$

Lemma 5. *Let G_1 and G_2 be two connected components in $DG(G)$. If $H_1 \subseteq \widetilde{H}$ and $H_2 \subseteq \widetilde{H}$ are perfect phylogenic resolutions of G_1 and G_2 respectively, then $H_1 \cup H_2$ is a perfect phylogenic resolution of $G_1 \cup G_2$.*

Proof. Consider any pair of resolutions in $H_1 \cup H_2$ of two genotypes $g, g' \in G_1 \cup G_2$. If g and g' are not both in G_1, nor in G_2, then there is no edge between them in $DG(G)$, meaning that they are independent. Hence, the pair of resolutions is non-conflicting by definition. If $g, g' \in G_1$ or $g, g' \in G_2$, then by Lemma 2, the pair of resolutions is non-conflicting as both H_1 and H_2 define a perfect phylogeny. It follows that all resolutions in $H_1 \cup H_2$ are pairwise non-conflicting, and so again by Lemma 2, $H_1 \cup H_2$ defines a perfect phylogeny. $\square$

In view of lemma 5, every connected component can be resolved individually, and the union of the chosen haplotypes will give the desirable resolution of G.

2.3 (*,1)- and (2,*)-Bounded Cases

We next turn to show how to use the properties of the dependency graph $DG(G)$ to solve various bounded-cases of CPPH. We begin with the simple cases of (1,*)-bounded and (*,2)-bounded genotype matrices.

Lemma 6. *Any pair of genotypes g_1 and g_2 in a (*,1)-bounded genotype matrix are not directly dependent.*

According to Lemma 6, if G is a (*,1)-bounded genotype matrix, then the dependency graph $DG(G)$ has no edges, and its connected components are of size one. The algorithm will choose for H' one pair of haplotypes from every set H_i, $1 \leq i \leq n$, and H' will define a perfect phylogeny according to Lemma 5.

Lemma 7. *If g_1 and g_2 are two genotypes in a (2,*)-bounded genotype matrix, then w.l.o.g. they are not directly dependent.*

Theorem 1. CPPH *with (*,1)- or (2,*)-bounded genotype matrices is polynomial-time solvable.*

2.4 (5,2)-Bounded Case

It is convenient to mark every edge $\{g, g'\}$ in the dependency graph $DG(G)$ with the indices of the 2-columns g and g' share. Observe that in the (*,2)-bounded case, a specific index will appear only on one edge of the dependency graph, since there are no more than two genotypes that have 2 in the same column. Thus, for the (5,2)-bounded case, we have the following important property:

Lemma 8. *If G is a (5,2)-bounded genotype matrix then $DG(G)$ has maximum degree 2.*

The lemma above implies, that if G is a (5,2)-bounded genotype matrix then every connected component in $DG(G)$ is either a path or a cycle. Furthermore, in the first case of the lemma, the connected component is of size 2, and all solutions for it can be determined trivially. We therefore focus on the second case of the lemma, where each edge in a given component is marked with at most 3 indices. We will initially assume that the component is a path with all edges labeled by two indices, and then show how to easily extend our ideas to the case of edges labeled by three indices, and to the case of a cycle.

Consider a connected component G' in $DG(G)$ which is a path comprised of r genotypes $g_1, \ldots, g_r$, where g_1 and g_r are vertices of degree 1, and g_i is connected to g_{i+1}, $1 \leq i < r$, by an edge which is marked with two indices. We partition the internal genotypes of the path G' into three types, depending on the number of possible resolutions available for them. Let $g_i \in G'$ be genotype in G' for some $1 < i < r$, where $\{g_{i-1}, g_i\}$ is labeled with two indices ab in $DG(G)$, $1 \leq a < b \leq m$, and $\{g_i, g_{i+1}\}$ is labeled with two indices cd, $1 \leq c < d \leq m$:

- Genotypes of type I have all four possible resolutions of ab and cd in $\widetilde{H}(g_i)$.
- Genotypes of type II have only three out of four possible resolutions of ab and cd in $\widetilde{H}(g_i)$.
- Genotypes of type III have only two out of four possible resolutions of ab and cd in $\widetilde{H}(g_i)$.

Note that these are the only possible cases, since we assume that any pair of columns label an edge can be resolved either equally or unequally, otherwise there is only one way to resolve it and we can remove this edge from the graph.

We next show how to find a perfect phylogenic resolution in $\widetilde{H}$ for the case of a path. Furthermore, we show that the path represents all legal perfect phylogenic resolutions.

Lemma 9. *In the case of a component which is a path with all edges labeled by two different indices there is always a perfect phylogenic resolution in* $\widetilde{H}$*.*

We next show how to extend the above path algorithm to the case where some edges are labeled with three indices in the (5,2)-bounded case. Consider an edge $\{g, g'\}$ labeled with three indices abc, $1 \le a < b < c \le m$ in $DG(G)$. First we check that $\widetilde{H}(g)$ includes only pairs of haplotypes that have a non-conflicting resolution in $\widetilde{H}(g')$ and vice versa. We do so by checking all possible resolutions for columns abc in $\widetilde{H}(g)$ and $\widetilde{H}(g')$ and deleting the resolutions that appear only in one of those sets. This step ensures that if neither of the sets remains empty (which means there is no solution), then the algorithm will choose for abc a perfect phylogenic resolution. Note that the deletion of haplotype pairs may require deletion of other haplotype pairs in $\widetilde{H}(g_1), .., \widetilde{H}(g_\ell)$. Since that we will repeat step 4 of the preprocessing described in section 2.1. Observe that in the (5,2)-bounded case, the other edges connected to g and g' are labeled with at most two indices (since g and g' have at most five 2s each). According to that there are at most eight possible resolutions available for g and g', which means we can use the same path algorithm from lemma 9 while this time the algorithm may have more resolution options to choose from for the three indices edges.

We are left to show how to extend the path algorithm to the case of a cycle. Consider a cycle of r genotypes. The extension can be easily done by pulling out one edge from the cycle, for example, w.l.o.g., the edge $\{g_r, g_1\}$ what leaves us with a path comprises of r genotypes $g_1, g_2, ..., g_{r-1}, g_r$. We now check what are the possible resolutions for that path according to the way they resolve the columns label the edges $\{g_1, g_2\}$ and $\{g_{r-1}, g_r\}$. Note that there are exactly four types of possible resolutions according to this definition. The way of checking whether there exist any resolution of a specific type, for example the type that resolves the columns label $\{g_1, g_2\}$ and $\{g_{r-1}, g_r\}$ equally, is to run the path algorithm twice, in the first time starting from g_1 by choosing an equal resolution to the columns label $\{g_1, g_2\}$ and continue until reaching an edge with two possible resolutions. The second run will start from g_r and do the same on the opposite direction. If any of those runs reach the end of the path with the wrong resolution that means there is no resolution of the specific type. After knowing what types of resolutions exists, it is only left to check whether the columns label the removed edge $\{g_r, g_1\}$ have a resolution that does not conflict with any of those types.

Theorem 2. CPPH *with (5,2)-bounded genotype matrices is polynomial-time solvable.*

2.5 (3,3)-Bounded Case

In a (3,3)-bounded genotype matrix there are three cases of direct dependency for every genotype g.

1. g is directly dependent with two other genotypes sharing the same 22-columns. In this case any genotype of the three cannot be dependent with any other genotype since it left with at most one 2 to share, and so the corresponding connected component in $DG(G)$ is of size 3.
2. g is directly dependent with three other genotypes. In this case all four genotypes cannot be dependent with any other genotypes since they have at most one 2 to share, and so the corresponding connected component is of size 4.
3. g is directly dependent with exactly one other genotype g', and they have exactly one pair of 22-columns in common. In this case, g' can be directly dependent with another genotype g'', and so forth. The corresponding connected component in this case is a path, where each edge is labeled with two indices.

In the first two cases, all perfect phylogeny solutions can be determine whether the connected component has a prefect phylogeny resolution in $\widetilde{H}$ by simple exhaustive search. In the last case, we know by lemma 9 that the connected component necessarily has a perfect phylogenic resolution in $\widetilde{H}$, and we can use the algorithm described in that lemma for finding an actual solution. Observe that the algorithm will work correctly since in the third case described above, any pair of columns labels at most one edge across the path.

Theorem 3. CPPH *with (3,3)-bounded genotype matrices is polynomial-time solvable.*

3 Constrained Parsimony Haplotyping

We next consider the CPH problem. In [26], Sharan *et al.* showed that PH is fixed parameter tractable (see [7] for a formal definition) when parameterized by the size k of the resolution of G (i.e.there are k distinct haplotypes in the resolution). Here, we show an analogous result for CPH. Our algorithm will perform relatively efficiently (in comparison to brute-force type algorithms) in cases of $\ell << n$. Our approach involves solving a dynamic program to determine whether there is any $H' \subseteq H$ of size $\kappa \leq k$ which resolves G. Throughout the section we use G_i, $1 \leq i \leq \ell$, to denote the subset of genotypes $\{g_1, \dots, g_i\} \subseteq G$.

Probably the first dynamic-programming solution to come to mind, is to compute all possible resolutions $H' \subseteq H$ of G_i from the resolutions of G_{i-1}. However, the number of κ-subsets resolving G_i might be $\Omega(n^k)$, which is too much. We therefore take an alternative route. Instead of computing the actual subsets which resolve G_i, we will compute abstract "blueprints" of these subsets, formally defined in the following definition:

Definition 6 (κ-plan). *Let κ be an integer in $\{1,\ldots,k\}$. A* κ-plan *is a string of length $i \leq \ell$ over the alphabet $\{\{x,y\} \mid 1 \leq i \leq j \leq \kappa\}$.*

Let $H' = \{h_1,\ldots,h_\kappa\}$ be a resolution of $G_i = \{g_1,\ldots,g_i\}$, with some of the haplotypes in H' possibly equal. A κ-plan p is *associated with* H' if when $\{x,y\}$ is the j'th letter in p, $1 \leq x \leq y \leq \kappa$ and $1 \leq j \leq i$, then h_x and h_y resolve g_j. We will say that p is *valid* for G_i if there is a resolution of G_i associated with p. In this way, a valid κ-plan does not describe the actual resolution of G_i, but it does provide all relevant information concerning which genotypes are resolved using the same haplotypes.

Definition 7 ($\mathcal{P}[\kappa,i], P[\kappa,i]$). *Let κ be an integer in $\{1,\ldots,k\}$, and i be an integer in $\{1,\ldots,\ell\}$. We denote by $\mathcal{P}[\kappa,i]$ the set of all κ-plans of length i, and by $P[\kappa,i] \subseteq \mathcal{P}[\kappa,i]$ the set of all valid κ-plans for $G_i = \{g_1,\ldots,g_i\}$.*

Lemma 10. $|P[\kappa,i]| \leq |\mathcal{P}[\kappa,i]| \leq k^{O(k^2)}$ *for any $\kappa \leq k$ and $i \leq \ell$.*

Our algorithm proceeds by computing $P[\kappa,i]$ in increasing values of κ and i. The base-cases of this computation are

1. $P[\kappa,1] = \mathcal{P}[\kappa,1]$ for all $1 \leq \kappa \leq k$, and
2. $P[1,i] = P[2,i] = \emptyset$ for all $2 \leq i \leq \ell$.

Clearly, G can be resolved using $\kappa \leq k$ haplotypes if and only if $G_\ell = G$ has at least one valid κ-plan. Hence, assuming we can correctly compute $P[\kappa,i]$ for all $1 \leq \kappa \leq k$ and $1 \leq i \leq \ell$, the correctness of our algorithm is immediate. What remains to be described is the dynamic-programming step for computing $P[\kappa,i]$.

For this, we will first need to introduce some terminology. Let p be some κ-plan which is valid for G_i, and let $h \in H$ be some haplotype. For a given $x \in \{1,\ldots,\kappa\}$, we say that *the assignment of $h_x = h$ is compatible with p* if there is a resolution $H' = \{h_1,\ldots,h_\kappa\}$ of G_i associated with p such that $h_x = h$. We extend this terminology also for assignments of pairs of haplotypes $h_x = h$ and $h_y = h'$, $h,h' \in H$ and $x \neq y \in \{1,\ldots,\kappa\}$. The dynamic-programming step for computing $P[\kappa,i]$ is as follows:

1. $P[\kappa,i] \leftarrow P[\kappa-1,i]$.
2. For each $p \in P[\kappa-2,i-1]$:
   - Concatenate $\{\kappa,\kappa-1\}$ to the end of p, and add this new κ-plan to $P[\kappa,i]$.
3. For each $h,h' \in H$ resolving g_i, for each $p \in P[\kappa-1,i-1]$, and for each $x \in \{1,\ldots,\kappa-1\}$:
   - Check whether the assignment of $h_x = h$ is compatible with p. If so, concatenate $\{x,\kappa\}$ to the end of p, and add this new κ-plan to $P[\kappa,i]$.
4. For each $h,h' \in H$ resolving g_i, for each $p \in P[\kappa,i-1]$, and for each $x \neq y \in \{1,\ldots,\kappa\}$:
   - Check whether the assignment of $h_x = h$ and $h_y = h'$ is compatible with p. If so, concatenate $\{x,y\}$ to the end of p, and add this new κ-plan to $P[\kappa,i]$.

Note that as we know p is associated with some resolution of G_{i-1}, we can determine in polynomial-time whether assignments are compatible with p. This can be done as follows: Suppose we want to determine whether $h_x = h$ is compatible with p. We mark all positions j, $1 \leq j \leq i-1$, with a letter $\{x, y\}$ in p. For each such position j, we compute $h_y = h'$ from g_j and h. Here, there are three possible outcomes – (i) we have reached a contradiction with a previous assignment, or (ii) we have discovered a new haplotype, or (iii) none of the previous two happens. In the first case we determine incompatibility. In the second case we continue with the checking process. In the third case, since we know that p is a κ-plan for G_{i-1}, we can safely determine compatibility. Checking whether $h_x = h$ and $h_y = y$ is compatible with p is done similarly. The entire process is performed in $O(\kappa)$ rounds, with each round requiring $O(\ell m)$ time, and so its total time complexity is $O(\kappa \ell m) = O(k^3 m)$.

Theorem 4. CPH *parameterized by $k = |H|$ is fixed-parameter tractable.*

References

1. The international hapmap project. Nature 426, 789–796 (2003)
2. Bafna, V., Gusfield, D., Lancia, G., Yooseph, S.: Haplotyping as perfect phylogeny: A direct approach. Journal of Computational Biology 10, 323–340 (2003)
3. Barzuza, T., Beckmann, J.S., Shamir, R., Pe'er, I.: Computational problems in perfect phylogeny haplotyping: Xor-genotypes and tag sNP's. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 14–31. Springer, Heidelberg (2004)
4. Brown, D., Harrower, I.M.: A new integer programming formulation for the pure parsimony problem in haplotype analysis. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 254–265. Springer, Heidelberg (2004)
5. Damaschke, P.: Fast perfect phylogeny haplotype inference. In: Lingas, A., Nilsson, B.J. (eds.) FCT 2003. LNCS, vol. 2751, pp. 183–194. Springer, Heidelberg (2003)
6. Ding, Z., Filkov, V., Gusfield, D.: A linear-time algorithm for the perfect phylogeny haplotyping (pph) problem. Journal of Computational Biology 13, 522–553 (2006)
7. Downey, R., Fellows, M.: Parameterized Complexity. Springer, Heidelberg (1999)
8. Eskin, E., Halperin, E., Karp, R.: Efficient reconstruction of haplotype structure via perfect phylogeny. Journal of Bioinformatics and Computational Biology 1, 1–20 (2003)
9. Gramm, J., Nierhoff, T., Sharan, R., Tantau, T.: On the complexity of haplotyping via perfect phylogeny. In: Proceedings of RECOMB Satellite Workshop on Computational Methods for SNPs and Haplotypes (2004)
10. Gramm, J., Nierhoff, T., Sharan, R., Tantau, T.: Haplotyping with missing data via perfect path phylogenies. Discrete Applied Mathematics 155, 788–805 (2007)
11. Greenspan, G.: Geiger D. Model-based inference of haplotype block variation. In: Research in Computational Molecular Biology (RECOMB 2003), pp. 131–137 (2003)
12. Gusfield, D.: Haplotyping as perfect phylogeny: Conceptual framework and efficient solutions (extended abstract). In: Proceedings of RECOMB, pp. 166–175 (2002)
13. Gusfield, D.: Haplotype inference by pure parsimony. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 144–155. Springer, Heidelberg (2003)

14. Gusfield, D., Orzack, S.H.: Haplotype inference. In: Aluru, S. (ed.) CRC Handbook on Bioinformatics (2005)
15. Gusfield, D., Song, Y., Wu, Y.: Algorithms for Imperfect Phylogeny Haplotyping (IPPH) with a Single Homoplasy or Recombination Event. In: Casadio, R., Myers, G. (eds.) WABI 2005. LNCS (LNBI), vol. 3692, pp. 152–164. Springer, Heidelberg (2005)
16. Halldorsson, B., Bafna, V., Edwards, N., Lipert, R., Yooseph, S., Istrail, S.: A survey of computational methods for determining haplotypes. In: Proceedings of RECOMB Satellite on Computational Methods for SNPs and Haplotype Inference, pp. 26–47 (2003)
17. Halperin, E., Eskin, E.: Haplotype reconstruction from genotype data using imperfect phylogeny. Bioinformatics 20, 1842–1849 (2004)
18. Halperin, E., Karp, R.M.: Perfect phylogeny and haplotype assignment. In: Proceedings of RECOMB, pp. 10–19 (2004)
19. Hudson, R.: Gene genealogies and the coalescent process. Oxsford Survey of Evolutionary Biology 7, 1–44 (1990)
20. van Iersel, L., Keijsper, J., Kelk, S., Stougie, L.: Beaches of islands of tractability: Algorithms for parsimony and minimum perfect phylogeny haplotyping problems. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 80–91. Springer, Heidelberg (2006)
21. Kimmel, G., Shamir, R.: The incomplete perfect phylogeny haplotype problem. Journal of Bioinformatics and Comutational Biology 3, 359–384 (2005)
22. Lancia, G., Pinotti, C., Rizzi, R.: Haplotyping population by pure parsimony: Complexity, exact and approximation algorithms. INFORMS Journal on Computing, special issue on Comutational Biology 16, 348–359 (2004)
23. Lancia, G., Rizzi, R.: A polynomial case of the parsimony haplotyping problem. Operations Research Letters 34, 289–295 (2006)
24. Rastas, P., Koivisto, M., Mannila, H., Ukkonen, E.: A hidden markov technique for haplotype reconstruction. In: Casadio, R., Myers, G. (eds.) WABI 2005. LNCS (LNBI), vol. 3692, pp. 140–151. Springer, Heidelberg (2005)
25. Satya, R.V., Mukherjee, A.: An optimal algorithm for perfect phylogeny haplotyping. Journal of Computational Biology 13(4), 897–928 (2006)
26. Sharan, R., Halldorsson, B., Istrail, S.: Islands of tractability for parsimony haplotyping. IEEE/ACM Transactions on Computational Biology and Bioinformatics 3, 303–311 (2006)
27. Tavare, S.: Calibrating the clock: Using stochastic process to measure the rate of evolution. In: Lander, E., Waterman, M. (eds.) Calculating the Secrets of Life (1995)
28. Wang, L., Xu, L.: Haplotype inference by maximum parsimony. Bioinformatics 19, 1773–1780 (2003)

Efficient Inference of Haplotypes from Genotypes on a Pedigree with Mutations and Missing Alleles (Extented Abstract)

Wei-Bung Wang and Tao Jiang

Department of Computer Science, University of California, Riverside, CA 92506, USA
`weiw@cs.ucr.edu`, `jiang@cs.ucr.edu`

Abstract. Driven by the international HapMap project, the haplotype inference problem has become an important topic in the computational biology community. In this paper, we study how to efficiently infer haplotypes from genotypes of related individuals as given by a pedigree. Our assumption is that the input pedigree data may contain *de novo* mutations and missing alleles but is free of genotyping errors and recombinants, which is usually true for tightly linked markers. We formulate the problem as a combinatorial optimization problem, called the *minimum mutation haplotype configuration* (MMHC) problem, where we seek haplotypes consistent with the given genotypes that incur no recombinants and require the minimum number of mutations. This extends the well studied *zero-recombinant haplotype configuration* (ZRHC) problem. Although ZRHC is polynomial-time solvable, MMHC is NP-hard. We construct an *integer linear program* (ILP) for MMHC using the system of linear equations over the field $F(2)$ that has been developed recently to solve ZRHC. Since the number of constraints in the ILP is large (exponentially large in the general case), we present an incremental approach for solving the ILP where we gradually add the constraints to a standard ILP solver until a feasible haplotype configuration is found. Our preliminary experiments on simulated data demonstrate that the method is very efficient on large pedigrees and can infer haplotypes very accurately as well as recover most of the mutations and missing alleles correctly.

1 Introduction

Human beings have been fighting against diseases such as cancer, stroke, heart disease, asthma, depression, and schizophrenia for decades. It is believed that many of these diseases are caused by genetic factors. *Gene mapping*, which attempts to establish connections between diseases and some specific genetic variations, is a very important and active area of genetics. More specifically, it aims at locating genes of interest (*e.g.* genes responsible for certain diseases) relative to genetic markers (such as microsatellites and *single nucleotide polymorphisms*, or SNPs) on chromosomes. A set of genetic markers and their positions (called *marker loci*) define a *genetic map* of chromosomes. In *diploid* organisms like human, chromosomes (other than sex chromosomes) form pairs. Each pair of chromosomes consists of a *paternal* chromosome inherited from the father and a *maternal* chromosome inherited from the mother. Hence, each genetic marker on a pair of chromosomes occurs at the same location of both paternal and maternal chromosomes. However, the marker may have different states (called *alleles*) on the two chromosomes. The set of its two alleles is called the *genotype* of the marker and the assignment of the two alleles to the paternal and maternal chromosomes is called the *haplotype* (or *phase*) of the marker. The haplotype information of genetic markers is of tremendous value to gene mapping and other genetic analyses (such as linkage analysis) because it gives a more accurate description of the inheritance process than the genotype information. Its importance can also be seen from the international HapMap project launched in 2002 [19]. Since genotype data instead of haplotype data are routinely collected in practice, especially in large-scale sequencing projects, due to cost considerations, efficient and accurate computational methods for the inference of haplotypes from genotypes over a set of marker loci, which is also commonly referred to as *phasing*, have been extensively studied in the literature. See [11] for a recent survey on these methods as well as the basic concepts involved in haplotype inference.

The existing computational methods for haplotype inference can be divided into three groups according to the genotype data that they deal with: methods for *population* data involving unrelated individuals (see *e.g.* [6,13,17]), methods for *pedigree* data consisting of individuals (typically from an extended family) that are related by the parent-child relationship (see *e.g.* [1,8,9,10,15,16,21,23]), and methods for *pooled samples* (see *e.g.* [20,22]). The methods for population data usually consider tightly linked markers that may involve mutations but no recombinants, while the methods for pedigree data usually assume that the data may have zero or few recombinants but is free of mutations (*i.e.* the Mendelian law of inheritance holds). Here, we are interested in only pedigree data.

Some real pedigree data may actually contain mutations. In particular, a *de novo mutation* is a mutation that is present for the first time in a family member as a result of a mutation in a germ cell (egg or sperm) of one of the parents or in the fertilized egg itself. It has been found that the detection and analysis of mutations in a pedigree could provide a good alternative for some genetic variation research [3,5,14]. In fact, Ellegren [5] has stated that "To reveal the mutational contribution to overall genetic variability, the most straightforward and conclusive way is the direct detection of mutation events in pedigree genotyping." However, *de novo* mutations violate the Mendelian law of inheritance, and hence pedigree data with such mutations cannot be properly handled by the above common haplotype inference methods. When these methods are faced with data with mutations, they typically treat the loci involving mutations as genotyping errors and delete such loci. Very few haplotype inference methods in the literature deal with pedigree data that contain mutations (one such method is a genetic algorithm in [18]).

In this paper, we study haplotype inference on pedigree data on tightly linked markers that have no recombinants but may contain a small number of *de novo* mutations (or simply, mutations). Since mutation is a rare event, we formulate the problem as a combinatorial optimization problem, called the *minimum mutation haplotype configuration* (MMHC) problem, where we look for a haplotype solution consistent with the given genotype data that incur no recombinants and require the minimum number of mutations. Our hypothesis is a solution with the minimum number of mutations is likely the true solution. Moreover, we are only interested in solutions where each locus has at most one mutation in the pedigree. This restriction is reasonable given Kimura's infinite-site model *et al.* [7] which suggests that the probability of multiple mutations at the same locus is low enough to be negligible. This extends the well studied *zero-recombinant haplotype configuration* (ZRHC) problem where we try to find a consistent haplotype solution incurring no recombinants or mutations. Although ZRHC is polynomial-time solvable [9], we can prove that MMHC is NP-hard by a reduction from NAE-3SAT (the proof is omitted in this extended abstract). We construct an *integer linear program* (ILP) for MMHC using the system of linear equations over the field $F(2)$ that has been developed in [9,12,21] for solving ZRHC in almost linear time. Since the number of constraints in the ILP is quite large (exponentially large in general) when the input pedigree is large, we present an incremental approach for solving the ILP.

An outline of our incremental approach is as follows. Given a pedigree data, we set up a system of linear equations over $F(2)$ introduced in [12,21] for ZRHC, but conditional on mutations. We convert the linear system to an ILP instance for MMHC where the constraints generally describe the relation between the equations and mutations. A small set of the constraints in the ILP are identified as the *core* constraints, and a standard ILP solver GLPK (the GNU Linear Programming Kit from `http://www.gnu.org/softward/glpk`) is invoked on the partial ILP instance with only the core constraints. The ILP solution describes an assignment of mutations in the pedigree which can be used to remove the conditions in the linear system. By using Gaussian elimination, we can check if the linear system is consistent. If it is consistent, a haplotype configuration (with the minimum number of mutations) is returned. Otherwise, we find the inconsistent equations and add some new constraints to the core to force their consistency. This process is repeated until an ILP solution that satisfies its corresponding linear system has been found. Note that, the incremental approach to solving the ILP is crucial here because the ILP instance cannot be efficiently and explicitly constructed as its number of constraints grows exponentially in the pedigree size in general. Also note that, with the advance in sequencing technology, larger and larger pedigrees are being genotyped and analyzed in practice. For example, in [2,4], haplotype inference was performed on pedigrees of sizes 368 and 1149, respectively.

We have implemented the algorithm and tested it on pedigree data that were simulated with random mutations and missing alleles. (Real pedigree data often have up to 20% missing alleles.) The experimental results demonstrate that our method can infer haplotypes with a very high accuracy. It can also detect most of the mutations and impute most of the missing alleles correctly. Moreover, it is found that the algorithm usually terminates after a small number of iterations without ever having to invoking ILP solver on the complete ILP instance consisting of all the constraints. As a comparison, we have also considered the straightforward approach for solving the ILP with all the constraints considered at once on binary tree pedigrees (*i.e.* each pair of parents has only one child). The ILP instance can be efficiently constructed for binary tree pedigrees. It is found that our algorithm is much faster than the straightforward approach.

The rest of the paper is organized as follows. In Section 2, we incorporate mutations into the system of linear equations introduced in [12,21] for ZRHC to obtain a system of conditional linear equations for MMHC. Section 3 describes the ILP formulation for MMHC, and the incremental approach for solving the ILP. In Section 4, we discuss the implementation of the algorithm and test its performance on some simulated pedigree data with random mutations and missing alleles. Section 5 concludes the paper with a few remarks.

2 A System of Conditional Linear Equations for MMHC

We review the system of linear equations over $F(2)$ introduced in [12,21] for solving ZRHC and extend the system to take into account mutations.

2.1 The Linear System

Let n denote the number of the individuals in the input pedigree and m the number of marker loci of each individual. For simplicity, we assume in this paper that all alleles are bi-allelic (denoted as 0 or 1) and the input pedigree is free of mating loops (and thus a tree pedigrees). Tree pedigrees are very common among human pedigrees. Our techniques can be extended to general pedigrees. The genotype of individual j is denoted as a ternary vector $\mathbf{g}_j$ whose kth entry $g_j[k]$ represents the genotype at locus k of individual j as follows:

$$\begin{cases} g_j[k] = 0 & \text{if both alleles are 0's} \\ g_j[k] = 1 & \text{if both alleles are 1's} \\ g_j[k] = 2 & \text{if the locus is heterozygous} \end{cases} \tag{1}$$

The value of $g_j[k]$ is *unknown* if the alleles are missing. For each locus k of individual j, we define a binary variable $p_j[k]$ over $F(2)$ to indicate the paternal allele at the locus:

$$\begin{cases} p_j[k] = 0 & \text{if } g_j[k] = 0 \\ p_j[k] = 1 & \text{if } g_j[k] = 1 \\ p_j[k] = 0 & \text{if } g_j[k] = 2 \text{ and allele 0 is paternal} \\ p_j[k] = 1 & \text{if } g_j[k] = 2 \text{ and allele 1 is paternal} \end{cases} \tag{2}$$

In other words, the binary vector $\mathbf{p}_j$ represents the paternal haplotype of individual j. To represent the maternal haplotype, we need another binary vector $\mathbf{w}_j$ to indicate if each locus of individual j is heterozygous. That is, $w_j[k] = 0$ if

$g_j[k] = 0$ or 1, and $w_j[k] = 1$ if $g_j[k] = 2$. Clearly, the sum $\mathbf{p}_j + \mathbf{w}_j$ (over $F(2)$) represents the maternal haplotype of individual j.

Suppose that individual i is a parent of individual j. To unify the representation of the haplotype that j inherited from i, define a binary vector $\mathbf{d}_{i,j}$ as follows: $\mathbf{d}_{i,j} = 0$ if i is j's father and $\mathbf{d}_{i,j} = \mathbf{w}_j$ if i is j's mother. Therefore, $\mathbf{p}_j + \mathbf{d}_{i,j}$ represents the haplotype that j got from i. Define $h_{i,j} = 0$ if $\mathbf{p}_j + \mathbf{d}_{i,j}$ is i's paternal haplotype and $h_{i,j} = 1$ otherwise. Then $\mathbf{p}_i + h_{i,j} \cdot \mathbf{w}_i$ represents the haplotype that i passed to j. The binary variables $h_{i,j}$ thus fully describe the inheritance pattern in an ZRHC instance. Finally, define $\mu_{i,j}[k] = 1$ if the there is a mutation at locus k when i passes the haplotype $\mathbf{p}_i + h_{i,j} \cdot \mathbf{w}_i$ to j, and $\mu_{i,j}[k] = 0$ otherwise. For technical reasons, we view $\mu_{i,j}[k]$ as an integer from $\mathbb{Z}$ instead of $F(2)$. For convenience, we make these three vectors symmetric by defining $\mathbf{d}_{j,i} = \mathbf{d}_{i,j}$, $h_{j,i} = h_{i,j}$, and $\mu_{j,i} = \mu_{i,j}$. Using these notations, we can derive a *conditional equation* over $F(2)$:

$$\begin{cases} p_i[k] + h_{i,j} \cdot w_i[k] = p_j[k] + d_{i,j}[k] & \text{if } \mu_{i,j}[k] = 0 \\ p_i[k] + h_{i,j} \cdot w_i[k] = p_j[k] + d_{i,j}[k] + 1 & \text{if } \mu_{i,j}[k] = 1 \end{cases} \tag{3}$$

Since we assume that each locus has at most one mutation in the pedigree,

$$0 \leq \sum_{i,j} \mu_{i,j}[k] \leq 1 \quad \forall k \tag{4}$$

Note that the summation is over $\mathbb{Z}$ instead of $F(2)$. Hence, the MMHC problem can be formally defined as follows. Given an input pedigree and genotype data $\mathbf{g}_j$ for each individual j, find a solution to each $\mathbf{p}_j$, $h_{i,j}$ and $\mu_{i,j}$ that satisfies all the (conditional) constraints in Equations (3) and (4) and minimizes the sum $\sum_{i,j,k} \mu_{i,j}[k]$.

2.2 Pre-Determined Variables

The above linear system has $O(mn)$ variables and equations. As in [12,21], we can convert the system to an equivalent linear system involving only the h-variables which is much smaller (there are only $O(n)$ h-variables). This requires us to *pre-determine* the values of some p-variables. The situation is complicated a little bit by the presence of the μ-variables.

Let us consider a p-variable $p_j[k]$ where the marker of individual j at locus k is not missing, and several scenarios.

1. $g_j[k] \neq 2$. By Equation (2), $p_j[k] = g_j[k]$. In this case, $p_j[k]$ is *pre-determined*. We will refer to $p_j[k]$ as the *intended p-value* of the locus, denoted as $v(j,k) = p_j[k]$.
2. $g_j[k] = 2$ and exactly one parent, denoted as i, is homozygous at locus k. See Figure 1(a). We have $w_i[k] = 0$ by definition. According to Equation (3), $p_j[k]$ is known if and only if $\mu_{i,j}[k]$ is known. We say that $p_j[k]$ is *semi-determined* in this case. We also define $\mu_{i,j}[k]$ as the *anchor* of $p_j[k]$ and denote $a(j,k) = \{\mu_{i,j}[k]\}$. Since the value of $p_j[k]$ on the condition $\mu_{i,j}[k] = 0$ is preferred, we denote $v(j,k) = g_i[k] + d_{i,j}[k]$.

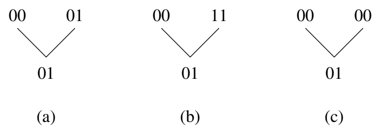


Fig. 1. Determining a p-variable. Consider the p-value of the child in the trio. (a) It equals 0 as long as there is no mutation from the father and it is semi-determined. (b) It equals 0 and there cannot be any mutation. It is pre-determined. (c) It is undetermined but there must be a mutation. It is doubly-determined.

3. $g_j[k] = 2$, both parents i_1 and i_2 of j are homozygous at locus k, and $g_{i_1}[k] \neq g_{i_2}[k]$. See Figure 1(b). Since each locus has at most one mutation, $\mu_{i_1,j}[k]$ and $\mu_{i_2,j}[k]$ cannot both be 1. Hence, $\mu_{i_1,j}[k] = \mu_{i_2,j}[k] = 0$. In this case, $p_j[k]$ is pre-determined, and we denote $v(j,k) = p_{i_1}[k] + d_{i_1,j}[k]$.
4. $g_j[k] = 2$, both parents i_1 and i_2 are homozygous at locus k, and $g_{i_1}[k] = g_{i_2}[k]$. See Figure 1(c). In this case, one of $\mu_{i_1,j}[k]$ and $\mu_{i_2,j}[k]$ equals 1 and the other 0. Thus, $p_j[k]$ has two anchors: $a_1(j,k) = \{\mu_{i_1,j}[k]\}$ and $a_2(j,k) = \{\mu_{i_2,j}[k]\}$. Each anchor gives rise to a preferred value for $p_j[k]$, $v_1(j,k) = p_{i_1}[k] + d_{i_1,j}[k]$ and $v_2(j,k) = p_{i_2}[k] + d_{i_2,j}[k]$, respectively. In this case we call $p_j[k]$ *doubly-determined*.
5. All other cases. The variable $p_j[k]$ is *undetermined* and the variable $v(j,k)$ is undefined.

If a $p_j[k]$ is pre-determined or undetermined, we define $a(j,k) = \emptyset$. Similarly, we might be able to pre-determine μ-variable $\mu_{i,j}[k]$ in some cases.

1. $g_i[k] = g_j[k] \neq 2$. Since the top equation in Equation (3) holds, we let $\mu_{i,j}[k] = 0$ and it is pre-determined.
2. $g_i[k] \neq g_j[k]$ and both loci are homozygous. Since the bottom equation in Equation (3) holds, we set $\mu_{i,j}[k] = 1$. This μ-variable is pre-determined. Moreover, all the other μ-variables at locus k must equal 0 and are pre-determined too.
3. Some p-variable at locus k is doubly determined. All the μ-variables at locus k other than this p-variable's anchors must equal 0 and are thus pre-determined.
4. All other cases. The variable $\mu_{i,j}[k]$ stays undetermined.

2.3 A More Compact Linear System

Following [12,21], we can set up a linear system in terms of the h-variables. The idea is to consider paths in the pedigree connecting individuals with pre/semi/doubly-determined p-variables and derive (conditional) equality constraints on the h-variables on such paths based on Equation (3).

Consider a locus k and a path $j_0, j_1, \ldots, j_r$ in the input (tree) pedigree, where individuals j_i and j_{i+1} have the parent-child relationship. Suppose that $p_{j_0}[k]$ and $p_{j_r}[k]$ are pre-determined, semi-determined or doubly-determined, and

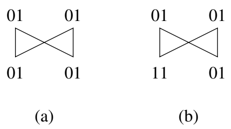


Fig. 2. Two possible cycle constraints from a local cycle. (a) The sum of the four h-variables is 0. (b) The sum of the four h-variables is 1.

$g_{j_1}[k] = \cdots = g_{j_{r-1}}[k] = 2$. We call the path $j_0, j_1, \ldots, j_r$ an *all-heterozygous path* at locus k. If $p_{j_0}[k]$ and $p_{j_r}[k]$ are pre-determined or semi-determined, we define a *path constraint* connecting j_0 and j_r:

$$v(j_0,k) + v(j_r,k) + \sum_{i=0}^{r-1} \left(h_{j_i,j_{i+1}} + d_{j_i,j_{i+1}}[k]\right) = 0$$
$$\text{if all elements in } a(j_0,k) \cup a(j_r,k) \cup \bigcup_{i=0}^{r-1} \{\mu_{j_i,j_{i+1}}[k]\} \text{ equal } 0 \qquad (5)$$

If we denote $\mathcal{M} = a(j_0,k) \cup a(j_r,k) \cup \bigcup_{i=0}^{r-1}\{\mu_{j_i,j_{i+1}}[k]\}$, $\mathcal{H} = \bigcup_{i=0}^{r-1}\{h_{j_i,j_{i+1}}\}$, and $c = v(j_0,k) + v(j_r,k) + \sum_{i=0}^{r-1} d_{j_i,j_{i+1}}[k]$, then the path constraint can also be represented by the triple $(\mathcal{H}, \mathcal{M}, c)$ which denotes:

$$\sum_{h_{i,j} \in \mathcal{H}} h_{i,j} = c \qquad \text{iff } \mu_{i,j}[k] = 0 \quad \forall \mu_{i,j}[k] \in \mathcal{M} \qquad (6)$$

If j_0 or j_r is doubly-determined, we can construct two path constraints in the same way: one using $v_1(\cdot)$ and $a_1(\cdot)$ and the other using $v_2(\cdot)$ and $a_2(\cdot)$.

Consider a *local cycle* consisting of father i_1, mother i_2, and two adjacent children j_1, j_2. If both parents are heterozygous at locus k, we can obtain four conditional equations from Equation (3) by replacing i with i_1, i_2, and j with j_1, j_2. (See Figure 2.) The summation of these conditional equations forms a *cycle constraint*:

$$\begin{aligned} h_{i_1,j_1} + h_{i_1,j_2} + h_{i_2,j_1} + h_{i_2,j_2} &= d_{i_1,j_1}[k] + d_{i_1,j_2}[k] + d_{i_2,j_1}[k] + d_{i_2,j_2}[k] \\ &= w_{j_1}[k] + w_{j_2}[k] \\ \text{iff } \mu_{i_1,j_1}[k] = \mu_{i_1,j_2}[k] &= \mu_{i_2,j_1}[k] = \mu_{i_2,j_2}[k] = 0 \end{aligned} \qquad (7)$$

This constraint will also be denoted as $(\mathcal{H}, \mathcal{M}, c)$ where $\mathcal{H} = \{h_{i_1,j_1}, h_{i_1,j_2}, h_{i_2,j_1}, h_{i_2,j_2}\}$, $\mathcal{M} = \{\mu_{i_1,j_1}[k], \mu_{i_1,j_2}[k], \mu_{i_2,j_1}[k], \mu_{i_2,j_2}[k]\}$, and $c = w_{j_1}[k] + w_{j_2}[k]$.

If both parents are homozygous at locus k, then the p-variables of both children must be pre-determined or doubly-determined. However, the two children are not connected by any all-heterozygous path and thus no path constraint is derived. On the other hand, if exactly one parent is heterozygous at locus k, then both children are semi-determined and there is a path constraint between the two children through the heterozygous parent.

For each locus and every pair of pre/semi/doubly-determined p-variables connected by an all-heterozygous path, we construct a path constraint (or two if one of the p-variables is doubly-determined, or four if both p-variables are doubly-determined) as above. Since the pedigree is a tree, the number of such path constraints is at most $O(mn)$. Similarly, for each locus and local cycle, if both parents are heterozygous at the locus, we construct a cycle constraint as above. The number of such cycle constraints is also bounded by $O(mn)$. Let $\mathcal{E}$ denote the set of these constraints.

The results in [12] show that the linear system formed by the above constraints (without the conditions) in terms of the h-variables is equivalent to the linear system defined by Equation (3) (without the conditions) in terms of the h- and p-variables. In other words, a feasible solution to the h-variable can be extended to a feasible solution to both the h- and p-variables. It is easy to see that the same equivalence holds with the conditions.

Note that, loci with missing alleles could be included in the linear system in Equation (3) (as p-variables). However, they are excluded from the above path/cycle constraints on h-variables. Some of the missing alleles will be imputed using Equation (3) after the h-variables are determined.

3 The ILP for MMHC and Incremental Approach

We construct an ILP for MMHC based on the above linear system in h-variables. Recall that the objective of the ILP is

$$\text{Minimize} \quad \sum_{i,j,k} \mu_{i,j}[k]. \tag{8}$$

We give all the constraints of the ILP in Sections 3.1 and 3.2. Section 3.3 presents more details of the incremental approach to solving the ILP. In Section 3.4, we describe how to obtain a solution for MMHC after solving the ILP (and the linear system) and deal with missing alleles.

3.1 The Core Constraints

All the constraints in Equation (4) are core constraints of the ILP. For each path/cycle constraint $(\mathcal{H}, \mathcal{M}, c)$ in $\mathcal{E}$, we introduce an *equation variable*:

$$E_{\mathcal{H}} = \sum_{h_{i,j} \in \mathcal{H}} h_{i,j} \tag{9}$$

We then add an *equation constraint* for each $(\mathcal{H}, \mathcal{M}, c)$:

$$\begin{cases} E_{\mathcal{H}} - \sum_{\mu_{i,j}[k] \in \mathcal{M}} \mu_{i,j}[k] = 0 & \text{if } c = 0 \\ E_{\mathcal{H}} + \sum_{\mu_{i,j}[k] \in \mathcal{M}} \mu_{i,j}[k] = 1 & \text{if } c = 1 \end{cases} \tag{10}$$

In other words, either the linear equation in $(\mathcal{H}, \mathcal{M}, c)$ holds, or there is exactly one mutation in $\mathcal{M}$. Therefore, the core constraints of the ILP include all the constraints in Equations (4), and (10). The number of these core constraints is clearly bounded by $O(mn)$.

3.2 Consistency Constraints

Now we need some constraints to make sure that the assignment of the equation variables are consistent with each other. Consider, for example, three sets of h-variables $\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$ that appear in the linear system such that $\mathcal{H}_1 \triangle \mathcal{H}_2 \triangle \mathcal{H}_3 = \emptyset$. (Here, $\triangle$ is the symmetric difference operator.) If $E_{\mathcal{H}_1} = 0$ and $E_{\mathcal{H}_2} = 0$, which are equivalent to $\sum_{h_{i,j} \in \mathcal{H}_1} h_{i,j} = 0$ and $\sum_{h_{i,j} \in \mathcal{H}_2} h_{i,j} = 0$, then we must have $\sum_{h_{i,j} \in \mathcal{H}_3} h_{i,j} = \sum_{h_{i,j} \in \mathcal{H}_1} h_{i,j} + \sum_{h_{i,j} \in \mathcal{H}_2} h_{i,j} = 0$, or equivalently $E_{\mathcal{H}_3} = 0$. The sum of $E_{\mathcal{H}_1}, E_{\mathcal{H}_2}, E_{\mathcal{H}_3}$ must be even. To guarantee such a relation among the three equation variables, we need include the following *consistency constraints*:

$$C(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3) : \begin{cases} E_{\mathcal{H}_1} + E_{\mathcal{H}_2} + (1 - E_{\mathcal{H}_3}) \geq 1 \\ E_{\mathcal{H}_1} + (1 - E_{\mathcal{H}_2}) + E_{\mathcal{H}_3} \geq 1 \\ (1 - E_{\mathcal{H}_1}) + E_{\mathcal{H}_2} + E_{\mathcal{H}_3} \geq 1 \\ (1 - E_{\mathcal{H}_1}) + (1 - E_{\mathcal{H}_2}) + (1 - E_{\mathcal{H}_3}) \geq 1 \end{cases} \tag{11}$$

These constraints ensure that $(E_{\mathcal{H}_1}, E_{\mathcal{H}_2}, E_{\mathcal{H}_3}) \neq (0,0,1), (0,1,0), (1,0,0), (1,1,1)$, respectively. Therefore, illogical combinations of $E_{\mathcal{H}_1}, E_{\mathcal{H}_2}, E_{\mathcal{H}_3}$ are prohibited, and only legitimate combinations are allowed in a feasible solution.

In general, suppose that $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_r$ is any collection of sets of h-variables that appear in the linear system such that $\mathcal{H}_1 \triangle \mathcal{H}_2 \triangle \cdots \triangle \mathcal{H}_r = \emptyset$. To construct the consistency constraints for their corresponding equation variables, we introduce new variables $S_i = \triangle_{i=1}^{j} \mathcal{H}_i$ and their corresponding variables E_{S_i}. We then construct a series of consistency constraints:

$$C(\mathcal{H}_1, \ldots, \mathcal{H}_r) = C(\mathcal{H}_1, \mathcal{H}_2, S_2) \cup C(S_{r-2}, \mathcal{H}_{r-1}, \mathcal{H}_r) \cup \bigcup_{i=3}^{r-2} C(S_{i-1}, \mathcal{H}_i, S_i) \tag{12}$$

The core constraints and consistency constraints form the complete ILP instance. Note that the number of consistency constraints is generally exponential in n. The following lemma states that these constraints are sufficient for MMHI. Its proof is omitted in this extended abstract.

Lemma 1. *Consider a feasible solution to the (complete) ILP defined above. We can convert the conditional linear system in Section 2.3 to an unconditional linear system using the values of the equation variables in the solution. The linear system must be consistent.*

3.3 The Incremental Approach

Since the complete ILP instance cannot be efficiently constructed in general, we start from an incomplete ILP instance with only the core constraints (no consistency constraints). A standard ILP solver GLPK is invoked to find a solution to the equation variables $E_{\mathcal{H}}$. The equation variable values specifies a set of (unconditional) linear equations from the conditional linear equations in $\mathcal{E}$. We can solve the this system of linear equations by applying Gaussian elimination. However, the linear system may be inconsistent, *i.e.*, there may be a set of equation variables $E_{\mathcal{H}_1}, E_{\mathcal{H}_2}, \ldots, E_{\mathcal{H}_r}$ such that $\triangle_{i=1}^{r}\mathcal{H}_i = \emptyset$ but $\sum_{i=1}^{r} E_{\mathcal{H}_i}$ is odd determined by GLPK. When such an inconsistency occurs, there must be a subset of equations $\left\{\sum_{h\in\mathcal{H}_i} h = c_i\right\}_{i=1}^{r}$ such that $\sum_{i=1}^{r} c_i = 1$ but $\sum_{i=1}^{r}\sum_{h\in\mathcal{H}_r} h = 0$. Hence $\triangle_{i=1}^{r}\mathcal{H}_i = \emptyset$, and we add the consistency constraints shown in Equation (12) to the ILP instance. We then invoke GLPK again. This process is iterated until a solution is found to yield a consistent system of linear equations.

Although in theory this process may take many iterations, more than 95% of the time in our experiment a consistent solution was found in the very first iteration using only the core constraints. Moreover, the process never took more than three iterations in our experiment. This observation can be explained as follows. For each equation constraint in Equation (10), the ILP solver GLPK tends to assign c to the variable $E_{\mathcal{H}}$ given $(\mathcal{M}, \mathcal{H}, c)$ to minimize the number of mutations, if this assignment does not result in conflicting equations. Since the number of mutations is small, most equations should indeed hold. In addition, we usually have a lot of pre-determined μ-variables, which could force GLPK to assign the other variables correctly.

3.4 Phasing, Missing Allele Imputation, and Mutation Detection

Once a consistent (unconditional) linear system is found, solving the system by Gaussian elimination assigns the values of all h-variables. GLPK also assigns the values of all μ-variables in the last iteration. Therefore, we can resolve the p-variables by using the *propagation algorithm* in [12,21]. The basic idea is to propagate known (*i.e.* pre/semi/doubly-determined) p-variable values to undetermined p-variables along the edges in the pedigree by repeatedly applying Equation (3). The p-variables that are left unresolved by the propagation algorithm will be deemed as free in the solution. Note that, the resolved p-variables could allow us to impute missing alleles at some loci (by possibly using some ancestral p-variable and relevant h-variables if necessary), although perhaps not at all loci.

If there are no missing alleles, then the above would produce a consistent solution to the MMHC instance. However, the presence of missing alleles may cause conflict between the assigned values of the μ-variables and those of the

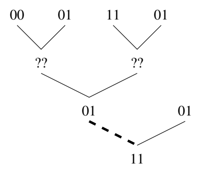


Fig. 3. Missing alleles may prevent us from obtaining path/cycle constraints. In the figure, if there were no missing data, there should have been two path constraints through the dotted line. The μ-variable on the dotted line is free because the two path constraints are not included in the ILP instance.

p-variables and h-variables. This is because some μ-variables do not appear in any conditional equation. These μ-variables only appear in the objective function and the constraints in Equation (4). Let us call this type of μ-variables *free.* (See Figure 3 for an example free μ-variable.) Clearly, the free μ-variables were set to 0 by GLPK to minimize the objective function. This assignment could be in conflict with the p-variable and h-variable values, because their associated path/cycle constraints were not included in the ILP instance. We will try to fix the problem by re-evaluating the free μ-variables using the determined p-variables and h-variables and Equation (3). For any free μ-variable in conflict, we change its value to 1 (which incurs a new mutation).

However, some of these changes might be incorrect (or redundant), and such incorrect changes may potentially lead to other conflicts with the p-variable and h-variable values. When a change leads to more conflicts, we know for sure that the change is wrong (because there can be at most one mutation at the same locus), as stated in the following lemma whose proof is omitted in this extended abstract.

Lemma 2. *If assigning* $\mu_{i_1,j_1}[k]{=}1$ *leads to another conflict that forces* $\mu_{i_2,j_2}[k]{=}1$*, then both* $\mu_{i_1,j_1}[k]$ *and* $\mu_{i_2,j_2}[k]$ *should equal* 0*.*

Whenever we find two mutations at the same locus, we force their corresponding μ-variables to 0 in the ILP instance (by adding two new constraints), and run GLPK and the propagation algorithm again. Note that, these two μ-variables are no longer viewed as free since they now appear in some constraints in the ILP instance. This process is repeated until all μ-variable values are consistent with the p-variable and h-variable values.

4 Experimental Results

We have implemented our algorithm in C, denoted as MMPhase. A detailed pseudocode of MMPhase is omitted in this extended abstract and will given in the full paper. In this section, we test MMPhase on pedigree data with randomly simulated genotypes, mutations and missing alleles to perform an empirical evaluation of its performance and efficiency. We also compare the speed of MMPhase with that of the straightforward method for solving the MMHC ILP (*i.e.*, running GLPK on all the constraints in a single iteration).

We first compare the speeds of MMPhase and the straightforward method. Since the number of consistency constraints is exponential in the pedigree size n and locus number m in general (even for trees), we implement the straightforward method only for binary trees. When the pedigree is a binary tree, we do not have cycle constraints. For each path constraint along the path between j_1 and j_2, let $\mathcal{H}_1$ be the set of the h-variables on the path from the root of the binary tree to j_1, $\mathcal{H}_2$ the set of the h-variables on the path from the root to j_2, and $\mathcal{H}_3$ the set of h-variables on the path from j_1 to j_2. We put the consistency constraint $C(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3)$ into the ILP instance. This will provide a sufficient set of consistency constraints which will guarantee a feasible solution to MMHC.

Table 1. The average running times on 100 randomly generated replicates for each pedigree size. The pedigrees are full binary trees.

Pedigree size	Straightforward	Incremental
63	.443s	.144s
127	2.98s	.750s
255	20.3s	4.39s
511	180s	29.0s
1023	29.2m	2.13m

Table 2. The performance of MMPhase under various configurations of the parameters. The default setting includes the pedigree of size 52, 50 marker loci, 10% missing alleles, and 3% mutations. 100 replicated are generated for each configuration of the parameters. Starting from the default setting, we vary the missing rate in (a), the mutation rate in (b), the pedigree in (c), and the number of loci in (d).

Missing rate	Correctly imputed missing alleles	Correctly detected mutations	Correctly phased markers	Running time
0%	—	78.31%	99.98%	2.02s
5%	74.70%	70.20%	98.49%	1.49s
10%	68.97%	62.58%	92.72%	1.24s
20%	69.03%	59.69%	92.75%	.900s

(a)

Mutation rate	Correctly imputed missing alleles	Correctly detected mutations	Correctly phased markers	Running time
1%	73.15%	73.33%	96.75%	1.23s
3%	68.97%	62.58%	92.72%	1.24s
10%	73.11%	69.57%	96.73%	1.47s

(b)

Pedigree size	Correctly imputed missing alleles	Correctly detected mutations	Correctly phased markers	Running time
29	75.34%	52.26%	94.51%	.298s
52	68.97%	62.58%	92.72%	1.24s
128	73.49%	52.11%	93.99%	27.0s

(c)

Locus number	Correctly imputed missing alleles	Correctly detected mutations	Correctly phased markers	Running time
20	73.13%	67.00%	96.78%	.250s
50	68.97%	62.58%	92.72%	1.24s
200	73.09%	65.42%	96.82%	10.8s

(d)

Note that, the number of such consistency constraints is $O(mn)$. Interesting, the incremental approach implemented in MMPhase may theoretically use more consistency constraints in the worst case because of creating redundant variables, although it usually uses a smaller number of consistency constraints in practice. We consider full binary trees of sizes from 63 to 1023, and run both algorithms on 100 randomly generated genotype data with 50 loci, 10% missing alleles,

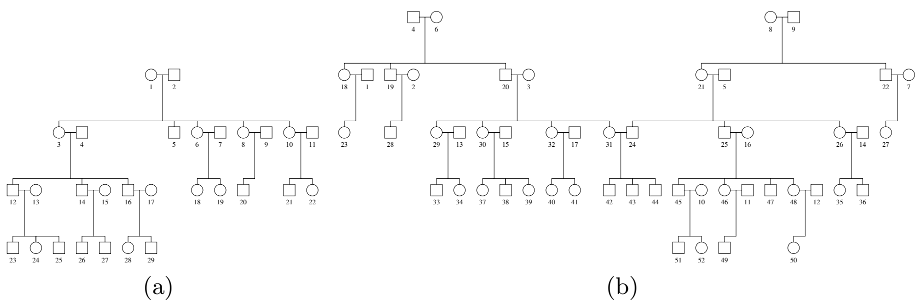


Fig. 4. Three pedigrees are used to test the performance of MMPhase. The first has 29 individuals and is shown in (a). The second has 52 individuals and is shown in (b). The third has 128 individuals and is too large to fit in the page.

and 3% mutations (*i.e.* 3% of the loci are mutated in inheritance). (Actually, haplotypes are generated in the simulations and then converted to genotypes as an input of the algorithms.) Table 1 shows the average running times of both algorithms on each full binary tree. We observe that MMPhase is much faster than the straightforward method, and the speedup ratio increases as the pedigree size gets bigger. For example, the ratio is about 14 on full binary trees of size 1023. We also observe that the solutions from both algorithms are sometimes slightly different but they always require the same number of mutations which is smaller than the actual number of mutations simulated (the detailed results are not shown).

Next we test the performance of MMPhase in terms of the percentage of correctly phased markers, the percentage of correctly imputed missing alleles, and the percentage of correctly detected mutations. (A simulated mutation is correctly detected if there is an inferred mutation that coincides with its location exactly.) We use three real human pedigrees from the literature as shown in Figure 4. 100 replicates of genotype data is simulated on each of these pedigrees with each of several configurations of the number of marker loci, the missing allele rate and the mutation rate. Our default setting of simulation uses the pedigree of size 52, 50 marker loci, 10% missing alleles, and 3% mutations. To observe how each of these parameters affects the performance MMPhase, we vary one parameter at a time in the test. Table 2 illustrates the test results. We observe that, as shown in Table 2(a), higher missing rates lead to faster performance since fewer path/cycle constraints are added to the ILP instance. Not surprisingly, higher missing rates also result in fewer correctly detected mutations and fewer correctly phased markers. Table 2(b) shows that the performance is not very sensitive to the mutation rate. Table 2(c) and Table 2(d) show that the pedigree and number of marker loci mainly affect the running time.

In conclusion, MMPhase is very efficient and can infer haplotypes very accurately. It can also recover most of the mutations and missing alleles correctly.

Note that, our criterion for correctly detecting a mutation is very stringent since in some cases the mutation could be shifted in the pedigree without affecting the feasibility of the solution (especially when missing alleles are present).

5 Concluding Remarks and Acknowledgements

Genotyping errors are very common in practice and can easily be confused with mutations. It would be interesting to extend the method to deal with both mutations and genotyping errors. The research is supported in part by NIH grant LM008991 and NSF grant IIS-0711129.

References

1. Abecasis, G.R., Cherny, S.S., Cookson, W.O., Cardon, L.R.: Merlin — rapid analysis of dense genetic maps using sparse gene flow trees. Nature Genetics 30(1), 97–101 (2002)
2. Albers, C.A., Heskes, T., Kappen, H.J.: Haplotype inference in general pedigrees using the cluster variation method. Genetics 177(2), 1101–1116 (2007)
3. Badaeva, T.N., Malysheva, D.N., Korchagin, V.I., Ryskov, A.P.: Genetic variation and *De Novo* mutations in the parthenogenetic caucasian rock lizard *Darevskia unisexualis*. PLoS ONE 3(7), e2730 (2008)
4. Baruch, E., Weller, J.I., Cohen-Zinder, M., Ron, M., Seroussi, E.: Efficient inference of haplotypes from genotypes on a large animal pedigree. Genetics 172(3), 1757–1765 (2006)
5. Ellegren, H.: Microsatellite mutations in the germline: Implications for evolutionary inference. Trends in Genetics 16(12), 551–558 (2000)
6. Gusfield, D.: Inference of haplotypes from samples of diploid populations: Complexity and algorithms. J. Computational Biology 8(3), 305–323 (2001)
7. Kimura, M., Crow, J.F.: The number of alleles that can be maintained in a finite population. Genetics 49, 725–738 (1964)
8. Lander, E.S., Green, P.: Construction of multilocus genetic linkage maps in humans. In: Proc. of the National Academy of Sciences. Genetics, vol. 84, pp. 2363–2367 (1987)
9. Li, J., Jiang, T.: Efficient inference of haplotypes from genotypes on a pedigree. J. Computational Biology 1(1), 41–69 (2003)
10. Li, J., Jiang, T.: Computing the minimum recombinant haplotype configuration from incomplete genotype data on a pedigree by integer linear programming. J. Computational Biology 12(6), 719–739 (2005)
11. Li, J., Jiang, T.: A survey on haplotype algorithms for tightly linked markers. J. Bioinformatics and Computational Biology 6(1), 241–259 (2008)
12. Liu, L., Jiang, T.: Linear-time reconstruction of zero-recombinant mendelian inheritance on pedigrees without mating loops. Genome Informatics 19, 95–106 (2007)
13. Niu, T., Qin, Z.S., Xu, X., Liu, J.S.: Bayesian haplotype inference for multiple linked single-nucleotide polymorphisms. Am. J. Hum. Genet. 70(1), 157–169 (2002)
14. Olson, T.M., Doan, T.P., Kishimoto, N.Y., Whitby, F.G., Ackerman, M.J., Fananapazir, L.: Inherited and *de novo* mutations in the cardiac actin gene cause hypertrophic cardiomyopathy. J. Molecular and Cellular Cardiology 32(9), 1687–1694 (2000)

15. Qian, D., Beckmann, L.: Minimum-recombinant haplotyping in pedigrees. Am. J. Hum. Genet. 70(6), 1434–1445 (2002)
16. Sobel, E., Lange, K., O'Connell, J.R., Weeks, D.E.: Haplotyping algorithms. In: Speed, T., Waterman, M.S. (eds.) Genetic Mapping and DNA Sequencing. IMA Volumes in Mathematics and its Applications, vol. 81, pp. 89–110. Springer, Heidelberg (1996)
17. Stephens, M., Smith, N.J., Donnelly, P.: A new statistical method for haplotype reconstruction from population data. Am. J. Hum. Genet. 68(4), 978–989 (2001)
18. Tapadar, P., Ghosh, S., Majumder, P.P.: Haplotyping in pedigrees via a genetic algorithm. Human Heredity 50(1), 43–56 (2000)
19. The Internaltional HapMap Consortium. The international HapMap project. Nature 426, 789–796 (2003)
20. Wang, S., Kidd, K.K., Zhao, H.: On the use of DNA pooling to estimate haplotype frequencies. Genetic Epidemiology 24(1), 74–82 (2003)
21. Xiao, J., Liu, L., Xia, L., Jiang, T.: Fast elimination of redundant linear equations and reconstruction of recombination-free mendelian inheritance on a pedigree. In: 18th Annual ACM-SIAM Symposium on Descrete Algorithms, pp. 655–664 (2007)
22. Yang, Y., Zhang, J., Hoh, J., Matsuda, F., Xu, P., Lathrop, M., Ott, J.: Efficiency of single-nucleotide polymorphism haplotype estimation from pooled DNA. Proc. of the National Academy of Sciences 100, 7225–7230 (2002)
23. Zhang, K., Sun, F., Zhao, H.: HAPLORE: A program for haplotype reconstruction in general pedigrees without recombination. Bioinformatics 21(1), 90–103 (2005)

Author Index