Proceedings of the Prague Stringology Conference 2011

Edited by Jan Holub and Jan Žďárek

August 2011

Prague Stringology Club
http://www.stringology.org/

Proceedings of the Prague Stringology Conference 2011
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, Praha 6, 160 00, Czech Republic.
URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811
Printed by Česká technika – Nakladatelství ČVUT, Thákurova 550/1, Praha 6, 160 41, Czech Republic

© Czech Technical University in Prague, Czech Republic, 2011

ISBN 978-80-01-04870-2

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Maxime Crochemore	(King's College London, United Kingdom)
František Franěk	(McMaster University, Canada)
Jan Holub, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Costas S. Iliopoulos	(King's College London, United Kingdom)
Shunsuke Inenaga	(Kyushu University, Japan)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Thierry Lecroq	(Université de Rouen, France)
Bořivoj Melichar, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Yoan J. Pinzón	(Universidad Nacional de Colombia, Colombia)
Marie-France Sagot	(INRIA Rhône-Alpes, France)
William F. Smyth	(McMaster University, Canada)
Bruce W. Watson	(FASTAR Group (Stellenbosch University and University of Pretoria, South Africa))

Organizing Committee

Miroslav Balík, *Co-chair*	Jan Janoušek	Ladislav Vagner
Jan Holub, *Co-chair*	Bořivoj Melichar	Jan Žďárek

External Referees

Golnaz Badkobeh	Derrick Kourie	Elise Prieur-Gaston
Loek Cleophas	Arnaud Lefebvre	Simon Puglisi
Simone Faro	Wataru Matsubara	Mikaël Salson
Rémi Forax	Solon Pissis	Dana Shapira

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2011 (PSC 2011) which was organized by the members of the Prague Stringology Club at the Department of Theoretical Computer Science, the Faculty of Information Technology of the Czech Technical University in Prague. The conference was held on August 29–31, 2011 and it focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the program committee. Nineteen papers were selected, based on originality and quality. Seventeen of them were accepted as regular papers and two as short papers for presentations at the conference. This volume contains not only these selected papers but also an abstract of one invited talk devoted to a survey of the last developments in the exact string matching.

The Prague Stringology Conference has a long tradition. PSC 2011 is the sixteenth event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2010 preceded this conference. The proceedings of these workshops and conferences had been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions were published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science. The series of stringology conferences was interrupted in 2007 when the members of the Prague Stringology Club were honoured to organize Conference on Implementation and Application of Automata 2007 (CIAA 2007).

The Prague Stringology Club was founded in 1996 as a research group in the Czech Technical University in Prague. The goal of the Prague Stringology Club is to study algorithms on strings, sequences, and trees with emphasis on automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful of invited talks. However, since PSCW'97 the papers and talks are selected by a rigorous peer review process. The objective is not only to present new results in stringology and related areas, but also to facilitate personal contacts among the people working on these problems.

I would like to thank all those who had submitted papers for PSC 2011 as well as the reviewers. Special thanks go to all the members of the program committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC 2011. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2011

Jan Holub

Table of Contents

Invited Talk

Contributed Talks

2001–2010: Ten Years of Exact String Matching Algorithms

Simone Faro[1] and Thierry Lecroq[2]

[1] Università di Catania, Dipartimento di Matematica e Informatica, Viale Andrea Doria 6, I-95125 Catania, Italy, faro@dmi.unict.it
[2] University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France, Thierry.Lecroq@univ-rouen.fr

The *online exact string matching problem* consists in finding *all* occurrences of a given pattern p in a text t. It is an extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, data compression, computational biology and chemistry.

In the last decade more than 50 new algorithms have been proposed for the problem, which add up to a wide set of (almost 40) algorithms presented before 2000 [1]. We will review the most efficient string matching algorithms presented in the last decade in order to bring order among the dozens of articles published in this area.

We performed comparisons between 85 exact string matching algorithms with 12 texts of different types [4]. We divide the patterns into four classes according to their length m: very short ($m \leq 4$), short ($4 < m \leq 32$), long ($32 < m \leq 256$) and very long ($m > 256$). We proceed in the same way for the alphabets according to their size σ: very small ($\sigma < 4$), small ($4 \leq \sigma < 32$), large ($32 \leq \sigma < 128$) and very large ($\sigma > 128$). According to our experimental results (see Figure 1), we conclude that the following algorithms are the most efficient in the following situations:

- SA [11]: very short patterns and very small alphabets.
- TVSBS [10]: very short patterns and small alphabets, and long patterns and large alphabets.
- FJS [5]: very short patterns and large and very large alphabets.
- EBOM [3]: short patterns and large and very large alphabets.
- SBNDM-BMH and BMH-SBNDM [6]: short patterns and very large alphabets.
- HASHq [8]: short and large patterns and small alphabets.
- FSBNDM [3]: long patterns and large and very large alphabets.
- SBNDMq [2]: long pattern and small alphabets.
- LBNDM [9]: very long patterns and very large alphabets.
- SSEF [7]: very long patterns.

Among these algorithms all but one (the SA algorithm) have been designed during the last decade, four of them are based on comparison of characters, one of them is based on automata while six of them are bit-parallel algorithms.

In order to ease further works for developing fast exact string matching algorithms, we developed smart (string matching algorithms research tool, http://www.dmi.unict.it/~faro/smart/) which is a tool that provides a standard framework for researchers in string matching. It helps users to test, design, evaluate and understand existing solutions for the exact string matching problem. Moreover it provides the implementation of (almost) all string matching algorithms and a wide corpus of text buffers.

Simone Faro, Thierry Lecroq: 2001–2010: Ten Years of Exact String Matching Algorithms, pp. 1–2.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

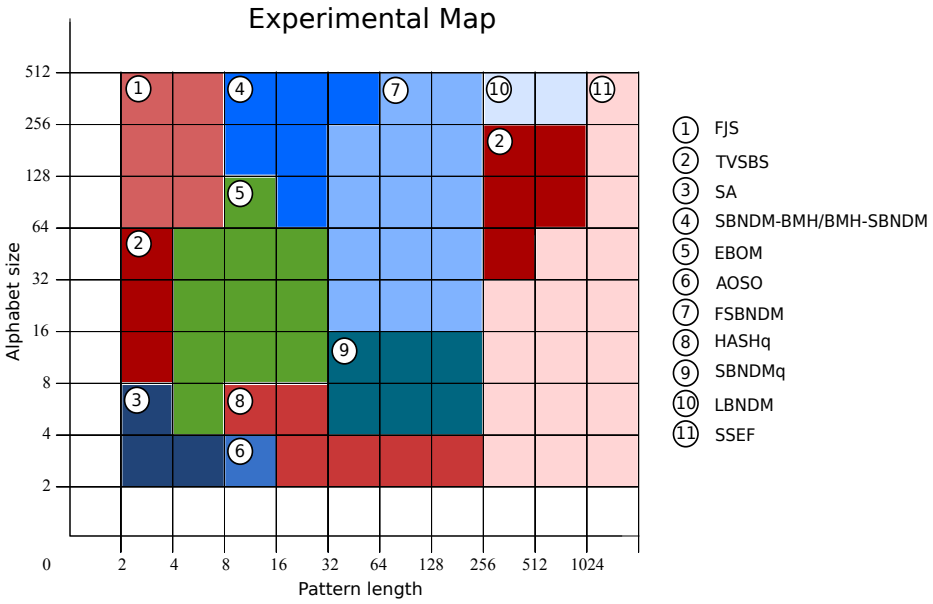


Figure 1. Experimental map of the best results obtained in our evaluation. Comparison based algorithms are presented in red gradations, automata based algorithms are presented in green gradations and bit parallel algorithms are presented in blue gradations.

References

1. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College Publications, 2004.
2. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Tuning BNDM with q-grams*, in Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2009, I. Finocchi and J. Hershberger, eds., New York, New York, USA, 2009, SIAM, pp. 29–37.
3. S. Faro and T. Lecroq: *Efficient variants of the Backward-Oracle-Matching algorithm*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 146–160.
4. S. Faro and T. Lecroq: *The exact string matching problem: a comprehensive experimental evaluation*, Report arXiv:1012.2547, Computing Research Repository, 2010.
5. F. Franek, C. G. Jennings, and W. F. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discret. Algorithms, 5(4) 2007, pp. 682–695.
6. J. Holub and B. Durian: *Talk: Fast variants of bit parallel approach to suffix automata*, in The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, `http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf`, 2005.
7. M. O. Külekci: *Filter based fast matching of long patterns by using simd instructions*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 118–128.
8. T. Lecroq: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
9. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proceedings of the 10th International Symposium on String Processing and Information Retrieval SPIRE'03, M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, eds., vol. 2857 of Lecture Notes in Computer Science, Manaus, Brazil, 2003, Springer-Verlag, Berlin, pp. 80–94.
10. R. Thathoo, A. Virmani, S. S. Lakshmi, N. Balakrishnan, and K. Sekar: *TVSBS: A fast exact pattern matching algorithm for biological sequences.* J. Indian Acad. Sci., Current Sci., 91(1) 2006, pp. 47–53.
11. S. Wu and U. Manber: *Fast text searching allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.

Variations of Forward-SBNDM*

Hannu Peltola and Jorma Tarhio

Department of Computer Science and Engineering,
Aalto University, P.O.B. 15400, FI-00076 Aalto, Finland
{hannu.peltola,jorma.tarhio}@aalto.fi

Abstract. Forward-SBNDM is a recently introduced variation of the BNDM algorithm for exact string matching. Forward-SBNDM reads a text character following an alignment of the pattern. We present a generalization of this lookahead idea and apply it to SBNDMq for $q \geq 3$. As a result we get several new variations of SBNDMq. We introduce a greedy skip loop for SBNDM2. In addition, we tune up our algorithms and the reference algorithms with 2-byte read. According to our experiments, the best of the new variations are in several cases faster than the winners of recent algorithm comparisons.

Keywords: string matching, BNDM, 2-byte read, q-grams

1 Introduction

After the advent of the Shift-Or [2] algorithm, bit-parallel string matching methods have gained more and more interest. The BNDM (Backward Nondeterministic DAWG Matching) algorithm [17] is a nice example of an elegant, compact, and efficient piece of code for exact string matching. BNDM simulates the nondeterministic finite automaton of the reverse pattern even without constructing the actual automaton.

SBNDM2 [6,11] is a simplified variation of BNDM. SBNDM2 starts processing of an alignment by reading two characters. Recently Faro and Lecroq [7] introduced Forward-SBNDM, a lookahead version of the SBNDM2 algorithm. In this paper we present a generalization of the lookahead idea and give new variations of SBNDMq [6] for $q \geq 3$. SBNDMq starts processing of an alignment by reading q characters. In addition, we introduce a greedy skip loop for SBNDM2. Our point of view is practical efficiency of exact string matching algorithms. According to our experiments, the best of the new variations are in several cases faster than the winners of recent algorithm comparisons [6,9].

We use the following notations. Let a pattern $P = p_1 p_2 \cdots p_m$ and a text $T = t_1 t_2 \cdots t_n$ be two strings over a finite alphabet Σ. The task of exact string matching is to find all occurrences of P in T. Formally we search for all positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_1 p_2 \cdots p_m$. In the pseudocode of the algorithms we use some notations of the programming language C: '|', '&', '~', '<<', and '>>' represent bitwise operations OR, AND, one's complement, left shift, and right shift, respectively. The register width (or word size informally speaking) of a processor is denoted by w.

The rest of the paper is organized as follows. Since our work is based on SBNDMq and Forward-SBNDM, we start with presenting these algorithms in Section 2. In Section 3 we generalize Forward-SBNDM with wider lookahead and longer q-grams. In Section 4 the greedy skip loop is presented. Section 5 reviews the results of our experiments before concluding remarks in Section 6.

* Supported by the Academy of Finland (grant 134287).

Hannu Peltola, Jorma Tarhio: Variations of Forward-SBNDM, pp. 3–14.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

2 Previous algorithms

2.1 SBNDM*q*

SBNDMq [6] is a variation of SBNDM [18], a simplified version of BNDM, applying q-grams. The pseudocode is shown as Alg. 1. $F(i, q)$ on line 6 is a shorthand notation for the expression

$$B[t_i] \,\&\, (B[t_{i+1}] \ll 1) \,\&\, \cdots \,\&\, (B[t_{i+q-1}] \ll (q-1)).$$

Algorithm 1 SBNDMq ($P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n$)

```
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j))
 4: i ← m − q + 1
 5: while i ≤ n − q + 1 do
 6:     D ← F(i, q)
 7:     if D ≠ 0 then
 8:         j ← i − (m − q + 1)
 9:         repeat
10:             i ← i − 1
11:             D ← (D << 1) & B[t_i]
12:         until D = 0
13:         if j = i then
14:             report occurrence at j + 1
15:             i ← i + s_0
16:     i ← i + m − q + 1
```

At each alignment, SBNDMq first reads q characters $t_i, \ldots, t_{i+q-1}$ before testing the state vector D. If D is zero, this q-gram (i.e., the string of q characters) is not a factor (i.e. a substring) of P, and then the pattern can be shifted forward $m - q + 1$ positions. If D is not zero, a single character at a time is read to the left until the suffix $t_i \cdots t_{j+m}$ of the alignment is not any more a factor of P. If $t_i \cdots t_{j+m}$ is not a factor of P and $i > j$ holds, the pattern is shifted forward and the next alignment starts at t_{i+1}.

In the original BNDM, the inner loop also recognizes prefixes of the pattern. The leftmost one of the found prefixes determines the next alignment of BNDM. SBNDMq does not care of prefixes, but shifts the pattern simply past the text character which nullifies D.

When an occurrence of the pattern is found, the shift is s_0, which corresponds to the distance to the leftmost prefix of the pattern in itself and which is easily computed from the pattern (see [6]). We skip the details, because a conservative value $s_0 = 1$ works well in practice. In the subsequent algorithms of this paper we use the value $s_0 = 1$.

2.2 Forward-SBNDM

Forward-SBNDM, a lookahead version of SBNDM2, was introduced by Faro and Lecroq [7]. The idea of the algorithm is the following. As in SBNDM2, a 2-gram x_1x_2 is read before testing the state vector D. In SBNDM2, x_1x_2 is matched with the end of the pattern. In Forward-SBNDM, only x_1 is matched with the end of the pattern,

Algorithm 2 Forward-SBNDM ($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
Require: 1 ≤ m < w
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 1
 2: for j ← 1 to m do
 3:       B[p_j] ← B[p_j] | (1 << (m − j + 1))
    /* Searching */
 4: i ← m
 5: while i ≤ n do
 6:       D ← (B[t_{i+1}] << 1) & B[t_i]
 7:       if D ≠ 0 then
 8:            j ← i
 9:            repeat
10:                 i ← i − 1
11:                 D ← (D << 1) & B[t_i]
12:            until D = 0
13:            i ← i + m − 1
14:            if j = i then
15:                 report occurrence at j + 1
16:                 i ← i + 1
17:       i ← i + m
```

and x_2 is a lookahead character. By lookahead characters we mean the text characters immediately following the current alignment. Note that $B[x_2]$ can nullify several bits of D, and therefore x_2 enables longer shifts. The pseudocode of Forward-SBNDM is shown as Alg. 2.

After reading x_1x_2 in Forward-SBNDM there are three possibilities to proceed. (i) If x_1x_2 is a factor of P, reading continues leftwards. (ii) If x_1x_2 is not a factor of P and if x_1 matches the last character of P, reading continues leftwards. The extra set bit in the end of B vectors ensures that the state vector D does not get nullified in this case. (iii) If x_1x_2 is not a factor of P and if x_1 does not match the last character of P, then D becomes zero and the pattern is shifted m positions and shift is one longer than in SBNDM2.

Because the length of the occurrence vector B of each character is $m + 1$ in Forward-SBNDM, the upper limit for the pattern length is thus $w - 1$. The extra bit is the rightmost one, and its value is always one, because the lookahead character is not allowed to interfere with recognition of a valid occurrence of P.

In a way Forward-SBNDM is a cross of SBNDM2 and Sunday's QS [19]. QS was the first algorithm to use a lookahead character for shifting. Another famous algorithm using two lookahead characters is by Berry and Ravindran [3].

3 Generalization: Forward-SBNDM*q*

Ďurian et al. [5,6] reported that SBNDMq is efficient also for $q > 2$ on modern processors, although the number of read characters increases with q. This increment can be considerable in the case of short patterns, but this straightforward method is faster on average than SBNDM in most cases. Based on this observation we decided to examine whether a longer lookahead than one as in Forward-SBNDM would be beneficial for SBNDMq. So based on SBNDMq we constructed Forward-SBNDM(q, f), where the lookahead f can be any integer between 0 and $q - 1$. Our preliminary experiments

convinced us that longer lookaheads would be beneficial. The pseudocode is given as Alg. 3.

Algorithm 3 Forward-SBNDM(q, f) ($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
Require: q − f ≤ m ≤ w − f and 0 ≤ f < q
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← (~0) >> (w − f)          /* 1^f */
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j + f))
    /* Searching */
 4: i ← m − q + f
 5: while i ≤ n − q + 1 do
 6:     D ← F(i, q)
 7:     if D ≠ 0 then
 8:         j ← i − (m − q + f + 1)
 9:         repeat
10:             i ← i − 1
11:             D ← (D << 1) & B[t_i]
12:         until D = 0
13:         if j = i then
14:             report occurrence at j + 1
15:             i ← i + 1
16:     i ← i + m − q + f + 1
```

Note that Forward-SBNDM(q,0) is in practice the same as SBNMDq if $s_0 = 1$ is selected. If we keep $f - q$ in a precomputed variable, then even the search part of Forward-SBNDM(q, f) is independent of the value of f. Note also that Forward-SBNDM$(2, 1)$ corresponds to the original Forward-SBNDM.

Because the length of the occurrence vector B of each character is $m + f$, the upper limit for the pattern length is thus $w - f$. In addition it is required that $0 < q - f \leq m$. When changing q, only line 6 needs to be updated. Note that like SBNDMq, Forward-SBNDM(q, f) may read a few characters beyond the text (line 6) and also one character before the text (line 11).

Providing $m \leq w$, the worst case time complexity of BNDM is $\mathcal{O}(mn)$, but the average time complexity is sublinear. The space complexity of BNDM is $\mathcal{O}(|\Sigma|)$. It is straightforward to show that Forward-SBNDM(q, f) inherits these complexities when $m \leq w - f$.

Let $t_i \cdots t_{i+q-1} = x_1 \cdots x_{q-f} y_1 \cdots y_f$ be the q-gram read on line 6. As in the case of Forward-SBNDM, there are three possibilities to proceed. (i) If $x_1 \cdots x_{q-f} y_1 \cdots y_f$ is a factor of P, reading continues leftwards. (ii) If $x_1 \cdots x_{q-f} y_1 \cdots y_f$ is not a factor of P and if $x_1 \cdots x_{q-f}$ matches the suffix of P, reading continues leftwards. The extra set bits in the end of B vectors ensure that the state vector D does not get nullified. (iii) If $x_1 \cdots x_{q-f} y_1 \cdots y_f$ is not a factor of P and if $x_1 \cdots x_{q-f}$ does not match the suffix of P, then the pattern is shifted and the next alignment ends at t_{i+m}. The shift is $m - q + f + 1$, which is f positions more than in SBNDMq.

The disadvantage of Forward-SBNDM(q, f) is that there is a larger risk to fall to the slow loop on lines 8–15, because the probability $F(i, x)$ to be nonzero is higher for $x = q - f$ than for $x = q$.

Example. Let P be `abcdefgh`. The maximal shifts of SBNDM2 and SBNDM3 are 7 and 6, respectively. The maximal shift of Forward-SBNDM(3,2) is 7. Let us consider

a text $T =$`...xabcdey...` If SBNDM2 reads a 2-gram `de`, it scans back until `x`. If Forward-SBNDM(3,2) reads 3-gram `dey`, it immediately skips 7 positions onwards, because `d` is not a suffix of P and `dey` is not a factor of P.

Variation. The way how f lookahead characters are handled takes f low order bits in the state vector D, which reduces the maximal length of the pattern. This could be circumvented by using on line 6 a distinct occurrence vector table C_k (corresponding to B) for each of the q text positions. Then $F(i, q)$ is interpreted as

$$C_1[t_i] \,\&\, C_2[t_{i+1}] \,\&\, \cdots \,\&\, C_q[t_{i+q-1}],$$

where $C_k[x] = ((B[x] \ll f) + 2^f - 1) \gg (q - k)$ where B is B of SBNDMq as well as on line 11. Note that $2^f - 1$ ensures that the f lowest order bits are set. The justification for deleting the f high order bits by a left shift in the computation of $C_k[x]$ is that they are not needed in the algorithm, because we can assume that $q < w/2$ holds.

Implementation note. In the C language the right operand of a shift operation must be shorter than the width of the left operand. Therefore on line 1 of Alg. 3, shifting has to be made in two parts or handled e.g. with if clause, when $f = 0$.

4 Greedy skip loop

Many string matching algorithms apply so called skip loop, which is used for fast scanning before entering the matching phase. E.g. a basic skip loop of SBNDM is the following:

$$\textbf{while } B[t_i] = 0 \textbf{ do } i \leftarrow i + m.$$

Faro and Lecroq [7,8] introduce several interesting variations of skip loop. In the variation (originally for an algorithm of SBNDM2 type)

$$\textbf{while } B[t_i] = 0 \textbf{ do } i \leftarrow i + d[t_{i+m}] \qquad (1)$$

the maximal step is $2m$, where d is a shift table based on the bad character heuristics a.k.a. the occurrence heuristics. We tried several variations of (1), but we did not succeed improving the speed of our algorithms in our test setting.

Here we present a new type of skip loop for SBNDM2. We call it greedy, because in some cases it reads lookahead characters that it does not utilize. The pseudocode is given as Alg. 4.

Two 2-grams are read in the skip loop. If both do not appear in P, the shift is $2m - 2$. If the former appears in P, the latter is not read (the operator && denotes a short-circuit AND) and the computation proceeds as in SBNDM2. If only the latter 2-gram $t_k t_{k+1}$ appears in P, the next operation is a shift of $m - 2$. This means that the new former 2-gram is $t_{k-1}t_k$. Here also a shift of $m - 1$ would be possible, but that alternative is a bit slower in practice, because we already know that $t_k t_{k+1}$ is a factor of P.

It would be straightforward to generalize the greedy loop for SBNDMq. Instead of reading two 2-grams, the loop may hold reading of two q-grams or a q-gram and a 2-gram.

The form of the greedy skip loop is based on the observation that the cost of side assignments is very small. We tried several variations of the greedy loop on several processors. Unfortunately no variation was clearly the best.

Algorithm 4 Greedy-SBNDM2 ($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
Require: 1 ≤ m < w
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:       B[p_j] ← B[p_j] | (1 << (m − j))
    /* Searching */
 4: i ← m − 1
 5: while i ≤ n do
 6:       while (D ← ((B[t_{i+1}] << 1) & B[t_i])) = 0 &&
          ((B[t_{i+m}] << 1) & B[t_{i+m−1}]) = 0 do
 7:           i ← i + 2m − 2
 8:       if D ≠ 0 then
 9:           j ← i
10:           repeat
11:               i ← i − 1
12:               D ← (D << 1) & B[t_i]
13:           until D = 0
14:           i ← i + m − 1
15:           if j = i then
16:               report occurrence at j + 1
17:               i ← i + 1
18:       else
19:           i ← i + m − 2
```

5 Experimental results

We implemented Greedy-SBNDM2 and Forward-SBNDM(q, f) versions up to $q \leq 8$ and for $f \leq \min\{q-1, 5\}$. For efficiency f and q were compile time constants. For each variation we implemented two versions. The standard version corresponds otherwise to the pseudocode, but the test of the outer loop was eliminated and a copy of the pattern was placed as a stopper after the last text character t_n. The b-version applies 2-byte read, where two bytes are read with one instruction. As a result a part of bit shifts was moved to preprocessing as explained below. Otherwise the search part of the b-version is identical with the corresponding standard version.

2-byte read. Reading several bytes at a time is a well-known technique. Fredriksson [10] was probably the first who analyzed its advantage. A string matching algorithm applying 2-byte read is in practice much faster than the traditional version. In some cases the speedup becomes close to two, which is the theoretical limit. The cost of reading one or two bytes is almost the same on most x86 processors. Only crossing a word border causes small overhead [14]. A noteworthy additional advantage is the possibility to move computation from the scanning phase to preprocessing. When applying 2-byte read in an algorithm of BNDM type, we replace a C language expression `B[t[i]] & (B[t[i+1]]<<1)` by `B2[*(uint16_t*)(t+i)]`, where `(uint16_t*)` is a typecast and `t+i` is a reference (pointer) to the character `t[i]`. The table `B2` is computed during preprocessing. When processing a 4-gram, it is advantageous to process it as two separate 2-byte reads (see [6,14] for details) in order to decrease the penalty of crossing word borders. The same holds for even larger values of q.

Unaligned 2-byte reads work also on some other CPU architectures than x86. During preprocessing we take care of endianess (the order in which integer values are stored as bytes in the computer memory). Let `x` and `y` be two succes-

sive characters. The indexing of the table `B2` is different. On a little endian machine (like x86) `B2[(y<<8)+x]=B[x]&(B[y]<<1)` and on a big endian machine `B2[(x<<8)+y]=B[x]&(B[y]<<1)` is applied. If you regard 2-byte read as a machine level thing, you may accept a lighter version applying only the array of 2-byte integers. Depending on the input, `B2[(t[i+1]<<8)+t[i]]` is slightly faster than the original expression in many x86 processors.

Reference algorithms. In addition to variations of SBNDMq we tested four other algorithms:

- BR [3] by Berry and Ravindran,
- EBOM [7] by Faro and Lecroq,
- Hash3 [16] (originally New3) by Lecroq, and
- BMH2 [20,14], a 2-gram variation of the Horspool algorithm [12].

We updated each algorithm with a stopper handling and made a b-version in the same way explained above for Forward-SBNDM(q, f).

Concerning BMH2, many researchers have worked out related variations [1,15,20,21]. The basic idea has been mentioned already in the original article of Boyer and Moore [4]. BR is a cross of BMH2 and Sunday's QS algorithm [19]. In BMH2 the shift is based on the last 2-gram of the text window aligned with the pattern, whereas BR applies the 2-gram locating two positions further to the right. EBOM is an efficient implementation of the oracle automaton utilizing 2-grams.

Because Hash3 applies a 3-gram, the application of 2-byte read is a bit different. The statements

```
h = text[i-2];
h = ((h<<1) + text[i-1]);
h = ((h<<1) + text[i]);
```

are replaced by

```
h = d2[*(uint16_t*)(text+i-2)]+text[i];
```

BMH2 and BR are examples of old algorithms. BR was among the first algorithms to discredit the connection with the number of character reads and efficiency. EBOM and Hash3 are the winners of several test cases in a recent comparison [9].

We use the shorthands FSB and GSB for Forward-SBNDM and Greedy-SBNDM, respectively. FSB(q, f)b for odd q was implemented so that the q-gram is processed using $(q-1)/2$ consecutive 2-byte reads followed by one 1-byte read. Because FSB(q,0) is in practice the same as SBNMDq, $q = 2, 3, \ldots$, the former ones also serve as reference methods, because the latter ones are among the best in our recent comparison [6].

Computers and test setting. We run the main tests on two computers. The first one was IBM ThinkPad X61s having Intel Core 2 Duo Processor L7300 (32 KiB L1 data cache). The test environment was Cygwin. The second computer was a Dell Precision T1500 containing Intel Core i7-860 2.8 GHz CPU (8 KiB L1 data cache/core) running with the 64-bit Ubuntu kernel 2.6.35-30. The programs in C were compiled with the gcc compiler version 3.4.4 in IBM and 4.4.5 in Dell to run either in the 32-bit mode or in the 64-bit mode (only in Dell) using the optimization level `-O3`.

In the main tests we used three texts: English (4 MB), DNA (2 MB), and binary (2 MB). The English text was the KJV Bible. Sets of patterns of various lengths were

randomly taken from each text. Each set contained 200 patterns. Neither end of the English patterns was aligned with boundaries of English words.

All the algorithms were tested in a testing framework of Hume and Sunday [13]. The data was in the main memory so that I/O time had no effect to speed measurements. The search speeds shown are averages of 100 runs (if not otherwise told). Accuracy of the results is about 1 %. For organizational reasons the test sets of ThinkPad X61s and Dell T1500 were not identical.

With 32-bit bitvectors the maximum pattern length for FSB(*,3) is 29. Therefore some results of FSB(4,3) for length 30 are missing.

Text 1: English. The search speeds on English data are shown in Tables 1 and 2. The best speed for each pattern set has been boxed. Both GSB2 and EBOM were among the fastest standard algorithms for $m \leq 15$. Also FSB(3,1) (not tested for Table 1), FSB(4,0), FSB(4,1), and FSB(4,2) worked well for longer patterns. Among the b-versions GSB2b was good for short patterns. FSB(4,0)b, FSB(4,1)b, and FSB(4,2)b were excellent for $m \geq 7$.

As explained in Section 3, FSB(4,f), $f > 0$, was developed from SBNDM4 $\simeq$ FSB(4,0). For most values of m, one of FSB(4,f) was faster than FSB(4,0). The same was true for the b-versions, but the gain on Dell extended further. Note that for $m = 4$, FSB(4,0) and FSB(4,0)b process the whole pattern in the outer loop of the algorithm, and shift is always one! As explained in Section 4, GSB2 was developed from SBNDM2 $\simeq$ FSB(2,0). GSB2 was faster than FSB(2,0) for short patterns. The same was true for the b-versions, but the gain on ThinkPad extended further.

Note that FSB(2,1) $\simeq$ original Forward-SBNDM was slower than SBNDM2 $\simeq$ FSB(2,0). (The same was true for the b-versions.) We made an additional test with an alphabet of 128 characters in order to verify that FSB(2,1) is faster than FSB(2,0) in a text of a larger alphabet as shown in [9].

Relative speedup of 2-byte read is shown in Table 3. Numbers are arithmetic means of the speed ratios calculated from the data of Table 2. The overall average speedup was 1.52 in this test set. The speedup was the biggest for $m = 4$ and decreased as patterns get longer. Note that two of the algorithms went over the limit of two, possibly due to advantageous pipelining.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	0.74	1.22	1.52	1.84	2.03	2.42	1.26	1.80	2.00	2.24	2.44	2.92
FSB(2,0)	0.71	1.16	1.43	1.74	1.95	2.30	1.06	1.57	1.80	2.05	2.25	2.67
FSB(2,1)	0.66	0.99	1.23	1.56	1.82	2.25	0.79	1.17	1.40	1.74	2.02	2.47
FSB(4,0)	0.16	0.60	1.02	1.68	2.25	3.23	0.34	1.27	2.05	3.13	3.78	4.71
FSB(4,1)	0.31	0.73	1.15	1.78	2.34	3.27	0.63	1.49	2.18	3.17	3.78	4.73
FSB(4,2)	0.43	0.85	1.23	1.81	2.40	3.29	0.85	1.61	2.26	3.11	3.78	4.59
FSB(4,3)	0.48	0.81	1.12	1.64	2.12	–	0.72	1.21	1.63	2.31	2.88	–
BMH2	0.38	0.63	0.86	1.13	1.39	1.76	0.71	1.18	1.64	2.20	2.66	3.19
BR	0.57	0.83	1.07	1.42	1.88	2.40	0.68	0.98	1.23	1.74	2.16	2.67
Hash3	0.19	0.47	0.73	1.18	1.59	2.31	0.22	0.55	0.85	1.36	1.82	2.59
EBOM	0.84	1.26	1.50	1.74	1.89	2.17	1.15	1.62	1.77	1.96	2.10	2.39

Table 1. Searching speed of algorithms GB/s (per a single pattern) using English text and patterns on ThinkPad X61s.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	1.41	2.23	2.67	3.23	3.68	4.35	2.15	3.11	3.59	4.13	4.47	5.21
FSB(2,0)	1.24	2.04	2.60	3.24	3.76	4.55	1.99	2.97	3.50	4.09	4.49	5.29
FSB(2,1)	1.12	1.71	2.15	2.79	3.29	4.08	1.52	2.20	2.68	3.34	3.92	4.65
FSB(3,1)	1.03	1.92	2.67	3.77	4.69	6.21	1.86	3.29	4.35	5.69	6.68	7.94
FSB(4,0)	.300	1.16	1.97	3.22	4.33	6.37	.568	2.13	3.51	5.43	7.05	9.51
FSB(4,1)	.565	1.37	2.11	3.28	4.35	6.34	1.42	3.26	4.78	7.02	8.57	10.4
FSB(4,2)	.802	1.56	2.26	3.35	4.44	6.28	1.86	3.48	4.80	6.73	8.28	10.1
FSB(4,3)	.831	1.40	1.95	2.85	3.79	–	1.32	2.16	2.93	4.24	5.51	–
BMH2	.710	1.18	1.60	2.16	2.75	3.52	1.34	2.27	3.13	4.37	5.48	7.03
BR	1.09	1.59	2.06	2.88	3.65	4.78	1.24	1.81	2.35	3.27	4.19	5.43
Hash3	.414	1.02	1.60	2.53	3.39	5.01	.436	1.07	1.67	2.64	3.53	5.23
EBOM	1.23	1.99	2.48	3.07	3.49	4.15	1.60	2.42	2.91	3.45	3.84	4.51

Table 2. Searching speed of algorithms GB/s (per a single pattern) using English text and patterns on Dell T1500 in 32-bit mode using 32 bit bitvectors.

algorithm	speedup
GSB2	1.32
FSB(2,0)	1.34
FSB(2,1)	1.24
FSB(3,1)	1.56
FSB(4,0)	1.72
FSB(4,1)	2.15
FSB(4,2)	2.03
FSB(4,3)	1.51
BMH2	1.96
BR	1.14
Hash3	1.05
EBOM	1.17

Table 3. Average speedup of 2-byte read based on Table 2.

Text 2: DNA. The search speeds are shown in Tables 4 and 5. On DNA data, larger values of q were better than on natural language. On the other hand the probability to fall to the slow loop, i.e. the inner loop of an algorithm, increases with f.

According to Table 4 GSB2 was slightly faster than FSB(2,0) in every case, and FSB(4,1) was better than FSB(4,0) for short patterns. Otherwise the lookahead versions of FSB(4,0) were not significantly better than FSB(4,0). FSB(2,1) was faster than FSB(2,0) for longer patterns, but neither of them was then competitive with faster algorithms.

Table 5 shows that the lookahead versions were in many cases clearly faster than the versions without lookahead for $m = 10, 20, 30$.

Text 3: Binary. The search speeds are shown in Tables 6 and 7. Large values of q were good with binary data.

The relatively good performance of FSB(4,3) in Table 6 is surprising. With FSB(4,3) only one text character comes from the alignment, and therefore the probability to fall to the slow loop is quite high. Among the standard versions, BMH2 was the fastest for $m = 4$.

Results in Table 7 indicate that 8-grams worked best for $m \geq 20$, and lookahead characters gave clear advantage only for $m = 10$.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	0.33	0.49	0.62	0.85	1.10	1.58	0.44	0.59	0.72	0.95	1.24	1.76
FSB(2.0)	0.33	0.47	0.60	0.83	1.06	1.50	0.40	0.55	0.69	0.93	1.21	1.69
FSB(2.1)	0.28	0.46	0.63	0.88	1.14	1.57	0.30	0.49	0.67	0.96	1.22	1.72
FSB(4.0)	0.16	0.57	0.94	1.50	1.94	2.64	0.34	1.17	1.81	2.64	3.04	3.69
FSB(4.1)	0.28	0.66	1.01	1.51	1.93	2.53	0.52	1.19	1.75	2.45	2.86	3.43
FSB(4.2)	0.32	0.63	0.94	1.35	1.72	2.32	0.46	1.01	1.49	2.04	2.48	3.05
FSB(4.3)	0.23	0.44	0.65	1.01	1.27	–	0.30	0.57	0.83	1.27	1.54	–
BMH2	0.32	0.51	0.64	0.84	0.94	1.10	0.48	0.74	0.86	1.20	1.34	1.54
BR	0.25	0.34	0.39	0.54	0.59	0.68	0.27	0.37	0.46	0.59	0.65	0.74
Hash3	0.17	0.41	0.65	1.00	1.31	1.80	0.21	0.50	0.79	1.22	1.59	2.12
EBOM	0.34	0.44	0.54	0.70	0.88	1.20	0.37	0.48	0.58	0.75	0.93	1.27

Table 4. Searching speed of algorithms GB/s (per a single pattern) using DNA text and patterns on ThinkPad X61s.

patterns→	10	20	30	40	50	60	10	20	30	40	50	60
↓algorithm	standard version						b-version with 2-byte read					
GSB2	1.19	2.01	2.92	3.80	4.69	5.60	1.22	2.14	3.08	3.91	4.90	5.69
FSB(4,0)	2.25	4.42	5.72	6.62	7.37	8.26	3.42	5.79	6.91	7.96	8.68	9.66
FSB(4,1)	2.25	4.19	5.37	6.40	7.13	7.88	3.41	5.65	6.66	7.78	8.53	9.48
FSB(5,0)	1.81	4.46	6.55	8.37	9.53	10.9	2.77	6.27	8.52	10.5	11.8	13.2
FSB(5,1)	2.04	4.59	6.63	8.34	9.51	10.7	3.05	6.40	8.61	10.4	11.9	13.3
FSB(6,0)	1.26	3.62	5.76	7.77	9.44	11.0	2.49	6.88	10.2	12.6	14.5	16.6
FSB(6,1)	1.55	3.95	6.18	8.17	9.69	11.2	2.92	7.16	10.3	12.8	14.5	16.8
FSB(6,2)	1.76	4.11	6.29	8.20	9.72	11.2	3.20	7.14	10.2	12.5	14.4	16.1
FSB(6,3)	1.86	4.07	6.05	7.89	9.31	10.8	3.07	6.59	9.17	11.4	13.5	15.2
FSB(7,0)	.882	3.02	5.04	7.00	8.68	10.2	1.56	5.25	8.53	11.5	13.1	15.2
FSB(7,1)	1.10	3.23	5.21	7.16	8.89	10.3	1.93	5.60	8.96	11.6	13.2	15.0
FSB(7,2)	1.31	3.38	5.34	7.23	8.86	10.4	2.27	5.81	9.06	11.6	13.4	15.1
FSB(7,3)	1.49	3.54	5.46	7.33	8.93	10.4	2.57	6.04	9.13	11.5	13.1	14.7
BMH2	1.27	1.89	2.15	2.41	2.45	2.60	1.89	2.79	3.16	3.55	3.59	3.81
BR	.805	1.11	1.26	1.43	1.44	1.50	.859	1.20	1.35	1.53	1.55	1.60
Hash3	1.35	2.72	3.67	4.41	4.93	5.41	1.36	2.90	3.97	4.83	5.39	5.93
EBOM	1.09	1.76	2.38	2.99	3.51	4.09	1.13	1.84	2.46	3.08	3.66	4.20

Table 5. Searching speed of algorithms GB/s (per a single pattern) using DNA text and patterns on Dell T1500. Speeds are averages of 300 runs with 64-bit code.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	.127	.213	.301	.457	.588	.917	.156	.238	.322	.485	.578	.946
FSB(2,0)	.137	.210	.289	.434	.561	.878	.127	.211	.299	.458	.578	.937
FSB(2,1)	.139	.219	.309	.465	.600	.905	.125	.218	.319	.484	.645	.962
FSB(4,0)	.131	.249	.334	.470	.606	.886	.244	.321	.426	.571	.721	1.03
FSB(4,1)	.146	.243	.337	.481	.621	.902	.178	.298	.414	.574	.736	1.05
FSB(4,2)	.128	.228	.327	.479	.631	.918	.145	.264	.393	.579	.750	1.07
FSB(4,3)	.124	.222	.323	.492	.652	–	.121	.239	.366	.548	.724	–
BMH2	.164	.187	.199	.208	.215	.212	.207	.231	.247	.246	.267	.263
BR	.115	.120	.130	.127	.130	.129	.129	.136	.132	.140	.140	.139
Hash3	.110	.204	.265	.331	.356	.384	.124	.226	.294	.367	.394	.425
EBOM	.122	.175	.231	.331	.430	.618	.120	.180	.237	.335	.437	.633

Table 6. Searching speed of algorithms GB/s (per a single pattern) using binary text and patterns on ThinkPad X61s.

patterns→	10	20	30	40	50	60	10	20	30	40	50	60
↓algorithm	standard version						b-version with 2-byte read					
GSB2	.586	1.15	1.68	2.20	2.70	3.20	.574	1.15	1.70	2.24	2.77	3.29
FSB(4,0)	.661	1.16	1.73	2.31	2.88	3.44	.739	1.30	1.91	2.53	3.09	3.68
FSB(4,1)	.632	1.19	1.76	2.34	2.90	3.46	.722	1.34	1.96	2.57	3.16	3.72
FSB(5,0)	.846	1.32	1.76	2.29	2.81	3.36	.978	1.48	1.98	2.59	3.19	3.78
FSB(5,1)	.815	1.31	1.76	2.28	2.82	3.33	.945	1.48	2.03	2.63	3.24	3.84
FSB(6,0)	.870	1.66	2.14	2.60	3.03	3.48	1.27	2.12	2.59	3.02	3.52	4.03
FSB(6,1)	.960	1.73	2.19	2.63	3.07	3.53	1.29	2.10	2.56	3.07	3.57	4.03
FSB(6,2)	.909	1.64	2.14	2.59	3.06	3.54	1.16	1.97	2.49	2.99	3.53	4.09
FSB(6,3)	.776	1.46	1.99	2.50	3.01	3.53	.935	1.73	2.31	2.89	3.49	4.07
FSB(7,0)	.755	1.99	2.80	3.39	3.86	4.29	1.22	2.85	3.72	4.33	4.69	5.13
FSB(7,1)	.874	2.03	2.77	3.39	3.84	4.32	1.38	2.87	3.73	4.29	4.69	5.04
FSB(7,2)	.935	1.99	2.71	3.32	3.82	4.26	1.40	2.74	3.54	4.14	4.50	4.95
FSB(7,3)	.883	1.83	2.53	3.14	3.65	4.16	1.24	2.45	3.20	3.81	4.26	4.78
FSB(7,4)	.787	1.58	2.25	2.84	3.40	3.91	.970	1.96	2.71	3.35	3.88	4.41
FSB(8,0)	.543	2.02	3.14	4.06	4.76	5.36	1.05	3.45	4.99	6.00	6.49	7.16
FSB(8,1)	.696	2.10	3.18	4.05	4.77	5.35	1.30	3.53	5.01	5.93	6.56	7.14
FSB(8,2)	.812	2.14	3.17	4.00	4.71	5.28	1.45	3.49	4.90	5.84	6.39	6.90
BMH2	.369	.375	.373	.371	.383	.384	.496	.502	.500	.497	.511	.513
BR	.228	.214	.232	.223	.222	.233	.244	.229	.248	.239	.237	.248
Hash3	.610	.820	.834	.796	.832	.865	.613	.826	.847	.843	.872	.874
EBOM	.422	.767	1.09	1.37	1.64	1.87	.436	.781	1.11	1.39	1.68	1.91

Table 7. Searching speed of algorithms GB/s (per a single pattern) using binary text and patterns on Dell T1500. Speeds are averages of 300 runs with 64-bit code.

Other processors. We tested the algorithms also in several other computers having a x86 processor (Pentium III or newer). The relative performance of the algorithms was mostly similar. The only exception was Atom N450, on which BMH2b was a clear winner.

Memory usage and preprocessing time. All b-versions using 2-byte read require additional 262 kB (bitvectors of 32) or 524 kB (bitvectors of 64). The initialization of the additional table takes about 15–20 milliseconds per 200 patterns. Preprocessing of Forward-SBNDM(q, f) is more laborious when $f > 0$. In our tests the preprocessing time increased at most 6 %.

6 Concluding remarks

For long we believed that the tuned algorithms of Hume and Sunday [13] were the final solution for exact string matching of natural language. Only long patterns offered space for improvement. But the development of processor technology changed the situation: new algorithms, especially those applying bit-parallelism, can be much faster than the old ones.

In this paper, we have presented a generalization of the Forward-SBNDM algorithm and introduced the Greedy-SBNDM2 algorithm. We have shown that the new algorithms are competitive for several pattern lengths in three types of text. Generally the number of lookahead characters f has smaller influence than the q-gram size. Lookahead characters can appreciably increase the shift length in the case of pattern lengths $q - f \leq m \leq 3q$ and thus give significant improvement to the search speed.

In addition we tested the effect of 2-byte read. The speedup of 2-byte read varied from a few percents to over two. It is clear that 2-byte read should be used whenever it is possible.

When comparing the search speed of two string matching algorithms, several factors affect the result: processor, compiler, stage of tuning, text, pattern. Even a small change in the pattern may switch the order of the algorithms. Thus there is no absolute truth which algorithm is better. Because the continuing development of processor and compiler technologies, it is also difficult to anticipate, how present algorithms manage after a few years. We have experienced several times how the speed order of old algorithms has changed when switching to a new computer.

References

1. R. Baeza-Yates: Improved string searching. *Softw. Pract. Exp.*, 19(3):257–271, 1989.
2. R. Baeza-Yates and G. Gonnet: A new approach to text searching. Commun. ACM 35(10):74–82, 1992.
3. T. Berry and S. Ravindran: A fast string matching algorithm and experimental results. Proc. of the Prague Stringology Club Workshop '99, Czech Technical University, Prague, Czech Republic, Collaborative Report DC-99-05, pp. 16–28, 1999.
4. R. S. Boyer and J. S. Moore: A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
5. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: Tuning BNDM with q-grams. In *Proc. ALENEX09, Tenth Workshop on Algorithm Engineering and Experiments*: 29–37, 2009.
6. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: Improving practical exact string matching. *Information Processing Letters* 110(4):148–152, 2010.
7. S. Faro and T. Lecroq: Efficient variants of the backward-oracle-matching algorithm. *International Journal of Foundations of Computer Science* 20(6): 967–984, 2009.
8. S. Faro and T. Lecroq: An efficient matching algorithm for encoded DNA sequences and binary strings. In *Proc. CPM 2009, Combinatorial Pattern Matching, 20th Annual Symposium*, LNCS 5577: 106–115, Springer, 2009.
9. S. Faro and T. Lecroq: The exact string matching problem: a comprehensive experimental evaluation. CoRR abs/1012.2547, 2010.
10. K. Fredriksson: Shift-or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003.
11. J. Holub and B. Ďurian: Fast variants of bit parallel approach to suffix automata. Presentation in: *The Second Haifa Annual International Stringology Research Workshop.*
12. R. N. Horspool: Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
13. A. Hume and D. M. Sunday: Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.
14. P. Kalsi, H. Peltola, and J. Tarhio: Exact string matching algorithms for biological sequences. In *Proc. BIRD 2008, 2nd International Conference on Bioinformatics Research and Development*, Communications in Computer and Information Science 13:417–426, Springer, 2008.
15. J. Y. Kim and J. Shawe-Taylor: Fast string matching using an n-gram algorithm. *Softw. Pract. Exp.*, 24(1):79–88, 1994.
16. T. Lecroq: Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.
17. G. Navarro and M. Raffinot: Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.
18. H. Peltola and J. Tarhio: Alternative algorithms for bit-parallel string matching. In *Proc. SPIRE'03, 10th International Conference on String Processing and Information Retrieval*, *Lecture Notes in Computer Science* 2857:80–93, 2003.
19. D. M. Sunday: A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
20. J. Tarhio and H. Peltola: String matching in the DNA alphabet. *Softw. Pract. Exp.*, 27(7):851–861, 1997.
21. R. F. Zhu and T. Takaoka: On improving the average case of the Boyer–Moore string matching algorithm. *Journal of Information Processing*, 10(3):173–177, 1987.

On Compile Time Knuth-Morris-Pratt Precomputation

Justin Kourie[1], Bruce Watson[2,1], and Loek Cleophas[3,1]

[1] FASTAR Research Group, Department of Computer Science, University of Pretoria, 0002 Pretoria, Republic of South Africa
(justin@fastar.org)
[2] FASTAR Research Group, Centre for Knowledge Dynamics and Decision-making, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa
(bruce@fastar.org)
[3] Software Engineering & Technology Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
(loek@fastar.org)

Abstract. Many keyword pattern matching algorithms use precomputation subroutines to produce lookup tables, which in turn are used to improve performance during the search phase. If the keywords to be matched are known at *compile time*, the precomputation subroutines can be implemented to be evaluated at compile time versus at run time. This will provide a performance boost to run time operations. We have started an investigation into the use of metaprogramming techniques to implement such compile time evaluation, initially for the Knuth-Morris-Pratt (KMP) algorithm. We present an initial experimental comparison of the performance of the traditional KMP algorithm to that of an optimised version that uses compile time precomputation. During implementation and benchmarking, it was discovered that C++ is not well suited to metaprogramming when dealing with strings, while the related D language is. We therefore ported our implementation to the latter and performed the benchmarking with that version. We discuss the design of the benchmarks, the experience in implementing the benchmarks in C++ and D, and the results of the D benchmarks. The results show that under certain circumstances, the use of compile time precomputation may significantly improve performance of the KMP algorithm.

Keywords: Knuth-Morris-Pratt algorithm, compile time precomputation, metaprogramming, string processing time

1 Introduction

Keyword pattern matching is a mature field in computing science which has produced a large number of efficient keyword matching algorithms [4,10,7]. Such algorithms play a central role in a wide range of research domains such as molecular biology, information retrieval, pattern recognition, compiling, data compression, program analysis and security [8].

Taxonomies of keyword pattern matching algorithms as well as the SPARE Parts and SPARE Time toolkits implementing these taxonomies have been described in [13,2,3]. One of the benefits provided by these taxonomies is that they reveal commonalities between the algorithms and group them accordingly in the overall taxonomy. Of particular interest to this research are the common precomputation subroutines shared by various pattern matching algorithms.

Generally speaking, such precomputation subroutines take as input the keywords to be matched by the primary algorithms and produce *lookup tables* as output. The lookup tables are then used by the primary matching algorithms when a mismatch

Justin Kourie, Bruce Watson, Loek Cleophas: On Compile Time Knuth-Morris-Pratt Precomputation, pp. 15–29.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

between the target keywords and text occurs to proceed more intelligently than primary algorithms not using such precomputed lookup tables. The precomputation subroutines which create the lookup tables are evaluated at *run time*. If however, the keywords to be matched are known at *compile time*, the precomputation subroutines can be implemented to be evaluated at compile time versus at run time. This will provide a performance boost to run time operations. Such compile time evaluation can be achieved using techniques such as metaprogramming or partial evaluation, depending on the implementation language being used.

We have initiated a research endeavour to investigate the application of metaprogramming in implementing such precomputation algorithms. In doing this, the magnitude of performance gains as well as the challenges and drawbacks to the metaprogramming approach will be explored. As a starting point of a broader investigation, this paper considers the classical Knuth-Morris-Pratt (KMP) pattern matching algorithm [6,13] as a case study for an experimental initial benchmark. The objective of our experiments was to investigate whether compile time evaluation of precomputation KMP subroutines could be profitable to KMP keyword pattern matching.

Experimentation is clearly a suitable approach to employ in pursuing this objective. As such, we constructed an experimental benchmark based on the KMP algorithm, to provide the data required to analyse both the advantages and disadvantages of compile time precomputation subroutines. In implementing the two variants of the algorithm to be benchmarked, we initially chose C++ as an implementation language, based on its support for metaprogramming as well as our previous experience in developing SPARE Parts [14] and SPARE Time [2] (both having been implemented using this language). However, our initial experiments showed that C++ does not have the compile time string processing capacity required to implement the type of benchmarks we had in mind for the research.

As a result, we opted to port our implementation to the related language, D. The resulting benchmark in D compares the performance of the traditional KMP algorithm with that of an optimised version which performs the precomputation of its lookup table at compile time.

It should be noted that our primary motivation at this stage is not a desire for massive performance gains. Rather, we focus on understanding the practical requirements involved in optimising a traditional pattern matching algorithm at compile time. As side effects of this focus, we find some interesting scenarios where such optimisations can be justified.

Furthermore, we contrast our initial implementation in C++ with our latter D implementation where appropriate, not as a language debate, but rather to draw attention to how important it is to use the right tool for the job in implementing the type of pattern matching optimisation our research is dealing with.

1.1 Overview

We present some basic definitions in Section 2. Thereafter, an overview of the experiment's design is given in Section 3 before briefly discussing some of the issues encountered during implementation in Section 4. Section 5 presents some hypotheses, the results of our experiments, and an analysis of both. Finally, Section 6 presents concluding remarks and ideas for future work.

2 Basic Definitions

Notational conventions used are as follows. Array subscripts are assumed to be 0-based. A subarray of array A over the range $[i, j)$ is denoted by $A_{[i..j)}$. Textual input is taken from some alphabet Σ. The text used is denoted by $x \in \Sigma^+$. A set of keywords, $K \subseteq \Sigma^+$ is also used, such that $\forall k \in K : (|k| \leq |x|)$.

2.1 Algorithmic Computations

For any algorithm a, we denote by $T(a)$ the time measured in milliseconds which it takes a to complete its execution.

In essence, the primary search algorithm of Knuth-Morris-Pratt uses a precomputed lookup table when a mismatch between the target keyword and the text occurs. This allows forward shifts of more than one position in the text to occur, hence leading to more efficient matching than in a naive algorithm. The KMP algorithm's precomputation function take as input the keywords to be matched by the primary algorithms and produce the lookup tables as output. We assume the reader to be familiar with the details of the primary and precomputation algorithms, and do not present them in detail here. Rather, we assume the following:

Precomputation KMP_{pre} denotes the precomputation function, mapping a keyword $k \in K$ to a lookup table LT_k for keyword k. KMP_{pre} defines the function at the heart of the benchmark. Not only is it timed individually for analysis, it also is used by both the run time and compile time variants of the KMP algorithm. Descriptions of KMP_{pre} and LT_k can be found in e.g. [15,6].

Main Search $KMP_{main}(LT,\ k,\ x,\ CB)$ is the main search procedure implementing the KMP algorithm; a procedure which searches for keyword k in text x aided by LT_k and, if a match of keyword k occurs at x_i, evaluates function $CB(i)$ to determine how to proceed. In this variant, the procedure yields control flow to some *callback function CB* whenever a match occurs—passing the index of the matched keyword to *CB*. *CB* then performs some custom operations specific to the particular *CB* received as input. If *CB* returns **true** after having completed its operations, KMP_{main} resumes its search from where it left off. If however, *CB* returns **false** the search is aborted.[1]

Traditional KMP Algorithm $KMP_{trad}(k,\ x,\ CB)$ defines a procedure for the traditional KMP pattern matching algorithm, which constructs LT_k at run time and then executes the main search. This defines the traditional KMP algorithm to be benchmarked against its optimised counterpart.

Optimised KMP Algorithm For each $k \in K$, procedure $KMP^k_{opt}(x,\ CB)$ which can search only for k in x but for which LT_k is predefined. This defines the optimised KMP pattern matching algorithm, which precomputes its lookup table for some $k \in K$ at compile time using metaprogramming.

3 Designing the Benchmark

Having defined its constituent terms, the benchmark's design can now be described. In doing so a simplified data flow diagram is described to give an overview of the

[1] Note that because matches never occur in our benchmarking context (as no $k \in K$ occurs in x), *CB* is never actually evaluated. As the former is not the case in typical KMP usage, we nevertheless include the function here. Many practical implementations of pattern matching algorithms, including the ones in SPARE Parts [14] and SPARE Time [2], use a callback function.

benchmark's process flow. This approach makes the description more concise whilst distancing itself from implementation specific details. The design does however, assume a programming paradigm which will allow for KMP_{pre} to be evaluated at compile time—as this is the fundamental optimisation being investigated.

Before describing the data flow diagram, several further definitions are required. They have been defined here due to their lower-level nature and direct relevance to the benchmark's data flow.

This benchmark is unorthodox in that it requires a highly generic approach to its compilation process. Specifically, in order to analyse a wide range of different output data, it must be possible to change the values of all $k \in K$ arbitrarily at compile time. The design therefore incorporates a *seed string* or *seed* s, not occurring in text x, to serve as variable input to the compilation process itself.[2] The seed acts as a catalyst in determining the generation of K, as will be discussed shortly.

Definition 1 (Program Code) *Let PC be the benchmark's program code after all metaprogramming has been evaluated. As such, KMP^k_{opt} for $k \in K$ as well as the timed instructions necessary to construct the output data Ω (see below) are defined in PC.*

PC can be seen as an intermediary artefact consisting of the code defined by the programmer and the code generated by the compiler after all metaprogramming code has been evaluated.

Definition 2 (Benchmark Binary) *B is defined as the fully compiled binary representation of PC.*

Whereas *PC* is an intermediary artefact, B on the other hand is a fully compiled program which is ready to be executed.

The set of output data generated by execution of our benchmark B, called Ω, consists of three parts:

- *Precomputation timing data*, pairing a given $k \in K$ and the time taken to compute LT_k. We denote the bag (multiset) of such pairs by P_Ω.
- *Traditional KMP search timing data*, pairing the length of a given $k \in K$ and the time taken to compute $KMP_{trad}(k,\ x,\ CB)$ (where x and CB are assumed to be fixed for the entire benchmark). We denote the multiset of such pairs by T_Ω.
- *Optimised KMP search timing data*, pairing the length of a given $k \in K$ and the time taken to compute $KMP^k_{opt}(x,\ CB)$ (where x and CB are again assumed to be fixed). We denote the multiset of such pairs by O_Ω.

Note 3 (Shifts by One) *It is important to note that in our benchmarks, we are interested in determining the differences between the running times of the traditional KMP algorithm and its variant for which precomputation has been performed at compile time. For both variants, the same keyword set K and text x are used, and hence the same shifts are used in both cases and the KMP* search *time will not differ among the two. We are therefore not concerned with whether the particular benchmark keyword set K and text x guarantee a certain behaviour of the Knuth-Morris-Pratt search algorithm per se, e.g. worst-case or average case behavior. To have consistent performance, we opted to always use a seed string s and text x such that the (sub-)alphabet whose characters occur in s and that whose characters occur in x are disjoint. As a*

[2] This can be achieved for example by using a compiler directive.

consequence of this choice, a mismatch will always occur on the first comparison in the KMP search algorithm, and the shift applied in the main text will always equal 1*. As noted above, the actual choice of text, keyword set and shifts applied does not matter, as long as the algorithms are compared on the same combination of text, keyword set and shifts.*

Definition 4 (Benchmark Pipeline) *Let BP denote the data flow and state transitions in the benchmark. This "benchmark pipeline" (depicted in Figure 1) operates as follows:*

- *The first compilation state, C_1, receives seed string s as input and generates keyword set K of size $n = |s|$ as output, such that:*

$$k_1 = s_{[0..1)},\ k_2 = s_{[0..2)},\ \ldots,\ k_n = s_{[0..|s|)}$$

- *The second compilation state, C_2, receives keyword set K as input and then evaluates all metaprogramming before generating PC as output.*
- *The third compilation state, C_3, receives PC as input and compiles benchmark binary B as output.*
- *The benchmark is then run in the last state, R. After loading x into memory as input and executing its timing instructions, B yields Ω for analysis.*

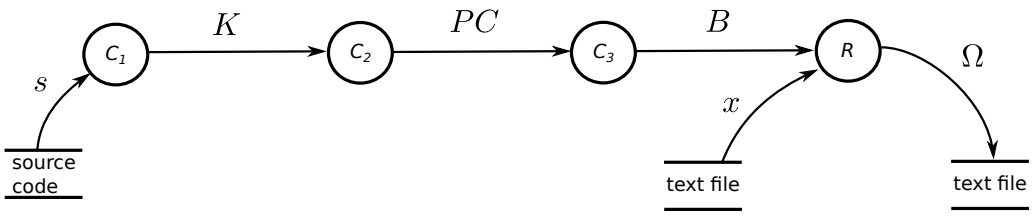


Figure 1: The Benchmark Pipeline

Observe that the text x used in the benchmark may be varied over runs, and that the benchmark can also be recompiled for different values of seed s. These two observations essentially provide the desired flexibility required to generate a wide range of data for analysis.

4 Implementing the Benchmark

The benchmark was initially implemented in C++ in order to extend the SPARE Parts toolkit [13] to include a compile time optimised KMP search. In this implementation, Boost's Meta Programming Library (MPL) [12] was used to support compile time string operations. Unfortunately, it turned out that even with the use of this library, C++ metaprogramming proved to be unsuitable. The benchmark was then ported to D and the resulting implementation was used to perform the benchmarking instead [5]. As the design and class structure of the D and the C++ implementations hardly differ, we do not present that of the C++ implementation here, but only discuss the problems with that implementation, the choice for D over C++, and the design of the D implementation.

4.1 Problems with C++ implementation

Despite being both powerful and flexible, C++ metaprogramming has never supported compile time string operations out of the box. Though excellent supporting libraries such as Boost's MPL enable this ability, its intrinsically constrained nature is the primary reason for abandoning the C++ effort.

Table 1 below summarises the problems encountered with the C++ implementation of the benchmark. As can be seen, four out of the five problems relate to compile time string operations—a feature not provided by and completely unsuited to the design of C++. Out of all the factors listed, the huge performance issues with large strings proved to be the turning point in the implementation effort. After precompiling headers to save overhead, expanding the system's virtual memory, increasing the kernel's default memory map allocation for processes and one too many heap exhaustions—it became obvious that another language should be pursued.

Problem	**Description**
Constrained string length	Length restrictions, due to performance limitations of the Boost MPL, fundamentally constrain $\|K\|$, which means that a thorough analysis of Ω cannot be done.
Intrinsic string overhead	The overhead required just to declare MPL strings is relatively high due to the complex hackery which enables the feature.
Maximum string overhead	Changing the maximum string length (defaulting to 32) to be above 128 characters results in critical compiler overhead. When setting the length to be greater than around 228 characters, heap exhaustion was repeatedly experienced.
Poor string writability	MPL string syntax makes it tedious to change s and recompile the benchmark with different input.
Array initialisation	Initialising an array with variant compile time data values is a fundamentally tricky problem which either requires potentially exponential use of the preprocessor, or language features which are not part of the current C++ standard.

Table 1: Summary of C++ Implementation Issues

4.2 Choice for D implementation

We selected the D language [5] for implementation following our experience with the C++ implementation. D was designed and originally implemented by Walter Bright and belongs to the family of C/C++/Objective-C. The main intent behind its design was to improve on C++ by being a cleaner and smaller language. D offers powerful language features which address all the issues and challenges we encountered using C++, while its similarity to C++ made our benchmark easy to port.

As D fully supports both object orientation and templates, just like C++ does, the design and object model used for the C++ implementation could be reused without modification. The port of the code itself turned out to be trivial—with any major differences between the two languages actually making the implementation easier than before. For example, D's `foreach` construct and auto type inferencing remove the need for a lot of boiler plate code in loops and type declarations.

The most striking justification for using D over C++ however, is the complete absence of the obstacles encountered with metaprogramming in C++. Firstly, D provides native support for compile time string operations. This means that no constraints on string length are placed outside of the standard system limits, and that

writability is no longer an issue either. Secondly, and more importantly, D offers two approaches to metaprogramming:

Template Metaprogramming Template metaprogramming in D employs many techniques similar to those used in C++ (e.g., repetition and selection can be affected through template specialisation) but is far more powerful. Templates can serve as generic namespaces and support a much broader range of template parameters than their C++ counterparts. In addition, a compile time selection statement can be used instead of template specialisation, which avoids considerable overhead.

Compile Time Function Evaluation (CTFE) D's CTFE feature on the other hand, embeds an interpreter in the compilation environment and allows the programmer to direct it to evaluate ordinary functions at compile time. The feature does require that functions meet certain constraints in order to be evaluated, but the constraints are liberal enough to allow for KMP_{pre} to be implemented as a normal, imperative function. No template metaprogramming is required.

4.3 Implementation Overview

The architecture of the benchmark illustrates how the process states defined by *BP* (see Definition 4) are realised.

As presented here, the architecture is fairly technology neutral, and only assumes support for objects and compile time metaprogramming. The object model itself is straightforward and easy to understand as illustrated in Figure 2. The D benchmark uses a slightly simplified design: as indicated before, it uses D's Compile-Time Function Evaluation. As a consequence, the use of a specific C++ compile time precomputation implementation—`CT_KMP_pre`—is replaced by the *compile time use* of the traditional run time precomputation implementation—`RT_KMP_pre`.

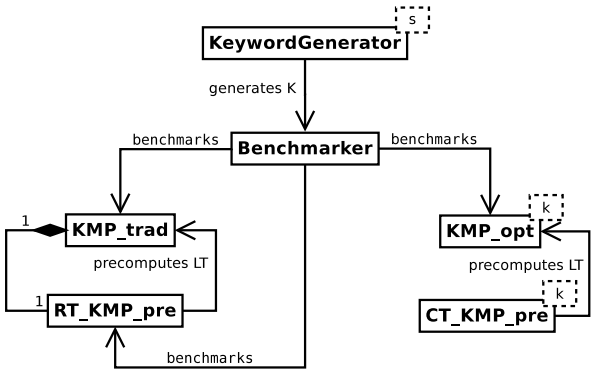


Figure 2: Benchmark Reference Model

The object model essentially consists of three parts, corresponding to control functionality, functionality for the traditional KMP search, and functionality for the optimised KMP search.

Control: This is implemented by classes `KeywordGenerator` and `Benchmarker`. Class `KeywordGenerator` is a template class which wraps the metafunctions[3] used to gen-

[3] "Metafunctions" here refer simply to the techniques employed to operate on data at compile time. For a more detailed definition, and an excellent discussion on what this means in the context of C++, see Chapters 1 and 2 of [1].

erate K from the compile time string s. `KeywordGenerator` thus implements the process defined in state C_1 of BP.

`Benchmarker` implements the control logic which ties the various aspects of the benchmark together. It also implements the timing and output code required. After obtaining K from `KeywordGenerator` at compile time, `Benchmarker` uses it to generate the code and evaluate the metafunctions required to produce PC (see Definition 1). `Benchmarker` thus implements the process defined in state C_2 of BP.

Furthermore, `Benchmarker` loads x from a file at run time and directly controls how Ω is produced (e.g., whether Ω is written to standard output or to file). `Benchmarker` therefore also implements the processes defined in state R of BP.

Optimised KMP search: `KMP_opt` is a template class used by `Benchmarker` to instantiate KMP^k_{opt} for each $k \in K$. For each given k template parameter the resulting template instantiation uses the compile time evaluation of `RT_KMP_pre` for that specific k (in the case of D) or uses `CT_KMP_pre` (in the case of C++) to generate LT_k. `Benchmarker` then times the search functions of objects instantiated for each generated class and constructs O_Ω as a result.

`CT_KMP_pre` is a template class which generates LT_k from a compile time string k. In the C++ implementation, the class is used as a delegate by `KMP_opt` in compiling LT_k into each of `KMP_opt`'s generated classes.

Traditional KMP search: `KMP_trad` is an ordinary class implementing the procedure defined by KMP_{trad}. `Benchmarker` instantiates a `KMP_trad` object for each $k \in K$. In doing so, the creation of each LT_k is delegated to `RT_KMP_pre` at run time. `Benchmarker` times both the delegated request and the ensuing search in order to create T_Ω.

`RT_KMP_pre` implements the run time version of the function defined by KMP_{pre}. `Benchmarker` instantiates the class for each $k \in K$. Each object has a function which returns LT_k for the k it was constructed with. By timing these operations separately, `Benchmarker` creates the remaining dataset P_Ω.

5 Results Analysis and Interpretation

In this section, we discuss the results obtained using the benchmarking. First, we present a number of hypotheses to guide the results analysis, as well as an overview of the benchmarking platform used. We then present the results together with our analysis and interpretation of them.

5.1 Hypotheses

The benchmark is designed to output data such that the relationship between the time, t, taken to search for a keyword k, and the keyword's length $|k|$ can be examined. In order to direct the analysis several hypotheses are proposed. Two of the hypotheses (called S_1 and S_2 below) act as sanity tests to verify the correct functioning of the benchmark. The other two (called P_1 and P_2 below) are postulated in the hope that more understanding can be gained as to when exactly compile time optimisation can prove useful to keyword pattern matching. Figure 3 helps conceptualise the hypotheses which follow in presenting a hypothetical plot of the datasets contained in Ω.

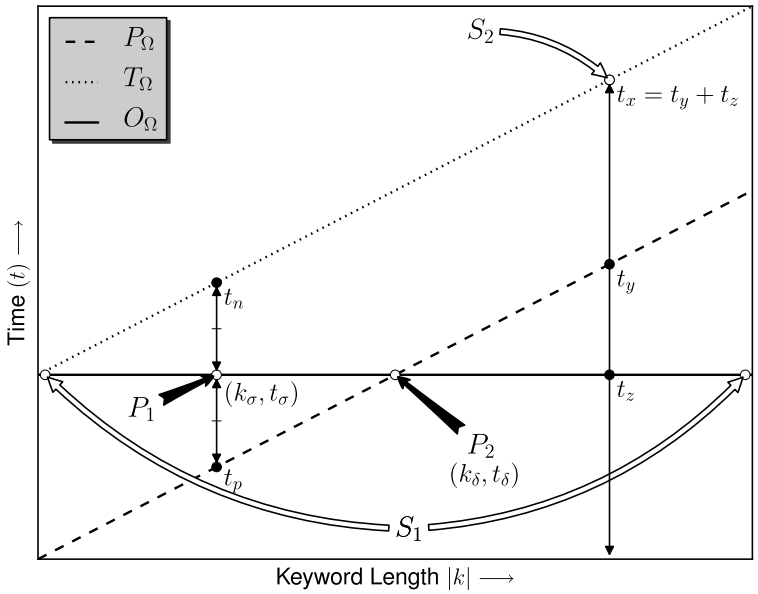


Figure 3: Hypothetical Results

Hypothesis 5 (S_1)

$$\forall (k_i, t_i) \in O_\Omega : \Big[\Big(\forall (k_j, t_j) \in O_\Omega : (t_i = t_j)\Big) \wedge \Big(\forall (k_i, t_p) \in T_\Omega : (t_i < t_p)\Big)\Big]$$

All time values in the pairs belonging to the multiset O_Ω are equal. Furthermore, all such time values are less than all time values in the pairs belonging to T_Ω.

Because each KMP^k_{opt} already has LT_k defined by definition, and because the same x and CB are used when measuring all search times; the time taken to perform the optimised KMP search must be constant. As mentioned before, the alphabet used for text x and that used for seed s and hence keywords k_i are disjoint, hence the main search time depends on the length of text x, but not on any of the keywords k_i. In other words, the same (arbitrary case) search is repeated by each KMP^k_{opt}—therefore the times of those searches must be the same.

Furthermore, since KMP_{trad} always computes some LT_k, its total running time for the same values of x and CB must take longer than any such search for which LT_k has been predefined, i.e. using KMP^k_{opt}. This explains the second part of the hypothesis.

Corollary 6 (S_2)

$$\forall (k_i, t_x) \in T_\Omega, (k_i, t_y) \in P_\Omega, (k_i, t_z) \in O_\Omega : \Big(t_x = t_y + t_z\Big)$$

Each time value in each pair belonging to T_Ω is equivalent to the sum of the time values in the corresponding pairs belonging to P_Ω and O_Ω respectively.

Following from the first hypothesis, KMP_{trad} always takes longer than KMP^{k}_{opt} by the time it takes to compute LT_k for any $k \in K$.[4]

The discussion of S_1 and S_2 above should make it clear that these predicates should hold at all points throughout the benchmark. Any marked deviance from these conditions is a sign that something has gone awry. Of course they cannot be expected to hold perfectly true in practice, due to systemic factors that affect time measurement (e.g., OS scheduling, CPU instruction caching etc.). As a result, *minor* deviances from these sanity tests will be ignored. However, major deviances flag potential implementation problems. In our implementation and benchmarking efforts, such deviances appeared when using the C++ implementations of the traditional and optimised KMP algorithm. This lead us to investigate the suitability of C++ for our comparison, eventually leading to the causes for its insuitability as listed in Section 4.1, and to our abandonment of C++ in favour of D as the implementation language. As the results in the next section will show, no major deviances from the above 'sanity check' predicates were observed with the D implementation.

Hypothesis 7 (P_1)

$$\exists(k_\sigma, t_\sigma) \in O_\Omega, (k_\sigma, t_n) \in T_\Omega, (k_\sigma, t_p) \in P_\Omega : \Big(t_n - t_\sigma = t_\sigma - t_p\Big)$$

A $k_\sigma \in K$ exists such that $T(KMP^{k_\sigma}_{opt}(x,\ CB))$ is faster than $T(KMP_{trad}(k_\sigma,\ x,\ CB))$ by the same amount as it is slower than $T(KMP_{pre}(k_\sigma))$.

Though an optimised search is always faster than a traditional search for the same keyword in the same text, the question that lingers is "when does the dividend gained really start to matter?". A heuristic is introduced here to try and answer that question.

The keyword length for which the traditional search is exactly equal to the precomputation time plus optimal search time represents an interesting boundary value. It is shown as P_1 in Figure 3. Note that at this point, precomputation time is exactly half of the optimised search time, so that the traditional search time is equal to 1.5 times the optimised search time. The definition below is one way of expressing the relationship between these respective search- and precomputation times.

Definition 8 (Beneficial Heuristic (Benheur) Equation)

$$t_\sigma = \frac{t_n + t_p}{2}$$

Let t_σ be a heuristic aid in determining when optimisation may be useful where:

- *t_σ is the time taken by an optimised search for a given $k \in K$.*
- *t_n is the time taken by a traditional search for $k \in K$.*
- *t_p is the time difference between t_n and t_σ.*

Hypothesis 9 (P_2)

$$\exists(k_\delta, t_\delta) \in O_\Omega, (k_\delta, t_i) \in P_\Omega : (t_\delta = t_i)$$

There exists some $k_\delta \in K$ such that $T(KMP^{k_\delta}_{opt}(x,\ CB)) = T(KMP_{pre}(k))$.

[4] Note that this assumes an implementation language that is as efficient in its compile time code execution as in its run time code execution, which may not always be the case for a language such as C++.

If it is taken as a given that a keyword being searched for is known at compile time, and it is shown that an optimised search can be completed before the traditional search even begins; there arises a strong argument *against* using the traditional search. The following definition is made for completeness:

Definition 10 (Delta Point)

$$t_\delta = t_i$$

Let t_δ be the delta point where compile time optimisation becomes preferable to traditional searching for known keywords where:

- *t_δ is the time taken by an optimised search for a given $k \in K$.*
- *t_i is the difference between the traditional search and the optimised search.*

5.2 Benchmarking Platform

Table 2 summarises the details of the benchmarking platform. All benchmarking was run in a minimal environment with only essential services running. Furthermore, one of the CPU cores was allocated to run the benchmark's system process in isolation, with the rest of the processes guaranteed to execute only on the other core. This is easily achieved using the `taskset` [11] utility. Another utility, `schedtool` [9] was used to reassign a FIFO scheduler policy to the benchmark's process. By disabling pre-emptive scheduling for the process, much more representative sample data could be obtained due to minimal extraprocess interference.

Architecture:	Intel Core2 6400 @2.13 GHz
Operating System:	Linux 2.6.34 (Gentoo sources release 6)
RAM:	2 GB
C++ Compiler:	GCC 4.5.1 (Gentoo patches 1.1)
D Compiler:	Digital Mars dmd 2.0

Table 2: Platform Specification

5.3 Results Interpretation

As seen in Figure 4, the C++ benchmark performed substantially faster than the D benchmark. This may be due (in part) to the decision to reduce the D compiler's optimisation level (which was interfering with the sample data's consistency). The C++ results however, are not even in the order of the bounds defined by the sanity check implied by S_2. Due to the very small inputs being benchmarked, it is suspected that language implementation factors (e.g., the time of object construction) were responsible for disproportionally tainting the output dataset. Due to the limited scope of the C++ results, further investigation was deemed unnecessary.

Interestingly, the data we analysed contained a point where the heuristic given in Definition 8 holds precisely. This is shown by the three squares in Figure 5a (page 27). The delta point mentioned in Definition 10 is seen also to occur in the order of where it was predicted. Though such relations remained approximately constant throughout our results, the results are not generalisable due to the arbitrary case analysis of the benchmark.

It was also noted during our analysis that the ratio between $|x|$ and $|k|$ has some practical implications. Take for example $|x|$ and $|k|$ in Figure 5a around the point of

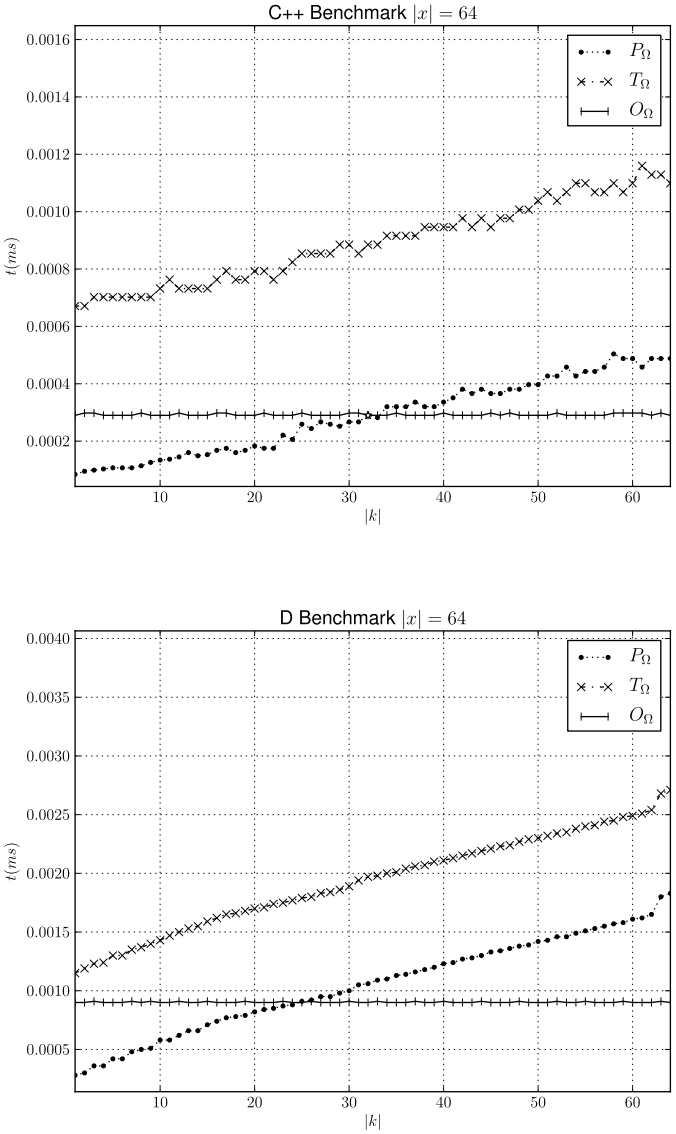
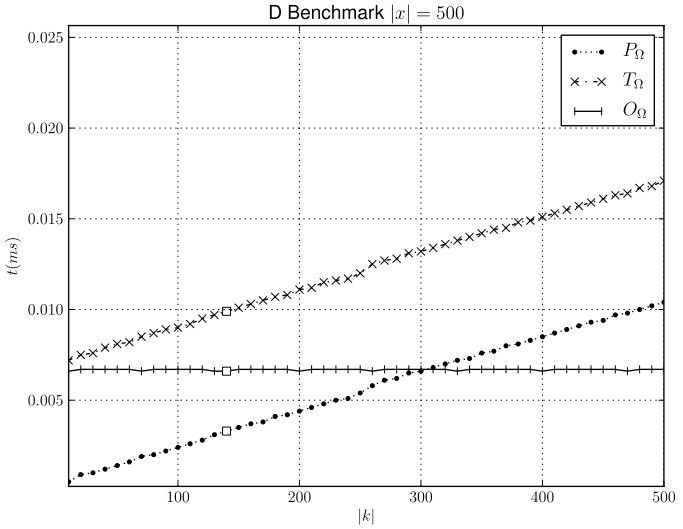


Figure 4: C++ vs D Results

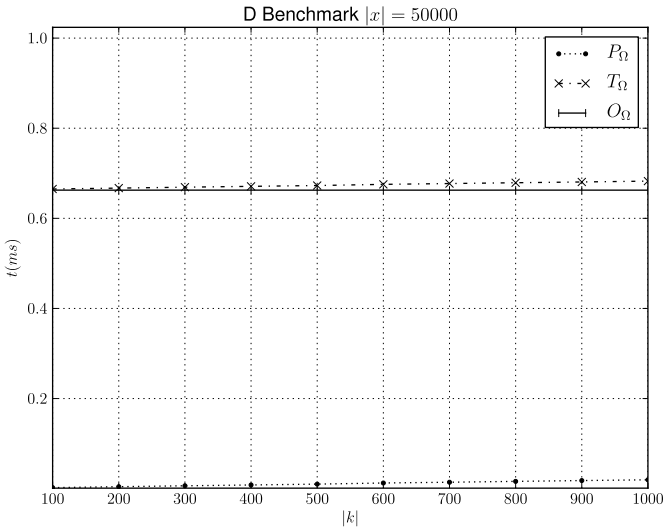
the heuristic match. Given that KMP^k_{opt} is arguably of practical interest at this point, we note that $|k| \approx \frac{|x|}{4}$. As no *smaller* ratios of $|k|$ to $|x|$ showed a favourable case for KMP^k_{opt} in our analysis, we note that the strong cases for the use of metaprogramming occurred where $|k| \geq \approx \frac{|x|}{4}$. Our analysis also noted that where $|k| \leq \approx \frac{|x|}{50}$, the difference between the KMP^k_{opt} and KMP_{trad} becomes so small as to be practically negligible (as seen in Figure 5b).

6 Concluding Remarks

The benchmarking of the optimised and traditional KMP algorithms, although based on a particular case analysis in which no keyword matches occur, has lead to interesting results. Firstly, it was seen that performance gains are most significant when the proportional difference in size between the search text and the keyword is small.



(a) Heuristic Match



(b) Minimal Gains

Figure 5: Cases For and Against Optimisation

This suggests that compile time optimisation may prove practical where this is the case, such as in intrusion detection systems.

Similarly, it was seen that performance gains become redundant when the proportional difference between search text and keyword size is very large. Therefore compile time optimisation appears to be less applicable in domains where such relations are typical, for example DNA pattern matching.

The clear design of the research objectives, implementation structure, and hypotheses are seen to form the basis of a good benchmark design. Such definitions also reinforce the repeatability and correctness of the experiment.

Furthermore, succinct information has been provided on the limitations of C++ regarding its suitability for optimised keyword pattern matching. C++ metaprogramming is shown to be fundamentally unsuited for even moderately demanding compile time string operations. In retrospect this is not very surprising. Template metaprogramming (let alone template *string* metaprogramming) is simply not a feature C++ was designed for. Indeed, any "feature" that is not explicitly designed from the ground up, remains—by definition—in the realm of craft.

This contrasts strongly with the solid engineering behind the D programming language. Though the D benchmarks are of limited use as far as scientific observation is concerned, they serve as a very good proof of concept. D is seen to provide highly robust metaprogramming support—exactly what is required for computationally demanding compile time optimisations. Its native support for compile time string operations as well as its Compile Time Function Execution feature, are only two of the reasons that made implementing the benchmark in D a suitable decision.

6.1 Further Research

This work presents a base for several areas of further research, including the following topics:

- The benchmark model described in Section 3 can be refined and extended to support the analysis of many of the algorithms identified in [13] and [2]. Such an effort would use the terms and definitions from those works in order to synthesise more consistently with the taxonomies they describe.
- The metaprogramming features offered by other languages such as LISP and Haskell can be investigated for possible applications in keyword pattern matching. Again, by following in the example of [13] and [2], an optimised toolkit could be constructed to supplement SPARE Parts and SPARE Time.
- Research investigating the performance gains of applying compile time optimisation to pattern matching in real world systems would prove interesting. In particular, it would be advisable to consider applications where the proportional difference between the search text size and keyword size is small.
- When the next C++ standard is published, it would be interesting to evaluate the language's overall suitability to metaprogramming. Such a review could make use of comparisons to other programming languages with an emphasis on qualitative software engineering "ilities" (e.g., maintainability, modifiability, scalability etc.).

Acknowledgements

We thank the anonymous referees for their feedback, and thank Linda Marshall and Derrick Kourie for their contributions to this research and its presentation.

References

1. D. Abrahams: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2005.
2. L. Cleophas: *Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms*, master's thesis, Eindhoven University of Technology, 2003.
3. L. Cleophas, B. W. Watson, and G. Zwaan: *A New Taxonomy of Sublinear Right-To-Left Scanning Keyword Pattern Matching Algorithms.* Sci. Comput. Program., 75 November 2010, pp. 1095–1112.
4. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, 2007.
5. *D Programming Language 2.0*: `http://www.digitalmars.com/d/2.0/index.html`.
6. D. E. Knuth, J. Morris, and V. R. Pratt: *Fast Pattern Matching in Strings.* SIAM Journal on Computing, 6(2) June 1977, pp. 323–350.
7. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
8. *Pattern Matching Pointers*: `http://www.cs.ucr.edu/~stelo/pattern.html`.
9. *schedtool(8) – Linux man page*: `http://linux.die.net/man/8/schedtool`.
10. W. Smyth: *Computing Patterns in Strings*, Addison-Wesley, 2003.
11. *taskset(1) – Linux man page*: `http://linux.die.net/man/1/taskset`.
12. *The Boost MPL Library*: version 1.43.0, `http://www.boost.org/doc/libs/1_43_0/libs/mpl/doc/index.html`.
13. B. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, 1995.
14. B. W. Watson and L. Cleophas: *SPARE Parts: A C++ toolkit for String PAttern REcognition.* Software—Practice & Experience, 34(7) June 2004, pp. 697–710.
15. B. W. Watson and G. Zwaan: *A Taxonomy of Keyword Pattern Matching Algorithms*, Computing Science Report 92/27, Technische Universiteit Eindhoven, 1992.

Efficient Eager XPath Filtering over XML Streams

Kazuhito Hagio, Takashi Ohgami, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
{kazuhito.hagio, takashi.oogami}@i.kyushu-u.ac.jp
{bannai, takeda}@inf.kyushu-u.ac.jp

Abstract. We address the embedding existence problem (often referred to as the filtering problem) over streaming XML data for Conjunctive XPath (CXP). Ramanan (2009) considered Downward CXP, a fragment of CXP that involves downward navigational axes only, and presented a streaming algorithm which solves the problem in $O(|P||D|)$ time using only $O(|P|height(D))$ bits of space, where $|P|$ and $|D|$ are the sizes of a query P and an XML data D, respectively, and $height(D)$ denotes the tree height of D. Unfortunately, the algorithm is *lazy* in the sense that it does not necessarily report the answer even after enough information has been gathered from the input XML stream. In this paper, we present an *eager* streaming algorithm that solves the problem with same time and space complexity. We also show the algorithm can be easily extended to Backward CXP a larger fragment of CXP.

1 Introduction

Efficient processing of XML streams is receiving much attention due to its growing range of applications such as stock and sports tickers, traffic information systems, electronic personalized newspapers, and entertainment delivery. Existing approaches assume that user interests are written as tree-shaped queries in *XPath*, a language for specifying the selection of element nodes within XML data trees. There are two variations of the problem: the embedding existence (EmbExist) and the query evaluation (QueryEval). The former is, given an XPath tree P and an XML data tree D, to determine whether there exists an embedding of P into D. The latter is, given P, D, and a node q_{out} of P, to determine the set of element nodes that q_{out} matches over all embeddings of P in D. A great deal of studies have been undertaken on the problems (see an excellent survey [1]). In this paper, we focus on EmbExist.

XPath supports a number of powerful modalities and it is rather expensive to process. In practice, many applications do not need the expressive power of the full language and use only a fragment of XPath. One such fragment is a conjunctive, navigational fragment named Conjunctive XPath (CXP). For non-streaming D, Gottlob et al. [2] and Ramanan [6] presented *in-memory* algorithms which solve QueryEval (and therefore EmbExist) for CXP in $O(|P||D|)$ time using $O(|D|)$ space. On the other hand, several studies have been undertaken on developing *streaming* algorithms for both the problems, with a restriction on navigational axes.

Downward CXP (DCXP) is a fragment of CXP where navigational axes are limited to the child and descendant axes. Ramanan [7] showed that for DCXP, there is a streaming algorithm which solves EmbExist in $O(|P||D|)$ time using only $O(|P|height(D))$ bits of space, where $height(D)$ denotes the tree height of D. Gou and Chirkova [3] also presented an algorithm which takes $O(|P||D|)$ time and $O(r(P,D)|P|\log height(D))$ bits of space, where $r(P,D)$ denotes the recursion depth

Kazuhito Hagio, Takashi Ohgami, Hideo Bannai, Masayuki Takeda: Efficient Eager XPath Filtering over XML Streams, pp. 30–44.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

of D w.r.t. Q[1]. Unfortunately, both the algorithms are *lazy* in the sense that they do not necessarily report the answer even after enough information has been gathered from input XML stream.

Main contribution. In this paper we present an *eager* streaming algorithm which solves EmbExist for DCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space. We then extend it to *Backward CXP* (BCXP), a larger fragment of CXP where some additional navigational axes are allowed.

The remainder of this paper is as follows. In Section 2 we define CXP and its fragments DCXP and BCXP, and then formulate our problem. In Section 3 we show a lazy algorithm which is essentially the same as the one presented by Ramanan in [7]. In Section 4 we describe how to modify the algorithm *eager*. In Section 5 we extend these two algorithms to BCXP. In Section 6 we mention related work and in Section 7 we conclude this paper.

2 Preliminaries

2.1 Notation

Let A be a finite alphabet. An element of A^* is called a *string*. A string y is said to be a *substring* of another string w if w can be written as $w = xyz$ for some strings x, z. For a string w, the i-th symbol of w is denoted by $w[i]$, and the substring of w that begins at position i and ends at position j is denoted by $w[i..j]$.

Let R, S be any binary relations on a set X. The *composition* of R and S is $R \circ S = \{\langle x, z\rangle \mid \langle x, y\rangle \in R \text{ and } \langle y, z\rangle \in S\}$. Let $R^0 = I_X = \{\langle x, x\rangle \mid x \in X\}$, and let $R^n = R \circ R^{n-1}$ for $n \geq 1$. Then, the *transitive closure* of R is $R^+ = \bigcup_{n=1}^{\infty} R^n$, and the *reflexive, transitive closure* of R is $R^* = \bigcup_{n=0}^{\infty} R^n$. The *inverse* of R is $R^{-1} = \{\langle x, y\rangle \mid \langle y, x\rangle \in R\}$. Let $R(y) = \{x \in X \mid \langle x, y\rangle \in R\}$.

2.2 XML data tree and XML data

Let Σ be a set of *tag names*. An *XML data tree* is an ordered tree with nodes v labeled by $label(v)$ in Σ, and is denoted by D. Let $\mathcal{N}_D$ denote the set of nodes in D. The cardinality of $\mathcal{N}_D$ is called the *size* of D and denoted by $|D|$. Let $<_{pre}$ denote the pre-order on $\mathcal{N}_D$.

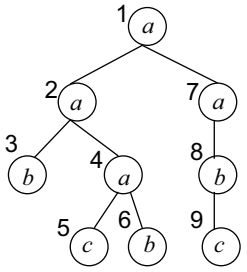
Let $\overline{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$. For any $u \in \mathcal{N}_D$, let

$$\mathcal{S}(u) = \begin{cases} a\ \bar{a}, & u \text{ is a leaf;} \\ a\ \mathcal{S}(v_1) \cdots \mathcal{S}(v_k)\ \bar{a}, & u \text{ is an internal node with children } v_1, \ldots, v_k. \end{cases}$$

where $a = label(u)$. We note that $\mathcal{S}(u)$ is a string over $\Sigma \cup \overline{\Sigma}$. The *serialized representation* $\mathcal{S}(D)$ of an XML data tree D is defined to be $\mathcal{S}(r)$ where r is the root of D. The serialized representations of XML data trees are called the *XML data*. In this paper, we assume that the input XML data tree is given in the form of XML data, and identify an XML data tree D and its serialized representation $\mathcal{S}(D)$ if no confusion occurs. Thus we simply denote by $D[i]$ the symbol $\mathcal{S}(D)[i]$, and by $D[i..j]$ the substring $\mathcal{S}(D)[i..j]$, respectively. We often use N as the length of $\mathcal{S}(D)$.

An example of XML data tree and the corresponding XML data are shown in Fig. 1.

[1] Since $r(P, D) = O(height(D))$ the space requirement can be $O(|P|height(D) \log height(D))$ which is worse than $O(|P|height(D))$. On the other hand, $r(P, D)$ is often smaller than $height(D)$ in some practical cases.



t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$D[t]$	a	a	b	$\bar{b}$	a	c	$\bar{c}$	b	$\bar{b}$	$\bar{a}$	$\bar{a}$	a	b	c	$\bar{c}$	$\bar{b}$	$\bar{a}$	$\bar{a}$

Figure 1. An example of XML data tree D is displayed on the left and its serialized representation $D[1..N]$ is shown on the right. We have $|D| = |\mathcal{N}_D| = 9$ and $N = 18$. The node numbered 4 of D corresponds to interval $[5, 10]$ of $D[1..N]$.

In XML data $D = D[1..N]$, every $v \in \mathcal{N}_D$ corresponds to an interval $[s(v), e(v)]$ with $1 \leq s(v) < e(v) \leq N$ such that v *starts at position* $s(v)$ and *ends at position* $e(v)$. We note that symbols $a \in \Sigma$ and $\bar{a} \in \overline{\Sigma}$, respectively, correspond to start and end tags of XML data.

Proposition 1. *For any* $u, v \in \mathcal{N}_D$, $u <_{pre} v \iff s(u) < s(v)$.

2.3 Conjunctive XPath, embedding, occurrence

We consider two binary relations on $\mathcal{N}_D$

$$\mathsf{child} = \{\langle u, v\rangle \mid u \text{ is a child of } v\},$$
$$\mathsf{nextSib} = \{\langle u, v\rangle \mid u \text{ is the next sibling of } v\}$$

and their inverses $\mathsf{parent} = \mathsf{child}^{-1}$ and $\mathsf{prevSib} = \mathsf{nextSib}^{-1}$. These four binary relations and their transitive and reflexive transitive closures are called *axes*. Additionally, the identity $\mathsf{self} = \{\langle v, v\rangle \mid v \in \mathcal{N}_D\}$, the abbreviation $\mathsf{following} = \mathsf{child}^* \circ \mathsf{nextSib}^+ \circ \mathsf{parent}^*$ and its inverse $\mathsf{preceding} = \mathsf{following}^{-1}$ are also axes[2].

A *conjunctive XPath (CXP) tree* is an unordered tree such that

- the nodes p are labeled by $\mathit{label}(p) \in \Sigma \cup \{\star\}$, where $\star$ is a special symbol not in Σ; and
- the edges are labeled by axes.

Let P be a CXP tree. The *size* of P, denoted by $|P|$, is the number of nodes. Let $P.rt$ denote the root of P. For any non-root node q of P, let $\chi(q)$ denote the label of the edge between q and its parent. For a node q of P, let $\mathit{sub}(q)$ denote the subtree of P rooted at q. An *embedding* of P into D is a function φ that maps nodes of P to nodes of D such that

- $\mathit{label}(q) \in \{\star, \mathit{label}(\varphi(q))\}$ for any node q of P; and
- $\langle \varphi(p), \varphi(q)\rangle \in \chi(q)$ for any non-root node q of P with parent p.

We note that function φ is not necessarily an injection, unlike the standard setting of tree pattern matching (see, e.g. [4]). Figure 2 illustrates embeddings of CXP tree into XML data tree.

[2] The descendant, descendant-or-self, ancestor, ancestor-or-self, preceding-sibling, and following-sibling axes of XPath1.0 (`http://www.w3.org/TR/xml`) correspond to child^+, child^*, parent^+, parent^*, $\mathsf{prevSib}^+$, and $\mathsf{nextSib}^*$, respectively. We note that the original definition of XPath1.0 excludes nextSib, $\mathsf{nextSib}^*$, prevSib, and $\mathsf{prevSib}^*$.

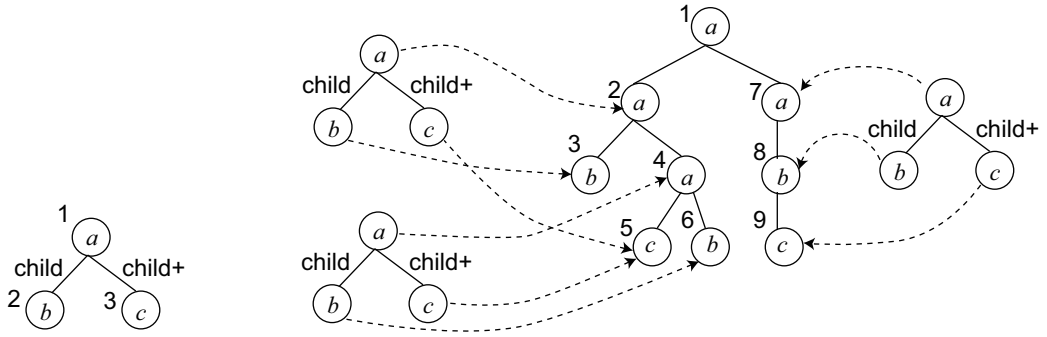


Figure 2. An example CXP tree is shown on the left, and its embeddings into the XML data tree of Fig. 1 are illustrated on the right.

A CXP tree P is said to *occur at* $v \in \mathcal{N}_D$ if there exists an embedding φ of P into D with $\varphi(P.rt) = v$. An *occurrence* of P in D is a node $v \in \mathcal{N}_D$ at which P occurs. Let $Occ(P, D)$ denote the set of occurrences of P in D.

A CXP tree P is said to be *unsatisfiable* if no node of D is an occurrence of P for any D, and *satisfiable*, otherwise. We assume that the input CXP tree is satisfiable throughout this paper.

2.4 Problem statement

*Problem 2 (*EmbExist*).* Given a CXP tree P and an XML data D, determine whether there exists an embedding of P into D.

*Problem 3 (*QueryEval*).* Given a CXP tree P, a node q_{out} of P, and an XML data D, compute $Eval(P, q_{out}, D) = \{\varphi(q_{out}) \mid \varphi \text{ is an embedding of } P \text{ into } D\}$.

EmbExist is often referred to as the filtering problem. The next is a slightly strengthened version of EmbExist.

*Problem 4 (*AllOcc*).* Given a CXP tree P and an XML data D, compute $Occ(P, D)$.

We note that AllOcc is essentially the same as EmbExist and is a special case of QueryEval where q_{out} is the root of P. In this paper we focus on AllOcc.

A *streaming algorithm* for AllOcc is an algorithm which scans an XML data $D = D[1..N]$ and emits, for every $x \in \mathcal{N}_D$, the pair $\langle x, b_x \rangle$ during one pass through $D[1..N]$, where b_x denotes a Boolean value indicating whether P occurs at x. A streaming algorithm for AllOcc is *eager* if it emits the pair $\langle x, b_x \rangle$ with minimum delay for every $x \in \mathcal{N}_D$.

2.5 Fragments of CXP

The *downward* axes are child, child$^+$, child*, and self. The *forward* (resp. *backward*) axes are the downward axes plus nextSib, nextSib$^+$, nextSib* and following (resp. prevSib, prevSib$^+$, prevSib* and preceding). The fragments of CXP with downward, forward and backward axes are denoted by DCXP, FCXP and BCXP, respectively. Figure 3 illustrates the fragments of CXP.

Theorem 5 ([7]). *There is a streaming algorithm that solves* AllOcc *for DCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

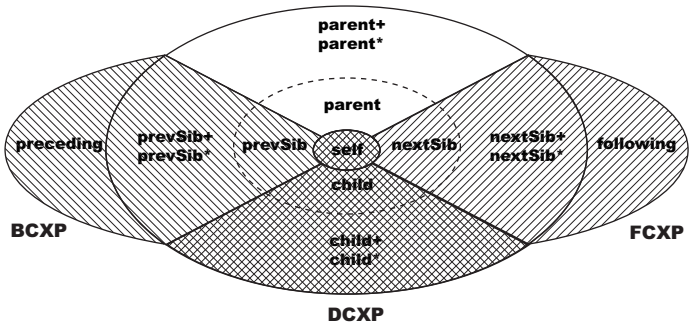


Figure 3. Fragments of CXP are illustrated.

3 Lazy Algorithm for DCXP

In this section, we describe a lazy algorithm for solving AllOcc for DCXP, which simplifies a predicate evaluator presented by Ramanan in [7]. Throughout this section D is any fixed XML data.

3.1 Introducing predicates M and T

Definition 6. *For any node p of a CXP tree P and any $u \in \mathcal{N}_D$, let*

$$M(p,u) = \mathsf{T} \Leftrightarrow sub(p) \text{ occurs at } u.$$

Since $Occ(P,D)$ is the set of nodes $v \in \mathcal{N}_D$ such that $M(P.rt,v) = \mathsf{T}$, we consider computing the values $M(P.rt,v)$ for all $v \in \mathcal{N}_D$ for any P. For this purpose, we use another predicate $T(\cdot,\cdot)$ defined below.

For any non-root node q of P, let $sub^+(q)$ be the tree obtained from the tree $sub(q)$ by adding a new root node r with label $\star$ an edge from r to q labeled $\chi(q)$. (See an example in Fig. 4.)

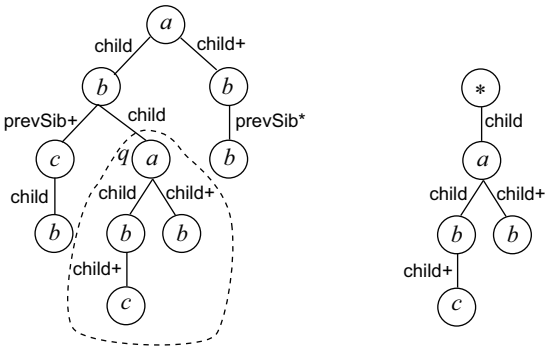


Figure 4. An example subtree $sub(q)$ of CXP tree is surrounded with broken line on the left, and the corresponding $sub^+(q)$ is displayed on the right.

Definition 7. *For any non-root node q of a CXP tree P and any $u \in \mathcal{N}_D$, let*

$$T(q,u) = \mathsf{T} \Leftrightarrow sub^+(q) \text{ occurs at } u.$$

Examples of M and T can be found in Fig. 5.

$M(q,v)$

q \ v	1	2	3	4	5	6	7	8	9
1	F	T	F	T	F	F	T	F	F
2	F	F	T	F	F	T	F	T	F
3	F	F	F	F	T	F	F	F	T

$T(q,v)$

q \ v	1	2	3	4	5	6	7	8	9
1									
2	F	T	F	T	F	F	T	F	F
3	T	T	F	T	F	F	T	T	F

Figure 5. The values of functions M and T for the XML tree D and the CXP tree P of Fig. 2.

Proposition 8. *For any non-root node q of a CXP tree P and any $u \in \mathcal{N}_D$,*

$$T(q,u) = \bigvee_{\langle v,u\rangle \in \chi(q)} M(q,v).$$

We can prove the following lemma:

Lemma 9. *For any node p of a CXP tree P and any $u \in \mathcal{N}_D$,*

$$M(p,u) = \big(label(p) \in \{\star, label(u)\}\big) \wedge \big(\bigwedge_{q \text{ is a child of } p} T(q,u)\big).$$

Proof. Directly from the definitions of M and T. □

3.2 Algorithm

Now we assume that P is a DCXP tree. We have the following lemma:

Lemma 10. *For any node q of a DCXP tree P and any $u \in \mathcal{N}_D$,*

$$T(q,u) = \begin{cases} \bigvee_{v \in \mathsf{child}(u)} M(q,v), & \text{if } \chi(q) = \mathsf{child}; \\ \bigvee_{v \in \mathsf{child}(u)} (T(q,v) \vee M(q,v)), & \text{if } \chi(q) = \mathsf{child}^+; \\ M(q,u) \vee \bigvee_{v \in \mathsf{child}(u)} T(q,v), & \text{if } \chi(q) = \mathsf{child}^*; \\ M(q,u), & \text{if } \chi(q) = \mathsf{self}. \end{cases}$$

Proof. By Proposition 8. □

Algorithm 1 follows directly from Lemmas 9 and 10. It essentially processes the nodes v of D in post-order. Arrays $M[\cdot,\cdot]$ and $T[\cdot,\cdot]$ are used to store the values of $M(\cdot,\cdot)$ and $T(\cdot,\cdot)$, respectively. We note that the values of $M(q,u)$ and $T(q,u)$ should be stored only for the ancestors u of current v, and therefore the space requirement for M and T is $O(|P|height(D))$ bits.

Theorem 11. *Algorithm 1 (lazily) solves* AllOcc *for DCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

4 Eager Algorithm for DCXP

In this section we modify Algorithm 1 to be eager.

Algorithm 1: A lazy streaming algorithm that solves AllOcc for DCXP.

```
1  class LazyDCXP
2      void run(CXPTree P; XMLData D[1..N])
3          initialize M[·,·] and T[·,·] to F;
4          for t := 1 to N do
5              if D[t] ∈ Σ then do nothing;
6              if D[t] ∈ Σ̄ then
7                  let v be the node of D with t = e(v);
8                  endTag(P, v);
9      void endTag(CXPTree P; XMLDataNode v)
10         foreach node q of P in post-order do updateM(q, v);
11     void updateM(CXPTreeNode q; XMLDataNode v)
12         M[q,v] := (label(q) ∈ {⋆, label(v)}) ∧ (⋀_{c is a child of q} T[c,v]); // by Lemma 9
13         if q is root node then
14             emit ⟨v, M[q,v]⟩;
15         else
16             updateT_V(q, v);
17     void updateT_V(CXPTreeNode q; XMLDataNode v) // by Lemma 10
18         if χ(q) = self  then T[q,v] := M[q,v];
19         if χ(q) = child* then T[q,v] := T[q,v] ∨ M[q,v];
20         if v has parent u then
21             if χ(q) = child  then T[q,u] := T[q,u] ∨ M[q,v];
22             if χ(q) = child+ then T[q,u] := T[q,u] ∨ M[q,v] ∨ T[q,v];
23             if χ(q) = child* then T[q,u] := T[q,u] ∨ T[q,v];
```

4.1 Precise definition of eagerness

First, we formally define what is meant by *eager*. To represent predicates M and T for varying D, we explicitly specify superscript D as M^D and T^D.

Definition 12. *Let P be any CXP tree and let $D = D[1..N]$ be an XML data. For any node p of P, any $u \in \mathcal{N}_D$, and any $t \in [s(u), N]$, let*

$$M_t^D(p,u) = \left\{ M^{D'}(p,u) \;\middle|\; D' \text{ is an XML data with } D'[1..t] = D[1..t] \right\}.$$

Intuitively, $M_t^D(p,u)$ is the set of possible values of $M^D(p,u)$ just after reading the t-th symbol of $D[1..N]$. It is thus a subset of $\{\mathsf{T},\mathsf{F}\}$, and can be either $\{\mathsf{T}\}$, $\{\mathsf{F}\}$, or $\{\mathsf{T},\mathsf{F}\}$. Let us denote the values $\{\mathsf{T}\},\{\mathsf{F}\},\{\mathsf{T},\mathsf{F}\}$ simply by $\mathsf{T},\mathsf{F},\mathsf{U}$, respectively. In what follows, we omit superscript D and simply write as $M_t(p,u)$ if no confusion occurs. Fig. 6 illustrates the values of M_t for the XML data tree D and the CXP tree P of Fig. 2 where $t = 1,\ldots,18$.

Definition 13. *For any node p of a CXP tree P and for any $u \in \mathcal{N}_D$, let $\mathsf{time}_M(p,u)$ be the smallest integer $t \in [s(u), N]$ such that $M_t(p,u) \neq \mathsf{U}$.*

Proposition 14. *$M_t(p,u) = \mathsf{U}$ for any $t \in [s(u), \mathsf{time}_M(p,u) - 1]$ and $M_t(p,u) = M(p,u) \neq \mathsf{U}$ for any $t \in [\mathsf{time}_M(p,u), N]$.*

We are now ready to define the concept of eagerness.

$M_t(q,v)$

		1 [1, 18]	2 [2, 11]	3 [3, 4]	4 [5, 10]	5 [6, 7]	6 [8, 9]	7 [12, 17]	8 [13, 16]	9 [14, 15]
$t=1$	1	U								
	2	**F**								
	3	**F**								
$t=2$	1	U	U							
	2	F	**F**							
	3	F	**F**							
$t=3$	1	U	U	**F**						
	2	F	F	**T**						
	3	F	F	**F**						
$t=4$	1	U	U	F						
	2	F	F	T						
	3	F	F	F						
$t=5$	1	U	U	F	U					
	2	F	F	T	**F**					
	3	F	F	F	**F**					
$t=6$	1	U	**T**	F	U	**F**				
	2	F	F	T	F	**F**				
	3	F	F	F	F	**T**				
$t=7$	1	U	T	F	U	F				
	2	F	F	T	F	F				
	3	F	F	F	F	T				
$t=8$	1	U	T	F	**T**	F	**F**			
	2	F	F	T	F	F	**T**			
	3	F	F	F	F	T	**F**			
$t=9,10,11$	1	U	T	F	T	F	F			
	2	F	F	T	F	F	T			
	3	F	F	F	F	T	F			
$t=12$	1	U	T	F	T	F	F	U		
	2	F	F	T	F	F	T	**F**		
	3	F	F	F	F	T	F	**F**		
$t=13$	1	U	T	F	T	F	F	U	**F**	
	2	F	F	T	F	F	T	F	**T**	
	3	F	F	F	F	T	F	F	**F**	
$t=14$	1	U	T	F	T	F	F	**T**	F	**F**
	2	F	F	T	F	F	T	F	T	**F**
	3	F	F	F	F	T	F	F	F	**T**
$t=15$	1	U	T	F	T	F	F	T	F	F
	2	F	F	T	F	F	T	F	T	F
	3	F	F	F	F	T	F	F	F	T
$t=16,17$	1	U	T	F	T	F	F	T	F	F
	2	F	F	T	F	F	T	F	T	F
	3	F	F	F	F	T	F	F	F	T
$t=18$	1	**F**	T	F	T	F	F	T	F	F
	2	F	F	T	F	F	T	F	T	F
	3	F	F	F	F	T	F	F	F	T

$T_t(q,v)$

		1 [1, 18]	2 [2, 11]	3 [3, 4]	4 [5, 10]	5 [6, 7]	6 [8, 9]	7 [12, 17]	8 [13, 16]	9 [14, 15]
$t=1$	1									
	2	U								
	3	U								
$t=2$	1									
	2	U	U							
	3	U	U							
$t=3$	1									
	2	U	**T**	U						
	3	U	U	U						
$t=4$	1									
	2	U	T	**F**						
	3	U	U	**F**						
$t=5$	1									
	2	U	T	F	U					
	3	U	U	F	U					
$t=6$	1									
	2	U	T	F	U	U				
	3	**T**	**T**	F	**T**	U				
$t=7$	1									
	2	U	T	F	U	**F**				
	3	T	T	F	T	**F**				
$t=8$	1									
	2	U	T	F	**T**	F	U			
	3	T	T	F	T	F	U			
$t=9,10,11$	1									
	2	U	T	F	T	F	**F**			
	3	T	T	F	T	F	**F**			
$t=12$	1									
	2	U	T	F	T	F	F	U		
	3	T	T	F	T	F	F	U		
$t=13$	1									
	2	U	T	F	T	F	F	**T**	U	
	3	T	T	F	T	F	F	U	U	
$t=14$	1									
	2	U	T	F	T	F	F	T	U	U
	3	T	T	F	T	F	F	**T**	**T**	U
$t=15$	1									
	2	U	T	F	T	F	F	T	U	**F**
	3	T	T	F	T	F	F	T	T	**F**
$t=16,17$	1									
	2	U	T	F	T	F	F	T	**F**	F
	3	T	T	F	T	F	F	T	T	F
$t=18$	1									
	2	**F**	T	F	T	F	F	T	F	F
	3	T	T	F	T	F	F	T	T	F

Figure 6. The values of functions M_t and T_t for the XML data tree D and the CXP tree P of Fig. 2, where $t = 1, \ldots, 18$. Value changes from the previous t are emphasized in boldface.

Definition 15. *A streaming algorithm that solves* AllOcc *is* eager *if, for every* $u \in \mathcal{N}_D$ *it emits* $\langle u, M(P.rt, u)\rangle$ *just after processing* $D[t^*]$ *where* $t^* = \mathsf{time}_M(P.rt, u)$.

For time_M, we can prove the following:

Proposition 16. *If* P *is a BCXP tree, then* $\mathsf{time}_M(p, u) \in [s(u), e(u)]$ *for any node* p *of* P *and for any* $u \in \mathcal{N}_D$.

Proof. Let φ be any embedding of $sub(p)$ into D with $\varphi(p) = u$, if exists. Since the axes of P are limited to backward ones, for any node q of $sub(p)$, $\varphi(q) \in \mathsf{preceding}(u) \cup \mathsf{child}^*(u)$ and therefore $e(\varphi(q)) \leq e(u)$. □

4.2 Introducing T_t

We extend the Boolean operations $\wedge, \vee, \neg$ to domain $\{\mathsf{T}, \mathsf{F}, \mathsf{U}\}$ by: $\mathsf{T} \wedge \mathsf{U} = \mathsf{U} \wedge \mathsf{T} = \mathsf{U}$, $\mathsf{T} \vee \mathsf{U} = \mathsf{U} \vee \mathsf{T} = \mathsf{T}$, $\mathsf{F} \wedge \mathsf{U} = \mathsf{U} \wedge \mathsf{F} = \mathsf{F}$, $\mathsf{F} \vee \mathsf{U} = \mathsf{U} \vee \mathsf{F} = \mathsf{U}$, and $\mathsf{U} \wedge \mathsf{U} = \mathsf{U} \vee \mathsf{U} = \neg\mathsf{U} = \mathsf{U}$. For convenience, let $M_t(p, u) = \mathsf{U}$ for any $t \in [0, \ldots, s(u) - 1]$, although $M_t(p, u)$ is undefined for such t.

Definition 17. *For any non-root node* q *of a CXP tree* P*, any* $u \in \mathcal{N}_D$*, and for any* $t \in [s(u), N]$*, let*

$$T_t(q, u) = \bigvee_{\langle v,u\rangle \in \chi(q)} M_t(q, v).$$

Then we have:

Lemma 18. *For any node* p *of a CXP tree* P*, for any* $u \in \mathcal{N}_D$*, and for any* $t \in [0, N]$*,*

$$M_t(p, u) = \big(label(p) \in \{\star, label(u)\}\big) \wedge \Big(\bigwedge_{q \text{ is a child of } p} T_t(q, u)\Big).$$

Proof. By Lemma 9 and the definitions of M_t and T_t. □

Define $\mathsf{time}_T(p, u)$ in a way similar to $\mathsf{time}_M(p, u)$. Then:

Proposition 19. *If* P *is a BCXP tree, then the following statements hold for any node* p *of* P *and for any* $u \in \mathcal{N}_D$.

- $\mathsf{time}_T(p, u) \in [s(u), e(u)]$.
- *If* $\chi(p) \in \{\mathsf{prevSib}, \mathsf{prevSib}^+, \mathsf{preceding}\}$ *then* $\mathsf{time}_T(p, u) = s(u)$.
- *If* $\chi(p) \in \{\mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{self}, \mathsf{prevSib}^*\}$ *and* $T(p, u) = \mathsf{F}$ *then* $\mathsf{time}_T(p, u) = e(u)$ *(due to the assumption that* P *is satisfiable).*

Proof. Let φ be any embedding of $sub^+(p)$ into D with $\varphi(p') = u$ where p' is the parent of p, if exists. Since the axes of P are limited to backward ones, for any node q of $sub^+(p)$, $\varphi(q) \in \mathsf{preceding}(u) \cup \mathsf{child}^*(u)$ and therefore $e(\varphi(q)) \leq e(u)$. Thus we have $\mathsf{time}_T(p, u) \in [s(u), e(u)]$. Suppose $\chi(p) \in \{\mathsf{prevSib}, \mathsf{prevSib}^+, \mathsf{preceding}\}$. Then, $e(\varphi(q)) \leq e(\varphi(p)) < s(\varphi(p')) = s(u)$ and we have $\mathsf{time}_T(p, u) = s(u)$. Suppose $\chi(p) \in \{\mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{self}, \mathsf{prevSib}^*\}$ and $T(p, u) = \mathsf{F}$. There is a possibility that a descendant v of u (possibly $u = v$) appears that makes $T(p, u)$ T until reading the end-tag of u. Thus we have $\mathsf{time}_T(p, u) = e(u)$. □

4.3 Algorithm

Again, we restrict ourselves to the DCXP trees. We have:

Lemma 20. *For any node q of a DCXP tree P, for any $u \in \mathcal{N}_D$, and for any $t \in [s(u), e(u)]$,*

$$T_t(q,u) = T_{t-1}(q,u) \vee \bigvee_{\langle v,u\rangle \in \chi(q) \text{ and } t \in [s(v),e(v)]} ((M_{t-1}(q,v) = \mathsf{U}) \wedge (M_t(q,v) = \mathsf{T})).$$

Proof. Let $\langle v,u\rangle \in \chi(q)$. Since $\chi(q) \in \{\mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{self}\}$, we have $[s(v),e(v)] \subseteq [s(u),e(v)]$. The lemma follows from Definition 17. □

Our eager algorithm follows from Lemmas 18 and 20. It can be summarized as Algorithm 2. It initializes the entries of arrays M and T by U and then incrementally rewrites them to T or F so that $M[q,u]$ and $T[q,u]$ are, respectively, identical to $M_t(q,u)$ and $T_t(q,u)$ for every $t \in [s(u),N]$. When a node v is found such that $M[q,v]$ just changes from U to T for some q with $\chi(q) = \mathsf{child}$ (resp. child^+, child^*, and self), it rewrites $T[q,u]$ for the parent u of v (resp. a proper ancestor u of v, an ancestor u of v, and $u = v$ itself). Whenever $T[q,u]$ changes, we evaluate $M[p,u]$ for parent p of q, and if $M[p,u]$ changes into T then we repeat this process. When a node v' is found such that $M[P.rt,u] \neq \mathsf{U}$, we output the pair $\langle u, M[P.rt,u]\rangle$.

Let us call the *t-th operating cycle* the t-th iteration of the **for**-loop in function *run*() of Algorithm 2.

Lemma 21. *For any DCXP tree P, after the t-th operating cycle of Algorithm 2, the value of $M[p,u]$ is identical to $M_t(p,u)$ for any node p of P and for any ancestor u of v, where v is the node of D such that $t = s(v)$ or $t = e(v)$.*

Proof. When p is a leaf, Lemma 18 implies that $M_t(p,u) = M_{s(u)}(p,u) = (label(p) \in \{\star, label(u)\})$ for any $t \in [s(u),N]$. At $t = s(u)$, the algorithm sets $M[p,u]$ to $(label(p) \in \{\star, label(u)\})$ in execution of $updateM(p,u)$ and then never changes it. Thus $M[p,u]$ holds $M_t(p,u)$ for $t \in [s(u),N]$ for leaves p of P. For internal nodes p, $M_t(p,u)$ depends on the values $T_t(q,u)$ for the children q of p. The algorithm invokes $updateM(p,u)$ whenever $T[q,u]$ changes from U into T, and each execution of $updateM(p,u)$ updates the value $M[p,u]$ according to Lemma 18. Thus $M[p,u]$ correctly holds $M_t(p,u)$ if $T[q,u]$ correctly holds $T_t(q,u)$ for every child q of p.

In execution of $updateM$, the algorithm invokes *liftUp* and updates $T[q,u]$ from $T_{t-1}(q,u)$ to $T_t(q,u)$ according to Lemma 20, whenever $M[q,v]$ changes from U into T for a child v of u (resp. a proper descandant v of u, a descendant v of u, and $u = v$ itself), if $\chi(p) = \mathsf{child}$ (resp. child^+, child^*, and self). Hence $T[q,u]$ correctly holds $T_t(q,u)$. □

Lemma 22. *Algorithm 2 runs in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Proof. The space complexity is $O(|P|height(D))$ bits as for Algorithm 1. To estimate the time complexity, we have only to consider the total cost of executing *liftUp*. We note that *liftUp* is invoked only when the value $M[q,v]$ is changed from U into T and that the value $T[q,v]$ is changed from U into T in each execution of the **while**-loop in *liftUp*. Thus the total time is $O(|P||D|)$. □

Theorem 23. *Algorithm 2 eagerly solves* AllOcc *for DCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Proof. By Lemmas 21 and 22. □

Algorithm 2: An eager streaming algorithm that solves AllOcc for DCXP.

```
1  class EagerDCXP
2      void run(CXPTree P; XMLData D[1..N])
3          initialize M[·,·] and T[·,·] to U;
4          for t := 1 to N do
5              if D[t] ∈ Σ then
6                  let v be the node of D with t = s(v);
7                  startTag(P, v);
8              if D[t] ∈ Σ̄ then
9                  let v be the node of D with t = e(v);
10                 endTag(P, v);
11     void startTag(CXPTree P; XMLDataNode v)
12         foreach node q of P in post-order do updateM(q, v);
13     void endTag(CXPTree P; XMLDataNode v)
14         foreach node q of P in post-order do
15             if M[q, v] = U then updateM(q, v);
16             if T[q, v] = U then T[q, v] := F;
17     void updateM(CXPTreeNode q; XMLDataNode v)
18         M[q, v] := (label(q) ∈ {⋆, label(v)}) ∧ (⋀_{c is a child of q} T[c, v]);
19         if q is root node then
20             if M[q, v] ≠ U then emit the pair ⟨v, M[q, v]⟩;
21         else
22             if M[q, v] = T then updateT_V(q, v);
23     void updateT_V(CXPTreeNode q; XMLDataNode v)
24         let u be the parent of v;
25         if χ(q) = child   then liftUp(q, u, T);
26         if χ(q) = child+  then liftUp(q, u, F);
27         if χ(q) = child*  then liftUp(q, v, F);
28         if χ(q) = self    then liftUp(q, v, T);
29     void liftUp(CXPTreeNode q; XMLDataNode u; bool once)
30         let p be the parent of q;
31         while u ≠ nil and T[q, u] = U do
32             T[q, u] := T; updateM(p, u);
33             if once then return;
34             u := the parent of u;
```

5 Extension to BCXP

5.1 Lazy algorithm for BCXP

The statement of Lemma 10 is extended to BCXP trees by adding four cases:

Lemma 24. *For any node q of a BCXP tree P and any $v \in \mathcal{N}_D$,*

$$T(q,v) = \begin{cases} M(q,w), & \text{if } \chi(q) = \mathsf{prevSib}; \\ T(q,w) \vee M(q,w), & \text{if } \chi(q) = \mathsf{prevSib}^+; \\ T(q,w) \vee M(q,v), & \text{if } \chi(q) = \mathsf{prevSib}^*; \\ T(q,z) \vee ((z \notin \mathsf{parent}(v)) \wedge M(q,z)), & \text{if } \chi(q) = \mathsf{preceding}, \end{cases}$$

where w is the previous sibling of v and z is the previous node of v w.r.t. $<_{pre}$. (Let $M(q,w) = T(q,w) = \mathsf{F}$ when w does not exist and let $M(q,z) = T(q,z) = \mathsf{F}$ when z does not exist.)

Proof. It is rather straightforward in the cases of $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, and $\mathsf{prevSib}^*$. We consider the case of $\chi(q) = \mathsf{preceding}$. Let z be the previous node of u w.r.t. $<_{pre}$. There are two cases.
Case 1: When v is not a leftmost sibling. Let w be the immediately left sibling of v. Then z is the rightmost descendant of w. In this case we have $\mathsf{preceding}(v) = \mathsf{preceding}(z) \cup \{z\}$.
Case 2: When v is a leftmost sibling. Then z is the parent of v. In this case we have $\mathsf{preceding}(v) = \mathsf{preceding}(z)$. □

Based on Lemmas 9 and 24, Algorithm 1 is extended as Algorithm 3 to cope with BCXP trees. What needs to be stored are (1) the values of $M(q,u)$ and $T(q,u)$ for the ancestors u of v; (2) the values of $M(q,w)$ and $T(q,w)$ for the previous sibling w of v; and (3) the values of $M(q,z)$ and $T(q,z)$ for the previous node z of v w.r.t. $<_{pre}$. The space requirement for M and T is still $O(|P|height(D))$ bits.

Theorem 25. *Algorithm 3 (lazily) solves* AllOcc *for BCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Algorithm 3: A lazy streaming algorithm that solves AllOcc for BCXP.

```
1  LazyBCXP extends LazyDCXP
       // methods run(), endTag(), updateT_V() inherit from LazyDCXP
       // method updateM() overrides the one in LazyDCXP
       // method updateT_H() is a newly added method
2      void updateM(CXPTreeNode q; XMLDataNode v)
3          M[q,v] := (label(q) ∈ {⋆, label(v)}) ∧ (⋀_{c is a child of q} T[c,v]);
4          if q is root node then
5              emit ⟨v, M[q,v]⟩;
6          else
7              updateT_V(q,v);
8              updateT_H(q,v); // inserted
9      void updateT_H(CXPTreeNode q; XMLDataNode v) // by Lemma 24
10         if v has previous sibling w then
11             if χ(q) = prevSib    then T[q,v] := M[q,w];
12             if χ(q) = prevSib+   then T[q,v] := T[q,w] ∨ M[q,w];
13             if χ(q) = prevSib*   then T[q,v] := T[q,w];
14         if χ(q) = prevSib*   then T[q,v] := T[q,v] ∨ M[q,v];
15         if v has previous node z w.r.t. <_pre then
16             if χ(q) = preceding then
17                 if v has previous sibling then  T[q,v] := T[q,z] ∨ M[q,z];
18                 else  T[q,v] := T[q,z];
```

Algorithm 4: An eager streaming algorithm that solves AllOcc for BCXP.

```
1  EagerBCXP extends EagerDCXP
       // methods run(), endTag(), updateM(), liftUp() inherit from EagerDCXP
       // methods startTag(), updateT_V() override the ones in EagerDCXP
2      void startTag(CXPTree P; XMLDataNode v)
3          foreach node q of P in post-order do
4              updateM(q, v);
5              LazyBCXP::updateT_H(q, v); // added
6      void updateT_V(CXPTreeNode q; XMLDataNode v)
7          let u be the parent of v;
8          if χ(q) = child      then liftUp(q, u, T);
9          if χ(q) = child+     then liftUp(q, u, F);
10         if χ(q) = child*     then liftUp(q, v, F);
11         if χ(q) = self       then liftUp(q, v, T);
12         if χ(q) = prevSib*   then liftUp(q, v, T); // added
```

5.2 Eager algorithm for BCXP

Lemma 26. *For any non-root node q of a BCXP tree P with $\chi(q) \in \{\mathsf{prevSib}, \mathsf{prevSib}^+, \mathsf{prevSib}^*, \mathsf{preceding}\}$, for any $v \in \mathcal{N}_D$, and for any $t \in [s(v), N]$,*

$$T_t(q,v) = \begin{cases} M_{s(v)-1}(q,w), & \text{if } \chi(q) = \mathsf{prevSib}; \\ T_{s(v)-1}(q,w) \vee M_{s(v)-1}(q,w), & \text{if } \chi(q) = \mathsf{prevSib}^+; \\ T_{s(v)-1}(q,w) \vee M_t(q,v), & \text{if } \chi(q) = \mathsf{prevSib}^*; \\ T_{s(v)-1}(q,z) \vee ((z \notin \mathsf{parent}(v)) \wedge M_{s(v)-1}(q,z)), & \text{if } \chi(q) = \mathsf{preceding}, \end{cases}$$

where w is the previous sibling of v and z is the previous node of v w.r.t. $<_{pre}$.

Proof. Recall Lemma 24. By Propositions 16 and 19, we have $T(q,v) = T_{e(v)}(q,v) \neq \mathsf{U}$ and $M(q,v) = M_{e(v)}(q,v) \neq \mathsf{U}$. Since $e(w) \leq s(v) - 1$, we also have $M(q,w) = M_{e(w)}(q,w) = M_{s(v)-1}(q,w) \neq \mathsf{U}$ and $T(q,w) = T_{e(w)}(q,w) = T_{s(v)-1}(q,w) \neq \mathsf{U}$. Thus the lemma holds for the cases of $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, and $\mathsf{prevSib}^*$. Since $e(z) \leq s(v) - 1$, we have $M(q,z) = M_{e(z)}(q,z) = M_{s(v)-1}(q,z)$ and $T(q,z) = T_{e(z)}(q,z) = T_{s(v)-1}(q,z)$. Thus the lemma holds for the case of $\chi(q) = \mathsf{preceding}$. □

Our eager algorithm for BCXP is obtained as an extension of Algorithm 2 and is summarized as Algorithm 4. Lemma 26 tells us that for $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, or $\mathsf{preceding}$, the values $T_t(q,v)$ are determined when the start-tag of v is read, namely, at $t = s(v)$. Line 5 is thus added to *startTag*(). On the other hand, the values $T_t(q,v)$ can change until reading the end-tag of v for $\chi(q) = \mathsf{prevSib}^*$, and therefore Line 12 is added to *updateT*$_V$().

The statement of Lemma 21 also holds for Algorithm 4:

Lemma 27. *For any BCXP tree P, after the t-th operating cycle of Algorithm 4, the value of $M[p,u]$ is identical to $M_t(p,u)$ for any node p of P and for any ancestor u of v, where v is the node of D such that $t = s(v)$ or $t = e(v)$.*

Proof. Comparing to the proof of Lemma 21, we have only to prove that $T[q,v]$ correctly holds $T_t(q,v)$ for any node q of P such that $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, $\mathsf{prevSib}^*$, or $\mathsf{preceding}$, assuming that $M[q,v]$ correctly holds $M_t(q,v)$.

In the three cases except $\chi(q) = \mathsf{prevSib}^*$, the values $T_t(q, v)$ are determined to T or F at $t = s(u)$, and the algorithm sets $T[q, v]$ to $T_t(q, v)$ by calling $update_H T(q, v)$ of LazyBCXP. In the case of $\chi(q) = \mathsf{prevSib}^*$, the values $T_t(q, v)$ can be U even for $t > s(u)$. Thus, it invokes $liftUp(q, v)$ to update $T[q, v]$ whenever $M[q, v]$ changes into T in execution of $updateM$. □

Lemma 28. *Algorithm 4 runs in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Proof. It requires only $O(|P|height(D))$ bits of space as Algorithm 3 does. To show its $O(|P||D|)$ time complexity, we have only to consider the total cost of executing $liftUp$. By the same discussion in the proof of Lemma 22, the total time is $O(|P||D|)$. □

Theorem 29. *Algorithm 4 eagerly solves* AllOcc *for BCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Proof. By Lemmas 27 and 28.

6 Related Work

By '*streaming algorithms*' we mean algorithms that perform the task in a single pass through the XML document, while keeping only small critical portions of the data in main memory for later use. Allowing $O(|D|)$ space enables us to store the whole streaming data in a buffer, to which any in-memory algorithm could be applied. Hence it is natural to allow only $o(|D|)$ space in the data complexity.

However, it is known that solving QueryEval over XML streams requires storing candidates for the answer nodes which take $\Omega(|D|)$ space in the worst case. For this reason, the space requirement is usually measured in terms of $maxcands(P, D)$, defined to be the maximum number of nodes of D that can be candidates for output, at any one instant.

Olteanu [5] presented an algorithm that uses $O(height(D)^2|P| + height(D) \cdot n \cdot maxcands(P, D))$ space and $O(height(D)|P||D|)$ time, where n is the number of location steps in P (i.e., the number of ancestors of q_{out}). Gou and Chirkova [3] presented an algorithm that uses $O(r(P, D)|P| + maxcands(P, D))$ space and $O(|P||D|)$ time, they claim. However, Ramanan [8] recently showed an $\Omega(n \cdot maxcands(P, D))$ lower bound for QueryEval for worst case P. This means that there is no algorithm for QueryEval that uses $O(f(height(D), |P|) + maxcands(P, D))$ space, for any function f, and therefore the claimed space upper bound of [3] is not achievable. On the other hand, Ramanan [7] presented an eager algorithm for QueryEval that runs in $O((|P| + height(D) \cdot n)|D|)$ time using $O(height(D)|P| + n \cdot maxcands(P, D))$ space. This space requirement matches the lower bound by Ramanan [8].

7 Conclusion

In this paper we addressed AllOcc. Efficiently solving AllOcc is of importance since it is useful not only in XML stream filtering but also in evaluating predicates in solving QueryEval. In such applications *eagerness* is a desirable feature. The previous AllOcc algorithm is due to Ramanan [7], which was presented as a predicate evaluator in his QueryEval algorithm. It takes only $O(|P|height(D))$ bits of space and $O(|P||D|)$ time but one drawback is its laziness as pointed out in [7]. We simplified the algorithm and then successfully modified it to be *eager*, without increasing time and space complexities.

References

1. M. Benedikt and C. Koch: *XPath leashed.* ACM Comput. Surv., 41(1) 2008.
2. G. Gottlob, C. Koch, and R. Pichler: *Efficient algorithms for processing XPath queries.* ACM TODS, 30(2) June 2005, pp. 444–491.
3. G. Gou and R. Chirkova: *Efficient algorithms for evaluating XPath over streams*, in SIGMOD'07, 2007, pp. 269–280.
4. P. Kilpeläinen and H. Mannila: *Ordered and unordered tree inclusion.* SIAM J. Comput., 24(2) 1995, pp. 340–356.
5. D. Olteanu: *SPEX: Streamed and progressive evaluation of XPath.* IEEE Transactions on Knowledge and Data Engineering, 19(7) 2007, pp. 934–949.
6. P. Ramanan: *Covering indexes for XML queries: Bisimulation − simulation = negation*, in VLDB'03, 2003, pp. 165–176.
7. P. Ramanan: *Worst-case optimal algorithm for XPath evaluation over XML streams.* J. Comput. Syst. Sci., 75(8) 2009, pp. 465–485.
8. P. Ramanan: *Memory lower bounds for XPath evaluation over XML streams.* J. Comput. Syst. Sci., 2010, in press.

Inexact Graph Matching by "Geodesic Hashing" for the Alignment of Pseudoknoted RNA Secondary Structures

Mira Abraham and Haim J. Wolfson

Blavatnik School of Computer Science
Raymond and Beverly Sackler Faculty of Exact Sciences
Tel Aviv University, Tel Aviv 69978, Israel
mira@post.tau.ac.il
wolfson@tau.ac.il (corresponding author)

Abstract. Non-coding RNAs are transcripts that do not encode proteins play key roles in many biological processes. The alignment of their secondary structures has become a major tool in RNA functional annotation. Many of the non-coding RNAs contain pseudoknots as a structural motif, which proved to be functionally important. We present HARP, a heuristic algorithm for the pairwise alignment of non-restricted (arbitrary) classes of pseudoknotted RNA secondary structures. HARP applies "geodesic hashing" to perform inexact matching of specially defined reduced RNA secondary structure graphs. The method proved to be efficient both in time and memory and was successfully tested on a benchmark of available experimental structures with known function. A web server is available at http://bioinfo3d.cs.tau.ac.il/HARP/.

Keywords: non-coding RNA, RNA structure alignment, secondary structure with pseudoknots, geometric hashing, geodesic distances, inexact graph matching

1 Introduction

Ribonucleic acid (RNA) molecules were once considered as mere carriers of the genetic information. Today it is known that many RNA transcripts or RNA segments within a transcript do not undergo translation. These sequences are often key players in numerous cellular processes such as chromosome replication, telomere maintenance, translation regulation and RNA modification. The increased interest in RNA function, and the assumption that the structure of a molecule reflects its function, motivates the development of efficient RNA structural alignment tools.

As in proteins, RNA structure is described at three levels. The **primary** structure is the RNA sequence drawn from an alphabet of 4 letters A,C,G,U. The **secondary** structure is a planar graph that consists of base-paired regions among single stranded regions. The **tertiary** structure consists of the 3D coordinates of the RNA molecule atom centers. Although there is statistical indication that tertiary structure similarity is more indicative of function similarity than secondary structure similarity [1], RNA structure usually means RNA secondary structure. This is due to the abundance of available modelled RNA secondary structures and the scarceness of RNA tertiary structures. The secondary structure is fully defined by the set of interacting base pairs $\{(i,j) : 1 \leq i < j \leq n\}$, where n is the length of the RNA molecule. Non-interacting consecutive nucleotides are called *loops*.

An RNA *helix (stem region)* is defined as a series of consecutive (in opposite directions) interacting base pairs $\{(i,j),(i+1,j-1)\ldots\}$. RNA helices, H_1 and H_2, are considered *crossing*, *non-nested* or *pseudoknotted* if $\exists(i_1,j_1) \in H_1, \exists(i_2,j_2) \in H_2$,

Mira Abraham, Haim J. Wolfson: Inexact Graph Matching by "Geodesic Hashing" for the Alignment of Pseudoknoted RNA Secondary Structures, pp. 45–58.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

so that $i_1 < i_2 < j_1 < j_2$ otherwise the helices are considered as *non-crossing* or *nested* (Fig. 1). The term *pseudoknot* refers to a set of crossing helices.

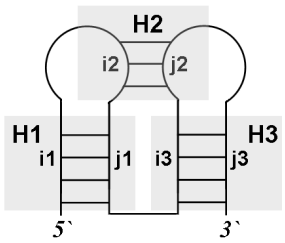


Figure 1. Pseudoknot example: H1 and H2 are crossing helices. H1 and H3 are non-crossing helices.

Many of the RNAs are known to have pseudoknots. Pseudoknots have been proved to be essential for the enzymatic activity of their RNAs, such as HDV ribozyme or self-splicing group I introns [31]. In other RNAs, such as 16S ribosomal RNA [27] and telomerase RNA [33], pseudoknots proved to be important for structural stability. Pseudoknots may also serve as drug targets as they are essential for induced frame-shifting in many viral RNAs [31].

Since an RNA secondary structure is represented by a graph, where vertices correspond to nucleotides and edges connect chemically interacting nucleotides/vertices, the pairwise RNA secondary structure alignment task is equivalent to detection of subgraph isomorphism. Moreover, due to possible insertions/deletions, which do not alter functionality, the algorithm should be tolerant enough to detect also "almost" isomorphic substructures. In the easier case, when pseudoknots are disregarded, the RNA graphs have been represented as rooted ordered trees, which led to polynomial algorithms for the detection of minimal edit distance between these trees [37,3,15,16,25,29]. Another approach for RNA secondary structure alignment that also disregards pseudoknots models the RNAs as "Multiple Graph Layers" (MiGaL) [4]. The case in which only one of the aligned structures is allowed to have pseudoknots [23] has been shown to be polynomial [17] as well. RNA structure alignment in the general case, where both structures can include pseudoknots, was shown to be NP-hard [38].

In order to circumvent the complexity challenge of the general alignment of RNA secondary structures, which include arbitrary pseudoknots, one can resort to the following strategies: (i) design algorithms that guarantee optimal solution for restricted classes of pseudoknots (limited problems). (ii) design heuristic algorithms that deal with the general problem (non-restricted classes of pseudoknots), however do not guarantee an optimal solution. Providing an optimal solution even for a limited problem is still complex, therefore, the currently available approaches suffer from both high time and high space complexity [26,12] which naturally creates limitations on both the complexity of the problem and the size of the input in addition to problem restrictions. In contrast to restricted problems algorithms, LARA [5], is a state-of-the-art heuristic method based on integer linear programming (ILP). LARA does not guarantee finding the optimal solution, however, it solves the general problem in relatively low time and space requirements. Heuristic algorithms, which tackle this, so called, "inexact graph matching task", have been also intensively studied in the structural pattern recognition community [7,19]. Another major application field is protein interaction network alignment in systems biology [11].

Here we present HARP – Geodesic **H**ashing **A**lignment of **R**NA secondary structures including **P**seudoknots. It is a novel, efficient heuristic method for solving the general (none-restricted) pairwise pseudoknotted RNA secondary structure alignment. HARP's technique was motivated by the Geometric Hashing [22,36] ideas, which were introduced for object recognition in Computer Vision. The presented algorithm performs inexact matching of directed graphs. HARP was applied to a benchmark of 31 structures from 16 functional groups showing the algorithm's ability to efficiently and accurately align RNA secondary structures, which include pseudoknots, successfully distinguishing homologous structures from non-homologous ones.

2 Method

The input to the HARP algorithm are two RNA secondary structures represented by graphs, were the vertices represent the nucleic acid bases and edges connect bases, which are paired by chemical bonds. The output of the algorithm is a pair of large subgraphs (preferably of maximal size), one from each molecule, which are "almost" isomorphic.

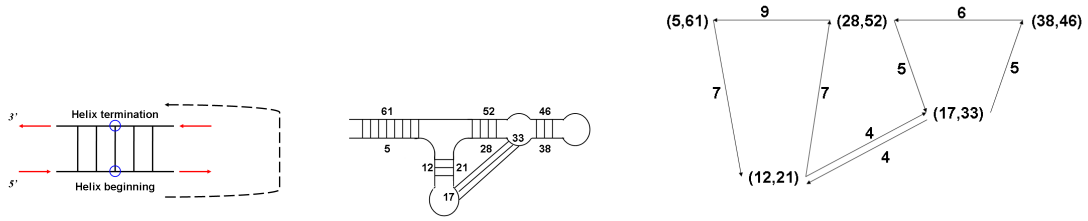
Informally, the algorithm performs as follows. First, the secondary structure graph is reduced to a directed graph, where the vertices represent significant enough helices and the edges connect chemically interacting helices. The edges are directed from 5‘ to 3‘. Next, for each pair of vertices (nicknamed **basis**) a "local view" of the other accessible vertices is created. This is done by recording for each "viewed" vertex its directed "geodesic" distances from both basis vertices. The distance information is stored in a look-up table. Since similar substructures should produce identical "views" for each pair of corresponding bases, we are interested to align bases with similar "views". Such local alignments are efficiently retrieved from the look-up table. In the final stage the local alignments are clustered and merged to produce a maximal size alignment. Since this is done by a greedy procedure, maximality is not guaranteed. In the experimental benchmark the resulting alignments are large and biologically meaningful. The alignment technique is motivated by the Geometric Hashing method [22,36].

Our **Reduced Graph Representation** exploits the biological fact that RNA molecules usually form long double helices (stems), which are connected by stretches of non paired nucleic acids (loops). Extremely short double helices are usually unstable and therefore can be disregarded.

In the reduced graph representation each **stable helix** is represented by a **vertex**. It is described by the indices of the base pair at its middle by nicknaming the lower index of that base pair as *helix beginning* and the higher index as *helix termination* (see Fig. 2(a)). Each vertex stores the following triplet: (i) helix beginning; (ii) helix termination; (iii) helix length.

A **directed edge** connects two vertices if their corresponding helices are in contact. The edge direction is determined by the polymerization direction from 5' to 3'(see Fig. 2 (a)). The **weight of an edge** is set to the minimal number of nucleotides needed in order to connect the two vertices. This weight is correlated with the number of interactions that connect the helices and thereby provides an upper bound on their 3D distance. For two vertices v_i and v_j whose helix beginning points are i_b and j_b and termination points are i_t and j_t the distance is set to: $weight(v_i, v_j) = \min\{|i_b - j_b|, |i_b - j_t|, |i_t - j_b|, |i_t - j_t|\}$. An example of converting a secondary structure to the reduced graph representation is given in Fig. 2 (b) and

(c). Finally, for a given graph the directed distances between each pair of vertices are calculated by finding the all pairs shortest directed paths [9].



(a) (b) (c)

Figure 2. Graph representation. (a) Vertex representation: The helix is represented by the indices of the base-pair in its middle. The indices are determined by the position of the nucleotides on the RNA chain according to the polymerization direction (5‘ to 3‘) described by a dashed arrow. Edge direction also corresponds to the polymerization direction. **(b)** and **(c) Converting an RNA molecule to a graph:** The RNA secondary structure and its corresponding directed weighted graph. For each helix and vertex the *helix beginning* and *helix termination* are indicated (helix length is absent). The edge direction and weight are calculated as explained in the text.

Detection of Similar Local Environments: Let $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ be the reduced graph representations of the input RNA molecules. In order to detect locally similar substructures we follow the ideas of Geometric Hashing ([22,36]) adapting them for directed geodesic distances in graphs. First, each graph is pre-processed to encode the directed distances of each vertex from any pair of other vertices, which will be nicknamed – basis pairs. This redundant encoding is stored in a look-up table for efficient retrieval.

Specifically, let $v_i, v_j \in V_1$ where $i < j$, be a basis. For each $v_k \in V_1, k \neq i, j$, we consider two directed triangles – the **forward triangle:** $v_i v_k v_j$ and the **backward triangle:** $v_j v_k v_i$. The "length" of a directed triangle edge vu is the sum of weights in the shortest path connecting v to u. The triangle edges that touch vertex v_k are called *indexing edges*, while the edge connecting the basis vertices is called the *basis edge*. The lengths of the indexing edges are used to access a two dimensional look-up table, where the following information is stored: the vertex v_k, the "forward" or "backward" triangle type, and the basis pair $v_i v_j$. Each such vertex v_k will be encoded as $(x^f_{v_k}, y^f_{v_k})$ for the forward triangle and $(x^b_{v_k}, y^b_{v_k})$ for the backward triangle. Since this representation is done for each basis pair it is highly redundant. This redundancy ensures, that significant local alignments will not be missed.

Local Alignment Seeds: In this stage we efficiently detect bases in G_1 and in G_2 that have "similar views", namely "almost" coincide on the distances of other vertices from them.

For a specific basis $v_i, v_j \in V_2$ where $i < j$, and for each node $v_k \in V_2$, the forward and backward triangles are calculated. We examine the look-up table entries within ε-vicinity[1] from the entry that corresponds to the lengths of the indexing edges of the forward and backward triangle respectively. We now list all triangles that satisfy the following: (i) **Type:** the triangle must be of the same type (forward or backward) as

[1] The ε defines our search radius (see Table 3 in the appendix for RADIUS SEARCH default parameter). The calculated distance between the entries is the l_2 distance.

the query triangle; (ii) **Length:** its basis edge is of similar length to the basis edge of the query triangle (up to an ε).

The set of triangles of some basis in V_1 that satisfy these conditions defines a "similar view" or a correspondence list between G_1 and G_2.

Refinement of Local Alignments: The correspondence lists computed at the previous stage are not necessarily one-to-one mappings, since a triangle in one graph may have more than one distance-congruent triangle in the other graph under the same "view". To resolve the conflicts we apply bipartite graph matching to refine the correspondence lists for pairs of bases, which scored more than 3 hits in the local alignment procedure.

For such a pair of bases, we define a bipartite graph $G_t = \langle V_{G_1} \cup V_{G_2}, E_t \rangle$ where V_{G_1} and V_{G_2} are the vertices of G_1 and G_2 respectively. We connect $v_i \in V_{G_1}$ with $v_j \in V_{G_2}$ if (for the given bases) the l_2 distance between the forward points $(x^f_{v_i}, y^f_{v_i})$ and $(x^f_{v_j}, y^f_{v_j})$ as well as the the backward points $(x^b_{v_i}, y^b_{v_i})$ and $(x^b_{v_j}, y^b_{v_j})$ is less than ε. We set the weight of edge, e_{v_i,v_j}, connecting v_i and v_j to: $w(e_{v_i,v_j}) = \frac{1}{C_f(1+d(v_i,v_j))} \frac{l(v_i)l(v_j)}{1+|l(v_i)-l(v_j)|^2}$, where C_f is a constant factor, $l(a)$ is the length of the helix represented by vertex a and d is the average l_2 distance between the forward and backward representation of the vertices. The weight $w(e_{v_i,v_j})$ is inversely proportional to the distance between the matched vertices and directly proportional to the length of the helices, which they represent. Based on experiments, the default value of C_f was set to 10.

The correspondence list (for a given pair of bases), which is calculated by the bipartite matching algorithm, is further pruned by removing pairs of vertices with non matching neighbors, namely, with the majority of their surrounding vertices not matched, as well as removing matched small connected components.

Alignment Extension: The local alignments of the previous stage should be further combined and extended to the largest biologically significant alignment. In the current development stage of the algorithm we have adopted a straightforward greedy approach, which starts with the largest matched set of vertices and at each step adds the local alignment that: (i) overlaps the currently matched set; (ii) adds the biggest number of vertex matches (disregarding the overlap). The procedure terminates when there are no new matches that can be added to the set. The resulting final alignment is the output of our algorithm.

Scoring: We evaluate the quality of the one-to-one mapping by the sum of matched nucleotide base-pairs ($S_{bp}(R_1, R_2)$) as a fraction of the total number of base pairs in the stable helices, which is $NS_{bp}(R_1, R_2) = \frac{S_{bp}(R_1,R_2)}{min(bp_1,bp_2)}$, where bp_1 and bp_2 are the total numbers of base-pairs in the stable helices in R_1 and R_2 respectively.

3 Results

In this section we have conducted an all-against-all alignment experiment on a benchmark of RNAs. First we present the performance of HARP on a benchmark of 31 RNA structures and present in parallel the performance of LARA, a current state-of-the-art arbitrary class pseudoknot alignment method. Then, we provide a more detailed analysis of HARP's all-against-all alignment scores. Finally, we provide a thorough examination of the HARP's alignments for some biologically interesting functional groups.

The all-against-all alignment experiment was conducted on a benchmark of 31 RNA structures belonging to 16 functional groups. All of these have high resolution 3D structures in the PDB [6]. The secondary structures were extracted by the X3DNA program from the 3DNA package [24]. To avoid redundancy the structures have less than 75 % sequence identity [34][2]. In order to avoid bias due to the size, functional group size was limited to 4. The complete list of PDBs of each functional group is listed in Table 2 of the Appendix.

3.1 Performance of HARP and comparison to LARA

We have calculated for both methods and for each functional group an average pairwise alignment score and a p-value. Since our run of LARA failed on the functional group of largest size (the 23S ribosomal RNA (rRNA) subunit whose average size is ∼2800 nucleotides) due to high memory requirements[3], this functional group was omitted from the comparison.

The **average score** of a functional group is calculated as the average of the normalized scores for each pair of structures belonging to that functional group. The normalized score of HARP is the calculated NS_{bp} (see "Scoring" paragraph of the "Methods" section). The corresponding normalized score for LARA was calculated as the alignment's identity score for a pair of structures divided by the size of the shorter structure of that pair.

A **p-value** is assigned to the score a of a pair of RNA structures within the same functional group f_i. $p(a) = \frac{N_a(R)}{\|R\|}$ where the random group R is the set of all scores between RNA structures R_i and R_j where $R_i \in f_i$ and $R_j \notin f_i$. $N_a(R)$ is the number of scores in R that have a score value above a. The size of the random group used for the p-value calculations includes functional groups of size 1. Specifically, HARP's random groups size are $\|R\| = 58$ or $\|R\| = 108$ for functional group size of 2 and 4 respectively. LARA's random groups size (due to exclusion of the 23S rRNA) are $\|R\| = 50$ or $\|R\| = 92$ for functional group size of 2 and 4 respectively.

The average scores and p-values for the different functional groups are presented in Figure 3(a) and (b). Generally, on the given benchmark for most functional groups HARP's average identity score is higher than LARA's score. Figure 3(b) also indicates that HARP's p-values are generally better than LARA's.

Ideally, an alignment method should successfully differentiate between pairs of structures from the same functional group and pairs of structures from different functional groups. In other words, the method's alignment scores within functional groups should be distinguishable from alignment scores between functional groups. This quality is well reflected in the receiver operating characteristic (ROC) curve. The ROC curve, which is widely used to evaluate a method's performance, plots the TPR (True Positive Rate) vs. the FPR (False Positive Rate) for different thresholds. The TPR corresponds to the sensitivity of the method and calculated TPR = TP + FN where TP is the number of correctly predicted pairs of the same functional group and FN is the number of incorrectly predicted pairs of different functional groups. The FPR corresponds to (1- specificity) of the method and calculated FPR = FP + TN where

[2] The sequence identity score was calculated as the number of matched nucleotides divided by the size of the smaller structure.

[3] All runtests for both methods were performed on the same PC workstation (Pentium© 4 1800 MHz processor with 1 GB internal memory) under the Linux operating system.

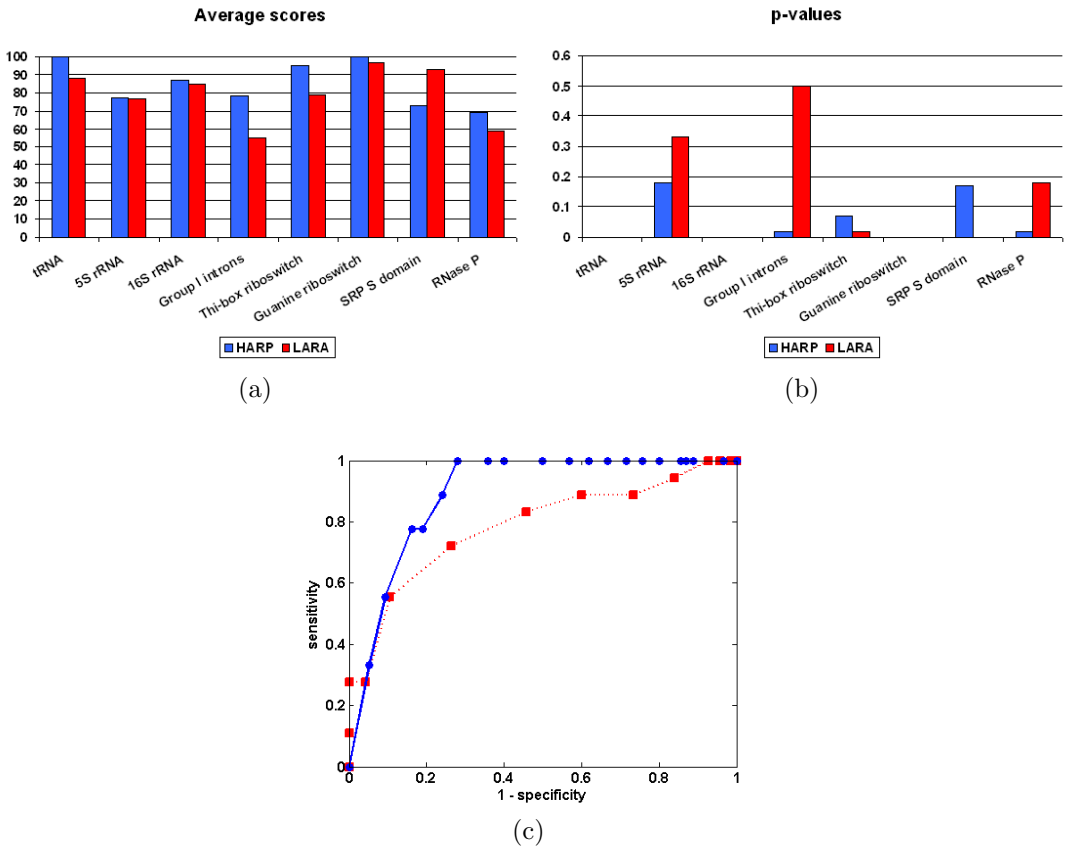


Figure 3. Comparison between HARP and LARA: (a)–(b) Average scores and p-values by functional groups: The average score and p-values presented here are only of functional of at least two structures. The group I introns is an abbreviation for group I self splicing introns pre-cleavage. RNase P is an abbreviation for RNase P catalytic domain. Average scores are given in percents. **(c) ROC curves for similar function predictors:** The curves are represented as blue circles and red rectangles for HARP and LARA respectively. The area underneath the curves is 0.89 and 0.79 for HARP and LARA respectively.

FP is the number of incorrectly predicted pairs of the same functional group and TN is the number of correctly predicted pairs of different functional groups.

The area under a ROC curve is used to evaluate the method's performance. The bigger the area underneath the ROC curve the more successful is the method in differentiating pairs of the same functional group from pairs of different functional groups. The ROC curves of HARP and LARA on the discussed benchmark are illustrated in Figure 3(c). The areas underneath the curves of HARP and LARA are 0.89 and 0.79 respectively[4].

In conclusion, for the presented benchmark, HARP's performance is better than LARA's as expressed by the following measures. First, HARP is able to align structures of very large size (e.g, the 23S rRNA). Second, HARP generally maintains higher average scores with lower p-values. Third, compared with LARA, HARP has improved ability to differentiate between pairs of structures of the same functional group from pairs of structures of different functional groups.

[4] The ROC curves were calculated for an identical data set for both HARP and LARA, omitting the 23S rRNA. Adding the 23S rRNA to HARP's data set improves its performance slightly as the area underneath the curve increases to 0.9 instead of 0.89.

3.2 Detailed analysis of HARP's scores

Table 1 details HARP's average scores and p-values for all functional groups. The highest average score (100 %) was achieved in two functional groups: tRNA and the Guanine riboswitch. Both functional groups include small structures consisting of 4 helices that differ in their topology: the guanine riboswitch contains a pseudoknot while tRNA does not. The average normalized sequence identity scores [5] for both functional groups as determined by ClustalW [34] are much lower: 50.7 % and 59.0 % for the tRNA and the Guanine riboswitch functional groups respectively. This could be accounted for compensatory mutations that do not alter the overall structure. In both groups the average pairwise tertiary identity scores as determined by ARTS [10] are also lower than those of HARP: 75.1 % and 81.7 % for the tRNA and the Guanine riboswitch functional groups respectively. This can be explained by hinges that alter the tertiary structure but have no effect on the secondary structure nor on the general functionality of the structure.

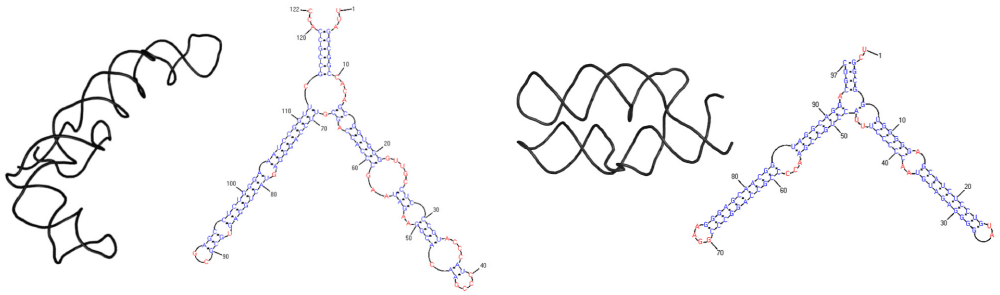
The lowest average score was achieved in the RNase P catalytic domain functional group (68.9 %). This can be explained by the fact that the molecules of RNase P, though sharing overall similar secondary structure, have some insertions and deletions (as the two differ in the number of stable helices, 16 and 19). These insertion/deletions are also expressed in LARA's low score (59.0 %) for the same structures.

Functional group	Group size	Average size (nucleotides)	Average score	p-value
tRNA	4	78	100 %	0
23S rRNA	4	2852	71.9 %	0
5S rRNA	4	120	77.2 %	0.18
16S rRNA	2	1530	86.7 %	0
Self splicing group I introns pre-cleaving	2	224	78.0 %	0.02
Thi-box riboswitch	2	80	95.0 %	0.07
Guanine riboswitch	2	69	100 %	0
SRP S domain	2	114	73.2 %	0.17
RNase P catalytic domain	2	298	68.9 %	0.02

Table 1. HARP's Detailed Statistics

Only three functional groups received a p-value bigger than 0.05: 5S rRNA, SRP S domain and the Thi-box riboswitch with p-values 0.18, 0.17 and 0.07 respectively. These p-values are attributed to high scores between the three functional groups. The biggest overall similarity between functional groups was observed between the 5S rRNA and the SRP S domain. These molecules, though having an overall similar secondary structure, do not have a common function nor a common tertiary structure (see Figure 4). Since these molecules do not contain pseudoknots and therefore are applicable to tree editing distance based methods, we have aligned them with RNAforester receiving quite similar results to those of HARP's alignment: average normalized RNAforester identity score 92 % compared with 75 % normalized HARP identity score (NS_{bp}). The high p-value of Thi-box riboswitch is explained in the same manner. This functional group has an overall similar secondary structure to both 5S rRNA and SRP S domain.

[5] The average normalized sequence identity score is calculated as the average of the normalized scores for all the structures pairs within the functional group. The normalized scores are calculated as the identity score divided by the size of the smallest structure of the pair.



(a) (b)

Figure 4. Different 3D Structures with Similar 2D Structures. (a) Secondary structure and tertiary structure of a 5S rRNA molecule (PDB:1yjw, chain 9). **(b)** Secondary structure and tertiary structure of a Signal Recognition Particle (SRP) molecule (PDB:1lng, chain B). The two RNA molecules share very little spatial similarity and have no known function in common. Nevertheless, the two molecules have very similar 2D structures with over 90 % RNAforester identity score.

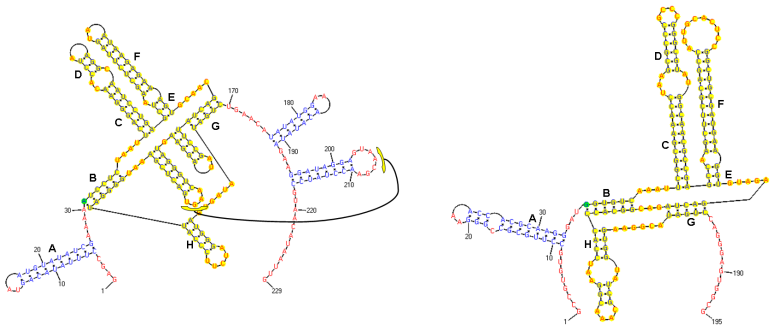
3.3 Detailed examination of HARP's alignments

Below, we provide a thorough examination of the HARP's alignments for some biologically interesting functional groups. We focus on the more challenging alignments, alignments of functional groups of size greater than 200.

Self splicing group I intron Self splicing group I introns catalyze their own excision from precursor RNA transcripts. The pseudoknotted region of the molecule is conserved throughout the different catalytic stages [30,13]. The pseudoknot actually establishes the ribozyme's catalytic core [2]. The secondary structure alignment of two self splicing group I introns done by HARP captures the entire pseudoknotted area (Figure 5). The self splicing group I introns structure variability is mainly in the 3D peripheral helices that are relatively remote from the active site. The main structure, consisting of the helices adjacent to the catalytic site, is conserved over all organisms [8]. The HARP alignment illustrated in Figure 5 includes all the helices that are 3D adjacent to the active site. Specifically, the matched helix labeled A corresponds to P2 in the P1-P2 domain. The matched helices labeled B, G and H correspond to helices P3, P7 and P8 in the P3-P9 domain. The matched helices labeled C, D, E, F correspond to helices P4, P5, P6 and P6a in the P4-P6 domain. The alignment does not include helices P7.1, P7.2, P9 and P9.1. These helices are present in only one of the structures, reflecting the variance between remote species: Homo Sapiens (PDB id 1zzn chain B) and Bacteriophage twort (PDB id 1y0q chain A).

Ribonuclease P Ribonuclease P is the enzyme that cleaves the tRNA at its 5′ and is therefore essential for the tRNA maturation. It is composed of two domains: the specificity domain and the catalytic domain. The currently solved ribonuclease P structures belong to two types according to its organism: Ancestral bacteria (A-type) and Bacillus (B-type). The two types have relatively similar secondary structures of the catalytic domain, while having considerable differences in the secondary and tertiary structures of the specificity domains [21,10].

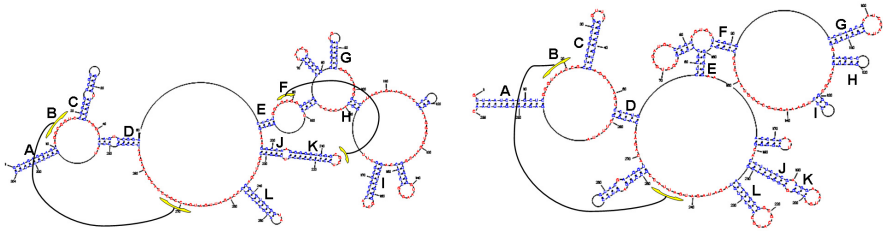
The solved tertiary structures are: PDB ids 2a2e chain A and 2a64 chain A, belonging to types A and B in correspondence. The alignment of the catalytic domains is illustrated in Fig. 6. The pseudoknot region that is conserved in both molecules was previously postulated to be important for the dynamics of the molecule [18].



(a) (b)

Figure 5. Alignment of the secondary structures of self splicing group I introns: (a) PDB id 1zzn chain B, 10 stable helices.**(b)** PDB id 1y0q chain A, 13 stable helices. The yellow arcs in the 1zzn chain B structure correspond to a helix. Matched helices are labelled by identical letters.

Even though there are two mismatches in this alignment (helices H, F) the rest of the alignment (matched 10 helices) including the conserved pseudoknot is consistent with the literature [18,35].



(a) (b)

Figure 6. Alignment of the secondary structures of the catalytic domains of ribonucleaseP: (a) PDB id 2a2e chain A, 19 stable helices **(b)** PDB id 2a64 chain A, 16 stable helices. The yellow arcs in each structure connects a helix. Matched helices are labelled by identical letters.

Ribosomal RNA Ribosomal RNA is the central component of the protein synthesis process. The prokaryotic ribosome is composed of a small unit containing the small subunit (30S) an the large subunit (50S). The small subunit includes the 16S ribosomal RNA that is ∼1500 nucleotides long that constitutes ∼85 stable helices. The large subunit includes the 5S and the 23S ribosomal RNAs that are ∼120 and ∼2800 nucleotides long and contain ∼7 and ∼140 stable helices respectively. Below are presented the results for the 16S and the 23S subunits. We focus on these units as they are of greater challenge, being very large and containing pseudoknots. Due to the large size and high complexity of the structures their alignment could not be illustrated but only literally described.

The alignment of the 16S subunit was performed on the two 16S structures, PDB ids 1yl4 chain A and 2i2u chain, containing 82 and 85 stable helices and 436 and 501 base-pairs in correspondence. The resulting alignment contains a match of 75 helices that overall include 378 base-pairs. The alignment contains 4 mismatched helices resulting in a correct match of 71 helices of 358 base-pairs (error rate of 5.3 %).

The correctly aligned helices include the central pseudoknot in 16S ribosomal RNA (indices (17, 918) in the 1yl4 chain A structure). This pseudoknot is known to be essential for ribosome stability [27]. The helices that were not matched by HARP are due to insertions/deletions.

The 23S functional group contains four structures. The pairwise alignments within the functional group had an average score of 71.9 % and 67.3 % true matching base-pairs (disregarding mismatches). In all alignments in this group the two pseudoknots are conserved. One of these pseudoknots is described by Steinberg et al. [32] as a G-motif. This motif is a highly conserved structural motif in ribosomal RNA. The other pseudoknot is also highly conserved connecting the molecule's beginning and ending.

4 Conclusions

We have presented a new heuristic algorithm, HARP, that aligns RNA secondary structures of non-restricted (arbitrary) classes of pseudoknots. HARP introduces a reduced graph representation of the secondary structures and aligns these reduced graphs by "geodesic hashing".

Evaluation of the experiments carried out on a relatively large benchmark demonstrates the biological significance of the obtained alignments. Those high quality alignments, are competitive with the results of a current state-of-the-art available RNA alignment method dealing with non-restricted classes of pseudoknots.

HARP is highly efficient: the average running time for a pair of pseudoknotted structures of the 23S ribosomal subunit ($\sim$ 2800 nucleotides) is less than a minute on a relatively weak single processor PC (Pentium© 4 1800 MHz processor with 1 GB internal memory under the Linux operating system). Currently, HARP is the only arbitrary class pseudoknots alignment method capable of aligning such big structures.

The presented algorithm is a general method for inexact matching of directed (and non directed) graphs. It detects large local "almost isomorphic" sub-structures. The algorithm's performance in other application domains will be examined.

5 Acknowledgements

This work is part of MA M.Sc. thesis. The research of HJW has been supported in part by the Israel Science Foundation (grant no. 1403/09) and the TAU Minerva-Minkowski Geometry center.

References

1. M. Abraham, O. Dror, R. Nussinov, and H. J. Wolfson: *Analysis and classification of RNA tertiary structures.* RNA, 14 2008, pp. 2274–2289.
2. P. L. Adams, M. R. Stahley, A. B. Kosek, J. Wang, and S. A. Strobel: *Crystal structure of a self-splicing group I intron with both exons.* Nature, 430 2004, pp. 45–50.
3. J. Allali and M.-F. Sagot: *Novel tree edit operations for RNA secondary structure comparison*, vol. 3240, Springer, Berlin/Heidelberg, 2004, pp. 412–425.
4. J. Allali and M.-F. Sagot: *String Processing and Information Retrieval*, Springer, Berlin/Heidelberg, 2005, ch. A Multiple Graph Layers Model with Application to RNA Secondary Structures Comparison, pp. 348–359.
5. M. Bauer, G. W. Klau, and K. Reinert: *Accurate multiple sequence-structure alignment of RNA sequences using combinatorial optimization.* BMC Bioinformatics, 8 2007, p. 271.

6. H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne: *The protein data bank.* Nucleic Acids Res., 28 2000, pp. 235–242.
7. H. Bunke and G. Allermann: *Inexact graph matching for structural pattern recognition.* Pattern Recognition Letters, 1 1983, pp. 245–253.
8. J. H. Cate, A. R. Gooding, E. Podell, K. Zhou, B. L. Golden, C. E. Kundrot, T. R. Cech, and J. A. Doudna: *Crystal structure of a group I ribozyme domain: Principles of RNA packing.* Science, 273 1996, pp. 1678–1685.
9. T. Cormen, C. Leiserson, and R. Rivest: *Introduction to Algorithms*, MIT Press, U.S.A, 1990.
10. O. Dror, R. Nussinov, and H. J. Wolfson: *ARTS: Alignment of RNA tertiary structures.* Bioinformatics, 21 Suppl. 2 2005, pp. ii1–ii7, `http://bioinfo3d.cs.tau.ac.il/ARTS`.
11. B. K. et al.: *Pathblast: a tool for alignment of protein interaction networks.* Nucleic Acids Res., 32 2004, pp. W83–W88.
12. P. A. Evans: *Finding Common RNA Pseudoknot Structures in Polynomial Time*, vol. 4009, Springer, 2006, pp. 223–232.
13. B. L. Golden, H. Kim, and E. Chase: *Crystal structure of a phage Twort group I ribozyme-product complex.* Nat. Struct. Mol. Biol., 12 2005, pp. 82–89.
14. F. Guo, A. R. Gooding, and T. R. Cech: *Structure of the Tetrahymena ribozyme: base triple sandwich and metal ion at the active site.* Mol. Cell, 16 2004, pp. 351–362.
15. M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz: *Local similarity in RNA secondary structures*, in Proceedings of Computational Systems Bioinformatics (CSB 2003), C. Stanford, ed., 2003, pp. 159–168.
16. I. L. Hofacker: *Vienna RNA secondary structure server.* Nucleic Acids Res., 31 2003, pp. 3429–3431.
17. T. Jiang, G. Lin, B. Ma, and K. Zhang: *A general edit distance between RNA structures.* Journal of Computational Biology, 9(2) 2002, pp. 371–388.
18. A. V. Kazantsev, A. A. Krivenko, D. J. Harrington, S. R. Holbrook, P. D. Adams, and N. R. Pace: *Crystal structure of a bacterial ribonuclease P RNA.* Proc. Natl. Acad. Sci. USA, 102 2005, pp. 13392–13397.
19. S. Kosinov and T. Caelli: *Inexact multisubgraph matching using graph eigenspace and clustering models*, in SSPR & SPR, vol. 2396 of Lecture Notes in Computer Science, Springer Verlag, 2002, pp. 133–142.
20. A. S. Krasilnikov, Y. Xiao, T. Pan, and A. Mondragon: *Basis for structural diversity in homologous RNAs.* Science, 1 2004, pp. 104–107.
21. A. S. Krasilnikov, X. Yang, T. Pan, and A. Mondragon: *Crystal structure of the specificity domain of ribonuclease P.* Nature, 13 2003, pp. 760–764.
22. Y. Lamdan and H. Wolfson: *Geometric Hashing: A General and Efficient Model-Based Recognition Scheme*, IEEE Computer Society Press, Tampa, Florida, USA, December 1988, pp. 238–249.
23. G. H. Lin, B. Ma, and K. Zhang: *Edit distance between two RNA structures.* Proceedings of the fifth annual international conference on Computational biology, ACM Press, 2001, pp. 211–220.
24. X.-J. Lu and W. K. Olson: *3DNA: a software package for the analysis, rebuilding and visualization of three-dimensional nucleic acid structures.* Nucleic Acids Res., 31 2003, pp. 5108–5121.
25. R. Mikhaiel, G. Lin, and E. Stroulia: *Simplicity in RNA Secondary Structure Alignment: Towards biologically plausible alignments*, IEEE Computer Society Washington, DC, USA, 2006, pp. 149–158.
26. M. Mhl, S. Will, and R. Backofen: *Lifting Prediction to Alignment of RNA Pseudoknots*, Springer Berlin, Heidelberg, 2009, ch. 5541, pp. 285–301.
27. R. A. Poot, C. W. A. Pleij, and J. van Duin: *The central pseudoknot in 16S ribosomal RNA is needed for ribosome stability but is not essential for 30S initiation complex formation.* Nucleic Acids Res., 24 1996, pp. 3670–3676.
28. P. Schimmel and K. Tamura: *tRNA structure goes from L to λ.* Cell, 113 2003, pp. 276–278.
29. S. Siebert and R. Backofen: *MARNA: multiple alignment and consensus structure prediction of RNAs based on sequence structure comparisons.* Bioinformatics, 21 2005, pp. 3352–3359.

30. M. R. Stahley and S. A. Strobel: *Structural evidence for a two-metal-ion mechanism of group I intron splicing.* Science, 309 2005, pp. 1587–1590.
31. D. W. Staple and S. E. Butcher: *Pseudoknots: RNA structures with diverse functions.* PLoS Biology, 3(6) 2005, p. 213.
32. S. V. Steinberg and Y. I. Boutorine: *G-ribo: a new structural motif in ribosomal RNA.* RNA, 13 2007, pp. 549–554.
33. C. A. Theimer, C. A. Blois, and J. Feigon: *Structure of the human telomerase RNA pseudoknot reveals conserved tertiary interactions essential for function.* Molecular Cell, 17 2005, pp. 671–682.
34. J. D. Thompson, D. G. Higgins, and T. J. Gibson: *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice.* Nucleic Acids Res., 22 1994, pp. 4673–4680.
35. A. Torres-Larios, K. K. Swinger, A. S. Krasilnikov, T. Pan, and A. Mondragon: *Crystal structure of the RNA component of bacterial ribonuclease P.* Nature, 437 2005, pp. 584–587.
36. H. Wolfson and I. Rigoutsos: *Geometric hashing: An overview.* IEEE Computational Science and Eng, 4(4) 1997, pp. 10–21.
37. K. Zhang and D. Shasha: *Simple fast algorithms for the editing distance between trees and related problems.* SIAM J. Computing, 18(6) 1989, pp. 1245–1262.
38. K. Zhang, L. Wang, and B. Ma: *Computing similarity between RNA structures*, vol. 1654, Springer, 1999, pp. 281–293.

6 Appendix

Functional group	**PDB Codes**
23S rRNA	1vqm chain 0, 1vp0 chain B 2hgu chain A, 2i2v chain B
5S rRNA	1vp0 chain A, 1yjw chain 9 2hgu chain B, 2awb chain A
16S rRNA	1yl4 chain A, 2i2u chain A
tRNA	1efw chain D, 1ttt chain F 2dxi chain D, 2hgr chain D
lambda form tRNA	1j2b chain C
Self splicing group I introns pre-cleaving	1y0q chain A, 1zzn chain B
Self splicing group I introns post-cleaving	1x8w chain B
Thiamine pyrophosphate (thi-box riboswitch)	2cky chain B, 2hoo chain A
Guanine riboswitch	1y26 chain X, 1u8d chain A
Signal recognition partical (SRP) S domain	1mfq chain A, 1z43 chain A
Ribonuclease P (RNase P) catalytic domain type A and B	2a2e chain A, 2a64 chain A
Ribonuclease P (RNase P) specificity domain type A	1u9s chain A
Ribonuclease P (RNase P) specificity domain type B	1nbs chain B
MLV Psi site	1s9s chain A
muPsi	2ihx chain B
S-adenosylmethionine riboswitch	2gis chain A

Table 2. HARP Data Set: The first four letters of an RNA name are the PDB code, followed by the chain id. Ribonuclease P of type A (Archeal) or B (Bacterial) are known to have different secondary (and tertiary) structure. The differences between the two types are more expressed in the specificity domain [20]. Therefore the Ribonuclease P structures were divided to the above three functional groups. Self splicing group I introns also change their secondary (and tertiary) structure upon cleavage, loosing both their exons and therefore having different functional groups for different stages of the catalysis [14]. Lambda form tRNA differs from the canonical "L shaped" tRNA in both secondary and tertiary structure [28] and therefor was considered separately.

Parameter name	**Parameter description**	**Default value**
MIN HELIX LENGTH	The minimal number of consecutive base-pairs that define a stable helix.	3
MAX DIST BASIS	The maximal (geodesic) distance between two vertices. Above this threshold the two vertices will not be considered as basis. Note that the graph diameter of the largest single structure is 640.	50
RADIUS SEARCH (ε)	The maximal l_2 distance between the positions of two matched points under a certain transformation.	15
C_f	The ratio between the distance constraint and size constraint in the determination of the bipartite edges weights.	10
MIN SOLUTION SIZE	The minimal number of matched helices under a given transformation. Under this threshold the transformation is ignored.	3
MIN NEIGHBOR NUM	The minimal number of matched immediate neighbors in order to consider the pair of vertices as matched.	2
MIN SIZE CONNECTED COMPONENT	The minimal size of connected component of the matched vertices. Under this size vertices matches are ignored.	3

Table 3. HARP parameters description and default values: This table presents the default values used by the HARP program. The parameters enable flexibility of the program according to various sets of demands coming from end-users. However, for most experiments the default parameter values were sufficient. These parameters were determined by a trial and error process.

Analyzing Edit Distance on Trees: Tree Swap Distance is Intractable

Martin Berglund

Department of Computing Science, Umeå University
90187 Umeå, Sweden
mbe@cs.umu.se

Abstract. The string correction problem looks at minimal ways to modify one string into another using fixed operations, such as for example inserting a symbol, deleting a symbol and interchanging the positions of two symbols (a "swap"). This has been generalized to trees in various ways, but unfortunately having operations to insert/delete nodes in the tree *and* operations that move subtrees, such as a "swap" of adjacent subtrees, makes the correction problem for trees intractable. In this paper we investigate what happens when we have a tree edit distance problem with *only* swaps. We call this problem tree swap distance, and go on to prove that this correction problem is NP-complete. This suggests that the swap operation is fundamentally problematic in the tree case, and other subtree movement models should be studied.

1 Introduction

String edit distance is an old, well-known and thoroughly studied concept, most commonly used in the context of *string correction problems*. An edit distance (of which there are many kinds) defines some small set of operations on strings. An instance of the string correction problem corresponding to a given edit distance is a question of the form "can the string s be transformed into s' by applying at most k edit operations?" In more complex cases the string correction problem may associate different costs to the edit operations, having k serve as a total budget.

One of the most frequently used types of edit distance is Levenshtein distance [7], which features the three operations `delete`, `insert`, and `replace`. These can be applied to any position in a string, to delete a single symbol, insert a single symbol, and replace a single symbol by another, respectively. A popularly applied extension, called Damerau-Levenshtein distance [3], adds a fourth operation, `swap`, which swaps the position of any two symbols in a string. For both of these distances the string correction problem is very efficiently solvable if all operations have the same cost. A more general variant is called the *extended string-to-string correction problem*, which uses the four Damerau-Levenshtein operations, but allows the problem instance to assign each operator an arbitrary integer cost [11]. In general this makes the correction problem strongly NP-complete [10], a fact that we will make use of later.

As this area is well-explored and successful in the string case it is of great interest to extend the same ideas to the tree case [8,9]. This work has been very successful for the "insert", "delete" and "replace" operations, but the "swap" operation has most often been left out [12,5,2]. This is in fact a necessity, as the problem quickly becomes intractable when subtree movement is introduced as an operation. This follows trivially from the fact that tree edit distance on unordered trees is NP-complete [13], by duplicating nodes one can create a situation where the swaps are so much cheaper than a `delete`/`insert` operation that the problem becomes equivalent to the unordered

Martin Berglund: Analyzing Edit Distance on Trees: Tree Swap Distance is Intractable, pp. 59–73.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

one. Still, swaps and other subtree movement operations remain very interesting in practice in very diverse fields such as XML processing, computational biology, natural language processing and many others. Approximations have been considered, for example [1] introduces swaps into tree edit distance but the algorithm as given actually restricts each node to participate in at most one swap, so arbitrary reorderings are not possible.

While much work has been done to restrict the swaps to make the problem tractable we will here instead take a step back and consider the "tree swap distance" problem. In this restriction of tree edit distance *only* the `swap` operation is allowed, reducing the problem to finding the least number of swaps necessary to reorder one tree into another. Unfortunately the end result is that we demonstrate that even this problem is NP-complete, suggesting that the `swap` operation may be a computationally bad choice to model subtree movement operations.

2 Preliminaries

Let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, 2, 3, \ldots\}$. For all $n \in \mathbb{N}$ let $[n]$ denote the set $\{1, \ldots, n\}$. An *alphabet* Σ is a finite set of symbols. Going forward we will simply use Σ to mean some appropriate alphabet without specifying it precisely. The empty string/sequence is denoted by ϵ. The set of all strings over an alphabet Σ is denoted Σ^* and is defined as $\Sigma^* = \{\epsilon\} \cup \{\alpha v \mid \alpha \in \Sigma, v \in \Sigma^*\}$. The length of a string $v \in \Sigma^*$ is denoted $|v|$. The set of sequences over an arbitrary set S is also denoted S^*, the sequence $s_1, \ldots, s_n$ is referred to as an n-tuple. When expedient we may abuse notation and confuse the n-tuple $s_1, \ldots, s_n$ with the string $s_1 \cdots s_n$.

An n by n matrix (all our matrices are square) is an n-tuple of n-tuples $M = ((x_{1,1}, \ldots, x_{1,n}), \ldots, (x_{n,1}, \ldots, x_{n,n}))$ with $x_{i,j} \in \mathbb{N}$ for all $i, j \in [n]$. We say that $x_{i,j}$ is on row i and column j, and denote it by $M_{i,j}$.

A tree t consists of a root node labeled by some symbol $\alpha \in \Sigma$ and a tuple of zero or more direct child subtrees $(t_1, \ldots, t_n)$ (for any $n \in \mathbb{N}$) over the same alphabet. t is denoted by $\alpha[t_1, \ldots, t_n]$. For a tree $\alpha[]$ with zero children we may abbreviate it as simply α. The set of all trees over Σ, denoted by T_Σ, is defined as $T_\Sigma = \Sigma \cup \{\alpha[t_1, \ldots, t_n] \mid \alpha \in \Sigma, n \in \mathbb{N}, t_1, \ldots, t_n \in T_\Sigma\}$.

The set of positions in a tree is defined by a function $pos : T_\Sigma \to 2^{\mathbb{N}^*}$. For any $k \in \mathbb{N}$, including zero, $\alpha \in \Sigma$ and $t_1, \ldots, t_k \in T_\Sigma$ the definition of $pos\big(\alpha[t_1, \ldots, t_k]\big)$ is $\{\epsilon\} \cup \big\{(i, v_1, \ldots, v_n) \mid i \in \{1, \ldots, k\}, (v_1, \ldots, v_n) \in pos(t_i)\big\}$. That is, a position $p \in pos(\alpha[t_1, \ldots, t_n])$ denotes the root note α if $p = \epsilon$, otherwise p is of the form $(i, v_1, \ldots, v_n)$ referring to the position $(v_1, \ldots, v_n)$ in the subtree t_i.

3 The Extended String-to-String Correction Problem

A (pre-existing) problem that we will make use of in the coming proof will now be defined. Later on we will use a reduction from an instance of the *extended string-to-string correction problem* (ESSCP) to our problem to show strong NP-hardness. The ESSCP is known to be NP-complete (problem [SR20] in [4]), shown in the case where the cost of inserts and replacements is made infinite and when swaps and deletes are given a constant cost [10]. The formulation by Wagner in [10] allows arbitrary costs for deletes and any non-zero cost for swaps, while the formulation in [4] fixes both costs to 1. Here we opt to set the cost of a single swap to 1 and the cost of deletes to

0, this causes no loss of generality, since the number of deletes in a solution is always the difference in length between the source and target strings. The problem definition is divided into three parts, for all $\alpha_1 \cdots \alpha_n \in \Sigma^*$:

Definition 1 (String deletes). *For all* $\{d_1, \ldots, d_m\} \subseteq [n]$ *we define the delete function as* $delete(\alpha_1 \cdots \alpha_n, \{d_1, \ldots, d_m\}) = \alpha_{i_1} \cdots \alpha_{i_{n-m}}$ *where* $i_1 < \cdots < i_{n-m}$ *and* $\{i_1, \ldots, i_{n-m}\} = [n] \setminus \{d_1, \ldots, d_m\}$.

Definition 2 (String swaps). *We define the swap function by letting* $\mathtt{swap}(s, \epsilon) = s$ *for all strings* s *and for all* $(s_1, \ldots, s_m) \in [n-1]^*$ *letting*

$$\mathtt{swap}(\alpha_1 \cdots \alpha_n, (s_1, \ldots, s_m)) = \mathtt{swap}(\alpha_1 \cdots \alpha_{s_1-1}\alpha_{s_1+1}\alpha_{s_1}\alpha_{s_1+2} \cdots \alpha_n, (s_2, \ldots, s_m)).$$

Definition 3 (The delete/swap ESSCP). *An instance of the delete/swap ESSCP (over some alphabet* Σ*) is a tuple* $(S, T, b) \in \Sigma^* \times \Sigma^* \times \mathbb{N}$. *The instance is a "yes" instance (the answer is "yes") if and only if there exists some* $D \subseteq [|S|]$ *and* $W \in [|S| - |D| - 1]^*$ *such that* $\mathtt{swap}(\mathtt{delete}(S, D), W) = T$ *with* $|W| \leq b$. *We denote the set of all such "yes" instances* ESSCP_{ds}.

There are a couple of important things to notice here.

- The definition is stated so that all deletes happen before any swap. This is not a restriction of the problem, since there is no instance where it is better to delete something after moving it around.
- b is in all interesting instances polynomial in the size of the instance, since all reorderings can be realized in less than n^2 swaps. We therefore, without loss of generality, assume b to be coded in unary in the input, so $\mathrm{ESSCP}_{\mathrm{ds}}$ is strongly NP-complete.
- Swaps of unrelated symbols can be reordered freely. One recurring example is that if $\mathtt{swap}(\alpha_1 \cdots \alpha_n, W)$ is such that the symbol α_i is moved to the end of the string by W we can trivially restructure W to start with the sequence $i, i+1, \ldots, n-1$, without making W longer. That is, if a minimal swap sequence moves the symbol in position i to the last position n then doing this before anything else cannot make the swap sequence longer, since keeping the symbol in the middle of the string for longer serves no purpose.

4 Swap Assignment Problem

Now we will define the first original problem, the swap assignment problem. We will demonstrate that this problem is strongly NP-complete by a reduction from $\mathrm{ESSCP}_{\mathrm{ds}}$. This problem will serve as a stepping stone to demonstrate NP-completeness for the tree swap distance problem.

This problem is quite similar to the classical assignment problem [6], except a starting assignment is given, and an optimal assignment is to be reached by swapping adjacent assignments. The swap function is defined exactly as in the string case, when the matrix is viewed as a string of rows.

Definition 4 (Matrix Row Swap). *For an* n *by* n *matrix* M *the swap function is defined by for all* $W \in [n-1]^*$ *simply viewing the matrix as a string of rows:* $(M_{1,1}, \ldots, M_{1,n}) \cdots (M_{n,1}, \ldots, M_{n,n})$ *and applying the string swap* $\mathtt{swap}(M, W)$.

Definition 5 (The Swap Assignment Problem). *An instance of the swap assignment problem is a tuple (M, b) where $b \in \mathbb{N}$, and M is an n by n matrix. The instance is a "yes" instance if and only if there exists some $W \in [n-1]^*$ such that*

$$b \geq |W| + \sum_{i=1}^{n} \mathtt{swap}(M, W)_{i,i}.$$

We denote the set of all such "yes" instances SAP.

Let us look at a small instance to better understand the problem.

Example 6. As an example swap assignment problem instance we can take (M, b) with $b = 9$ and M as below.

$$M = \begin{bmatrix} 4 & 5 & 16 & 0 \\ 3 & 4 & 16 & 0 \\ 2 & 3 & 0 & 16 \\ 1 & 2 & 16 & 16 \end{bmatrix} \quad M' = \begin{bmatrix} 4 & 5 & 16 & 0 \\ 1 & 2 & 16 & 16 \\ 2 & 3 & 0 & 16 \\ 3 & 4 & 16 & 0 \end{bmatrix}.$$

Since we can use the swaps $W = 3, 2, 3$ to construct $M' = \mathtt{swap}(M, W)$ as shown above, it follows that $(M, b) \in$ SAP. M' has the diagonal sum 6 which together with the three swaps adds up to exactly 9. We could also equivalently solve the problem instance using the swap-sequence $W' = 1, 3, 2, 3$ which produces a diagonal cost of $3 + 2 + 0 + 0 = 5$ but, on the other hand, requires 4 swaps, again giving a total of 9.

The ESSCP_{ds} (Definition 3) can be reduced to the swap assignment problem in a slightly tricky to visualize but functionally straightforward way.

Definition 7 (ESSCP to Swap Assignment Reduction). *Take a delete/swap ESSCP instance $(s_1 \cdots s_n, t_1 \cdots t_m, b)$ (we assume that $m \leq n$, otherwise it is trivial). Then construct a swap assignment problem instance (M, b') where the n by n matrix M is constructed by taking:*

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq m \text{ and } s_i = t_j, \\ b' + 1 & \text{if } j \leq m \text{ and } s_i \neq t_j. \\ n + i - j & \text{if } j > m, \end{cases},$$

and $b' = b + n(n - m)$.

This definition is not really intuitive, but a short example should explain the idea of how this represents an ESSCP instance.

Example 8. Let us consider the delete/swap ESSCP instance $(aacb, abc, 1)$. This has a fairly simple solution, delete one of the "a" symbols and swap the "b" and "c". The reduction computes $b' = 1 + 4(4 - 3) = 5$ and the matrix

$$M = \begin{bmatrix} 0 & 6 & 6 & 1 \\ 0 & 6 & 6 & 2 \\ 6 & 6 & 0 & 3 \\ 6 & 0 & 6 & 4 \end{bmatrix}.$$

We will look at the left part first, the part that corresponds to the first two cases of the construction. All these cells are set either to 0 or to $b' + 1$, which means that

none of the non-zero cells *may ever* be on the diagonal of a solution, since the sum would always be greater than the budget. So, the first three positions on the diagonal (counting from the upper left) must be made zero in a solution, the three corresponds to the length of the target string. The idea is that a zero on the diagonal in this first part corresponds to a correctly matched symbol. The cells on the right-hand side only come into play on the last part of the diagonal, the bottom few rows of the result. The rows moved to the bottom correspond to symbols that get deleted.

The motivation for the weight $n + i - j$ in case 3 of the reduction is that if we wish to delete some symbol in the original string problem we have a fixed cost (zero), but to move a row to the bottom of the matrix has different cost depending on where the row starts out, since different numbers of swaps need to be used. The cost the rows that end up at the bottom contribute to the diagonal is there to counteract this. Let us look at the two ways to solve this instance, see Figure 1. Here we show the

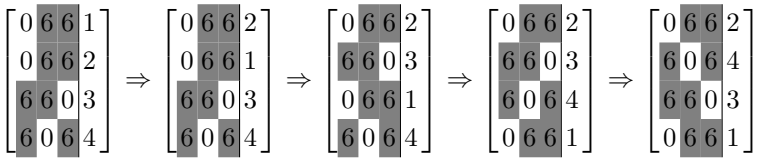


Figure 1: A solution for the the swap assignment problem instance produced by reducing from $(aacb, abc, 1) \in \mathrm{ESSCP}_{\mathrm{ds}}$

solution equivalent to deleting the first "a", by swapping the top row down to the bottom with the first three swaps. This row then contributes cost 1 to the diagonal, for a total cost of 4 to get rid of the first symbol. Then we swap the rows that were originally 3 and 4 (going from "acb" to "abc") to move the zeros to the diagonal. The total cost of the solution is 5, which fits the budget b'.

What is key is that the solution can choose to delete any symbol without the cost being different. So let us look at the other possibility, where we delete the second "a" instead, shown in Figure 2. Here we start by swapping the second row, corresponding

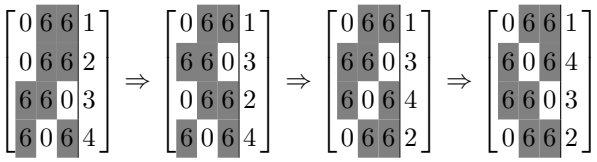


Figure 2: An alternative solution for the swap assignment problem instance produced by reducing from $(aacb, abc, 1) \in \mathrm{ESSCP}_{\mathrm{ds}}$

to the second "a" into the last position. This takes only 2 swaps, but this row contributes a cost of 2 to the diagonal, again making the delete cost exactly 4. A final swap of the original row three and four again produces a solution with cost 5.

This illustrates the key property of the construction, deletions are substituted with moving the rows in question into bottom positions, and the costs in the rows are constructed so that a row that is originally far from the bottom gets a proportionally larger "discount" on the diagonal sum to pay for the extra swaps needed to delete

them. The formula for the rightmost column is $n + i - j$, the subtraction of j comes into play when multiple symbols are deleted. Since not all rows can go to the bottom position later deletions will have a shorter distance to travel than the first ones, this is counteracted by the costs being greater in the "discount columns" further left. As a final example see the slightly larger instance in Figure 3.

$$\left[\begin{array}{ccc|cc} 12 & 12 & 0 & 2 & 1 \\ 12 & 0 & 12 & 3 & 2 \\ 0 & 12 & 12 & 4 & 3 \\ 0 & 12 & 12 & 5 & 4 \\ 12 & 12 & 0 & 6 & 5 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|cc} 0 & 12 & 12 & 4 & 3 \\ 12 & 0 & 12 & 3 & 2 \\ 12 & 12 & 0 & 6 & 5 \\ 12 & 12 & 0 & 2 & 1 \\ 0 & 12 & 12 & 5 & 4 \end{array}\right]$$

Figure 3: Reducing $(cbaac, abc, 1) \in \mathrm{ESSCP}_{\mathrm{ds}}$ produces the swap assignment problem instance with the left matrix and budget $b' = 11$. "Deleting" a row ends up with a cost of 5 counting swaps and diagonal cost. On the right is the solution which performs the swaps $4, 1, 2, 3, 1$ for a total cost of 11. This solution corresponds to deleting the last "a", deleting the first "c" and finally swapping the remaining "b" and "a".

Lemma 9. *The reduction in Definition 7 produces a swap assignment problem instance that answers "yes" if and only if the original delete/swap ESSCP instance answers "yes".*

Proof (Sketch). Starting with the "if" direction, take some $(s_1 \cdots s_n, t_1 \cdots t_m, b) \in \mathrm{ESSCP}_{\mathrm{ds}}$. Let the deletes and swaps that solves this instance be $\{d_1, \ldots, d_{n-m}\} \subseteq [n]$ and $W \in [m-1]^*$. Construct (M, b') using the reduction. Assume that $d_1 > d_2 > \cdots > d_{n-m}$ then construct the swaps:

$$W_d = d_1, d_1 + 1, \ldots, n-1, d_2, d_2 + 1, \ldots, n-2, \ldots, d_{n-m}, \ldots, m$$

That is, take row d_1, which corresponds to the last (position-wise) symbol deleted in the original string, and swap it into the last position in the matrix. Then swap row d_2 (second to last deleted position) into the second to last position in the matrix and so on. Now construct $W' = W_d W$ (concatenating the two), after applying the swaps W_d the top m rows in the matrix correspond to the positions which are *not* deleted, and we perform the swaps in W on these.

Now we will just demonstrate that $(M, b') \in \mathrm{SAP}$ using W' as the solution. $|W'| = |W_d| + |W|$ and $|W_d|$ contains $(n-i) - d_i$ swaps to place the row initially at d_i into position $n-i$, for each $i \in [n-m]$. So the row (initially at) d_i will contribute $M_{d_i, n-i}$ to the final diagonal sum. The range of i means that $M_{d_i, n-i} = n + d_i - (n-i) = d_i + i$ (since all these positions are filled by the third case in the construction of M in Definition 7). Taking the swaps and diagonal contribution together each of the d_i rows contribute to the total cost by $(n-i) - d_i + d_i + i = n$, meaning that

$$|W_d| + \sum_{i=m+1}^{n} \mathtt{swap}(M, W')_{i,i} = (n-m)n.$$

This establishes that $b' = b + (n-m)n \geq |W'| + \sum_{i=m+1}^{n} \mathtt{swap}(M, W')_{i,i} = |W| + (n-m)n$, since $b \geq |W|$ and $|W'| = |W_d| + |W|$.

All that needs to be added is the remainder of the diagonal, so next we show that $\sum_{i=1}^{m} \texttt{swap}(M, W')_{i,i}$ is zero. Take $M' = \texttt{swap}(M, W_d)$ and $S' = \texttt{delete}(s_1 \cdots s_n, D)$ and simply note that if the symbol in position i in S' started out in position l then row i in M' started out in position l in M. The next step for both S' and M' is to apply W, meaning that row $j \in [m]$ in the matrix started out as row i if and only if symbol in position j in the final string was originally s_i. Since this is a solution for the ESSCP instance this means that $s_i = t_j$ which means that row i in M ends up in position j in $\texttt{swap}(M, W')$ if and only if $s_i = t_j$. It follows that the new row contributes $M_{i,j}$ to the diagonal, and the construction of M is such that set $M_{i,j} = 0$ when $s_i = t_j$.

Since we showed that $b' \geq |W'| + \sum_{i=m+1}^{n} \texttt{swap}(M, W')_{i,i}$ above and showed that $\sum_{i=1}^{m} \texttt{swap}(M, W')_{i,i} = 0$ here it follows that $b' \geq |W'| + \sum_{i=1}^{n} \texttt{swap}(M, W')_{i,i}$ so $(M, b') \in \text{SAP}$.

The "only if" direction remains but works in a very similar way. Assume that $(M, b') \in \text{SAP}$ is constructed from some delete/swap ESSCP instance (S, T, b). Let W' be the swaps that solve (M, b'). Notice that if such a solution W' exists then a solution exists which has the structure $W' = W_d W$ (that is, which first swaps all the $n - m$ bottom rows into position), if row i is going to be swapped into position n nothing can be gained by not doing so as the first thing in the swap sequence. Using this we can extract the solution to the string problem instance, deleting the symbols corresponding to rows swapped below the mth row. The solution to (M, b') also cannot do better than the fixed cost $(n - m)(n - 1)$ for swaps and diagonal of these bottom rows, and it has to place the top m rows so that they all contribute zero to the diagonal (all other positions being $b' + 1$ which is impossible in a solution), which corresponds directly to matching symbols correctly. □

Corollary 10. *The swap assignment problem is strongly NP-complete.*

This follows since ESSCP_{ds} is strongly NP-complete and the reduction constructs a polynomially sized matrix containing numbers that are all bounded by a polynomial in the original instance (recall that b is polynomial in all relevant cases and assumed to be unary). The problem is in NP since no swap sequence ever needs to be longer than n^2, allowing W' to be guessed.

5 Swap Even-Cost Assignment Problem

Now we will define a very minor restriction on the swap assignment problem. This will turn out to be key to make the final reduction to the tree swap distance problem simple.

Definition 11. *Let $2 \,|\, x$ denote that x is even ($x \in \{0, 2, 4, 6, \ldots\}$), let $2 \nmid x$ denote that x is odd.*

Definition 12 (Swap Even-Cost Assignment Problem). *An instance of the swap even-cost assignment problem is a swap assignment problem instance (M, b) such that $2 \mid M_{i,j}$ for all $i, j \in [n]$. The answer to (M, b) is "yes" if and only if $(M, b) \in \text{SAP}$. We denote the set of all "yes" instances as* SecAP*.*

We will quickly establish that all swap assignment problem instances have an equivalent swap even-cost assignment problem instance.

Definition 13. *Let* $h(x) = \left\lceil \frac{x}{2} \right\rceil$.

Definition 14 (Reducing SAP to SecAP). *Let* (M, b) *be an instance of the swap assignment problem with* M *an* n *by* n *matrix, we then construct* (M', b')*, where* M' *is a* $2n$ *by* $2n$ *matrix, by letting* $b' = b + \frac{n(n-1)}{2}$ *and taking*

$$M'_{i,j} = \begin{cases} M_{i,h(j)} & \text{if } i \leq n,\ 2 \nmid j \text{ and } 2 | M_{i,h(j)}, \\ b'' & \text{if } i \leq n,\ 2 \nmid j \text{ and } 2 \nmid M_{i,h(j)}, \\ M_{i,h(j)} - 1 & \text{if } i \leq n,\ 2 | j \text{ and } 2 \nmid M_{i,h(j)}, \\ b'' & \text{if } i \leq n,\ 2 | j \text{ and } 2 | M_{i,h(j)}, \\ 0 & \text{if } i > n \text{ and } h(j) = i - n, \\ b'' & \text{if } i > n \text{ and } h(j) \neq i - n, \end{cases}$$

where b'' *is the smallest even number strictly larger than* b'.

This definition is also a bit daunting but the underlying thinking is fairly straightforward, let us look at an example.

Example 15. We will start with an instance of the swap assignment problem instance (M, b), where $b = 11$ and M is shown on the left in Figure 4. For this example $b' = 14$,

$$M = \begin{bmatrix} 2 & 3 & 3 \\ 9 & 4 & 12 \\ 1 & 2 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 16 & 16 & 2 & 16 & 2 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 16 & 0 & 2 & 16 & 8 & 16 \\ 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 16 & 16 & 16 & 16 & 0 & 0 \end{bmatrix}$$

Figure 4: Example of applying the even-cost reduction to a swap assignment problem instance

so $b'' = 16$. Let us look at the upper half of the matrix first. The thing to notice about this part is that for all $i, j \in [n]$ there are for each pair $(M_{2i-1,j}, M_{2i,j})$ only two cases, either the pair is $(M_{i,j}, 16)$ if $M_{i,j}$ was even, or it is $(16, M_{i,j} - 1)$ if $M_{i,j}$ was odd.

This starts making sense when we look at the lower half of the matrix, which is filled with rows such that for each $j \in [n]$ the row at position $n + j$ can only be in either position $2j - 1$ or $2j$ in a valid solution (since that brings the rows zero positions to the diagonal, and b'' is guaranteed to be more than the budget). This means that any valid solution will be structured so that for each $j \in [n]$ one of the positions $2j - 1$ and $2j$ contains the row originally in position $n + j$ (in all other positions it would contribute b'' to the diagonal making the solution impossible) and the other position contains some row originally in the top half (since all rows from the bottom half are already accounted for). The $\frac{n(n-1)}{2}$ part of the budget is exactly enough to pay for the minimal such interspersing (where the row from the top half is the one at the $2j - 1$ position since that is closer).

Let $i \in [n]$ be the initial position of the row from the top that ends up in position $2j - 1$ or $2j$, this row is supposed to simulate the cost $M_{i,j}$ on the diagonal. If $M_{i,j}$ is even this is easy, the row can be placed at position $2j - 1$ (since it will

have $M'_{i,2j-1} = M_{i,j}$), if $M_{i,j}$ contained an odd number however the construction has made $M_{i,2j-1} = b''$, which forces the solution to take an extra swap to bring the row to position $2j$. This extra swap fixes the cost that was lost when the construction rounded down $M'_{i,2j} = M_{i,j} - 1$.

To make this more visual see Figure 5. Since this solution involves a total of

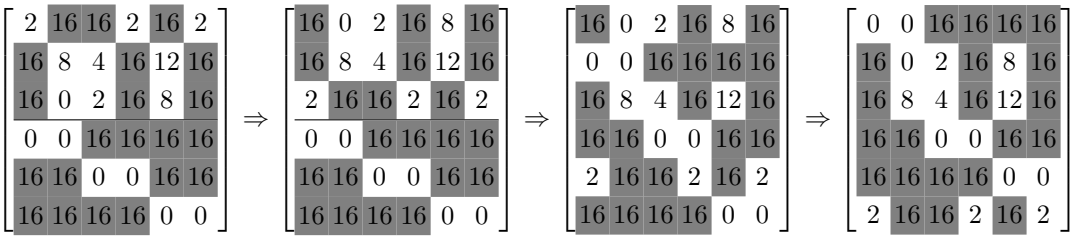


Figure 5: Some steps of the solution of the problem instance in Figure 4

seven swaps several are done in each step. Let us first note that a solution for the original (pre-reduction) instance in Figure 4 is to swap $2, 1, 2$, giving a diagonal sum of $1 + 4 + 3 = 8$ and a total solution cost of 11. In Figure 5 we have the original reduced matrix on the left, in the first step we do the same three swaps $2, 1, 2$. In the next step we intersperse the rows from the bottom half with the top with the swaps $3, 2, 4$. This however leaves us with 16 in two places on the diagonal, and have to finish with the swaps $1, 4$. These last swaps are key. Notice how the diagonal in the original instance ended up being $1 + 4 + 3$, the first and last positions are odd. The construction took these odd numbers, rounded them down to something even and placed this rounded result on the right side of its horizontal "pair" in the top row. This forces the solution to do extra swaps to bring the rows down one step further, paying the cost that was removed by the rounding. In total the solution here makes 8 swaps, and has a diagonal sum of 6, for a total cost of 14, exactly the budget b'.

Lemma 16. *For every swap assignment problem instance (M, b) (M is n by n) the reduction in Definition 14 produces a swap even-cost assignment problem instance (M', b') such that $(M', b') \in$ SecAP if and only if $(M, b) \in$ SAP.*

Proof (Sketch). Assume that $(M, b) \in$ SAP. Let W be a swap sequence that solves (M, b). Then construct a (minimal) swap sequence W_i such that

$$\texttt{swap}(a_1 \cdots a_n b_1 \cdots b_n, W_i) = a_1 b_1 a_2 b_2 \cdots a_n b_n,$$

and, let $W_o = o_1 \cdots o_m$ be such that $o_1 < \cdots < o_m$ and $2 \nmid \texttt{swap}(M, W)_{i,i}$ if and only if $i \in \{o_1, \ldots, o_m\}$. Then $W' = W W_i W_o$ (the concatenation) is a solution for (M', b'). This sequence of swaps being a solution is quickly established, noting that $|W_i| = \frac{n(n-1)}{2}$ which accounts for the difference between b' and b, and then noting that the construction makes all the swaps in W_o necessary.

The other direction amounts to assuming the existence of W' and then extracting the W part which concerns the internal order of the n first rows. □

Corollary 17. *The swap even-cost assignment problem is strongly NP-complete.*

This follows from the above. The reduction from the strongly NP-complete swap assignment problem is clearly polynomial, the matrix dimensions are doubled and the values in the matrix grow on the order of $\mathcal{O}(n^2)$. The problem is in NP, since SecAP is simply SAP with inputs restricted to even numbers.

6 Tree Swap Distance Problem

This section will reach the goal of the paper, defining the tree swap distance problem and then demonstrating that it is strongly NP-complete by a reduction from SecAP. Let us define the problem.

Definition 18 (Tree Swap). *Take any tree $t = \alpha[t_1, \ldots, t_n] \in T_\Sigma$ and any $P = (p_1, \ldots, p_m) \in pos(t)$ such that $(p_1, \ldots, p_{m-1}, (p_m + 1)) \in pos(t)$. Then define the single-swap function*

$$\mathtt{swap}_1(t, P) = \begin{cases} \alpha[t_1, \ldots, t_{p_1-1}, \mathtt{swap}_1(t_{p_1}, (p_2, \ldots, p_m)), t_{p_1+1}, \ldots, t_n] & \text{if } m > 1, \\ \alpha[t_1, \ldots, t_{p_1-1}, t_{p_1+1}, t_{p_1}, t_{p_1+2}, \ldots, t_n] & \text{otherwise.} \end{cases}$$

The full swap function is for (appropriate) positions $P_1, \ldots, P_p$ defined as

$$\mathtt{swap}(t, (P_1, \ldots, P_p)) = \mathtt{swap}_1(\ldots \mathtt{swap}_1(\mathtt{swap}_1(t, P_1), P_2) \ldots, P_p).$$

The definition of swaps for trees is slightly unwieldy, but the swap function takes a tree and a sequence of tree positions (which are integer sequences). The positions identify, in order, the subtree which should next swap position with its sibling immediately to the right. Notice that P_i for $i > 1$ does not refer to a position in the tree t but to a position in an intermediary tree, it may be that $P_i \notin pos(t)$. An example is shown in Figure 6.

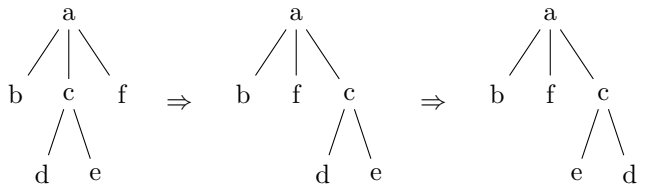


Figure 6: An example of applying the tree swaps $((2), (3, 1))$ to a small tree. That is, going from the first to second tree we swap the position 2, referring to the second child of the root, next the position $(3, 1)$ is swapped, referring to the first child of the rightmost child subtree of the root.

The definition of the tree swap distance problem now follows a familiar formula.

Definition 19 (The Tree Swap Distance Problem). *An instance of the tree swap distance problem is a tuple (t, t', b) where $t \in T_\Sigma$ is the start tree, $t' \in T_\Sigma$ is the target tree and $b \in \mathbb{N}$ is the budget. The instance is a "yes" instance if and only if there exists some $P_1 \in \mathbb{N}^*, \ldots, P_n \in \mathbb{N}^*$ such that $n \leq b$ and $t' = \mathtt{swap}(t, (P_1, \ldots, P_n))$. We denote the set of all such "yes" instances* TSwD.

The next definition is used to make it easier to talk about minimal swap sequences.

Definition 20 (Minimal budget for TSwD). *For all $t, t' \in T_\Sigma$ let $mincost(t, t') = b$, where $b \in \mathbb{N}$ is the smallest number for which $(t, t', b) \in$ TSwD. If no such number exists let $b = \infty$.*

The reduction from SecAP to TSwD requires some building blocks. A visual example of the different types of notation defined below is shown later in Figure 8.

Definition 21 (Number Tree). *Assume that* $0, 1 \in \Sigma$. *For some symbol* $\alpha \in \Sigma$ *and* $x, y \in \mathbb{N}$ *such that* $x \leq y$ *we let* $\alpha[x : y]$ *denote the tree* $\alpha[p_1, \ldots, p_{y+1}]$ *where* $p_i = 0$ *for all* $i \neq x + 1$ *and* $p_{x+1} = 1$.

For example, $\alpha[2 : 3] = \alpha[0, 0, 1, 0]$. We call these trees "number trees". Notice that for all $x, x', y \in \mathbb{N}$ such that $x \leq y$ and $x' \leq y$ it holds that $mincost(\alpha[x : y], \alpha[x' : y]) = |x - x'|$. That is, the minimum number of swaps needed to turn $\alpha[x : y]$ into $\alpha[x' : y]$ is exactly $|x - x'|$. The tree $\alpha[x : y]$ serves the purpose to represent the number x, with the minimal swap distance to any other $\alpha[x' : y]$ being the absolute difference between x and x'.

Definition 22 (Number Trees with Neutral Elements). *Assume that for each* $\alpha \in \Sigma$ *there exists a distinct* $\alpha' \in \Sigma$. *Then for all* $x, y \in \{0, 2, 4, 6, \ldots\}$ *let* $\alpha\langle x : y\rangle$ *denote the following special tree.*

$$\alpha\langle x : y\rangle = \alpha\left[\alpha\left[\frac{x}{2} : \frac{y}{2}\right], \alpha'\left[\frac{y-x}{2} : \frac{y}{2}\right]\right].$$

Additionally let $\alpha\langle \bot : y\rangle$ *denote the special tree* $\alpha\left[\alpha\left[0 : \frac{y}{2}\right], \alpha'\left[0 : \frac{y}{2}\right]\right]$, *called a "neutral" tree.*

So, for example $\alpha\langle 2 : 6\rangle$ is the tree $\alpha[\alpha[0, 1, 0, 0], \alpha'[0, 0, 1, 0]]$. These trees have the property that for all $x, x', y \in \{0, 2, 4, 6, \ldots\}$ it holds that $mincost(\alpha\langle x : y\rangle, \alpha\langle x' : y\rangle) = |x - x'|$. This should not be a surprise, these trees behave like the earlier number trees, only the necessary swaps are split across two subtrees, and we lose the capability to represent odd numbers in the process. The gain lies in the neutral trees, it holds that $mincost(\alpha\langle \bot : y\rangle, \alpha\langle x : y\rangle) = \frac{y}{2}$ completely *independently of the value* x.

Definition 23 (Multi-number Trees). *For some* $\alpha \in \Sigma$ *and* $k \in \mathbb{N}$ *assume that we have the distinct symbols* $\alpha_1, \ldots, \alpha_k \in \Sigma$. *Then, for all* $x_1, \ldots, x_k \in \mathbb{N} \cup \{\bot\}$, *such that either* $x_i \leq y$ *or* $x_i = \bot$ *for all* $i \in [n]$, *let* $\alpha\langle(x_1, \ldots, x_k) : y\rangle$ *denote the tree*

$$\alpha[\alpha_1\langle x_1 : y\rangle, \ldots, \alpha_k\langle x_k : y\rangle].$$

This means that

$$mincost(\alpha\langle(x_1, \ldots, x_n) : y\rangle, \alpha\langle(x'_1, \ldots, x'_n) : y\rangle) = \sum_{i=1}^{n} |x_i - x'_i|,$$

for all $x_1, x'_1, \ldots, x_n, x'_n, y \in \mathbb{N}$ such that $x_i \leq y$ and $x'_i \leq y$ for all $i \in [n]$.

Now all the building blocks necessary to reduce a swap even-cost assignment problem instance to a tree swap problem instance are ready.

Definition 24 (Reducing SecAP to TSwD). *Let* (M, b) *be an instance of the swap even-cost assignment problem as in Definition 12. We then construct the instance* (t, t', b') *of the tree swap distance problem as follows. Assume that* M *is an* n *by* n *matrix, let* τ *be the largest integer that occurs in* M. *Then let* $b' = b + \frac{n(n-1)\tau}{2}$ *and construct*

$$\begin{aligned} t = \alpha[\, &\beta\langle(M_{1,1}, \ldots, M_{1,n}) : \tau\rangle, \\ &\beta\langle(M_{2,1}, \ldots, M_{2,n}) : \tau\rangle, \\ &\vdots \\ &\beta\langle(M_{n,1}, \ldots, M_{n,n}) : \tau\rangle], \end{aligned}$$

and

$$t' = \alpha[\beta\langle(0, \bot, \bot, \ldots, \bot) : \tau\rangle,\\ \beta\langle(\bot, 0, \bot, \ldots, \bot) : \tau\rangle,\\ \vdots\\ \beta\langle(\bot, \bot, \ldots, \bot, 0) : \tau\rangle],$$

that is, $t' = \alpha[t_1, \ldots, t_n]$ *such that for all* $i \in [n]$ *we have* $t_i = \beta\langle(x_1, \ldots, x_n) : \tau\rangle$ *where* $x_j = \bot$ *for all* $j \neq i$ *and* $x_i = 0$.

The dense notation may make this reduction hard to visualize, let us look at an example.

Example 25. Let (M, b) be an instance of the swap even-cost assignment problem, letting $b = 3$ and

$$M = \begin{bmatrix} 4\,0 \\ 2\,2 \end{bmatrix}.$$

Now we construct the tree swap distance problem instance (t, t', b') by applying the reduction from Definition 24. From M we see that $\tau = 4$, so the budget becomes $b' = 3 + \frac{2(2-1)4}{2} = 7$. The constructed trees are

$$t = \alpha[\beta\langle(4, 0) : 4\rangle, \beta\langle(2, 2) : 4\rangle],\\ t' = \alpha[\beta\langle(0, \bot) : 4\rangle, \beta\langle(\bot, 0) : 4\rangle].$$

To get past the notation the full tree t is shown in Figure 7, and the tree t' (as well as a breakdown of which subtrees correspond to which piece of notation) is shown in Figure 8.

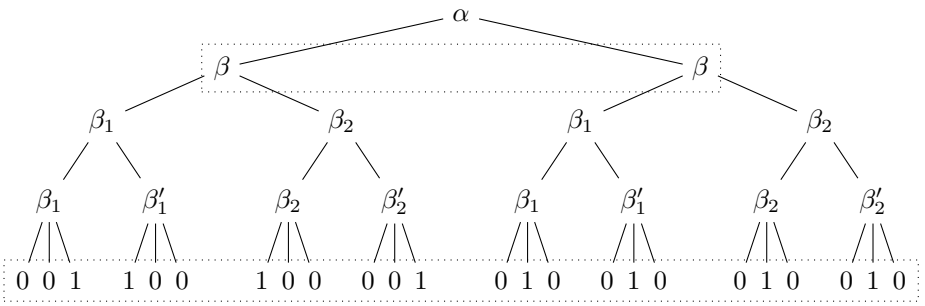


Figure 7: The tree t constructed in the reduction in Example 25. Notice that any solution only needs to perform swaps on the nodes in the dotted rectangles, all other nodes are already in their only possible internal order (compare to t' in Figure 8).

Using these figures it is not hard to see how the solutions to (M, b) and (t, t', b') correspond to each other. (M, b) has a single solution, swapping the two rows (which gives a diagonal sum of 2, for a total cost of 3, which is exactly the budget), making no swap is not an option since the initial diagonal sum is 6, which is over the budget.

The decision to swap the rows in M or not corresponds to the decision whether or not to swap the $\beta\langle\ldots\rangle$-subtrees in t. The reader can easily verify by inspecting Figure 7 and 8 that it takes 10 swaps to move the 0/1 nodes around to match t' if we do not swap the $\beta\langle\ldots\rangle$-subtrees first, which is over the budget (in fact, it is over the budget by the same amount as the initial order of M is for that instance). If the two

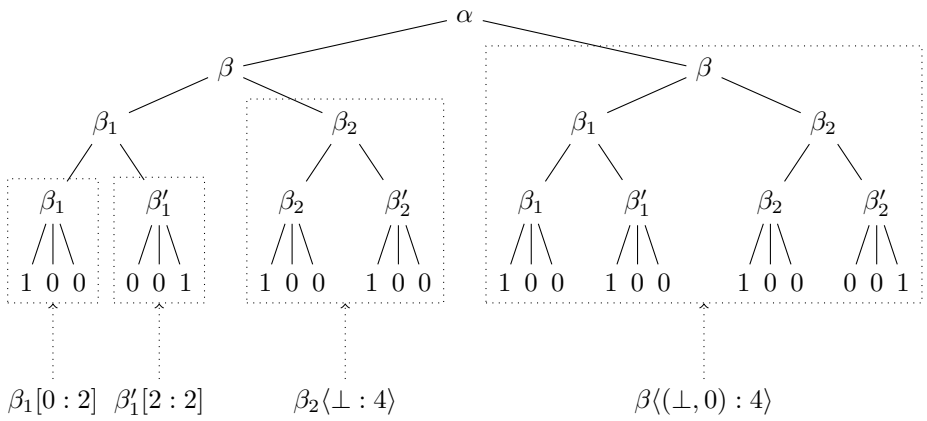


Figure 8: The tree t' constructed in the reduction in Example 25. The dotted arrows shows the notation we use to describe the indicated parts of the tree.

$\beta\langle\dots\rangle$-subtrees are swapped however, we can reorder the 0/1 nodes in the resulting tree in only 6 swaps, for a total cost of 7, exactly the budget b'.

Hopefully the example has clarified the general idea of this reduction, but a proof sketch follows which further illustrates how it functions in the general case.

Lemma 26. *For every swap even-cost assignment problem instance (M, b) and tree swap distance problem instance (t, t', b') constructed from (M, b) by the reduction in Definition 24 it holds that $(t, t', b) \in$ TSwD if and only if $(M, b) \in$ SecAP.*

Proof (Sketch). We reuse the notation of the reduction. First notice that there are only two levels of swapping to consider in t. The immediate subtrees can be reordered since all are of the β multi-number kind, this is the interesting part. In addition the leaves will be swapped to move around the 0/1 sequences that are there to represent numbers, but this is abstracted by our number trees and can only be done in one trivial way once the top-level swaps are decided. The nodes in between are marked with distinct symbols.

Now let us look at the sub-subtrees in t'. There are n^2 of them, organized into n subtrees, each of which represents a row. For each $i \in [n]$ look at position i, i in t', this tree is of the form $\beta_i\langle 0 : \tau\rangle$, whereas for all $i, j \in [n]$ such that $i \neq j$ the subtree at position i, j is of the form $\beta_j\langle \bot : \tau\rangle$. These $n(n-1)$ trees will be matched up with some β_j sub-subtree in t at a constant cost of $\frac{\tau}{2}$ each, incurring a constant and unavoidable cost of $\frac{n(n-1)\tau}{2}$, leaving exactly b of the budget for the remainder.

This leaves the n "diagonal" subtrees of the form $\beta_i\langle 0 : \tau\rangle$ in t'. Assume that W in M moves row i into position j, incurring some swap cost and a diagonal cost of $M_{i,j}$. If we apply W directly to t this would move subtree $\beta\langle M_{i,1}, \dots, M_{i,n}\rangle$ into position to match the tree in t' that contains the zero number tree $\beta_j\langle 0 : \tau\rangle$ in position j. This means that the cost incurred, beyond the already accounted for constant cost associated with the $n-1$ neutral trees will be $\mathit{mincost}(\beta_j\langle M_{i,j} : \tau\rangle, \beta_j\langle 0 : \tau\rangle)$, which is exactly $M_{i,j}$ by the construction of the number trees. So, to recap, applying W at the top level leaves us with the constant cost of $\frac{n(n-1)\tau}{2}$ plus $|W|$ plus $M_{i,j}$ for each row moved from position i to position j by W. Which is exactly the same cost that applying W in M incurs plus $\frac{n(n-1)\tau}{2}$, and since $b' = b + \frac{n(n-1)\tau}{2}$ this makes the problem instances equivalent. We did the argument starting from W, but we

can trivially extract the swaps which deal with the immediate subtrees in t from a solution to (t, t', b'), making the other direction very straightforward. □

Corollary 27. *The tree swap distance problem is strongly NP-complete.*

As before the problem being in NP is trivial since the swap sequence never needs to be longer than n^2 so we may guess it. The reduction being polynomial is not hard to see, though the details become somewhat lengthy. There are on the order of $\mathcal{O}(\tau n^2)$ nodes in the trees, but SecAP is strongly NP-complete so this unary representation is not problematic.

7 Conclusion

Treating a problem where the only conclusion is negative, the problem being intractable, is never quite the ideal outcome. On the other hand it was already known that tree edit distance with subtree movement is problematic, and the efforts to integrate limited forms of swaps have been ongoing for some time. As such it is useful to establish that swaps are *inherently* problematic in trees. This hints that better results may be achieved if one considers simpler measures, such as linear distance, where all subtrees are reordered simultaneously and the cost of moving a subtree from position i to position j is exactly $|i - j|$ independent of whether the trees in between are moved. This would allow the Hungarian algorithm [6] to be leveraged in the tree case, giving a polynomial algorithm.

The problem itself may also be useful for complexity analysis of other swap problems, since it is at its core very simple both to explain and intuitively understand.

Hopefully this rather fundamental problem being proven NP-complete will also serve as a useful stepping stone for other complexity-theoretical work.

References

1. D. T. Barnard, G. Clarke, and N. Duncan: *Tree-to-tree correction for document trees*, Tech. Rep. 1995-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1995.
2. P. Bille: *A survey on tree edit distance and related problems.* Theor. Comput. Sci., 337(1–3) 2005, pp. 217–239.
3. F. J. Damerau: *A technique for computer detection and correction of spelling errors.* Commun. ACM, 7(3) 1964, pp. 171–176.
4. M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
5. P. N. Klein: *Computing the edit-distance between unrooted ordered trees*, in In Proceedings of the 6th annual European Symposium on Algorithms (ESA, Springer-Verlag, 1998, pp. 91–102.
6. H. W. Kuhn: *The hungarian method for the assignment problem.* Naval Research Logistics Quarterly, 2(1–2) 1955, pp. 83–97.
7. V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions, and reversals.* Soviet Physics Doklady, 10(8) 1966, pp. 707–710.
8. S. M. Selkow: *The tree-to-tree editing problem.* Inf. Process. Lett., 6(6) 1977, pp. 184–186.
9. K.-C. Tai: *The tree-to-tree correction problem.* J. ACM, 26 July 1979, pp. 422–433.
10. R. A. Wagner: *On the complexity of the extended string-to-string correction problem*, in STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing, New York, NY, USA, 1975, ACM, pp. 218–223.
11. R. A. Wagner and R. Lowrance: *An extension of the string-to-string correction problem.* J. ACM, 22(2) 1975, pp. 177–183.

12. K. Zhang and D. Shasha: *Simple fast algorithms for the editing distance between trees and related problems.* SIAM J. Comput., 18(6) 1989, pp. 1245–1262.
13. K. Zhang, R. Statman, and D. Shasha: *On the editing distance between unordered labeled trees.* Information Processing Letters, 42(3) 1992, pp. 133–139.

A Parameterized Formulation for the Maximum Number of Runs Problem*

Andrew Baker[1], Antoine Deza[1,2], and Frantisek Franek[1]

[1] Advanced Optimization Laboratory
Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
[2] Equipe Combinatoire et Optimisation
Université Pierre et Marie Curie, Paris, France
{bakerar2,deza,franek}@mcmaster.ca
http://optlab.cas.mcmaster.ca/

Abstract. A parameterized approach to the problem of the maximum number of runs in a string was introduced by Deza and Franek. In the approach referred to as the *d-step approach*, in addition to the usual parameter the length of the string, the size of the string's alphabet is considered. The behaviour of the function $\rho_d(n)$, the maximum number of runs over all strings of length n with exactly d distinct symbols, can be handily expressed in the terms of properties of a table referred to as the $(d, n-d)$ *table* in which $\rho_d(n)$ is the entry at the dth row and $(n-d)$th column. The approach leads to a conjectured upper bound $\rho_d(n) \leq n-d$ for $2 \leq d \leq n$. The parameterized formulation shows that the maximum within any column of the $(d, n-d)$ table is achieved on the main diagonal, i.e. for $n = 2d$, and motivates the investigation of the structural properties of the run-maximal strings of length n bounded by a constant times the size of the alphabet d. We show that $\rho_d(n) = \rho_{n-d}(2n-2d)$ for $2 \leq d \leq n < 2d$, $\rho_d(2d) \leq \rho_{d-1}(2d-1)+1$ for $d \geq 3$, $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2) = \rho_{d-3}(2d-3)$ for $d \geq 5$, and $\{\rho_d(n) \leq n-d \text{ for } 2 \leq d \leq n\} \Leftrightarrow \{\rho_d(9d) \leq 8d \text{ for } d \geq 2\}$. The results allow for an efficient computational verification of entries in the $(d, n-d)$ table for higher values of n and point to a plausible way of either proving the maximum number of runs conjecture by showing that possible counter-examples on the main diagonal would exhibit an impossible structure, or to discover an unexpected counter-example on the main diagonal of the $(d, n-d)$ table. This approach provides a purely analytical proof of $\rho_d(2d) = d$ for $d \leq 15$ and, using the computational results of $\rho_2(d+2)$ for $d = 16, \ldots, 23$, a proof of $\rho_d(2d) = d$ for $d \leq 23$.

Keywords: string, runs, maximum number of runs, parameterized approach, $(d, n-d)$ table

1 Introduction

The problem of determining the maximum number of runs in a string has a rich history and many researchers have contributed to the effort. The notion of a run is due to Main [17], the term itself was introduced in [13]. Kolpakov and Kucherov [14,15] showed that the function $\rho(n)$, the maximum number of runs over all strings of length n, is linear. Several papers dealt with lower and upper bounds or expected values for $\rho(n)$, see [2,3,4,8,10,11,12,18,19,20,21,23] and references therein.

The counting estimates leading to the best upper bounds [3,4] rely heavily on a computational approach and seem to reach a point where it gets highly challenging, bordering intractability, to verify the results or make further progress. A few

* This work was supported by the *Natural Sciences and Engineering Research Council of Canada* and *MITACS*, and by the *Canada Research Chair program*, and made possible by the facilities of the *Shared Hierarchical Academic Research Computing Network* (http://www.sharcnet.ca/).

Andrew Baker, Antoine Deza, Frantisek Franek: A Parameterized Formulation for the Maximum Number of Runs Problem, pp. 74–82.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

researchers tried a structural approach. Rytter's three neighbour lemma can be considered one such attempt, along with the ongoing work of W. Smyth *et al.* [6,7,16,22].

A parameterized approach to the investigation of the structural aspects of run-maximal strings was introduced by Deza and Franek [5]. In addition to considering the length of the string they introduced the parameter d giving the function $\rho_d(n)$, the maximum number of runs over all strings of length n with exactly d distinct symbols. These values are presented in what we refer to as $(d, n-d)$ table because the value of $\rho_d(n)$ is the entry at the row d and the column $n-d$, rather than the more usual row d and column n. It is just a different presentation of the values $\rho_d(n)$, but it points to some interesting aspects and possible recurrences of the function $\rho_d(n)$. Based on the results presented in this paper and elsewhere, we believe that the table captures the essence of the behaviour of the function $\rho_d(n)$.

In Table 1, the computed entries for the first 10 rows and the first 10 columns are presented (the entries were computed using the FJW algorithm, see [9]) and the other entries are indicated. Several properties of the table were presented in [5], the most important being the fact that $\rho_d(n) \leq n-d$ for $2 \leq d \leq n$ is equivalent with $\rho_d(2d) \leq d$ for $d \geq 2$. In other words, if the diagonal obeys the upper bound $n-d$, so do all the entries in the table everywhere. Though in the related literature, the *maximum number of runs conjecture* – or simply *runs conjecture* – refers to the hypothesis that $\rho(n) \leq n$, in this paper we will take it to be $\rho_d(n) \leq n-d$.

We discuss several additional properties of the $(d, n-d)$ table, the behaviour of the function $\rho_d(n)$ on or nearby the main diagonal, and discuss some structural properties of run-maximal strings on the main diagonal. The results allow for the extension of computational verification of the maximum number of runs conjecture to higher values of n and also indicate a viable approach to an analytical investigation of the conjecture by either showing a possible counter-example to the conjecture would have to exhibit an impossible structure, or exhibiting a counter-example on the main diagonal of the $(d, n-d)$ table and direct calculation of entries for smaller columns.

Let us remark, that although we believe with the majority of the researchers in the field that the conjecture is true and hence view the d-step approach as a possible tool to prove it, if a counter-example exists, one must be on the main diagonal and we believe it will easier to find there as the run-maximal strings of length being twice the size of the alphabet seem to exhibit a richer structure than general run-maximal strings. A counter-example would be in essence a quite striking result.

2 Notation and Preliminaries

Throughout this paper, we refer to k-tuples: a symbol which occurs exactly k times in the string under consideration. Specially named k-tuples are the *singleton* (1-tuple), *pair* (2-tuple), *triple* (3-tuple), *quadruple* (4-tuple), and *quintuple* (5-tuple).

Definition 1. *A safe position in a string $\boldsymbol{x}$ is one which, when removed from $\boldsymbol{x}$, does not result in two runs being merged into one in the resulting new string.*

A safe position does not ensure that the number of runs will not change when that position is removed, only that no runs will be lost through being merged; runs may still be destroyed by having an essential symbol removed. Safe positions are important in that they may be removed from a string while only affecting the runs which contain them. For an illustration consider $\boldsymbol{x}[1..9] = abab\boldsymbol{a}abab$. Position 5 (in bold) is not safe,

d \ $n-d$	1	2	3	4	5	6	7	8	9	10	11	12
1	**1**	*1*	1	1	1	1	1	1	1	1	$\rho_1(12)$	.
2	1	**2**	*2*	3	4	5	5	6	7	8	$\rho_2(13)$	.
3	1	2	**3**	*3*	4	5	6	6	7	8	$\rho_3(14)$	.
4	1	2	3	**4**	*4*	5	6	7	7	8	$\rho_4(15)$	.
5	1	2	3	4	**5**	*5*	6	7	8	8	$\rho_5(16)$	.
6	1	2	3	4	5	**6**	*6*	7	8	9	$\rho_6(17)$	.
7	1	2	3	4	5	6	**7**	*7*	8	9	$\rho_7(18)$	.
8	1	2	3	4	5	6	7	**8**	*8*	9	$\rho_8(19)$	.
9	1	2	3	4	5	6	7	8	**9**	*9*	$\rho_9(20)$	.
10	1	2	3	4	5	6	7	8	9	**10**	$\rho_{10}(21)$	.
11	.	.	.	.	.	.	.	.	$\rho_{11}(20)$	$\rho_{11}(21)$	$\boldsymbol{\rho_{11}(22)}$	.
12	.	.	.	.	.	.	.	.	.	.	.	.

Table 1. Values for $\rho_d(n)$ with $1 \leq d \leq 10$ and $1 \leq n-d \leq 10$. For more values, see [1]. The **bold** entries denote the **main diagonal** referred in the text, while the entries in *italics* denote the second diagonal.

for if we remove it, the run *abab* starting at position 1 and the run *abab* starting at position 6 will be merged together: *abababab*. Similarly position 6 is not safe, as its removal would merger two runs. On the other hand, position 4 is safe. If we remove it, we get *abaaabab* which destroys 1 run *abab*, however it does not cause any two runs to merge.

When the position of a symbol is unambiguous, we may thus refer to a *safe symbol* rather than to its position – for instance we can talk about a safe singleton, or about the first member of a pair being safe, etc.

At various points we will need to relabel all occurrences of a symbol in a string or substring. Let $\boldsymbol{x}^a_b$ denote the string $\boldsymbol{x}$, in which all occurrences of a are replaced by b, and vice versa. $S_d(n)$ refers to the set of strings of length n with exactly d distinct symbols. For a string $\boldsymbol{x}$, $\mathcal{A}(\boldsymbol{x})$ denotes the alphabet of $\boldsymbol{x}$, while $r(\boldsymbol{x})$ denotes the number of runs of $\boldsymbol{x}$.

Lemma 2. *There exists a run-maximal string in $S_d(n)$ with no unsafe singletons for $2 \leq d \leq n$.*

Proof. Let $\boldsymbol{x}$ be a run-maximal string in $S_d(n)$. We will show that one of the following conditions must hold:

- (i) $\boldsymbol{x}$ has no singletons, or
- (ii) $\boldsymbol{x}$ has exactly one singleton which is safe, or
- (iii) $\boldsymbol{x}$ has exactly one singleton which is unsafe, and there exists another run-maximal string $\boldsymbol{x'} \in S_d(n)$ where $\boldsymbol{x'}$ has no unsafe singletons, or
- (iv) $\boldsymbol{x}$ has more than one singleton, all of which are safe.

Let $\boldsymbol{x}$ have some unsafe singletons.

First, consider the case that $\boldsymbol{x}$ has exactly one singleton, C, which is unsafe: $\boldsymbol{x} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}C a\boldsymbol{v}a\boldsymbol{v}\boldsymbol{w}$, where $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ are (possibly empty) strings, and $a \in \mathcal{A}(\boldsymbol{x}) - \{C\}$. Let $\boldsymbol{x'} = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}(Ca\boldsymbol{v}a\boldsymbol{v}\boldsymbol{w})^a_C = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}(aC\boldsymbol{v}^a_C C\boldsymbol{v}^a_C\boldsymbol{w}^a_C) = \boldsymbol{u}a\boldsymbol{v}a\boldsymbol{v}aC\tilde{\boldsymbol{v}}C\tilde{\boldsymbol{v}}\tilde{\boldsymbol{w}}$. Clearly, $\boldsymbol{x'} \in S_d(n)$,$r(\boldsymbol{x'}) \geq r(\boldsymbol{x})$, so $\boldsymbol{x'}$ is run-maximal and has no singletons.

Next, consider the case that $\boldsymbol{x}$ has at least two singletons C, D, of which one is unsafe, C. Without loss of generality, we can assume C occurs before D : $\boldsymbol{x} =$

$\boldsymbol{uavavCavavwDz}$, where $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{z}$ are (possibly empty) strings and $a \in \mathcal{A}(\boldsymbol{x}) - \{C, D\}$. Let $\boldsymbol{x_1} = \boldsymbol{uavav}(C\boldsymbol{avavwD}\boldsymbol{z})^a_C = \boldsymbol{uavavaC\tilde{v}C\tilde{v}\tilde{w}D\tilde{z}}$. Clearly, $\boldsymbol{x_1} \in S_d(n)$ and $r(\boldsymbol{x_1}) \geq r(\boldsymbol{x})$. We then modify $\boldsymbol{x_1}$ by removing the safe symbol a immediately to the left of the first occurrence of C, yielding $\boldsymbol{x_2}$. Finally, we add a second copy of D adjacent to the original D, restoring the original length: $\boldsymbol{x_3} = \boldsymbol{uavavC\tilde{v}C\tilde{v}\tilde{w}DD\tilde{z}}$. $\boldsymbol{x_3} \in S_d(n)$ and $r(\boldsymbol{x_3}) > r(\boldsymbol{x_2}) \geq r(\boldsymbol{x_1}) \geq r(\boldsymbol{x})$, which contradicts the run-maximality of $\boldsymbol{x}$. □

Lemma 3 is a simple observation that for a position to be unsafe, a symbol must occur twice to the left and twice to the right of that position.

Lemma 3. *If a string $\boldsymbol{x}$ consists only of singletons, pairs, and triples, then every position is safe.*

A corollary of Lemma 3 is that the maximum number of runs in a string with only singletons, pairs, and triples is limited by the number of pairs and triples. Specifically, $r(\boldsymbol{x}) = \#pairs + \lfloor \frac{3}{2} \#triples \rfloor$. This is because a pair can only be involved in a single run, and a triple can be involved in at most 2 runs. The densest structure achievable is through overlapping triples in the pattern *aababb*, which has 3 runs for every two triples. The pairs, meanwhile, are maximized through adjacent copies.

3 Run-maximal strings below the main diagonal and in the immediate neighbourhood above

We first remark that every value below the main diagonal in the $(d, n-d)$ table is equal to the value on the main diagonal directly above it. In other words, the values on and below the main diagonal in a column are constant.

Proposition 4. *We have $\rho_d(n) = \rho_{n-d}(2n-2d)$ for $2 \leq d \leq n < 2d$.*

Proof. Consider a run-maximal string $\boldsymbol{x} \in S_d(n)$, where $2 \leq d \leq n < 2d$. By Lemma 2, we can assume $\boldsymbol{x}$ has no unsafe singletons. Since $n < 2d$, $\boldsymbol{x}$ must have a singleton, and hence it must be safe. We can remove this safe singleton, yielding a new string $\boldsymbol{y} \in S_{d-1}(n-1)$ and so $\rho_d(n) = r(\boldsymbol{x}) = r(\boldsymbol{y}) \leq \rho_{d-1}(n-1)$. Recall the following inequality noted in [5]:

$$\rho_d(n) \leq \rho_{d+1}(n+1) \text{ for } 2 \leq d \leq n \tag{1}$$

Thus, $\rho_{d-1}(n-1) = \rho_d(n)$. □

Proposition 4 together with inequality (1) gives the following equivalency noted in [5]: $\{\rho_d(n) \leq n-d \text{ for } 2 \leq d \leq n\} \Leftrightarrow \{\rho_d(2d) \leq d \text{ for } 2 \leq d\}$.

If there is a counter-example to the conjectured upper bound, then the main diagonal must contain a counter-example. If it falls under the main diagonal, then by Proposition 4 there must be a counter-example on the main diagonal – i.e. it can be *pushed up*, and if it falls above the main diagonal, by the inequality (1), there must be a counter-example on the main diagonal – i.e. the counter-example can be *pushed down*.

We extend Proposition 4 to bound the behaviour of the entries in the immediate neighbourhood above the main diagonal in the $(d, n-d)$ table. Proposition 5 establishes that the difference between the entry on the main diagonal and the entry immediately above it is at most 1. In addition, the difference is 1 if and only if every run-maximal string in $S_d(2d)$ consists entirely of pairs; otherwise, the difference is 0.

Proposition 5. *We have* $\rho_d(2d) \leq \rho_{d-1}(2d-1)+1$ *for* $d \geq 3$.

Proof. Let $\boldsymbol{x} \in S_d(2d)$ be a run-maximal string with no unsafe singletons (by Lemma 2). If $\boldsymbol{x}$ does not have a singleton, then it consists entirely of pairs. It is clear that the pairs must be adjacent and that $r(\boldsymbol{x}) = d$ and so $\boldsymbol{x} = aabbcc\ldots$ Removing the first a and renaming the second to b, $\boldsymbol{y} = bbbcc\ldots \in S_{d-1}(2d-1)$ and $\rho_{d-1}(2d-1) \geq r(\boldsymbol{y}) = r(\boldsymbol{x}) - 1 = \rho_d(2d) - 1$. If $\boldsymbol{x}$ has a singleton, since it is safe we can remove it forming a string $\boldsymbol{y} \in S_{d-1}(2d-1)$ so that $\rho_{d-1}(2d-1) \geq r(\boldsymbol{y}) = r(\boldsymbol{x}) = \rho_d(2d)$, and so $\rho_{d-1}(2d-1) = \rho_d(2d)$. □

We have seen that the gap between the first entry above the diagonal and the diagonal entry is at most 1. Proposition 6 establishes that the three entries just above the diagonal are identical.

Proposition 6. *We have* $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2) = \rho_{d-3}(2d-3)$ *for* $d \geq 5$.

Proof. Let $\boldsymbol{x}$ be a run-maximal string in $S_{d-1}(2d-1)$. By Lemma 2 we can assume that either it has a safe singleton or no singletons at all. In the former case, we can remove the safe singleton obtaining $\boldsymbol{y} \in S_{d-2}(2d-2)$ so that $\rho_{d-2}(2d-2) \geq r(\boldsymbol{y}) \geq r(\boldsymbol{x}) = \rho_{d-1}(2d-1)$, and so $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2)$. In the latter case, $\boldsymbol{x}$ consists of pairs and one triple, and thus, by Lemma 3, all positions are safe. Therefore, we can move all the pairs to the end of the string, yielding $\boldsymbol{y} = aaabbcc\ldots \in S_{d-1}(2d-1)$ and by removing the first a and renaming the remaining as to cs, $\boldsymbol{z} = ccbbcc\ldots \in S_{d-2}(2d-2)$. It follows that $\rho_{d-2}(2d-2) \geq r(\boldsymbol{z}) = r(\boldsymbol{y}) = r(\boldsymbol{x}) = \rho_{d-1}(2d-1)$, and so $\rho_{d-1}(2d-1) = \rho_{d-2}(2d-2)$.

Let $\boldsymbol{x}$ be now a run-maximal string in $S_{d-2}(2d-2)$. Again, if $\boldsymbol{x}$ has a singleton, we can assume by Lemma 2 it is safe and form $\boldsymbol{y}$ by removing the singleton. $\boldsymbol{y} \in S_{d-3}(2d-3)$ and $\rho_{d-3}(2d-3) \geq r(\boldsymbol{y}) \geq r(\boldsymbol{x}) = \rho_{d-2}(2d-2)$. If $\boldsymbol{x}$ does not have a singleton, then $r(\boldsymbol{x}) = d-1$. To see this, consider the two cases:

(*i*) $\boldsymbol{x}$ consists of two triples and several pairs. The most runs which may be obtained in such a string, after grouping the pairs at the end of the string, is through the arrangement $aababbccddee\ldots$ In this case, there are $d-4$ runs from the pairs, and 3 runs from the triples, giving a total of $d-1$ runs.

(*ii*) $\boldsymbol{x}$ consists of a quadruple and several pairs. The most runs which may be obtained in this case is from a string with either the structure $aabbaaccddee\ldots$, or $aabaabccddee\ldots$, where all the pairs have been grouped at the end, except for the pair of bs which is used to break up the quadruple. In both cases, there are $d-4$ runs involving characters c onward, and three runs involving the characters a and b, again giving a total of $d-1$ runs.

Now consider a string $\boldsymbol{y} = aabbaabbcdee\ldots \in S_{d-2}(2d-2)$, which has two quadruples (of as and bs), two singletons (c and d), and several pairs ($e\ldots$). This string has $d-6$ runs from the pairs ee onward, and 5 runs from the characters a and b, giving a total of $d-1$ runs, i.e. $r(\boldsymbol{x}) = r(\boldsymbol{y})$. The singleton c in $\boldsymbol{y}$ being clearly safe, we can remove it and continue as in the previous case. □

Remark 7 below providing a lower bound for the first 4 entries above the main diagonal of the $(d, n-d)$ table, is a corollary of the inequality $\rho_{d+s}(n+2s) \geq \rho_d(n)+s$, noted in [5], applied to $\rho_2(k) = k-3$ for $k = 5, 6, 7$ and 8.

Remark 7. We have $\rho_{d-k}(2d-k) \geq d-1$ for $k = 1, 2, 3$ and 4 and $d \geq 6$.

4 Structural properties of run-maximal strings on the main diagonal

We explore structural properties of the run-maximal strings on the main diagonal. These results yield properties for run-maximal strings that have their length bounded by nine times the number of distinct symbols they contain. We can thus shift the critical region of the $(d, n-d)$ table as summarized in the Theorem 8, the proof for which can be found at the end of this section.

Theorem 8. *We have* $\{\rho_d(n) \leq n-d \text{ for } 2 \leq d \leq n\} \Leftrightarrow \{\rho_d(9d) \leq 8d \text{ for } d \geq 2\}$.

Proposition 9 describes useful structural properties of run-maximal strings on the main diagonal. The proof of the proposition relies on a few lemmas that will be mostly presented without their entire proofs, just a few examples will be given to illustrate the method. They all deal with the same basic scenario: assuming we know that the table obeys the conjecture for all columns to the left of column d, which is the first *unknown* column, we investigate the run-maximal strings of $S_d(2d)$.

Proposition 9. *Let* $\rho_{d'}(2d') \leq d'$ *for* $2 \leq d' < d$. *Let* $\boldsymbol{x}$ *be a run-maximal string in* $S_d(2d)$. *Either* $r(\boldsymbol{x}) = \rho_d(2d) = d$ *or* $\boldsymbol{x}$ *has at least* $\lceil \frac{7d}{8} \rceil$ *singletons, and no symbol occurs exactly 2, 3, . . . , 8 times in* $\boldsymbol{x}$.

Proof. The proof that each symbol must be a singleton or occur at least 9 times is a direct result of the lemmas which make up the remainder of this section. Then, let $\boldsymbol{x} \in S_d(2d)$ be run-maximal, m_1 denote the number of singletons, and m_2 the number of non-singleton symbols of $\boldsymbol{x}$. We have $m_1 + 9m_2 \leq 2d$ and $m_1 + m_2 = d$, which implies that $m_2 \leq d/8$ and hence $m_1 \geq \lceil 7d/8 \rceil$. □

Proposition 9 provides a purely analytical proof that $\rho_d(2d) = d$ for $d \leq 15$, and using the computation of $\rho_2(d+2)$ for $d = 16, \ldots, 23$, that $\rho_d(2d) = d$ for $d \leq 23$.

Corollary 10. *We have* $\rho_d(2d) = d$ *for* $d \leq 23$ *and* $\rho_d(n) \leq n-d$ *for* $n-d \leq 23$.

Proof. Assume that run-maximal $\boldsymbol{x} \in S_d(2d)$ satisfies $r(\boldsymbol{x}) = \rho_d(2d) > d$. By Proposition 9, $\boldsymbol{x}$ consists only of singleton for $2 \leq d \leq 6$, $r(\boldsymbol{x}) = \rho_1(d+1) = 1$ for $8 \leq d \leq 15$, and $d < r(\boldsymbol{x}) = \rho_2(d+2)$ for $16 \leq d \leq 23$, which are impossible. □

In Lemmas 11,12, and 13 we assume that for $2 \leq d' < d$, the conjecture holds, i.e. $\rho_{d'}(2d') \leq d'$. Note that it is equivalent to $\rho_{d'}(n') \leq n' - d'$ for $2 \leq d' \leq n'$ when $n' - d' < d$. We consider a run-maximal string $\boldsymbol{x} \in S_d(2d)$ containing a k-tuple. We show that either the string $\boldsymbol{x}$ obeys the conjectured upper bound, or can be manipulated to obtain a new string $\boldsymbol{y}$ with a larger alphabet of the same or shorter length. We ensure that the manipulation process does not destroy more runs than the amount the alphabet is increased or the length decreased. This allows us to estimate the number of runs in $\boldsymbol{y}$ based on the values in the table for some $d' < d$. In essence, we manipulate a string from column d to a string from some column $d' < d$ while monitoring the number of runs.

Lemma 11. *Let* $\rho_{d'}(2d') \leq d'$ *for* $2 \leq d' < d$. *Let* $\boldsymbol{x} \in S_d(2d)$ *be run-maximal. Either* $r(\boldsymbol{x}) = \rho_d(2d) = d$ *or* $\boldsymbol{x}$ *does not contain a pair.*

Proof. Assume that $\boldsymbol{x}$ does not obey the conjectured upper bound and so $r(\boldsymbol{x}) > d$. Let us assume that $\boldsymbol{x}$ contains a pair of C's and so $\boldsymbol{x} = \boldsymbol{u}C\boldsymbol{v}C\boldsymbol{w}$. Change the first occurrence of C to a new symbol $D \notin \mathcal{A}(\boldsymbol{x})$ to obtain $\boldsymbol{y} = \boldsymbol{u}D\boldsymbol{v}C\boldsymbol{w}$. Since a pair can be in at most one run (see for instance [5]), we destroyed at most one run and increased the alphabet size by one, so $d - 1 \geq \rho_{d+1}(2d) \geq r(\boldsymbol{y}) \geq r(\boldsymbol{x}) - 1$. It follows that $d \geq r(\boldsymbol{x})$, a contradiction with our earlier assumption. □

Lemma 12. *Let $\rho_{d'}(2d') \leq d'$ for $2 \leq d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) = d$ or $\boldsymbol{x}$ does not contain a triple.*

Proof. (A sketch) If $\boldsymbol{x}$ does not obey the conjecture and has a triple of C's, the triple can be involved in at most two runs. We change the first two occurrences of C to new symbols D and E obtaining $\boldsymbol{y} \in S_{d+2}(2d)$. This destroys at most two runs while increasing the size of the alphabet by 2, a contradiction with our assumption. □

For k-tuples of higher degree, $4 \leq k \leq 8$, the approach is very similar, but since such a k-tuple can be in multiple runs, the discussion of cases become more complex and thus we summarize all these results without a proof in Lemma 13.

Lemma 13. *Let $\rho_{d'}(2d') \leq d'$ for $2 \leq d' < d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. Either $r(\boldsymbol{x}) = \rho_d(2d) = d$ or $\boldsymbol{x}$ does not contain a k-tuple, $4 \leq k \leq 8$.*

While the previous lemmas were provided for entries on the main diagonal, the result can be generalized to any entry in column $n - d$ where $\rho_{d'}(n') \leq n' - d'$ for $n' - d' < n - d$. Either $\rho_d(n) \leq n - d$, or no run-maximal $\boldsymbol{x} \in S_d(n)$ has a pair, triple, ..., 8-tuple. The induction hypothesis only requires that all entries to the left of the *unknown* column satisfy the conjecture; there is no restriction within the *unknown* column.

Having proven Proposition 9, we can present the proof of Theorem 8:

Proof. The proof follows directly from Proposition 9. If the conjecture does not hold, let d be the first column for which $\rho_d(2d) > d$. Let $\boldsymbol{x} \in S_d(2d)$ be run-maximal. By Proposition 9, $\boldsymbol{x}$ has at least $k = \lceil \frac{7d}{8} \rceil$ singletons, and by Lemma 2 they must all be safe. Let us form $\boldsymbol{y}$ by removing all these safe singletons. This gives a string $\boldsymbol{y} \in S_{d-k}(2d - k)$ violating the conjecture, i.e. $r(\boldsymbol{y}) > d$. $d' = d - k = \frac{d}{8}$ and $d = 8d'$ and $2d - k = 9d'$. Thus we found a $\boldsymbol{y} \in S_{d'}(9d')$ such that $r(\boldsymbol{y}) > 8d'$. □

When investigating a single column, the first counter-example in the column cannot have a singleton, as otherwise the counter-example could be *pushed up*. Nor, by Proposition 9, can it contain a k-tuple for $2 \leq k \leq 8$. Theorem 8 together with these facts give a simplified way to computationally *verify* that the whole column d satisfies the conjecture: *show that there are no counter-examples for $2 \leq d' \leq \frac{d}{8}$, and only strings with no k-tuples, $1 \leq k \leq 8$, need to be considered when looking for the counter-examples.*

5 Conclusion

The properties presented in this paper constrain the behaviour of the function $\rho_d(n)$ as presented in the $(d, n - d)$ table below the main diagonal and in an immediate neighbourhood above the main diagonal. One of the main contributions lies in the characterization of structural properties of the run-maximal strings on the main diagonal, giving yet another property equivalent with the maximum number of runs

conjecture. Not only do these results provide a faster way to computationally check the validity of the conjecture for greater lengths, they indicate a possible way to prove the conjecture along the ideas presented in Proposition 9 and its proof: a first counter-example on the main diagonal could not possibly have a k-tuple for any conceivable k. We were able to carry the reasoning up to $k = 8$, but these proofs are not easy to scale up as the combinatorial complexity increases. The hope and motivation for further research along these lines is that there is a common thread among all these various proofs that may lead to a uniform method ruling out all the k-tuples and thus proving the conjecture, or to exhibit an unexpected counter-example on the main diagonal of the $(d, n-d)$ table.

References

1. A. Baker, A. Deza, and F. Franek: *Run-maximal strings.* website, 2011, `http://optlab.mcmaster.ca/~bakerar2/research/runmax/index.html`.
2. M. Crochemore and L. Ilie: *Maximal repetitions in strings.* Journal of Computer and System Sciences, 74(5) 2008, pp. 796–807.
3. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture.* Lecture Notes in Computer Science, 5029 2008, pp. 290–302.
4. M. Crochemore, L. Ilie, and L. Tinta: *The "runs" conjecture.* website, 2011, `http://www.csd.uwo.ca/faculty/ilie/runs.html`.
5. A. Deza and F. Franek: *A d-step analogue for runs on strings*, AdvOL-Report 2010/02, McMaster University, 2010.
6. K. Fan, S. J. Puglisi, W. F. Smyth, and A. Turpin: *A new periodicity lemma.* SIAM Journal on Discrete Mathematics, 20(3) 2006, pp. 656–668.
7. K. Fan, W. F. Smyth, and R. J. Simpson: *A new periodicity lemma.* Lecture Notes in Computer Science, 3537 2005, pp. 257–265.
8. F. Franek and J. Holub: *A different proof of Crochemore-Ilie lemma concerning microruns*, in London Algorithmics 2008: Theory and Practice, College Publications, London, UK, 2009, pp. 1–9.
9. F. Franek, M. Jiang, and C. Weng: *An improved version of the runs algorithm based on Crochemore's partitioning algorithm*, AdvOL Report 2011/03, Advanced Optimization Laboratory, Dept. of Comp. and Software, McMaster University, 2011.
10. F. Franek, R. Simpson, and W. Smyth: *The maximum number of runs in a string*, in Proceedings of 14th Australasian Workshop on Combinatorial Algorithms AWOCA 2003, Seoul National University, Seoul, Korea, 2008.
11. F. Franek and Q. Yang: *An asymptotic lower bound for the maximal number of runs in a string.* International Journal of Foundations of Computer Science, 19(1) 2008, pp. 195–203.
12. M. Giraud: *Not so many runs in strings*, in LATA 2008, Tarragona, Spain, 2008.
13. C. S. Iliopoulos, D. Moore, and W. F. Smyth: *A characterization of the squares in a Fibonacci string.* Theoretical Computer Science, 172 1997, pp. 281–291.
14. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 596–604.
15. R. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in Proc. 12th Intl. Symp. on Fund. of Comp. Sci. 1999, vol. 1684, 1999, pp. 374–385.
16. E. Kopylov and W. F. Smyth: *The three squares lemma revisited.* to appear.
17. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Mathematics, 25 1989, pp. 145–153.
18. W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara: *A series of run-rich strings.* Lecture Notes in Computer Science, 5457 2009, pp. 578–587.
19. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proceedings of PSC 2008, Czech Technical University, Prague, Czech Republic, 2008, pp. 140–145.
20. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *Lower bounds for the maximum number of runs in a string.* website, 2011, `http://www.shino.ecei.tohoku.ac.jp/runs/`.

21. S. J. Puglisi, R. J. Simpson, and W. F. Smyth: *How many runs can a string contain?* Theoretical Computer Science, 401 2008, pp. 165–171.
22. S. J. Puglisi, W. F. Smyth, and A. Turpin: *Some restrictions on periodicity in strings*, in Proceedings of the 16th Australasian Workshop on Combinatorial Algorithms, 2005, pp. 415–428.
23. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound.* Lecture Notes in Computer Science, 3884 2006, pp. 184–195.

Finding Long and Multiple Repeats with Edit Distance

Maria Federico[1], Pierre Peterlongo[2], Nadia Pisanti[3], and Marie-France Sagot[4,5]

[1] Dipartimento di Ingegneria dell'Informazione, University of Modena and Reggio Emilia, Italy
[2] INRIA Rennes – Bretagne Atlantique, EPI Symbiose, Rennes, France
[3] Dipartimento di Informatica, University of Pisa, Italy
[4] Laboratoire de Biométrie et Biologie Evolutive, University of Lyon 1, France
[5] INRIA Rhône-Alpes, France

Abstract. We present a tool for detecting long similar fragments that occur two or more times in a set of biological sequences. The problem has interesting applications in the analysis of biological sequences and their correlation, and becomes computationally challenging when a certain non negligible number of insertions, deletions and substitutions are allowed. For this reason exact exhaustive methods are hardly of practical use. In this paper we introduce a tool, FilmRed, that performs this task, and that manages instances whose size and parameters combination cannot be handled by any existing tool. This is achieved by using a filter as a preprocessing step, and by using the information that the filter has gathered also in the successive inference phase. To the best of our knowledge, FilmRed is the first *ab initio* tool that can deal with repeats occurring possibly several times, that have length of hundreds or thousands bases, and whose occurrences may differ in even more than 10 % of their positions in terms of substitutions and indels.

Keywords: long repeats, multiple repeats, LTR, transposable elements, edit distance

1 Introduction

Genomes are made of an astonishing amount of repeated fragments, in particular in complex organisms as eukaryotes. These repeats are approximate replications of portions of genomes having different ranges and characteristics depending on their origin and function. As for satellites, this can be tandem repeats of few hundred base pairs, segmental duplications of length at least one thousand base pairs and some type of transposons issued from the *copy and paste* process (retrotransposons). For long time, these repeats, mainly occurring in the intergenic regions, were considered as *junk dna*. However, mentality has changed; transposons, for instance, are now believed to have role in immune system [7] and gene regulation [14]. Depending on the species and of the kind of studied repeated element, the average number of occurrences of a repeat, its length and its divergence between occurrences show a large variability. In this paper, we focus on the problem of finding long multiple repeats that may appear dispersed along one whole genome or chromosome, or are common to different genomes/chromosomes. The proposed tool is designed for calling repeats that are multiple (whose occurrences number may be strictly bigger than 2), long (typically of length ≥ 100 base pairs), and approximate (each pair of occurrences may show substitutions, insertions or deletions in up to 10 to 15 % of their length).

The identification of such repeats, in particular in large and numerous genomes and when the divergence authorized between repeat occurrences is high, is a particularly difficult computational problem. Indeed, exact methods to find multiple repeats

Maria Federico, Pierre Peterlongo, Nadia Pisanti, Marie-France Sagot: Finding Long and Multiple Repeats with Edit Distance, pp. 83–97.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

use dynamic programming, leading to a time complexity in $O(n^r)$ with n the average length of sequences and r their number or the number of occurrences of searched repeats. Such a time complexity is unacceptable for practical use unless with toy examples. However, some tools as Speller [19], Smile [12] or Risotto [16] are designed to fin elements that are multiple and possibly spread over large (set of) genomes. However, this tools focus on particular user defined motifs that are indeed repeats but usually small, anchored with mandatory substrings, well conserved and mostly accepting only substitutions between occurrences. On the other hand, there exists a broad range of heuristic based algorithms to find repeats. Some make use of *seeds* for anchoring repeats before the application of dynamic programming and usually perform progressive alignments: combining pairwise alignments beginning with the most similar pair and progressing to the most distantly related. Even if efficient, such tools do not ensure the accuracy and completeness of the found results.

In order to find multiple repeats in reasonable time, it is possible to preprocess the data using a filter. In this framework, a filter is a tool that quickly discards fragments of sequences that may not belong to any searched repeat. After the filtering phase, usually the remaining dataset is much smaller than the original one, allowing the application of a time consuming algorithm. The user may refer to [17] for a state of the art about string filtration.

We propose FilmRed, a combinatorial approach that combines filtering and alignment phases. It is based on the Tuiuiu [15] filter. Tuiuiu is to date the state of the art filter as it filters for multiple repeats while previous filters are designed for repeats having only two occurrences or not taking into account indels. FilmRed (i) uses Tuiuiu as a preprocessing step and (ii) uses pieces of information collected during Tuiuiu runtime to detect, after filtering, real repeats and to find their precise borders and locations, thus finalising the repeats inference task.

2 The filter Tuiuiu and preliminary definitions

A *string* is a concatenation of zero or more symbols from an alphabet Σ. A string s of length n on Σ is represented also by $s[0]s[1]\cdots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. The length of s is denoted by $|s|$. We denote by $s[i,j]$ the *substring* $s[i]s[i+1]\cdots s[j]$ of s.

We will focus on the problem of finding (L,r,d)-*Erepeats*, defined as follow:

Definition 1 ((L,r,d)-Erepeat). *Given a set $\mathcal{S}$ of one or more input strings, a length $L > 0$, an integer $r \geq 2$, and an edit distance $0 \leq d < L$, we call a (L,r,d)-Erepeat a set $\{\omega_1,\ldots,\omega_r\}$ of r words having length in the range $[L-d, L+d]$ occurring in the sequences of $\mathcal{S}$ such that for all $i,j \in [1,r]$, $d_E(\omega_i,\omega_j) \leq d$, where $d_E(\omega,\omega')$ denotes the edit distance between two strings ω and ω'.*

The definition can be used to model repeats inside one sequence ($|\mathcal{S}| = 1$) or among several sequences ($|\mathcal{S}| > 1$). In the latter case, one can also enforce that the r words occur over r distinct sequences (and thus one needs $|\mathcal{S}| \geq r$). In both cases, should it be $r = 2$, the problem could be solved in quadratic time with dynamic programming just aligning the whole input against itself, but for multiple repeats like those we target, this solution is not feasible. Current exact exhaustive methods can manage input data of very limited size and/or detect repeats with very small values of d (the approximation measure), while again our target is higher as we want

d to be as much as 10 % or 15 % of L. On the other hand, heuristics do not guarantee to find all real repeats and, in general, the quality of their result much depends on the absence of noise in the data.

Filters, and in particular lossless filters[1], have been introduced with the goal of speeding up any method (exact or heuristic) by means of a drastic reduction of the input size obtained with the elimination of most of the data that does not contain any repeat. There is a twofold practical impact of filters: exact methods can push (possibly much) further the threshold of their applicability, while heuristics can gain in speed and possibly even obtain results of better quality. In general, a filter is a good filter if it is much faster than the search that it preprocesses (otherwise one would rather directly perform the search), and it is at the same as *selective* as possible, thus leaving the least amount of false positives, which are fragments of the input conserved by the filter and that turn out not to contain a repeat.

The lossless filter TUIUIU is specifically designed to preprocess the inference of (L, r, d)-*Erepeats*, and indeed it takes in input the parameters L, r, and d, as well as the input sequence(s). The tool slides a window w of length L along the whole input, checking whether there are at least $r - 1$ other fragments with which w fulfills a specifically designed strong and fast-to-check necessary condition for being at edit distance at most d. If this is the case, then the window w is kept, and it is discarded otherwise. The windows taken into account are those starting at each possible position of the sequences (these are roughly as many as the input size n), while the fragments for which the condition is checked against w are actually overlapping blocks of size $L + b + d$ occurring every b positions, where b is the smallest power of 2 larger than d. This choice, already done by previous filters [4,18] allows to consider only n/b blocks, thus gaining in speed. If a window w fulfills the necessary condition with a block B, then we say that B *is a friend* of w. The size chosen for blocks ensures that any occurrence of a word of an $(L, 2, d)$-Erepeat is always totally contained in at least one such block (and possibly in two), and hence the filter is still lossless. On the other hand, taking into account blocks rather than all possible fragments of size in $[L - d, L + d]$ starting positions, the selectivity of the filter becomes a bit weaker, as the necessary condition is checked against a block larger than the window, and in particular strictly greater than $L + d$ which is the largest possible size to be at edit distance at most d from a window of size L: this can be an additional source of false positives. In other words, the fact that a window has a block as a friend, does not necessarily mean that the block contains a fragment of size $L + d$ that fulfills the condition with w, and in this case the block is retained without deserving it.

Summing up, the choice made in TUIUIU has actually been to design a very strong necessary condition for two strings to be at edit distance at most d, and to insert this checking in a suitable framework that detects fragments of the input data that fulfill the requirement with respect to at least $r - 1$ others (belonging to distinct input strings when the requirement for the repeat is to occur in r distinct sequences). Doing this, the necessary condition for $(L, 2, d)$-*Erepeats* is actually turned into one for (L, r, d)-*Erepeats* for any $r \geq 2$. The fact that TUIUIU keeps a window w that has at least $r - 1$ blocks as friends gives reasonable hope that w and each one of these fragments are at edit distance at most d, but there is no indication that these

[1] With *lossless filters*, we refer to methods that filter the data ensuring that no fragments that may contain a similarity is removed.

fragments among themselves are at pairwise distance at most d and also that they will all be kept by the filter. Indeed, any (or both) of the following cases can hold:

1. One or more pair(s) of the friends of w may not even fulfill the necessary condition between themselves. In other words, the window has enough friends but these are not enough friends of each other. If one represents friendships as an edge, this condition can be seen as a guaranteed star shape structure with the window w as a center, while the requirement would actually be a clique.
2. It may turn out that a friend block of a window w is filtered out later during the filtering process because it does not contain any retained window. We call this case *empty block*. In such a case, if finally too many blocks friends of w are empty, then w would be left with less than non empty $r-1$ friend blocks and thus should be disposed.

Both cases can lead to w be a false positive should all these fragments be necessary to w for being part of a (L, r, d)-*Erepeat*. For this reason TUIUIU performs an extra check for empty blocks and, above all, multiple passes, to sensibly reduce the amount of such false positives with very little extra time requirement. For more details about TUIUIU and its optimization the reader can refer to [15,6]. In general, when using TUIUIU, we make a double pass as a default choice.

3 The algorithm FilmRed

In this section we will describe the pipeline of the algorithm FilmRed (FInding Long Multiple Repeats with Edit Distance) that is designed to exploit informations raised by TUIUIU to find (L, r, d)-*Erepeats*. We start with some observations about windows contained in overlapping blocks because the relation between windows and blocks that contain them is critical at some steps of the algorithm, and also because we will eventually merge overlapping blocks that turn out to contain a repeat in order to highlight possibly longer repeats.

3.1 Overlapping blocks and blocks merging

In this section we denote with c and c' the starting position of a block. In general we have $0 \leq c \leq n-1$, but all observations and definitions of this section regard cases in which the block contains at least a window and possibly it is not the rightmost block of the input sequence, and hence in such cases c has a more tight upper bound.

Observation 1 *Given a sequence S and an integer d. Let b be the smallest power of 2 larger than d. Any word w of length L in S can be totally contained in at most two consecutive blocks of size $L+b+d$. In particular:*

- *words $w = S[j, j+L-1]$ with $j \in [c, c+b-1]$ (and $c \leq n-b$) belong only to the block $B_i = S[c, c+b+d+L-1]$;*
- *words $w = S[k, k+L-1]$ with $k \in [c+b, c+b+d-1]$ (and $c \leq n-b-d-L+1$) belong to the consecutive blocks $B_i = S[c, c+b+d+L-1]$ and $B_{i+1} = S[c+b, c+2b+d+L-1]$.*

Definition 2. *Given a sequence S, two blocks B and B', starting in S at positions c and c' respectively, are* overlapped *iff $|c'-c| < L-(b+d)$.*

We define the *merging* of two consecutive blocks in the following manner:

Definition 3 (block merging). *Given a sequence S, let $B_i = S[c, c+b+d+L-1]$ and $B_{i+1} = S[c+b, c+2b+d+L-1]$ be two consecutive blocks in S. A larger block $B'_{i+1} = S[c, c+2b+d+L-1]$ of size $L+2b+d$ is obtained* merging *blocks B_i and B_{i+1}.*

The definition can be extended in a straightforward way to the merging of k consecutive blocks.

Definition 4. *Given a sequence S and k consecutive blocks $B_i, B_{i+1}, \ldots, B_{i+(k-1)}$ of S, such that B_i starts at position c in S, merging the k blocks we obtain an enlarged block $B'_{i+(k-1)} = S[c, c+kb+d+L-1]$ of size $L+kb+d$.*

3.2 Description of the algorithm

In this section we list the steps of the algorithm FilmRed.

Step 1: filtering step. The first step is actually to simply apply tuiuiu with double pass, hence including the optimization that allows to discard also some false positives due to empty blocks. With respect to the plain filter introduced in [15,6], in order to collect information which is useful to speed up the successive steps, we extend as follows this first phase. Information about not empty blocks that are friends of each window kept by the filter is stored in the array data structure *friendsOfWindow* whose size is the number of possible windows of length L (that is $n-L$, where n is the length of the input sequences). The entry *friendsOfWindow*$[w]$ of a specific window w contains the list of blocks that are friend of w.
At the end of this step, the portion of the input that is left is the one containing kept windows. In this way, a consistent percentage of the initial sequences is removed, and we are left with actual repeats plus some false positives. The possible cases of false positives have actually been described in Section 2 and, summing up, they can be due to one or more of the following reasons:

FP_{rect} : due to choice of checking the filtering condition for windows of size L against blocks of size $L+d+b$.
FP_{cond} : due to the fact that the condition the filter checks is only a necessary condition, but not sufficient.
FP^* : due to the condition being checked between a window and $r-1$ or more blocks (*star* shape) rather than between all such blocks (or actualy windows inside them).

Step 2: Semiglobal alignment. In this step, all windows kept by the filter after Step 1 are aligned to all its friend blocks. Only windows that result to have at least $r-1$ other fragments that are at edit distance smaller than d are actually kept. In other words, this step eliminates all FP_{rect} and FP_{cond} false positives. More specifically, this is achieved as follows. For each kept window w, a semiglobal alignment between w and B is performed for all blocks B in *friendsOfWindow*$[w]$. The window has length L while the block has length $L+b+d$. We build a rectangular dynamic programming matrix with the window w on rows and the block B on columns. The matrix is initialized with zero on the first row, indels and mismatches cost 1 and matches cost 0. In order to require that w is entirely involved in the alignment, while for B it is

enough to involve a substring, we check the last row: if there is a value lower or equal to d, then B contains a repeat of w, that is a substring of length in $[L-d, L+d]$ such that its edit distance from w is at most d; otherwise, the friendship of B with w was a false positive and B is removed from the list *friendsOfWindow*$[w]$. If with the removal of blocks from *friendsOfWindow*$[w]$ we obtain a list of size lower than $r-1$, then w is no longer a window to be kept, and is thus removed.

Each one of such alignments takes time $L(L+b+d)$, and the number of alignments to be performed depends from the dataset and from the efficiency of the filtering phase, that can only be evaluated experimentally (see Section 4 for experimental results). A theoretical complexity analysis based on the worst case scenario would result in a catastrophic expectation, not at all supported by practical cases. Among the reasons for which this step will actually result feasible there is the fact that we apply a simple optimization with relevant practical impact: when there are consecutive windows to be taken into account, there exists a relationship between the minimum cost of the alignment of a window w against a block B, and the minimum cost of the alignment of the successive window w' (that is, the window starting just one position after where w starts) and the same block B (who is likely to belong to *friendsOfWindow*$[w']$ if it did belong to *friendsOfWindow*$[w]$). When considering w' after that w has been processed, we are virtually removing the first row of the alignment between w and B, and adding an extra row on the bottom. If we denote with $dist(win, blo)$ the minimum value at the bottom row of the semiglobal alignment of a window win and a block blo, then we have that

$$dist(w,B) - 1 \leq dist(w',B) \leq dist(w,B) + 1$$

Therefore, storing for each block B, the minimum cost of the alignment with the last aligned window w, it is possible to determine lower and upper bounds of the alignment cost between B and the successive window w'. As a result, if $dist(w,B) \in [d, d+1]$, then the alignment between w' and B must be computed, but if $dist(w,B) \leq d-1$ (resp. $dist(w,B) > d+1$), then we know for free that $dist(w',B) \leq d$ (resp. $dist(w',B) > d$), and the alignments do not need to be computed.

During this Step, new empty blocks can be introduced: a false positive can be detected and discarded, and hence it may turn out that a block belonging to a list *friendsOfWindow*$[w]$ for some w is actually empty, that is, no window inside it is kept anymore. For this reason, a strategy of removal of empty blocks is performed also during the alignment step. This has the twofold effect of removing on the fly some FP^* and also to spare some alignment computations.

At the end of this step, all false positives FP_{cond} and FP_{rect} have been removed because now for all windows w the *friendsOfWindow*$[w]$ data structure only stores blocks containing at least one substring x of length in $[L-d, L+d]$ whose edit distance with w is $\leq d$. Nevertheless, some FP^* possibly still remain. These will be removed in the next step.

Step 3: Clique detection among blocks. At the beginning of this step, we have a set of windows that can be either real repeats or FP^* false positives. For each such window w we do know that in each block belonging to *friendsOfWindow*$[w]$ there is fragment at edit distance no greater than d with w, but this is not enough to guarantee w is part of an actual repeat. In order to ensure that, it should be that in any B_i (resp. B_j) of such blocks (actually in at least $r-1$ of them) there is a fragment f_i (resp. f_j) such that (i) $d_E(f_i, w) \leq d$, and (ii) for all pairs B_i and B_j of

blocks in this set, we have $d_E(f_i, f_j) \leq d$. The existence, for each block, of a fragment that fulfills condition (i) is guaranteed by previous steps, but the point is that the same f_i must fulfill condition (ii) as well. A possible way to see the problem we are about to address, is to represent each window and each fragment f_i as a node of a graph, and to place an edge between two nodes if these are at edit distance at most d: in this view, the selection made up to this step ensures that each w is a center of star shaped subgraph that has at least $r-1$ rays, but the actual requirement for this subgraph is now to be a clique. Indeed, a block may contain several (possibly and probably overlapping) fragments that are similar enough to another fragment and to w, but the requirement is that a block should be able to pick a single fragment that is similar enough to all other fragments. Somehow, the windows/block asymmetry is what causes this possible shift that may result into a mislocation of the repetition. Also with the goal of overtaking this problem, we relax the constraint over the length L transforming the *friendsOfWindow* data structure in the array of lists *friendsOfBlock* storing for each block B the list of not empty and overlapped blocks that are friends of the windows contained in B and kept after the alignment step.

The construction of the *friendsOfBlock* data structure is performed during the semi-global alignment step contemporarily to the update of the *friendsOfWindow* array. When we find that a window w_j has at least $r-1$ non overlapped friend blocks, if B_i is the block that contains w_j, we add the list of friend blocks stored in *friendsOfWindow*$[j]$ in *friendsOfBlock*$[i]$. Note that for the Observation 1 the window w_j can belong to two consecutive blocks B_i and B_{i+1}, hence in this case the list of friend blocks stored in *friendsOfWindow*$[j]$ is added to both *friendsOfBlock*$[i]$ and *friendsOfBlock*$[i+1]$.

The *friendsOfBlock* data structure is the adjacency list representation of the graph in which maximal cliques composed of at least r non overlapped friend blocks should be looked for. Blocks composing the found cliques contain occurrences of real multiple repeats having length in $[L-d, L+d]$ and that we can identify and visualize by aligning all the blocks of each clique.

Shifting from windows to blocks at this stage introduces an heuristic step that decreases the complexity of the clique detection task (because the size of the graph is reduced) and maintains the method lossless (*i.e.*, no (L, r, d)-*Erepeat* is missed) even though it might prevent the removal of some FP^*. We must say that in practice, in all our experiments (that is, all those reported in Section 4, and many more), we have never observed such kind of FP^*. Nevertheless, these can theoretically exist.

a. Finding maximal cliques.
   The Bron-Kerbosch [2] algorithm is an algorithm to find maximal cliques in an undirected graph. That is, it lists all subsets of vertices with the two properties that each pair of vertices in one of the listed subsets is connected by an edge, and no listed subset can have any additional vertices added to it while preserving its complete connectivity. We use this algorithm and, namely, the optimised version reported in [2]. This variant of the algorithm involves the selection of a "pivot" vertex for which in [8] two pivot selection strategies are investigated: we tested both on several and distinct types of biological sequences, and we end up choosing as pivot the vertex with largest degree because this strategy always outperforms the one based on random selection.
b. Removing clique redundancy.
   The graph represented by the *friendsOfBlock* array contains overlapped blocks as

friends of a block, therefore the Bron-Kerbosch algorithm performed over such graph finds a set of maximal cliques composed of overlapped blocks, in the sense that there is no clique that is a subset of another one. We have to perform the clique detection considering also overlapped blocks in order to enumerate exhaustively all real repeats. Nevertheless, it is possible that two different cliques in the set actually represent the same repeat. Indeed, for each pair of consecutive entries i and $i+1$ in *friendsOfBlock* corresponding to two consecutive blocks B_i and B_{i+1} that share a window w_t kept after the semi-global alignment step because it has at least $r-1$ non overlapped friend blocks, if w_t is not a FP^* the Bron-Kerbosch algorithm finds two cliques: $C = B_i \cup friendsOfWindow[w_t]$ and $C' = B_{i+1} \cup friendsOfWindow[w_t]$. Aligning blocks of both cliques C and C', the same repeat is found, because the only two different blocks between C and C' contain the same occurrence w_t of the repeat (even if it is possible that blocks B_i and B_{i+1} contain also other overlapped occurrences of the same repeat).
This kind of redundancy in the output is actually avoided storing in *friendsOfBlock*$[i]$ and *friendsOfBlock*$[i+1]$ also blocks B_{i+1} and B_i respectively. In this way the Bron-Kerbosch algorithm finds only the clique $C = B_i, B_{i+1} \cup friendsOfWindow[t]$.
On the other hand, the same type of redundancy now occurs within a single clique, because B_i and B_{i+1} represent the same occurrence of the repeat. Furthermore the same situation may happen also for other blocks in the list of friend blocks of w_t stored in *friendsOfWindow*$[t]$: indeed, for each friend block B_j that contains a kept window w_k belonging also to B_{j+1}, both B_j and B_{j+1} are friends of w_t, while representing the same occurrence of the repeat.

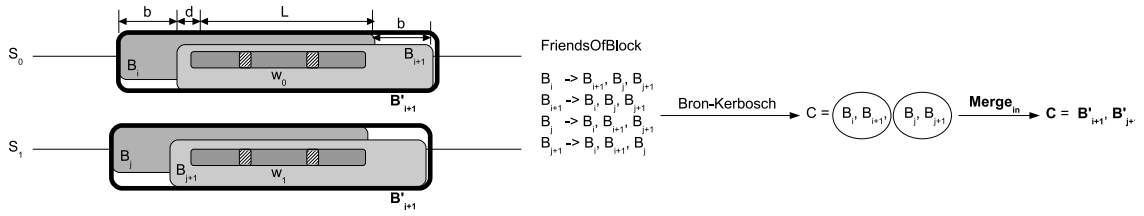


Figure 1. *Merge*$_{in}$ operation: given $r = 2$, windows w_0 and w_1 are two repeat occurrences shared by consecutive blocks B_i,B_{i+1} in sequence S_0 and B_j,B_{j+1} in sequence S_1 respectively. The Bron-Kerbosch algorithm finds the clique $C = B_i, B_{i+1}, B_j, B_{j+1}$ in which B_i and B_{i+1} (resp. B_j and B_{j+1}) contain the same occurrence. The *Merge*$_{in}$ operation consists in merging consecutive blocks inside the same clique. The white dashed areas possibly contain errors.

In order to remove such kind of redundancy inside a clique we merge consecutive blocks composing it, therefore if we found a clique $C = B_i, B_{i+1}, B_j, B_{j+1}$, we return the clique $C' = B'_{i+1}, B'_{j+1}$. In particular, the merging inside cliques is performed when a new block is added to a candidate clique. Note that the *merge*$_{in}$ operation is applied also for overlapped blocks that are present inside a clique, because they represent overlapped occurrences of the same repeat. We denote the merging of consecutive or overlapped blocks inside a clique as *Merge*$_{in}$. An example is shown in Figure 1.
Of course, more than two consecutive or overlapped blocks may be present in a clique, if they contain overlapped occurrences of the same repeat; in such case only one block that is the union of all consecutive and overlapped blocks is returned as part of the clique, therefore in the alignment we will see only one long occurrence.

Assuming that we perform the merging of consecutive and overlapped blocks inside each found clique, it may happen that the set of found cliques contains subsets of cliques each composed of consecutive blocks:

$$\begin{aligned} C &= B_i, B_j, B_k, B_t \\ C' &= B_{i+1}, B_{j+1}, B_{k+1}, B_{t+1} \\ C'' &= B_{i+2}, B_{j+2}, B_{k+2}, B_{t+2} \\ &\vdots \\ C^n &= B_{i+n}, B_{j+n}, B_{k+n}, B_{t+n} \end{aligned}$$

In this case it is plausible to think that in the input sequences there exists a longer repeat whose occurrences are the concatenation of the occurrences of shorter $n+1$ overlapped repeats with a certain degree of error d represented by cliques $C, C', C'', \ldots, C^n$. When the number of cliques composed of consecutive blocks is huge it means that the user chose a not very accurate value for the length L of repeats to be sought and this produces a little readable output difficult to be managed. To address this problem we decide to merge consecutive blocks contained in the $n+1$ cliques and to return only the clique $C = B'_{i+n}, B'_{j+n}, B'_{k+n}, B'_{t+n}$ representing the longer repeat. We denote the merging of consecutive blocks between

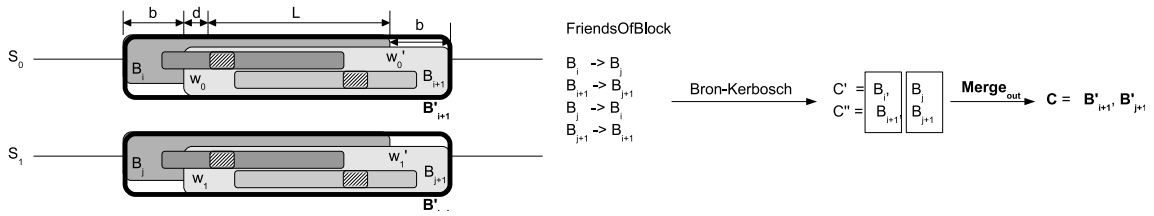


Figure 2. *Merge*$_{out}$ operation: given $r = 2$, windows w_0 and w_1 are two occurrences of the same repeat contained in blocks B_i in sequence S_0 and B_j in sequence S_1 respectively. Consecutive blocks B_{i+1} and B_{j+1} contain windows w'_0 and w'_1 which are overlapped to w_0 and w_1, and are occurrences of another repeat. The Bron-Kerbosch algorithm finds two cliques C' and C'' composed of consecutive blocks, but actually in the sequences there exists a long repeat whose occurrences are the concatenation of w_0 and w_1 in sequence S_0, and of w'_0 and w'_1 in sequence S_1. The *Merge*$_{out}$ operation consists in merging consecutive blocks between different cliques. The white dashed areas possibly contain errors.

different cliques as *Merge*$_{out}$. An example is shown in Figure 2.
The two situations of having consecutive and overlapped blocks inside the same cliques and in different cliques may happen simultaneously.
On the contrary, the merging of consecutive blocks contained in two different cliques is not performed if there exist at least two blocks that are not consecutive or overlapped in the two cliques.
Given that:

- as observed in Section 3.1 the enlarged blocks obtained from the merging of k consecutive blocks, have size at most $L + kb + d$;
- each block contains overlapped occurrences of a repeat of length L with an edit distance at most d from each other occurrence of the repeat,

then the enlarged blocks contain occurrences of repeats of length at most $L + kb + d$ with at most kd errors (if areas containing errors in overlapped windows of consecutive blocks are not overlapped, as in Figure 2).

In order to obtain such kind of compressed output, each clique found by the Bron-Kerbosch algorithm (possibly composed of enlarged blocks raised from the merging of consecutive blocks inside the clique) is compared with all previously found cliques and its blocks are merged with blocks of cliques composed of consecutive blocks.

Once the blocks containing the actual repeats have been detected, and that the noise due to redundancy there has been removed, then we are left with the output fulfilling the requirements.

4 Experiments and Discussion

4.1 Applications of FilmRed

This section shows results of an extensive set of tests performed to validate FilmRed to find (L, r, d)-*Erepeats* in biological datasets containing one or more whole genomes or chromosomes of Sunflower, Saccharomyces Cerevisiae, and in the CFTR dataset ([3]) containing, for five different organisms (chicken, cow, human, mouse and tetra), as many ortholog regions of the Cystic Fibrosis Transmembrane conductance Regulator gene. Performances of the different steps of FilmRed will be evaluated in terms of running time. Furthermore, we will also evaluate its selectiveness ability, in terms of amount of data left after the first and the second steps of FilmRed (that is, the filtering step, and the semiglobal alignment steps that removes FP_{cond} and FP_{rect}, respectively). The *selection* of these two steps is defined as the ratio between the number of non-removed overlapped substrings of length L and the total number of overlapped substrings of length L present in the input sequences. Formally, the selection of both steps 1) and 2) of FilmRed is given by:

$$sel = \frac{\text{number of words of length L kept by FilmRed step}}{\text{number of words of length L in the input sequences}}.$$

Obviously, given that both phases are lossless, the smaller the selection, the better. On the other hand, for step 3) (that is, the clique detection step and the redundancy removal, respectively), we report the number of output cliques.

Tests with Sunflower BAC sequences. A possible application to biological data in which an accurate (L, r, d)-*Erepeats* finder can be employed, is that of detecting LTR sequences (LTR is the acronym for Long Terminal Repeats, that are the sequences of about 300 bp length repeated at both ends of a transposable element). In order to check whether this assumption is correct, as a first experiment we applied FilmRed to four different datasets composed of a single BAC sequence of the Sunflower, using length parameters that agree with the expected structure of LTRs ($L = 200, 300$, with $d = 20, 30$, respectively).

Table 1 and Table 2 report results of FilmRed for one of these four datasets denoted as *bacKnapp* and containing 107161 bases, using respectively $r = 2$ and $r = 3$. The results for the other three data sets were practically equivalent to that we report.

Analyzing in detail the performance of the single steps of FilmRed we observe that, as expected, the most time consuming step is the semiglobal alignment between windows and friend blocks (except for the very special case of last line of Table 2 that

		Filter		Semiglobal Align		Clique detection		Total
L	d	time(s)	sel	time(s)	sel	time	#cliques	time(s)
200	20	0.50	12.47 %	5.13	10.32 %	0.00	8	5.63
300	30	0.49	12.12 %	10.85	9.85 %	0.01	5	11.34

Table 1. Performances of the different phases of FilmRed to find (L, r, d)-*Erepeats* on the Sunflower *bacKnapp* dataset (107161 bases), with $r = 2$.

we will specifically comment later). However, we are able to perform the alignment task in reasonable time for all parameters (and this holds for all the four datasets), because the pre-processing filtering step sensibly reduces the input size.

The other step with an high theoretical computational complexity is clique detection performed on the graph of friend blocks. However, we observe that, even though the Bron-Kerbosch algorithm applied on an n-vertex graph has a time complexity exponential in n, the clique detection phase is very fast in all tests, and even more when $r = 3$ instead of $r = 2$, that is when the clique is less trivial, because it is performed on really small graphs of friend blocks, thanks to the filtering of input sequences and the little amount of false positives remaining after the semiglobal alignment step. For what concerns the number of detected cliques, we can deduce that our strategy of compression of the output allows us to obtain a restricted output. Indeed, FilmRed returns very few repeats, especially when $r = 3$. Finally, the last two lines of Table 2 report tests in which the allowed edit distance is pushed quite far (45 edit operations allowed in a 300 bases long repeat means 15 % of the involved bases): no new result raises in this LTR finding task, but we can see that the time performances of FilmRed are good, even if the filters helps much less and takes more time.

In addition, in order to validate our results, we compared repeats found by FilmRed in the Sunflower with the ones found by the signature-based repeat finding tool LTR_Finder [22]. Given that no annotation is available yet, then the output of such a tool is the only result we can compare to. We observed that all the repeats identified by the other tool are found also by FilmRed. The latter, however, returns also further repeats, which are not identified by the former. These results suggest that FilmRed can provide a fast solution to the problem of finding long repeats modeling LTRs.

		Filter		Semiglobal Align		Clique detection		Total
L	d	time(s)	sel	time(s)	sel	time	#cliques	time(s)
200	20	0.44	3.32 %	3.38	1.10 %	0.00	3	3.82
300	30	0.46	3.42 %	7.36	0.98 %	0.00	2	7.82
200	25	0.59	5.66 %	4.24	2.57 %	0.00	3	4.83
300	45	178.25	41.70 %	35.59	3.15 %	0.00	2	213.84

Table 2. Performances of the different phases of FilmRed to find (L, r, d)-*Erepeats* on the Sunflower *bacKnapp* dataset (107161 bases), with $r = 3$.

Tests with Saccharomyces Cerevisiae genomes. We performed experiments on the dataset *s288c+w303* composed of three whole genomes (16 chromosomes each) of three different strains of *S. cerevisiae*: RefSeq (that is fully annotated in the *Saccharomyces* Genome Database), S288c and W303, for a total of 26392324 bases. The dataset was pre-processed by the regender tool [1] (the reported size is that after regender is applied) in order to remove the resident genome (*i.e.*, the total immotile

DNA), which is equal among all the strains and does not contain mobile elements like transposable elements. The goal of applying FilmRed to this dataset is to detect transposable elements and LTRs that are shared by the three strains, and that could not be detected by means of a traditional global alignment because in general, being part of the most mobile DNA, have lost their colinearity.

		Filter		Semiglobal Align		Clique detection		Total
L	d	time(s)	sel	time(s)	sel	time	#cliques	time(s)
200	20	29.44	0.17 %	744.48	0.09 %	6.30	24	780.22
300	30	31.68	0.16 %	1473.65	0.07 %	2.13	13	1507.46
5000	500	9.00	0	-	-	-	-	9.00

Table 3. Performances of the different phases of FilmRed to find (L, r, d)-*Erepeats* on the *s288c+w303* dataset (26392324 bases) of the S. Cerevisiae, with $r = 3$.

Table 3 reports results of tests performed to find (L, r, d)-*Erepeats* characterized by the following parameters: $r = 3$, $L = 200, 300, 5000$ with $d = 20, 30, 500$, respectively, in the *s288c+w303* dataset. We chose these parameters based on the peculiar structure of the transposons that can be evinced from the annotation of RefSeq provided in the *Saccharomyces* Genome Database (available at `http://www.yeastgenome.org`): they are long between 5000 and 6000 bases and are delimited by two LTRs of 200-300 bases.

Basically, all the observations we made for the sunflower data set hold here as well, including the fact that our tool is a good candidate to detect LTRs: for this data set an annotation is available, and hence in this case we could really validate our results. In particular, we checked whether the repeats found by FilmRed in this dataset of S. Cerevisiae correspond to real LTRs whose annotation is available in the *Saccharomyces* Genome Database (`http://www.yeastgenome.org`). We found that repeats output using parameters $L = 300$, $d = 30$, $r = 2$ actually correspond to real LTRs, or are part of retrotransposons, or they match with the sequence of putative proteins of unknown function. For example, blocks composing a detected clique contain occurrences of the following annotated LTRs: YCLWdelta3 and YCLWdelta5 in chromosome III, YDRWdelta19 and YDRWdelta28 in chromosome IV, and YLRWdelta14 and YLRWdelta23 in chromosome XII. For longer repeated sequences such as transposons and retrotrasposons, nothing is selected, as expected, probably because the edit distance with 10 % of edit operations is not the right framework to capture transposons' divergence.

4.2 Comparison with other tools

As already pointed out, to the best of our knowledge, FilmRed is the first *ab initio* tool that can deal with repeats occurring in possibly more than two sequences, that have length of hundreds or thousands of bases, and whose occurrences may differ in even more than 10 % of their positions in terms of substitutions and indels. For this reason we cannot compare FilmRed with other methods solving the same problem. In this section we report results of experiments performed to compare FilmRed with existing methods for local similarity search. In particular, as the major strength of FilmRed is its capacity to identify repeats in more than two input sequences, we concentrated our attention on existing tools for multiple local sequence alignment. It is important to note, however, that the output provided by FilmRed and the one

provided by multiple local alignment tools are different, because the tasks addressed by the two kinds of tool are different. Indeed, FilmRed returns repeats and their occurrences, while the output of multiple local alignment tools is the alignment of whole input sequences in which we can identify local similarity areas (the repeats) looking at the alignment.

We compared FilmRed with some of the most popular multiple local alignment tools on the CFTR dataset, which is the smallest dataset (5.5 Mbases) composed of more than two sequences that we have studied in our work. Experiments were run on an Intel(R) Quad-core Xeon(R) E5405/2 GHz with 10 GB of RAM.

Table 4 reports results of experiments performed on the CFTR dataset ([3]) composed of 5518041 bases. Experiments were performed using parameters: $L = 100$, $r = 5$ and $d = 7, 12, 14, 15$, with $r = 5$.

	Filter		Semiglobal Align		Clique detection		Total
d	time(s)	sel	time(s)	sel	time	#cliques	time(s)
7	64.20	0.05 %	56.56	0	-	-	120.76
12	1017.51	0.01 %	0.88	0	-	-	1018.39
14	3772.65	0.02 %	1.41	0.001 %	0.00	1	3774.06
15	7128.19	0.65 %	740.01	0.003 %	0.01	1	7868.21

Table 4. Performances of the different phases of FilmRed to find (L, r, d)-*Erepeats* on the CFTR dataset (5518041 bases), with $L = 100$ and $r = 5$.

We can see that for low values of d, no repeat is detected, while for larger d there is a repeat that, besides the fact that its occurrences pairwise show 15 % of differences, our tool is fast to find.

Tool	Class	Result
MSA [11]	exact	manages sequences at most 50 character long
ClustalW [21]	progressive	runs for more than 38 hours
TCoffee [13]	progressive	runs out of memory
Kalign [10,9]	progressive	runs for more than 28 hours
DiAlign [20]	iterative	runs out of memory
MUSCLE [5]	iterative	runs out of memory

Table 5. Results of several multiple local sequence alignment tools on the CFTR dataset.

We have tried to search for other tools able to find the same results (that is, for example, the existing repeat of L=100 bases long occurring in all five sequences of the CFTR data set, and with up to 14 % pairwise edit distance between occurrences that is detected by FilmRed) with which we could compare the performances of FilmRed. Table 5 summarizes the results of the comparison. As we can see, none of the tested tools was able to manage in reasonable time and without huge memory usage, inputs as large as the one provided by sequences in the CFTR dataset. On the contrary, as shown in Table 4, FilmRed ends its computation in reasonable time on this dataset with these parameters.

5 Conclusion and Perspectives

The problem of finding *long repeats* approximated with edit distance, modelling transposable elements in biological sequences, is computationally challenging when a certain non negligible number of insertions, deletions and substitutions are admitted in

repeat occurrences. For this reason the exhaustive discovery of such repeats might be unfeasible for many instances. We proposed an ab initio method, called FilmRed, which is, to the best of our knowledge, the first tool that can deal with repeats occurring possibly several times, that have length of hundreds or thousands of bases, and whose occurrences may differ in even more than 10 % of their positions in terms of substitutions and indels. This is achieved by using a filter as a preprocessing step in order to discard as many as possible fragments of sequences that are guaranteed not to contain any searched repeat, and using the information gathered during the filtering phase in order to speed up a successive dynamic programming based alignment step performed to infer the repeats. Although, in theory, the current version of FilmRed might return some false positives due to the introduction of a localized heuristic step in the method, we have never observed them in practice. Future work will consist in clearly evaluating the false positive rate and finding a new way for fixing the problem.

References

1. G. Battaglia, R. Grossi, N. Pisanti, R. Marangoni, and G. Menconi: *Inferring mobile elements in S.Cerevisiae strains*, in Proceedings of International Conference on Bioinformatics Models, Methods and Algorithms (BIOINFORMARTICS), 2011. In press.
2. C. Bron and J. Kerbosch: *Algorithm 457: finding all cliques of an undirected graph.* Communication of ACM, 16(9) 1973, pp. 575–577.
3. M. Brudno et al.: *LAGAN and Multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic DNA.* Genome Research, 13 2003, pp. 721–731.
4. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron: *q-gram based database searching using a suffix array (QUASAR)*, in ACM Conference on Research in COmputational Molecular Biology (RECOMB 1999), 1999, pp. 77–83.
5. R. C. Edgar: *Muscle: multiple sequence alignment with high accuracy and high throughput.* Nucleic Acids Research, 32 2004, pp. 1792–1797.
6. M. Federico, P. Peterlongo, and N. Pisanti: *An optimized filter for finding multiple repeats in DNA sequences*, in Proceedings of the 8th ACS/IEEE International Conference on COmputer Systems and Applications (AICCSA 2010), IEEE Computer Society Press, 2010, pp. 1–8.
7. J. M. Jones and M. Gellert: *The taming of a transposon: V(D)J recombination and the immune system.* Immunological Reviews, 200(1) 2004, pp. 233–248.
8. I. Koch: *Fundamental study: Enumerating all connected maximal common subgraphs in two graphs.* Theoretical Computer Science, 250 2001, pp. 1–30.
9. T. Lassmann, O. Frings, and E. L. L. Sonnhammer: *Kalign2: high-performance multiple alignment of protein and nucleotide sequences allowing external features.* Nucleic Acid Research, 37(3) 2009, pp. 858–865.
10. T. Lassmann and E. L. Sonnhammer: *Kalign – an accurate and fast multiple sequence alignment algorithm.* BMC Bioinformatics, 6 2005.
11. D. J. Lipman, S. F. Altschul, and J. D. Kececioglu: *A tool for multiple sequence alignment*, in Proceedings of National Acadademy of Sciences, 1989, pp. 4412–4415.
12. L. Marsan and M.-F. Sagot: *Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification.* Journal of Computational Biology, 7(3–4) 2000, pp. 345–362.
13. C. Notredame, D. G. Higgins, and J. Heringa: *T-Coffee: a novel method for fast and accurate multiple sequence alignment.* Journal of Molecular Biology, 302 2000, pp. 205–217.
14. V. Pereira, D. Enard, and A. Eyre-Walker: *The Effect of Trasposable Element Insertions on Gene Expression Evolution in Rodents.* PLoS one, 4(2) 2009, p. e4321.
15. P. Peterlongo, G. T. Sacomoto, A. P. do Lago, N. Pisanti, and M.-F. Sagot: *Lossless filter for multiple repeats with bounded edit distance.* Algorithms for Molecular Biology, 4 2009.

16. N. Pisanti, A. M. Carvalho, L. Marsan, and M.-F. Sagot: *Risotto: Fast extraction of motifs with mismatches*, in LATIN, 2006, pp. 757–768.
17. N. Pisanti, M. Giraud, and P. Peterlongo: *Filters and seeds approaches for fast homology searches in large datasets*, in Algorithms in computational molecular biology, M. Elloumi and A. Y. Zomaya, eds., John Wiley & sons, 2010.
18. K. Rasmussen, J. Stoye, and E. Myers: *Efficient q-gram Filters for finding all epsilon-matches over a given length.* Journal of Computational Biology, 13(2) 2006, pp. 296–308.
19. M.-F. Sagot: *Spelling approximate repeated or common motifs using a suffix tree*, in LATIN, 1998, pp. 374–390.
20. A. R. Subramanian, M. Kauffmann, and B. Morgenstern: *DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment.* Algorithms for Molecular Biology, 3 2008.
21. J. D. Thompson, D. G. Higgins, and T. J. Gibson: *Clustal W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice.* Nucleic Acids Research, 22 1994, pp. 4673–4680.
22. Z. Xu and H. Wang: *LTR_FINDER: an efficient tool for the prediction of full-length LTR retrotransposons.* Nucleic Acids Research, 35 2007, pp. W265–W268.

An Improved Version of the Runs Algorithm Based on Crochemore's Partitioning Algorithm

Frantisek Franek*, Mei Jiang**, and Chia-Chun Weng

Department of Computing & Software
Faculty of Engineering
McMaster University
Hamilton, Ontario
Canada L8S 4K1
{franek,jiangm5,wengc2}@mcmaster.ca

Abstract. Though there are in theory linear-time algorithms for computing runs in strings, recently two of the authors implemented an $O(n \log n)$ algorithm to compute runs that was based on the Crochemore's partitioning repetitions algorithm. The algorithm preserved the running complexity of the underlying Crochemore's algorithm; however, the static memory requirement – already large at $14n$ integers for a string of length n – was increased significantly to $O(n \log n)$ integers. The purpose and advantage of this algorithm was its speed. In this paper we present a more advanced version of the extension of the Crochemore's algorithm for computing runs. This version in addition to maximal repetitions, computes runs and primitively rooted distinct squares. Its implementation completely does away with the extra memory required for the previous version and through some additional memory saving techniques, the overall memory need was reduced to $13n$ integers.

Keywords: repetition, run, distinct squares, string, periodicity, suffix array, LCP array, Lempel-Ziv factorization

1 Introduction

Crochemore's repetitions algorithm, often also referred to as Crochemore's partitioning algorithm, was introduced in 1981 [2] and was the first $O(n \log n)$ – and hence optimal – algorithm to compute maximal repetitions in a string of length n. The big advantage of the algorithm was its independence on the size of the alphabet of the string. Its disadvantage was in the implementation, as the data structures required for keeping track of the refinement process and the gaps require a substantial storage – originally estimated in about $20n$ of integers – and a complex machinery to update and maintain them. In 2003, Franek, Smyth, and Xiao [6] implemented the algorithm using several memory saving techniques lowering the requirement to $14n$ integers. An additional advantage of their implementation was that the memory was static, or to be more precise, allocated all at once at the outset of the algorithm as the working of the algorithm did not require any dynamic allocation or deallocation of memory. This approach led to an implementation with quite fast running times.

Since the advent of linear-time algorithms to compute suffix arrays [8,10,11], an avenue opened for true linear-time algorithms to compute runs.

* Supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada

** Supported in part by Queen Elizabeth II Graduate Scholarship in Science and Technology, and Ontario Graduate Scholarship Program

Frantisek Franek, Mei Jiang, Chia-Chun Weng: An Improved Version of the Runs Algorithm Based on Crochemore's Partitioning Algorithm, pp. 98–105.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

Such algorithms follow the general strategy of

(a) compute suffix array using any of the linear-time algorithms, for instance [8,10,17,11,18],
(b) compute LCP (longest common prefix) array using any of the linear-time algorithms, for instance [9,16],
(c) compute Lempel-Ziv factorization using any of the linear-time algorithms, for instance [1,3],
(d) compute some runs that include all leftmost runs from the Lempel-Ziv factorization using Main's algorithm [14,15],
(e) from the runs computed in (d), compute all runs using Kolpakov-Kucherov's approach [12,13].

The laborious and circuitous strategy for linear-time algorithms suggests that performance of such algorithms may not be satisfactory. Franek and Jiang [4,5] extended the original Crochemore's repetitions algorithm to compute runs with a plan to benchmark the algorithm and compare it with any implementation of the linear-time algorithm for computing runs. Their implementation was based on Franek, Smyth, Xiao's implementation [6] for its optimized memory handling. The approach was quite straightforward: the maximal repetitions as reported were collected and consolidated into runs. This necessitated additional data structures of size $O(n \log n)$ integers. The program still exhibited fast running times, but the memory requirement was too substantial and required dynamic handling of memory during processing, which is quite a detriment to fast performance.

The reason to revisit the algorithm and modify it was to lower the memory requirement, eliminate the need of dynamic memory allocation and deallocation during processing, and prepare the stage for the parallelization. This report describes the new implementation that requires only a single allocation of $13n$ integers at the outset of the algorithm, preserves all the advantages of the previous implementations, computes not only the maximal repetitions – *as the original Crochemore's algorithm does*, but also the runs – *as the Franek and Jiang's implementation does*, and in addition it computes the number of primitively rooted distinct squares. Moreover, the algorithm in this form is well-posed for parallelization in the shared-memory model. We refer to this algorithm as FJW.

2 Preliminaries

For $e \geq 2$ and a non-empty string w, $(ww)^e$ is a *repetition of power* e in a string x if there are strings u and v, possibly empty, so that $x = u(ww)^e v$. w is referred to as the *generator* of the repetition, while the size of the generator is referred to as the *period* of the repetition. If $e = 2$, we talk of a *square*. A string is *primitive* if it is not a repetition. A repetition is *primitively rooted* if its generator is primitive. A repetition $(ww)^e$ in $x = u(ww)^e v$ is *maximal* if w is neither a suffix of u nor a prefix of v. For a string $x = x[0..n1]$, a repetition can be encoded as a triple (s, p, e), where s is the starting position of the repetition, p is the period, and e is the power.

A more succinct notion is that of a run. In a string $x = x[0..n1]$ a quadruple (s, p, e, t) encodes a *run* if

(a) for any $0 \leq i \leq t$, $(s{+}i, p, e)$ is a maximal repetition,
(b) either $s = 0$ or $(s1, p, 2)$ is not a square, i.e. the run cannot be extended to the left,

(c) either $s{+}t = n1$ or $(s{+}t{+}1, p, 2)$ is not a square, i.e. the run cannot be extended to the right,
(d) the generator $x[s..s{+}p1]$ is primitive.

The maximum number of maximal repetitions in a string of length n is $O(n \log n)$, see [2]. On the other hand, the maximum number of runs is $\leq 1.029n$, see [19]. In [4,5], Franek and Jiang used Crochemore's repetitions algorithm to generate all maximal primitively rooted repetitions, collect them in a data structure of size $O(n \log n)$ and then in $O(n \log n)$ time process the collected repetitions and consolidate them into runs. Though the repetitions computed by Crochemore's algorithm are not in any particular order – except the fact that repetitions of the same period are computed at the same stage, a detailed examination of the gap function revealed that there is no need to collect the repetitions, that the runs can be directly inferred from the information provided by the gap function.

To be able to discuss the gap function and show how the runs can be directly inferred, we need to briefly discuss the mechanism of the Crochemore's repetitions algorithm.

3 Brief description of Crochemore's repetitions algorithm

In mathematical terms, the algorithm is simple and elegant and relies on the refinements of classes of equivalence of the positions of the input string $x = x[0..n1]$. An equivalence $\approx_k$ is defined on the set of indices $\{0, \ldots, n1\}$ by $i_1 \approx_k i_2$ if and only if $x[i_1..i_1{+}k1] = x[i_2..i_2{+}k1]$. In simple terms, two positions are $\approx_k$ equivalent, if the substrings of length k starting at those two positions are the same. In all times, the algorithm maintains an ascending order of the indices in each class, though no particular order of the classes themselves.

At the first level, the algorithm computes by brute force the classes of equivalence $\approx_1$. These classes in fact represent all the positions with the same alphabet symbol. On each following level k, all classes of equivalence $\approx_k$ are computed. Note that each class from level $k1$ is either preserved as a class on level k, or is partitioned into several disjoint classes which we will refer to as *family*. That is why the Crochemore's algorithm is also referred to as the partitioning algorithm. It is clear that once a class has size 1, it cannot be partitioned any further. The processing ends when all classes are of size 1.

The classes, indeed, contain all information of all possible repeats of substrings of x. It is straightforward to see that a primitively rooted square of period p must be represented by two consecutive indices i_1 and i_2 in the same class of $\approx_p$ so that $|i_1 i_2| = p$.

The main complication of the algorithm lies in the process of refinements. If the refinements were carried out directly through references to the input string, the running complexity would be unacceptable $O(n^2)$. However, the refinement of the class on level k can be carried out by using other classes on level k which allows to discard the original string once the classes on the first level had been computed. This approach, though much better than the refinement through direct reference to the input string, would still lead to the running complexity of $O(n^2)$. If in each family we take a largest class by size and designate it *big* and all other as *small*, we can carry the full refinement of all the classes using just the small classes. Since any position can

occur in at most $O(\log n)$ small classes, this approach gives the running complexity of $O(n \log n)$.

Not to destroy the $O(n \log n)$ complexity, we cannot afford to scan the classes when looking for squares and ultimately for maximal repetitions. Throughout the whole process of refinement, a function $Gap(p)$ is maintained that gives a list of all indices that are exactly p distance from its predecessor in the class, more precisely: when processing level k, if $i_2 \in Gap(p)$, then $i_1 = i_2 p$ is in the same class of equivalence $\approx_k$ as i_2 and these two indices are consecutive in the class. We will describe the $Gap()$ function in more detail in the next section dealing with the implementation of the FJW algorithm.

4 Implementation of the FJW

We first describe the implementation and its data structures without any regard for the size of required memory. This leads to an implementation requiring $19n$ of integers. Then we use several techniques to reduce the required memory to $13n$ integers. We will present the data structures as static, but for practical reasons – we do not want to recompile the program each time a different string is to be processed, all the structures are allocated once at the outset of the program's processing. The structures are essentially arrays used to emulate doubly-linked lists, stacks, and queues.

The first seven arrays deal with classes:

1. An integer array $CStart\,[0..n1]$ stores the very first element of a class, i.e. $CStart[i] = j$ means that the first element of class i is j. *This emulates a pointer to the beginning of a class.*
2. An integer array $CEnd[0..n1]$ stores the very last element of a class, i.e. $CEnd[i] = j$ means that the last element of class i is j. *This emulates the pointer to the end of a class.*
3. An integer array $CNext[0..n1]$ stores the next element in the class or `null`. Thus $CNext[i] = j$ indicates that i and j are in the same class and that j is the next element after i, while $CNext[i] =$`null` indicates the i is the last element in the class. *This emulates the forward links.*
4. An integer array $CPrev[0..n1]$ stores the previous element in the class or `null`. Thus $CPrev[i] = j$ indicates that i and j are in the same class and that j is the element just before i, while $CPrev[i] =$`null` indicates the i is the first element in the class. *This emulates the backward links.*
5. An integer array $CMember[0..n1]$ stores the membership of each element, i.e. $CMember[i] = j$ means that i belongs to the class j.
6. An integer array $CSize[0..n1]$ stores the sizes of classes, i.e. $CSize[i] = j$ means that class i has size j.
7. An integer array $CEmpty[0..n1]$ is used as a stack of empty classes to be used.

The following four arrays deal with families:

1. An integer array $FStart[0..n1]$ is used as a stack. $FStart[i] = j$ thus means that class j is the first class in the family i.
2. An integer array $FNext[0..n1]$ emulates the forward links in a list of classes in a family.
3. An integer array $FPrev[0..n1]$ emulates the backward links.

4. An integer array $FMember\,[0\,..\,n1]$ stores the family membership, i.e. $FMember[i] = j$ means that class i belongs to family j.

The following four arrays deal with the refinement process:

1. An integer array $Refine[0..n1]$. $Refine[i] = j$ means that an element from class i should be moved to class j.
2. An integer array $RStack[0..n1]$ is used as a stack. It is used to remember which items in *Refine*[] were occupied, so it can be cleared without any need to traverse the whole array *Refine*[] which would destroy the $O(n \log n)$ complexity.
3. An integer array $Sel[]$ is used as a queue. It is the queue of all elements of all small classes.
4. An integer array $Sc[]$ is used as a queue of small classes. $Sc[i] = j$ indicates j is the last element of a small class. Thus the information in $Sel[]$ and $Sc[]$ implements a list of elements of small classes with indicators where one small class ends and the next small class starts.

The last four arrays implement the gap function:

1. An integer array $Gap[0..n1]$. $Gap[i] = j$ indicates that the first element in the gap list for i is j, i.e. j's predecessor in the class is ji.
2. An integer array $GMember[0..n1]$. $GMember[i] = j$ means that i belongs to the gap list j.
3. An integer array $GNext[0..n1]$ emulates the forward links in the gap lists.
4. An integer array $GPrev[0..n1]$ emulates the backward links in the gap lists.

The C++ code for this version is the file `crochB.cpp` and electronically available at [20].

In the next version, `crochB1.cpp`, also posted at [20], the array $GMember[]$ is replaced by a function $GMember()$ and the memory requirement is reduced to $18n$ integers. $GMember()$ can be directly computed:

$$GMember(i) = \begin{cases} \texttt{null} & \text{if } i \text{ is not member of any class,} \\ \texttt{null} & \text{if } i \text{ is the first member of a class,} \\ iCPrev[i] & \text{otherwise.} \end{cases}$$

Version `crochB3.cpp` reduces the memory requirement further to $17n$ integers. Consider any family doubly-linked list, its beginning can be determined by two means: $FStart[i] = j$ or $FPrev[j] =$`null`. Thus, we can do away with $FMember[]$ array and replace it by a function that is utilizing the redundant space in $FStart[]$ and $FNext[]$:

$$FMember(i) = \begin{cases} FStart[i] & \text{if the stack pointer is null,} \\ FNext[FPrev[FStart[i]]] & \text{if } i \leq \text{the stack pointer,} \\ FStart[i] & \text{otherwise.} \end{cases}$$

We also introduce a function $FEnd()$ computed from $FStart[]$ and $FPrev[]$: $FEnd(i) = FPrev[FStart[i]]$.

In the next version, `crochB4.cpp`, $CEmpty[]$ and $Sc[]$ are made to share the same memory segment, reducing the memory requirement to $16n$ integers.

Version `crochB5.cpp` distributes $CEnd[]$ and $CSize[]$ over $CStart$, $CNext$, and $CPrev$, thus reducing the memory requirement further to $14n$ integers. Therefore, $CEnd(i) = CPrev[CStart[i]]$ and $CSize(i) = CNext[CPrev[CStart[i]]]$.

If we limit the maximal possible length of an input string from `UNSIGNED LONG MAX` to `LONG MAX`, which for a 32-bit long it is 2,147,483,647 and thus large enough, we can virtualize $CMember[]$ over $Gap[]$, $GNext[]$, and $GPrev[]$, reducing the memory requirement to $13n$ integers. Thus, the function to set the value of $CMember(e)$ to c:

```
if (Gap[e] == null || Gap[e] < 0)
     if (c == null)
         Gap[e] = null;
     else
         Gap[e] = 0-1-c;
else
    if (c == null)
        GNext[GPrev[Gap[e]]] = null;
    else
        GNext[GPrev[Gap[e]]] = 0-1-c;
```

and the function to get the value of $CMember(e)$:

```
if (Gap[e]==null)
     return null;
else
     if (Gap[e] < 0)
         return 0-1-Gap[e];
     else
         if (GNext[GPrev[Gap[e]]] == null)
             return null;
         else
             return 0-1-GNext[GPrev[Gap[e]]];
```

The version `crochB7.cpp` is just a polished version of `crochB6.cpp` with the additional features discussed in the next section.

5 The gap function and computations of distinct squares, maximal repetitions, and runs

Throughout the process of refinement, the gap function is maintained. In order to protect the running complexity of $O(n \log n)$, every time an element is removed from a class, the gap function is updated; and any time an element is added to a class, the gap function is updated again. When computing the next level from the current one, $Gap[p]$ points to the first element whose immediate predecessor in its class is exactly at distance p, while $GNext[]$ and $GPrev[]$ allow us to traverse the whole list in either direction and to update the list in constant time. Notice that if $Gap[p] = i$ and we are dealing with level p of refinement, then there is a primitively rooted square starting at position $GPrev[i]$ of period p.

Computing Distinct Squares – `traceSquares()` method

The gap function can be used to compute primitively rooted distinct squares. As we traverse the gap list, once we identify the first primitively rooted square in each class,

we ignore the identification of the rest from the same class as they are all identical squares. We use *Refine*[] and *RStack*[] that are only needed during the refinement process as auxiliary data structures here to indicate the last class we already have a representative from in order not to get another representative from the same class. Note that the program can either output the number of distinct squares, the triples (s, p, e) encoding the squares identified, or the squares as identified substrings of the input string – we refer to is as *pretty print*. To use *pretty print*, the string alphabet should be the lower case letters $a, b, \ldots$

Computing Maximal Repetitions – `traceMaxReps()` method

For the maximal primitively rooted repetitions, again either their number can be output, the individual repetitions in their encoding into triples or *pretty print* can be used. The algorithm traverses the gap list, and for each entry it checks how far left and how far right it can extend the square. Thus, during the tracing at level p, all the individual squares identified are consolidated into maximal repetitions. A brief description on how the algorithm determines if the square can be extended to the left: the entry i from the gap list $Gap[p]$ indicates that there is a primitively rooted square starting at position ip. Then the algorithm checks if the square can be extended to the left – i.e. is there a square of period p starting at position $i2p$ and determined by ip. It is possible that the position ip is in the gap list further away. In order not to process the square starting at $i2p$ and determined by ip, we again use *Refine*[] and *Rstack*[] to indicate that this entry has already been processed.

Computing Runs – `traceRuns()` method

The computation of runs is performed by $TraceRuns()$. The idea is very similar to that of tracing maximal repetitions: the identified primitively rooted squares are consolidated to runs. If you look at the leading square of a run (s, p, e, t) that must be primitively rooted by definition, at every position $s+i$, $0 \leq i \leq (e2) \cdot p+t$ there is a primitively rooted square. This fact is based on a simple observation that a rotation of a primitive string is also primitive. In the algorithm, we have to consolidate the run from all of the primitively rooted squares encoded in the gap function. Thus, having identified a square, not only we must check if it can be extended left or right as a repetition, we have to check if it can be shifted left or right. Again, we are using *Refine*[] and *RStack*[] as auxiliary data structures to indicate which of the elements of the gap list had been previously processed as the part of tracing, so we do not process them again.

6 Conclusion

We present a new implementation of an extension of the Crochemore's repetitions algorithm that computes primitively rooted distinct squares, primitively rooted maximal repetitions, or runs. The running complexity of the original repetitions algorithm is preserved, and thus is $O(n \log n)$ where n is the length of the input string. In comparison to the previous implementation of the Crochemore's partitioning algorithm, the memory required is reduced to $13n$ integers. In comparison to the previous implementation of an extension to compute runs, there is no additional memory requited and no dynamic allocation or deallocation of the memory during the processing as

all the required memory is allocated once at the outset of the program. The resulting algorithm implemented in C++ is very fast and all the versions described in this paper can be downloaded from [20]. Since this report does not include bench-marking and comparisons with other runs algorithms, the future work must include the bench-marking and comparisons with the fastest algorithms [1,7] regardless their complexity, and, of course, with the known linear implementations.

References

1. G. Chen, S. J. Puglisi, and W. F. Smyth: *Fast and practical algorithms for computing all the runs in a string*, in Proc. the 18th annual symposium on Combinatorial Pattern Matching (CPM 2007), 2007, pp. 316–327.
2. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inform. Process. Lett., 12(5) 1981, pp. 297–315.
3. M. Crochemore, L. Ilie, and W. F. Smyth: *A simple algorithm for computing the lempel-ziv factorization*, in Proc. the 17th Data Compression Conference (DCC 2008), 2008, pp. 482–488.
4. F. Franek and M. Jiang: *Crochemore's repetitions algorithm revisited – computing runs.* to appear in Int. Jour. of Foundations of Comp. Sci.
5. F. Franek and M. Jiang: *Crochemore's repetitions algorithm revisited – computing runs*, AdvOL Report 2009/11, Advanced Optimization Laboratory, Dept. of Comp. and Software, McMaster University, 2009.
6. F. Franek, W. F. Smyth, and X. Xiao: *A note on Crochemore's repetitions algorithm, a fast space-efficient approach.* Nordic J. Computing, 10(1) 2003, pp. 21–28.
7. K. Hirashima, H. Bannai, W. Matsubara, A. Ishino, and A. Shinohara: *Bit-parallel algorithms for computing all the runs in a string*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 203–213.
8. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proc. 30th Internat. Colloq. on Automata, Languages & Programming (ICALP 2003), no. 2719 in LNCS, Springer, 2003, pp. 943–955.
9. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. 12th Symposium on Combinatorial Pattern Matching (CPM 2001), no. 2089 in LNCS, Springer, 2001, pp. 181–192.
10. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Linear-time construction of suffix arrays*, in Proc. the 14th Annual Conference on Combinatorial Pattern Matching (2003), no. 2676 in LNCS, Springer, 2003, pp. 186–199.
11. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proc. 14th Annual Symp. Combinatorial Pattern Matching, R. Baeza-Yates, E. Chàvez, and M. Crochemore, eds., no. 2676 in LNCS, Springer, 2003, pp. 200–210.
12. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1999, pp. 596–604.
13. R. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in Proc. 12th Intl. Symp. on Fund. of Comp. Sci. 1999, no. 1684 in LNCS, 1999, pp. 374–385.
14. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Maths. – Combinatorics and Complexity, 25(1–2) 1989, pp. 145–153.
15. M. G. Main and R. J. Lorentz: *An $O(n \log n)$ algorithm for finding all repetitions in a string.* J. Algorithms, 5(3) 1984, pp. 422–432.
16. G. Manzini: *Two space saving tricks for linear time LCP array computation*, in Proc. SWAT 2004, no. 3111 in LNCS, Springer, 2004, pp. 372–383.
17. G. Nong, S. Zhang, and W. H. Chan: *Linear time suffix array construction using d-critical substrings*, in Proc. 20th Combinatorial Pattern Matching (CPM 2009), Lille, France, June 2009, pp. 54–67.
18. S. Zhang and G. Nong: *Fast and space efficient linear suffix array construction*, in Proc. IEEE Data Compression Conference (IEEE DCC), IEEE Computer Society, March 2008, p. 553.
19. `http://www.csd.uwo.ca/faculty/ilie/runs.html`.
20. `http://www.cas.mcmaster.ca/~franek/research.html`.

Computing the Number of Cubic Runs in Standard Sturmian Words

Marcin Piątkowski[2⋆] and Wojciech Rytter[1,2]

[1] Department of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
rytter@mimuw.edu.pl
[2] Faculty of Mathematics and Informatics,
Nicolaus Copernicus University, Toruń, Poland
martinp@mat.umk.pl

Abstract. The *standard Sturmian* words are extensively studied in combinatorics of words. They are enough complicated to have many interesting properties and at the same time they are highly compressible. In this paper we present compact formulas for the number $\rho^{(3)}$ of cubic runs in any standard word. We show also that

$$\lim_{|w|\to\infty} \frac{\rho^{(3)}(w)}{|w|} = \frac{5\Phi+3}{13\Phi+9} \approx 0.36924841$$

and present the sequence of strictly growing standard words achieving this limit. The exact asymptotic ratio is here irrational, contrary to the situation of squares and runs in the same class of words. Furthermore we design an efficient algorithm computing the number of cubic runs in standard words in linear time with respect to the size of the compressed representation (recurrences) describing the word. The explicit size of the word can be exponential with respect to this representation. This is yet another example of a very fast computation on highly compressible texts.

Keywords: standard Sturmian words, repetitions, cubic runs, algorithms

1 Introduction

Repetitions in strings are important in combinatorics on words and many practical applications, see for instance [6], [11], [19] and [20]. The structure of repetitions is almost completely understood for the class of Fibonacci words, see [15], [17], [24], however it is not well understood for general words.

Runs are repetitions in which the period repeats at least twice. Highly repetitive segments, in which the repetitions ratio is at least 3, called the *cubic runs*, were introduced and studied in [10].

We say that a number i is a period of the word w if $w[j] = w[i+j]$ for all i with $i+j \leq |w|$. The minimal period of w will be denoted by $period(w)$. We say that a word w is periodic if $period(w) \leq \frac{|w|}{2}$. A word w is said to be *primitive* if w is not of the form z^k, where z is a finite word and $k \geq 2$ is a natural number.

A *maximal repetition* (a *run*, in short) in a word w is an interval $\alpha = [i..j]$ such that $w[i..j] = u^k v$ $(k \geq 2)$ is a nonempty periodic subword of w, where u is of the minimal length and v is a proper prefix (possibly empty) of u, that can not be extended (neither $w[i-1..j]$ nor $w[i..j+1]$ is a run with the period $|u|$). *Cubic runs*

⋆ The research supported by Ministry of Science and Higher Education of Poland, grant N N206 258035.

Marcin Piątkowski, Wojciech Rytter: Computing the Number of Cubic Runs in Standard Sturmian Words, pp. 106–120.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

are defined in the same way, but we require that the period repeat at least three times ($k \geq 3$).

A run α can be properly included as an interval in another run β, but in this case $period(\alpha) < period(\beta)$. The value of the run $\alpha = [i..j]$ is the factor $val(\alpha) = w[i..j]$. When it creates no ambiguity we identify sometimes run with its value and the period of the run $\alpha = [i..j]$ with the subword $w[i..period(w)]$ – called also the *generator* of the repetition. The meaning will be clear from the context. Observe that two different runs could correspond to the identical subwords, if we disregard their positions. Hence runs are also called the maximal *positioned* repetitions.

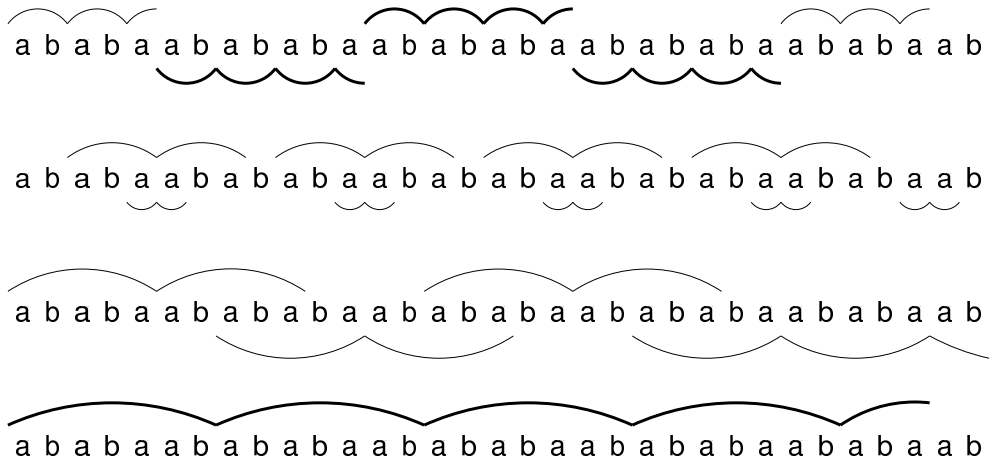


Figure 1. The structure of repetitions in the word $\mathrm{Sw}(1, 2, 1, 3, 1)$. There are 19 runs and 4 cubic runs (marked in **bold**).

Example 1. Let $w = abababaababababaababababaababababaab$.
There are 5 runs with the period $|a|$:

$$w[5..6] = a^2, \qquad w[12..13] = a^2, \qquad w[19..20] = a^2,$$
$$w[26..27] = a^2, \qquad w[31..32] = a^2,$$

5 runs with the period $|ab|$ (including 3 cubic runs):

$$w[1..5] = (ab)^2a, \qquad w[6..12] = (ab)^3a, \qquad w[13..19] = (ab)^3a,$$
$$w[20..26] = (ab)^3a, \qquad w[27..31] = (ab)^2a,$$

4 runs with the period $|aba|$:

$$w[3..8] = (aba)^2, \qquad w[10..15] = (aba)^2,$$
$$w[17..22] = (aba)^2, \qquad w[24..29] = (aba)^2,$$

4 runs with the period $|ababa|$:

$$w[1..10] = (ababa)^2, \qquad w[8..17] = (ababa)^2,$$
$$w[15..24] = (ababa)^2, \qquad w[22..33] = (ababa)^2ab,$$

and 1 (cubic) run with the period $|ababaab|$:

$$w[1..31] \;=\; (ababaab)^4aba.$$

All together we have 19 runs and 4 cubic runs, see Figure 1 for comparison.

Denote by $\rho(w)$ and $\rho^{(3)}(w)$ the number of runs and cubic runs in the word w, and by $\rho(n)$ and $\rho^{(3)}(n)$ the maximal number of runs and cubic runs in the words of length n respectively. The most interesting and open conjecture about maximal repetitions is:

$$\rho(n) \ < \ n.$$

In 1999 Kolpakov and Kucherov (see [16]) showed that the number $\rho(w)$ of runs in a string w is $O(|w|)$, but the exact multiplicative constant coefficient is still unknown. The best known results related to the value of $\rho(n)$ are

$$0.944575712\, n \ \leq \ \rho(n) \leq \ 1.029\, n.$$

The upper bound is by [8], [9] and the lower bound is by [13], [14], [18], [27]. The best known results related to $\rho^{(3)}(n)$ are (due to [10]):

$$0.41\, n \ \leq \ \rho^{(3)}(n) \ \leq \ 0.5\, n.$$

For the class $\mathcal{S}$ of standard Sturmian words there are known exact formulas for the number of runs and squares and their asymptotic behavior, see [2] and [22] for details. In this case we have

$$\lim_{n\to\infty} \frac{\rho(n)}{n} \ = \ 0.8.$$

This paper is devoted to the investigation of the structure of cubic runs in standard Sturmian words. We present the exact recurrence formulas for the number $\rho^{(3)}(w)$. Next we derive the algorithm computing $\rho^{(3)}(w)$ for any word $w \in \mathcal{S}$ in linear time with respect to the compressed representation of w, hence logarithmic time with respect to the length of the whole word w. We show also, that for any standard word w, we have

$$\rho^{(3)}(w_k) \ \leq \ 0.36924841\, |w|,$$

and construct the sequence $\{w_k\}$ of strictly growing standard words, for which we have

$$\lim_{k\to\infty} \frac{\rho^{(3)}(w_k)}{|w_k|} \ = \ \frac{5\Phi + 3}{13\Phi + 9} \ \approx \ 0.36924841.$$

Some useful applets related to problems considered in this paper can be found on the web site: `http://www.mat.umk.pl/~martinp/stringology/applets/`

2 Standard Sturmian words

Standard Sturmian words (standard words in short) are one of the most investigated class of strings in combinatorics on words, see for instance [1], [4], [5], [7], [19], [25], [26], [28] and references therein. They have very compact representations in terms of sequences of integers, which has many algorithmic consequences.

A *directive sequence* is an integer sequence: $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$, where $\gamma_0 \geq 0$ and $\gamma_i > 0$ for $i = 1, 2, \ldots, n$. The standard word corresponding to γ, denoted by $\mathrm{Sw}(\gamma)$, is described by the recurrences of the form:

$$\begin{aligned} x_{-1} &= b, & x_0 &= a, \\ x_1 &= x_0^{\gamma_0} x_{-1}, & x_2 &= x_1^{\gamma_1} x_0, \\ &\vdots & &\vdots \\ x_n &= x_{n-1}^{\gamma_{n-1}} x_{n-2}, & x_{n+1} &= x_n^{\gamma_n} x_{n-1}, \end{aligned} \tag{1}$$

where $\mathrm{Sw}(\gamma) = x_{n+1}$.

A sequence of words $\{x_i\}_{i=0}^{n+1}$ is called the standard sequence. Every word occurring in a standard sequence is a standard word, and every standard word occurs in some standard sequence. We assume that the standard word given by the empty directive sequence is a and $\mathrm{Sw}(0) = b$. The class of all standard words is denoted by $\mathcal{S}$.

Example 2. Consider the directive sequence $\gamma = (1, 2, 1, 3, 1)$. We have:

$$\begin{aligned} x_{-1} &= b \\ x_0 &= a \\ x_1 &= (x_0)^1 \cdot x_{-1} &&= a \cdot b \\ x_2 &= (x_1)^2 \cdot x_0 &&= ab \cdot ab \cdot a \\ x_3 &= (x_2)^1 \cdot x_1 &&= ababa \cdot ab \\ x_4 &= (x_3)^3 \cdot x_2 &&= ababaab \cdot ababaab \cdot ababaab \cdot ababa \\ x_5 &= (x_4)^1 \cdot x_3 &&= ababaababababaababababaababababa \cdot ababaab \end{aligned}$$

and finally

$$\mathrm{Sw}(1, 2, 1, 3, 1) \;=\; ababaababababaababababaababababaababaab.$$

The special case of standard words are Fibonacci words, which are formed by repeated concatenation in the same way that the Fibonacci numbers are formed by repeated addition. By definition the are given by directive sequences of the form $\gamma = (1, 1, \ldots, 1)$ (n-th Fibonacci word F_n corresponds to a sequence of n ones).

Observe that for $\gamma_0 > 0$ we have standard words starting with the letter a and for $\gamma_0 = 0$ we have standard words starting with the letter b. In fact the word $\mathrm{Sw}(0, \gamma_1, \ldots, \gamma_n)$ can be obtained from $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ by switching the letters a and b.

Observe also that for even $n > 0$ the standard word x_n has the suffix ba, and for odd $n > 0$ it has the suffix ab. Moreover, every standard word consists either of repeated occurrences of the letter a separated by single occurrences of the letter b or repeated occurrences of the letter b separated by single occurrences of the letter a. Those letters are called the *repeating letter* and the *single letter*, respectively. If the repeating letter is a (letter b respectively), the word is called the Sturmian word of the type a (type b respectively), see the definition 6.1.4 in [23] for comparison.

Remark 3. Without loss of generality we consider here the standard Sturmian words of the type a, therefore we assume that $\gamma_0 > 0$. The words of the type b can be considered similarly and all the results hold.

The number $N = |\mathrm{Sw}(\gamma)|$ is the (real) size of the word, while $(n+1) = |\gamma|$ can be thought as its compressed size. Observe that, by the definition of standard words, N is exponential with respect to n. Each directive sequence corresponds to a *grammar-based compression*, which consists in describing a given word by a context-free grammar G generating this (single) word. The size of the grammar G is the total length of all productions of G. In our case the size of the grammar is proportional to the length of the directive sequence.

3 Morphic reduction of standard words

The recurrent definition of standard words leads to the simple characterization by the composition of morphisms. Let $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a directive sequence. We associate with γ a sequence of morphisms $\{h_i\}_{i=0}^{n}$, defined as:

$$h_i : \begin{cases} a \longrightarrow a^{\gamma_i} b \\ b \longrightarrow a \end{cases} \qquad \text{for } 0 \le i \le n. \tag{2}$$

Lemma 4. *For $0 \le i \le n$ the morphism h_i transforms a standard word into another standard word, and we have:*

$$\begin{aligned} \mathrm{Sw}(\gamma_n) &= h_n(a), \\ \mathrm{Sw}(\gamma_i, \gamma_{i+1}, \ldots, \gamma_n) &= h_i\Big(\mathrm{Sw}(\gamma_{i+1}, \gamma_{i+2}, \ldots, \gamma_n)\Big). \end{aligned}$$

Proof. We will prove the above lemma by the induction on the length of the directive sequence. Recall that the standard word given by the empty directive sequence is a. For $|\gamma| = 1$ we have, by definition of standard words and the morphism h_n,

$$\mathrm{Sw}(\gamma_n) = a^{\gamma_n} b = h_n(a).$$

Assume now that $|\gamma| = k \ge 2$ and for directive sequences shorter than k the thesis holds. We have then:

$$\begin{aligned} \mathrm{Sw}(\gamma_i, \ldots, \gamma_n) &= [\mathrm{Sw}(\gamma_i, \ldots, \gamma_{n-1})]^{\gamma_n} \cdot \mathrm{Sw}(\gamma_i, \ldots, \gamma_{n-2}) \\ &\overset{ind.}{=} \Big[h_i\Big(\mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_{n-1})\Big)\Big]^{\gamma_n} \cdot h_i\Big(\mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_{n-2})\Big) \\ &= h_i\Big([\mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_{n-1})]^{\gamma_n} \cdot \mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_{n-2})\Big) \\ &= h_i\Big(\mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_n)\Big), \end{aligned}$$

which concludes the proof. □

Remark 5. As a direct conclusion from Lemma 4 we have that the standard word corresponding to the directive sequence $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$ is given as:

$$\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n) = h_0 \circ h_1 \circ \cdots \circ h_n(a). \tag{3}$$

The inverse morphism h_i^{-1} can be seen as a reduction of the word $\mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$ to the word $\mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_n)$ and allows us to reduce the computation of cubic runs in $\mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$ to the same computation in $\mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_n)$.

Denote by $|w|_a$ the number of occurrences of the letters a in the word w. We define the function, which will be useful in the rest of this paper. For a directive sequence $\gamma = (\gamma_0, \ldots, \gamma_n)$ and an integer $0 \leq k \leq n+1$ we define

$$N_\gamma(k) = |\mathrm{Sw}(\gamma_k, \gamma_{k+1}, \ldots, \gamma_n)|_a \,. \tag{4}$$

Moreover, for $k > n+1$, we define $N_\gamma(k) = 0$.

Remark 6. As a direct conclusion from the above definition, the equation (1) and the equation (2) we have that for $0 \leq k \leq n$ the numbers $N_\gamma(k)$ satisfy:

$$N_\gamma(k) = \gamma_k \, N_\gamma(k+1) + N_\gamma(k+2). \tag{5}$$

Example 7. Let $\gamma = (1, 2, 1, 3, 1)$ be a directive sequence. We have then

$$\begin{aligned}
\mathrm{Sw}(1,2,1,3,1) &= ababaababababaababababaababababaab & N_\gamma(0) &= 19,\\
\mathrm{Sw}(2,1,3,1) &= aabaaabaaabaaabaaba & N_\gamma(1) &= 14,\\
\mathrm{Sw}(1,3,1) &= abababaab & N_\gamma(2) &= 5,\\
\mathrm{Sw}(3,1) &= aaaba & N_\gamma(3) &= 4,\\
\mathrm{Sw}(1) &= ab & N_\gamma(4) &= 1,\\
\mathrm{Sw}(\varepsilon) &= a & N_\gamma(5) &= 1.
\end{aligned}$$

Remark 8. In case of Fibonacci words the numbers $N_\gamma(k)$ are Fibonacci numbers:

$$N_\gamma(k) \;=\; |F_{n-k-1}| \;=\; f_{n-k-1}. \tag{6}$$

4 Formulas for the number of cubic runs

In this section we present and prove formulas for the number of cubic runs in any standard word, that depend only on its compressed representation – the directive sequence. The following zero-one functions for testing the parity of a nonnegative integer i will be useful to simplify those formulas:

$$\mathit{even}(i) \;=\; \begin{cases} 1 & \text{for even } i \\ 0 & \text{for odd } i \end{cases} \qquad \text{and} \qquad \mathit{odd}(i) \;=\; \begin{cases} 1 & \text{for odd } i \\ 0 & \text{for even } i \end{cases}.$$

We begin with the characterization of possible periods of cubic runs in standard words. The following lemma is a consequence of the very special structure of subword graphs (especially their compacted versions) of those words. See [3] and [25] for more information.

Lemma 9. *The period of each cubic run in the standard word* $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ *is of the form* x_i*, where* x_i*'s are as in equation (1).*

To prove the above lemma it is sufficient to show that no factor of the word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$, that does not satisfy the condition given there, could be the generator of a cubic run. We can use similar argumentation as in proof of Theorem 1 in [12]. The details are omitted in this version.

The main idea of the computation of cubic runs in a standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ is the partition of them into three separate categories depending on the length of their periods. We say that a cubic run is:

short – if it has the period of the form a or $a^k b$,
medium – if it has the period of the form x_2,
large – if it has the period of the form x_i, for $i > 2$.

Denote by $\rho_S^{(3)}(w)$, $\rho_M^{(3)}(w)$ and $\rho_L^{(3)}(w)$ the number of short, medium and large cubic runs in the word w, respectively. We will consider each type separately.

Example 10. Recall the word $w = \mathrm{Sw}(1, 2, 1, 3, 1)$ from Example 1. We have:

– 3 short cubic runs (period ab),
– no medium cubic run,
– 1 large cubic run (period $ababaab$),

see Figure 1 for comparison.

4.1 Short runs

We start with the computation of the *short* cubic runs. These are the cubic runs with the periods of the form a or $a^k b$. Their number depends on the values of γ_0 and γ_1.

Lemma 11. *The number $\rho_{S_1}^{(3)}$ of cubic runs with the period a in the standard word $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ equals:*

$$\rho_{S_1}^{(3)}(w) \;=\; \begin{cases} 0 & \text{for } \gamma_0 = 1 \\ N_\gamma(2) - odd(n) & \text{for } \gamma_0 = 2 \\ N_\gamma(1) & \text{for } \gamma_0 > 2 \end{cases} . \tag{7}$$

Proof. First assume that $\gamma_0 > 2$. Every cubic run with the period a in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ equals a^{γ_0} or a^{γ_0+1} and is followed by the single letter b. Due to Lemma 4 every such cubic run in $\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ corresponds to the letter a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$. Hence in this case we have $N_\gamma(1)$ cubic runs with the period a.

Assume now that $\gamma_0 = 2$. In this case the word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ consists of the blocks of the two types: aab and $aaab$. Only the blocks of the second type include the cubic run with the period a. Due to Lemma 4 every such cubic run in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ corresponds to the letter b followed by the letter a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$. Hence the number of such cubic runs equals the number of blocks ba in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$.

Recall that for an even length of the directive sequence $|(\gamma_1, \ldots, \gamma_n)|$ (n is even) the word $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ ends with ba and in this case the number of cubic runs with the period a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ equals the number of the letters b in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$, namely $N_\gamma(2)$. For an odd length of the directive sequence $|(\gamma_1, \ldots, \gamma_n)|$ (n is odd) the word $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ ends with ab and the last letter b does not correspond to a cubic run in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$. In this case the number of runs with the period a in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ is one less than the number of the letters b in the word $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$, namely $N_\gamma(2) - 1$.

Finally assume that $\gamma_0 = 1$. In this case the word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ consists of the blocks of the two types: ab and aab. None of them includes a cubic run with the period a, and this completes the proof. □

Lemma 12. *The number* $\rho^{(3)}_{S_2}$ *of cubic runs with the period* $a^k b$ *in the standard word* $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ *equals:*

$$\rho^{(3)}_{S_2}(w) \;=\; \begin{cases} 0 & \text{for } \gamma_1 = 1 \\ N_\gamma(3) - even(n) & \text{for } \gamma_1 = 2 \\ N_\gamma(2) & \text{for } \gamma_1 > 2 \end{cases}. \tag{8}$$

Proof. Notice that, due to equation (2) and Lemma 4, cubic runs with the periods of the form $a^k b$ in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ correspond to cubic runs with the period a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$. Similar reasoning as above establishes the desired formula. □

4.2 Medium runs

Recall that a cubic run is called *medium* if it has the period of the form x_2. Observe that medium cubic runs appear in standard words generated by directive sequences of the length at least 3. We have to consider two cases: the directive sequences of the length 3 and the longer directive sequences. The values of γ_0 and γ_1 does not affect the number of medium cubic runs, hence to simplify the calculations we can assume in further proofs that $\gamma_0 = \gamma_1 = 1$.

We start with counting medium runs in standard words generated by directive sequences of the length greater than 3.

Lemma 13. *Let* $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ *be a standard word and* $n \geq 3$*. The number of medium cubic runs in* w *equals:*

$$\rho^{(3)}_{M}(w) \;=\; \begin{cases} N_\gamma(4) - 1 & \text{for } \gamma_2 = 1 \\ N_\gamma(3) & \text{for } \gamma_2 \geq 2 \end{cases}. \tag{9}$$

Proof. We start with the assumption that $\gamma_2 > 2$. In this case every factor of the form $x_3 = x_2^{\gamma_2} x_1$ includes one cubic runs with the period x_2. Hence the number of such cubic runs equals the number of factors x_3 in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$, namely $N_\gamma(3)$ (due to Lemma 4).

Assume now that $\gamma_2 = 2$. The word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ can be represented as a sequence of concatenated words x_3 and x_2 and has the form:

$$x_3^{\alpha_1} x_2 x_3^{\alpha_2} x_2 \cdots x_3^{\alpha_s} x_2 x_3 \qquad \text{or} \qquad x_3^{\beta_1} x_2 x_3^{\beta_2} x_2 \cdots x_3^{\beta_s} x_2.$$

Observe that $x_3 = x_2 x_2 x_1$ and every occurrence of x_3 in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ either follows some occurrence of x_2 or is followed by some occurrence of x_2. In the first case we have $x_2 \cdot x_3 = x_2 \cdot x_2 x_2 x_1$ and there is a cubic run with period x_2. In the second case we have $x_3 \cdot x_2 = x_2 x_2 x_1 \cdot x_2$, and there is also a cubic run with period x_2, since x_1 is a prefix of x_2. Therefore the number of medium cubic runs in this case equals the number of the factors x_3 in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$, namely $N_\gamma(3)$.

Finally assume that $\gamma_2 = 1$. The word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ can be represented as a sequence of concatenated words x_3 and x_4 and has the form:

$$x_4^{\alpha_1} x_3 x_4^{\alpha_2} x_3 \cdots x_4^{\alpha_s} x_3 x_4 \qquad \text{or} \qquad x_4^{\beta_1} x_3 x_4^{\beta_2} x_3 \cdots x_4^{\beta_s} x_3.$$

We have $x_3 = x_2 x_1$ and $x_4 = x_2 x_1 \cdots x_2 x_1 \cdot x_2$. Therefore only the last one occurrence of x_4 in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ does not correspond to a cubic run with the period x_2 and we have $N_\gamma(4) - 1$ such cubic runs in this case. This completes the proof. □

Lemma 14. *The number of medium cubic runs in the word* $w = \mathrm{Sw}(\gamma_0, \gamma_1, \gamma_2)$ *equals:*

$$\rho_M^{(3)}(w) \;=\; \begin{cases} 1 & \text{for } \gamma_2 > 2 \\ 0 & \text{for } \gamma_2 \leq 2 \end{cases}. \tag{10}$$

Proof. We have $\mathrm{Sw}(\gamma_0, \gamma_1, \gamma_2) = x_2^{\gamma_2} x_1$. Hence there is only one medium run (with the period x_2) if $\gamma_2 > 2$ and no medium run otherwise. □

4.3 Large runs

Recall that a cubic run is called *large* if it has the period of the form x_i for $i > 2$, where x_i are as in the equation (1). We reduce the problem of counting large cubic runs to the one for counting medium cubic runs, using morphic representation of standard words introduced in previous section.

Let h be a morphism, $v = a_1 a_2 \cdots a_k$ be a word of the length k and let $w = h(v)$. The morphism h defines the partition of w into segments $h(a_1)$, $h(a_2)$,..., $h(a_t)$. These segments are called the *h-blocks*. We say that a factor x of the word w is *synchronized* with the morphism h in w if and only if each occurrence of x in w starts at the beginning of some h-block and ends at the end of some h-block. Observe that every factor in w that is synchronized with h corresponds to some factor in v, hence the morphism h preserves the structure of the factors that are synchronized with it.

Example 15. Let $w = \mathrm{Sw}(1, 2, 1, 3, 1)$ and $v = \mathrm{Sw}(2, 1, 3, 1)$ be standard words and h_0 be a morphism defined as:

$$h_0 : \begin{cases} a \longrightarrow ab \\ b \longrightarrow a \end{cases}.$$

Recall that

$$\mathrm{Sw}(1, 2, 1, 3, 1) = h_0(\mathrm{Sw}(2, 1, 3, 1)),$$

$$\mathrm{Sw}(1, 2, 1, 3, 1) = abababaababaababababaababababaababaab,$$

$$\mathrm{Sw}(2, 1, 3, 1) = aabaaabaaabaaabaaba.$$

The factors $w[6..8] = aba$ and $w[13..17] = abaab$ are not synchronized with h_0, because both of them end in the middle some h_0-block. From the other hand, the factor $w[22..28] = ababaab$ and all its occurrences in w (namely $w[1..7]$, $w[8..14]$ and $w[15..21]$) start at the beginning of some h_0-block and end at the end of some h_0-block. Hence the factor $w[22..28]$ is synchronized with the morphism h_0. Moreover it corresponds with the factor $v[13..16] = aaba$, see Figure 2 for comparison.

Lemma 16. *The periods of* large *cubic runs in the standard word* $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ *are synchronized with the morphism* h_0.

Proof. Let h_0 be the morphism defined as

$$h_0 : \begin{cases} a \longrightarrow a^{\gamma_0} b \\ b \longrightarrow a \end{cases}.$$

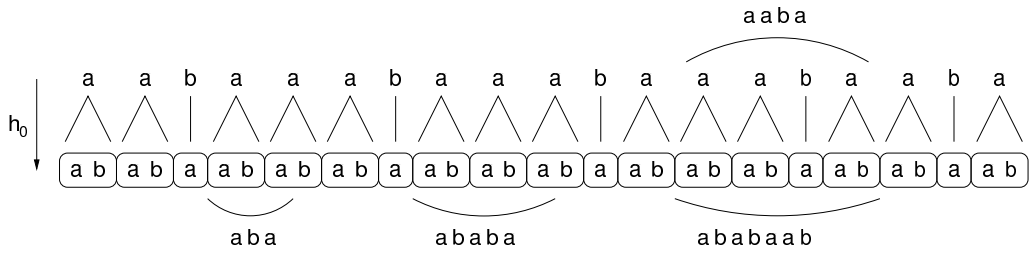


Figure 2. The factors *aba* and *ababa* do not synchronize with the morphism h_0 in the word $\mathrm{Sw}(1,2,1,3,1)$, while the factor *abababaab* (in fact the period of the large cubic run) is synchronized with h_0 and corresponds to the factor *aaba* in the word $\mathrm{Sw}(2,1,3,1)$.

Due to Lemma 4 we have $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n) = h_0(\mathrm{Sw}(\gamma_1,\ldots,\gamma_n))$. Moreover, h_0 determines the partition of $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ into h_0-blocks of the form $a^{\gamma_0}b$ and a, see Figure 2 for the partition of $\mathrm{Sw}(1,2,1,3,1)$.

Recall that the period of each large cubic run in $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ is of the form x_i, where $i \geq 3$. By definition of standard words the factor x_i starts with $a^{\gamma_0}b$, hence at the beginning of some h_0-block.

For odd $i \geq 3$ the subword x_i ends with $x_1 = a^{\gamma_0}b$, hence at the end of some h_0-block, and is obviously synchronized with h_0.

For even $i \geq 3$ the factor x_i ends with

$$x_3 \cdot x_2 \;=\; x_2^{\gamma_2} x_1 \cdot x_1^{\gamma_1} x_0 \;=\; x_2^{\gamma_2} \cdot (a^{\gamma_0}b)^{\gamma_1+1}a.$$

First assume that x_i is followed by the block $a^{\gamma_0}b$. The single letter a at the end of x_i is then the whole h_0-block and x_i is synchronized with the morphism h_0.

Assume now that x_i ends with $(a^{\gamma_0}b)^{\gamma_1+1}a$ and is followed by $(a^{\gamma_0-1}b)$, namely it ends in the middle of some h_0-block. In this case we have the occurrence of the factor $(a^{\gamma_0}b)^{\gamma_1+2}$ in $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$, which is reduced by the morphism h_0^{-1} to the factor $a^{\gamma_1+2}b$ in $\mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$. By definition, the standard word $\mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$ can include only the blocks of the two types: the short block – $a^{\gamma_1}b$ and the long block – $a^{\gamma_1+1}b$, hence we have the contradiction and the proof is complete. □

The following lemma, which is a direct conclusion from Lemma 16, allows us to reduce the problem of counting large cubic runs in $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ to counting large and medium cubic runs in $\mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$.

Lemma 17. *Let* $w = \mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ *and* $v = \mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$ *be standard words. The number of* large *cubic runs in* w *is given by the recurrence*

$$\rho_L^{(3)}(w) \;=\; \rho_L^{(3)}(v) \;+\; \rho_M^{(3)}(v).$$

Proof. Lemma 16 implies that the morphism defined in the equation (2) preserves the structure of long cubic runs in standard words. Recall that the word $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ is reduced by h_0^{-1} to the word $\mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$. Therefore, every large cubic run α in $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ corresponds to some cubic run β in $\mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$.

Due to Lemma 9 the period of the cubic run α is of the form x_i, where $i \geq 3$. The corresponding cubic run β is either large (for $i > 3$) or medium (for $i = 3$). Hence to compute all large cubic runs in $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ it is sufficient to compute all large and medium cubic runs in $\mathrm{Sw}(\gamma_1,\ldots,\gamma_n)$. □

The following theorem summarizes all the formulas developed above.

Theorem 18. *Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence, $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ and $w_i = \mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$, for $0 \leq i \leq n$, be standard words. The number of cubic runs in w is given as:*

$$\rho^{(3)}(w) \;=\; \rho^{(3)}_{S_1}(w) \;+\; \rho^{(3)}_{S_1}(w) \;+\; \sum_{i=0}^{n-2} \rho^{(3)}_{M}(w_i). \tag{11}$$

Proof. The thesis of the theorem follows by combining the formulas (7), (8), the formula (9) repeated $n-2$ times, and finally the formula (10). □

Example 19. Consider a directive sequence $\gamma = (1, 2, 1, 3, 1)$. We compute the number of cubic runs in $\mathrm{Sw}(1, 2, 1, 3, 1)$ using the formulas mentioned above. We have:

short cubic runs with period a:	0
short cubic runs with period $a^k b$:	$\lvert aaaba\rvert_a - 1 = 3$
medium cubic runs:	$\lvert ab\rvert_a - 1 = 0$
large cubic runs:	$\rho^{(3)}_M(2,1,3,1) + \rho^{(3)}_M(1,3,1) = \lvert ab\rvert_a + 0 = 1$

Altogether there are 4 cubic runs, see Example 1 and Figure 1 for comparison.

4.4 Algorithm for computation of the number of cubic runs

The formulas investigated above allow us to develop an efficient algorithm computing the number of cubic runs in any standard Sturmian word.

Theorem 20. *Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence and $\mathrm{Sw}(\gamma)$ be a standard word. We can count the number of cubic runs in $\mathrm{Sw}(\gamma)$ in linear time with respect to the length of the directive sequence $|\gamma|$ (logarithmic time with respect to the length of the whole word $|\mathrm{Sw}(\gamma)|$).*

Proof. The formulas (7), (8), (9) and (10) for the number of cubic runs in a standard word $\mathrm{Sw}(\gamma)$ depend directly on the components of the directive sequence γ and the numbers $N_\gamma(k)$. We can compute the numbers $N_\gamma(n)$, $N_\gamma(n-1)$, …, $N_\gamma(1)$ by consecutive iteration of the equation (5). In each step i of the computation we remember the number of cubic runs related to the value of the γ_i. The number of iterations performed by the algorithm correspond directly to the length of the directive sequence, hence it has the time complexity $\mathrm{O}(|\gamma|)$. See Algorithm 1 for more details. □

5 Asymptotic behaviour of the number of cubic runs

This section is devoted to the computation of the asymptotic limit

$$\lim_{|w|\to\infty} \frac{\rho^{(3)}(w)}{|w|} \tag{12}$$

for $w \in \mathcal{S}$.

Algorithm 1: *Counting-Cubic-Runs*(Sw(γ))

```
1  (x, y, cr) ⟵ (1, 0, 0);
2  if γ_n > 2 then cr ⟵ cr + 1;
3  for i := n to 3 do
4  |   (x, y) ⟵ (γ_i · x + y, x);
5  |   if γ_{i−1} ≥ 2 then cr ⟵ cr + x;
6  |   else cr ⟵ cr + y − 1;
7  if γ_1 = 2 then
8  |   cr ⟵ cr + x;
9  |   if n is even then cr ⟵ cr − 1;
10 (x, y) ⟵ (γ_2 · x + y, x);
11 if γ_1 > 2 then cr ⟵ cr + x;
12 if γ_0 = 2 then
13 |   cr ⟵ cr + x;
14 |   if n is odd then cr ⟵ cr − 1;
15 if γ_0 > 2 then cr ⟵ cr + γ_1 · x + y;
16 return cr;
```

Theorem 21. *Let $w = \mathrm{Sw}(\gamma)$ be a standard word. Then we have*

$$\rho^{(3)}(w) \leq 0.36924841\,|w|.$$

Moreover there is infinite and strictly growing sequence of standard words achieving this asymptotic limit.

Proof. To prove the above theorem we will construct a directive sequence corresponding to a standard word for which the number of cubic runs will be maximal in relation to their length.

Let $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma)$ be a standard word. The number of cubic runs with the period of the form a in w corresponds directly to the value γ_0, see equation (7). The word w consists of blocks of the two types: $a^{\gamma_0}b$ and $a^{\gamma_0+1}b$. For $\gamma_0 \geq 3$ every such block contains a desired cubic run, for $\gamma_0 = 2$ only the second type of blocks contains a short cubic run, and for $\gamma_0 < 2$ there is no short cubic run in w. Moreover, for $\gamma_0 > 3$ the number of considered cubic runs does not change while the length of the word increases significantly.
For $\gamma_0 = 2$ we have, by the equations (5) and (7):

$$|w| = (\gamma_1 + 1)\,N_\gamma(2) + 3\,\gamma_1\,N_\gamma(3) \qquad \text{and} \qquad \rho^{(3)}_{S_1}(w) = N_\gamma(2) \pm 1,$$

and for $\gamma_0 = 3$ we have:

$$|w| = (4\,\gamma_1 + 1)\,N_\gamma(3) + 4\,N_\gamma(3) \qquad \text{and} \qquad \rho^{(3)}_{S_1}(w) = \gamma_1\,N_\gamma(2) + N_\gamma(3).$$

Therefore for the change of the value γ_0 from 2 to 3 the increase of $\rho^{(3)}_{S_1}$ (namely: $(\gamma_1 - 1)\,N_\gamma(2) + N_\gamma(3)$) is significant in relation to the increase of the length of the whole word (namely: $\gamma_1\,N_\gamma(2) + \gamma_1\,N_\gamma(3)$). Hence $\gamma_0 = 3$ is the optimal value. It does

not affect the number of cubic runs with longer periods, hence we assume in further discussion its optimal value.

The number of cubic runs with the period of the form $a^k b$ in w depends on the value of γ_1, see equation (8). Similar argumentation as above shows that γ_1 must be greater than 1 and no more than 3. For $\gamma_1 = 2$ we have, by the equations (5) and (8):

$$|w| = (9\gamma_2 + 4)N_\gamma(3) + 9N_\gamma(4) \qquad \text{and} \qquad \rho^{(3)}_{S_2}(w) = N_\gamma(3) \pm 1,$$

and for $\gamma_1 = 3$ we have:

$$|w| = (13\gamma_2 + 4)N_\gamma(3) + 13N_\gamma(4) \qquad \text{and} \qquad \rho^{(3)}_{S_2}(w) = \gamma_2 N_\gamma(3) + N_\gamma(4).$$

Therefore the change of the value of γ_1 from 2 to 3 increases the number of cubic runs by: $(\gamma_2 - 1)N_\gamma(3) + N_\gamma(4) \pm 1$ and at the same time increases the length of the word by: $4\,\gamma_2\,N_\gamma(3) + 4\,N_\gamma(4)$. Hence we conclude that $\gamma_1 = 2$ is the optimal value.

The number of medium cubic runs in the word w corresponds to the value of γ_2. It is easy to see that γ_2 must be at most 2, otherwise the length of the word increases significantly and the value $\rho^{(3)}_M(w)$ does not change.
For $\gamma_2 = 1$ we have, by the equations (5) and (9):

$$|w| = (13\gamma_3 + 9)N_\gamma(4) + 13N_\gamma(5) \qquad \text{and} \qquad \rho^{(3)}_M(w) = N_\gamma(4) - 1.$$

and for $\gamma_2 = 2$ we have:

$$|w| = (22\gamma_3 + 9)N_\gamma(4) + 22N_\gamma(5) \qquad \text{and} \qquad \rho^{(3)}_M(w) = \gamma_3 N_\gamma(4) + N_\gamma(5).$$

Therefore the change of the value of γ_2 from 1 to 2 increases the number of cubic runs by: $(\gamma_3 - 1)\,N_\gamma(4) + N_\gamma(5) + 1$ and at the same time increases the length of the word by: $9\,\gamma_3\,N_\gamma(4) + 9\,N_\gamma(5)$. Hence we conclude that $\gamma_2 = 1$ is the optimal value.

We compute large cubic runs by reduction of them to medium runs, see Lemma 17. Applying $n-2$ times the above argumentation for the medium cubic runs we conclude that optimal value of γ_3, γ_4, …, γ_{n-1} is also 1. Similarly for $\gamma_n > 1$ there is one additional long run whereas the length of the word increases more than two times.

We have shown above, that the maximal value of the quotient of the number of cubic runs to the length of the word is achieved by the standard words generated by directive sequences of the form $\gamma = (3, 2, 1, 1, \ldots, 1)$. Now we are ready to compute the value of the asymptotic limit from the equation (12).

Let us consider a sequence of standard words:

$$w_k \;=\; (3, 2, \underbrace{1, 1, 1, \ldots, 1}_{k}).$$

We have by definition of standard words and Remark 8:

$$|w_k| \;=\; 13\,N_\gamma(3) + 9\,N_\gamma(4) \;=\; 13\,f_{k-1} + 9\,f_{k-2},$$

and by Theorem 18 and Remark 8:

$$\rho^{(3)}(w_k) \;=\; 5\,N_\gamma(3) + 3\,N_\gamma(4) - k \pm 1 \;= 5\,f_{k-1} + 3\,f_{k-2}.$$

We have also that:

$$\lim_{n\to\infty} \frac{f_n}{f_{n-1}} \;=\; \Phi \;\approx 1.61803390,$$

hence

$$\lim_{k\to\infty} \frac{\rho^{(3)}(w_k)}{|w_k|} \;\approx\; \frac{5\Phi + 3}{13\Phi + 9} \;\approx\; 0.36924841,$$

and this completes the proof. □

References

1. J. Allouche and J. Shallit: *Automatic Sequences. Theory, Applications, Generalizations.*, Cambridge University Press, 2003.
2. P. Baturo, M. Piątkowski, and W. Rytter: *The number of runs in Sturmian words*, in Proceedings of the 13th international conference on Implementation and Applications of Automata, vol. 5148 of Lecture Notes in Computer Science, Springer, 2008, pp. 252–261.
3. P. Baturo, M. Piątkowski, and W. Rytter: *Usefulness of directed acyclic subword graphs in problems related to standard Sturmian words.* International Journal of Foundations of Computer Science, 20(6) 2009, pp. 1005–1023.
4. P. Baturo and W. Rytter: *Compressed string-matching in standard Sturmian words.* Theoretical Computer Science, 410(30–32) 2009, pp. 2804–2810.
5. J. Berstel: *Sturmian and Episturmian words: a survey of some recent results*, in Proceedings of the 2nd international conference on Algebraic informatics, vol. 4728 of Lecture Notes in Computer Science, Springer, 2007, pp. 23–47.
6. J. Berstel and J. Karhumaki: *Combinatorics on words: a tutorial.* Bulletin of the EATCS, 79 2003, pp. 178–228.
7. J. Berstel, A. Lauve, C. Reutenauer, and F. Saliola: *Combinatorics on Words: Christoffel Words and Repetitions in Words*, CRM monograph series, Providence, R.I: American Mathematical Society, 2009.
8. M. Crochemore and L. Ilie: *Analysis of maximal repetitions in strings*, in Proceedings of the 32nd International Conference on Mathematical Foundations of Computer Science, vol. 4708 of Lecture Notes in Computer Science, Springer, 2007, pp. 465–476.
9. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution of the "runs" conjecture*, in Proceedings of the 19th annual symposium on Combinatorial Pattern Matching, vol. 5029 of Lecture Notes in Computer Science, Springer, 2008, pp. 290–302.
10. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *On the maximal number of cubic runs in a string*, in Proceedings of the International Conference on Implementation and Applications of Automata, 2010, pp. 227–238.
11. M. Crochemore and W. Rytter: *Jewels of Stringology: text algorithms*, World Scientific, 2003.
12. D. Damanik and D. Lenz: *Powers in Sturmian sequences.* European Journal of Combinatorics, 24(4) 2003, pp. 377–390.
13. F. Franek, R. J. Simpson, and W. F. Smyth: *The maximum number of runs in a string*, in Proceedings of 14th Australian Workshop on Combinatiorial Algorithms, 2003, pp. 26–35.
14. F. Franek and Q. Yang: *An asymptotic lower bound for the maximal number of runs in a string.* Jnternational Journal of Foundations of Computer Science, 19(1) 2008, pp. 195–203.
15. C. S. Iliopoulos, D. Moore, and W. F. Smyth: *A characterization of the squares in a Fibonacci string.* Theoretical Computer Science, 172(1–2) 1997, pp. 281–291.
16. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1999, pp. 596–604.
17. R. M. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in Proceedings of 12th International Symposium on Fundamentals of Computation Theory, vol. 1684 of Lecture Notes in Computer Science, Springer, 1999, pp. 374–385.
18. K. Kusano, W. Matsubara, A. Ishino, H. Bannai, and A. Shinobara: *New lower bound for the maximum number of runs in a string.* Computing Research Repository, abs/0804.1214 2008.
19. M. Lothaire: *Algebraic Combinatorics on Words*, vol. 90 of Encyclopedia of mathematics and its application, Cambridge University Press, 2002.
20. M. Lothaire: *Applied Combinatorics on Words*, vol. 105 of Encyclopedia of Mathematics and its Application, Cambridge University Press, 2005.
21. M. Piątkowski: *Stringological applets* . http://www.mat.umk.pl/~martinp/stringology/applets.
22. M. Piątkowski and W. Rytter: *Asymptotic behaviour of the maximal number of squares in standard Sturmian words*, in Proceedings of the 14-th Prague Stringology Conference, Czech Technical University, 2009, pp. 237–248, accepted to International Journal of Foundations of Computer Science.

23. N. Pytheas Fogg: *Substitutions in Dynamics, Arithmetics and Combinatorics*, vol. 1794 of Lecture Notes in Mathematics, Springer, 2002.
24. W. Rytter: *The structure of subword graphs and suffix trees of Fibonacci words.* Theoretical Computer Science, 363(2) 2006, pp. 211–223.
25. M. Sciortino and L. Zamboni: *Suffix automata and standard Sturmian words*, in Proceedings of the 11th International Conference on Developments in Language Theory, vol. 4588 of Lecture Notes in Computer Science, Springer, 2007, pp. 382–398.
26. J. Shallit: *Characteristic words as fixed points of homomorphisms*, Tech. Rep. CS-91-72, University of Waterloo, Department of Computer Science, 1991.
27. J. Simpson: *Modified Padovan words and the maximum number of runs in a word.* Australian Journal of Combinatorics, 46 2010, pp. 129–145.
28. H. Uscka-Wehlou: *Digital lines, Sturmian words, and continued fractions*, PhD thesis, Department of Mathematics, Upspsala University, 2009.

Inferring Strings from Suffix Trees and Links on a Binary Alphabet

Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819–0395, Japan
{tomohiro.i,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. A suffix tree, which provides us with a linear space full-text index of a given string, is a fundamental data structure for string processing and information retrieval. In this paper we consider the reverse engineering problem on suffix trees: Given an unlabeled ordered rooted tree T accompanied with a node-to-node transition function f, infer a string whose suffix tree and its suffix links for inner nodes are isomorphic to T and f, respectively. By introducing new characterizations of suffix trees, we show that the reverse engineering problem on suffix trees on a binary alphabet can be solved in linear time in the input size.

1 Introduction

1.1 Suffix Trees

Suffix trees, one of the most well known and widely-used text indexing structures, have played a central role in combinatorial pattern matching and its applications. A multitude of important problems can efficiently be solved using suffix trees [1,10].

A suffix tree of a string w is a compacted trie which represents all the suffixes of w. Each edge of the suffix tree is labeled with a substring y of w, and the edge string y is represented by a pair (i, j) of positions such that the substring of w that begins at position i and ends at position j is identical to y. In this way the suffix tree can be represented with linear space in the length of w. Linear-time suffix tree construction algorithms proposed in [19,15,18] utilize auxiliary edges called *suffix links*. There exists a suffix link from node v to node u if the substring represented by u is identical to the string that is obtained by removing the first character of the substring represented by v.

1.2 Our Contribution

We consider the reverse engineering problem on suffix trees, i.e., given an ordered rooted tree T with its edges *unlabeled*, determine whether there exists a string w such that the edge-unlabeled suffix tree of w is isomorphic to T. If one exists, then output such a string. We emphasize that this problem is very challenging, intuitively, due to the following reasons:

- The length of each edge string is not given.
- The mapping from strings to edge-unlabeled suffix trees is not injective.

As a first step towards solving the problem, we restrict the alphabet to a binary one. Also, we assume that suffix links of inner nodes are given as input. We show that, with these conditions, we can solve the reverse engineering problem on suffix trees in linear time in the size of the input tree T. We remark that if suffix links of leaves are also given, then the problem can be easily solved in linear time. However, it is much more difficult to reverse engineer a string only from suffix links of inner nodes.

Tomohiro I, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Inferring Strings from Suffix Trees and Links on a Binary Alphabet, pp. 121–131.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

1.3 Related Work

Inferring a string from other string data structures has been widely studied. An algorithm to find a string having a given border array was presented in [8], which runs in linear time for an unbounded alphabet. A simpler linear-time solution for the same problem for a bounded alphabet was shown in [5]. Linear-time and $O(n^{1.5})$-time inferring algorithms for parameterized versions of border arrays, on a binary alphabet and an unbounded alphabet, respectively, were proposed [11,13]. Linear-time inferring algorithms for suffix arrays [7,2], KMP failure tables [6,9], prefix tables [3], cover arrays [4], palindromic structures [12], directed acyclic word graphs [2] and directed acyclic subsequence graphs [2] have been proposed, which provide us with further insight concerning the data structures. Also, it was recently revealed that the time complexity of reverse problem of runs depends on the alphabet size [14].

Counting and enumerating some of the above-mentioned data structures have also been studied in the literature [16,17,11,13,12].

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i:j] = \varepsilon$ if $j < i$. For any characters $a, b \in \Sigma$, we write as $a \prec b$ if a is lexicographically smaller than b.

Let $T = (V_T, E_T)$ be an ordered rooted tree. The root node of T is denoted by $\bot_T$. V_T^{in} and V_T^{leaf}, respectively, denote the set of the inner nodes and the set of the leaf nodes of T. For any $v \in V_T$, let $V_T(v) = V_{T'}$, $V_T^{in}(v) = V_{T'}^{in}$ and $V_T^{leaf}(v) = V_{T'}^{leaf}$, where T' is the subtree of T rooted at v. For any $v \in V_T$, the set of children of v, the i-th child of v and the parent of v are denoted by $\mathsf{children}(v)$, $\mathsf{child}_i(v)$ and $\mathsf{par}(v)$, respectively.

The *suffix tree* of a string w, denoted by $ST(w)$, is a compacted trie which represents all the suffixes of w. Let us assume that w ends with a terminal symbol $\$ \notin \Sigma$, where $\$$ is lexicographically smaller than any character in Σ, so that $ST(w)$ has exactly $|w|$ leaves. Every edge $e \in E_{ST(w)}$ is labeled with a substring of w. We call it the *edge string* of e. For any $v \in V_{ST(w)}^{in}$, all edge strings coming out from v must begin with distinct characters, and the children of v are sorted in lexicographic order of edge strings, namely, for any $1 \leq i < |\mathsf{children}(v)|$, the first symbol of the edge string for $(v, \mathsf{child}_i(v))$ must be lexicographically smaller than that for $(v, \mathsf{child}_{i+1}(v))$. Every node $v \in V_{ST(w)}$ corresponds to a string obtained by concatenating edge strings on the path from $\bot_{ST(w)}$ to v.

The *suffix link* $sl_w : V_{ST(w)} \to (V_{ST(w)} \cup \{\top\})$ of $ST(w)$ is a function such that for any $v \in (V_{ST(w)} - \{\bot_{ST(w)}\})$ that corresponds to a string x, $sl_w(v) = u$ where u is the node that corresponds to $x[2:|x|]$. For the root node $\bot_{ST(w)}$, let $sl_w(\bot_{ST(w)}) = \top$, where $\top$ is an auxiliary node.

Suffix tree $ST(w)$ for string $w = \mathtt{ababaaa\$}$ is shown in Figure 1. An auxiliary node $\top$ is abbreviated in the figure.

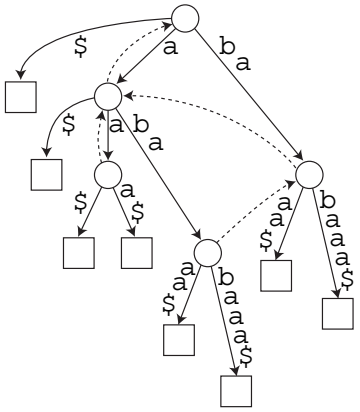


Figure 1. Suffix tree $ST(w)$ and for string $w =$ `ababaaa$`. Suffix links of inner nodes are shown by dotted arrows.

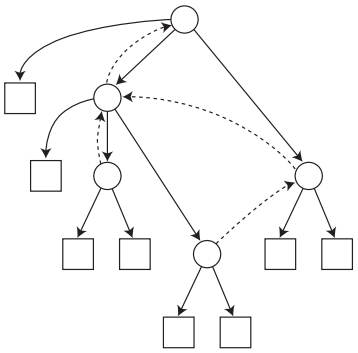


Figure 2. A valid input of Problem 1.

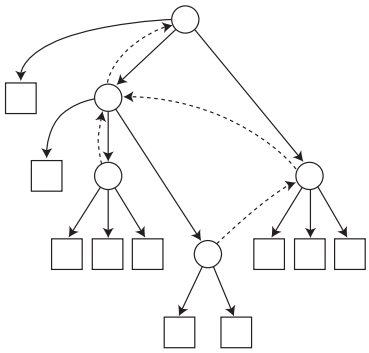


Figure 3. An invalid input of Problem 1.

In this paper, we tackle the following problem:

Problem 1 (Reverse Engineering Problem on Suffix Trees).

Input: An ordered pair (T, f) of an unlabeled ordered rooted tree T and a function $f : V_T^{in} \to (V_T^{in} \cup \{\top\})$.

Output: A string w for which the unlabeled graph induced from $ST(w)$ and sl_w is isomorphic to the graph induced from T and f, if such exists.

Here we say that a string w *realizes* (T, f) if w is a solution to Problem 1 w.r.t. (T, f). If there exists a string which realizes (T, f), then (T, f) is said to be a valid input.

Figure 2 illustrates a valid input of Problem 1; `aaababa$`, `aababaa$`, `abaaaba$` and `ababaaa$` realize the tree. On the other hand, Figure 3 illustrates an invalid input of Problem 1; no strings realize the tree. In the figures, the tree T and the function f of an input (T, f) are depicted by solid arrows and dotted arrows, respectively. Also, the inner nodes and the leaf nodes of T are described by the circles and the squares, respectively.

3 Algorithm

3.1 Characterization of Suffix Trees

In this subsection, we give new characterizations of suffix trees and show how to construct a string whose suffix tree and its suffix links for inner nodes are isomorphic

to a given (T, f). Let us remark that the facts to be presented in this subsection apply also to alphabets of an arbitrary size, not only to binary ones.

Proposition 2. *For any string w, $|\mathsf{children}(\bot_{ST(w)})|$ represents the number of distinct characters occurring in w.*

Since a terminal symbol \$ is the lexicographically smallest and occurs exactly once in w, the following proposition holds.

Proposition 3. *For any string w, $\mathsf{child}_1(\bot_{ST(w)})$ is a leaf node of $ST(w)$.*

The next proposition follows from the definition of suffix links.

Proposition 4. *For any string w and $v_0 \in V^{in}_{ST(w)}$, there exists a sequence $v_1, v_2, \ldots, v_k$ of nodes such that $v_k = \top$ and $v_i = sl_w(v_{i-1})$ for any $1 \leq i \leq k$.*

The above proposition says that there exists a path of suffix links from any inner node v_0 to the auxiliary node $\top$.

Proposition 5. *For any string w and $u, v \in V^{in}_{ST(w)}$, if $sl_w(u) = sl_w(v)$ and the longest common ancestor between u and v is not $\bot_{ST(w)}$, then $u = v$.*

Proof. Let x and x' be the strings corresponding to u and v, respectively. $sl_w(u) = sl_w(v)$ implies that $|x| = |x'|$ and $x[2 : |x|] = x'[2 : |x'|]$. Since the longest common ancestor between u and v is not $\bot_{ST(w)}$, $x[1] = x'[1]$. Hence $x = x'$, i.e., u and v are identical. □

The next corollary follows from the above propositions.

Corollary 6. *For any string w and $v \in V^{in}_{ST(w)}$, $|\{u \in V^{in}_{ST(w)} \mid sl_w(u) = v\}| < |\mathsf{children}(\bot_{ST(w)})|$.*

Let (T, f) be an input of Problem 1. Since it can be checked in linear time whether (T, f) satisfies the conditions of Propositions 3, 4 and 5, in what follows we assume that (T, f) satisfies those conditions. Also, since every string terminates with an endmarker \$, we assume that every inner node of T has at least two children. In addition, we assume $\mathsf{par}(\bot_T) = \top, \mathsf{children}(\top) = \{\bot_T\}$ and $f(\bot_T) = \top$.

For any $v \in V_T$, we define $sldepth(v)$ by the number of links from v to the root node, namely,

$$sldepth(v) = \begin{cases} 0 & \text{if } v = \bot_T, \\ sldepth(f(v)) + 1 & \text{if } v \in (V^{in}_T - \bot_T), \\ sldepth(\mathsf{par}(v)) + 1 & \text{if } v \in V^{leaf}_T. \end{cases}$$

Lemma 7. *Let (T, f) be an input of Problem 1. If $f(v)$ is a descendant of $f(\mathsf{par}(v))$ for any $v \in (V^{in}_T - \{\bot_T\})$, then $sldepth(\mathsf{par}(v)) < sldepth(v)$ holds for any $v \in V_T$.*

Proof. When v is a leaf node, by definition $sldepth(v) = sldepth(\mathsf{par}(v)) + 1$. Then we prove that for any $u \in V^{in}_T$ and $v \in (V^{in}_T(u) - \{u\})$, $sldepth(u) < sldepth(v)$, by induction on the value of $sldepth(v)$. As a base statement, when $u = \bot_T$, the statement holds due to $sldepth(\bot_T) = 0$. As an induction step, assume that the statement holds for any $u \in V^{in}_T$ with $sldepth(u) < k$, and consider any $u \in V^{in}_T$ with $sldepth(u) = k$. Since $f(v)$ is a descendant of $f(u)$ and $sldepth(f(u)) = k - 1$, $sldepth(f(u)) < sldepth(f(v))$ holds, and hence, $sldepth(u) = sldepth(f(u)) + 1 < sldepth(f(v)) + 1 = sldepth(v)$. Therefore, the lemma holds. □

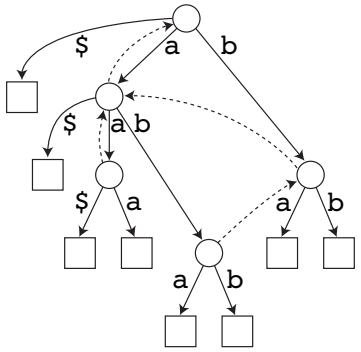


Figure 4. A valid labeling to the input shown in Figure 2.

We consider a labeling function $g : E_T \to (\Sigma \cup \{\$\})$ that associates each edge with a character in $\Sigma \cup \{\$\}$. For ease of description, we let $g((\top, \bot_T))$ denote an arbitrary character in $\Sigma \cup \{\$\}$, i.e., the auxiliary edge $(\top, \bot_T)$ is labeled with all characters in $\Sigma \cup \{\$\}$. Since we intend g to describe the first character of each edge string, let us assume that g satisfies the following preconditions:

1. $g((\bot_T, \mathsf{child}_1(\bot_T))) = \$$,
2. for any $v \in V_T$ and $1 \leq i < |\mathsf{children}(v)|$, $g((v, \mathsf{child}_i(v))) \prec g((v, \mathsf{child}_{i+1}(v)))$,
3. for any $(\mathsf{par}(v), v) \in E_T$, there exists a child u of $f(\mathsf{par}(v))$ s.t. $g((\mathsf{par}(v), v)) = g((f(\mathsf{par}(v)), u))$ and $f(v) \in V(u)$.

If there is no such function g, then (T, f) is invalid. We can check in linear time whether or not there exists a labeling g for which Preconditions 1, 2 and 3 hold, as follows. Firstly, we set $g((\bot_T, \mathsf{child}_i(\bot_T)))$ for any $1 \leq i < |\mathsf{children}(\bot_T)|$ so that they satisfy Preconditions 1 and 2. Secondly, for any $(\mathsf{par}(v), v) \in E_T$ with $v \in V_T^{in}$ and $\mathsf{par}(v) \neq \bot_T$, we determine $g((\mathsf{par}(v), v))$ according to Precondition 3 (note that they are uniquely determined), and check if they satisfy Precondition 2 or not. Finally, for any $(\mathsf{par}(v), v) \in E_T$ with $\mathsf{par}(v) \neq \bot_T$ and $v \in V_T^{leaf}$, we check if $g((\mathsf{par}(v), v))$ satisfies Preconditions 2 and 3 or not.

A labeling g is said to be a *valid labeling* if there exists a string w which realizes (T, f) and the first character of each edge string of $ST(w)$ coincides with the character defined by g.

Figure 4 illustrates one of the valid labelings to the input shown in Figure 2. Note that the first character of every edge string of $ST(\texttt{ababaaa\$})$ shown in Figure 1 coincides with the valid edge label of the tree shown in Figure 4.

For any labeling g and $v \in V_T$, let

$$L_g(v) = |\{u \in V_T^{leaf} \mid f(\mathsf{par}(u)) = \mathsf{par}(v), g((\mathsf{par}(u), u)) = g((\mathsf{par}(v), v))\}| \text{ and}$$
$$D_g(v) = L_g(v) + \sum_{u \in \mathsf{children}(v)} D_g(u) = \sum_{u \in V_T(v)} L_g(u).$$

Figure 5 illustrates the relationship between a node $v \in V_T$ and the leaves related to $L_g(v)$.

To show intuitive meanings of the above notations, let us consider a valid labeling g, and a string w which realizes (T, f) and g. For any $v \in V_T$ and $u \in V_T^{leaf}$ satisfying $f(\mathsf{par}(u)) = \mathsf{par}(v)$ and $g((\mathsf{par}(u), u)) = g((\mathsf{par}(v), v))$, we see that a string ax occurs exactly once in w, where x is the substring corresponding to v and ax is a prefix of the suffix corresponding to u. In addition, for any ancestor v' (corresponding to

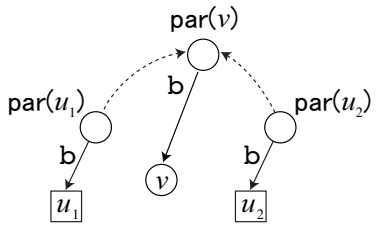


Figure 5. Relationship between a node $v \in V_T$ and the leaves related to $L_g(v)$. In this case, $L_g(v) = 2$.

a substring x') of v, since x' is a prefix of x, one of the occurrences of x' in w is attributed to ax, i.e., to the leaf node u. Note that all the information comes from (T, f) and g, and is independent of a solution w, that is, from the standpoint of the reverse engineering problem, they are constraints to determine w. Hence $D_g(v)$, the sum of the values $L_g(u)$ for all $u \in V_T(v)$, implies the constraints by which v is affected.

The following lemma describes a necessary condition for g to be valid.

Lemma 8. *Let (T, f) be an input of Problem 1 and g be a labeling function. If g is a valid labeling, then $|V_T^{leaf}(v)| \geq D_g(v)$ for any $v \in V_T$.*

Proof. Let w be a string which realizes (T, f) and g. Let $v \in V_T$ and x be a string corresponding to v. Since ax (a is any character in $\Sigma \cup \{\$\}$) occurs at least $D_g(v)$ times in w, $D_g(v)$ is bounded by the number of occurrences of x, that is, $|V_T^{leaf}(v)|$. □

For a labeling function g satisfying the condition of Lemma 8, we introduce a directed multiedge graph, called a *suffix tour graph* w.r.t. g, as follows.

Definition 9 (Suffix Tour Graph). *Let (T, f) be an input of Problem 1 and g be a labeling function such that $|V_T^{leaf}(v)| \geq D_g(v)$ for any $v \in V_T$. The suffix tour graph $STG_g = (V_G, E_G)$ w.r.t. g is defined as follows:*

$$\begin{aligned} V_G &= V_T, \\ E_G &= \{(u, v) \mid u \in V_T^{leaf}, f(\mathsf{par}(u)) = \mathsf{par}(v), g((\mathsf{par}(u), u)) = g((\mathsf{par}(v), v))\} \\ &\quad \cup \{(u, v)^k \mid (u, v) \in E_T, k = |V_T^{leaf}(v)| - D_g(v)\}, \end{aligned}$$

where $(u, v)^k$ is a k-multiedge from u to v.

Figure 7 illustrates the suffix tour graph w.r.t. the labeling shown in Figure 6. In Figure 6, the number in each node v represents the value $|V_T^{leaf}(v)| - D_g(v)$.

Lemma 10. *Let (T, f) be an input of Problem 1 and g be a labeling function. STG_g is an Eulerian graph (possibly disjoint).*

Proof. It suffices to show that the indegree and the outdegree of any $v \in V_T$ are equal. Let v be any node in V_T^{in}. It follows from the definition of E_G that the indegree and outdegree of v are $L_g(v) + |V_T^{leaf}(v)| - D_g(v)$ and $\sum_{u \in \mathsf{children}(v)}(|V_T^{leaf}(u)| - D_g(u))$, respectively. By taking a subtraction between them, we get

$$\begin{aligned} &(L_g(v) + |V_T^{leaf}(v)| - D_g(v)) - (\sum_{u \in \mathsf{children}(v)} (|V_T^{leaf}(u)| - D_g(u))) \\ &= L_g(v) - D_g(v) + \sum_{u \in \mathsf{children}(v)} D_g(u) = 0. \end{aligned}$$

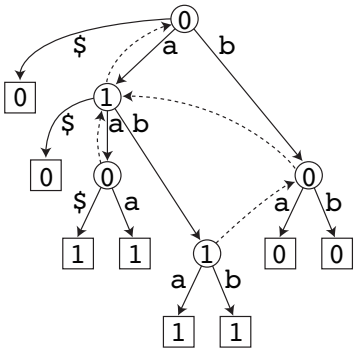


Figure 6. The value $|V_T^{leaf}(v)| - D_g(v)$ of each node v is overlaid on Figure 4.

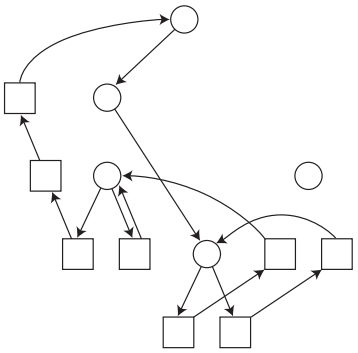


Figure 7. The suffix tour graph w.r.t. the labeling shown in Figure 6.

For any $v \in V_T^{leaf}$, the outdegree of v is 1. Since $D_g(v) = L_g(v)$, in both cases of $D_g(v) = 0$ and $D_g(v) = 1$, the indegree of v is 1. Therefore the lemma holds. □

Lemma 11. *Let (T, f) be an input of Problem 1 and g be a labeling function. The size of $STG_g = (V_G, E_G)$ is $O(|V_T|)$.*

Proof. Since $|V_G| = |V_T|$ and $|\{(u, v) \mid u \in V_T^{leaf}, f(\mathsf{par}(u)) = \mathsf{par}(v), g((\mathsf{par}(u), u)) = g((\mathsf{par}(v), v))\}| = |V_T^{leaf}|$, what we have to do is to estimate the number of edges in $\{(u, v)^k \mid (u, v) \in E_T, k = |V_T^{leaf}(v)| - D_g(v)\}$. It follows from Lemma 7 that $sldepth(u) < sldepth(v)$ for any $(u, v) \in E_G \cap E_T$. Let us call such edges "downward edges". On the other hand, the number of "upward edges", edges $(u, v) \in E_G$ holding $sldepth(u) > sldepth(v)$, is at most $|V_T^{leaf}|$. Since STG_g is an Eulerian graph, the number of "downward edges" is bounded by that of "upward edges", namely, is bounded by $|V_T^{leaf}|$. □

Lemma 12. *Let (T, f) be an input of Problem 1 and g be a labeling function. g is a valid labeling if and only if $STG_g = (V_G, E_G)$ has an Eulerian cycle which contains $\perp_T$ and all leaves of T.*

Proof. **if part:** Let us consider an Eulerian cycle which starts with $\perp_T$ and ends with the edge $(\mathsf{child}_1(\perp_T), \perp_T)$, and for any $1 \le i \le n$, let ℓ_i be the i-th leaf to pass in the cycle, where $n = |V_T^{leaf}|$ (note that the cycle contains each leaf just one time). Let w be a string of length n such that for any $1 \le i \le n$, $w[i] = g((\perp_T, v))$ with $(\perp_T, v) \in E_T$ and $\ell_i \in V_T^{leaf}(v)$. Then we prove that w realizes (T, f) and g.
Let $(v_p, v) \in E_T$ and $\ell_i, \ell_j \in V_T^{leaf}(v)$. It follows from the definition of E_G that both ℓ_{i+1} and ℓ_{j+1} are in $V_T^{leaf}(u)$, where $u \in \mathsf{children}(f(v))$ such that $g((f(v), u)) = g((v_p, v))$. Recursively applying this $h = sldepth(v_p)$ times, we get $w[i : i + h] = w[j : j + h]$ and $w[i + h] = w[j + h] = g((v_p, v))$. Here let us associate v with $w[i : i + h]$. Then for any $v \in V_T$, the substring associated with v occurs in w exactly $|V_T^{leaf}(v)|$ times, and w contains no other substring. Hence, w realizes (T, f) and g.

only if part: Let w be a string of length n which realizes (T, f) and g. Since STG_g is an Eulerian graph, it suffices to show that $\perp_T$ and all leaves of T are connected by some path on STG_g.
For any $1 \le i < n$, let u and v be the leaf node corresponding to $w[i : n]$ and $w[i + 1 : n]$, respectively. Let v_1 be the node such that $(u, v_1) \in E_G$. Evidently

$v \in V_T^{leaf}(v_1)$. Then we prove that $|V_T^{leaf}(v_j)| > D_g(v_j)$ for any $1 < j \leq k$, where $v_k = v$ and $\mathsf{par}(v_j) = v_{j-1}$ for any $1 < j \leq k$. Assume on the contrary that $|V_T^{leaf}(v_j)| = D_g(v_j)$ for some $1 < j \leq k$. It means that ax (a is any character in $\Sigma - \{w[i]\}$) occurs $|V_T^{leaf}(v_j)| = D_g(v)$ times in w, where x is a string corresponding to v_j. This contradicts that $w[i]x$ occurs in w. Consequently there is a path from u to v, and hence, all leaves are connected by some path. In addition, it is clear that $\perp_T$ and $\mathsf{child}_1(\perp_T)$ (the node related to $w[n:n]$) is connected by $(\mathsf{child}_1(\perp_T), \perp_T)$. Therefore STG_g has an Eulerian cycle which contains $\perp_T$ and all leaves of T.

□

Lemma 13. *For any input (T, f) of Problem 1 and any labeling function g,*

1. *we can check whether or not there exists a string w which realizes (T, f) and g,*
2. *we can compute a string which realizes (T, f) and g, if such exists,*

in linear time in the size of T.

Proof. First of all, if $|V_T^{leaf}(v)| < D_g(v)$ for some $v \in V_T$, then g is invalid. Otherwise, we consider an Eulerian cycle on STG_g. If a cycle which contains $\perp_T$ and all leaves of T is found, we can construct a string which realizes (T, f) and g as discussed in Lemma 12. If no such cycles are found, then g is invalid. Since an Eulerian cycle in a given Eulerian graph can be computed in linear time in the size of the graph, it follows from Lemma 11 that these operations can be done in $O(|V_T|)$ time. □

3.2 Linear Time Algorithm for a Binary Alphabet

The following theorem is the main result of this paper.

Theorem 14. *On a binary alphabet, Problem 1 can be solved in linear time.*

Proof. Let (T, f) be an input of Problem 1. By Lemma 13, given a valid labeling, we can construct a string which realizes (T, f). Then the remaining task is to search for a valid labeling.

In the binary case, the following conditions are needed for (T, f) to be valid:

- For any $v \in V_T^{in}$, the number of children of v is 2 or 3.
- For any $v \in V_T^{in}$, if $|\mathsf{children}(v)| = 3$, the first child of v is a leaf node.
- For any $v \in (V_T^{in} - \{\perp_T\})$, if $|\mathsf{children}(v)| = 3$, then $|\mathsf{children}(f(v))| = 3$.

Any input that does not satisfy these conditions is filtered out in the process (discussed in Subsection 3.1) of checking whether or not there exists g which satisfies Preconditions 1, 2 and 3. Also, remark that the checking process uniquely determines the label $g((v_p, v))$ for any $(v_p, v) \in E_T$ with $v_p = \perp_T$ or $v \in V_T^{in}$.

Let $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. For any $v \in V_T^{in}$ with $|\mathsf{children}(v)| = |\mathsf{children}(f(v))|$, the label $g((v, \mathsf{child}_i(v))$ is fixed to $g((f(v), \mathsf{child}_i(f(v)))$ for any $1 \leq i \leq |\mathsf{children}(v)|$. Hence, the degree of freedom to determine a labeling g lies only in nodes $v \in V_T^{in}$ such that $|\mathsf{children}(v)| = 2$ and $|\mathsf{children}(f(v))| = 3$. Moreover, by Lemma 8, we need to choose g with $|V_T^{leaf}(v)| \geq D_g(v)$ for any $v \in V_T$. In particular, for the binary case, the next arguments hold:

- If there exist $u, v \in V_T^{in}$ with $u \neq v$, $f(u) = f(v) = q$ and $|\mathsf{children}(u)| = |\mathsf{children}(v)| = |\mathsf{children}(q)| = 3$, then the input (T, f) is invalid. This is because, if such exists, then it leads to $|V_T^{leaf}(q')| = 1 < 2 \leq D_g(q')$, where $q' = \mathsf{child}_1(q)$.

- Let $q \in V_T^{in}$ such that $|\mathsf{children}(q)| = 3$ and there is not $v \in V_T^{in}$ with $|\mathsf{children}(v)| = 3$ and $f(v) = q$. Note that such a node is unique if the above condition holds. Then, it follows from Corollary 6 that $|\{v \in V_T^{in} \mid f(v) = q, |\mathsf{children}(v)| = 2\}| \leq 2$.
- For any $v \in V_T^{in}$ with $|\mathsf{children}(v)| = 2$ and $|\mathsf{children}(f(v))| = 3$, if there exists $u \in V_T^{in}$ with $f(u) = f(v)$, $|\mathsf{children}(u)| = 3$ and $|\mathsf{children}(f(u))| = 3$, then $g((v, \mathsf{child}_1(v))) = \mathsf{a}$ and $g((v, \mathsf{child}_2(v))) = \mathsf{b}$.

Putting these together, we see that the number of possible labelings is at most 5, namely, in the maximum case, there are the following five possible allocations

$$\begin{aligned}&\langle g(u, \mathsf{child}_1(u)), g(u, \mathsf{child}_2(u)), g(v, \mathsf{child}_1(v)), g(v, \mathsf{child}_2(v))\rangle \\ &\in \{\langle \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{b}\rangle, \langle \$, \mathsf{a}, \mathsf{a}, \mathsf{b}\rangle, \langle \$, \mathsf{b}, \mathsf{a}, \mathsf{b}\rangle, \langle \mathsf{a}, \mathsf{b}, \$, \mathsf{a}\rangle, \langle \mathsf{a}, \mathsf{b}, \$, \mathsf{b}\rangle\},\end{aligned}$$

where $u, v \in V_T^{in}$ such that $u \neq v$, $f(u) = f(v) = q$, $|\mathsf{children}(u)| = |\mathsf{children}(v)| = 2$ and $|\mathsf{children}(q)| = 3$.

Figure 8 illustrates all the possible labelings to the input shown in Figure 2.

Thus, we only have to check at most five labeling functions. It follows from Lemma 13 that each labeling can be checked in $O(|V_T|)$ time, and hence, Problem 1 can be solved in linear time in the input size. □

4 Conclusions and Future Work

We presented a linear time algorithm to solve the reverse engineering problem on suffix trees for a binary alphabet, where inner nodes of input suffix trees are augmented with suffix links. The algorithm is designed based on combinatorial properties of suffix trees. There remain a lot to do on the reverse engineering problem of suffix trees. For instance: Can we enumerate all strings that realize a given unlabeled tree? Can we solve the problem for larger alphabets? Can we solve the problem without suffix links?

References

1. A. Apostolico: *The myriad virtues of subword trees.* Combinatorial Algorithms on Words, F12 1985, pp. 85–96.
2. H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda: *Inferring strings from graphs and arrays*, in Proc. MFCS 2003, vol. 2747 of LNCS, 2003, pp. 208–217.
3. J. Clément, M. Crochemore, and G. Rindone: *Reverse engineering prefix tables*, in Proc. STACS 2009, 2009, pp. 289–300.
4. M. Crochemore, C. Iliopoulos, S. Pissis, and G. Tischler: *Cover array string reconstruction*, in Proc. CPM 2010, vol. 6129 of LNCS, 2010, pp. 251–259.
5. J.-P. Duval, T. Lecroq, and A. Lefebvre: *Border array on bounded alphabet.* Journal of Automata, Languages and Combinatorics, 10(1) 2005, pp. 51–60.
6. J.-P. Duval, T. Lecroq, and A. Lefebvre: *Efficient validation and construction of border arrays and validation of string matching automata.* RAIRO - Theoretical Informatics and Applications, 43(2) 2009, pp. 281–297.
7. J.-P. Duval and A. Lefebvre: *Words over an ordered alphabet and suffix permutations.* Theoretical Informatics and Applications, 36 2002, pp. 249–259.
8. F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang: *Verifying a border array in linear time.* J. Comb. Math. and Comb. Comp., 42 2002, pp. 223–236.
9. P. Gawrychowski, A. Jez, and L. Jez: *Validating the Knuth-Morris-Pratt failure function, fast and online*, in Proc. CSR 2010, vol. 6072 of LNCS, 2010, pp. 132–143.
10. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, New York, 1997.

11. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Counting parameterized border arrays for a binary alphabet*, in Proc. LATA 2009, vol. 5457 of LNCS, 2009, pp. 422–433.
12. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Counting and verifying maximal palindromes*, in Proc. SPIRE 2010, vol. 6393 of LNCS, 2010, pp. 135–146.
13. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Verifying a parameterized border array in $O(n^{1.5})$ time*, in Proc. CPM 2010, vol. 6129 of LNCS, 2010, pp. 238–250.
14. W. Matsubara, A. Ishino, and A. Shinohara: *Inferring strings from runs*, in Proc. PSC 2010, 2010, pp. 150–160.
15. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23(2) 1976, pp. 262–272.
16. D. Moore, W. F. Smyth, and D. Miller: *Counting distinct strings.* Algorithmica, 23(1) 1999, pp. 1–13.
17. K.-B. Schürmann and J. Stoye: *Counting suffix arrays and strings.* Theoretical Computer Science, 395(2-3) 2008, pp. 220–234.
18. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
19. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.

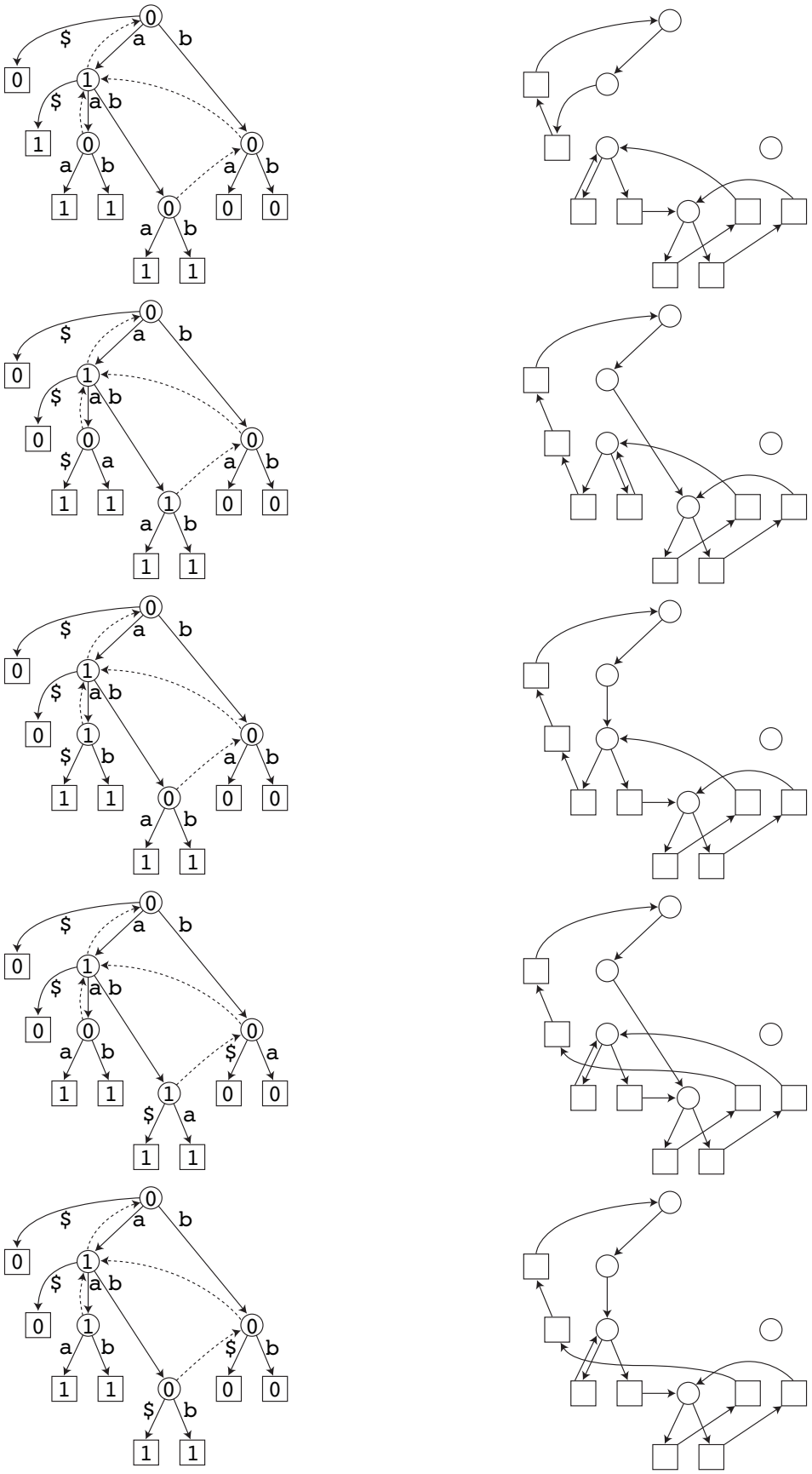


Figure 8. All the possible labelings. The top one is invalid and the others are valid.

Minimization of Acyclic DFAs

Johannes Bubenzer

University of Potsdam
Department of Linguistics
Karl-Liebknecht-Strasse 24-25
14476 Potsdam
bubenzer@uni-potsdam.de

Abstract. There exists a linear time algorithm for the minimization of acyclic deterministic finite-state automata (ADFA) due to Dominique Revuz [17]. The algorithm has different phases involving the computation of a *height* for each state, sorting states of the same height and joining equivalent states. We present a new linear time algorithm for the task at hand. The algorithm is conceptionally simpler and computes the minimal automaton in only a single depth-first traversal over the automaton. The algorithm utilizes the notion of a right-language *Register* known from algorithms for the incremental construction of minimal ADFAs from lists. In an evaluation we compare the running times of both algorithms. The results show that the new algorithm is significantly faster than Revuz' algorithm.

Keywords: minimization, acyclic deterministic finite state automata, algorithmic

1 Introduction

The DFA minimization problem dates back to the 1950s with the works of [12] and [14]. In the meantime a multitude of different minimization algorithms were discussed. There are several different known algorithms, with different running times and properties. Additionally, there are algorithms that only minimize automata of specific types. In this paper we present a new algorithm for the minimization of acyclic DFAs and compare it to Revuz' well-established algorithm [17] for this problem.

In section 2 we give an overview on different minimization algorithms for the general case of DFAs. Mathematical preliminaries are presented in section 3. Then we describe the best known minimization algorithm for acyclic DFAs and its proof in section 4. In section 5 we propose a new minimization algorithm for the acyclic case. We prove its correctness and discuss its advantages over the former algorithm. Finally, we describe the results of an evaluation comparing both algorithm (section 6).

2 Overview

First, we describe DFA minimization algorithms that do not pose any restriction about acyclicity. The now classical Moore-Algorithm [14] shows $O(n^2)$ time and memory complexity for n being the number of states of the automaton to be minimized. It works by marking non-equivalent states. This procedure starts with pairs of final and non-final states marked as non-equivalent. In the subsequent steps all those states leading into non-equivalent states are marked as non-equivalent, until no more states to be marked are found. The fastest known algorithm is the Hopcroft algorithm [10], it runs in $O(n \log n)$. Due to the complexity of the algorithm and its proof, a few papers were published that explain and re-explain the Hopcroft algorithm ([8], [13]).

Johannes Bubenzer: Minimization of Acyclic DFAs, pp. 132–146.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

The algorithm is derived from the classical Moore-Algorithm and maintains sets of possibly equivalent states. Those state-sets are then split up in a special way such that the number of splitting operations along with the costs of splitting a state-set leads to the aforementioned complexity.

There is also an incremental algorithm [21], further enhanced in [23]. The algorithm determines the minimal automaton in cubic time, but can be halted at any time – resulting in a partially minimized automaton.

The Brzozowski-algorithm [2] poses a notable exception to the algorithms described here. The algorithm determines the minimal automaton $\mathcal{A}_m$ by applying a chain of regular operations to the source automaton $\mathcal{A}$.

$$\mathcal{A}_m \longleftarrow det(inv(det(inv(\mathcal{A})))) \tag{1}$$

The inner automaton inversion $inv()$ and determinization $det()$ are minimizing the suffix part of the automaton. The outer inversion and determinization then minimize the prefix part. The determinization operation used is the subset method, which ensures that the suffix part is only broken up if required for the prefix compression. The algorithm shows exponential worst case complexity, since the complexity of determinization is exponential. But it performs exceptionally well in practice [4].

For certain subsets of the deterministic finite-state automata, one can achieve minimization in linear time. The best know algorithm is due to Revuz [17] – it minimizes acyclic DFAs in linear time. This approach is discussed in depth in section 4. [1] extend Revuz' algorithm and show that the O(n)-complexity bound can also be achieved for a bigger subclass, the so called single-cycle automata. These are automata where each state can have at most one outgoing transition leading into a cycle.

There also exist algorithms for the direct compilation of minimal ayclic DFAs from lists of words. [22] poses an excellent and profound summary of the work in this field. Furthermore, [3] describes improved algorithms for the task at hand.

A taxonomy of the most important finite state minimization algorithms can be found in [19] and in [20]. The presented paper is an elaboration on parts of the work done in the author's thesis [3].

3 Preliminaries

In this section we introduce the basic definitions and theorems regarding languages and automata needed in this paper. A more complete introduction using similar notions can be found in [11].

3.1 Alphabets, Words and Languages

An *alphabet* Σ is a finite set of symbols. Any finite sequence of concatenations of symbols $w = a_1 \cdot a_2 \cdots a_n \mid a_i \in \Sigma$ from Σ is called a *word* over Σ. The symbol $w[i] = a_i \in \Sigma$ of a word w is the symbol at the i'th position. The length of a word $|w|$ is the number of concatenated symbols. By ε we denote the empty word ($|\varepsilon| = 0$). We denote by Σ^* the *free monoid* generated by Σ. That is the set of all words over Σ including ε, along with identity ε and the concatenation $\cdot$ as operation. By Σ^+ we denote the *free semigroup* generated by Σ, that is the set of all non-empty words together with the concatenation operation. Any subset $L \subseteq \Sigma^*$ of Σ^* is called a *language*. For convenience we assume each alphabet to be a total order together with some operation $<$.

3.2 Automata and Languages

A *deterministic finite-state automaton* (DFA) $\mathcal{A} = \langle Q, q_0, \Sigma, \delta, F \rangle$ consists of a finite set of states Q, a designated start-state $q_0 \in Q$, an alphabet Σ, a set of final states $F \subseteq Q$ and a transition function $\delta : Q \times \Sigma \mapsto Q$. The transition function is extended to the acceptance of words in the usual way:

$$\begin{aligned} \delta^*(q, \varepsilon) &= q \\ \delta^*(q, aw) &= \delta^*(\delta(q, a), w) \end{aligned}$$

for all states $q \in Q$ and $a \in \Sigma$ is a symbol, $w \in \Sigma^*$ is a word. A word w is said to be *accepted* by the automaton, if $\delta^*(q_0, w) \in F$. The language $\mathcal{L}(\mathcal{A})$ accepted by a DFA $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

Each language accepted by some DFAs is called a *regular language*.

The *right-language* $\overrightarrow{\mathcal{L}}(q)$ of a state $q \in Q$ is defined as the set of words accepted by an automaton started in q:

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* \mid \delta^*(q, w) \in F\}$$

We denote by $\delta'(q)$ the set of outgoing transitions from state q:

$$\delta'(q) = \{\langle a, p \rangle \mid a \in \Sigma,\ p \in Q,\ \delta(q, a) = p\}$$

The destination state of a transition $t \in \delta'(q)$ is denoted by t_{next} and the transition symbol by t_{sym}.

A typical way to implement the set of outgoing transitions is as an array of transitions. We use the term *transition array* in the following to refer to the actual implementation of the set of outgoing transitions.

We define the *signature* $\overrightarrow{q}$ of a state q, to be q's set of outgoing transitions unified with ε iff the state is final:

$$\overrightarrow{q} = \begin{cases} \delta'(q) \cup \{\varepsilon\} & \texttt{if } q \in F \\ \delta'(q) & \texttt{else} \end{cases}$$

In the latter we will often refer to the signature as being a sequence of finality attribute and states rather than a set. A sequential signature holds the transitions ordered by the transition symbol, the finality attribute being the first element if present.

A state q is called *accessible* if it is reachable from the start-state:

$$\exists w \in \Sigma^* \mid \delta^*(q_0, w) = q$$

A state q is said to be *co-accessible* if there is a path from q to a final state.

$$\exists w \in \Sigma^* \mid \delta^*(q, w) \in F$$

A DFA is *connected* iff each of its states is both accessible and co-accessible. Each non-connected DFA can be easily transformed into a connected DFA by removing all non-accessible and all non-co-accessible states as well as transitions from and

into these states from the automaton. Note that this operation does not change the language of the automaton.

A DFA is called *acyclic* deterministic finite-state automaton (ADFA) if no state is connected to itself by a chain of transitions and *cyclic* otherwise.

We will assume in the following, that the transition arrays of states are sorted by their transition symbols. We can assume this without loss of generality as shown in the following proof.

Lemma 1. *We can sort the transition arrays of all states of a DFA in linear time.*

Proof. We are looking at DFAs and therefore the number of outgoing transitions from one state is bounded by the constant $|\Sigma|$. Since Σ is a constant, we can sort the transition array of a single state in time relative to $|\Sigma|$. This is done for each state in $|Q|$ once. Therefore the computation takes linear time wrt. the size of the automaton.

3.3 Minimality

One important property of DFAs is that they have a state *minimal* representation. This ensures small automata for given languages. It follows from the Myhill-Nerode theorem ([15], [16]), that for each regular language $\mathcal{R}$, there exists exactly one *minimal deterministic finite-state automaton* (MDFA) $\mathcal{A}$ (excluding isomorphism) accepting the language $\mathcal{R}$ ($\mathcal{L}(\mathcal{A}) = \mathcal{R}$) with a minimal number of states. Further it follows that the minimal automaton for a given language $\mathcal{L}$ is exactly that automaton accepting $\mathcal{L}$ which has only states with mutually different right-languages:

Definition 2 (Minimal DFA). *A connected DFA $\mathcal{A}$ is minimal if all states have mutually different right languages:*

$$min(\mathcal{A}) \iff \forall q, p \in Q : \overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(p) \tag{2}$$

We call two states $p, q \in Q$ equivalent iff they have the same right-languages ($\overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p)$) and inequivalent otherwise.

Theorem 3 (Minimal ADFA). *An acyclic DFA is minimal if all states have mutually different right-language signatures.*

Proof. Let us assume that in an acyclic connected DFA $\mathcal{A}$ all states have different signatures but at least two states accept the same right-language: Then the following statement must be true:

$$\begin{aligned}
&\exists p, q \in Q : \overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p) \wedge \forall q, p \in Q : \overrightarrow{q} \neq \overrightarrow{p} && (3)\\
\Rightarrow &\exists p, q \in Q : \overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p) \wedge \overrightarrow{q} \neq \overrightarrow{p} && (4)\\
\Leftrightarrow &\exists p, q \in Q : \overrightarrow{\mathcal{L}}(q) = \overrightarrow{\mathcal{L}}(p) \wedge (q \in F \neq p \in F && (5)\\
&\qquad\qquad \vee \exists a \in \Sigma : \delta(q, a) \neq \delta(p, a)) && (6)
\end{aligned}$$

It is impossible that both states have equal right languages if one is final and the other is not. Therefore both states have to differ in their outgoing transitions to fulfill the last equation. This could be for two reasons.
(1) There is a transition for a symbol $a \in \Sigma$ for one state but not for the other. This would violate the equation since then one state accepts a word starting with a while the other doesn't and their right-languages would differ. (2) There is a transition with

symbol $a \in \Sigma$ leading into a state q_2 for q and to p_2 for p and $q_2 \neq p_2$. By precondition we know that both q_2 and p_2 have different right-language signatures. Further they need to have the same right-languages to make the statement become true. But then by the same argument there must exist two states q_3 and p_3 and so on up to infinity. This is impossible since the automaton is acyclic.

Theorem 4 (Equal signatures). *Two states $q, p \in Q$ with equal right language signatures ($\overrightarrow{q} = \overrightarrow{p}$) are equivalent.*

Proof. Since both states lead into the same destination states with the same labels and are either both final or non-final they are equivalent, obviously.

Definition 5 (Minimal states). *A state $q \in Q$ is called a minimal state iff no equivalent state p exists in the underlaying automaton and all successor states of q are minimal. A final state with no successor states is a minimal state iff no other final state without successor states exists.*

Definition 6 (Minimal signatures). *The signature of a state $q \in Q$ is called minimal signature $\overrightarrow{q}_m$ iff all of its successor states are minimal states.*

Theorem 7 (Differing minimal signatures). *Two states $q, p \in Q$ of an ADFA with different minimal signatures ($\overrightarrow{q}_m \neq \overrightarrow{p}_m$) are not equivalent.*

Proof. Let us assume that the opposite would be true and that two equivalent states $q, p \in Q$ with different minimal signatures exist. Then we arrive at equation 2 of the proof of theorem 3. We already showed there that such a situation is impossible in an acyclic DFA.

4 Algorithm for the acyclic case

The case of minimization of acyclic DFAs can be done in linear time, wrt. to the size of the input automaton. In the following we describe the linear time algorithm for acyclic connected DFA by [17]. This algorithm consists of two phases. In the first phase a height is calculated for each state and all states are grouped together according to the their height. The *height* of a state is its maximal distance to a final state. In the second phase the states of the same height are sorted according to their signature starting with states of the lowest height and proceeding to the highest. States of the same height and signature are equivalent according to their right-language (given that all states of smaller height are already minimized). Those equivalent states are then minimized and incoming transitions are adjusted accordingly. The critical computation is the sorting of the states. As shown below this can be done in linear time, using a special kind of bucket-sort. In the following we will informally describe Revuz' algorithm.

Definition 8. *The height h_q of a state q is the length of the longest path to a final state. And can be defined with the following recursive formula.*

$$h_q = \begin{cases} 0 & if\ \delta'(q) = \emptyset \\ 1 + max(h_p) : \langle a, p \rangle \in \delta'(q) & else \end{cases}$$

Theorem 9 (Computing heights). *The computation of the heights of all states of a DFA $\mathcal{A}$ can be done in linear time.*

Proof. We show this by proving that the height can be computed by a depth first traversal. In a depth first traversal over an acyclic DFA, it is guaranteed that for each state q all successor states are processed before q is fully processed. Therefore a depth first traversal over $\mathcal{A}$ is conducted. If a state q has no outgoing transitions, it is of height 0. Otherwise the heights of its successors are computed and q receives a height of one plus the height of the successors with the biggest height as indicated in definition 8. A depth first traversal takes linear time.

After the height of a state q is computed, q is inserted into a partition P_{h_q} of states with height h_q. Partitions are stored in an array of size $|Q|$, since the maximum possible height of a state is $|Q| - 1$ if all states form a single path. In the next step, states of the same height are sorted according to their signature and all states with equal signature are then merged into one.

Radix Sort Radix sort [18], a variant of bucket sort, is a sorting algorithm which can sort an array of (sequential) elements in linear time. The elements to be sorted are sequences of singular values, drawn from some fixed alphabet (for example bytes). The alphabet has to be known beforehand. Initially, buckets are created, one for each symbol in the alphabet. The elements are sorted into those buckets according to their least significant value. Then, again the elements in the buckets are rearranged according to the next least significant bit and so on. When different length sequences occur, the shorter sequences are treated as if their most significant values were padded with the lowest alphabet symbol. In each pass, at most n elements are rearranged and the rearrangement happens at most as often as the length of the longest sequence. Therefore the algorithm takes linear time.

Sorting states of the same height Given an array of states of the same height, one can use radix sort to sort the array in linear time. In this case the size of the alphabet $|\Sigma|$ appears as constant factor k in the complexity bound. To improve this, Revuz suggests the following computation. An array A of size $|\Sigma|$, which is preinitialized with zero is held. This is used to temporary rewrite the signatures of all states of the current height level. Now, all elements of the signatures of the states are scanned and each element is looked up in the array A. If the element is already present, the signature is rewritten by the value of the element in the array. Otherwise, the element is inserted into the array and a value of one plus the number of elements currently in the array is assigned to it. Additionally, the element is pushed onto a pushdown store used after the sorting to reinitialize the array A. After rewriting all signatures, radix sorting can be done with a constant factor k equal to the number of distinct elements in the signatures of the states of the current level.

A somewhat simpler way with the same complexity would be to use a hash to index the states of one height. Equivalent states would then be merged to the same slot.

Theorem 10 (Linear time). *Minimization using Revuz algorithm takes linear time.*

Proof. Computation of the height partitioning for all states takes linear time. Sorting the states of one height partition takes linear time, wrt. the size of the partition. Determining states of equal signature obviously takes linear time, too, since equivalent states are sorted to adjacent positions. Rewriting equivalent states (and the transitions of their successors) such that one class representative is chosen, is also a linear

time operation. Since each height partition is computed exactly once and the number of states of all partitions sum up to $|Q|$. We conclude that minimization according to Revuz takes linear time.

As was just shown, all the individual steps performed by the algorithm take linear time. It remains to show that the algorithm indeed computes the minimal automaton. We will show this with the following recursive proof. First, we show that upon the computation of a certain height level, all states of that level have minimal signatures.

Theorem 11 (Minimal signatures). *States of a certain height level have minimal signatures when they are considered.*

Proof. The algorithm starts by considering states of height 0 states that are final and have no outgoing transitions. Those states have minimal signatures, obviously.
Given a state q of height $h_q > 0$ is considered. Since in that case all successor states of q have smaller heights, they were considered before and are already minimized by assumption. Therefore the state q has a minimal signature.

As shown in theorem 4, states with equal signatures are equivalent and can therefore be merged together. Next, we show that upon merging all equivalent states of the given height wrt. their signatures, all states of that height are truly minimal.

Theorem 12 (After computation of a height level all of its states are minimal).

Proof. A state is minimal iff no other state with the same right language exists. We show by contradiction that this is impossible. Let us assume there exists a equivalent state p of height h_p (with the same right language) as a state q of the current height h_q after merging the states of height level h_q. The state p could be
(1) *of smaller height.* Since the longest path from q to a final state has length h_q and the longest path of p has length h_p and $h_q > h_p$, the right language of q contains a word of length h_q which is not in the right language of p. Then q and p can not be equivalent.
(2) *of bigger height.* Since the longest path from q to a final state has length h_q and the longest path of p has length h_p and $h_p > h_q$, the right language of p contains a word of length h_p which is not in the right language of q. Again, both states can not be equivalent then.
(3) *of the same height.* Then p and q have different signatures since otherwise they would have been merged together. But the signatures are minimal as shown above. By theorem 7 we known that the states can not be equivalent.

Theorem 13 (Minimal ADFA). *The algorithm computes the minimal ADFA.*

Proof. The algorithm computes all height levels up to the biggest. By theorem 12 we known that upon the computation of a height level all of its states are minimal. Also, no states of lower heights are changed within the computation of a specific height level. Therefore after computation of the biggest height level all states in the automaton are minimal.

5 Improved algorithm for the acyclic case

In the following we describe a new algorithm for the minimization of acyclic connected DFAs. We prove its correctness and linear running time. The algorithm turns out to be faster in practice than Revuz' as supported by experimental evaluation in section 6. It is also more intuitive in our opinion. Daciuk [5] has described a similar algorithm for minimizing tries. But surprisingly the generalization to arbitrary ADFAs was not described in the literature before.

The algorithm maintains a data-structure mapping right-languages to states, which we will call the *Register*. This terminology was introduced by [6] in the context of compiling lists of words to MDFAs. In the following, we will treat the Register as if it stores right-languages as keys. Since the right-language of a state can contain up to $|\Sigma|$ transitions and there are up to $|Q|$ distinct right-languages for each DFA, such a Register would require $O(|\Sigma|n) = O(n)$ space at the most. Using a hash-table, lookup and storage of a right-language requires constant time. Nevertheless, in practical applications one could compute an integer hash-key from the right-language transition-array using a sufficient hash-function. The hash value does contain a pointer to the desired state then. Doubly representation of the transition array (in the state and as hash key) can therefore be avoided. Then the constant $|\Sigma|$-factor would disappear in the memory-complexity estimation of the Register.

A mapping from states to minimized states (*StateMap*) is filled by the algorithm. The *StateMap* is required since we need to be able to detect if some state encountered by the algorithm were already minimized before.

The algorithm merely consists of a depth-first traversal over the states of the automaton. If it encounters a state q with no outgoing transitions, q has the trivial right-language signature $\overrightarrow{q} = \langle\varepsilon\rangle$. Note that such a state is always final, because we are assuming connected DFAs. If no state with this right-language was encountered before, q is the new representative of the class and is kept. Otherwise q is replaced by the trivial state encountered first. If the algorithm encounters a state q with at least one outgoing transition, the decision on the minimal right-language of q is delayed until all successor states are minimized. Then q will also either form the representative of a new class of minimized states, or be replaced by the earlier detected representative of q's right-language class. The (recursive) pseudo-code is given in algorithm 1. For convenience, we assume the algorithm to be a method of the ADFA to be minimized. To minimize an ADFA the method is called with the start state q_0, an empty *Register* and the *StateMap* as arguments.

In the following, we will prove that the algorithm indeed computes the MDFA for a given DFA. As precondition we require the input automaton to be deterministic, acyclic and connected.

Definition 14. *We define a strict partial order $<$ over the states of an (acyclic) DFA, such that*

$$\forall p, q \in Q : p < q \iff \exists \alpha \in \Sigma^+ : \delta^*(p, \alpha) = q \tag{7}$$

Lemma 15. *The algorithm computes the states wrt. the order given by $<$.*

Proof. The algorithm performs a depth-first (preorder) traversal. This kind of traversal is known to compute states wrt. the order given by $<$.

Algorithm 1: Depth-first minimization of acyclic DFAs

```
1  begin minimize(State q, Register R, StateMap M)
2      foreach Transition t ∈ δ'(q) do
3          if M[t_next] == undef then
4              minimize(t_next, R, M)
5          t_next = M[t_next]
6      if R[q⃗] == undef then
7          M[q] = R[q⃗] = q
8      else
9          M[q] = R[q⃗]
10         deleteState(q)
```

We will prove that the algorithm indeed computes the right MDFA given a DFA in the following way. We state as a loop invariant, that at the beginning of the execution of the algorithm no states are mapped to wrong representatives (of their right language), and therefore no errors were made so far. We will then show by contradiction, that a situation where the loop invariant changes (that is: an error is introduced) can not exist.

Lemma 16. *The DFA is correctly minimized after the execution of the algorithm.*

Proof. Assume Lemma 16 does not hold. This means that either two states with the same right language exist or the FSA is not deterministic or not connected anymore.

Deterministic: The input automaton is deterministic by precondition. This could only change if either transitions are altered or added. Transitions are only altered in line 5, but there only the destination state changes. No transitions are added. We conclude that the automaton is deterministic afterwards.

Connected: The input automaton is connected by precondition. To lose this property requires either the finality attribute of a state to change or a state to be added or a state to be deleted leaving its incoming transitions or transitions to be deleted leaving their destination states unreachable. The finality attribute never changes. No states are added. States are only deleted in line 10. But each state that is deleted is marked in the $StateMap$ by an equivalent state. Deletion occurs only at the first invocation of a given state q. At each later invocation of q, the equivalent state is used (in line 5) and all occurrences (as destination of transitions) of q are replaced by it. We conclude that the automaton is connected afterwards.

Equivalent states: Let us assume there are at least two distinct states with the same right-language after minimization. The states in $StateMap$ are those which are already treated and minimized. The assumption implies that after the execution of the algorithm, there are at least two states in $StateMap$ that are mapped to distinct states, but are equivalent nevertheless. At the beginning of the algorithm there is no state registered in $StateMap$. Therefore the invariant holds, that at that time no two states are minimized and mapped to distinct states, but are equivalent. That means it must be possible to identify a line in the algorithm where the invariant changes. That is, before the execution of the line invariant holds and afterwards it doesn't hold anymore. This could only happen in line 7 or 9, since only there changes to the $StateMap$ are made.

Let us assume the invariant changes in line 7, this means we are treating a state q, which has a right language that was not seen before. Since we have not seen the q's right-language before (but by invariant all right-languages were computed correctly so far), no state can possibly be equivalent to the current one. Therefore the only reason for an error at this point could be, that q's right-language was computed wrong. We know from the strict partial order $<$ that all successor states were visited and minimized before the current state. We also know that they are correct and that the right-language vector is sorted. Since the right language results from the direct successor states q, the only possibility that q's right-language is wrong is, that the computation of the right-language of successor state's were wrong. This contradicts the assumption, that the invariant changes at this point and can therefore not be true. Also if q's right-language is trivial (q has no successors) no error could possibly be introduced.

Given the invariant changes in line 9. There, a state p with the same right-language as the state q is found and the state q is mapped onto p. To fulfill the assumption that an error is introduced, this mapping has to be wrong. This means that the class that q is mapped to is not the class q belongs in. p's right-language cannot be wrong by invariant. Therefore the right-language assigned to q must be wrong. This cannot be true, by the same argument as in the case of line 7.

Lemma 17. *The minimization takes linear time in the number of states of the input automaton.*

Proof. The algorithm performs a depth-first traversal. Each state is processed exactly once in the function *minimize*. Also the destination state of each transition of each state is processed and redirected at most once. Lookup and storage in the state register and state map require constant time. Therefore the minimization takes linear time.

5.1 Comparison

In comparison to Revuz' algorithm the new one neither requires an external sorting phase, nor does it require states to be sorted into buckets. All work is done in the single depth-first traversal. This makes the new algorithm quite easy to understand and implement. The complexity of Revuz' algorithm is hidden in the sorting phase, whereas most of the complexity of the new algorithm lays in the implementation of the Register. The register can be implemented as a general purpose hash, whereas Revuz' sorting is not general purpose. We believe that the new algorithm is therefore the better choice for the minimization of DFAs. Since Revuz' also requires factorization of states into buckets, we also believe that the constant factors of the new algorithm are lower and that it will execute faster. In the evaluation section we will strengthen this statement by showing that the new algorithm is faster in practice.

6 Evaluation

The new algorithm achieves improvements regarding the constant factors of the running time over the original algorithms but the asymptotic time-complexity is unchanged. To asses the advantage of the new algorithm we conducted experiments on random data as well as on natural language data sets. In this section we present the

results of our evaluation. As input to the minimization algorithms we used trie-DFAs compiled from randomly generated word lists. In the following we describe the random lists along with the sampling methods used to generate the data sets. Thereafter we describe the natural language data-sets used in the evaluation. Finally we present the results for the new minimization algorithm in comparison to the original algorithm. The experiments in this section were conducted on a computer with 4-core Intel Core^{TM} i5 750 processor (2.66 GHz), 8 GB of RAM running a Linux operating system with 64-bit architecture.

6.1 Random Data

We tried to design our experiments such that the influence of different parameters can be assessed from the results. The following parameters were varied in the individual experiments:

1. alphabet size
2. maximum string length
3. length of the input list
4. sampling distributions

We performed of our experiments on random data sets using two different alphabet sizes, namely $|\Sigma| = 5$ and $|\Sigma| = 50$. For the maximum string length we evaluated lists of short strings of at most 10 characters and lists of long strings of up to 50 characters. We varied the length of the input lists over the values $\{1000, 100000, 200000, \ldots, 1000000\}$ for each experiment. We used two different random sampling distributions to generate the words of the random lists. On the one hand the uniform (discrete) random distribution over strings up to the maximum length were evaluated. On the other hand a random distribution where each string-length up to the maximum string length gets the same probability whilst strings of a certain length are uniformly distributed.

In the uniform random distribution each string w over the alphabet Σ of length up to the maximum length l has the same probability $p(w)$:

$$p(w) = \frac{1}{\sum_{i=0}^{l} |\Sigma|^i} \tag{8}$$

The uniform distribution produces far more strings of the maximum length than of smaller lengths.

Under the distribution with equally distributed lengths a word w with length $|w| = n$ over the alphabet Σ with maximum word length l has the probability $p(w)$:

$$p(w) = \frac{1}{l|\Sigma|^n} \tag{9}$$

Note that we excluded the empty word ε from being generated by this second distribution.

For each combination of the different parameters described we created 16 different random lists. Running times were obtained by running both algorithms on a trie obtained from each random list. The average of the different runs on a specific parameter combination were computed for each algorithm. We actually report the percentage rates of the running times of the different algorithms. For each experiment the running time of the slowest algorithm were considered as 100 %.

6.2 Natural Language Data

To verify the performance of the algorithms with more realistic data sets we used word lists of different natural languages. We compared the running times of the algorithms on lists of the 10000 most frequent words for German, English, Dutch and French. The word lists were obtained from the website of the project Deutscher Wortschatz [7]. For each word list we compiled a trie and took the average running time over 1000 runs for each minimization algorithm. The results for the natural language data sets are reported as percentages computed in the same way as described above.

6.3 Results

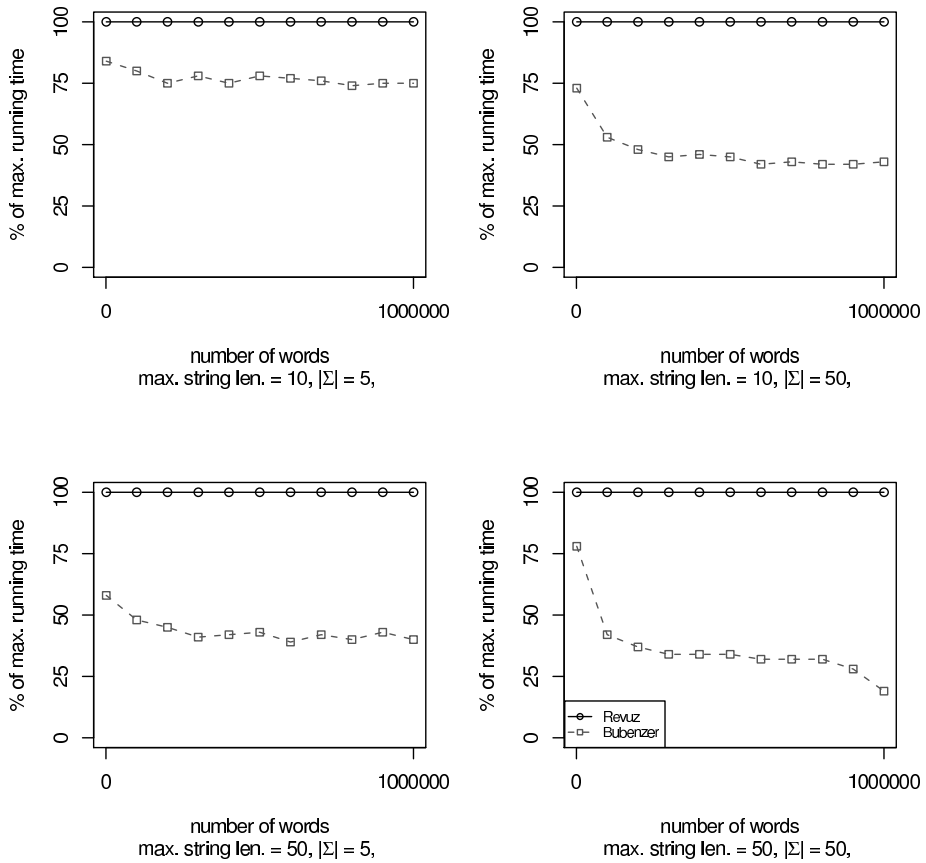


Table 1. Performance of ADFA minimization methods under uniform distribution. String lengths and alphabet sizes vary over the plots.

We implemented the new minimization algorithm in and integrated it into the FSM⟨2.0⟩-library [9] in the C++ programming language. FSM⟨2.0⟩ contains methods to create, maintain, save and load automata which enabled us to build up the DFAs required for the experiments, easily. The library also contains a well optimized version of Revuz' algorithm and was therefore suited for (unbiased) experiments comparing both minimization algorithms. All experiments involved two phases. The performance of the second phase was actually measured.

1. In the first phase a trie-DFA was build up from a list of words. This DFA was then stored in a binary file (which can be loaded fast).

2. The second phase contained in loading the binary automaton and running the desired minimization algorithm.

Therefore the time for loading the binary automaton is always included in the reported results. Further note that the FSM⟨2.0⟩-library handles different automata representations and finite-state machine types. Caused by this multi-functionality, there may be a certain computational overhead which is included in the results obtained.

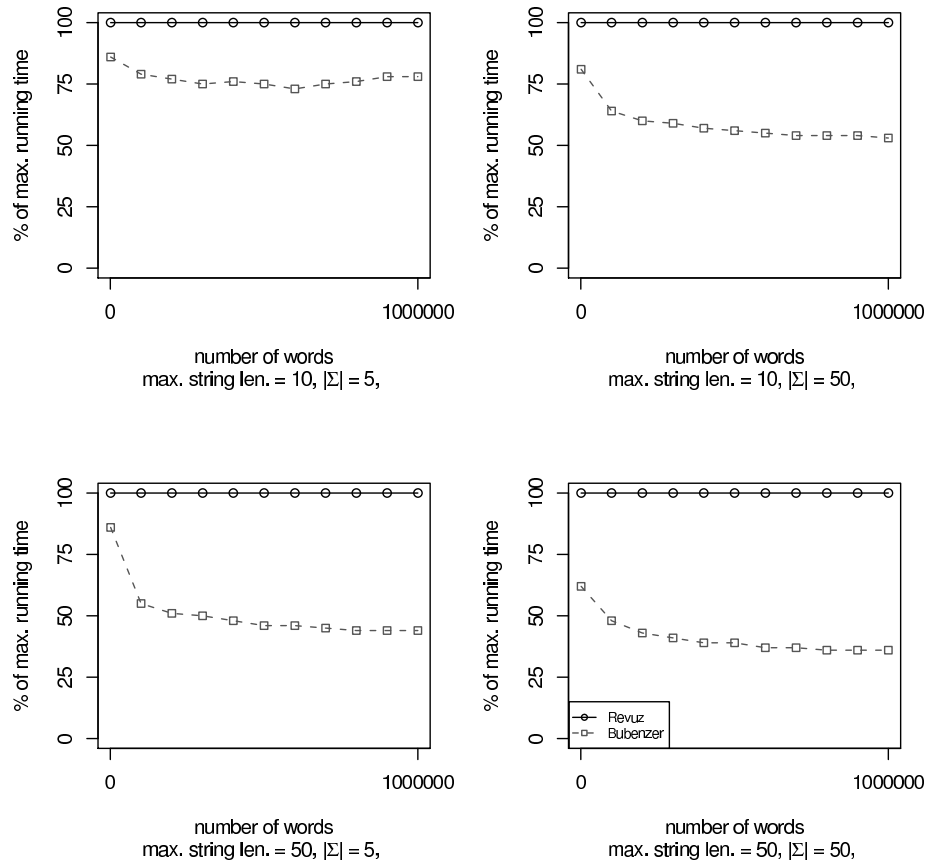


Table 2. Performance of ADFA minimization methods on data with uniform distributed lengths. String lengths and alphabet sizes vary over the plots.

In table 1 the results of both minimization algorithms are compared for data sampled from the uniform distribution. It is evident that the new algorithm performs a lot faster than the original one. The gap between both algorithm grows both with increasing alphabet size and increasing maximum word length. The difference for short lists is quite small. This is because the program overhead of initialization, preallocation and loading the unminimized automaton is quite big in comparison to the actual minimization algorithm. The curve for the new algorithm shows a slope for very large lists in the plot with alphabet size $|\Sigma| = 50$ and maximum string length 50. The reason may be that the implementation of Revuz' algorithm uses too much memory and begins to swap data to the hard drive. To make a profound statement about this further research is required.

The algorithms perform comparable on the lists generated with uniform distributed lengths (table 2). The differences between both implementations are smaller.

Language	Revuz	Bubenzer
German	1.0	0.76
Rnglish	1.0	0.73
French	1.0	0.73
Dutch	1.0	0.76

Table 3. Performance of ADFA minimization algorithms on natural language data sets.

This is because the total amount of data is smaller in this case since the distribution produces shorter words far more frequently. For the same reason the slope that occurred in the last experiment doesn't arise in the plots.

We obtained similar results for the natural language data-sets as can be seen in table 3. The word lists are quite small in comparison to the random lists. It can be expected that the gap between both algorithms would increase with longer input lists in a similar way as with the random data.

7 Conclusions

We presented a new minimization algorithm for acyclic DFAs and proved its correctness. Further we evaluated the performance of the algorithm against Revuz' well-established algorithm for this case. Our results show that the new algorithm is significantly faster in practice.

References

1. J. Almeida and M. Zeitoun: *Description and analysis of a bottom-up DFA minimization algorithm.* Information Processing Letters, 107(2) 2008, pp. 52–59.
2. J. A. Brzozowski: *Canonical regular expressions and minimal state graphs for definite events*, in Mathematical theory of Automata, Volume 12 of MRI Symposia Series, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962, pp. 529–561.
3. J. Bubenzer: *Construction of minimal ADFAs*, Diplomarbeit, Universität Potsdam, Germany, 2011.
4. J.-M. Champarnaud, A. Khorsi, and T. Paranthoën: *Split and join for minimizing: Brzozowski's algorithm*, in PSC'02: The Prague Stringology Conference '02, Czech Technical University in Prague, 2002.
5. J. Daciuk: *Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings*, in Implementation and Application of Automata, Springer, 2003, pp. 127–152.
6. J. Daciuk, B. W. Watson, S. Mihov, and R. E. Watson: *Incremental construction of minimal acyclic finite-state automata.* Computational Linguistics, 26(1) 2000, pp. 3–16.
7. Deutscher Wortschatz: `http://wortschatz.uni-leipzig.de`, 2011.
8. D. Gries: *Describing an algorithm by Hopcroft.* Acta Informatica, 2 1973, pp. 97–109.
9. T. Hanneforth: *fsm2 – A scripting language interpreter for manipulating weighted finite-state automata*, in Anssi Yli-Jyrä et al. (eds): Finite-State Methods and Natural Language Processing, 8th International Workshop, Berlin, 2009, Springer, pp. 13–30.
10. J. E. Hopcroft: *An $n \log n$ algorithm for minimizing the states in a finite automaton*, in The Theory of Machines and Computations, Z. Kohavi, ed., Academic Press, 1971, pp. 189–196.
11. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Company, Reading, MA, 1979.
12. D. A. Huffman: *The synthesis of sequential switching circuits.* Journal of the Franklin Institute, 257(3) 1954, pp. 161–190.

13. T. Knuutila: *Re-describing an algorithm by Hopcroft*. Theor. Comput. Sci., 250 January 2001, pp. 333–363.
14. E. F. Moore: *Gedanken-experiments on sequential machines*, in Automata Studies, Princeton, 1956, pp. 129–153.
15. J. Myhill: *Finite automata and the representation of events*, Tech. Rep. WADD TR-57-624, Wright Patterson Air Force Base, Ohio, 1957.
16. A. Nerode: *Linear automaton transformations*. Proceedings of the American Mathematical Society, 9 1958, pp. 541–544.
17. D. Revuz: *Minimization of acyclic deterministic automata in linear time*. Theoretical Computer Science, 92(1) 1992, pp. 181–189.
18. R. Sedgewick: *Algorithms in C++, 3rd Ed.*, Addison-Wesley, Reading, MA, 1998.
19. B. W. Watson: *A taxonomy of finite automata construction algorithms*, Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.
20. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.
21. B. W. Watson: *An incremental DFA minimization algorithm*, in ESSLLI Workshop 2001, 2001.
22. B. W. Watson: *Constructing minimal acyclic deterministic finite automata*, PhD thesis, University of Pretoria, Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa, 2010.
23. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm*. Nat. Lang. Eng., 9 March 2003, pp. 49–64.

Notes on Sequence Binary Decision Diagrams: Relationship to Acyclic Automata and Complexities of Binary Set Operations

Shuhei Denzumi[1], Ryo Yoshinaka[2], Hiroki Arimura[1], and Shin-ichi Minato[1,2]

[1] Graduate School of IST, Hokkaido University, Japan
[2] ERATO MINATO Discrete Structure Manipulation System Project, JST, Japan
{denzumi, ry, arim, minato}@ist.hokudai.ac.jp

Abstract. Manipulation of large sequence data is one of the most important problems in string processing. Recently, Loekito *et al.* (Knowl. Inf. Syst., 24(2), 235–268, 2009) have introduced a new data structure, called *Sequence Binary Decision Diagrams (SeqBDDs, or SDDs)*, which are descendants of both acyclic DFAs (ADFAs) and binary decision diagrams (BDDs). SDDs can compactly represent sets of sequences as well as minimal ADFAs, while SDDs allow efficient set operations inherited from BDDs. A novel feature of the SDDs is that different SDDs can share equivalent subgraphs and duplicated computation in common to save the time and space in various operations. In this paper, we study fundamental properties of SDDs. In particular, we first present non-trivial relationships between sizes of minimum SDDs and minimal ADFAs. We then analyze the complexities of algorithms for Boolean set operations, called the binary synthesis. Finally, we show experimental results to confirm the results of the theoretical analysis on real data sets.

1 Introduction

1.1 Background

Compact string indexes for storing sets of strings are fundamental data structures in computer science, and have been extensively studied in the decades [2,4,5,6,9,19]. Examples of compact string indexes include: tries [1,5], finite automata and transducers [6,10], suffix trees [15], suffix arrays [14], DAWGs [2], and factor automata (FAs) [19]. By the rapid increase of massive amounts of sequence data such as biological sequences, natural language texts, and event sequences, these compact string indexes have attracted much attention and gained more importance [5,9]. In such applications, an index have not only to compactly store sets of strings for *searching*, but also have to efficiently manipulate them with various set operations, e.g., *merge, intersection*, and *subtraction*.

Minimal acyclic deterministic finite automata (*ADFAs*) [5,6,10] are one of such index structures that fulfill the above requirement based on finite automata theory, and have been used in many sequence processing applications [13,18]. However, they have drawback of complicated procedures for minimization and various set operations caused by multiple branching of the underlying directed acyclic graph structure. To overcome this problem, Loekito *et al.* [12] proposed the class of *sequence binary decision diagrams* (*sequence BDDs*, or abbreviated as *SDDs* in this paper), which is a compact representation for sets of strings that allows a variety of operations for sets of strings. An SDD is a node-labeled graph structure, which resembles to an acyclic DFA in binary form, but with the minimization rule which is different from one for

Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, Shin-ichi Minato: Notes on Sequence Binary Decision Diagrams: Relationship to Acyclic Automata and Complexities of Binary Set Operations, pp. 147–161.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

a minimal DFA. A novel feature of the SDDs is their ability to share equivalent subgraphs and results of similar intermediate computation between different SDDs, which avoids redundant generation of nodes and computation.

1.2 Main results

In this paper, we present theoretical analysis of two fundamental problems on sequence binary decision diagrams, which have not been studied before: the relationship to acyclic automata and the complexities of binary set operations as follows.

The relationship to acyclic automata. The structure of SDDs apparently resembles that of Acyclic Deterministic Finite Automata (ADFAs), which are a classical model for representing string sets. While a state of an ADFA may have many outgoing edges, a node of an SDD always has two outgoing edges, which can be seen as just the "first-child next-sibling" representation of a branching with many edges. Indeed one can find a straightforward translation from an ADFA to an SDD and vice versa. However, there are subtle differences between those data structures and actually an SDD can be even more compact than the corresponding ADFA. We show that the minimum SDD is never larger than the minimum ADFA but the minimum ADFA can be $|\Sigma|$ times larger than the minimum SDD for the same language over Σ.

The computational complexities of binary set operations. Next, we study the complexity of the *binary synthesis*, which are binary operations for minimal SDDs, such as *union*, *intersection*, and *subtraction*, which directly construct a minimal SDD. Specifically, we study upper and lower bounds for the time complexity of the binary synthesis algorithms. Loekito *et al.* [12] have proposed algorithms for union and subtraction, which are similar to the apply algorithm Bryant [3], and have conjectured that they run in input-output linear-time. We generalize their algorithms into an algorithm $\mathsf{Meld}_\diamond$ which uniformly implements eight set operations in the style of Knuth's *melding* operation for BDDs [11]. We show an upper bound that its time complexity is quadratic in the input size, and linear in the size of non-reduced version of the output size. Moreover, we show a lower bound that $\mathsf{Meld}_\diamond$ actually requires quadratic time in input size for some infinite series of inputs using a technique recently devised for BDD [3], giving matching upper bound.

Experimental results. Finally, we run experiments on real data sets. We first observed that minimal SDDs were superior to minimal DFAs when large subgraphs were shared in inputs and outputs due to the node-sharing across multiple SDDs. We also observed that each binary synthesis operation took less than seconds to take set operations $\cup$, $\cap$, and $\setminus$ of two input SDDs with around three to four thousands of nodes each, which were relatively smaller than the running time for set construction [7].

1.3 Related works

There have been a number of researches on manipulation of finite automata in automata theory and string algorithms. The textbook [10] gives classic examples of a quadratic-time algorithm for computing the union, intersection, and subtraction of two DFAs, and a state-minimization algorithm for a given DFA. Daciuk, Mihov, Watson, and Watson [6] presented an incremental algorithm for constructing the minimal ADFA for a set of strings. Blumer *et al.* [2] and Crochemore [4] gave linear-time algorithms for construction of the minimal state ADFAs for the set of all factors of an

input string. Compared with a straightforward two-stage algorithm for binary synthesis for ADFAs using product followed by state-minimization [10], the advantages of the proposed $\mathsf{Meld}_\diamond$ are its simplicity and efficiency that it directly computes the output by applying the *on-the-fly minimization* [7]. Using binary synthesis, Denzumi *et al.* [7] presented a simple linear-time algorithm for incremental construction of SDDs, and a recursive top-down algorithm for construction of factor SDDs.

SDDs inherit many of their features from *binary decision diagrams* (*BDDs*), which are compact representation for storing and manipulating combinatorial structures developed in logic design community [3,11,16,21]. Especially, BDDs equipped with binary synthesis operation were invented by Bryant [3] in the 80s for dealing with Boolean functions, while their variant with node-sharing and zero-suppress rules, called *zero-suppressed BDDs* (*ZDDs*), were proposed by Minato [16] in the 90s for sparse combinatorial sets. On their early history, reduced BDDs were constructed from tree-like circuits through offline minimization. After the invention of the *binary synthesis* algorithm by Bryant [3], it became popular to build large BDDs on-the-fly in real applications. Loekito *et al.* [12] discovered that if we remove the ordering constraint on the 1-edges from ZDDs, the resulting variant of ZDDs, which actually are SDDs, has a similar structure to ADFAs in binary form and suitable to storing and manipulating sets of strings. This observation led to the invention of SDDs [12].

Organization of this paper. In Section 2, we prepare basic notions and notations on SDDs. In Section 3, we give the size bounds for SDDs and DFAs. In Section 4, we give the time and space complexities of binary synthesis procedures for SDDs. In Section 5, we show some experimental results. In Section 6, we conclude this paper. For details of basic properties and algorithms related to SDDs not described in this paper, please consult the companion paper [7].

2 Preliminaries

In this section, we give basic definitions and notations in strings and sequence BDDs according to [11,12,16]. For the details of results not found here, please consult the companion paper [7]. An *ordered alphabet* is a pair $\langle \Sigma, \prec \rangle$ where Σ is a finite alphabet and $\prec$ is a total order on Σ. The order $\prec$ associated with Σ is often denoted by $\prec_\Sigma$ and the ordered alphabet is simply written Σ for legibility. A *string* on Σ is a sequence $s = s_1 \cdots s_n$ of letters $s_i \in \Sigma$ $(1 \leq i \leq n)$, where $|s| = n$ denotes the *length*. If $s = xyz$ for some $x, y, z \in \Sigma^*$, then we say that x is a *prefix*, y is a *factor*, and z is a *suffix* of s. A *string set* (or a *language*) is any finite $S \subseteq \Sigma^*$. We denote by $|S|$ the cardinality. For any $x \in \Sigma$, we define $x \cdot S = \{\, xy \mid y \in S \,\}$.

Sequence BDDs. Let dom be a countable domain of the nodes. A *sequence binary decision diagram* or a *sequence BDD* (abbreviated as *SDD*[1] here) is a directed acyclic graph (DAG) $B = \langle \Sigma, V, \tau, \mathbf{r}, \mathbf{0}, \mathbf{1} \rangle$ where $V = V(B) \subseteq \mathsf{dom}$ is a finite set of nodes, $\mathbf{r} \in V$ is called the *root* of B and $\mathbf{0}$ and $\mathbf{1} \in V$ are distinct nodes called the *0-* and *1-terminals*, resp. The nodes in $V_\mathrm{N} = V \backslash \{\mathbf{0}, \mathbf{1}\}$ are called *nonterminals*. Each node $v \in V_\mathrm{N}$ of B is labeled by a symbol $v.lab$ in Σ and has two children, the *0-child* and the *1-child*, denoted by $v.0$ and $v.1$, resp, which can be identical. We call the

[1] Note that the abbreviation SeqBDD is used to denote sequence BDD in the original paper by *Loekito et al.* [12]. We also note that the abbreviation SDD was also used for the *set decision diagrams* (Couvreur, Thierry-Mieg, Proc. FORTE 2005, LNCS 3731, 443–457, 2005) and the *spectral decision diagrams* (Thornton, Drechsler, Proc. DATE'01, IEEE, 713–719, 2001).

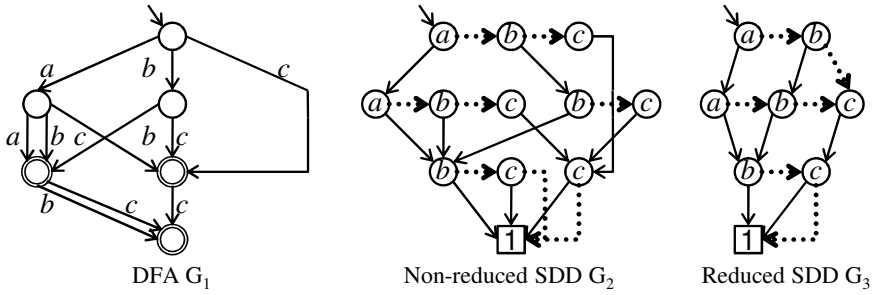


Figure 1. Examples of three index structures on $\Sigma_1 = \{a, b, c\}$ for the same string set $S_1 = \{aa, aab, aac, ab, abb, abc, ac, acc, bb, bbb, bbc, bc, bcc, c, cc\}$: a minimal DFA G_1 (left), a minimal DFA as a non-reduced SDD G_2 (middle), and a reduced SDD G_3 (right). In the figure, solid and dotted arrows indicate the 1- and 0-edge . The edges to the 0-terminal are omitted.

edge from v to $v.0$ and $v.1$ the *0-* and *1-edge* of v, resp. The information is formally described by a function $\tau : V_{\mathrm{N}} \to \Sigma \times V^2$ that assigns the triple $\tau(v) = \langle v.lab, v.0, v.1 \rangle$ to each $v \in V_{\mathrm{N}}$. An SDD must be *acyclic*, that is, one may assume a strict partial order $\succ_V$ on V such that $v \succ_V v.0$ and $v \succ_V v.1$ hold for any $v \in V_{\mathrm{N}}$. The 1-child and 0-child of a node correspond to the *leftmost-child* and the *right-sibling* in a DAG in *binary form* [1,11]. Siblings are deterministically ordered from *left* to *right* according to the order $\prec_\Sigma$. That is, we always have $v.lab \prec_\Sigma (v.0).lab$ unless $v.0$ is a terminal node. We assume that any SDD B is *well-defined* meaning that B is both acyclic and deterministic. We define the *size* of B by $|B| = |V_{\mathrm{N}}| = |V| - 2$, the number of non-terminals in B.

To each node $v \in V$, we inductively (w.r.t. $\succ_V$) assign a language $L_B(v)$ as follows:

(i) $L_B(\mathbf{0}) = \emptyset$;
(ii) $L_B(\mathbf{1}) = \{\varepsilon\}$;
(iii) $L_B(v) = L_B(v.0) \cup (v.lab) \cdot L_B(v.1)$.

Equivalently, $s \in L_B(v)$ iff there is a path from v to $\mathbf{1}$ such that one obtains s by concatenating the labels of the nodes whose 1-edges appear in the path. The language $L(B)$ of B is defined to be $L_B(\mathbf{r})$. We say that two SDDs B and B' are *equivalent* if $L(B) = L(B')$. An SDD B is said to be *minimal* if it has the smallest number of nodes among the equivalent SDDs, i.e., $|B| \leq |B'|$ for any SDD B' such that $L(B') = L(B)$. Figure 1 illustrates examples of SDDs together with the minimum deterministic finite automaton for the same language.

Reduced SDDs. A reduced SDD is a normal form of SDDs. An SDD is said to be *reduced* if it satisfies the following two conditions:

1. For any $u, v \in V_{\mathrm{N}}$, $\tau(u) = \tau(v)$ implies $u = v$ (*node-sharing rule*).
2. For any $v \in V_{\mathrm{N}}$, $v.1 \neq \mathbf{0}$ holds (*zero-suppress rule*).

The above rules say that no distinct non-terminal nodes have the same triple, and the 1-child of any non-terminal node v is not the 0-terminal. For any finite set of strings $L \subseteq \Sigma^*$, we can construct the *canonical SDD* for L [7] in a way similar to the minimal DFA in Myhill-Nerode theorem (e.g.,[10,20]). Actually, the next theorem gives a characterization of minimal SDDs in terms of a reduced SDD and the canonical SDD. See the companion paper [7] for the details.

Theorem 1 (Denzumi *et al.* [7]). *For any SDD B with the language $L = L(B)$, the following (1)–(3) are equivalent to each other.*

Global variable: *uniqtable*: hash table for triples.

Proc Getnode(x: letter, P_0, P_1: SDD):
1: **if** ($P_1 = 0$) **return** P_0; /* zero-suppress rule */
2: **else if** (($R \leftarrow$ *uniqtable*$[\langle x, P_0, P_1\rangle]$) exists) **return** R; /* node-sharing rule */
3: **else**
4: $R \leftarrow$ a new node with $\tau(R) = \langle x, P_0, P_1\rangle$; *uniqtable*$[\langle x, P_0, P_1\rangle] \leftarrow R$;
5: **return** R;

Figure 2. The Getnode procedure for on-the-fly minimization.

(1) B is a reduced SDD.
(2) B is a canonical SDD for L up to isomorphism.
(3) B is a minimal SDD.

Due to Theorem 1, for a fixed language L, we may call a reduced SDD for L *the* reduced SDD for L when we work modulo isomorphism. In order to satisfy the node-sharing rule, we maintain a hash table, called *uniqtable*, which is the inverse of τ. That is, it gives the unique node v such that $\tau(v) = \langle x, v_0, v_1\rangle$ (if exists) for the key $\langle x, v_0, v_1\rangle$. As is the case for BDDs and ZDDs, usually we consider only *reduced* SDDs. Hereafter we assume that all SDDs are reduced unless otherwise noted.

While an SDD represents a set of strings, we often would like to manipulate two or more sets of strings. In our *shared* SDD environment, the terminals **0** and **1**, the function τ and thus the hash *uniqtable* : $\Sigma \times \mathsf{dom} \times \mathsf{dom} \to \mathsf{dom}$ are shared by more than one SDD in common so that we can have a compact representation of a family of sets of strings. One may think of the shared SDD environment as a single SDD with multiple roots. By picking up a node v as the root, one can extract a subgraph as an SDD consisting of all the nodes that are reachable from v. For convenience, we often identify a node v with the SDD rooted by v extracted from the shared SDD environment. Hence $|v|$ represents the number of nonterminal nodes reachable from the node v.

Figure 2 shows the node allocation procedure Getnode, which is used as a subroutine in algorithms on SDDs. Throughout this paper, we assume that the hash table *uniqtable* is a global variable, and a look-up for it takes $O(1)$ time; We have to add additional $O(\log n)$ term if we use balanced binary tree dictionary [1]. In the shared and reduced SDD environment studied here and in [7,12], we use write-only construction, similarly to [8], such that any new SDD is constructed by adding a new node on the top of already constructed SDDs using a call of Getnode given existing nodes as its arguments. The next lemma guarantees that we always have reduced SDDs as long as we solely use Getnode to obtain a new node.

Lemma 2 (Denzumi *et al.* [7]). *Let B be any reduced SDD. For any symbol $x \in \Sigma$ and nodes $v_0, v_1 \in V(B)$ in B such that $v_0 \notin V_{\mathrm{N}}$ or $x \prec_\Sigma v_0.lab$, if we invoke $v = \mathsf{Getnode}(x, v_0, v_1)$ on B and add the result v to $V(B)$, then the resulting SDD B' with root v obtained from B is well-defined and reduced, too.*

Based on the procedure Getnode above, for example, we can implement an off-line minimization (i.e., reduction) algorithm Reduce for SDDs in linear time and space [7][2].

[2] The complexity analysis assumes that a look-up for the hash table *uniqtable* takes $O(1)$ time. A precise worst-case time complexity is $O(n \log n)$ if the hash table does not work efficiently.

Factually it simply makes a copy of an input SDD using Getnode, which merges equivalent nodes in *uniqtable*. See [7,12] for details.

3 Space-Bounds for Sequence Binary Decision Diagrams and Acyclic Automata

The structure of SDDs apparently resembles that of acyclic deterministic finite automata (ADFAs). There is a straightforward translation from an ADFA to an SDD and the other way around. However, we should note subtle differences between those formalisms. Actually SDD can be even more compact. This section discusses their relationship in detail.

3.1 Finite Automata

We presume a basic knowledge of the automaton theory. For a comprehensive introduction to the automaton theory, see [10] for example. A (partial) DFA is represented by a tuple $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, where Σ is the input alphabet, Γ is the state set, δ is the partial transition function from $\Gamma \times \Sigma$ to Γ, $q_0 \in \Gamma$ is the initial symbol and $F \subseteq \Gamma$ is the set of acceptance states. The partial function δ can be regarded as a subset $\delta \subseteq \Gamma \times \Sigma \times \Gamma$. We define the *size* of a DFA A, denoted by $|A|$, as the number of labeled edges in A, i.e., $|A| = |\delta|$.

The set of strings that lead the automaton A from a state q to an accept state is denoted by $L_A(q)$. The language $L(A)$ accepted by A is $L_A(q_0)$. A minimum DFA has no state q such that $L_A(q) = \emptyset$ and no distinct states q' and q'' such that $L_A(q') = L_A(q'')$. Since we are concerned with finite sets of strings, all DFAs discussed in this section are *acyclic* (ADFA). We say that A and B, which can be an ADFA or an SDD, are *equivalent* if $L(A) = L(B)$.

3.2 From ADFAs to SDDs

We first give a straightforward translation from an ADFA to an equivalent SDD, which may be non-reduced, and compare the sizes of them.

Theorem 3. *For any ADFA $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, there is an equivalent SDD $B = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \boldsymbol{r} \rangle$ such that $|V_{\mathrm{N}}| \leq |\delta|$. Moreover, for every positive integer $n \geq 1$, there is an ADFA A that admits no equivalent SDD B such that $|V_{\mathrm{N}}| < |\delta| = n$.*

Proof. For an ADFA $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, we construct an equivalent SDD $\mathbf{B}(A)$. Let

$$\deg(q) = |\{\, a \in \Sigma \mid \delta(q, a) \text{ is defined} \,\}|\,.$$

We define $\mathbf{B}(A) = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ as follows. The set of nodes is given by

$$V = \{\mathbf{0}, \mathbf{1}\} \cup \{\, [q, i] \mid q \in \Gamma \text{ and } 1 \leq i \leq \deg(q) \,\}.$$

For each $q \in \Gamma$ with $\deg(q) = k \geq 1$, let $a_1, \ldots, a_k \in \Sigma$ and $q_1, \ldots, q_k \in \Gamma$ be such that

- $\delta(q, a_i) = q_i$ for $i = 1, \ldots, k$,
- $a_1 \prec a_2 \prec \cdots \prec a_k$.

Define τ by

$$\tau([q,i]) = \begin{cases} \langle a_i, [q, i+1], \widehat{q_i} \rangle & \text{if } i < k, \\ \langle a_k, \mathbf{1}, \widehat{q_k} \rangle & \text{if } i = k \text{ and } q \in F, \\ \langle a_k, \mathbf{0}, \widehat{q_k} \rangle & \text{if } i = k \text{ and } q \notin F, \end{cases}$$

where

$$\widehat{q'} = \begin{cases} [q', 1] & \text{if } \deg(q') > 0, \\ \mathbf{1} & \text{if } \deg(q') = 0 \text{ and } q' \in F, \\ \mathbf{0} & \text{if } \deg(q') = 0 \text{ and } q' \notin F. \end{cases}$$

The root $\mathbf{r}$ of $\mathbf{B}(A)$ is $\widehat{q_0}$.

It is easy to see that $L_A(q) = L_{\mathbf{B}(A)}(\hat{q})$ for all $q \in \Gamma$. We note that the above construction can be done in linear time in $|\delta|$.

The first claim of the theorem can be verified by the above construction of $\mathbf{B}(A)$. The second claim is established by observing the minimum ADFA and the reduced SDD that accept the singleton $\{a^n\}$ for each positive integer n. For the detail of construction of the reduced (canonical) SDD from a string set, consult [7]. □

We remark that $\mathbf{B}(A)$ in the proof is not necessarily reduced for a minimum ADFA A.

Example 4. Let us compare the minimum ADFA A and the constructed SDD $\mathbf{B}(A)$ for the set $\{ab, b\}$ with $a \prec b$:

transition rules of A	corresponding nodes of $\mathbf{B}(A)$
$\delta(q_0, a) = q_1$	$\tau([q_0, 1]) = \langle a, [q_0, 2], [q_1, 1] \rangle$
$\delta(q_0, b) = q_2$	$\tau([q_0, 2]) = \langle b, \mathbf{0}, \mathbf{1} \rangle$
$\delta(q_1, b) = q_2$	$\tau([q_1, 1]) = \langle b, \mathbf{0}, \mathbf{1} \rangle$

$\mathbf{B}(A)$ is not reduced since $\tau([q_0, 2]) = \tau([q_1, 1])$ for $[q_0, 2] \neq [q_1, 1]$.

In this example, A has two distinct edges that are labeled with b and come into q_2, which should be merged into the same node in a reduced SDD. Hence the reduced SDD can be more compact than the minimum ADFA for the same language. We next discuss how much an SDD can be smaller than an ADFA through a translation from an SDD into an ADFA.

3.3 From SDDs to ADFAs

We next discuss how much an SDD can be smaller than an ADFA through a translation from an SDD into an ADFA. Let an SDD $B = \langle \Sigma, V, \tau, \mathbf{0}, \mathbf{1}, \mathbf{r} \rangle$ be given. We construct an ADFA $\mathbf{A}(B) = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$ such that $L(\mathbf{A}(B)) = L(B)$. We assume that $\mathbf{r} \neq \mathbf{0}$. Otherwise, the translation is trivial.

For each $P \in V_{\mathrm{N}}$, let $\widetilde{P} = [P_1, \ldots, P_k]$ be such that $P_1 = P$ and $\tau(P_i) = \langle a_i, P_{i+1}, R_i \rangle$ for some $R_i \in V$ for $i \leq k$ and $P_{k+1} \in \{\mathbf{0}, \mathbf{1}\}$. We define $\tilde{\mathbf{1}}$ to be $[\mathbf{1}]$. Let

- $\Gamma = \{\widetilde{\mathbf{r}}\} \cup \{\, \widetilde{P_1} \mid P_1 \neq \mathbf{0}$ is the 1-child of some $P \in V_{\mathrm{N}} \,\}$,
- $q_0 = \widetilde{\mathbf{r}}$,
- $F = \{\, \widetilde{P} \in \Gamma \mid \widetilde{P} = [P_1, \ldots, P_k]$ and $P_k = \mathbf{1} \,\}$,
- $\delta(\widetilde{P}, a_i) = \widetilde{R_i}$ if $\widetilde{P} = [P_1, \ldots, P_k]$ and $\tau(P_i) = \langle a_i, P_{i+1}, R_i \rangle$.

It is easy to see that $L_B(P) = L_{\mathbf{A}(B)}(\widetilde{P})$ for all $\widetilde{P} \in \Gamma$. This implies that if B is reduced, $\mathbf{A}(B)$ is minimum. Contrary to the translation from an ADFA into an equivalent SDD, this construction takes $O(|\Sigma||V_{\mathrm{N}}|)$ time. In fact, this is optimal. The following theorem implies that the reduced SDD can be about $|\Sigma|$ times more compact than the minimum ADFA for the same set of strings.

Theorem 5. *For any SDD $B = \langle \Sigma, V, \tau, \boldsymbol{r}, \mathbf{0}, \mathbf{1} \rangle$, one can construct in $O(|\Sigma||V_{\mathrm{N}}|)$ time the equivalent minimum ADFA $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$ such that $|\Gamma| \leq |V_{\mathrm{N}}| + 1$ and*

$$|\delta| \leq \begin{cases} |V_{\mathrm{N}}|(|V_{\mathrm{N}}|+1)/2 & \text{if } |V_{\mathrm{N}}| \leq |\Sigma|; \\ |\Sigma|(2|V_{\mathrm{N}}| - |\Sigma| + 1)/2 & \text{if } |V_{\mathrm{N}}| > |\Sigma|. \end{cases}$$

Moreover, there is an SDD B that admits no equivalent ADFA A for which the strict inequality holds.

Proof. The first claim, $|\Gamma| \leq |V_{\mathrm{N}}| + 1$, clearly holds by the conversion.

In order to establish the second part of the theorem, we give a variant of the construction of $\mathbf{A}(B)$. We define $\mathbf{C}(B)$ from B by replacing the definition of Γ in $\mathbf{A}(B)$ with $\Gamma = \{\, \widetilde{P} \mid P \in V - \{\mathbf{0}\} \,\}$. For $\mathbf{C}(B) = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, we prove the inequality by induction on $|V_{\mathrm{N}}|$. Clearly $\mathbf{A}(B)$ is not bigger than $\mathbf{C}(B)$, and thus this claim implies the theorem. In the following discussion, we ignore the root of B and the initial state of $\mathbf{C}(B)$, because it does not affect the discussion of their description size. For $|V_{\mathrm{N}}| = 1$, it is easy to see that the claim holds. Suppose that $|V_{\mathrm{N}}| > 1$. Let B' be obtained from B by deleting an arbitrary nonterminal node P that has no incoming edge.

If $|V_{\mathrm{N}}| \leq |\Sigma|$, we have $|\delta'| \leq (|V_{\mathrm{N}}|-1)|V_{\mathrm{N}}|/2$ by the induction hypothesis, where δ' denotes the transition set of $\mathbf{C}(B')$. By definition, $\mathbf{C}(B)$ can be obtained from $\mathbf{C}(B')$ by adding one state $\widetilde{P}$ and at most $|V_{\mathrm{N}}|$ outgoing edges from it. Hence

$$|\delta| \leq |\delta'| + |V_{\mathrm{N}}| \leq (|V_{\mathrm{N}}| - 1)|V_{\mathrm{N}}|/2 + |V_{\mathrm{N}}| = |V_{\mathrm{N}}|(|V_{\mathrm{N}}| + 1)/2.$$

If $|V_{\mathrm{N}}| > |\Sigma|$, we have $|\delta'| \leq |\Sigma|(2|V_{\mathrm{N}}| - |\Sigma| - 1)/2$ by the induction hypothesis. By definition, $\mathbf{C}(B)$ can be obtained from $\mathbf{C}(B')$ by adding one state $\widetilde{P}$ and at most $|\Sigma|$ outgoing edges from it. Hence

$$|\delta| \leq |\delta'| + |\Sigma| \leq |\Sigma|(2|V_{\mathrm{N}}| - |\Sigma| - 1)/2 + |\Sigma| = |\Sigma|(2|V_{\mathrm{N}}| - |\Sigma| + 1)/2.$$

We have proven the inequality.

In order to see that the above bound is tight, consider the reduced SDD and the minimum ADFA for the language $L_n = \{\, a_0^k a_{i_1} \dots a_{i_j} \mid 0 \leq k \leq n - |\Sigma|,\ 0 \leq j \leq \min\{m, n\},\ 1 \leq i_1 < \cdots < i_j \leq m \,\}$ over $\Sigma = \{a_0, \dots, a_m\}$ with $a_0 \prec a_1 \prec \cdots \prec a_m$. □

We note that if $a_m \prec \cdots \prec a_1 \prec a_0$, we have $|V'_{\mathrm{N}}| = |\delta|$ for the node set V' of the reduced SDD B' for L_n and the transition set δ of the minimum ADFA A for L_n in the proof of Theorem 5. Hence an order on Σ induces a reduced SDD that has asymptotically $|\Sigma|$ times more nodes than the one induced by another order on Σ.

Corollary 6. *For an order π on Σ and a finite language L over Σ, let $\mathbf{B}^\pi(L) = \langle \langle \Sigma, \pi \rangle, V^\pi, \tau^\pi, S, \mathbf{0}, \mathbf{1} \rangle$ be the reduced SDD for L that respects the order π over Σ. For any order π, ρ on Σ, we have $|V^\pi_{\mathrm{N}}| \leq |\Sigma||V^\rho_{\mathrm{N}}|$.*

Proof. Let δ be the transition set of the minimum automaton for L. By Theorem 3, $|V_{\mathrm{N}}^{\pi}| \leq |\delta|$. By Theorem 5, $|\delta| \leq |\Sigma||V_{\mathrm{N}}^{\rho}|$. Hence $|V_{\mathrm{N}}^{\pi}| \leq |\Sigma||V_{\mathrm{N}}^{\rho}|$. □

Through the conversion techniques presented above between ADFAs and SDDs and/or by Theorems 3 and 5, most known results on the size of minimum ADFAs can be translated into those on SDDs. A special case is where the set of all factors of a string is in concern. Let

$$\mathit{Fact}(w) = \{\, y \in \Sigma^* \mid w = xyz \text{ for some } x, z \in \Sigma^* \,\}.$$

The literature has intensively studied the *factor automata* for the set $\mathit{Fact}(w)$.

Theorem 7 (Blumer et al. [2], Crochemore [4]). *For $w \in \Sigma^*$, let Γ and δ be the state set and the transition set of the minimum ADFA for $\mathit{Fact}(w)$. Then $|\Gamma| \leq 2|w| - 2$ and $|\delta| \leq 3|w| - 4$.*

Corollary 8. *For $w \in \Sigma^*$, let V be the node set of the reduced SDD for $\mathit{Fact}(w)$. Then $|w| \leq |V_{\mathrm{N}}| \leq 3|w| - 4$.*

Proof. By Theorem 7 and Theorem 5. □

For $w = cb^n a$ with $a \prec b \prec c$, we have $|V_{\mathrm{N}}| = 3|w| - 4$.

Corollary 9. *For $w \in \Sigma^*$ and order π on Σ, let V^{π} be the node set of the reduced SDD for $\mathit{Fact}(w)$. Then $|V_{\mathrm{N}}^{\pi}| \leq |V_{\mathrm{N}}^{\rho}| + |w| - 1$. Moreover, there are w, π and ρ for which the equality holds.*

Proof. Let δ be the transition set of the minimum automaton for $\mathit{Fact}(w)$. We have $|V_{\mathrm{N}}^{\pi}| \leq |\delta|$ and $|\Gamma| \leq |V_{\mathrm{N}}^{\rho}| + 1$ by Theorems 3 and 5, respectively. Blumer et al. [2, Lemma 1.6] show that $|\delta| \leq |\Gamma| + |w| - 2$. Hence

$$|V_{\mathrm{N}}^{\pi}| \leq |\delta| \leq |\Gamma| + |w| - 2 \leq |V_{\mathrm{N}}^{\rho}| + |w| - 1.$$

In fact for $w = a^n b$, $\pi = \langle b \prec a \rangle$, $\rho = \langle a \prec b \rangle$, we have $|V_{\mathrm{N}}^{\pi}| = 2n-1$ and $|V_{\mathrm{N}}^{\rho}| = n-1$. □

4 Input- and Output-Sensitive Time-bounds for Binary Synthesis Operations

In this section, we consider time complexity of set operations on SDDs. In particular, given a binary set operation $\diamond \in \{\cup, \cap, \setminus, \ldots\}$, we consider the *synthesis problem* that receives two reduced SDDs P, Q and computes $R = P \diamond Q$, where $P \diamond Q$ denotes the reduced SDD such that $L(P \diamond Q) = L(P) \diamond L(Q)$. Bryant [3] presented in his seminal paper on BDDs, a recursive synthesis algorithm for all Boolean operations. Loekito *et al.* [12] gave its string-counterpart for union $\cup$ and difference $\setminus$. Below, we generalize the algorithm in [12] for a family of set operations, called *melding*, in the style of Knuth [11].

Global variable: *uniqtable*, *cache*: hash tables for triples and operations.

Algorithm $\mathsf{Meld}_\diamond(P, Q$: SDDs):
Output: The reduced SDD for the melding $P \diamond Q$ given $F_\diamond : \{0,1\}^2 \to \{0,1\}$;

```
 1: if (P = 0 or Q = 0 or P = Q)
 2:     if (F_◇[sign(P), sign(Q)] = 0) return 0; /* See text for F_◇. */
 3:     else if P ≠ 0 return P;
 4:     else if Q ≠ 0 return Q;
 5: else if ((R ← cache["Meld_◇(P,Q)"]) exists) return R;
 6: else
 7:     x ← P.lab; y ← Q.lab;
 8:     if (x ≺_Σ y) R ← Getnode(x, Meld_◇(P.0, Q), Meld_◇(P.1, 0));
 9:     else if (x ≻_Σ y) R ← Getnode(y, Meld_◇(P, Q.0), Meld_◇(0, Q.1));
10:     else if (x = y) R ← Getnode(x, Meld_◇(P.0, Q.0), Meld_◇(P.1, Q.1));
11:     cache["Meld_◇(P,Q)"] ← R;
12:     return R;
```

For convenience, we assume $\mathbf{1}.lab$ to be a symbol larger than any symbols in Σ.

Figure 3. An algorithm $\mathsf{Meld}_\diamond$ for built-in binary set operations $\diamond \in \{\cup, \cap, \setminus, \oplus, \ldots\}$.

4.1 The Family of Melding Operations

We give a family of binary set operations $\diamond$ called *melding* below. A *terminal operation table* is a binary Boolean function $F : \{0,1\}^2 \to \{0,1\}$ such that $F[0,0] = 0$. Clearly, there are exactly eight such tables. Let $\mathcal{O} = \{\cup, \cap, \setminus, /, \oplus, \emptyset, LHS, RHS\}$ be a set of names of set operations $\diamond : 2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$ on subsets Σ^*. We define $F_\diamond$ by: $F_\cup[x,y] = x \vee y$, $F_\cap[x,y] = x \wedge y$, $F_\setminus[x,y] = x \wedge \neg y$, $F_/[x,y] = \neg x \wedge y$, $F_\oplus[x,y] = x \oplus y$ (exclusive-or), $F_\emptyset[x,y] = 0$, $F_{LHS}[x,y] = x$, $F_{RHS}[x,y] = y$, where $x, y \in \{0,1\}$. For any SDD P, we define $\mathsf{sign}(P)$ to be 0 if $P = \mathbf{0}$ and 1 otherwise.

In Fig. 3, we give the algorithm $\mathsf{Meld}_\diamond$ that computes the reduced SDD $R = P \diamond Q$ for two SDDs P and Q given a terminal operation table $F_\diamond$. Clearly, the trivial operations $\emptyset$, LHS and RHS can be computed in constant time without $\mathsf{Meld}_\diamond$. Yet those are also uniformly described as $\mathsf{Meld}_\diamond$. A specified terminal operation table $F_\diamond$ uniquely determines melding operation $P \diamond Q$. In what follows, we assume that the inputs P and Q and the output R are built by using the same hash table *uniqtable*, where *uniqtable* is initialized with the empty relation before constructing P and Q. Moreover, our algorithm uses a hash table *cache* : $\mathsf{op} \times \mathsf{dom}^2 \to \mathsf{dom}$ that stores invocation patterns of operations for avoiding redundant computation, where op is the set of operation names. By a similar discussion in Knuth [11], we establish the following theorem. $\mathsf{Meld}_\diamond$ directly computes the output without producing redundant nodes.

Theorem 10 (correctness). *Let $\diamond \in \mathcal{O}$ be any of the eight operations. Given $F_\diamond$, the algorithm $\mathsf{Meld}_\diamond$ in Fig. 3 correctly computes the reduced SDD for $R = P \diamond Q$ exactly eight string set operations $P \diamond Q$, where the set operation $P \diamond Q$ is defined as follows:*

the union $P \cup Q$, the intersection $P \cap Q$,
the difference $P \setminus Q$, the inverse difference $P/Q = Q \setminus P$,
the symmetric difference $P \oplus Q = (P \setminus Q) \cup (Q \setminus P)$, the empty set $\emptyset$,
the left hand side $LHS(P,Q) = P$. the right hand side $RHS(P,Q) = Q$.

4.2 Input-Sensitive Complexity of Binary Synthesis

First, we start with input-sensitive analysis of the time complexity for the melding procedure. We prepare some necessary notations. Consider the algorithm $\mathsf{Meld}_\diamond$ of Fig. 3. Let us denote by $\mathsf{Meld}^0_\diamond$ and $\mathsf{Meld}^1_\diamond$ the first and second parts of the algorithm, that is, the top level if-clause and else-clause consisting of Lines 1 to 5 and Lines 6 to 12, respectively. For a procedure α, $\#\alpha(P,Q)$ denotes the number of times that α is executed during the computation of $\mathsf{Meld}_\diamond(P,Q)$. We assume that $|\mathbf{0}| = |\mathbf{1}| = 1$ for convenience.

Theorem 11 (input complexity of melding). *Let $\diamond$ be any melding operation. For reduced SDDs P and Q, the algorithm $\mathsf{Meld}_\diamond$ of Fig. 3 computes $R = P \diamond Q$ in $O(|P| \cdot |Q|)$ time and space.*

Proof. Consider the computation of $\mathsf{Meld}_\diamond(P,Q)$. Since the arguments P' and Q' of any subroutine call $\mathsf{Meld}_\diamond(P',Q')$, resp., are subgraphs of P and Q, the number of *distinct* calls for $\mathsf{Meld}_\diamond(P,Q)$ is at most $|P| \cdot |Q|$ (*Claim 1*). It also follows that *cache* has $O(|P| \cdot |Q|)$ entries. Since the table-lookup with *cache* at Line 5 eliminates duplicated calls, the $\mathsf{Meld}^1_\diamond$ can be executed at most once for each (P',Q'), and thus, we have $\#\mathsf{Meld}^1_\diamond \leq |P| \cdot |Q|$ (*Claim 2*). We observe that $\mathsf{Meld}_\diamond$ is called either (i) at the top-level or (ii) within $\mathsf{Meld}^1_\diamond$. Since exactly one of Line 8, 9, and 10 is executed in $\mathsf{Meld}^1_\diamond$, which contains at most two calls for $\mathsf{Meld}_\diamond$, we have $\#\mathsf{Meld}_\diamond \leq 2 \cdot \#\mathsf{Meld}^1_\diamond + 1$ (*Claim 3*). Combining Claims 2 and 3, we have that $\#\mathsf{Meld}_\diamond \leq 2 \cdot |P| \cdot |Q| + 1 = O(|P| \cdot |Q|)$. If each call of $\mathsf{Meld}_\diamond$ takes $O(1)$ time, then the time complexity is $O(|P| \cdot |Q|)$. On the other hand, each $\mathsf{Meld}_\diamond(P',Q')$ makes exactly one call for $\mathsf{Getnode}$ by adding a new node. Thus, the algorithm adds at most $|R| \leq \#\mathsf{Getnode} \leq \#\mathsf{Meld}_\diamond = O(|P| \cdot |Q|)$ nodes. Since the number of *cache*-entries is $O(|P| \cdot |Q|)$ and the function stack has depth no more than $\#\mathsf{Meld}_\diamond$, the space complexity is $O(|P| \cdot |Q|)$. □

From the proof of the above theorem, we have the following corollary.

Corollary 12 *For any melding operation $\diamond \in \mathcal{O}$, the reduced output size $|R|$ is bounded from above by $O(|P| \cdot |Q|)$.*

4.3 Pseudo Output Sensitive Complexity of Binary Synthesis

Next, we present output-sensitive analysis of the time complexity of the melding in the style of Wegener [21], which analyzed the time complexity of Boolean operations for BDDs based on the size of non-reduced BDDs. We define $R^* = P \diamond_* Q$ to be the (possibly non-reduced) SDD computed by $\mathsf{Meld}_\diamond$ equipped with the modification of $\mathsf{Getnode}$ in Fig. 3 by removing Line 1 and 2 for node-sharing and zero-suppress rules. Clearly, the non-reduced output size $|R^*|$ is bounded from above by $O(|P| \cdot |Q|)$.

Theorem 13 (output-sensitive complexity w.r.t. non-reduced output). *The reduced SDD for $R = P \diamond_* Q$ can be computed in $O(|R^*|)$ time and space by the algorithm $\mathsf{Meld}_\diamond$ in Fig. 3, where R^* is the non-reduced SDD for $P \diamond_* Q$.*

Proof. Consider the computation of $\mathsf{Meld}_\diamond$ of Fig. 3 equipped with $\mathsf{Getnode}^*$. Since each call of $\mathsf{Getnode}^*$ increases the output size by at least one, we have $\#\mathsf{Getnode}^* \leq |R^*|$ (*Claim 4*). Since exactly one of Line 8, 9, and 10 is executed in $\mathsf{Meld}^1_\diamond$ and it contains at least one call for $\mathsf{Getnode}$, we have $\#\mathsf{Meld}^1_\diamond \leq \#\mathsf{Getnode}^*$ (*Claim 5*). From

the proof for Theorem 11, we have $\#\mathsf{Meld}_\diamond \leq 2 \cdot \#\mathsf{Meld}^1_\diamond + 1$ (*Claim 3*). Combining Claims 3, 4, and 5 above, we now have $\#\mathsf{Meld}_\diamond \leq 2 \cdot \#\mathsf{Meld}^1_\diamond + 1 \leq 2 \cdot \#\mathsf{Getnode}^* + 1 \leq 2 \cdot |R^*| + 1 = O(|R^*|)$., and thus, we have the time complexity $O(|R^*|)$. Since *uniqtable* and *cache* contain at most $\#\mathsf{Getnode}^*$ and $\#\mathsf{Meld}_\diamond$ entries, resp., the space complexity follows from a similar argument to the proof for Theorem 11. □

4.4 A Lower Bound for the Time Complexity of Binary Synthesis

In the BDD community, there has been a strong belief that the quadratic input-sensitive complexities of the binary synthesis procedures for a number of variants of BDDs, including the BDDs and ZDDs, is output-linear time for most input instances, and there has been no super-linear lower bound for its time complexity. Recently, Yoshinaka et al. [22] show that this conjecture is not true for BDDs and ZDDs; They constructed an infinite sequence of input BDDs that demonstrated the quadratic lower bound for the time complexity of the melding for BDDs and ZDDs. Based on their discussion, below we show that the above quadratic input-sensitive complexity of the melding in terms of input size is optimal for SDDs in reality.

Theorem 14. *Let $\diamond$ be any melding operations. The algorithm $\mathsf{Meld}_\diamond$ of Fig. 3 requires $\Omega(|P| \cdot |Q|)$ time and space regardless of the output size, where P and Q are the input SDDs.*

Proof. Our example that the binary synthesis takes $O(|P| \cdot |Q|)$ time to compute $R = P \diamond Q$ where $|R|$ is linear in $|P| + |Q|$ is just a straightforward translation of the one from [22]. The theorem can be shown in a way similar to [22]. Here we give a rough sketch of the proof. Let $\Sigma = \{0, 1\}$. For a fixed positive integer n, we define

$$S = \{\, x_1 y_1 \cdots x_n y_n z_1 \cdots z_m \in \{0,1\}^{2n+m} \mid x_{\beta(z_1 \cdots z_m)} = 1 \,\},$$
$$T = \{\, x_1 y_1 \cdots x_n y_n z_1 \cdots z_m \in \{0,1\}^{2n+m} \mid y_{\beta(z_1 \cdots z_m)} = 1 \,\},$$

where $m = \lceil \log n \rceil$ and

$$\beta(z_1 \cdots z_m) = \begin{cases} 1 + \sum_{k=1}^{m} 2^{k-1} z_k & \text{if } \sum_{k=1}^{m} 2^{k-1} z_k < n; \\ 1 & \text{otherwise.} \end{cases}$$

We have

$$S \diamond T = \{\, x_1 y_1 \cdots x_n y_n z_1 \cdots z_m \in \{0,1\}^{2n+m} \mid F_\diamond[x_{\beta(z_1 \cdots z_m)}, y_{\beta(z_1 \cdots z_m)}] = 1 \,\}.$$

Let P and Q be the reduced SDD for S and T, resp.

We first show that $|P|, |Q|, |R| = O(2^n)$. It is easy to see that every node in P and Q represents a set of strings of a fixed length, since all strings in S and T have the same length $2n + m$. We define the *level* of a node to be $2n + m - k$ if the node represents a set of strings of length k. Since the membership of $x_1 y_1 \cdots x_n y_n z_1 \cdots z_m$ to S does not depend on any of y_i, it is not hard to see that there are at most $O(2^k)$ nodes of level $2k$ for $0 \leq k < n$. The number of nodes of level $2k + 1$ is at most twice as big as that of level $2k$. On the other hand, since there are at most 2^{2^k} distinct sets of strings of length k, there are at most $|\Sigma| \cdot 2^{2^k}$ nodes of level $2n + m - k$ for $0 \leq k \leq m = \lceil \log n \rceil$. All in all, $|P| = O(2^n)$. Similarly $|Q| = O(2^n)$. It is easy to see that for any $x_i, y_i, x'_i, y'_i \in \{0, 1\}$ such that $F_\diamond[x_i, y_i] = F_\diamond[x'_i, y'_i]$, we have

Data	Size (byte)	#line	#unique line	Ave. line len (byte)	$\|\Sigma\|$
BibleAll	4,047,392	30,383	30,129	133.2	62
BibleBi	7,793,268	767,854	154,479	10.1	27
Ecoli	4,638,690	1	1	4,638,690.0	4

Table 1. Outline of data sets

Data	Size (Kilo node)						Time (sec)			
SDD input	H_1	H_2	$\cup$	$\cap$	$\setminus$	$\triangle$	$\cup$	$\cap$	$\setminus$	$\triangle$
BibleAll(Fac)	3099	3082	6110	417	3415	3388	0.67	0.44	0.59	0.58
BibleBi	101	115	167	36	82	97	0.06	0.00	0.00	0.00
Ecoli(Fac)	4973	4970	9938	654	6346	6347	1.63	1.09	1.42	1.41

Table 2. Output size and running time of algorithms for binary synthesis

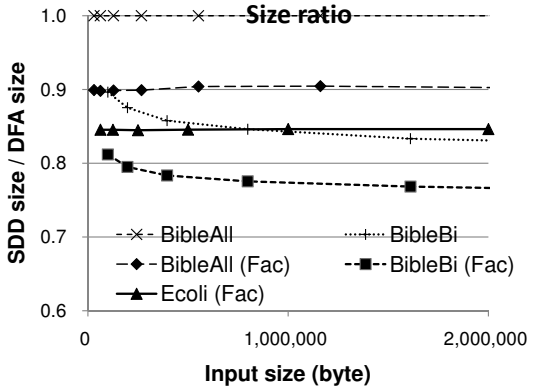


Figure 4. Ratio between the sizes SDDs and DFAs in binary format

$w_1 x_i y_i w_2 \in S \diamond T$ iff $w_1 x'_i y'_i w_2 \in S \diamond T$ for any $w_1 \in \{0,1\}^{2k}, w_2 \in \{0,1\}^{2n+m-k-2}$ with $k < n$. Hence we have $|R| = O(2^n)$ by a discussion similar to the one for $|P|, |Q| = O(2^n)$.

Second we show that $\#\mathsf{Meld}_\diamond \geq 2^{2n}$. For $w \in \{0,1\}^{2n}$, let P_w denote the node of P such that $L(P_w) = \{\, w' \mid ww' \in S \,\}$. In fact P has such a node for each w. Similarly we let Q_w be such that $L(Q_w) = \{\, w' \mid ww' \in T \,\}$. By definition, the algorithm calls $\mathsf{Meld}_\diamond(P_w, Q_w)$ for each w. Moreover, $P_{x_1 \cdots y_n} \neq P_{x'_1 \cdots y'_n}$ whenever $x_i \neq x'_i$ for some i and $Q_{x_1 \cdots y_n} \neq Q_{x'_1 \cdots y'_n}$ whenever $y_i \neq y'_i$ for some i. Therefore, for distinct $w, w' \in \{0,1\}^{2n}$, the pairs $\langle P_w, Q_w \rangle$ and $\langle P_{w'}, Q_{w'} \rangle$ are distinct. This means that $\#\mathsf{Meld}_\diamond \geq 2^{2n}$. □

5 Experiments

This section presents our experimental results on SDDs. Our first experiment has constructed SDDs and DFAs for the same sets of strings of real data and compared their sizes. Secondly we have implemented the binary synthesis algorithm $\mathsf{Meld}_\diamond$ and computed different binary operations on sets over SDDs.

Setting. The data sets used in our experiments are summarized in Table 1. BibleAll and BibleBi are sets of all sentences and all word bi-grams drawn from an English text `bible.txt` and Ecoli is a single DNA string in `ecoli.txt` in Canterbury corpus[3]. We implemented our shared and reduced SDD environment on the top of

[3] `http://corpus.canterbury.ac.nz/resources/`

the *SAPPORO BDD package* [17] for BDDs and ZDDs written in C and C++, where each node is encoded in a 32-bit integer and a node triple occupies approximately 30 bytes in average including hash entries in *uniqtable*. We also used another implementation of SDD environment in functional language *Erlang*. Experiments were run on a PC (Intel Core i7, 2.67 GHz, 3.25 GB memory, Windows XP SP3). About 1.5 GB of memory was allocated to the SDD environment in maximum.

Exp 1: Comparison of the size of indexes. Figure 4 shows the sizes of SDDs and DFAs for different sets of strings, where BibleAll (Fac), BibleBi (Fac) and Ecoli (Fac) mean the sets of all factors of sequences in the respective input files. We see that a minimal SDD is 0 to 23 percent more succinct than the equivalent minimal ADFA in binary format. In particular, the size ratio for factor sets is even smaller than that for the original string data. SDDs can search strings as fast as DFAs where edges of DFAs are represented by linked list.

Exp 2: Binary synthesis. We divided the source texts into two parts, the first half H1 and the second H2, and then performed $\mathsf{Meld}_\diamond$ on those parts for $\diamond \in \{\cup, \cap, \setminus, /\}$. The results are presented in Table 2. It took less than seconds to compute set operations $\diamond$ on two SDDs with around three to four millions of nodes each. The output size of $\mathsf{H1} \cup \mathsf{H2}$ is much larger than that of $\mathsf{H1} \cap \mathsf{H2}$, but the running time is not different that much.

Overall, we conclude that the shared and reduced SDD environment with the above algorithms is a practical choice for storing and manipulating string sets in large-scale string applications.

6 Conclusion

In this paper, we consider the class of sequence binary decision diagrams (SDDs) proposed by Loekito *et al.* [12], and studied two fundamental problems on sequence binary decision diagrams: the relationship to acyclic automata and the complexities of the binary synthesis operation. In Sec. 4, we showed the quadratic time complexity of the $\mathsf{Meld}_\diamond$ algorithm. In [7], it is shown that the $\mathsf{Meld}_\diamond$ runs in input linear time if one of the argument is the minimal SDD of linear shape corresponding to a string. Therefore, it would be an interesting future problem to study special cases that $\mathsf{Meld}_\diamond$ has input linear time complexity. It would be another problem to apply SDDs for studying the dynamic versions of sequence analysis problems such as the maximal repeat problem and the consistent string problem [9].

Acknowledgements

The authors are grateful to anonymous referees for many useful comments and suggestions that improve the quality of this paper. They would like to thank Takashi Horiyama, Takeru Inoue, Jun Kawahara, Takuya Kida, Toshiki Saitoh, Yasuyuki Shirai, Kana Shimizu, Yasuo Tabei, Koji Tsuda, Takeaki Uno, and Thomas Zeugmann for their discussions and valuable comments. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, MEXT/ JSPS Global COE Program, FY2007–2011, and ERATO MINATO Discrete Structure Manipulation System Project, JST.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas: *The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci.*, 40, 31–55, 1985.
3. R. E. Bryant: *Graph-based algorithms for boolean function manipulation. IEEE. Trans. Comput.*, C-35(8), 677–691, 1986.
4. M. Crochemore: *Transducers and repetitions. Theor. Comput. Sci.*, 45(1), 63–86, 1986.
5. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*. Cambridge, 2007.
6. J. Daciuk, S. Mihov, B. W. Watson, R. E. Watson: *Incremental construction of minimal acyclic finite-state automata. Computational Linguistics*, 26(1), 2000.
7. S. Denzumi, R. Yoshinaka, S. Minato, and H. Arimura: *Efficient algorithms on sequence binary decision diagrams for manipulating sets of strings. Technical Report*, DCS, Hokkaido U., TCS-TR-A-11-53, April 2011. (submitting)
8. R. Giegerich, S. Kurtz, and J. Stoye: *Efficient implementation of lazy suffix trees. Software Practice and Experience*, 33, 1035–1049, 2003.
9. D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
10. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Formal Language Theory, 2nd edition*. Addison-Wesley, 2001.
11. D. E. Knuth: *The Art of Computer Programming, vo.4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley, 2009.
12. E. Loekito, J. Bailey, and J. Pei: *A Binary decision diagram based approach for mining frequent subsequences. Knowl. Inf. Syst.*, 24(2), 235-268, 2009.
13. C. L. Lucchesi, and T. Kowaltowski: *Applications of finite automata representing large vocabularies. Software Practice and Experience*, 23(1), 15–30, 1993.
14. U. Manber and E. W. Myers: *Suffix arrays: a new method for on-line string searches. SIAM J. Comput.*, 22(5), 935–948, 1993.
15. E. M. McCreight: *A space-economical suffix tree construction algorithm. J. ACM*, 23, 262–272, 1976.
16. S. Minato: *Zero-suppressed BDDs and their applications. International Journal on Software Tools for Technology Transfer*, 3(2), 156-170, Springer, 2001.
17. S. Minato: *SAPPORO BDD package*. DCS, Hokkaido University, unreleased, 2011.
18. M. Mohri: *On some applications of finite-state automata theory to natural language processing. Natural Language Engineering*, 2(1), 61–80, 1996.
19. M. Mohri, P. Moreno, and E. Weinstein: *Factor automata of automata and applications. Proc. CIAA 2007*, 168-179, 2007.
20. D. Perrin: *Finite Automata. Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen (Eds.), Elsevier and MIT Press, 1–57, 1990.
21. I. Wegener: *Branching Programs and Binary Decision Diagrams: Theory and Applications*. SIAM, 2000.
22. R. Yoshinaka J. Kawahara, S. Denzumi, H. Arimura, and S. Minato: *Counter examples to the long-standing conjecture on the complexity of BDD binary operations. Technical Report*, DCS, Hokkaido U., TCS-TR-A-11-52, April 2011. (submitting to international journal)

Observations On Compressed Pattern-Matching with Ranked Variables in Zimin Words

Radosław Głowinski[2] and Wojciech Rytter[1,2]

[1] Department of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
rytter@mimuw.edu.pl
[2] Faculty of Mathematics and Informatics,
Nicolaus Copernicus University, Toruń, Poland
glowir@mat.umk.pl

Abstract. Zimin words are very special finite words which are closely related to the pattern-avoidability problem. This problem consists in testing if an instance of a given pattern with variables occurs in almost all words over any finite alphabet. The problem is not well understood, no polynomial time algorithm is known and its NP-hardness is also not known. The pattern-avoidability problem is equivalent to searching for a pattern (with variables) in a Zimin word. The main difficulty is potentially exponential size of Zimin words. We use special properties of Zimin words, especially that they are highly compressible, to design efficient algorithms for special version of the pattern-matching, called here *ranked matching*. It gives a new interpretation of Zimin algorithm in compressed setting. We discuss the structure of rankings of variables and compressed representations of values of variables.

1 Introduction

The research on pattern avoidability started in late 70's in the papers by Bean, Ehrenfeucht and McNaulty [1], and independently by Zimin [5]. In the avoidability problem two disjoint finite alphabets, $A = \{a, b, c, \dots\}$ and $V = \{x_1, x_2, x_3, \dots\}$ are given, the elements of A are letters (constants) and the elements of V are variables. We denote the empty word by ε. A pattern π is a sequences of variables. The language of a pattern with respect to an alphabet A consists of words $h(\pi)$, where h is any non-erasing morphism from V^* to A^+. We say that word w encounters pattern π (or pattern occurs in this word) when there exists a morphism h, such that $h(\pi)$ is a subword of w. In the other case w avoids π.

The pattern π is unavoidable on A if every long enough word over A encounters π, otherwise it is avoidable on A. If π is unavoidable on every finite A then π is said to be unavoidable.

Example 1. The pattern $\alpha\alpha$ is avoidable over $A = \{a, b, c\}$. Let

$$u = abcacbabcbac\dots$$

be the infinite word generated by morphism

$$\mu : a \to abc, b \to ac, c \to b$$

starting from the letter a. The word u avoids the pattern $\alpha\alpha$ (u is square-free), as shown in [4]. Hence $\alpha\alpha$ is not unavoidable, however $\alpha\beta\alpha$ is unavoidable.

Radosław Głowinski, Wojciech Rytter: Observations On Compressed Pattern-Matching with Ranked Variables in Zimin Words, pp. 162–172.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

The crucial role in avoidability problems play the words introduced by Zimin [5], called Zimin words, and denoted here by Z_k.

Definition 2. (of Zimin words) *Let*

$$Z_1 = 1,\ Z_k = Z_{k-1}\, k\, Z_{k-1}$$

Example 3. $Z_1 = 1,\ Z_2 = 121,\ Z_3 = 1213121,\ Z_4 = 121312141213121$

Observe that these words are exponentially long, however they have a very simple structure implying many useful properties. Define the Zimin morphism

$$\mu :\ 1 \to 121,$$

$$i \to i+1\ (\forall\ i > 1).$$

Fact 1

- *The morphism μ generates next Zimin word by mapping each letter according to μ. In other words: $Z_k = \mu(Z_{k-1})$.*
- *Each Zimin word, considered as a pattern, is unavoidable. Moreover it is a longest unavoidable pattern over k-th letter alphabet. There exists only one (up to letter permutation) unavoidable pattern of length $2^k - 1$ over a k-th letter alphabet and it is Z_k.*

The main property of Zimin words is that the avoidability problem is reducible to pattern-matching in Zimin words, see [5], [2].

Lemma 4. *π is an unavoidable pattern if and only if π occurs in Z_k, where k is the number of distinct symbols occurring in π.*

2 Compact representation of pattern instances

We say that a sequence u is j-interleaved iff for each two adjacent elements of u exactly one is equal to j.

The Zimin word Z_k can be alternatively defined as follows:

(A) Z_k starts with 1 and ends with 1;
(B) $|Z_k| = 2^k - 1$;
(C) For each $1 \le j \le k$ after removing all elements smaller than j the obtained sequence is j-interleaved.

Observation 1 *If we have a string $u \in \{1, 2, \ldots, k\}^+$, then u is a factor (subword) of Z_k iff it satisfies the condition (C).*

We omit the proof.

This gives a simple linear time algorithm to check if an explicitly given sequence is a factor of Z_k. However we are dealing with patterns, and instances of the pattern can be exponential with respect to the length of the pattern and the number of distinct variables. The instance is given by values of each variable which are factors of Z_k. Hence we introduce compact representation of factors of Zimin words.

We partition u into $w_1\, m\, w_2$, where m is a highest number in w (in every subword of Zimin word the highest number occurs exactly once). Then for each element i of w_1, respectively w_2, we remove i if there are larger elements to the left and to the right of this element. In other words if there is a factor $s\,\alpha\, i\,\beta\, t$, with $i < s$, $i < t$, we remove the element i. Denote by $compress(u)$ the result of removing all redundant i in u.

Observation 2 *$compress(u)$ uniquely encodes a factor of a Zimin word.*

Example 5.

$$compress(2141213121512131) = 24531$$

$$compress(Z_4) = compress(121312141213121) = 1234321$$

$$\begin{array}{ccccccccccccccccccc} & & \alpha & & & & \beta & & & & \gamma & & & & \beta & & & & \\ 1 & 2 & 1 & 3 & 1 & 2 & 1 & 4 & 1 & 2 & 1 & 3 & 1 & 2 & 1 & 4 & 1 & 2 & 1 & 3 & 1 & 1 & 2 & 1 \\ & & 1 & 3 & 2 & & & 1 & 4 & 3 & 1 & & & 2 & 5 & 3 & 2 & & & 1 & 4 & 3 & 1 & & \end{array}$$

Figure 1. Example of a compact representation. In the first line there is a pattern, in the second uncompressed valuation of variables and in the third compressed valuation.

Fact 2 *For any $u \in \{1, 2, \ldots, k\}^+$ the first and the last elements of u and $compress(u)$ are respectively equal.*

Notice that compressed representation of any subword of Z_k has at most $2k-1$ letters and the representation of all variables requires $O(k^2)$ memory (under the assumption that only $O(1)$ space is necessary for representing each number).

For a valuation (morphism) h of the variables by its ranking function R_h we mean the function which assigns to each variable x_i its rank: $R_h(x_i)$ denotes the maximal letter in $h(x_i)$.

For a pattern π and a given valuation of variable by $\pi_{(i)}$ we mean the pattern with variables of ranks smaller than i removed from π.
By $\mathcal{V}_i(\pi)$ we define the set of variables from π with rank i.
For two strings u, w we write $u \leq w$ iff u is a subword of w. By $\pi \rightarrow_h Z_k$ we mean that $h(\pi) \leq Z_k$ and by $\pi \rightarrow Z_k$ we mean that for some morphism h, $h(\pi) \leq Z_k$.

We extend the definition of j-interleaved sequence to patterns.

Definition 6. *Pattern π with compact representation of variables is j-interleaved iff for any two adjacent variables xy in π either x ends with j or y starts with j, in its compressed form.*

Theorem 7. *Assume we are given a pattern π of size n with k variables and a compact representation of values of the variables. Then we can check if the given compressed instance of π occurs in Z_k in time $O(nk)$.*

Proof. Assume we have an instance of the pattern $\pi = x_1 \cdots x_n$ with variables given in compressed form, such that for $1 \leq j \leq k$ π is j-interleaved when we remove all elements smaller than j from the compressed forms of the variables.

We define function red_i on strings over $\{1, 2, \ldots, k\}$ which removes from the string all elements smaller than i. Performing red_i on patterns removes all elements smaller than i from the compact representations of the variables.

Using Observation 2 we obtain unique full representation of variables denoted $y_1 = val(x_1), \ldots, y_n = val(x_n)$. Since y_i is a factor of a Zimin word $red_j(y_i)$ is j-interleaved. Because $red_j(\pi)$ is j-interleaved using Fact 2 we see that the concatenation $red_j(y_1)red_j(y_2)\cdots red_j(y_n)$ is j-interleaved (as a string). From Observation 1 we know that it is a factor of Z_k.

We showed that to determine if an instance of π occurs in Z_k we only need to check if for every $1 \leq j \leq k$ pattern $red_j(\pi)$ is j-interleaved. This can be done in total time $O(n\,k)$. □

3 Some properties of Zimin words and free sets

The ranking sequence associated with π is the sequence of ranks of consecutive variables in π.

Assume for a while that our pattern π is a permutation of n variables and we ask for the set of possible ranking sequences.

The ranking sequence has many useful properties:

1. Between every two occurrences of the same number a in ranking sequence there should be a number larger than a.
2. The ranking function is not necessarily injective (one to one).
3. If x_1x_2 and x_1x_3 are subwords of a pattern π, $x_1, x_2, x_3 \in V$ and $rank(x_3) < rank(x_2) < rank(x_1)$ and there exists a morphism φ that morphs p into Z_k then $\varphi(x_3)$ is a proper prefix of $\varphi(x_2)$.

Let $\overline{r}(\pi, h)$ be the set of ranks of variables in π for the valuation h. For example, for a pattern $\pi = \alpha\beta\alpha\gamma\beta\alpha$ and a valuation $h(\alpha) = 1, h(\beta) = 2, h(\gamma) = 31$ ranking sequence is $(1, 2, 1, 3, 2, 1)$ and $\overline{r}(\pi, h) = \{1, 2, 3\}$.

The following three facts are consequences of the proof of Zimin theorem (see [5] for details).

Lemma 8.

1. *If pattern π is unavoidable then there exists a morphism h such that $h(\pi)$ occurs in Z_k and $\min \overline{r}(\pi, h) = 1$.*
2. *If $\pi \rightarrow_h Z_k$ and $\min \overline{r}(\pi, h) = j + 1 > 1$ then there exists morphism g such that $\pi \rightarrow_g Z_k$ and $\overline{r}(\pi, g) = \{r - j : r \in \overline{r}(\pi, h)\}$.*
3. *If $\pi \rightarrow Z_k$ then there exists morphism h such that the set of ranks is an interval: $\overline{r}(\pi, h) = \{1, \ldots, m\}$, for some $1 \leq m \leq k$.*

We present Zimin algorithm based on free sets and σ-deletions.

Definition 9. *$F \subseteq V$ is a **free set** for $\pi \in V^+$ if and only if there exist sets $A, B \subseteq V$ such that $F \subseteq B \setminus A$ where, for all $xy \leq \pi, x \in A$ if and only if $y \in B$.*

Definition 10. *The mapping σ_F is a **σ-deletion** of π if and only if $F \subseteq V$ is a free set for π and $\sigma_F : V \rightarrow V \cup \{\varepsilon\}$ is defined by*

$$\sigma_F(x) = \begin{cases} x & \text{if } x \notin F \\ \varepsilon & \text{if } x \in F \end{cases}$$

The proof of the following fact can be found in [3], Zimin's algorithm is based on this fact.

Lemma 11. *π is an unavoidable pattern if and only if π can be reduced to ε by a sequence of σ-deletions.*

Unfortunately it is insufficient to remove only singleton free sets. There are patterns, which require the removing more than one element free sets, for example the pattern

$$\alpha\beta\alpha\gamma\alpha'\beta\alpha\gamma\alpha\beta\alpha'\gamma\alpha'\beta\alpha'$$

Therefore we can have exponentially many choices for free sets.

Lemma 12. *If $\pi \to Z_k$ then $R_1(\pi)$ is a free set.*

Proof. To satisfy the definition of a free set we need to give sets A and B, such that $R_1(\pi) \subset B \setminus A$ and all predecessors of variables from B are in A, all successors of variables from A are in B. We put all variables starting with 1 as the set B and all variables that do not end with 1 as set A. □

Lemma 13. *If $\pi \to Z_k$ then $\pi_{(2)} \to Z_{k-1}$.*

Proof. If $\pi \to Z_k$ then there exists morphism h, such that $h(\pi)$ is a subword of Z_k. $\mathcal{V}_1(\pi)$ is a set of variables x, such that $h(x) = 1$. We shall notice that if we remove all 1 from Z_k we obtain Z_{k-1}. We define a new morphism g for all variables from $\mathcal{V} \setminus \mathcal{V}_1$ as $g(x) = f(h(x))$, where f is a function that removes all occurrences of 1 from a word. Now we will show that $g(\pi_{(2)})$ is a subword of Z_{k-1}. We see that $g(\pi_{(2)}) = f(h(\pi_{(2)}) = f(h(\pi))$ because π differs from $\pi_{(2)}$ only in variables that equal 1. So occurrence $g(\pi_{(2)})$ equals $h(\pi)$ with all 1 deleted and $g(\pi_{(2)})$ is a subword of Z_{k-1}. □

Theorem 14. *A pattern π occurs in Z_k if and only if $\mathcal{V}_1(\pi)$ is a free set and $\pi_{(2)} \to Z_{k-1}$.*

Proof. "⇒" This is a consequence of Lemma 8 and Lemma 13.
"⇐" It follows from proof of Zimin theorem and can be found in [5]. □

4 Ranked pattern-matching

It is not known (and rather unlikely true) if the pattern-matching in Zimin words is solvable in polynomial time. We introduce the following polynomially solvable version of this problem.

Compressed Ranked Pattern-Matching in Zimin Words:

Input: given a pattern π with k variables and the ranking function R
Output: a compressed instance of an occurrence of π in Z_k with the given ranking function, or information that there is no such valuation, the values of variables are given in their compressed form

The algorithm for the ranked pattern matching can be used as an auxiliary tool for pattern-matching without any ranking function given. We can just consider all *sensible* ranking functions. It gives an exponential algorithm since we do not know what the rank sequence is. Although exponential, the set of *sensible* ranking sequences can be usefully reduced due to special properties of realizable rankings.

4.1 Application of 2-SAT

In one iteration we have not only to check if the set of variables of the smallest rank i is a free set but we have to compute which of them start/end with a letter i. However in a given iteration the letter i can be treated as '1'.

In our algorithms we will use the function $FirstLast(\pi, W)$ which solves an instance of 2-SAT problem. It computes which variables should start-finish with the smallest rank letter, under the assumption that variables from W equal the smallest rank letter, to satisfy local properties of the Zimin word.

In the function we can treat the smallest rank letter as 1. In Zimin word there are no two adjacent '1'. This leads to the fact: for any adjacent variables xy from π either x ends with '1' or y starts with '1'. If for a given pattern we know that some variables start with '1' (or end with '1') we deduce information about successors of this variable (or predecessors). For example if we have a pattern $\beta\alpha\beta\gamma$ and we know that α starts with '1' we deduce that β does not end with '1' and then deduce that γ starts with '1'. For a given set of variables that start and end with '1' we can deduce information about all other variables in linear time (with respect to the length of the pattern).

For every variable x from the pattern we introduce two logic variables: x^{first} is true iff x starts with '1', x^{last} is true iff x ends with '1'. Now for any adjacent variables xy we create disjunctions $x^{last} \vee y^{first}$ and $\neg x^{last} \vee \neg y^{first}$. If we write the formula

$$\begin{aligned} F =&(x_1^{last} \vee x_2^{first}) \wedge (\neg x_1^{last} \vee \neg x_2^{first}) \wedge (x_2^{last} \vee x_3^{first}) \wedge (\neg x_2^{last} \vee \neg x_3^{first}) \wedge \cdots \\ &\cdots \wedge (x_{n-1}^{last} \vee x_n^{first}) \wedge (\neg x_{n-1}^{last} \vee \neg x_n^{first}) \end{aligned}$$

we have an instance of 2-SAT problem. For the variables $y_1, \ldots, y_s \in W$, that we know that valuate as '1', we expand our formula to $F \wedge y_1^{first} \wedge y_1^{last} \wedge \cdots \wedge y_l^{first} \wedge y_s^{last}$.

Example 15. If for a pattern $\beta\alpha\beta\gamma\alpha$ we know that valuation of α will be '1' we produce formula

$$\begin{aligned} &(\beta^{first} \vee \alpha^{last}) \wedge (\neg\beta^{first} \vee \neg\alpha^{last}) \wedge (\alpha^{first} \vee \beta^{last}) \wedge \\ \wedge&(\neg\alpha^{first} \vee \neg\beta^{last}) \wedge (\beta^{first} \vee \gamma^{last}) \wedge (\neg\beta^{first} \vee \neg\gamma^{last}) \wedge \\ \wedge&(\gamma^{first} \vee \alpha^{last}) \wedge (\neg\gamma^{first} \vee \neg\alpha^{last}) \wedge (\alpha^{first}) \wedge (\alpha^{last}) \end{aligned}$$

In the general case there can be many solutions of the formula but in our example the only solution is: $\alpha^{first} = \alpha^{last} = \gamma^{first} = true$, $\beta^{first} = \beta^{last} = \gamma^{last} = false$, which means that α starts and ends with '1', β starts and ends with non-'1', γ starts with '1' and ends with non-'1'.

A positive solution to this problem is necessary for the existence of a valuation val of the variables from pattern π, such that $val(\pi) \leq Z_k$ and each variable x starts (ends) with '1' iff x^{first} is true (resp. x^{last} is true) in the solution.

It is well known that 2-SAT can be computed efficiently, consequently:

Lemma 16. *We can execute $FirstLast(\pi, W)$ in linear time.*

4.2 The algorithm: compressed and uncompressed versions

Now we present two versions of the algorithm deciding if pattern π occurs in Zimin word with given rank sequence and computing the values of variables, if there is an occurrence. First of these algorithms uses uncompressed valuations of variables and uses exponential space and second one operates on compressed valuations. If the answer is positive algorithms give valuations of variables, i.e. morphism *val* such that $val(\pi) \leq Z_k$.

Denote by $alph(\pi)$ the set of symbols (variables) in π. Let π be the pattern with given rank sequence which maximal rank equals K, $|\pi| = n, |alph(\pi)| = k$. We define the operation $firstdel(i, s), lastdel(i, s)$ of removing the first, last letter from s, respectively, if this letter is i (otherwise nothing happens), similarly define $firstinsert(i, s), lastinsert(i, s)$: inserting letter i at the beginning or at the end of s if there is no i.

In general, maximal rank can occur multiple times, but in this case, because of properties mentioned earlier in this paper, the pattern with this rank sequence is avoidable and cannot occur in Zimin word. Both algorithms assume that there is only one occurrence of the maximal rank (we denote by x_K the variable with the maximal rank).

Algorithm Uncompressed-Embedding(π, Z_K)
$K :=$ maximal rank
$\mathcal{V}_i$ is the set of variables of rank i, for $1 \leq i \leq K$
$val(x_K) := 1$;
for $i = K - 1$ **downto** 1 **do**
  for each $x \in \mathcal{V}_i$ **do** $val(x) := 1$;
  if $FirstLast(\pi_{(i)}, \mathcal{V}_i)$ **then**
    comment: *we know now which variables in* $\pi_{(i)}$ *start/finish with* i *due to evaluation of a corresponding 2SAT*
    for each $x \in \mathcal{V}_{i+1} \cup \mathcal{V}_{i+2} \cup \cdots \cup \mathcal{V}_K$ **do**
      $val(x) := \mu(val(x))$;
      if $not(x^{first})$ then $val(x) := firstdel(1, val(x))$;
      if $not(x^{last})$ then $val(x) := lastdel(1, val(x))$;
      if x^{first} then $val(x) := firstinsert(1, val(x))$;
      if x^{last} then $val(x) := lastinsert(1, val(x))$;
  else return false;

The next algorithm is a space-efficient simulation of the previous one. We are not using the morphism μ, instead of that the values of variables are maintained in a compressed form, we are adding to the left/right decreasing sequence of integers.

Algorithm Compressed-Embedding(π, Z_K)
$K :=$ maximal rank
$\mathcal{V}_i$ is the set of variables of rank i, for $1 \leq i \leq K$
$val(x_K) := K$;
for $i = K - 1$ **downto** 1 **do**
  for each $x \in \mathcal{V}_i$ **do** $val(x) := k$;
  if *FirstLast*$(\pi_{(i)}, \mathcal{V}_i)$ **then**
    for each $x \in \mathcal{V}_{i+1} \cup \mathcal{V}_{i+2} \cup \cdots \cup \mathcal{V}_K$ **do**
      if x^{first} then $val(x) :=$ *firstinsert*$(i, val(x))$;
      if x^{last} then $val(x) :=$ *lastinsert*$(i, val(x))$;
  else return false;

Example
Below we present an example of the Uncompressed-Embedding algorithm for

$$\pi = \quad \delta\ \alpha\ \gamma\ \beta\ \lambda\ \gamma\ \alpha\ \delta\ \alpha\ \gamma\ \beta\ \alpha$$

with the rank sequence $\quad 4\ 1\ 3\ 2\ 5\ 3\ 1\ 4\ 1\ 3\ 2\ 1$

$$\overbrace{\lambda_{\downarrow}}^{1}$$

First we set the variable with the highest rank. $val(\lambda) = 1$

$$\overbrace{\delta_{\downarrow}\lambda\delta_{\downarrow}}^{121}$$

$i = 4$. We set $val(\delta) = 1$ and morph $val(\lambda) = 121$.
From solution of 2-SAT we know that δ starts and ends with '1',
λ starts and ends with non-'1', we set $val(\lambda) = 2$.

$$\overbrace{\delta\gamma_{\downarrow}}^{121}\ \overbrace{\lambda\gamma_{\downarrow}}^{3}\ \overbrace{\delta\gamma_{\downarrow}}^{121}$$

$i = 3$. We have $val(\gamma) = 1, val(\delta) = 121, val(\lambda) = 3$.
From solution of 2-SAT: γ starts and ends with '1',
δ and λ start and end with non-'1',
therefore we set $val(\lambda) = 3$, $val(\delta) = 2$

$$121\ \overbrace{\delta}^{3}\ \overbrace{\gamma\ \beta_{\downarrow}\ \lambda}^{121}\ \overbrace{\gamma}^{4}\ \overbrace{\delta}^{121}\ \overbrace{\gamma\beta_{\downarrow}}^{3}\ \overbrace{\phantom{\gamma\beta_{\downarrow}}}^{121}$$

$i = 2 : val(\beta) = 1, val(\gamma) = 121, val(\delta) = 3, val(\lambda) = 4$.
From solution of 2-SAT: β starts and ends with '1', γ, δ start with '1',
end with non-'1', λ starts and ends with non='1'
Therefore $val(\gamma) = 12, val(\delta) = 13, val(\lambda) = 4$.

$$1213\ \overbrace{\delta\,\alpha_{\downarrow}\ \gamma}^{1214}\ \overbrace{\phantom{\delta\,\alpha_{\downarrow}\ \gamma}}^{1213}\ \overbrace{\beta}^{121}\ \overbrace{\lambda}^{5}\ \overbrace{\gamma\,\alpha_{\downarrow}}^{1213}\ \overbrace{\delta\,\alpha_{\downarrow}}^{1214}\ \overbrace{\gamma}^{1213}\ \overbrace{\beta\alpha_{\downarrow}}^{121}$$

$i = 1 : val(\alpha) = 1, val(\beta) = 121, val(\gamma) = 1213,$
$val(\delta) = 1214, val(\lambda) = 5$.
From solution of 2-SAT: α, λ start and ends with '1',
β starts with 1, ends with non-'1', γ, δ start and end with non-'1'.

Finally we have valuation of variables, such that $val(\pi) \leq Z_5$.

$$\begin{array}{cccccccccccc} & \delta & \alpha & \gamma & \beta & \lambda & \gamma & \alpha & \delta & \alpha & \gamma & \beta & \alpha \\ 12131 & \overbrace{214} & \overbrace{1} & \overbrace{213} & \overbrace{12} & \overbrace{151} & \overbrace{213} & \overbrace{1} & \overbrace{214} & \overbrace{1} & \overbrace{213} & \overbrace{12} & \overbrace{1} \end{array}$$

Example

Now we present an example of the *Compressed−Embedding* algorithm for a different pattern. Now the function *val* will be a compact representation instead of the full representation used in the last algorithm. We take:

$$\pi \ = \qquad \alpha\,\gamma\,\beta\,\delta\,\eta\,\alpha\,\gamma\,\beta\,\zeta\,\eta\,\alpha$$

and the ranking sequence

$$1\ 3\ 2\ 4\ 5\ 1\ 3\ 2\ 6\ 5\ 1$$

First we set the variable with the highest rank. $val(\zeta) = 6$

For $i = 5$ we have $\pi_{(5)} = \eta\zeta\eta$
We set $val(\eta) = 5$ and solve 2-SAT for
$F = (\eta^{last} \vee \zeta^{first}) \wedge (\neg\eta^{last} \vee \neg\zeta^{first}) \wedge (\zeta^{last} \vee \eta^{first}) \wedge$
$\wedge(\neg\zeta^{last} \vee \neg\eta^{first}) \wedge (\eta^{first}) \wedge (\eta^{last})$.
From $FirstLast(\pi_{(5)}, \eta)$ we know that ζ_{first} and ζ_{last} are false,
therefore we don't change $val(\zeta)$

$i = 4$. $\pi_{(4)} = \delta\eta\zeta\eta$
We set $val(\delta) = 4$ and execute $FirstLast(\pi_{(4)}, \delta)$. There are two solutions,
we choose one of them and obtain $\eta^{first} = \eta^{last} = 0$ and $\zeta^{first} = \zeta^{last} = 1$,
therefore we change $val(\zeta) = 464$.

$i = 3 : \pi_{(3)} = \gamma\delta\eta\gamma\zeta\eta$.
We set $val(\gamma) = 3$ and execute $FirstLast(\pi_{(3)}, \gamma)$. We know that
$\delta^{first} = \delta^{last} = 0$, $\eta^{first} = 1$, $\eta^{last} = 0$ and $\zeta^{first} = \zeta^{last} = 0$,
therefore we only add 3 at the beginning of $val(\eta)$, i.e. $val(\eta) = 35$

$i = 2 : \pi_{(2)} = \gamma\beta\delta\eta\gamma\beta\zeta\eta$.
We set $val(\beta) = 2$ and execute $FirstLast(\pi_{(2)}, \beta)$. We know that
$\eta^{first} = \eta^{last} = 1$ and rest of logic variables equal 0.
We only add 2 at the beginning and end of $val(\eta)$ ($val(\eta) = 2352$)

$i = 1 : \pi_{(1)} = \pi = \alpha\gamma\beta\delta\eta\alpha\gamma\beta\zeta\eta\alpha$.
We set $val(\alpha) = 1$ and execute $FirstLast(\pi_{(1)}, \alpha)$. We know that
$\gamma^{last} = \delta^{first} = \eta^{first} = \zeta^{first} = 1$ and rest of logic variables equal 0.
We add 1 at the beginning of δ, η, ζ and at the end of γ.
We change: $val(\gamma) = 31, val(\delta) = 14, val(\eta) = 1352, val(\zeta) = 1464$.

Finally we have the compressed valuation of the variables (below we show full representations):

α	β	γ	δ	η	ζ
1	2	3 1	1 4	1 3 5 2	1 4 6 4
1	2	3 1	1 4	13121512	141213121612131214

Our algorithms rely on the following lemma.

Lemma 17. *Let $i \in \{1, \ldots, k\}$ and the pattern $\pi_{(i+1)}$ occurs in Z_{k-i}. Pattern $\pi_{(i)}$ (equal $\pi_{(i+1)}$ with additional variables from $\mathcal{V}_i$) occurs in Z_{k-i+1} iff the corresponding 2-SAT problem has a solution. Moreover every immersion of $\pi_{(i)}$, such that valuations of variables from $\mathcal{V}_i$ equal '1', corresponds to the solution of 2-SAT problem, such that valuation of every variable satisfies logic constraints on first and last character.*

Proof.
" $\Leftarrow$ "
First we observe that solution of 2-SAT problem guarantees that $\mathcal{V}_i$ is a free set. We put $A = \{v \in \pi_{(i)} : v^{last} = 0\}$ and $B = \{v \in \pi_{(i)} : v^{first} = 1\}$, which satisfy Definition 9. From Theorem 14 $\pi_{(i)}$ occurs in Z_{k-i+1}. Now we use Zimin morphism μ on variables from $\pi_{(i+1)}$ and set every variable from $\mathcal{V}_i$ to '1', then we modify (adding or removing '1' at the beginning or end) every valuation accordingly to logic constraints from the 2-SAT solution. Similarly as in the Zimin theorem proof ([5]) we see that properties of A and B guarantee that modified valuations concatenate into proper immersion in Z_{k-i+1}.
" $\Rightarrow$ "
We have immersion of $\pi_{(i)}$ in Z_{k-i+1} with valuation $val(v)$, such that for $v \in \mathcal{V}_i$ $val(v) = 1$. We give a solution to 2-SAT problem as follows: for every variable v from $\pi_{(i)}$ we set $v^{first} = 1$ iff $val(v)$ starts with '1', $v^{last} = 1$ iff $val(v)$ ends with '1'. Because Zimin word is 1-interleaved this solution is correct. □

Theorem 18. *The compressed ranked pattern matching in Zimin words can be solved in time $O(nk)$ and (simultaneously) space $O(n+k^2)$, where n is the size of the pattern and k is the highest rank of a variable. A compressed instance of the pattern can be constructed within the same complexities, if there is any solution.*

Proof. We use the $Compressed - Embedding$ algorithm. First we embed $\pi_{(k)}$ into Z_1. Then we check for a subsequent i, $k - 1 \geq i \geq 1$, if it is possible to embed $\pi_{(i)}$ having embedding of $\pi_{(i+1)}$ using suitable 2-SAT and Lemma 17. Non-existence of immersion of $\pi_{(i)}$ implies that whole π does not occur into Z_k. Otherwise we get compressed valuation of variables, such that $val(\pi) \leq Z_k$.

We will consider time complexity of the Compressed-Embedding algorithm. Because $|V_1| \cup |V_2| \cup \cdots \cup |V_k| = k$ first **for each** loop executes exactly k times during whole execution.

We solve 2-SAT problem exactly $k - 1$ times for $\pi_{(i)}$ (length of formula is linear with respect to $|\pi_{(i)}|$) and for every i $|\pi_{(i)}| \leq |\pi| = n$. That gives complexity $O(nk)$ for this step.

We execute second **for each** loop $k - 1$ times. In each iteration we have:

$$|\mathcal{V}_{i+1} \cup \mathcal{V}_{i+2} \cup \cdots \cup \mathcal{V}_K| \leq |\mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_K| = k$$

Hence complexity for this step is $O(k^2)$.
Finally, the algorithm has time complexity $O(k + nk + k^2) = O(nk)$, because $k \leq n$.

We have some choice when applying 2-SAT, since many satisfying valuation are sometimes possible. By slightly modifying the application of 2-SAT we can obtain the following result.

Theorem 19. *For a given ranked pattern the compressed shortest instance and lexicographically smallest instance of the ranked pattern occurring in Z_k can be constructed in time $O(nk)$ and (simultaneously) space $O(n+k^2)$ (if there is any instance).*

References

1. D. R. Bean, A. Ehrenfeucht, and G. F. McNulty: *Avoidable patterns in strings of symbols*, Pacific J. Math, 1979.
2. C. E. Heitsch: *Generalized Pattern Matching and the Complexity of Unavoidability Testing*, Lecture Notes in Computer Science, 2001.
3. M. Lothaire: *Algebraic Combinatorics on Words*, Cambridge University Press, 2002.
4. A. Thue: *Über unendliche Zeichenreihen*, Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7, 1906.
5. A. I. Zimin: *Blocking sets of terms*, Mat. Sb. (N.S.) 119(161), 1982.

Improving Deduplication Techniques by Accelerating Remainder Calculations

Michael Hirsch[1], Shmuel T. Klein[2], and Yair Toaff[1]

[1] IBM – Diligent
Tel Aviv
{hirschm,yairtoaff}@il.ibm.com

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

Abstract. The time efficiency of many storage systems rely critically on the ability to perform a large number of evaluations of certain hashing functions fast enough. The remainder function $B \bmod P$, generally applied with a large prime number P, is often used as a building block of such hashing functions, which leads to the need of accelerating remainder evaluations, possibly using parallel processors. We suggest several improvements exploiting the mathematical properties of the remainder function, leading to iterative or hierarchical evaluations. Experimental results show a 2 to 5-fold increase in the processing speed.

1 Introduction

The probabilistic pattern matching algorithm due to Karp and Rabin [3] is based on the repeated evaluation of a so-called *rolling hash*: given is a text of length n and a pattern of length m, a hash function has to be applied on all the substrings of the text of length m. A naive implementation would thus yield a $\theta(nm)$ time complexity, which might be prohibitive. The rolling property of the hash exploits the fact that adjacent substrings are overlapping in all but their first and last characters, so that the hash of one substring can be calculated in constant time from the hash value of the preceding one, reducing the complexity to $O(n)$.

There are, however, important applications of the Karp-Rabin scheme, beyond pattern matching. Large storage and backup systems can be compressed by means of *deduplication*: locating recurrent sub-parts of the text, and replacing them by pointers to previous occurrences. One family of deduplication algorithms is known in the storage industry as CAS (Content Addressed Storage) and based on assigning a hash value to each data block [5,6]. Such systems detect only identical blocks and are not suitable when large block sizes are used. Replacing identity by similarity enables the use of much larger data chunks, as in the IBM ProtecTIER$^{(R)}$ product [1]. This system is based on the evaluation of a hash function for a large number of strings, and most of these evaluations can be done in constant time because of overlaps, as mentioned above.

In a typical setting, a very large repository, say, of the order of 1 PB $= 2^{50}$ bytes, will be partitioned into chunks of fixed or variable size, to each of which one or more *signatures* are assigned. The details of the deduplication algorithm are not relevant to our current discussion and the interested reader is referred to [1]. The signature of a chunk is usually some function of the set of hash values produced for each consecutive substring of k bytes within the chunk. The length k of these substrings, which we call

Michael Hirsch, Shmuel T. Klein, Yair Toaff: Improving Deduplication Techniques by Accelerating Remainder Calculations, pp. 173–183.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

seeds, may be 512 or more, so that the evaluation might put a serious burden on the processing time.

Given a chunk $C = x_1x_2 \cdots x_n$, where the x_i denote characters of an alphabet Σ, we wish to apply the hash function h on the set of substrings B_i of C of length k, $B_i = x_ix_{i+1} \cdots x_{i+k-1}$ being the substring starting at the i-th character of C. The constant time, however, for the evaluation of B_i is based on the fact that one may use the value obtained earlier for B_{i-1}, and this is obviously not true for the first value to be used. That is, B_1 needs an evaluation time proportional to k. Moreover, in deduplication systems based on similarity rather than on identity, once a chunk of the *reference* has been identified as being similar to a chunk of the *version*, a more fine-grained comparison of the two is needed.

Figure 1 is a schematic representation of the following typical scenario: given are two chunks which are already known to be similar, we need to identify as many of their matching parts as possible. To this end, the reference is partitioned into a sequence of non-overlapping seeds, and a hash value of each of these seeds is evaluated and stored in a table H_R. As to to version, the hash value of every seed at every possible byte offset is calculated and potential matches are located in H_R. If a match is found, say, $H_V[i] = H_R[j]$, it is almost certain that the string $v_iv_{i+1} \cdots v_{i+k-1}$ is identical to $r_{(j-1)k+1}r_{(j-1)k+2} \cdots r_{jk}$, so the strings can be accessed and we shall try to extend the match to the left and right of these seeds.

Since the rolling hash property does not apply to the seed-by-seed evaluations of the reference, each substring of size k requires a $O(k)$ processing time. The techniques in this paper are aimed at speeding up the initialization and non-overlapping hashing operations using local parallelism, by means of the availability of several processors.

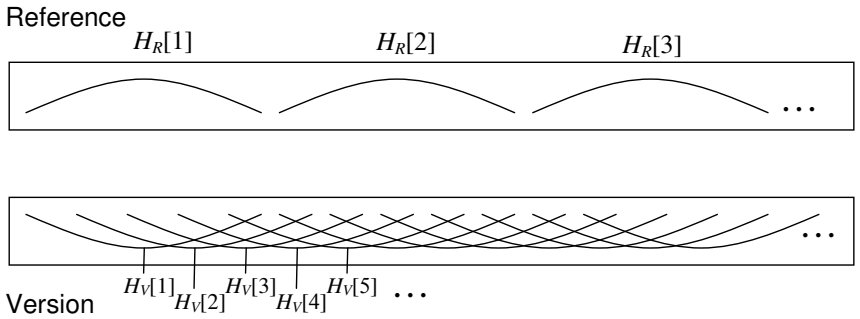


Figure 1. Searching for matching parts in similar chunks

The hash function we consider in this work is the remainder function modulo a prime number P, $h(B) = B \bmod P$, which is well known for yielding close to uniform distributions on many real-life input distributions. We interchangeably use B to denote a character string and the integer value represented by the binary string obtained by concatenating the ASCII codewords of the characters forming B. For example, the string `ABC` would be in ASCII 010000010100001001000011, so we would identify the string with the value 4,276,803. Two main improvements to the standard computation of the modulus are suggested: the first constructs a hierarchical structure enabling the use of several processors in parallel; the second exploits the fact that the computation can be performed iteratively to speed it up by calculating what we shall call *pseudo-hashes*.

2 Hierarchical evaluation of the remainder function

Consider the input string B partitioned into m subblocks of d bits each, denoted $A[0], \ldots, A[m-1]$, where $m = 2^r$ is a power of 2, and d is a small integer, so that d bits can be processed as an indivisible unit, typically $d = 32$ or 64. Given also is a large constant number P of length up to d bits, that will serve as modulus. Typically, but not necessarily, P will be a prime number, and for our application it is convenient to choose P close to a power of 2. For example, one could use $m = 64$, $d = 64$ and $P = 2^{55} - 55$. We would like to split the evaluation of $B \bmod P$ so as to make use of the possibility to evaluate functions of the $A[i]$ in parallel on m independent processors $p_0, p_1, \ldots, p_{m-1}$, which should yield a speedup. We have

$$B \bmod P = \left(\sum_{i=0}^{m-1} A[i] \times 2^{d(m-1-i)} \right) \bmod P$$

Considering it as a polynomial (set $x = 2^d$, then $B = \sum_{j=0}^{m-1} A[m-1-j]x^j$), we can use Horner's rule to evaluate it iteratively. We first need the constant C defined by

$$C = 2^d \bmod P. \tag{1}$$

Note then that if we have a string D of $2d$ bits and we want to evaluate $\overline{D} = D \bmod P$, then we can write $D = D_1 \times 2^d + D_2$, where D_1 and D_2 are the leftmost, respectively rightmost d bits of D. We get that

$$\overline{D} = \overline{\overline{D_1} \times \overline{2^d} + \overline{D_2}} = \overline{\overline{D_1} \times C + D_2}. \tag{2}$$

Generalizing to m blocks of d bits each, we get the iterative procedure of Figure 2.

Iterative evaluation of $B \bmod P$
$R \longleftarrow 0$
for $i \longleftarrow 0$ to $m-1$ do
    $R \longleftarrow (R \times C + A[i]) \bmod P$ (3)

Figure 2. Iterative evaluation of $B \bmod P$

A further improvement can then be obtained by passing to a hierarchical tree structure and exploiting the parallelism repeatedly in $\log m$ layers, using the m available processors. In Step 0, the m processors are used to evaluate $A[i] \bmod P$, for $0 \le i < m$, in parallel. This results in m residues, which can be stored in the original place of the m blocks $A[i]$ themselves, since P is assumed to fit into d bits. For our example values of m, d and P, only 55 of the 64 bits would be used.

In Step 1, only $\frac{m}{2}$ processors are used (it will be convenient to use those with even indices), and each of them works, in parallel, on two adjacent blocks: p_0 working on $A[0]$ and $A[1]$, p_2 working on $A[2]$ and $A[3]$, and generally p_{2k} working on $A[2k]$ and $A[2k+1]$, for $k = 0, 1, \ldots, \frac{m}{2} - 1$. The work to be performed by each of these processors is what has been described earlier for the block D. Again, the results will be stored in-place, that is, right-justified in $2d$-bit blocks, of which only the rightmost d bits (or less, depending on P), will be affected.

Hierarchical evaluation of $B \bmod P$

for $k \longleftarrow 0$ to $m-1$ do
    $A[k] \longleftarrow A[k] \bmod P$
for $i \longleftarrow 1$ to r do
    for $k \longleftarrow 0$ to $\frac{m}{2^i} - 1$ do
        use processor $p_{2^i k}$ to evaluate, in parallel,
            $A[2^i k + 2^i - 1] \longleftarrow (A[2^i k + 2^{i-1} - 1] \times C[i] + A[2^i k + 2^i - 1]) \bmod P$

Figure 3. Hierarchical parallel evaluation of $B \bmod P$

In Step 2, the $\frac{m}{4}$ processors whose indices are multiples of 4 are used, and each of them is applied, in parallel, on two adjacent blocks of the previous stage. That is, we should have applied now p_0 on $A[0]A[1]$ and $A[2]A[3]$, etc., but in fact we know that $A[0]$ and $A[2]$ contain only zeros, so we can simplify and apply p_0 on $A[1]$ and $A[3]$, and in parallel p_4 on $A[5]$ and $A[7]$, and generally, p_{4k} working on $A[4k+1]$ and $A[4k+3]$, for $k = 0, 1, \ldots, \frac{m}{4} - 1$. Again, the work to be performed by each of these processors is what has been described earlier for the block D since we are combining two blocks, with the difference that the new constant C should now be $2^{2d} \bmod P = \overline{C^2}$. The results will be stored right-justified in $4d$-bit blocks, of which, as before, only the rightmost d bits or less will be affected.

Continuing with further steps will yield a single operation after $\log m$ iterations. Note that the overall work is not reduced by this hierarchical approach, since the total number of applications of the procedure on block pairs is $\frac{m}{2} + \frac{m}{2} + \cdots = m - 1$, just as for the sequential evaluation. However, if we account only once for operations that are executed in parallel, the number of evaluations is reduced to $\log m$, which should result is a significant speedup.

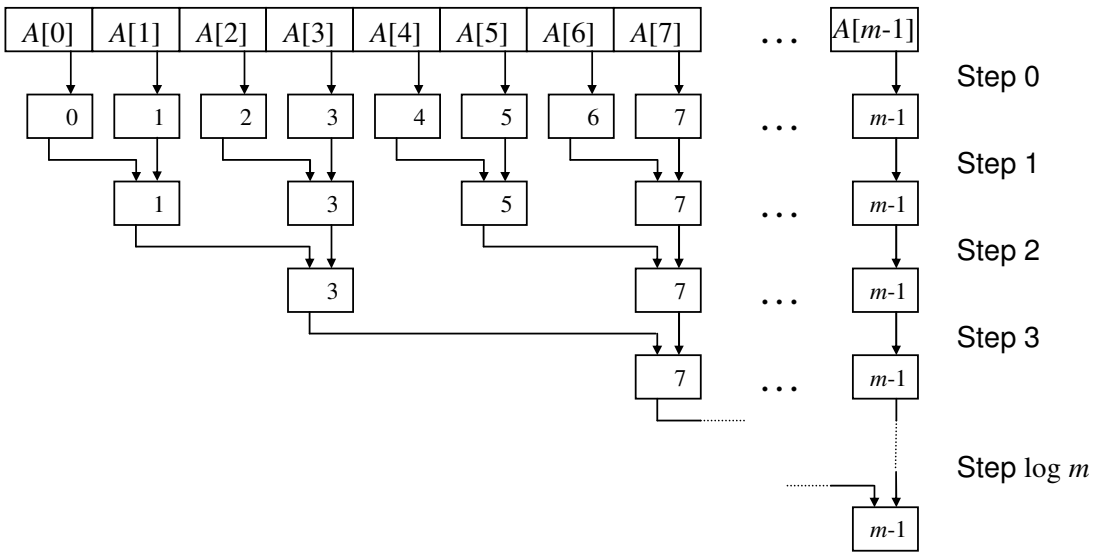


Figure 4. Schematic representation of the hierarchical evaluation

Summarizing, we first evaluate an array of constants

$$C[i] = \overline{C^{2^{i-1}}} = \overline{2^{d \times 2^{i-1}}}$$

to be used in step i for $i = 1, 2, \ldots, m-1$. This is easily done noticing that $C[1] = C$ and $C[i+1] = \overline{C[i]^2}$ for $i \geq 1$. The parallel procedure is then given in Figure 3, and a schematic view of the evaluation layers can be found in Figure 4.

3 Avoiding overflows

The algorithm as described above dealt with integers of d bits length. We shall, for the ease of description, use the values $d = 64$ and $P = 2^{55} - 55$ in the sequel, which correspond to real-life applications, but all the ideas can easily be generalized to any other appropriate values. When two 64 bit integers are multiplied as $R \times C$ in equation (3), even though the result is sought modulo P, which is a 55-bit integer, one temporarily needs 128-bit arithmetic, which yields a serious slow down of the performance.

One might think that to circumvent this, it suffices to work with smaller blocks, say, of $d = 32$ bits only. This will double the number of iterations, but could still result in a gain, if during multiplications the 64 bit limit is never exceeded. For the parallel implementation, the logarithmic number of parallel steps would only increase by 1. However, reducing d does not yet solve the problem, because R is a 55-bit integer, so when multiplied by the updated constant $C = 2^{32} \bmod P = 2^{32}$, we can get up to 87 bits. In order to get all the integers in this evaluation to be of length at most 64 bits (the maximum is reached when multiplying $R \times C$), so that no special 128-bit arithmetic would be needed, R has to be split and the modulus has to be applied not only at the end of each iteration.

Note that while we now assume that $d = 32$, the values of R are still stored in 64 bit integers. The way of splitting the 8 bytes representing R will be into the 23 rightmost bits and the complementing 41 leftmost bits. In fact, since the involved numbers are residues of $\bmod P$, where P is a 55 bit prime, the number of least significant non-zero bits in the left part is only $55 - 23 = 32$. The representation of R is therefore

$$R = R_L \times 2^{23} + R_R,$$

where R_L are the 41 (in fact, only 32) leftmost and R_R are the 23 rightmost bits of R, so

$$R \times C = R \times 2^{32} = R_L \times 2^{55} + R_R \times 2^{32},$$

and since $2^{55} \bmod P = 2^{55} \bmod (2^{55} - 55) = 55$, we get that

$$\overline{R \times C + A[i]} = \overline{R_L \times 55 + R_R \times 2^{32} + A[i]}.$$

The revised evaluation is given in Figure 5. Note that the mod P operation within the loop has been removed, and replaced by two mod operations following the loop. We thus call the intermediate values *pseudo-remainders*. The correctness of the procedure is based on the following

Theorem *The value of R is smaller than 2^{56}, that is, fits into 56 bits, at the end of each iteration.*

Revised iterative evaluation

$R \longleftarrow 0$
for $i \longleftarrow 0$ to $m-1$ do
    $R_L \longleftarrow R \,/\, 2^{23}$
    $R_R \longleftarrow R \bmod 2^{23}$
    $R \longleftarrow R_L \times 55 + R_R \times 2^{32} + A[i]$
end-for
if $R > P$ then
    $R \longleftarrow R - P$
if $R > P$ then
    $R \longleftarrow R - P$

Figure 5. Iterative evaluation without mod

Proof By induction on i, the index of iteration. For $i = 0$, at the beginning of the iteration, R and thus also R_L and R_R are 0. The value of R at the end of iteration 0 is therefore $A[0]$, which has only 32 bits, less than 56.

Suppose the assumption is true at the end of iteration i, and consider the beginning of iteration $i+1$. R_R has at most 23 bits by definition, and R_L has at most $56-23 = 33$ bits by the inductive assumption. Hence $R_R \times 2^{32}$ is of length at most 55 bits, and so is $R_R \times 2^{32} + A[i]$, since the 32 rightmost bits of $R_R \times 2^{32}$ are zero. The binary representation of 55 uses 6 bits, so $R_L \times 55$ is of length at most $33 + 6 = 39$ bits. At the end of the iteration, the length of R, obtained by adding a 39 bit number to a 55 bit number, must therefore be at most 56, and this limit is achieved only if a carry propagates beyond the leftmost bit of $R_R \times 2^{32}$. □

It follows from the Theorem that there is no overflow if we remove the repeated application of the modulo operator, and only perform a single (and rarely, two) modulus at the end of the iteration. This is the purpose of the last four lines. Since at the end, $R < 2^{56} = 2P + 110$, the modulus can be replaced by subtraction. If $P \leq R < 2P$, then $R \bmod P = R - P$. For the rare cases in which $2P \leq R < 2P + 110$ (only 110 out of the possible almost 2^{56} values of R), a second subtraction of P will be necessary.

To understand how all the mod operations within the iteration could be saved, recall that our objective was to calculate $B \bmod P$. It would thus suffice, mathematically speaking, to apply a single mod operation after having calculated B, but in practice, such an evaluation is not feasible, because we are dealing here with a $m \times d$ bit long number, which cannot be handled. The classical solution, generally used in modular exponentiation algorithms [2], is to exploit the properties of the modulo function, to repeatedly apply the modulus to subparts of the formula, so as to never let the operands on which the modulus has to be applied grow above the limit permitted by the hardware at hand. For example, representing B as a polynomial $B = \sum_{j=1}^{m} A[m-j]x^{j-1}$, where we have set $x = 2^{32}$, using Horner's rule, we get

$$\overline{B} = \overline{\left(\cdots\left(\overline{\left(\overline{A[0]x + A[1]}\right)x + A[2]}\right)x + \cdots\right)x + A[m-1]},$$

where after each multiplication and addition, $\bmod P$ is applied, so if we start with d bit numbers, at no stage of the evaluation do we use numbers larger than $2d$ bits.

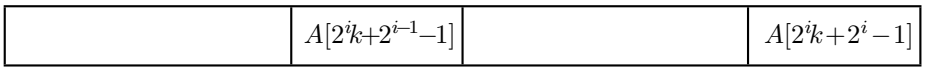
This was the approach in Section 2, and had as drawback that such a large number of modulo applications is expensive. The current suggestion reverts the process and removes again the internal modulo applications, but not entirely, since this would get us back to handling $m \times d$ bit numbers. Rather, it removes only a part of the internal operations, but leaves the cheap ones, basing ourselves on the fact that we work modulo a prime which is very close to a power of 2, namely $P = 2^{55} - 55$ in our example, but one can find such primes for any given exponent, see [4]. We thus get that $2^{55} \bmod P = 55$ in our case, an extremely small number relative to P, which can be used to decompose blocks into adjacent subblocks at a low price.

The algorithm presents a tradeoff between applying the remainder function only once (cheap but unfeasible because of the size of the numbers involved), and applying it repeatedly in every iteration (resulting in small numbers, but computationally expensive). We apply it only once (rarely twice) at the end, but managed by an appropriate decomposition of the numbers to remove the moduli and still force all the involved numbers to be small.

Note that this technique can not be applied generally in situations where the modulus is chosen as a large random prime number, as often done in cryptographic applications, since it critically depends on the fact that $2^{55} \bmod P$ is a small number. In our case, it uses only 6 bits, and the Theorem would still hold for values needing up to 22 bits, in which case $R_L \times (2^{55} \bmod P)$ is of length at most $33 + 22 = 55$ bits. The sum of two 55 bit numbers would then still fit into the 56 bits claimed in the induction. But for 23 bits, we could already overflow into 57 bits. If P is a random prime number of 55 bits, the expected length of $2^{55} \bmod P$ is 54 bits and will only extremely rarely fit into 22 bits. The application field of the technique is thus when repeated evaluations are needed, all modulo a *constant* P, which can therefore be chosen as some convenient prime just a bit smaller than a given power of two. This is the case in rolling hashes of the Rabin-Karp type we consider here.

4 Adapting the hierarchical method

We now turn to adapting the hierarchical method, which can be used in parallel with m processors, to 64-bit arithmetic to improve processing time. The input is a sequence of $n = 2^m$ blocks of $d = 64$ bits each. The hierarchical evaluation is done in $m = \log n$ layers, with layer i processing what we shall call *superblocks*, consisting of 2^i original d-bit blocks, $i = 0, 1, \ldots, m-1$. The scenario at layer i, for the superblock indexed k, $k = 0, 1, \ldots, \frac{n}{2^i} - 1$, is:



The superblock consists of two halves, and only the rightmost block (in fact, only its 55 rightmost bits) in each half is non-zero. The evaluation combines the two non-zero values and puts the output in the rightmost block, using the command

$$A[2^i k + 2^i - 1] \longleftarrow \Big(A[2^i k + 2^{i-1} - 1] \times C[i] + A[2^i k + 2^i - 1]\Big) \bmod P.$$

The values $C[i] = \overline{C^{2^{i-1}}} = \overline{2^{64 \times 2^{i-1}}}$ can be calculated as $C[1] = 2^{64} \bmod P$ and $C[i+1] = \overline{C[i]^2}$ for $i > 1$. For $P = 2^{55} - 55$, these values are given in Table 1.

i	$C[i]$	bits
1	28,160	15
2	792,985,600	30
3	16,336,612,484,973,479	55
4	8,143,640,278,601,598	55
5	5,745,742,201,926,802	55
6	16,594,324,020,821,548	55

Table 1. Constants for hierarchical evaluation

We thus need more than 64 bits to evaluate $A[2^i k + 2^{i-1} - 1] \times C[i]$ for $i > 1$. To fit into the 64-bit arithmetic constraint, we propose two strategies. The first is a generic one, that can be applied to any values of the parameters, and processes each layer in the same way. The second achieves some additional savings by adapting the specific values in our running example differently in each of the layers.

4.1 General uniform adaptation of the parameter values

The first iteration (layer 0), which applies the modulus on the original 64 bit blocks to produce 55 bit numbers, can be kept without change. For the higher layers, the input of which are two non-adjacent 55-bit blocks $A[2^i k + 2^{i-1} - 1]$ and $A[2^i k + 2^i - 1]$, the latter can be used as is, but the former has to be multiplied, so we split the block into 11 subblocks of length 5 bits.

Denote the 11 blocks forming $A[2^i k + 2^{i-1} - 1]$, from right to left, by $E[k, i, j]$, $j = 0, 1, \ldots, 10$, which gives

$$A[2^i k + 2^{i-1} - 1] = \sum_{j=0}^{10} E[k, i, j] \times 2^{5j}.$$

In addition, prepare a two dimensional table $CC[i, j]$ for the above values of i and j, defined by

$$CC[i, j] = \overline{C[i] \times 2^{5j}}.$$

Then

$$A[2^i k + 2^{i-1} - 1] \times C[i] + A[2^i k + 2^i - 1] = \sum_{j=0}^{10} E[k, i, j] \times CC[i, j] + A[2^i k + 2^i - 1].$$

Each term in the summation uses at most $5 + 55 = 60$ bits, so the sum of the 12 terms uses at most $60 + \lceil \log 12 \rceil = 64$ bits, as requested. In fact, since the elements $E[k, i, j]$ all belong to a small set $\{0, 1, \ldots, 31\}$, one can precompute a three dimensional table $CCC[i, j, p]$ defined, for $p = 0, \ldots, 31$ by

$$CCC[i, j, p] = \overline{CC[i, j] \times p} = \overline{C[i] \times 2^{5j} \times p}.$$

This reduces then the right hand side of the summation above to

$$\sum_{j=0}^{10} CCC\,[i, j, E[k, i, j]] + A[2^i k + 2^i - 1].$$

To take this idea of tabulating even a step further, note that the elements in the table are computed only once, so this could be done offline, and there, 128-bit operations could be permitted. Instead of partitioning $A[2^i k + 2^{i-1} - 1]$ into 11 blocks of 5 bits each, any other partition into $\lceil 55/q \rceil$ blocks of q bits each could be considered, if we were willing to extend the table $CCC[i, j, p]$ to the 2^q possible values of q-bit strings. Taking, for example, $q = 11$, we get 5 blocks of 11 bits and would have to consider 2048 possible values of p in $CCC[i, j, p]$. The number of bits needed to represent $CC[i, j] \times p$ would then be $55+11 = 66$, but this is evaluated only once, and what will finally be stored (and used afterwards) is $\overline{CC[i, j] \times p}$, which again needs only 55 bits; the sum of six 55-bit numbers fits into 58 bits, so there is no overflow.

The number of elements needed in the table CCC is $m \times \left\lceil \frac{55}{q} \right\rceil \times 2^q$. Table 2 brings the size of the table for a few sample values of q, for $m = 6$ as in our example. The number of 64-bit operations for the evaluation of each new value is equal to the number of blocks b: there are $b + 1$ terms to be added, but only $x - 1$ additions are needed to add x terms.

q	# blocks	# lines	# entries	Actual size
3	19	8	912	6.2 K
4	14	16	1344	9.1 K
5	11	32	2112	14.4 K
6	10	64	3840	26.3 K
7	8	128	6144	42 K
8	7	256	10752	74 K
9	7	512	21504	147 K
10	6	1024	36864	252 K
11	5	2048	61440	420 K
12	5	4096	122880	840 K
16	4	65536	1572864	10.5 M
20	3	1048576	18874368	126 M

Table 2. Size of auxiliary table for various values of q

We can thus choose the value of q according to the required tradeoff: the lower q, the less storage is needed for the CCC tables, but the more operations have to be performed. Taking for example values of q from 5 to 7, the tables would fit into 50K, but 9 to 12 operations have to be performed.

4.2 Specific adaptation of the parameter values for $m = 6$ and $d = 64$

The tradeoffs in Table 2 lead to the following suggestions for the lower layers. Consider layer 1, consisting of superblocks of 128 bits. Figure 6 represents the layout after iteration 0, in which two 55-bit strings have been evaluated (in grey in the figure). We partition the superblock as indicated, which yields as value:

$$D = D_{11} \times 2^{110} + D_{12} \times 2^{55} + D_{13}.$$

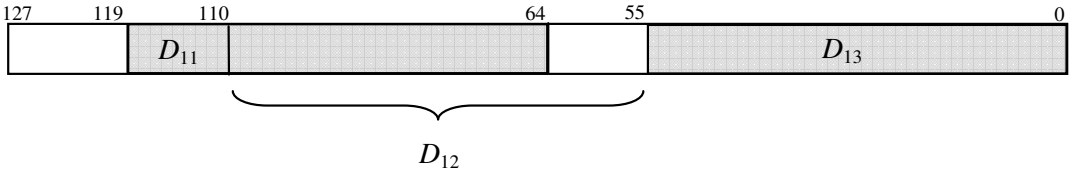


Figure 6. Layer 1: two blocks of 64 bits each

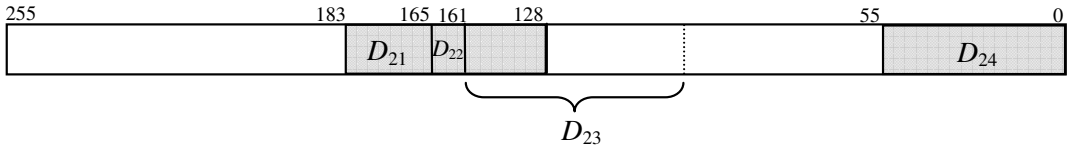


Figure 7. Layer 2: two blocks of 128 bits each

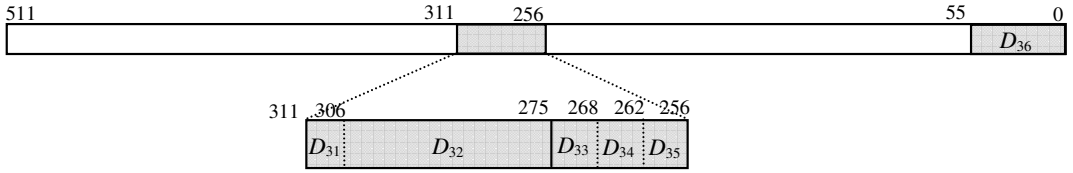


Figure 8. Layer 3: two blocks of 256 bits each

D_{13} uses only 55 bits; D_{12} also needs 55 bits, but is multiplied by $2^{55} \bmod P = 55$, which needs 6 bits, so together 61 bits; D_{11} needs 9 bits, and multiplied by $2^{110} \bmod P = 55^2 = 3025$, which needs 12 bits, so together 21 bits; their sum has therefore at most 62 bits, so only **two** 64-bit additions are needed.

For layer 2, we need a different layout, given in Figure 7. The superblock consists now of two subparts of 128 bits each. This partition yields the following equality:

$$D = D_{21} \times 2^{165} + D_{22} \times 2^{161} + D_{23} \times 2^{110} + D_{24}.$$

D_{24} uses only 55 bits; D_{23} is of length 51 bits, but is multiplied by $2^{110} \bmod P = 3025$, which needs 12 bits, so together 63 bits; D_{22} is of length 4 bits, and is multiplied by $2^{161} \bmod P$, which needs 55 bits, so together 59 bits; finally, D_{21} needs 18 bits, and is multiplied by $2^{165} \bmod P = 55^3 = 166375$, which needs 18 bits, so together 36 bits; their sum has therefore at most 64 bits, so only **three** 64-bit additions are needed.

Layer 3 will be the last with special treatment. A superblock, now consisting of two halves of 256 bits each, will be partitioned according to the layout given in Figure 8. The desired value of D is then obtained by adding the following terms:

$$\begin{aligned}
&D_{31} \times 2^{306}, \text{ in bits: } 5 + 55 = 60\\
&D_{32} \times 2^{275}, \text{ in bits: } 31 + 29 = 60\\
&D_{33} \times 2^{268}, \text{ in bits: } 7 + 55 = 62\\
&D_{34} \times 2^{262}, \text{ in bits: } 6 + 55 = 61\\
&D_{35} \times 2^{256}, \text{ in bits: } 6 + 55 = 61\\
&D_{36}, \qquad \text{in bits: } 55
\end{aligned}$$

Their sum has at most 64 bits, and only **five** 64-bit additions are needed.

It seems fair to consider the *amortized* global cost for evaluating the signature, since only at the lowest level, all the n processors are involved, and for the higher levels, specifically, for level i, the number of working processors is only $n/2^i$. The amortized number of 64-bit additions is therefore

$$1\times n+2\times\frac{n}{2}+3\times\frac{n}{4}+5\times\frac{n}{8}+11\times\left[\frac{n}{16}+\frac{n}{32}+\cdots\right]=n\times\left[1+1+\frac{3}{4}+\frac{5}{8}+\frac{11}{8}\right]=4.75n.$$

5 Experimental results

We have compared the above methods on randomly chosen input texts, several GB of our exchange database. Actually, the exact choice of the test data is not relevant, because the number of remainder operations performed is not data dependent.

	WS	**M2**	**X5**	**GPU**
baseline	114	139	168	595
hierarchical	229	200	377	1896
pseudo remainders	582	256	1067	2327

Table 3. Experimental comparison of performance

The following methods were tested: as baseline, we took a regular iterative evaluation, processing single bytes, that is, $d = 8$. In all our tests, the size of B was $m = 2^{12} = 4096$ bits or 512 bytes. The next method was a hierarchical implementation, according to Figure 3, with blocks of size $d = 64$, and using 128-bit arithmetic where necessary. Finally, we also ran the revised iterative method of Figure 5 using pseudo remainders, with $d = 32$ and 64-bit operations only.

The tests were run on several platforms: **WS**: a 3.2 GHz Intel PC Workstation, **M2**: an IBM 3850M2 server (2.93 GHz Intel Xeon X7350), **X5**: an IBM 3850X5 server (2.27 GHz Intel Xeon X7560), and **GPU**: an Nvidia GeForce GTX 465 graphics board, using copy to/from device. The results are presented in Table 3, all values giving the number of MB processed per second.

References

1. L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein: *The design of a similarity based deduplication system.* Proc. of the SYSTOR'09 Conference, 2009, pp. 1–14.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest: *Introduction to Algorithms*, MIT Press, 1990.
3. R. M. Karp and M. O. Rabin: *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, 1987, pp. 249–260.
4. *Primes just less than a power of two*: http://primes.utm.edu/lists/2small/.
5. S. Quinlan and S. Dorward: *Venti: A new approach to archival storage.* Proc. FAST'02, the 1st USENIX Conference on File And Storage Technologies, 2002, pp. 89–101.
6. B. Zhu, K. Li, and H. Patterson: *Avoiding the disk bottleneck in the data domain deduplication file system.* Proc. FAST'08, the 6th USENIX Conference on File And Storage Technologies, 2008, pp. 279–292.

Computing Abelian Periods in Words

Gabriele Fici[1], Thierry Lecroq[2], Arnaud Lefebvre[2], and Élise Prieur-Gaston[2]

[1] I3S, CNRS & Université de Nice-Sophia Antipolis, France
Gabriele.Fici@unice.fr
[2] Université de Rouen, LITIS EA4108, 76821 Mont-Saint-Aignan Cedex, France
{Thierry.Lecroq,Arnaud.Lefebvre,Elise.Prieur}@univ-rouen.fr

Abstract. In the last couple of years many works have been devoted to Abelian complexity of words. Recently, Constantinescu and Ilie (Bulletin EATCS 89, 167–170, 2006) introduced the notion of *Abelian period*. We show that a word w of length n over an alphabet of size σ can have $\Theta(n^2)$ distinct Abelian periods. However, to the best of our knowledge, no efficient algorithm is known for computing these periods. The Brute-Force algorithm computes all the Abelian periods either in time $O(n^3 \times \sigma)$ using $O(\sigma)$ space or in time $O(n^2 \times \sigma)$ using $O(n \times \sigma)$ space. We present an off-line algorithm running in time $O(n^2 \times \sigma)$ using $O(n+\sigma)$ space, thus improving the space complexity. This algorithm is based on a *select* function. We then present on-line algorithms that also enable to compute all the Abelian periods of all the prefixes of w. Experimental results show that the new off-line algorithm is faster than the Brute-Force one. Moreover, in most cases, one on-line algorithm, though having a worst case time complexity, is also faster than the Brute-Force one.

1 Introduction

An integer $p > 0$ is a (classical) period of a word w of length n if $w[i] = w[i + p]$ for any $1 \leqslant i \leqslant n-p$. Classical periods have been extensively studied in combinatorics on words [13] due to their direct applications in data compression and pattern matching.

The Parikh vector of a word w enumerates the cardinality of each letter of the alphabet in w. For example, given the alphabet $\Sigma = \{a, b, c\}$, the Parikh vector of the word $w = \texttt{aaba}$ is $(3, 1, 0)$. The reader can refer to [6] for a list of applications of Parikh vectors.

An integer p is an Abelian period of a word w if w can be written as $u_0u_1 \cdots u_{k-1}u_k$ where all the u_i's are of length p and have the same Parikh vector $\mathcal{P}$ for $0 < i < k$ and the Parikh vectors of u_0 and u_k are contained in $\mathcal{P}$ [9]. This definition matches the one of *weak repetition* (also called *Abelian power*) when u_0 and u_k are the empty word and $k > 2$ [10].

In the last couple of years many works have been devoted to Abelian complexity [11,2,7,4,12,1,5,17]. Efficient algorithms for Abelian pattern matching have been designed [8,6,14,15]. However, apart of the greedy off-line algorithm given in [10], neither efficient nor on-line algorithms are known for computing all the Abelian periods of a given word.

In this article we present several efficient off-line and on-line algorithms for computing all the Abelian periods of a given word. In Section 2 we give some basic definitions and notation. Section 3 presents off-line algorithms while Section 4 presents on-line algorithms. In Section 5 we give some experimental results on execution times. Finally, Section 6 contains conclusions and perspectives.

Gabriele Fici, Thierry Lecroq, Arnaud Lefebvre, Élise Prieur-Gaston: Computing Abelian Periods in Words, pp. 184–196.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

2 Definitions and notation

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet of cardinality σ and Σ^* the set of words on alphabet Σ. We set $ind(a_i) = i$ for $1 \leqslant i \leqslant \sigma$. We denote by $|w|$ the length of w. We write $w[i]$ the i-th symbol of w and $w[i..j]$ the factor of w from the i-th symbol to the j-th symbol, with $1 \leqslant i \leqslant j \leqslant |w|$. We denote by $|w|_a$ the number of occurrences of the symbol $a \in \Sigma$ in the word w.

The *Parikh vector* of a word w, denoted by $\mathcal{P}_w$, counts the occurrences of each letter of Σ in w, that is $\mathcal{P}_w = (|w|_{a_1}, \ldots, |w|_{a_\sigma})$. Notice that two words have the same Parikh vector if and only if one word is a permutation of the other.

We denote by $\mathcal{P}_w(i, m)$ the Parikh vector of the factor of length m beginning at position i in the word w.

Given the Parikh vector $\mathcal{P}_w$ of a word w, we denote by $\mathcal{P}_w[i]$ its i-th component and by $|\mathcal{P}_w|$ the sum of its components. Thus for $w \in \Sigma^*$ and $1 \leqslant i \leqslant \sigma$, we have $\mathcal{P}_w[i] = |w|_{a_i}$ and $|\mathcal{P}_w| = \sum_{i=1}^{\sigma} \mathcal{P}_w[i] = |w|$.

Finally, given two Parikh vectors $\mathcal{P}, \mathcal{Q}$, we write $\mathcal{P} \subset \mathcal{Q}$ if $\mathcal{P}[i] \leqslant \mathcal{Q}[i]$ for every $1 \leqslant i \leqslant \sigma$ and $|\mathcal{P}| < |\mathcal{Q}|$.

Definition 1 ([9]). *A word w has an Abelian period (h, p) if $w = u_0 u_1 \cdots u_{k-1} u_k$ such that:*

- $\mathcal{P}_{u_0} \subset \mathcal{P}_{u_1} = \cdots = \mathcal{P}_{u_{k-1}} \supset \mathcal{P}_{u_k}$,
- $|\mathcal{P}_{u_0}| = h$, $|\mathcal{P}_{u_1}| = p$.

We call u_0 and u_k resp. the *head* and the *tail* of the Abelian period. Notice that the length $t = |u_k|$ of the tail is uniquely determined by h, p and $|w|$, namely $t = (|w| - h) \bmod p$.

The following lemma gives a bound on the maximum number of Abelian periods of a word.

Lemma 2. *The maximum number of Abelian periods for a word of length n over the alphabet Σ is $\Theta(n^2)$.*

Proof. The word $(a_1 a_2 \cdots a_\sigma)^{n/\sigma}$ has Abelian period (h, p) for any $p \equiv 0 \bmod \sigma$ and $h < p$. □

A natural order can be defined on the Abelian periods.

Definition 3. *Two distinct Abelian periods (h, p) and (h', p') of a word w are ordered as follows: $(h, p) < (h', p')$ if $p < p'$ or $(p = p'$ and $h < h')$.*

We are interested in computing all the Abelian periods of a word. The algorithms we present in this paper can be easily adapted to give only the smallest Abelian period.

3 Off-line algorithms

3.1 Brute-Force algorithm

In Figure 1 we present a Brute-Force algorithm which computes all the Abelian periods of an input word w of length n. For each possible head of length h from 1 to $\lfloor (n-1)/2 \rfloor$ the algorithm tests all the possible values of p such that $p > h$ and $h + p \leqslant n$. This is a reformulation of the algorithm given in [10]. The algorithm easily adapts to give only the smallest Abelian period or the weak repetitions.

```
AbelianPeriod-BruteForce(w, n)
 1  for h ← 0 to ⌊(n − 1)/2⌋ do
 2     p ← h + 1
 3     while h + p ≤ n do
 4        if (h, p) is an Abelian period of w then
 5           Output(h, p)
 6        p ← p + 1
```

Figure 1. Brute-Force algorithm for computing all the Abelian periods of a word w of length n.

Example 4. For $w =$ `abaababa` the algorithm outputs $(1,2)$, $(0,3)$, $(2,3)$, $(1,4)$, $(2,4)$, $(3,4)$, $(0,5)$, $(1,5)$, $(2,5)$, $(3,5)$, $(0,6)$, $(1,6)$, $(2,6)$, $(0,7)$, $(1,7)$ and $(0,8)$. Among these periods $(1,2)$ is the smallest.

Theorem 5. *The algorithm* AbelianPeriod-BruteForce *computes all the Abelian periods of a given word of length n in time $O(n^3 \times \sigma)$ with an $O(\sigma)$ space or in time $O(n^2 \times \sigma)$ with a space in $O(n \times \sigma)$.*

Proof. The correctness of the algorithm comes directly from Definition 1. Each test in line 4 consists in comparing n/p Parikh vectors. Comparing two Parikh vectors can be done in $\Theta(\sigma)$ time. The test in line 4 is performed $\sum_{h=0}^{\lfloor (n-1)/2 \rfloor} \sum_{p=h+1}^{n-h} n/p = O(\sum_{h=1}^{n} \sum_{p=h}^{n} n/p) = O(n^2)$ times. With no preprocessing, this gives an overall time of $O(n^3 \times \sigma)$. If the Parikh vectors of all the prefixes of the word have been already computed, this can be done by computing the difference between two Parikh vectors (see [3]). This requires space and time in $O(n \times \sigma)$ and gives an overall time of $O(n^2 \times \sigma)$. □

3.2 Select-based algorithm

Let us introduce the *select* function [16] defined as follows.

Definition 6. *Let w be a word of length n over alphabet Σ, then $\forall\, a \in \Sigma$:*

- $select_a(w, 0) = 0$;
- $\forall\, 1 \leqslant i \leqslant |w|_a$, $select_a(w, i) = j$ *iff* j *is the position of the i-th occurrence of letter a in w;*
- $\forall\, i > |w|_a$, $select_a(w, i)$ *is undefined.*

In order to compute the *select* function, we consider an array S_w of n elements that stores the increasing ordered positions of a_1, then the increasing ordered positions of a_2 and so on up to the increasing ordered positions of a_σ. In addition to S_w, we also consider an array C_w of $\sigma + 1$ elements such that $C_w[i] = \sharp\{w[k] = a_j \mid j < i \text{ and } 1 \leqslant k \leqslant n\} + 1$ for $1 \leqslant i \leqslant \sigma$ and $C_w[\sigma + 1] = n + 1$. Then, for $i > 0$,

$$select_a(w, i) = \begin{cases} S_w[C_w[ind(a)] + i - 1], & \text{if } i \leqslant C_w[ind(a) + 1] - C_w[ind(a)] \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$C_w[i] - 1$ is the number of letters in w strictly smaller than a_i. Array C_w serves as an index to access S_w.

Example 7. For $w = \texttt{abaababa}$ the *select* function uses the following three arrays:

	1	2	3	4	5	6	7	8
w	a	b	a	a	b	a	b	a

	a	b
ind	1	2

	1	2	3
C_w	1	6	9

	1	2	3	4	5	6	7	8
S_w	1	3	4	6	8	2	5	7

Then $select_{\texttt{b}}(w, 2) = S_w[C_w[ind(\texttt{b})] + 2 - 1] = 5$, which means that the second `b` in w appears in position 5.

The ComputeSelect function (see Figure 2) computes the two arrays C_w and S_w used by the *select* function. This can be done in $O(n + \sigma)$ time and space. Once these two arrays have been computed, each call to the *select* function is answered in constant time.

```
ComputeSelect(w, n)
1  C_w[1] ← 1
2  for i ← 2 to σ + 1 do
3      C_w[i] ← C_w[i − 1] + P_w[i − 1]
4  for i ← 1 to σ do
5      P[i] ← 0
6  for i ← 1 to n do
7      S_w[C_w[ind(w[i])] + P[ind(w[i])] ← i
8      P[ind(w[i])] ← P[ind(w[i])] + 1
9  return (C_w, S_w)
```

Figure 2. Algorithm computing C_w and S_w arrays.

The Brute-Force algorithm tests all possible pairs (h, p) but it is clear that, given h, some pairs cannot be Abelian periods. For example, let $w = \texttt{abaaaaabaa}$ and $h = 2$. Since $\mathcal{P}_w(1, h)$ has to be included in $\mathcal{P}_w(h + 1, p)$, the pairs $(2, 3)$, $(2, 4)$ and $(2, 5)$ cannot be Abelian periods of w: the minimal p value such that $(2, p)$ can be an Abelian period is in fact 6, in order to include the second `b` of w. This remark leads us to give the following definitions and propositions.

Definition 8. *Let w be a word of length n on alphabet Σ. Then $\forall\, 0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$, $\mathcal{M}_w[h]$ is defined by*

$$\mathcal{M}_w[h] = \begin{cases} \min\{p \mid \mathcal{P}_w(1, h) \subset \mathcal{P}_w(h + 1, p)\} & \text{if } \forall\, a \in \Sigma,\ 2 \times |w[1..h]|_a \leqslant |w|_a \\ -1 & \text{otherwise.} \end{cases}$$

In other words, if $\forall\, a \in \Sigma$, $select_a(w, 2 \times |w[1..h]|_a)$ is defined then

$$\mathcal{M}_w[h] = \max\{h + 1, \max\{select_a(w, 2 \times |w[1..h]|_a) \mid a \in \Sigma\} - h\},$$

otherwise $\mathcal{M}_w[h] = -1$.

Proposition 9. *Let w be a word of length n on alphabet Σ and $0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$. If $\mathcal{M}_w[h] = -1$, then $\mathcal{M}_w[h'] = -1\ \forall\, h' \geqslant h$ and h' cannot be a head of an Abelian period of w.*

Proof. If $\mathcal{M}_w[h] = -1$, then by definition $\exists\, a \in \Sigma$ such that $2 \times |w[1..h]|_a > |w|_a$. Then, we cannot find a value p such that $|w[1..h]|_a \leqslant |w[(h + 1)..(h + p)]|_a$. It is clear that this is also true for all value $h' > h$. □

```
ComputeM(w, n, C_w, S_w)
 1  M_w[0] ← 0
 2  for a ∈ Σ do
 3     H[a] ← 0
 4  for h ← 1 to ⌊(n−1)/2⌋ do
 5     H[w[h]] ← H[w[h]] + 1
 6     s ← select_{w[h]}(w, 2 × H[w[h]])
 7     if s is defined then
 8        M_w[h] ← max{M_w[h − 1] − 1, s − h}
 9     else M_w[h] ← −1
10  for h ← 1 to ⌊(n−1)/2⌋ do
11     if M_w[h] = h then
12        M_w[h] ← h + 1
13  return M_w
```

Figure 3. Algorithm computing the $\mathcal{M}_w$ array.

The array $\mathcal{M}_w$ can be computed in time and space $O(n+\sigma)$, processing positions of w from left to right (see Figure 3).

Consider now the following definition.

Definition 10. *Let w be a word of length n on alphabet Σ. Then $\forall\, 0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$, $\mathcal{G}_w[h]$ is defined by*

$$\mathcal{G}_w[h] = \max\{\ \mathit{select}_a(w, i+1) - \mathit{select}_a(w, i) \mid a \in \Sigma,\\ h < \mathit{select}_a(w, i) < \mathit{select}_a(w, i+1) \leqslant n\}.$$

Actually, $\mathcal{G}_w[h]$ is the maximal value $j - i$ such that $h < i < j$ and $w[i] = w[j]$.

The array $\mathcal{G}_w$ can be computed in time and space $O(n+\sigma)$, processing positions of w from right to left (see Figure 4).

```
ComputeG(w, n)
 1  G_w[n] ← 0
 2  for a ∈ Σ do
 3     T[a] ← 0
 4  for h ← n to 1 do
 5     if T[w[h]] = 0 then
 6        T[w[h]] ← h
 7        G_w[h − 1] ← G_w[h]
 8     else d ← T[w[h]] − h
 9        T[w[h]] ← h
10        G_w[h − 1] ← max{G_w[h], d}
11  return G_w
```

Figure 4. Algorithm computing the $\mathcal{G}_w$ array.

Proposition 11. *Let w be a word of length n on alphabet Σ. Let $0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$. If $h < p < \max\{\mathcal{M}_w[h], \lfloor (\mathcal{G}_w[h]+1)/2 \rfloor\}$ then (h, p) is not an Abelian period of w.*

Proof. From the definition of $\mathcal{M}_w[h]$, it directly follows that if $p < \mathcal{M}_w[h]$, then (h, p) cannot be an Abelian period of w.

Given h, let $a \in \Sigma$ be such that there exists $1 \leqslant i < n$ and $\mathit{select}_a(w, i+1) - \mathit{select}_a(w, i) = \mathcal{G}_w[h]$. Let $j = \mathit{select}_a(w, i)$ and $j' = \mathit{select}_a(w, i+1)$. If $p < \lfloor (\mathcal{G}_w[h] + 1)/2 \rfloor$, $k = \min\{k' \mid h + k'p \geqslant j\}$ then $h + (k+1)p < j'$ and $|w[k + kp + 1\,..\,h + (k+1)p]|_a = 0$. Thus (h, p) cannot be an Abelian period of w (see Figure 5). □

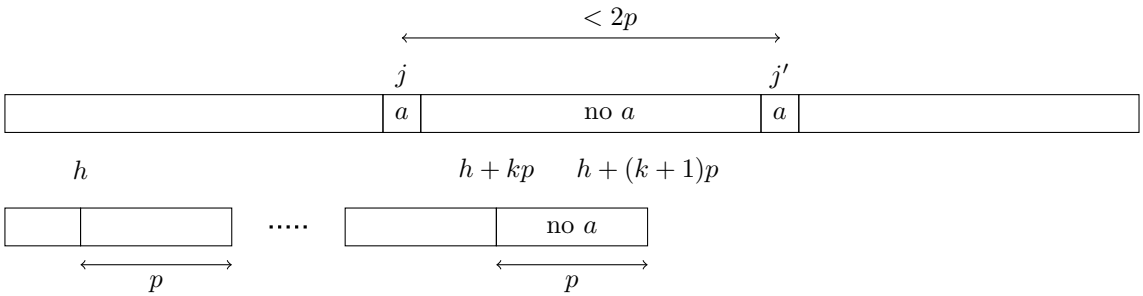


Figure 5. If the distance between two consecutive a's in w is greater than $2p$ then (h, p) cannot be an Abelian period of w for any $h < p$.

Arrays $\mathcal{M}_w$ and $\mathcal{G}_w$ give us, for every head length h, a minimal value for a possible p such that (h, p) can be an Abelian period of w. This allows us to skip a number of values for p that cannot give an Abelian period.

The following lemma shows how to check if (h, p) is indeed an Abelian period of w (except for the tail).

Lemma 12. *Let w be a word of length n on alphabet Σ. Let $\mathcal{H} = \mathcal{P}_w(1, h)$ and $\mathcal{P} = \mathcal{P}_w(h + 1, p)$. Let $i = h + kp$ such that $0 < k$, $p \leqslant n - i$ and (h, p) is an Abelian period of $w[1..i]$ (with an empty tail). Then the following two points are equivalent:*

1. *(h, p) is an Abelian period of $w[1..i + p]$.*
2. *for all $a \in \Sigma$*

$$select_a(w, \mathcal{H}[ind(a)] + \left(1 + \left\lfloor \frac{i}{p} \right\rfloor\right) \times \mathcal{P}[ind(a)]) \leqslant i + p.$$

Proof. Since (h, p) is an Abelian period of $w[1..i]$ with $i = h + kp$ for some $k > 0$ then $|w[1..i]|_a = \mathcal{H}[ind(a)] + k \times \mathcal{P}[ind(a)]$ for each letter $a \in \Sigma$. Notice that since $h < p$ then $k = \lfloor i/p \rfloor$.
$(1 \Rightarrow 2)$. The fact that (h, p) is an Abelian period of $w[1..i + p]$ implies that, for all $a \in \Sigma$, $|w[1..i + p]|_a = \mathcal{H}[ind(a)] + (k + 1) \times \mathcal{P}[ind(a)]$. Thus, by definition of *select*, $select_a(w, \mathcal{H}[ind(a)] + (1 + \lfloor i/p \rfloor) \times \mathcal{P}[ind(a)]) \leqslant i + p$.
$(2 \Rightarrow 1)$. The fact that $select_a(w, \mathcal{H}[ind(a)] + (1 + \lfloor i/p \rfloor) \times \mathcal{P}[ind(a)]) \leqslant i + p$ implies that $|w[1..i + p]|_a = \mathcal{H}[ind(a)] + (k + 1) \times \mathcal{P}[ind(a)]$. We know that $|w[1..i]|_a = \mathcal{H}[ind(a)] + k \times \mathcal{P}[ind(a)]$. By difference, $|w[i+1..i+p]|_a = \mathcal{P}[ind(a)]$. Since it is true for all $a \in \Sigma$, $\mathcal{P}_w(i+1, p) = \mathcal{P}$ and then (h, p) is an Abelian period of $w[1..i+p]$. □

Figure 6 presents the algorithm AbelianPeriod-Shift based on the previous lemma.

Proposition 13. *Algorithm* AbelianPeriod-Shift(h, p, w, n, C_w, S_w) *returns* TRUE *iff (h, p) is an Abelian period of the prefix of length $n - ((n - h) \bmod p)$ of w in time $O(\frac{n}{p} \times \sigma)$ and space $O(\sigma)$.*

Proof. The correctness comes directly from Lemma 12. The **while** loop in line 3 is executed n/p times and the **for** loop in line 4 is executed σ times, thus the time complexity is $O(\frac{n}{p} \times \sigma)$. This algorithm only requires the storage of the two Parikh vectors $\mathcal{P}_w(1, h)$ and $\mathcal{P}_w(h + 1, p)$. These vectors can be stored in space $O(\sigma)$ under the standard assumption that $\log n$ fits in a computer word. □

```
AbelianPeriod-Shift(h, p, w, n, C_w, S_w)
 1  (H, P) ← (P_w(1, h), P_w(h + 1, p))
 2  i ← h + p
 3  while i + p ⩽ n do
 4     for a ∈ Σ do
 5        s ← select_a(w, H[ind(a)] + (1 + ⌊i/p⌋) × P[ind(a)])
 6        if s is undefined or s > i + p then
 7           return FALSE
 8     i ← i + p
 9  return TRUE
```

Figure 6. Algorithm checking whether (h, p) is an Abelian period of the prefix of length $n - ((n - h) \bmod p)$ of w.

```
AbelianPeriod-Select(w, n)
 1  (C_w, S_w) ← ComputeSelect(w, n)
 2  M_w ← ComputeM(w, n, C_w, S_w)
 3  G_w ← ComputeG(w, n)
 4  h ← 0
 5  while h ⩽ ⌊(n − 1)/2⌋ and M_w[h] ≠ −1 do
 6     p ← max(M_w[h], ⌊(G_w[h] + 1)/2⌋)
 7     while h + p ⩽ n do
 8        if AbelianPeriod-Shift(h, p, w, n, C_w, S_w) then
 9           t ← (n − h) mod p
10           if P_w(n − t + 1, t) ⊂ P_w(h + 1, p) then
11              Output(h, p)
12        p ← p + 1
13     h ← h + 1
```

Figure 7. Algorithm computing all the Abelian periods of word w of length n, based on the *select* function.

Using Proposition 11 and Proposition 13, algorithm AbelianPeriod-Select, given in Figure 7, computes all the Abelian periods of a word w of length n.

Theorem 14. *Algorithm* AbelianPeriod-Select *computes all the Abelian periods of word w of length n in time $O(n^2 \times \sigma)$ and space $O(n + \sigma)$.*

Proof. The correctness of the algorithm comes from Proposition 11 and Proposition 13.

The *select* function and the arrays $\mathcal{M}_w$ and $\mathcal{G}_w$ can be computed in $O(n + \sigma)$ time and space. According to Proposition 11, the value of p computed in line 6 is the minimal value such that (h, p) can be an Abelian period of w. The AbelianPeriod-Shift function, called in line 8, simply verifies that (h, p) is an Abelian period of w in time $O(\frac{n}{p} \times \sigma)$. The test in line 10 is done in $O(p)$ time. The complexity of the **while** loop in line 7 is $O(\sum_{p=h+1}^{n} \frac{n}{p}) = O(n)$. Consequently, algorithm AbelianPeriod-Select computes all the Abelian periods of w in time $O(n^2 \times \sigma)$ and space $O(n + \sigma)$ (output periods are not stored). □

4 On-line algorithms

We now propose two on-line algorithms to compute all the Abelian periods of a word w using dynamic programming. When processing $w[i]$, in the first algorithm, using

a two dimensional array, we inspect all the possible values (h, p); in the second one, using heaps, we inspect the Abelian periods of $w[1..i-1]$ by groups built depending upon the tail length of the periods.

The following proposition states that if (h, p) is not an Abelian period of a word w, with $h + p \leqslant n = |w|$, then it cannot be an Abelian period of any word having w as prefix.

Proposition 15. *Let w be a word of length n and let h, p such that $h + p \leqslant n$. If (h, p) is not an Abelian period of w, then (h, p) is not an Abelian period of wa for any symbol $a \in \Sigma$.*

Proof. If (h, p) is not an Abelian period of w, at least one of the following three cases holds:

1. $\mathcal{P}_w(1, h) \not\subset \mathcal{P}_w(h + 1, p)$;
2. there exist two distinct indices $h \leqslant i, i' \leqslant |w| - p + 1$ such that $i = kp + h + 1$ and $i' = k'p + h + 1$ with k and k' two integers and $\mathcal{P}_w(i, p) \neq \mathcal{P}_w(i', p)$;
3. $t = (|w| - h) \bmod p$ and $\mathcal{P}_w(|w| - t + 1, t) \not\subset \mathcal{P}_w(|w| - p - t + 1, p)$.

If case 1 holds then $\mathcal{P}_{wa}(1, h) \not\subset \mathcal{P}_{wa}(h + 1, p)$ and (h, p) is not an Abelian period of wa. If case 2 holds then $\mathcal{P}_{wa}(i, p) \neq \mathcal{P}_{wa}(i', p)$ and (h, p) is not an Abelian period of wa. If case 3 holds then $\mathcal{P}_{wa}(|w| - t + 1, t + 1) \not\subseteq \mathcal{P}_{wa}(|w| - p - t + 1, p)$ and (h, p) is not an Abelian period of wa. □

4.1 Two dimensional array

We now propose an algorithm that uses a two dimensional array and Proposition 15 to compute all the Abelian periods of an input word w in an on-line manner. It processes the positions of w in increasing order. When processing position i, $T[h, p] = j$ iff $w[1..j]$ is the longest prefix of $w[1..i]$ having Abelian period (h, p). Thus if $j = i - 1$ the algorithm checks whether $w[1..i]$ has Abelian period (h, p) and updates $T[h, p]$ accordingly.

When $T[h, p] = i$ it means that $w[1..i]$ is the longest prefix of w that has (h, p) as an Abelian period. Thus when $T[h, p] = n$ it means that (h, p) is an Abelian period of w.

Example 16. For $w =$ `abaababa` the algorithm computes the following array T:

$h\backslash p$	1	2	3	4	5	6	7	8
0	1	3	8	6	8	8	8	8
1		8	6	8	8	8	8	
2			8	8	8	8		
3				8	8			

Cells $T[h, p] = |w|$ correspond to pairs (h, p) output by algorithm AbelianPeriodBruteForce of example 4. Empty cells on the left part of the array correspond to cases where $h \geq p$ and empty cells on the right part correspond to cases where $h + p > |w|$.

In order to improve the space complexity of this algorithm, the Abelian periods can be stored in a list instead of an array: When processing position i one only stores pairs (h, p) such that $w[1..i]$ has Abelian period (h, p); these pairs correspond to all

the cells of array T, computed by the previous algorithm, such that $T[h, p] = i$. At the end of this process, when $i = n$, this list contains all the Abelian periods of w, and only them.

The above algorithms computes all the Abelian periods of a word of length n on an alphabet of size σ in $O(n^3 \times \sigma)$ time using $O(n^2)$ space.

4.2 Heaps

The following proposition shows that the set of Abelian periods of a prefix of a word can be partitioned into subsets depending of the length of the tail. In some cases all the periods of a subset can be processed at once by inspecting only the smallest period of the subset.

Proposition 17. *Let w have s Abelian periods $(h_1, p_1) < (h_2, p_2) < \cdots < (h_s, p_s)$ such that $(|w| - h_i) \bmod p_i = t > 0$ for $1 \leqslant i \leqslant s$. If (h_1, p_1) is an Abelian period of wa for any symbol $a \in \Sigma$ then $(h_2, p_2), \ldots, (h_s, p_s)$ are also Abelian periods of wa.*

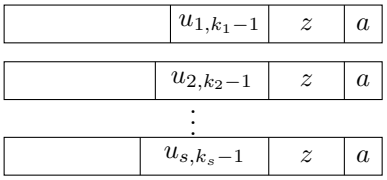


Figure 8. $w = u_{i,0}u_{i,1}\cdots u_{i,k_i-1}u_{i,k_i}$, $u_{i,k_i} = z$ for $1 \leqslant i \leqslant s$. If $\mathcal{P}_{za} \subseteq \mathcal{P}_{u_{1,k_1-1}}$ then $\mathcal{P}_{za} \subseteq \mathcal{P}_{u_{i,k_i-1}}$ for every $2 \leqslant i \leqslant s$.

Proof. Since $(h_1, p_1) < (h_2, p_2) < \cdots < (h_s, p_s)$ are Abelian periods of w, $w = u_{i,0}u_{i,1}\cdots u_{i,k_i-1}u_{i,k_i}$ with $|u_{i,0}| = h_i$, $|u_{i,j}| = p_i$ and $|u_{i,k_i}| = t$ for $1 \leqslant i \leqslant s$ and $1 \leqslant j \leqslant k_i$. If (h_1, p_1) is an Abelian period of wa, $\mathcal{P}_{u_{1,k_1}a} \subseteq \mathcal{P}_{u_{1,k_1-1}}$. Since $|u_{1,k_1}| = |u_{i,k_i}|$ and $|u_{1,k_1-1}| \leqslant |u_{i,k_i-1}|$ we have that $\mathcal{P}_{u_{i,k_i}a} \subseteq \mathcal{P}_{u_{i,k_i-1}}$ for $2 \leqslant i \leqslant s$. Thus $(h_2, p_2), \ldots, (h_s, p_s)$ are Abelian periods of wa (see Figure 8). □

The algorithm given in Figure 9 uses Proposition 17 for computing all the Abelian periods by gathering all the ongoing periods (h, p) with the same tail length together in a heap where the element at the root of the heap is the smallest period.

When processing $w[i]$, the algorithm processes every heap H for the different tail lengths:

- if the period (h, p) at the root of H is a period of $w[1..i]$ then by Proposition 17 all the elements of H are Abelian periods of $w[1..i]$. If the tail length becomes equal to p then (h, p) is removed from the current heap and is moved into a new heap corresponding to the empty tail.
- if the period (h, p) at the root of H is not a period of $w[1..i]$ then it is removed from H and the same process is applied until a pair (h', p') is an Abelian period of $w[1..i]$ or the heap becomes empty.
  In the former case, by Proposition 17, all the remaining elements of H are Abelian periods of $w[1..i]$. This is realized by function ExtractUntilOK in line 8.

Then all the degenerate cases (h, p) such that $h < p$ and $h + p = i$ have to be inserted in the heap corresponding to the empty tail (lines 12 to 15).

The function Root(H) returns the smallest element of the heap H, the function Insert(H, e) inserts element e in the heap H, while the function Remove(H) removes the smallest element of the heap H.

```
AbelianPeriod-heap(w, n)
 1  L ← list with one heap containing (0, 1)
 2  for i ← 2 to n do
 3     NewHeap ← ∅
 4     for all H ∈ L do
 5        (h, p) ← Root(H)
 6        t ← p − ((i − h) mod p)
 7        if P_w(i − t + 1, t) ⊈ P_w(i − t − p + 1, p) then
 8           ExtractUntilOK(H)
 9        else if t = p then
10              Remove(H)
11              Insert(NewHeap, (h, p))
12     h ← 0
13     while h < ⌊(i + 1)/2⌋ and P_w(1, h) ⊂ P_w(h + 1, i − h) do
14        Insert(NewHeap, (h, i − h))
15        h ← h + 1
16     L ← L ∪ NewHeap
17  return L
```

Figure 9. On-line algorithm for computing all the Abelian periods of a word w of length n using heaps.

Theorem 18. *The algorithm* AbelianPeriod-heap *computes all the Abelian periods of a given word of length n in time $O(n^2 \times (n \log n) \times \sigma)$ and space $O(n^2)$.*

Proof. The correctness of the algorithm comes from Proposition 17. The maximum number of heaps is $n/2$ and the total number of elements of all the heaps is $O(n^2)$ (Lemma 2). The space complexity for the list L is $O(n^2)$. The time complexity of the algorithm is due to the two **for** loops of lines 2 and 4 and the different calls to ExtractUntilOK in line 8 and Insert and Remove. The maximum number of heaps is $n/2$, and the maximum number of elements in a single heap is n. Thus, the total complexity for the calls to ExtractUntilOK, Insert and Remove in a single run of the **for** loop of line 4 is $O(n \log n)$. □

5 Experimental results

To compare practical performances of the different algorithms, they have been implemented in C in a homogeneous way and run on test sets of random words (1000 words each) of different lengths (from 10 to 2000) on different alphabet sizes (2, 3, 4, 8 and 16).

Tests were performed on a computer running Mac OS X with a 2.2 GHz processor and 2 GB RAM.

Figure 10 presents average running times over 1000 random words on alphabet size 16 of the algorithms AbelianPeriod-BruteForce, AbelianPeriod-Select and AbelianPeriod-Heaps. Corresponding values are given Figure 11. The results show that, as expected, the off-line algorithm using *select* function is indeed faster than the other ones. Moreover, our tests show that, for long words, the on-line algorithm using heaps becomes faster than the Brute-Force one. One can notice that the difference of running times between the three algorithms increases as the word length grows. Results for other alphabet sizes, natural languages texts or genomic sequences are not shown since they are similar to these ones.

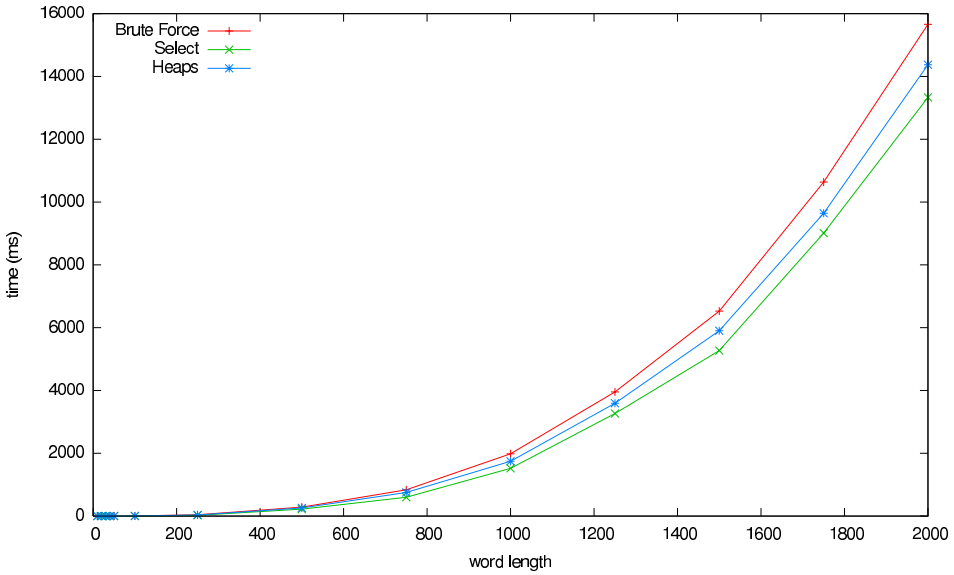


Figure 10. Average running times (in ms) over 1000 random words, of the Brute-Force, *select*-based and heaps-based algorithms on alphabet size 16.

algo|w|	10	20	30	40	50	100	250	500	750	1000	1250	1500	1750	2000
Brute-Force	0	0	0	0	0	4	43	286	836	1984	3952	6527	10636	15560
Select	0	0	0	0	0	2	25	221	599	1519	3267	5270	9009	13334
Heaps	0	0	0	0	0	3	32	260	752	1746	3592	5901	9643	14372

Figure 11. Values of average running times (in ms) of the Brute-Force, *select*-based and heaps-based algorithms on alphabet size 16.

6 Conclusion and perspectives

In this paper we presented different algorithms to compute all the Abelian periods of a word. This is the first attempt to give algorithms for computing all the Abelian periods of a word. In particular, we give a $O(n^2 \times \sigma)$ time off-line algorithm requiring $O(n + \sigma)$ space, thus reducing the space complexity compared to the Brute-Force algorithm. Moreover, in practice, this algorithm appears to be faster. It is even faster when one wants to compute Abelian periods (h, p) of a word w with at least two consecutive factors of length p having the same Parikh vector, *i.e.* $h + 2p \leqslant |w|$ (see Figure 12).

Cutting positions of an Abelian period (h, p) of a word w can be defined as follows:

$$Cut_w(h, p) = \{k = h + jp \mid 1 \leqslant k \leqslant |w| \text{ and } 0 \leqslant j\}.$$

An Abelian period (h, p) of w is non-deducible if there does not exist another Abelian period (h', p') of w such that $Cut_w(h, p) \subset Cut_w(h', p')$. In order to improve algorithm complexities, one way would consist in reporting only non-deducible Abelian periods.

It remains to obtain a bound on the minimal Abelian period given a word length and an alphabet size. Simple modifications of the presented algorithms would allow one to compute the minimal Abelian period of each factor of a word. This could have practical applications in areas such as bioinformatics and more precisely in the detection of DNA regions of homogeneous compositions.

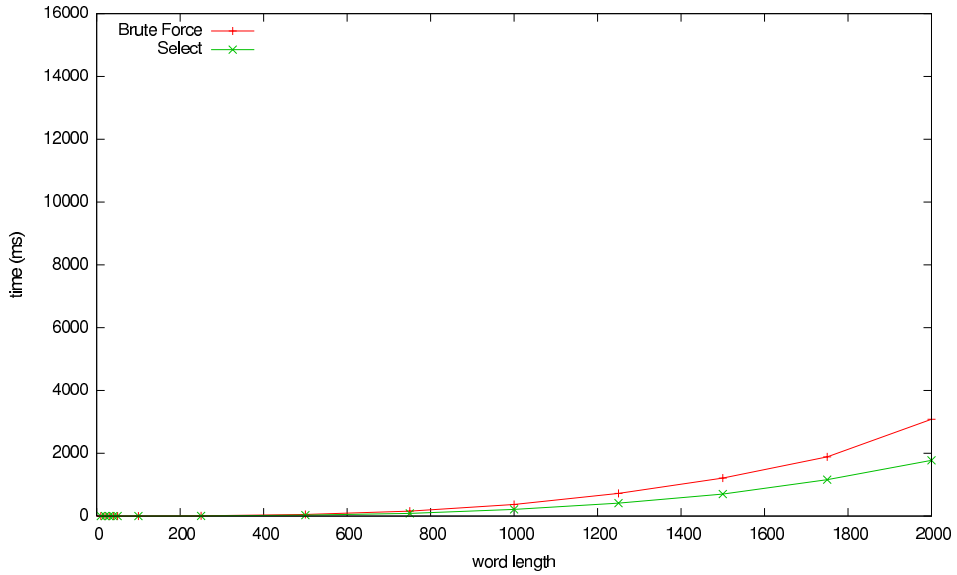


Figure 12. Average running times (in ms) over 1000 random words, of the Brute-Force and *select*-based algorithms on alphabet size 16, in the case where $h + 2p \leqslant |w|$. See the difference with Figure 10.

References

1. S. Avgustinovich, J. Karhumäki, and S. Puzynina: *On Abelian versions of Critical Factorization Theorem*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.
2. S. V. Avgustinovich, A. Glen, B. V. Halldórsson, and S. Kitaev: *On shortest crucial words avoiding abelian powers*. Discrete Applied Mathematics, 158(6) 2010, pp. 605–607.
3. G. Benson: *Composition alignment*, in Algorithms in Bioinformatics, Third International Workshop, WABI 2003, Budapest, Hungary, September 15-20, 2003, Proceedings, G. Benson and R. Page, eds., vol. 2812 of Lecture Notes in Computer Science, Springer, 2003, pp. 447–461.
4. F. Blanchet-Sadri, J. I. Kim, R. Mercas, W. Severa, and S. Simmons: *Abelian square-free partial words*, in Proceedings of the 4th International Conference Language and Automata Theory and Applications, A.-H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 94–105.
5. F. Blanchet-Sadri, A. Tebbe, and A. Veprauskas: *Fine and Wilf's theorem for abelian periods in partial words*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.
6. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *Algorithms for jumbled pattern matching in strings*. International Journal of Foundations of Computer Science, to appear.
7. J. Cassaigne, G. Richomme, K. Saari, and L. Q. Zamboni: *Avoiding abelian powers in binary words with bounded abelian complexity*. International Journal of Foundations of Computer Science, 22(4) 2011.
8. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, J. Holub and J. Žďárek, eds., Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009, pp. 105–117.
9. S. Constantinescu and L. Ilie: *Fine and Wilf's theorem for abelian periods*. Bulletin of the European Association for Theoretical Computer Science, 89 2006, pp. 167–170.
10. L. J. Cummings and W. F. Smyth: *Weak repetitions in strings*. Journal of Combinatorial Mathematics and Combinatorial Computing, 24 1997, pp. 33–48.
11. J. D. Currie and A. Aberkane: *A cyclic binary morphism avoiding abelian fourth powers*. Theoretical Computer Science, 410(1) 2009, pp. 44–52.
12. M. Domaratzki and N. Rampersad: *Abelian primitive words*, in Proceedings of the 15th Conference on Developments in Language Theory, G. Mauri and A. Leporati, eds., vol. 6795 of Lecture Notes in Computer Science, Springer, 2011.

13. M. Lothaire: *Algebraic Combinatorics on Words*, Cambridge University Press, 2002.
14. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Information Processing Letters, 110(18-19) 2010, pp. 795–798.
15. T. M. Moosa and M. S. Rahman: *Sub-quadratic time and linear size data structures for permutation matching in binary strings.* Journal of Discrete Algorithms, to appear.
16. G. Navarro and V. Mäkinen: *Compressed full-text indexes.* ACM Computing Surveys, 39(1) 2007.
17. A. V. Samsonov and A. M. Shur: *On abelian repetition threshold*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.

Computing Longest Common Substring/Subsequence of Non-linear Texts

Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819–0395, Japan
{kouji.shimohira,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. A non-linear text is a directed graph where each vertex is labeled with a string. In this paper, we introduce the longest common substring/subsequence problems on non-linear texts. Firstly, we present an algorithm to compute the longest common substring of non-linear texts G_1 and G_2 in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space, when at least one of G_1 and G_2 is acyclic. Here, V_i and E_i are the sets of vertices and arcs of input non-linear text G_i, respectively, for $1 \leq i \leq 2$. Secondly, we present algorithms to compute the longest common subsequence of G_1 and G_2 in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space, when both G_1 and G_2 are acyclic, and in $O(|E_1||E_2|+|V_1||V_2|\log|\Sigma|)$ time and $O(|V_1||V_2|)$ space if G_1 and/or G_2 are cyclic, where, Σ denotes the alphabet.

1 Introduction

We consider *non-linear texts*, which are directed graphs where vertices are labeled by strings. Pattern matching on non-linear texts was first considered in [3], where an $O(N + m|E| + R\log\log m)$ time algorithm for directed acyclic graphs. Here, m is the pattern length, N is the number of vertices, $|E|$ is the number of arcs, and R is the output size. The algorithm was improved in [6], where an $O(n + m|E|)$ time algorithm was shown. Here, n represents the total length of the string labels in the graph. Furthermore, in [1], an $O(n)$ time algorithm was shown for trees. The problem was solved for general directed graphs in [2], where an $O(n + |E|)$ time algorithm was developed. The approximate matching problem for non-linear texts was also considered in [2], where they showed that the problem can be solved in $O(m(n\log m + e))$ time when edit operations are only allowed in the pattern. Here, e denotes the number of arcs in the graph when the graph is converted so that each node is labeled by a single character. They also showed that the problem is NP-complete when edit operations are allowed on the non-linear text. Furthermore, in [5], the algorithm was improved to $O(m(n + e))$.

Note that previous work on pattern matching on non-linear texts assumed a linear pattern. In this paper, we study a more generalized version of the problem, and consider the *longest common substring* and *longest common subsequence* problems between two non-linear texts. Firstly, we present an algorithm to compute the longest common substring of non-linear texts G_1 and G_2 in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space, where V_i and E_i are the sets of vertices and arcs of input non-linear text G_i, respectively, for $1 \leq i \leq 2$. The algorithm works if one of G_1 and G_2 is acyclic. Secondly, we present algorithms to compute the longest common subsequence in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space if both G_1 and G_2 are acyclic, and in $O(|E_1||E_2| + |V_1||V_2|\log|\Sigma|)$ time and $O(|V_1||V_2|)$ space if G_1 and/or G_2 are cyclic. Cyclic non-linear texts represent infinitely many and long strings, but our algorithms solve the above problems quite efficiently. Our algorithms are natural

Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Computing Longest Common Substring/Subsequence of Non-linear Texts, pp. 197–208.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

extension of classical dynamic programming methods to compute longest common substring/subsequence of linear strings, and hence are easy to understand.

Very recently, an algorithm for determining the longest common subsequence between two *finite* languages was shown in [7]. The algorithm is a modification of the method based on weighted transducers [4], and requires $O(|\Sigma|^2|E_1||E_2|)$ time and space. Compared to this work, our algorithms are faster and also apply to infinite languages.

problem	text	pattern	time complexity	
Substring Matching	acyclic graph	linear	$O(n+m\|E\|)$	[6]
	tree	linear	$O(n)$	[1]
	graph	linear	$O(n+\|E\|)$	[2]
Approximate Matching	graph w/edit operations	linear	NP-complete	[2]
	graph	linear w/edit operations	$O(m(n+e))$	[5]
	text1	text2		
Longest Common Substring	acyclic graph	acyclic graph	$O(\|E_1\|\|E_2\|)$	(this work)
	graph	acyclic graph	$O(\|E_1\|\|E_2\|)$	(this work)
Longest Common Subsequence	acyclic graph	acyclic graph	$O(\|\Sigma\|^2\|E_1\|\|E_2\|)$	[7]
	acyclic graph	acyclic graph	$O(\|E_1\|\|E_2\|)$	(this work)
	graph	graph	$O(\|E_1\|\|E_2\| + \|V_1\|\|V_2\| \log \|\Sigma\|)$	(this work)

Table 1. Algorithms on non-linear text.

2 Preliminaries

2.1 Notation

Let Σ be a finite alphabet, and the elements of Σ^* are called *strings*. The *length* of a string w is denoted by $|w|$. The *empty string*, denoted by ε, is a string of length 0, and thus $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Strings x, y, and z are called a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. For any string w, let $\mathit{suffix}(w)$ denote the set of suffixes of w. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. The set of substrings of a string w is denoted by $\mathit{substr}(w)$. A string u is a *subsequence* of another string w if there exists a sequence of integers $i_1, \ldots, i_k$ with $k \geq 0$ such that $1 \leq i_1 < \cdots < i_k \leq |w|$ and $u = w[i_1] \cdots w[i_k]$.

A *directed graph* is an ordered pair (V, E) of set V of *vertices* and set $E \subseteq V \times V$ of *arcs*. A *path* in a directed graph $G = (V, E)$ is a sequence $v_0, \ldots, v_k$ of vertices such that $(v_{i-1}, v_i) \in E$ for every $i = 1, \ldots, k$. For any vertex $v \in V$, let $P(v)$ denote the set of paths that end at vertex v. The set of all paths in G is denoted by $P(G)$, namely, $P(G) = \{P(v) \mid v \in V\}$.

2.2 Longest common substring problem

The *longest common substring* problem is, given two strings x and y, to compute the length of longest common substrings of them. Although this problem can be solved in $O(|x| + |y|)$ time using the generalized suffix tree of x and y, we here mention a

dynamic programming based solution. Letting $C_{i,j}$ denote the maximum length of common suffixes of $x[1..i]$ and $y[1..j]$, it suffices to compute the maximum of $C_{i,j}$ over all the pairs (i, j). Since we have

$$C_{i,j} = \begin{cases} 1 + C_{i-1,j-1} & \text{if } i, j > 0 \text{ and } x[i] = y[j]; \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

the problem can be solved in $O(|x||y|)$ time.

2.3 Longest common subsequence problem

The *longest common subsequence* problem is, given two strings x and y, to compute the length of longest common subsequences of them. It is well-known that, this problem can be solved in $O(|x||y|)$ time by using the following recurrence:

$$C_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0; \\ 1 + C_{i-1,j-1} & \text{if } i, j > 0 \text{ and } x[i] = y[j]; \\ \max(C_{i-1,j}, C_{i,j-1}) & \text{if } i, j > 0 \text{ and } x[i] \neq y[j], \end{cases} \tag{2}$$

where $C_{i,j}$ is the length of longest common subsequence of $x[1..i]$ and $y[1..j]$.

2.4 Non-linear texts

A *non-linear text* is a directed graph with vertices labeled by strings, namely, it is a directed graph $G = (V, E, L)$ where V is the set of vertices, E is the set of arcs, and $L : V \to \Sigma^+$ is a labeling function that maps nodes $v \in V$ to non-empty strings $L(v) \in \Sigma^+$. For a path $p = v_0, \ldots, v_k \in P(G)$, let $L(p)$ denote the string spelled out by p, namely $L(p) = L(v_0) \cdots L(v_k)$. The size $|G|$ of a non-linear text $G = (V, E, L)$ is $|V| + |E| + \sum_{v \in V} |L(v)|$. Let $\mathit{substr}(G)$, $\mathit{suffix}(G)$, and $\mathit{subseq}(G)$ be the sets of substrings, suffices and subsequences of a non-linear text $G = (V, E, L)$, namely,

$$\begin{aligned} \mathit{substr}(G) &= \{\mathit{substr}(L(p)) \mid p \in P(G)\}, \\ \mathit{suffix}(G) &= \{\mathit{suffix}(L(p)) \mid p \in P(G)\}, \\ \mathit{subseq}(G) &= \{\mathit{subseq}(L(p)) \mid p \in P(G)\}. \end{aligned}$$

For a non-linear text $G = (V, E, L)$, consider a non-linear text $G' = (V', E', L')$ such that $L' : V' \to \Sigma$,

$$\begin{aligned} V' &= \{v_{i,j} \mid L'(v_{i,j}) = L(v_i)[j], v_i \in V, 1 \leq j \leq |L(v_i)|\}, \text{ and} \\ E' &= \{(v_{i,|L(v_i)|}, v_{k,1}) \mid (v_i, v_k) \in E\} \cup \{(v_{i,j}, v_{i,j+1}) \mid v_i \in V, 1 \leq j < |L(v_i)|\}. \end{aligned}$$

Namely, G' is a non-linear text in which each vertex is labeled with a single character and $\mathit{substr}(G') = \mathit{substr}(G)$. An example is shown in Figure 1. Since $|V'| = \sum_{v \in V} |L(v)|$, $|E'| = |E| + \sum_{v \in V}(|L(v)| - 1)$, and $\sum_{v' \in V'} |L(v')| = \sum_{v \in V} |L(v)|$, we have $|G'| = O(|G|)$. We remark that given G, we can easily construct G' in $O(|G|)$ time. Observe that $\mathit{subseq}(G) = \mathit{subseq}(G')$ also holds.

In the sequel we only consider non-linear texts where each vertex is labeled with a single character. For any non-linear text $G = (V, E, L)$ such that $L(v) \in \Sigma$ for any $v \in V$, it trivially holds that $\mathit{substr}(G) = \{L(p) \mid p \in P(G)\}$.

We sometimes call strings in Σ^* linear strings or linear texts, in order to clearly distinguish them from non-linear texts.

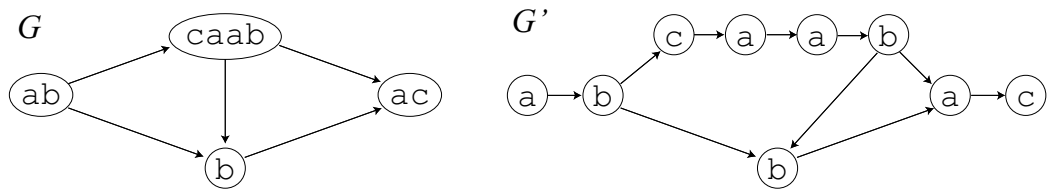


Figure 1. A non-linear text $G = (V, E, L)$ with $L : V \to \Sigma^+$ and its corresponding non-linear text $G' = (V', E', L')$ with $L' : V' \to \Sigma$.

3 Computing Longest Common Substring of Non-linear Texts

In this section, we tackle the problem of computing the length of longest common substrings of two input non-linear texts. The problem is formalized as follows.

Problem 1 (Longest common substring problem for non-linear texts).

Input: Non-linear texts $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$.
Output: The length of a longest string in $substr(G_1) \cap substr(G_2)$.

For example, see the non-linear texts G_1 and G_2 of Figure 2. The solution to the above problem is 5, since there is a longest common substring `abbaa` of G_1 and G_2.

For simplicity, let us first consider the case where the two input non-linear texts are both acyclic.

Theorem 2. *If G_1 and G_2 are acyclic, then Problem 1 can be solved in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space.*

Proof. Let $v_{1,i}$ and $v_{2,j}$ denote the i-th and j-th vertex in topological ordering in G_1 and in G_2, for $1 \leq i \leq |V_1|$ and $1 \leq j \leq |V_2|$, respectively. Let $C_{i,j}$ denote the length of a longest string in $suffix(L_1(P(v_{1,i}))) \cap suffix(L_2(P(v_{2,j})))$. $C_{i,j}$ can be calculated as follows.

1. If $L_1(v_{1,i}) = L_2(v_{2,j})$, there are two cases to consider:
   (a) If there are no arcs to $v_{1,i}$ or to $v_{2,j}$, i.e., $P(v_{1,j}) = \{v_{1,i}\}$ or $P(v_{2,j}) = \{v_{2,j}\}$, then clearly $C_{i,j} = 1$.
   (b) Otherwise, let $v_{1,k}$ and $v_{2,\ell}$ be any nodes s.t. $(v_{1,k}, v_{1,i}) \in E_1$ and $(v_{2,\ell}, v_{2,j}) \in E_2$, respectively. Let z be a longest string in $suffix(L_1(P(v_{1,i}))) \cap suffix(L_2(P(v_{2,j})))$. Assume on the contrary that there exists a string $y \in suffix(L_1(P(v_{1,k}))) \cap suffix(L_2(P(v_{2,\ell})))$ such that $|y| > |z| - 1$. This contradicts that z is a longest common suffix of $L_1(P(v_{1,i}))$ and $L_2(P(v_{2,j}))$, since $L_1(v_{1,i}) = L_2(v_{2,j})$. Hence $y \leq |z| - 1$. If $v_{1,k}$ and $v_{2,\ell}$ are vertices satisfying $C_{k,\ell} = |z| - 1$, then $C_{i,j} = C_{k,\ell} + 1$. Note that such $v_{1,k}$ and $v_{2,\ell}$ always exist.
2. If $L_1(v_{1,i}) \neq L_2(v_{2,j})$, then trivially $suffix(L_1(P(v_{1,i}))) \cap suffix(L_2(P(v_{2,j}))) = \{\varepsilon\}$. Hence $C_{i,j} = 0$.

Consequently we obtain the following recurrence:

$$C_{i,j} = \begin{cases} 1+\max(\{C_{k,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\} \cup \{0\}) & \text{if } L_1(v_{1,i}) = L_2(v_{2,j}); \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

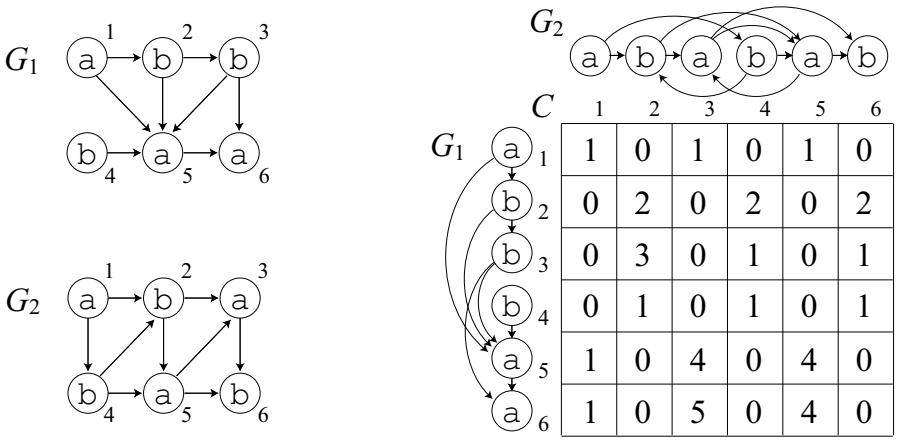


Figure 2. Example of dynamic programming for computing the length of a longest common substring of non-linear texts G_1 and G_2. Each vertex is annotated with its topological order. In this example, $\max C_{i,j} = 5$ and the longest common substring is `abbaa`.

We use dynamic programming to compute $C_{i,j}$ for all $1 \leq i \leq |V_1|$ and $1 \leq j \leq |V_2|$. Consider to compute $\max\{C_{k,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\}$. For each fixed $(v_{1,k}, v_{1,i}) \in E_1$, we refer the value of $C_{k,\ell}$ for all $1 \leq \ell < j$ such that $(v_{2,\ell}, v_{2,j}) \in V_2$, in $O(|E_2|)$ time. Therefore, the total time complexity for computing $\max\{C_{k,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\}$ is $O(|E_1||E_2|)$. Since we can sort vertices of G_1 and G_2 in topological ordering in linear time, the total time complexity is $O(|E_1||E_2|)$. The space complexity is clearly $O(|V_1||V_2|)$. □

An example of computing $C_{i,j}$ using dynamic programming is shown in Figure 2.

We remark that the recurrence of (3) is a natural generalization of that of (1) for computing the longest common substring of linear texts.

Furthermore, we can solve Problem 1 in case where *only one of the input non-linear texts is acyclic*:

Theorem 3. *If at least one of G_1 and G_2 is acyclic, then Problem 1 can be solved in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space.*

Proof. Assume w.l.o.g. that G_1 is acyclic. Recall the proof of Theorem 2. A key observation is that it indeed suffices to sort one of the input non-linear texts in topological ordering.

For any vertex $v_{2,j} \in V_2$ and positive integer h, let $P_h(v_{2,j})$ denote the set of paths of length not greater than h, which end at vertex $v_{2,j}$. Assume we have sorted vertices of G_1. Let $C_{i,j}$ denote the length of a longest string in $\mathit{suffix}(L_1(P(v_{1,i}))) \cap \mathit{suffix}(L_2(P_r(v_{2,j})))$, where r is the length of a longest path in $P(v_{1,i})$. We compute $C_{1,j}$ for each vertex $v_{2,j} \in V_2$ by: $C_{1,j} = 1$ if $L_1(v_{1,1}) = L_2(v_{2,j})$ and $C_{1,j} = 0$ otherwise. Then we compute $C_{i,j}$ for all $i > 1$ using the same recurrence as (3). Since the length of any element in $\mathit{substr}(G_1) \cap \mathit{substr}(G_2)$ is not greater than that of the longest path in G_1, $\max\{C_{i,j} \mid 1 \leq i \leq |V_1|, 1 \leq j \leq |V_2|\}$ equals to the length of a longest string in $\mathit{substr}(G_1) \cap \mathit{substr}(G_2)$. Consequently, G_2 does not have to be acyclic. □

A pseudo-code of our algorithm to solve the longest common substring problem for non-linear texts is shown in Algorithm 1.

Algorithm 1: Computing the length of longest common substring of non-linear texts.

Input: Acyclic non-linear text $G_1 = (V_1, E_1, L_1)$ and non-linear text $G_2 = (V_2, E_2, L_2)$.
Output: Length of a longest string in $substr(G_1) \cap substr(G_2)$.

```
1  topological sort G_1;
2  n ← |V_1|; m ← |V_2|;
3  Let C be an n × m integer array;
4  for i ← 1 to n do
5      for j ← 1 to m do
6          if f(v_{1,i}) = f(v_{2,j}) then
7              C_{i,j} ← 1;
8              forall v_{1,k} s.t. (v_{1,k}, v_{1,i}) ∈ E_1 do
9                  forall v_{2,ℓ} s.t. (v_{2,ℓ}, v_{2,j}) ∈ E_2 do
10                     if C_{i,j} < 1 + C_{k,ℓ} then
11                         C_{i,j} ← 1 + C_{k,ℓ};
12         else
13             C_{i,j} ← 0;
14 return max{C_{i,j} | 1 ≤ i ≤ n, 1 ≤ j ≤ m};
```

4 Computing Longest Common Subsequence Problem of Non-linear Texts

In this section, we tackle the problem of computing the length of longest common subsequence of two input non-linear texts. The problem is formalized as follows.

Problem 4 (Longest common subsequence problem for non-linear texts).

Input: Non-linear texts $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$.
Output: The length of a longest string in $subseq(G_1) \cap subseq(G_2)$.

For example, see the non-linear texts G_1 and G_2 of Figure 3. The solution to the above problem is 4, since there is a longest common subsequence `acdb` of G_1 and G_2.

In the sequel we present our algorithm to solve the above problem in case where both G_1 and G_2 are acyclic.

Theorem 5. *If G_1 and G_2 are acyclic, then Problem 4 can be solved in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space.*

Proof. Let $v_{1,i}$ and $v_{2,j}$ denote the i-th and j-th vertex in topological ordering in G_1 and in G_2, for $1 \leq i \leq |V_1|$ and $1 \leq j \leq |V_2|$, respectively. Let $C_{i,j}$ denote the length of a longest string in $subseq(L_1(P(v_{1,i}))) \cap subseq(L_2(P(v_{2,j})))$. $C_{i,j}$ can be calculated as follows.

1. If $L_1(v_{1,i}) = L_2(v_{2,j})$, there are two cases to consider:
   (a) If there are no arcs to $v_{1,i}$ or to $v_{2,j}$, i.e., $P(v_{1,i}) = \{v_{1,i}\}$ or $P(v_{2,j}) = \{v_{2,j}\}$, then clearly $C_{i,j} = 1$.
   (b) Otherwise, let $v_{1,k}$ and $v_{2,\ell}$ be any nodes s.t. $(v_{1,k}, v_{1,i}) \in E_1$ and $(v_{2,\ell}, v_{2,j}) \in E_2$, respectively. Let z be a longest string in $subseq(L_1(P(v_{1,i}))) \cap subseq(L_2(P(v_{2,j})))$. Assume on the contrary that there exists a string $y \in subseq(L_1(P(v_{1,k}))) \cap subseq(L_2(P(v_{2,\ell})))$ such that $|y| > |z| - 1$. This contradicts that z is a longest common subsequence of $L_1(P(v_{1,i}))$ and $L_2(P(v_{2,j}))$, since $L_1(v_{1,i}) = L_2(v_{2,j})$. Hence $|y| \leq |z| - 1$. If $v_{1,k}$ and $v_{2,\ell}$ are vertices satisfying $C_{k,\ell} = |z| - 1$, then $C_{i,j} = C_{k,\ell} + 1$. Note that such $v_{1,k}$ and $v_{2,\ell}$ always exist.

2. If $L_1(v_{1,i}) \neq L_2(v_{2,j})$, there are two cases to consider:
   (a) If there are no arcs to $v_{1,i}$ and to $v_{2,j}$, i.e.,$P(v_{1,i}) = \{v_{1,i}\}$ and $P(v_{2,j}) = \{v_{2,j}\}$, then clearly $C_{i,j} = 0$.
   (b) Otherwise, let $v_{1,k}$ and $v_{2,\ell}$ be any nodes s.t. $(v_{1,k}, v_{1,i}) \in E_1$ and $(v_{2,\ell}, v_{2,j}) \in E_2$, respectively. Let z be a longest string in $subseq(L_1(P(v_{1,i}))) \cap subseq(L_2(P(v_{2,j})))$. Assume on the contrary that there exists a string $y \in subseq(L_1(P(v_{1,k}))) \cap subseq(L_2(P(v_{2,j})))$ such that $|y| > |z|$. This contradicts that z is a longest common subsequence of $L_1(P(v_{1,i}))$ and $L_2(P(v_{2,j}))$, since $subseq(L_1(P(v_{1,k}))) \cap subseq(L_2(P(v_{2,j}))) \subseteq subseq(L_1(P(v_{1,i}))) \cap subseq(L_2(P(v_{2,j})))$. Hence $|y| \leq |z|$. If $v_{1,k}$ is a vertex satisfying $C_{k,j} = |z|$, then $C_{i,j} = C_{k,j}$. Similarly, if $v_{2,\ell}$ is a vertex satisfying $C_{i,\ell} = |z|$, then $C_{i,j} = C_{i,\ell}$. Note that such $v_{1,k}$ or $v_{2,\ell}$ always exists.

Consequently we obtain the following recurrence:

$$C_{i,j} = \begin{cases} 1+\max(\{C_{k,\ell} \,|\, (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\} \cup \{0\}) & \text{if } L_1(v_{1,i}) = L_2(v_{2,j}); \\ \max\left(\begin{array}{l}\{C_{k,j} \mid (v_{1,k}, v_{1,i}) \in E_1\} \cup \{C_{i,\ell} \,|\, (v_{2,\ell}, v_{2,j}) \in E_2\} \\ \cup \{0\}\end{array}\right) & \text{otherwise.} \end{cases} \quad (4)$$

We use dynamic programming to compute $C_{i,j}$ for all $1 \leq i \leq |V_1|$ and $1 \leq j \leq |V_2|$.

By similar arguments to the proof of Theorem 2, computing $\max\{C_{k,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\}$ takes $O(|E_1||E_2|)$ time.

Consider to compute $\max\{C_{k,j}, C_{i,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,k}, v_{2,j}) \in E_2\}$. For each fixed $(v_{1,k}, v_{1,i}) \in E_1$, we refer the value of $C_{k,j}$ for all $1 \leq j \leq |V_2|$ in $O(|V_2|)$ time. Similarly, for each fixed $(v_{2,\ell}, v_{2,j}) \in E_2$, we refer the value of $C_{i,\ell}$ for all $1 \leq i \leq |V_1|$ in $O(|V_1|)$ time. Therefore, the total time cost for computing $\max\{C_{k,j}, C_{i,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\}$ is $O(|V_2||E_1| + |V_1||E_2|)$.

Since we can sort vertices of G_1 and G_2 in topological ordering in linear time, the total time complexity is $O(|E_1||E_2|)$. The space complexity is clearly $O(|V_1||V_2|)$. □

An example of computing $C_{i,j}$ using dynamic programming is show in Figure 3. We remark that the recurrence of (4) is a natural generalization of that of (2) for computing the longest common subsequence of linear texts.

Algorithm 2 shows a pseudo-code of our algorithm to solve Problem 4 in case where both G_1 and G_2 are acyclic.

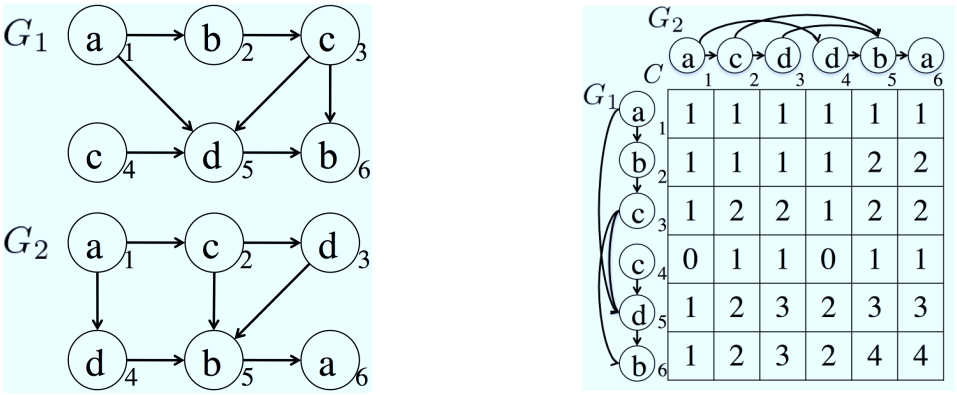


Figure 3. Example of dynamic programming for computing the length of a longest common subsequence of non-linear texts G_1 and G_2. Each vertex is annotated with its topological order. In this example, $\max C_{i,j} = 4$ and the longest common subsequence is `acdb`.

Algorithm 2: Computing the length of longest common subsequence of acyclic non-linear texts

Input: Two acyclic non-linear texts $G_1 = (V_1, E_1, L_1), G_2 = (V_2, E_2, L_2)$
Output: Length of a longest string in $subseq(G_1) \cap subseq(G_2)$

```
1  topological sort G1;
2  topological sort G2;
3  n ← |V1|; m ← |V2|;
4  Let C be an n × m integer array;
5  for i ← 1 to n do
6      for j ← 1 to m do
7          if f(v1,i) = f(v2,j) then
8              C_{i,j} ← 1;
9              forall v1,k s.t. (v1,k, v1,i) ∈ E1 do
10                 forall v2,ℓ s.t. (v2,ℓ, v2,j) ∈ E2 do
11                     if C_{i,j} < 1 + C_{k,ℓ} then
12                         C_{i,j} ← 1 + C_{k,ℓ};
13         else
14             C_{i,j} ← 0;
15             forall v1,k s.t. (v1,k, v1,i) ∈ E1 do
16                 if C_{i,j} < C_{k,j} then
17                     C_{i,j} ← C_{k,j};
18             forall v2,ℓ s.t. (v2,ℓ, v2,j) ∈ E2 do
19                 if C_{i,j} < C_{i,ℓ} then
20                     C_{i,j} ← C_{i,ℓ};
21 return max{C_{i,j} | 1 ≤ i ≤ n, 1 ≤ j ≤ m};
```

5 Computing Longest Common Subsequence of Cyclic Non-linear Texts

In this section, we present an algorithm to solve Problem 4 in case where the input non-linear texts are cyclic. We output ∞ if $subseq(G_1) \cap subseq(G_2)$ is infinite, and do the length of a longest string in $subseq(G_1) \cap subseq(G_2)$ otherwise.

We transform a cyclic non-linear text $G = (V, E, L)$ into an acyclic non-linear text $G' = (V', E', L')$ based on the strongly connected components. For each vertex $v \in V$, let $[v]$ denote the set of vertices that belong to the same strongly connected component. Formally, G' is defined as

$$V' = \{[v] \mid v \in V\},$$
$$E' = \{([v], [u]) \mid [v] \neq [u], (v', u') \in E \text{ for some } v' \in [v],\ u' \in [u]\} \cup \{(v, v) \mid |[v]| \geq 2\},$$

and $L'([v]) = \{L(v) \mid v \in [v]\} \subseteq \Sigma$. We regard each $[v]$ as a single vertex that is contracted from vertices in $[v]$. Observe that $subseq(G') = subseq(G)$.

An example of transformed acyclic non-linear texts is shown in Figure 4.

Theorem 6. *If G_1 and/or G_2 are cyclic, then Problem 4 can be solved in $O(|E_1||E_2| + |V_1||V_2| \log |\Sigma|)$ time and $O(|V_1||V_2|)$ space.*

Proof. We first transform cyclic non-linear texts G_1 and G_2 into corresponding acyclic non-linear texts G'_1 and G'_2, as described previously. Let $v'_{1,i}$ and $v'_{2,j}$ denote the i-th and j-th vertex in topological ordering in G'_1 and G'_2, for $1 \leq i \leq |V'_1|$ and $1 \leq j \leq |V'_2|$, respectively. Let S_1 and S_2 denote the sets of vertices which has a loop, namely, $S_1 = \{L'_1(v'_{1,i}) \mid (v'_{1,i}, v'_{1,i}) \in E'_1\}$ and $S_2 = \{L'_2(v'_{2,j}) \mid (v'_{2,j}, v'_{2,j}) \in E'_2\}$. If $S_1 \cap S_2 \neq \emptyset$, then let c be any character in $S_1 \cap S_2$. Clearly an infinite repetition c^* of c is a common subsequence of G_1 and G_2, and hence we output ∞.

In the sequel, consider the case where $S_1 \cap S_2 = \emptyset$. In this case, it is clear that $subseq(G_1) \cap subseq(G_2)$ is finite. Let $C_{i,j}$ denote the length of a longest string in $subseq(L'_1(P(v'_{1,i}))) \cap subseq(L'_2(P(v'_{2,j})))$. $C_{i,j}$ can be calculated as follows.

1. If $L'(v'_{1,i}) \cap L'(v'_{2,j}) \neq \emptyset$, there are two cases to consider:
   (a) If there are no arcs to $v'_{1,i}$ or to $v'_{2,j}$, i.e., $P(v'_{1,i}) = \{v'_{1,i}\}$ or $P(v'_{2,j}) = \{v'_{2,j}\}$, then clearly $C_{i,j} = 1$.
   (b) Otherwise, let $v'_{1,k}$ and $v'_{2,\ell}$ be any nodes s.t. $(v'_{1,k}, v'_{1,i}) \in E'_1$ and $(v'_{2,\ell}, v'_{2,j}) \in E'_2$, respectively. Let z be a longest string in $subseq(L'_1(P(v'_{1,i}))) \cap subseq(L'_2(P(v'_{2,j})))$. Assume on the contrary that there exists a string $y \in subseq(L'_1(P(v'_{1,k}))) \cap subseq(L'_2(P(v'_{2,\ell})))$ such that $|y| > |z| - 1$. This contradicts that z is a longest common subsequence of $L'_1(P(v'_{1,i}))$ and $L'_2(P(v'_{2,j}))$, since $L'_1(v'_{1,i}) \cap L'_2(v'_{2,j}) \neq \emptyset$. Hence $|y| \leq |z| - 1$. If $v'_{1,k}$ and $v'_{2,\ell}$ are vertices satisfying $C_{k,\ell} = |z| - 1$, then $C_{i,j} = C_{k,\ell} + 1$. Note that such $v'_{1,k}$ and $v'_{2,\ell}$ always exist.
2. If $L'(v'_{1,i}) \cap L'(v'_{2,j}) = \emptyset$, there are two cases to consider:
   (a) If there are no arcs to $v'_{1,i}$ and to $v'_{2,j}$, i.e., $P(v'_{1,i}) = \{v'_{1,i}\}$ and $P(v'_{2,j}) = \{v'_{2,j}\}$, then clearly $C_{i,j} = 0$.
   (b) Otherwise, let $v'_{1,k}$ and $v'_{2,\ell}$ be any nodes s.t. $(v'_{1,k}, v'_{1,i}) \in E'_1$ and $(v'_{2,\ell}, v'_{2,j}) \in E'_2$, respectively. Let z be a longest string in $subseq(L'_1(P(v'_{1,i}))) \cap subseq(L'_2(P(v'_{2,j})))$. Assume on the contrary that there exists a string $y \in subseq(L'_1(P(v'_{1,k}))) \cap subseq(L'_2(P(v'_{2,\ell})))$ such that $|y| > |z|$. This contradicts that z is a longest common subsequence of $L'_1(P(v'_{1,i}))$ and $L'_2(P(v'_{2,j}))$, since $subseq(L'_1(P(v'_{1,k}))) \cap$

$subseq(L'_2(P(v'_{2,j}))) \subseteq subseq(L'_1(P(v'_{1,i}))) \cap subseq(L'_2(P(v'_{2,j})))$. Hence $|y| \leq |z|$. If $v'_{1,k}$ is a vertex satisfying $C_{k,j} = |z|$, then $C_{i,j} = C_{k,j}$. Similarly, if $v'_{2,\ell}$ is a vertex satisfying $C_{i,\ell} = |z|$, then $C_{i,j} = C_{i,\ell}$. Note that such $v'_{1,k}$ $(k \neq i)$ or $v'_{2,\ell}$ $(\ell \neq j)$ always exists.

Consequently we obtain the following recurrence:

$$C_{i,j} = \begin{cases} 1+\max(\{C_{k,\ell} \mid (v'_{1,k}, v'_{1,i}) \in E'_1, (v'_{2,\ell}, v'_{2,j}) \in E'_2\} \cup \{0\}) & \text{If } L'(v'_{1,i}) \cap L'(v'_{2,j}) \neq \emptyset; \\ \max\left(\begin{array}{l}\{C_{k,j} \mid (v'_{1,k}, v'_{1,i}) \in E_1\} \cup \{C_{i,\ell} \mid (v'_{2,\ell}, v'_{2,j}) \in E_2\} \\ \cup \{0\}\end{array}\right) & \text{otherwise.} \end{cases} \quad (5)$$

It is well-known that we can transform G_1 and G_2 into G'_1 and G'_2 in linear time, based on strongly connected components.

For each self-loop such as $(v'_{1,i}, v'_{1,i}) \in E_1$, we refer the value of $C_{i,j}$ for all $1 \leq j \leq |V'_2|$ in $O(|V'_2|)$ time. Similarly, for each self-loop such as $(v'_{2,j}, v'_{2,j}) \in E_2$, we refer the value of $C_{i,j}$ for all $1 \leq i \leq |V'_1|$ in $O(|V'_1|)$ time. For the other arcs, we can compute $C_{i,j}$ for all $1 \leq i \leq |V'_1|$ and $1 \leq j \leq |V'_2|$ using dynamic programming in $O(|E'_1| \cdot |E'_2|)$ time, in a similar way as the previous section. Therefore the total time cost for computing $C_{i,j}$ is $O(|E'_1| \cdot |E'_2|)$.

Let Σ_1 and Σ_2 be the sets of characters that appear in G_1 and G_2, respectively. The time cost to compute $S_1 \cap S_2$ is $O(|\Sigma_1| \log |\Sigma_2| + |\Sigma_2| \log |\Sigma_1|)$ using a balanced tree. Assume $S_1 \cap S_2 = \emptyset$, and consider to compute $L'(v'_{1,i}) \cap L'(v'_{2,j})$. If $|L'(v'_{1,i})| > 1$ and $|L'(v'_{2,j})| > 1$, then we know $L'(v'_{1,i}) \cap L'(v'_{2,j}) = \emptyset$ since $S_1 \cap S_2 = \emptyset$. If $|L'(v'_{1,i})| = 1$ and/or $|L'(v'_{2,j})| = 1$, then $L'(v'_{1,i}) \cap L'(v'_{2,j})$ can be computed in $O(\log |\Sigma|)$ time using a balanced tree, where $|\Sigma| = \max\{|\Sigma_1|, |\Sigma_2|\}$. Therefore the total time cost to compare $L'(v'_{1,i})$ and $L'(v'_{2,j})$ for all $1 \leq i \leq |V'_1|$ and $1 \leq j \leq |V'_2|$ is $O(|V'_1||V'_2| \log |\Sigma|)$. The total time complexity becomes $O(|E_1|+|E_2|+|E'_1||E'_2|+|V'_1||V'_2| \log |\Sigma|+|\Sigma_1| \log |\Sigma_2|+|\Sigma_2| \log |\Sigma_1|) = O(|E_1||E_2| + |V_1||V_2| \log |\Sigma|)$, since $|\Sigma_1| \leq |V_1|$ and $|\Sigma_2| \leq |V_2|$. The total space complexity is $O(|V'_1||V'_2| + |\Sigma_1| \log |\Sigma_2| + |\Sigma_2| \log |\Sigma_1|) = O(|V_1||V_2|)$. □

An example of computing $C_{i,j}$ using dynamic programming is shown in Figure 4. A pseudo-code of our algorithm is shown in Algorithm 3.

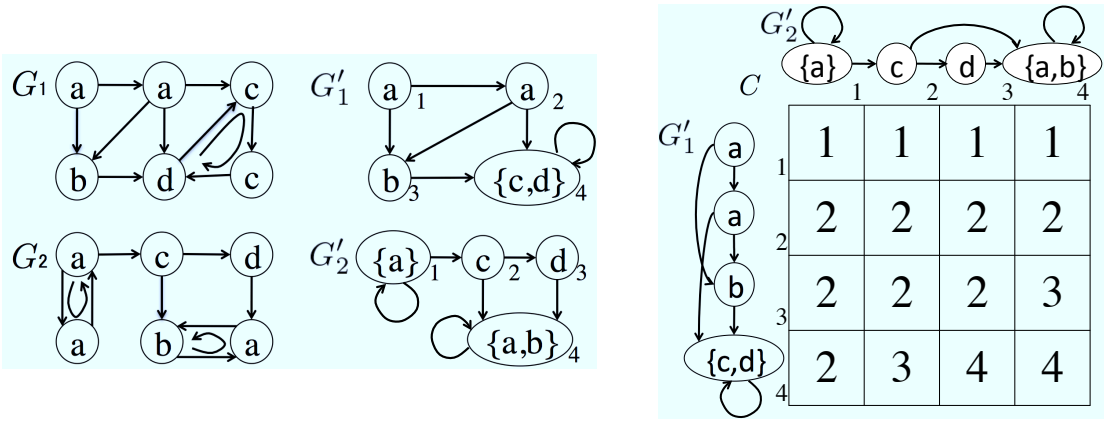


Figure 4. Example of dynamic programming for computing the length of a longest common subsequence of non-linear texts G_1 and G_2. G'_1 and G'_2 are non-linear texts which are transformed from G_1 and G_2 by grouping vertices into strongly connected components. Each vertex is annotated with its topological order. In this example, $\max C_{i,j} = 4$ and the longest common subseqence is `aacd`.

Algorithm 3: Computing the length of longest common subsequence of cyclic non-linear texts

Input: Two non-linear texts $G_1 = (V_1, E_1, L_1), G_2 = (V_2, E_2, L_2)$
Output: Length of a longest string in $subseq(G_1) \cap subseq(G_2)$

```
1  G'_1 ← Strongly Connected Components G_1;
2  G'_2 ← Strongly Connected Components G_2;
3  Let S_1 be a set of vertices which belong to cycles in G_1;
4  Let S_2 be a set of vertices which belong to cycles in G_2;
5  if S_1 ∩ S_2 ≠ ∅ then
6      return ∞ ;
7  else
8      topological sort G'_1;
9      topological sort G'_2;
10     Let C be an |V'_1| × |V'_2| integer array;
11     for i ← 1 to |V'_1| do
12         for j ← 1 to |V'_2| do
13             if (v'_{1,i}, v'_{1,i}) ∈ E'_1 then
14                 if (v'_{2,j}, v'_{2,j}) ∈ E'_2 then
15                     C_{i,j} ←Vertex-mismatch (v'_{1,i}, v'_{2,j});
16                 else if L(v'_{1,i}) ⊇ L(v'_{2,j}) then
17                     C_{i,j} ←Vertex-match (v'_{1,i}, v'_{2,j});
18                 else
19                     C_{i,j} ←Vertex-mismatch(v'_{1,i}, v'_{2,j});
20             else if (v'_{2,j}, v'_{2,j}) ∈ E'_2 then
21                 if L(v'_{1,i}) ⊆ L(v'_{2,j}) then
22                     C_{i,j} ←Vertex-match(v'_{1,i}, v'_{2,j});
23                 else
24                     C_{i,j} ←Vertex-mismatch(v'_{1,i}, v'_{2,j});
25             else if L(v'_{1,i}) = L(v'_{2,j}) then
26                 C_{i,j} ←Vertex-match(v'_{1,i}, v'_{2,j});
27             else
28                 C_{i,j} ←Vertex-mismatch(v'_{1,i}, v'_{2,j});
29 return max{C_{i,j} | 1 ≤ i ≤ |V'_1|, 1 ≤ j ≤ |V'_2|};
```

Algorithm 4: Vertex-match$(v_{1,i}, v_{2,j})$

```
1  C_{i,j} ← 1
2  forall v_{1,k} s.t. (v_{1,k}, v_{1,i}) ∈ E_1 do
3      forall v_{2,ℓ} s.t. (v_{2,ℓ}, v_{2,j}) ∈ E_2 do
4          if C_{i,j} < 1 + C_{k,ℓ} then
5              C_{i,j} ← 1 + C_{k,ℓ}
6  return C_{i,j}
```

Algorithm 5: Vertex-mismatch($v_{1,i}, v_{2,j}$)

```
C_{i,j} ← 0
forall v_{1,k} s.t. (v_{1,k}, v_{1,i}) ∈ E_1 do
    if C_{i,j} < C_{k,j} then
        C_{i,j} ← C_{k,j}
forall v_{2,ℓ} s.t. (v_{2,ℓ}, v_{2,j}) ∈ E_2 do
    if C_{i,j} < C_{i,ℓ} then
        C_{i,j} ← C_{i,ℓ}
return C_{i,j}
```

6 Conclusions

We considered the longest common substring and subsequence problems between two non-linear texts. We showed that when the texts are acyclic, the problem can be solved in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space by a dynamic programming approach. Furthermore, we extend our algorithm and consider the case where the texts can contain cycles, and presented an $O(|E_1||E_2| + |V_1||V_2| \log |\Sigma|)$ time and $O(|V_1||V_2|)$ space algorithm for the longest common subsequence problem. The longest common substring between general graphs is an open problem.

References

1. T. Akutsu: *A linear time pattern matching algorithm between a string and a tree*, in Proc. CPM'93, 1993, pp. 1–10.
2. A. Amir, M. Lewenstein, and N. Lewenstein: *Pattern matching in hypertext*, in Proc. WADS'97, vol. 1272 of LNCS, 1997, pp. 160–173.
3. U. Manber and S. Wu: *Approximate string matching with arbitrary costs for text and hypertext*, in Proc. IAPR, 1992, pp. 22–33.
4. M. Mohri: *Edit-distance of weighted automata: General definitions and algorithms.* Int. J. Found. Comput. Sci, 14(6) 2003, pp. 957–982.
5. G. Navarro: *Improved approximate pattern matching on hypertext.* Theoretial Computer Science, 237(1–2) 2000, pp. 455–463.
6. K. Park and D. K. Kim: *String matching in hypertext*, in Proc. CPM'95, 1995, pp. 318–329.
7. D. Q. Thang: *Algorithm to determine longest common subsequences of two finite languages*, in New Challenges for Intelligent Information and Database System, vol. 351/2011 of Studies in Computational Intelligence, 2011, pp. 3–12.

Algorithmics of Posets Generated by Words over Partially Commutative Alphabets*

Łukasz Mikulski, Marcin Piątkowski, and Sebastian Smyczyński

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Chopina 12/18, 87-100 Toruń, Poland
frodo@mat.umk.pl, martinp@mat.umk.pl, smyczek@mat.umk.pl

Abstract. It is natural to try to relate partially ordered sets (posets in short) and classes of equivalent words over partially commutative alphabets. Their common graphical representation are Hasse diagrams. We will investigate this relation in detail and propose an efficient on-line algorithm that decompresses a string to Hasse diagram. Further we propose a definition of the canonical representatives of classes of equivalent words. The advantage of this representation lies in the fact that we can enumerate the classes of equivalent words in a lexicographical order. We will give an algorithm which enumerates all distinct classes of words over partially commutative alphabets by their lexicographically minimal representatives.

Keywords: poset, Hasse diagram, partially commutative alphabets, algorithms, generations

Introduction

Many practical problems related to partially ordered sets have a very high time of computation. The examples of very hard tasks are #P-complete problem of counting number of posets linearisations [1] or NP-complete problem of computing the minimal number of jumps [10]. From less complex problems we can provide a problem of computing transitive reduction of a poset graph which has cubic time complexity.

One of the main reasons for such a situation is the dependence of the complexity exclusively on the number of elements of a poset. We show a stringologic approach to the posets that uses words over partially commutative alphabets and allows us to exploit the inner structure of a given poset. As a result, we achieve algorithms with complexity dependent not only on the number of elements but also on the size of the concurrent alphabet.

In the first section, we give some basic notions related to the formal languages theory, partial orders and the concurrent systems modeling. In Section 2 we will look more closely at the connections between words over semi-commutative alphabets, their dependency graphs and Hasse diagrams and graphs of partial orders. In the following section we will deal with decoding Hasse diagrams from strings and give an $O(nk^2)$ complexity algorithm. Here and subsequently n denotes the size of the poset and k – the size of the (possibly significantly smaller) alphabet.

The studies on the properties of words over partially commutative alphabets require an efficient tool for enumeration of distinct classes of equivalent words (in the sense of the independency relation). In the fourth section we deal with this practical

* The research partially supported by Ministry of Science and Higher Education of Poland, grant N N206 258035.

Łukasz Mikulski, Marcin Piątkowski, Sebastian Smyczyński: Algorithmics of Posets Generated by Words over Partially Commutative Alphabets, pp. 209–219.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

problem. The solution is motivated by a well known SEPA algorithm [4,5] and has similar complexity of a single step. Basically we will identify classes of equivalent words with their lexicographically smallest representatives. Those representatives are called *canonical*. Further we will show how to compute the considered representatives of all classes in the lexicographical order. The single step of this computation is based on the function which for a given class returns the next class in an order implied by the lexicographical order of their canonical representatives.

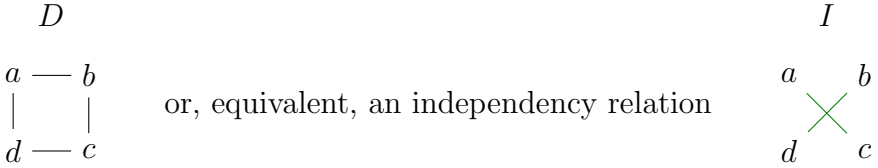
1 Basic notions

We use some basic notions of formal languages theory. By Σ we denote an arbitrary finite set, called *alphabet*. Elements of the alphabet are called *letters*. *Words* are sequences over the alphabet Σ. The sets of all finite words will be denoted by Σ^*.

The *concurrent alphabet* is a pair (Σ,D), where Σ is a common alphabet (finite set) and $D \subseteq \Sigma \times \Sigma$ is an arbitrary reflexive and symmetric relation, called *dependency relation*. With dependency we associate, as another relation, an *independency relation* $I = \Sigma \times \Sigma \setminus D$. Having the concurrent alphabet we define a relation that identifies similar words. We say that word $\sigma \in \Sigma^*$ is in relation $\equiv_D$ with word $\tau \in \Sigma^*$ if and only if there exists a finite sequence of commutation of subsequent, independent letters that leads from σ to τ. Relation $\equiv_D \subseteq \Sigma^* \times \Sigma^*$ is a congruence relation (whenever it will not be confusing, relation symbol D will be omitted).

After dividing set Σ^* by the relation $\equiv$ we get a quotient monoid. The elements of $\Sigma^*/_{\equiv}$ are often called *traces* (see [3,8,9]). This way, every word σ is related to a trace $\alpha = [\sigma]$, containing this word.

Example 1. To the alphabet $\Sigma = \{a, b, c, d\}$ we add a dependency relation

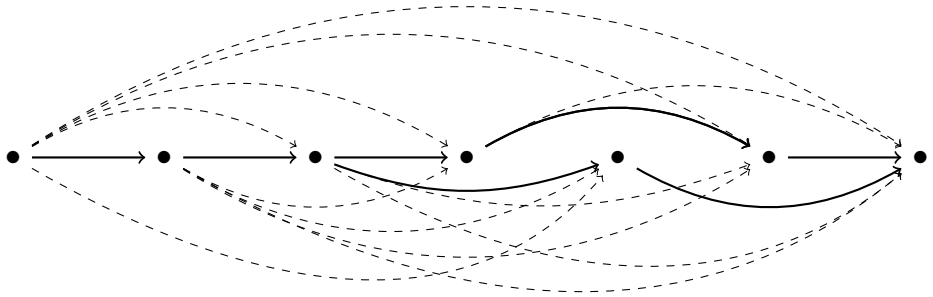


Words *abbaacd* and *abbcaad* are equivalent.

The *partial order* in a set X is a relation $\leq \subseteq X \times X$, such that $\leq$ is reflexive, antisymmetric and transitive. Pair $(X,\leq)$ is called *partially ordered set*, or shortly *poset*. With every poset we can associate its directed graph (digraph in short) $G = (X, E)$. The vertices are the elements of the poset. There is an edge between two vertices $x, y \in X$ if and only if $x < y$ (ie. $x \leq y$ but $x \neq y$). Such a graph is always acyclic. We can also define a Hasse diagram of poset $(X,\leq)$ by the transitive reduction of graph G.

Definition 2. *Let $G = (X, E)$ be an acyclic graph. The* Hasse diagram *of graph G i acyclic graph $H = (X, E' \subseteq E)$, such that an edge $(x, y) \in E'$ if and only if $(x, y) \in E$ and if there is $z \in X$ such that there are both path from x to z and from z to y then $x = z$ or $y = z$.*

Example 3. The graph of a poset. The dashed edges are not contained in Hasse diagram.



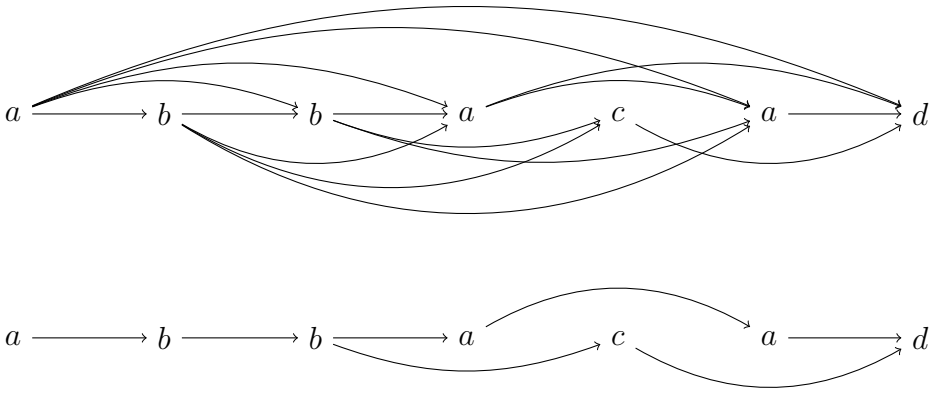
2 From partially commutative words to posets

With every word w over partially commutative alphabet (Σ, D) we can associate a poset. The preorder of this poset is induced by the dependency graph of a word w. A letter w_j is greater than a letter w_i if and only if $i < j$ and $w_i D w_j$. It is worth noting that two words are equivalent if and only if their dependency graphs are the same (isomorphic and respecting labelling).

Reflexive transitive closure of dependency graph of a word is basically a graph of a poset associated with the word. We can represent it by the graph of its transitive reduction, called Hasse diagram.

Example 4. A concurrent alphabet (Σ, D), dependency graph and Hasse diagram of word *abbacad* over that alphabet.

$$\Sigma = \{\, a, b, c, d\} \qquad D = \begin{array}{ccc} a & - & b \\ | & & | \\ d & - & c \end{array}$$



Lemma 5. *Every poset $(P, \leq)$ can be generated by a word over concurrent alphabet.*

Proof. For given poset $(P, \leq)$ let us define a concurrent alphabet (Σ, D) in such a way that $\Sigma = P$ and $p_1 D p_2$ if and only if $p_1 \leq p_2$ or $p_2 \leq p_1$. An abitrary linearisation of poset $(P, \leq)$ corresponds in natural way with a word $v \in \Sigma^*$ which generates a poset equal to $(P, \leq)$. □

The above observations allow us to represent every poset in a compressed way by a pair consisting of concurrent alphabet and a single word over that alphabet. In the next section we will provide an efficient algorithm that produces a Hasse diagram by decompressing a given word to its associated poset.

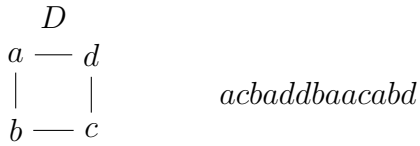
Further optimisation, possible only for Hasse diagrams which are minimal series-parallel graphs [14], lead us to another data structure which can be used to solve many problems in a simpler way (for instance #P-complete problem of counting number of linear expansions [1] is linear for such posets).

Definition 6. *Minimal Series-Parallel digraph (MSP) is a graph consisting of a single vertex and no edges or is constructed from two MSP – $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ – by the following operations:*

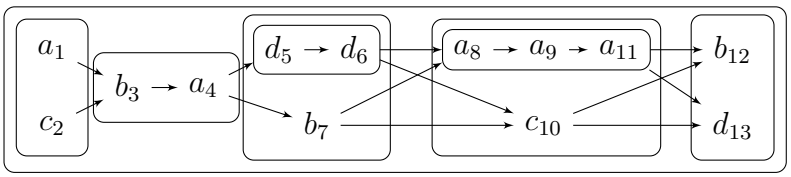
- *Parallel composition: $G_P = (V_1 \cup V_2, E_1 \cup E_2)$;*
- *Serial composition: $G_P = (V_1 \cup V_2, E_1 \cup E_2 \cup T_1 \times S_2)$;*

where T_1 is the set of sinks of G_1 and S_2 is a set of sources of G_2. In other words, series-parallel graphs can be represented as an expression built by series and parallel composition of graphs with single-vertex graphs as atoms.

Example 7. The dependent alphabet D, the word w and its Hasse diagram divided to series-parallel blocks.



$acbaddbaacabd$

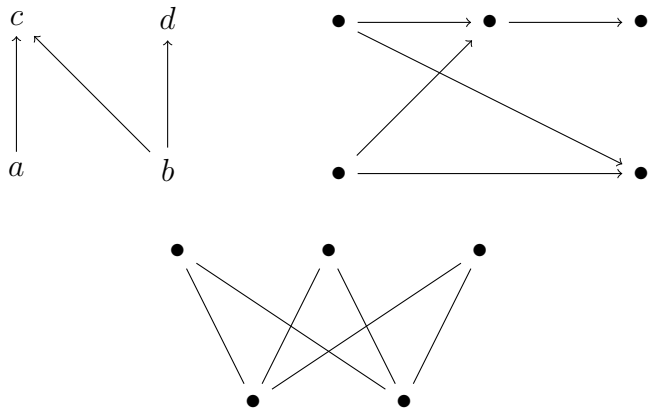


The properties of series-parallel graphs are deeply studied (see for instance [2,11,14]). A very useful determinant for sequential parallel graphs is their N-freeness [13].

Definition 8. *N-poset is a poset consisting of four elements a, b, c, d with relations $a < c, b < c$ and $b < d$ (drawing a graph of such poset with greater elements higher brings to mind capital letter N).[7]*

Definition 9. *N-free posets are posets whose Hasse diagrams do not contain an induced subgraph isomorphic with Hasse diagram of N-poset.*
In the case of undirected graphs, analogue is P4-free graph (a graph that does not contain induced path of length 3).

Example 10. N-poset, simple N-free poset and P4-free graph



In general, this type of graphs, also in context of partial orders, is deeply studied (see [6,12,14] and the references therein). However, observations worth mentioning are the following:

Lemma 11. *If a dependency graph D of an alphabet Σ is $P4-free$ then Hasse diagram of every partially commutative word $w \in \Sigma^*$ is $N-free$.*

The above discution gives another motivation for studies over efficient algorithms that constructs Hasse diagram.

3 Construction of Hasse diagram

This section is devoted to the problem of constructing the Hasse diagram (see Definition 2) for a given word over a concurrent alphabet (which is a transitive reduction of some poset). At the beginning we give an algorithm and its pseudo-code. After that, we discuss the complexity of our solution.

The algorithm exploits the knowledge of the structure of resulting diagram. We can summarize it in the following facts:

Lemma 12. *Let $w \in \Sigma^*$ be a word and $H(w) = (V, E_H)$ be a Hasse diagram of w. If there exists the edge connecting vertices labeled $w_i = a$ and $w_j = b$ then letters a and b do not appear in word w between indexes i and j.*

Proof. Let us denote the dependence digraph $G = (V, E)$ of a word w over concuurent alphabet (Σ, D). The existance of an edge between w_i and w_j in graph H implies that there is also an edge in graph G, hence letters a and b are dependent (formally aDb). Let us suppose that there exists a letter $w_k = c$ (for $i < k < j$) that is dependent both with a and b. Then by the Definition 2 there is a path in graph G between vertices w_i and w_j of length longer than one, so there is no edge between w_i and w_j in graph $H(w)$, which provides to contradiction and completes the proof. □

Lemma 13. *Let $w \in \Sigma^*$ be a word and $H(w)$ be a Hasse diagram of w. For each vertex there are no more than $k = |\Sigma|$ outgoing edges and no more than k ingoing edges.*

Proof. Let us denote the dependence digraph $G = (V, E)$ of a word w over concuurent alphabet (Σ, D). Let us suppose that there is a vertex w_i which has $k + 1$ outgoing edges. There are k letters in alphabet Σ, so two of these outgoing edges lead to two distinct vertices w_j and w_k $(i < j < k)$ labelled with the same letter. Withaout loss of genrality we can assume that $w_i = a$ and $w_j = w_k = b$. From the Lemma 12 there is no edge in graph $H(w)$ between vertices w_i and w_k, which proves that there are at most k outgoing edges. Similar reasonig allows us to prove second part of thesis and limit the number of ingoing edges. □

Lemma 14. *Let $w \in \Sigma^*$ be a word and $H(w) = (V, E)$ be a Hasse diagram of w. Ingoing edges of the given vertex v are fully determined by the last occurrences of the letters dependent with $v = w_i$. More formally, $(w_j, w_i = v) \in E$ if and only if $j < i$ and there is no vertex $w_k D w_i$ such that $j < k < i$ and there is a path from w_j to w_k.*

Proof. Let $(w_j = a, w_i)$ be an edge in $H(w)$. Lemma 12 implies that w_j must be the last occurrence of letter a in word w that preceedes w_i. Second part of the thesis follows direclty from the Definition 2 (see proof of the Lemma 12). □

Using foregoing observations we propose an additional structure that saves information about last occurrences of each letter processed so far. It allows us to immediately add new vertex to Hasse diagram, with all of its ingoing edges. Our structure consists of a list of dependency, a set of visibility (both of size at most k) and a pointer to last occurrence, for each letter of the alphabet Σ. The list D_a contains all letters dependent with a in LIFO (last in – first out) order of their last occurrence in the currently constructed part of the diagram. Set V_a contains all letters b whose last occurrences are visible from the last vertex labeled with a. In other words, there exists a path from $v_i = b$ to $v_j = a$ where v_i and v_j represent the last occurrences of those letters in hitherto diagram. Such elements v_i will be called sources of v_j. The last element is a pointer L_a which is basically a pointer to the last vertex labeled with the letter a in hitherto diagram. We will also use a temporary set V.

Before we start generating a Hasse diagram we set all pointers to *null* and all sets to be empty. The lists of dependences should be complete with all dependent letters, but the initial order does not matter. With such data we are ready to process a new letter a of a word w in on-line manner, updating the proposed structure after each step and creating a new vertex and edges. During adding the new vertex labeled with the letter a we clear set V and browse the list D_a. For each letter b from that list we check if the pointer L_b is not empty and if b does not belong to V (is not already visible from new vertex). If we succeed, we add a new edge from the vertex pointed by L_b to the newly created vertex. Addition of a new edge implies that there is also a path from every source of b to the recently created vertex. Therefore we add set V_b to our temporary set V. It is worth noting that the order of processing letters form list D_a is important because of dynamically changing set V.

After adding new edges, we have to update our structure. Firstly, we remove the letter a from each set V_b – the new vertex is now the last occurrence of letter a so it can not be a source at all. Next we switch the position of the letter a in every list D_b – the letter a is the most recent letter now. The last operation is the update of the set V_a to $V \cup a$ and pointer L_a to the position of the new vertex.

Algorithm 1: Hasse diagram

```
1  foreach a ∈ Σ do
2      L_a := 0; V_a := ∅;
3  for i := 1 to n do
4      a := w_i; V := ∅;
5      foreach b ∈ D_a in order of the last occurrence do
6          if L_b ≠ 0 and b ∉ V then
7              Insert an edge w_{L_b} → w_i;
8              V := V ∪ V_b;
9      foreach b ∈ Σ do
10         V_b := V_b/{a};
11     foreach b ∈ D_a do
12         Move a to the beginning D_b;
13     V_a := V ∪ a; L_a := i;
```

The correctness of the algorithm presented above bases on lemmas formulated at the beginning of this section. Let us discuss the memory and time complexity of our solution. The proposed data structure consists of k lists D of at most k items. It gives us k^2 elements. The k sets V can be implemented using $k \log k$ memory, we also need k pointers L. Summing up, the most significant part of this data structure is a set of list and the memory complexity is $O(k^2)$.

The presented algorithm is on-line, which gives a linear factor in time complexity. Let us analyze a single step of extending the diagram with a new vertex (processing a new letter). We can see there a sequence of three loops. The first one is the most significant. We have to compute at most k sums of subsets of set Σ. It gives us a factor k^2. The operation in the second loop (line 10) can be done in constant time. Furthermore, the operation in last loop (line 12) has logarithmic time complexity if we make use of priority queue but can be implemented in constant time. Summarizing, we have a complexity of $O(k^2)$ for each step of algorithm that in total gives $O(nk^2)$ time complexity for processing the whole word.

4 Generation of all disjoint traces

The problem with the compressed presentation of a poset discussed in the previous sections is that it is not unique. For a given ordered concurrent alphabet $(\Sigma = \{a_1 < a_2 < \cdots < a_k\}, D)$ and a word w, every other word v equivalent with w represents the same poset. To overcome this disadvantage we can use the notion of *canonical representative*. Basically, from all the representatives we choose the lexicographically minimal one as a canonical representative. All words that are canonical representatives of a trace are called *canonical words*. Obviously, all the words of the length not greater then one are canonical. The natural problem that arises, is to enumerate all nonequivalent words (in fact canonical representatives of traces) of length n for a given concurrent alphabet. In this section we deal with this problem.

The proposed algorithm is motivated by the well known SEPA algorithm. We identify and modify only the *working suffix* – the suffix of the given canonical word which makes it different from its successor in the lexicographic order. We begin enumeration with lexicographically minimal word $w = a_1 a_1 \cdots a_1$. Then, we consecutively modify

the current word to its successor in lexicographic order, skipping all noncanonical ones. To perform this enumeration effectively we will use the following facts:

Lemma 15. *If wv is a canonical word then both words w and v also are canonical. In other words prefixes and suffixes of canonical words are canonical.*

Proof. Let us suppose that word w is not canonical. Then there is a lexicographical smaller and equivalent with respect to the relation $\equiv_D$ word w'. From the equivalence of words w and w' we conclude that also words wv and $w'v$ are equivalent. The word $w'v$ is lexicographically smaller then word wv. Therefore the word wv cannot be canonical. Simlar argumentation shows that v is also a canonical word. □

Lemma 16. *In every canonical word w if there exists i such that letters w_i and w_{i+1} are independent then $w_i < w_{i+1}$.*

Proof. Suppose that $w_i \geq w_{i+1}$. If they are equal then by the definition they are dependent which contradicts the assumption of their independence, hence $w_i > w_{i+1}$. We can assume that $w = uw_iw_{i+1}v$ which is equivalent with respect to the relation $\equiv_D$ to the word $uw_{i+1}w_iv$ that is lexicographically smaller than w. That is in conflict with the assumption of cononicality of the word w and completes the proof. □

Lemma 17. *If there exists a substring $w_iw_{i+1}\cdots w_{j-1}w_j$ of canonical word w such that letter w_j is independent with all letters $w_i, w_{i+1}, \ldots, w_{j-1}$ then w_j is the maximal amongst these letters. More precisely,*

$$\forall_{l\in\{i,i+1,\ldots,j-1\}} w_j > w_l.$$

Proof. Let us denote the word $w = uw_i\cdots w_jv$ and suppose that one of the letters $w_k \in \{w_i, w_{i+1}, \ldots, w_{j-1}\}$ is smaller then w_j. Then, from the independence of each of these letters with w_j we have an equivalent with respect to the relation $\equiv_D$ to w word $w' = uw_i\cdots w_{k-1}w_jw_k\cdots w_{j-1}$. Obviously the word w' is lexicographically smaller then w, hence w cannot be a canonical word. □

Lemma 18. *If there exists a substring $w_iw_{i+1}\cdots w_{j-1}w_j$ of canonical word w such that letter w_j is independent with all letters $w_{i+1}, \ldots, w_{j-1}, w_j$ then w_i is the minimal amongst these letters. More precisely,*

$$\forall_{l\in\{i+1,\ldots,j-1,j\}} w_i < w_l.$$

Proof. Proof of the lemma is similar to the proof of Lemma 17. □

Definition 19. *Let $a \in \Sigma$ be a letter. By $\mathbf{C}_a^n$ we denote the set of all canonical words of length n which start with the letter a.*

It is an easy observation that the set $\mathbf{C}_a^n$ is nonempty. It contains at least the word a^n. Moreover, $\mathbf{C}_a^1 = \{a\}$.

Lemma 20. *Let $w_1 \in \Sigma$ be an arbitrary but fixed letter and $w = w_1w_2\cdots w_n$ be the lexicographically smallest word from $\mathbf{C}_{w_1}^n$ (for $n > 1$). Then the letter w_2 is the smallest letter dependent with the letter w_1 and the word $w_2\cdots w_n$ is the lexicographically smallest word from $\mathbf{C}_{w_2}^{n-1}$. Moreover, the sequence of letters $w_1, w_2, \ldots, w_n$ is nonencreasing and every two consecutive letters from this sequence are dependent.*

Proof. We give the proof by induction on the length n.
Let $w \in \mathbf{C}^2_{w_1}$. Then w is of the form w_1w_2, where w_2 is dependent with w_1 or strictly greater then w_1. Therefore, the smallest element of $\mathbf{C}^2_{w_1}$ is the word w_1w_2, where w_2 is the smallest letter dependent with w_1 (maybe w_1 itself). Other parts of the thesis are clearly satisfied.

Let us suppose that the thesis holds for all letters and lengths n smaller than k. We prove the case of letter w_1 and length k. Let us suppose, that word $w = w_1w_2 \cdots w_k$ is the lexicographically smallest word from $\mathbf{C}^k_{w_1}$. Then the letter w_2 is (similary to the case of length 2) dependent to w_1 and not greater then w_1. Moreover, from Lemma 15 the word $w_2 \cdots w_k$ is canonical. If it would not be the smallest word from the set $\mathbf{C}^{k-1}_{w_2}$, we could change it to the word of such property achieving better candidate for minimum, and the proof is complete. □

The foregoing lemmas provide us enough information of the structure of the canonical words to design the algorithm transforming given canonical word w into its successor. The algorithm consists of three steps:

1. Finding the last index i such that $w_i \neq a_k$. We know that index i is the starting position of the working suffix.
2. Computing the minimal letter a greater than w_i such that $w_1w_2 \cdots w_{i-1}a$ is canonical. It is implied by Lemma 15.
3. Generating the rest of the working suffix to obtain the minimal canonical word that starts from w_i.

To implement the second step we will introduce the oracle V. For every position i and every letter a the $V_i(a)$ answers to the question – is there a substring $w_jw_{j+1} \cdots w_{i-1}$ such that all letters $w_j, w_{j+1}, \ldots, w_{i-1}$ are independent from w_i and at least one letter from this substring is greater than w_i? In case of positive answer $V_i(a)$ gives the maximal letter from the longest of such substrings as a witness, otherwise it simply returns a. Such oracle can be constructed in linear time (with respect to n) using the following formula:

$$V_1(a) = a$$
$$V_i(a) = \begin{cases} a & : aDw_{i-1} \\ \max(a, V_{i-1}(a)) & : \text{otherwise} \end{cases}$$

For every letter a such that $V_i(a) = a$, the string $w_1w_2 \cdots w_{i-1}a$ is canonical.

For the efficient generation of the working suffix in step three we will use a precomputed table $D_{\min}$ such that

$$\forall_{a \in \Sigma}\ D_{\min}(a) = \min\{b \in \Sigma : aDb\}.$$

After generating a new canonical word we have to update the oracle V. The value of $V_j(a)$ depends only on $V_{j-1}(a)$ and letter w_{j-1}. Therefore, we only have to update oracle from V_{i+1} to V_n (for the whole working suffix). Moreover, if there exists such index l in the working suffix that $w_l = w_{l+1}$, then the rest of the suffix is constant (all letters equal to w_l) and computation of missing oracle values are trivial ($V_{l+2} = V_{l+3} = \cdots = V_n = V_{l+1}$).

Algorithm 2: Enumerate Canonical Words

```
1  w := a_1 a_1 ··· a_1;
2  OUTPUT w;
3  for i := 1 to n do
4      Update Oracle V_i;
5  repeat
6      i := last index such that w_i ≠ a_k;
7      repeat
8          w_i := succ(w_i);
9      until V_i(w_i) = w_i;
10     for j := i + 1 to n do
11         w_j := D_min(w_{j-1}); // Generate suffix
12     for j := i + 1 to n do
13         Update Oracle V_j;
14     OUTPUT w ;
15 until w = a_k a_k ··· a_k;
```

Algorithm 3: Update Oracle V_i

```
1  if i = 1 then
2      foreach a ∈ Σ do V_1(a) := a;
3  else if i > 2 and w_{i-2} = w_{i-1} then
4      V_i := V_{i-1};
5  else
6      foreach a ∈ Σ do
7          if aDw_{i-1} then
8              V_i(a) := a;
9          else
10             V_i(a) := max(w_{i-1}, V_{i-1}(a));
```

The observations mentioned above lead us to the Algorithms 2 and 3. Let us discuss the memory and time complexity. The used memory is obviously $O(nk)$, mostly used for oracle V. The time complexity of steps needed for generating the next canonical word depends on the length $\#SUFF$ of the working suffix (lines from 6 to 13 of Algorithm 2). The line 6 is linear with respect to $\#SUFF$. Loop in lines 7–9 perform at most k iterations. The next loop (lines 10 – 11), which generates a suffix, makes exactly $\#SUFF$ operations. The most complex work is done in the last loop, which updates the oracle. At most k times the execution of the procedure Update Oracle is nontrivial and computes whole V_i. The rest of computation (at maximum $\#SUFF$ times) will end up at line 4 of the Update Oracle procedure, which can by simply implemented as a reference copying. It gives $O(k^2 + \#SUFF)$ complexity of the last loop.

If we set k as a constant enlarging only n, the time complexity of the single step of successor generation is $O(\#SUFF)$ and is therefore optimal. Nevertheless, it would be very interesting to investigate the case when k is close to n. Probably this case needs another kind of optimization and new algorithms.

5 Summary and future work

In the paper we have discussed an approach to encode posets by strings. We used concurrent alphabets and well known notion of Hasse diagram, which is significantly smaller than the graph of a poset. We have shown that every poset can be represented by a pair consisting of a concurrent alphabet and a word over this alphabet. However, it is very interesting how to choose the best pair. The first criterion is the size of the concurrent alphabet (the one from the proof is taken in a very inefficient way). The second important property is preservation of N-freeness by achieving the P4-free dependency relation graph.

In the third section we gave an efficient algorithm that enables us to decompress a word into a Hasse diagram. Extending those results, we would like to equip Hasse diagrams (using additional data structures) with efficient concatenation and star operations.

Section four is devoted to the algorithm which enumerates all nonequivalent strings (in the sense of dependency relation). The main idea is motivated by SEPA algorithm. However, the innovative idea of using oracle allows us to construct an algorithm that is optimal (for constant size k of the alphabet) with respect to performed changes. The case of k close to n needs further work and new algorithms.

We believe that our results will have many theoretical and practical applications. For example, the extending of the results related to Hasse diagram may be very useful in verification of models, where partial orders or concurrent words play a key role.

References

1. G. Brightwell and P. Winkler: *Counting linear extensions is #P-complete*, in STOC: ACM Symposium on Theory of Computing (STOC), 1991.
2. O. Cogis and M. Habib: *Nombre de sauts et graphes série-parallèles.* ITA, 13(1) 1979.
3. V. Diekert and G. Rozenberg, eds., *The Book of Traces*, World Scientific, Singapore, 1995.
4. D. E. Knuth: *The Art of Computer Programming, Volume 3*, Addison-Wesley, Reading, 1973.
5. D. E. Knuth: *The Art of Computer Programming: Volume 4, Fascicle 3. Generating All Combinations and Partitions*, Addison-Wesley, 2005.
6. D. Kuske: *Infinite series-parallel posets: Logic and languages.* Lecture Notes in Computer Science, 1853 2000, pp. 648–662.
7. A. B. Kwiatkowska and M. M. Sysło: *On page number of n-free posets.* Electronic Notes in Discrete Mathematics, 24 2006, pp. 243 – 249, Fifth Cracow Conference on Graph Theory USTRON '06.
8. A. Mazurkiewicz: *Concurrent program schemes and their interpretations*, daimi report pb-78, Aarhus University, 1977.
9. Ł. Mikulski: *Projection representation of Mazurkiewicz traces.* Fundamenta Informaticae, 85 2008, pp. 399–408.
10. W. R. Pulleyblank: *On minimizing setups in precedence constrained scheduling*, Tech. Rep. 81185-OR, Univ. Bonn, Inst. f. Ökonometrie und OR, Bonn, 1981.
11. M. M. Sysło: *Minimizing the jump number for partially ordered sets: A graph-theoretic approach.* Order, 1 1984, pp. 7–19.
12. K. Takamizawa, T. Nishizeki, and N. Saito: *Linear-time computability of combinatorial problems on series-parallel graphs.* Journal of the ACM, 29(3) 1982, pp. 623–641.
13. J. Valdes: *Parsing Flowcharts and Series-Parallel Graphs*, Ph.D. dissertation, Stanford University, Stanford, 1978.
14. J. Valdes, R. E. Tarjan, and E. L. Lawler: *The recognition of series parallel digraphs*, in Eleventh Annual ACM Symposium on Theory of Computing (STOC '79), New York, Apr. 1979, ACM, pp. 1–12.

Binary Image Compression via Monochromatic Pattern Substitution: A Sequential Speed-Up

Luigi Cinque[1], Sergio De Agostino[1], and Luca Lombardi[2]

[1] Computer Science Department, Sapienza University, 00198 Rome, Italy
deagostino@di.uniroma1.it
[2] Computer Science Department, University of Pavia, 27100 Pavia, Italy

Abstract. A method for compressing binary images is monochromatic pattern substitution. Monochromatic rectangles inside the image are detected and compressed by a variable length code. Such method has no relevant loss of compression effectiveness if the image is partitioned into up to a thousand blocks and each block is compressed independently. Therefore, it can be implemented in parallel on both small and large scale arrays of processors with distributed memory and no interconnections. We show in this paper that such method has a speed-up if applied sequentially to the partitioned image. Experimental results show that the speed-up happens if the image is partitioned into up to 256 blocks and sequentially each block is compressed independently. It follows that the sequential speed-up can also be applied to a parallel implementation on a small scale system.

Keywords: lossless compression, binary image, sequential algorithm, parallel computing, distributed system

1 Introduction

A low-complexity binary image compressor has been designed in [3], which employs monochromatic pattern substitution and is implementable on small and large scale parallel systems. When it comes to parallel implementations, we wish to remark that parallel models have two types of complexity, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any parallel algorithm and has standard solutions, the communication cost of the computational phase after the distribution of the data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation to design a practical algorithm. The simplest model for this phase is, of course, a simple array of processors with no interconnections and, therefore, no communication cost. Compression via monochromatic pattern substitution has no relevant loss of effectiveness if the image is partitioned into up to a thousand blocks and each block is compressed independently. Therefore, it can be implemented in parallel on both small and large scale arrays of processors with distributed memory and no interconnections.
Another low-complexity compressor for binary images is BLOCK MATCHING [6], [7], which extends data compression via textual substitution to two-dimensional data by compressing sub-images rather than substrings [5], [8]. However, it does not work locally since it applies a generalized LZ1-type method with an unrestricted window and it is not scalable [2], [4].
In this paper, we show that monochromatic pattern substitution has a speed-up if

Luigi Cinque, Sergio De Agostino, Luca Lombardi: Binary Image Compression via Monochromatic Pattern Substitution: A Sequential Speed-Up, pp. 220–225.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

applied sequentially to the partitioned image. Experimental results show that the speed-up happens if the image is partitioned into up to 256 blocks and sequentially each block is compressed independently. It follows that the sequential speed-up can also be applied to a parallel implementation on a small scale system. Such speed-up depends on the fact that monochromatic rectangles crossing boundaries between blocks are not computed. We refine the partition by splitting the blocks horizontally and vertically and, after four refinements, experimentations show that no further improvement is obtained.
Compression via monochromatic pattern substitution is described in section 2. Section 3 presents the experimental results of the sequential speed-up. The experimental results of the sequential speed-up applied to small scale parallel computation are shown in section 4. Conclusions and future work are given in section 5.

Image	1 block	4 blocks	16 blocks	64 blocks	256 blocks
ccitt1	87.84	76.67	50.32	39.30	32.97
ccitt2	105.48	88.48	66.05	46.90	33.98
ccitt3	73.99	67.02	55.90	47.10	40.77
ccitt4	68.41	64.58	59.10	55.28	50.18
ccitt5	77.59	72.92	60.55	51.85	38.94
ccitt6	69.63	61.57	52.08	40.91	36.24
ccitt7	69.46	66.02	63.27	58.91	52.90
ccitt8	77.74	73.84	61.44	50.12	42.51

Figure 1. Sequential compression times on the CCITT images (ms.)

Image	1 block	4 blocks	16 blocks	64 blocks	256 blocks
ccitt1	43.67	38.75	26.87	20.95	17.85
ccitt2	49.21	41.61	32.10	23.47	17.76
ccitt3	38.46	35.44	31.31	26.83	23.31
ccitt4	38.11	36.46	34.73	33.02	30.35
ccitt5	40.23	37.83	33.11	26.79	22.65
ccitt6	36.26	32.79	28.56	22.85	20.33
ccitt7	37.59	35.97	35.01	33.26	30.46
ccitt8	38.39	36.74	31.79	26.42	22.82

Figure 2. Sequential decompression times on the CCITT images (ms.)

2 Monochromatic Pattern Substitution

Monochromatic rectangles inside the image are compressed by a variable length code. Such monochromatic rectangles are detected by means of a *raster* scan (row by row). If the 4×4 subarray in position (i, j) of the image is monochromatic, then we compute the largest monochromatic rectangle in that position else we leave it uncompressed. The encoding scheme for such rectangles uses a flag field indicating whether there is a monochromatic match (0 for the white ones and 10 for the black ones) or not (11). If the flag field is 11, it is followed by the sixteen bits of the 4×4 subarray (raw data).

Image	4 blocks	16 blocks	64 blocks	256 blocks
ccitt1	24.99	22.65	18.55	13.25
ccitt2	43.07	29.85	20.18	17.77
ccitt3	22.84	22.69	17.16	14.73
ccitt4	31.77	19.61	19.65	18.33
ccitt5	26.03	23.20	17.24	13.48
ccitt6	21.37	22.53	15.35	12.81
ccitt7	32.82	25.34	20.62	18.03
ccitt8	26.76	27.96	19.44	14.61

Figure 3. Parallel compression times on the CCITT images (ms.)

Image	4 blocks	16 blocks	64 blocks	256 blocks
ccitt1	13.10	9.56	7.02	5.83
ccitt2	18.56	13.07	8.78	5.69
ccitt3	12.97	9.02	8.29	6.61
ccitt4	20.86	11.02	9.35	8.71
ccitt5	15.42	10.16	8.10	6.67
ccitt6	11.43	9.13	6.99	6.12
ccitt7	21.51	10.72	9.40	8.59
ccitt8	10.83	10.65	8.18	7.05

Figure 4. Parallel decompression times on the CCITT images (ms.)

Image	1 block	256 blocks
1	410	310
2	400	310
3	420	310
4	420	310
5	410	310

Figure 5. Sequential compression times on the 4096×4096 pixels images (ms.)

Otherwise, we bound by twelve the number of bits to encode either the width or the length of the monochromatic rectangle. We use either four or eight or twelve bits to encode one rectangle side. Therefore, nine different kinds of rectangle are defined. A monochromatic rectangle is encoded in the following way:

- the flag field indicating the color;
- three or four bits encoding one of the nine kinds of rectangle;
- bits for the length and the width.

Four bits are used to indicate when twelve bits or eight and twelve bits are needed for the length and the width. This way of encoding rectangles plays a relevant role for the compression performance. In fact, it wastes four bits when twelve bits are required for the sides but saves four to twelve bits when four or eight bits suffice.
The procedure for computing the largest monochromatic rectangle with left upper corner in position (i, j) takes $O(M \log M)$ time, where M is the size of the rectangle [3]. The positions covered by the detected rectangles are skipped in the linear

Image	1 block	256 blocks
1	200	160
2	200	160
3	210	160
4	210	160
5	200	160

Figure 6. Sequential decompression times on the 4096×4096 pixels images (ms.)

Image	16 blocks	256 blocks
1	40	30
2	40	30
3	40	30
4	40	30
5	40	30

Figure 7. Parallel compression times on the 4096×4096 pixels images (ms.)

Image	16 blocks	256 blocks
1	20	10
2	20	10
3	20	10
4	20	10
5	20	10

Figure 8. Parallel decompression times on the 4096×4096 pixels images (ms.)

scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. The analysis of the running time of this algorithm involves a *waste factor*, defined as the average number of detected monochromatic rectangles covering the same pixel. We experimented that the waste factor is less than 2 on realistic image data. Therefore, the heuristic takes $O(n \log M)$ time. On the other hand, the decoding algorithm is linear.

3 The Sequential Speed-Up

The variable length coding technique explained in the previous section has been applied to the CCITT test set of bi-level images. The images of the CCITT test set are 1728×2376 pixels. If these images are partitioned into 4^k sub-images and the compression heuristic is applied independently to each sub-image, the compression effectiveness remains about the same for $1 \leq k \leq 4$. Though the waste factor decreases with the increasing of k. As mentioned in the previous section, the waste factor is less than 2 on realistic image data (as the CCITT test set) for $k = 0$ and decreases to about 1 when $k = 4$. It follows that if we refine the partition by splitting the blocks horizontally and vertically, after four refinements no further relevant speed-up is obtained (we consider a speed-up relevant if it has the order of magnitude of one centisecond). The experimental results in figure 1 show the speed-up obtained if the

image is partitioned into up to 256 blocks and sequentially each block is compressed independently. Obviously, there is a similar speed-up for the decompressor as shown in figure 2. Decompression is about twice faster than compression. Compression and decompression running times were obtained using a single core of a quad core with a CPU Intel Core 2 Quad Q9300 – 2.5 GHz with 3.25 GB of RAM.
The sequential speed-up can also be applied to a parallel implementation on a small scale system since the experimental results show that the speed-up happens for an image partitioned into less than 256 blocks. However, in order to decompress in parallel raw data are associated with the flag field 110 so that we can indicate with 111 the end of the encoding of a block. The parallel compression and decompression running times on the CCITT image test set are given in figures 3 and 4 using the four cores of the quadcore machine. Obviously, a similar experiment could be run using two refinemets or one refinement of the partition on a system with 16 or 64 cores respectively. Experimental results with 16 processors on test set of larger topographic images are presented in the next section.

4 Speeding-Up Parallel Computation

The compression effectiveness of the variable-length coding technique depends on the sub-image size rather than on the whole image. In fact, if we consider a test set of larger binary images as the five 4096×4096 pixels half-tone topographic images shown in [3] and these images are partitioned into 4^k sub-images, again we obtain about the same compression effectiveness for $1 \leq k \leq 4$ and a sequential speed-up with the increasing of k. On the other hand, no further speed-up is obtained for $k = 5$, that is, the waste factor seems to be determined by the number of refinements independently from the image size on realistic data. We give in figures 5 and 6 the compression and decompression running times with one processor of a 256 Intel Xeon 3.06 GHz CPU's machine (`avogadro.cilea.it`) on the five images before and after the partition into 256 blocks. The compression and decompression running times with 16 processors before and after the partition into 256 blocks are given in figures 7 and 8. This means that each processor works on a sub-image partitioned into 16 blocks when the number of blocks is 256. Running times are given as milliseconds, which are the time units used for the quadcore experiments, but the centisecond is the actual time unit employed with the avogadro machine and the running time is provided as an integer number.

5 Conclusions

In this paper, we showed that the most efficient way to apply monochromatic pattern substitution to binary image compression with a sequential algorithm is to compress independently the 256 blocks of a partitioned input image. Since a speed-up is obtained as well with partitions of lower cardinality, this can be used to improve the performance of parallel compression on a small scale distributed system. We presented experimental results with four and sixteen processors. As future work, we wish to experiment with more processors by implementing the procedure on a graphical processing unit [1].

References

1. L. Bianchi, R. Gatti, and L. Lombardi: *The future of parallel computing: Gpu vs cell - general purpose planning against fast graphical computation architecture, which is the best solution for general purpose computation*, in Proceedings GRAPP, 2008, pp. 419–425.
2. L. Cinque, S. D. Agostino, and L. Lombardi: *Speeding-up lossless image compression: Experimental results on a paralle machine*, in Proceedings Prague Stringology Conference, 2008, pp. 35–45.
3. L. Cinque, S. D. Agostino, and L. Lombardi: *Binary image compression via monochromatic pattern substitution: Scalability and effectiveness*, in Proceedings Prague Stringology Conference, 2010, pp. 103–115.
4. L. Cinque, S. D. Agostino, and L. Lombardi: *Scalability and communication in parallel low-complexity lossless compression.* Mathematics in Computer Science, 3 2010, pp. 391–406.
5. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
6. J. A. Storer: *Lossless image compression using generalized lz1-type methods*, in Proceedings IEEE Data Compression Conference, 1996, pp. 290–299.
7. J. A. Storer and H. Helfgott: *Lossless image compression by block matching.* The Computer Journal, 40 1997, pp. 137–145.
8. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of ACM, 29 1982, pp. 928–951.

Improving Exact Search of Multiple Patterns From a Compressed Suffix Array

Kalle Karhu

Department of Computer Science and Engineering
Aalto University
kalle.karhu@aalto.fi

Abstract. Self-indexes are largely studied and widely applied structures in string matching. However, the exact matching of multiple patterns using self-indexes is a topic that has not been the subject of concentrated study although it is an area that may have direct and indirect applications and uses in fields such as bioinformatics. This paper presents a method of improving the exact search of multiple patterns from a compressed suffix array. The proposed method is able to cut down run-times for the handled patterns by as much as 71.6 %. A set of 1000 patterns of length 1000 nucleotides each is found from a text of 50 MB in size 14.0 % faster than by searching the patterns using the locate functionality of the compressed suffix array.

Keywords: self-indexes, pattern matching, indexing, text algorithms

1 Introduction

Self-indexes have been an expanding area of research in recent years. As one leading example solution, FM-index [1] is widely used in many approaches and tools, large portions of which are closely related to bioinformatics [5,3]. In the problem frames of bioinformatics, it is not uncommon to search multiple sequences successively from the same text database. However, the possible improvements related to searching multiple patterns at once have not been studied very broadly to date, when considering the cases of searching text from an index structure.

The focus of this work is to seek possible improvements in one case of searching multiple patterns from an index structure. The index structure that is being considered is a self-index, the compressed suffix array (CSA) [6]. More specifically, this work focuses on the cases where one or more preprocessed sets of patterns are being searched from multiple preprocessed text databases. In such a problem frame, the preprocessing of a set of patterns needs to be done only once per set, but as the single set will be searched multiple times, the cost of the preprocessing is spread over multiple searches. Because of this, it is not reasonable to take the preprocessing times directly into account when looking at the run-time of a single search.

Moreover, the focus of this work is on exact matching which can be seen as a starting point for more practical implementations, including approximate matching. Even in bioinformatics, where exact matching is rarely sought after, it is noteworthy that a large number of successful tools use exact matching as part of a seed-and-extend methodology.

2 Methods

The workings of the proposed method are divided into three work phases: preprocessing of the text, preprocessing of the set of patterns, and searching the set of patterns

Kalle Karhu: Improving Exact Search of Multiple Patterns From a Compressed Suffix Array, pp. 226–231.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

from the text. The two preprocessing steps need to be done only once for each set of patterns and each text. The search phase uses both of these preprocessing steps to improve speed in the search.

2.1 Preprocessing of text

The text is preprocessed by making a compressed suffix array [6] of it. The implementation provided in the Pizza&Chili corpus [2] is used to produce this index.

The most important functionality for the searches that are the focus of interest of this work is the *locate* function. Locate function allows location of the *occ* occurrences of a query of length m from a text of length n in $O(m \log(n) + occ \cdot \log^{\epsilon}(n))$ time. Here ϵ belongs to $(0, 1)$, depending on the chosen space/time trade-off.

2.2 Preprocessing of patterns

The set of patterns is preprocessed in order to find a certain set of substrings of the patterns. The goal is to find a collection of substrings, where each substring would occur in a large number of patterns, while still occurring comparatively rarely in the text.

To achieve this goal, the proposed method uses a compression tool named Re-Pair [4] to find suitable substrings from the set of patterns. Re-Pair recursively replaces the most frequent pair of symbols in a text by a new symbol, reevaluating the frequencies of all of the symbol pairs with respect to the now-extended alphabet, repeating this process until there is no pair of adjacent symbols that occur twice. These new symbols correspond to phrases, which are substrings occurring more than once in the processed text. These phrases comprise the pool of potential substrings to be used in our method.

In order to limit the number of occurrences of the substrings in the text, a threshold value is set to limit the minimum length of a substring. After applying this threshold, the remaining substrings are searched from a CSA made from the set of patterns, to retrieve the number of patterns the substrings occur in and the offsets of the substring occurrences from the start of the patterns. After this search, the substrings are sorted in descending order, by the number of patterns in which the substring occurs. The patterns in which each phrase occurs and the respective offsets from the start of the pattern are saved, as this information is needed in the search phase.

2.3 Searching a set of patterns from text

In the search phase, the preprocessed set of patterns is searched from the preprocessed text. The substrings obtained during the preprocessing are searched from the text in descending order by the number of patterns in which they occur, using the locate functionality of the CSA. For each occurrence of a substring, possible occurrences of the patterns that include the substring are checked by character comparison. First the pattern is compared, character by character, with the text, starting from the beginning of the pattern, continuing up to the occurrence of the substring in the pattern. This is followed by comparing the characters of the pattern and the text, starting from the end of the pattern, moving towards the occurrence of the substring. If any mismatch is found during the exact match or if the whole pattern matches

the text, the search continues with processing the next pattern where the substring occurs.

When all occurrences of a substring have been checked with all of the patterns corresponding to the substring, all of these patterns are marked as *treated*. As all occurrences of a pattern are found when checking all occurrences of a substring of the pattern, the patterns that are marked as treated need not to be checked when handling later substrings.

After all of the substrings obtained from the preprocessing have been handled, the remaining patterns that are not yet marked as treated are searched using the locate functionality of the compressed suffix array for the full pattern. Alternatively, the search using the substrings can be terminated after a certain set amount of patterns have been marked as treated, finishing the remaining patterns with the locate functionality.

3 Results

3.1 Data

The text used was a DNA text of 50 MB in size, obtained from the Pizza&Chili corpus [2]. From this text, 1000 substrings of length 1000 nucleotides each were randomly picked, comprising the set of patterns. It was noticed that each of these patterns occurred exactly once in the text.

3.2 Experiments

All the following runs were done using a single Intel®Core™i7 CPU 860 @ 2.80 GHz on a PC with 16 GB RAM. The implementations were done with C++.

The pattern set described above was preprocessed as described in Section 2.2. The creation of the compressed suffix array for the patterns took 0.23 seconds, resulting in an index totaling 743.5 KB in size, using parameters samplerate = 16 and samplepsi = 128. The Re-Pair compression tool was run on the set of patterns to retrieve the full list of substrings occurring more than once as the side product of compressing the set of patterns, which took 0.44 seconds. Lastly, the occurrences of the substrings in the patterns were searched, varying the threshold determining the minimum substring lengths. This parameter was given values of 25, 28, 30, 33 and 35. Resulting run-times for this third step of preprocessing the pattern set are shown in Table 1, together with the total times taken by the preprocessing for each minimum substring length. The substring search times are averages over five repeats of searches.

Minimum substring length	25	28	30	33	35
Searching substring occurrences from patterns (s)	0.166	0.150	0.142	0.132	0.130
Total preprocessing time (s)	0.836	0.820	0.812	0.802	0.800

Table 1. The times taken by searching the substrings from the set of patterns as the function of minimum substring length, together with the total preprocessing times.

After this preprocessing of the pattern set, text was preprocessed by creating a compressed suffix array of it, using parameters samplerate = 16 and samplepsi = 128. The creation of the index took 22.69 seconds and the total size of the resulting index was 36.8 MB.

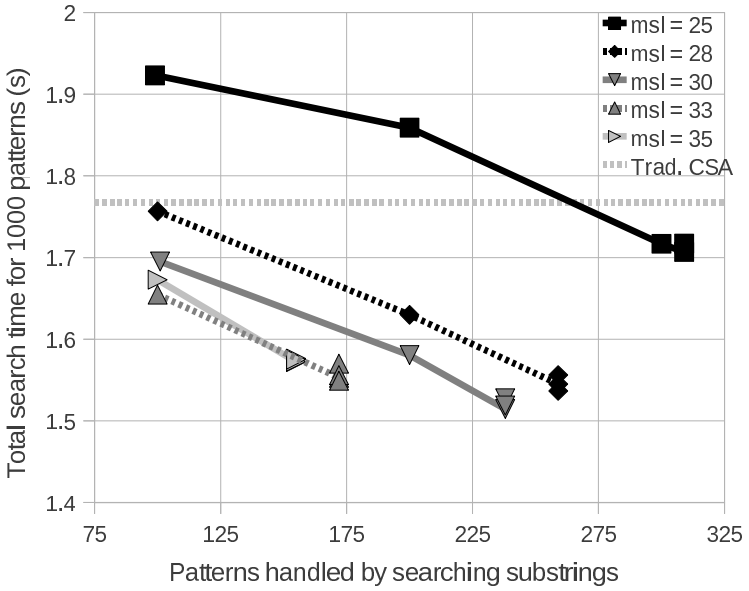


Figure 1. Time taken for searching the 1000 patterns from the text as the function of patterns handled by the proposed method, with varying minimum substring lengths. Dashed light gray line marks the average run-time for the pattern when using locate of CSA for the full patterns.

The preprocessing steps were followed by searching the set of patterns from the text. Searches were done separately for each of the minimum substring lengths. Additionally, the number of patterns allowed to be searched with the proposed method varied from 100 to 500. However, the actual number of patterns that had common substrings of required length within them was in some cases less than this, resulting in a smaller number of patterns being handled with the proposed method. The run-times of the proposed method were compared to searching all of the patterns with the locate functionality of the compressed suffix array implementation. Each of the runs were repeated 50 times. The average run times for each of the parameter sets and the traditional CSA are shown in Figure 1.

The average times for a pattern to be found by searching a substring and then extending are shown in Figure 2. For comparison, the average time for searching a pattern using the locate functionality for the full pattern is also shown in the figure.

Looking at the full runs of 1000 patterns, the best results were retrieved when using a minimum substring length of 30, resulting in 14.0 % saving in run-times, when 238 patterns were found by using the proposed method. When considering the average time for a single pattern to be found by searching the substring and then checking the exact match, the best results were retrieved when using a minimum substring length of 35, resulting in 71.6 % saving in run-times per pattern, when 155 patterns were found by using the proposed method.

Lastly, when a pattern was handled by searching the substring and then checking the exact match, the time the exact matching took was compared to the total time of this process. The portion of the time taken by the exact matching per pattern as the function of minimum sequence length is shown in Table 2.

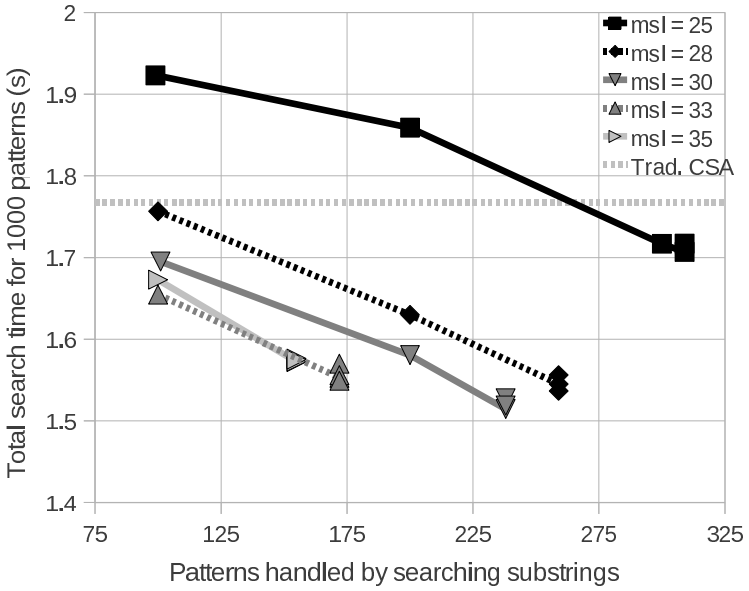


Figure 2. Average search time of a pattern when using the proposed method as the function of patterns handled by the proposed method, with varying minimum substring lengths. Dashed light gray line marks the average search time of a pattern when using locate of CSA for the full pattern.

Minimum substring length	25	28	30	33	35
Portion of time spent in exact matching (%)	14.9	14.5	14.4	8.1	7.9

Table 2. The portion of run-time taken by exact matching, when using the proposed method for a pattern, as the function of minimum substring length.

4 Conclusions

The results show two clear trends as far as run-times are concerned as a function of minimum substring length and the number of patterns handled by the proposed method. Firstly, it is clear that as the minimum substring length is raised, the average time it takes for a pattern to be found by the proposed method decreases. Secondly, as more patterns are handled by the proposed method, again the average time it takes for a pattern to be found by the proposed method decreases.

The first of these trends suggests that longer substrings are better in improving the run-times of the searches than short ones. This is very sensible, as longer substrings are expected to occur in the text less commonly, on average, resulting in fewer occurrences to be checked by the exact method. The latter trend suggests that the first substrings on the list, which occur in the largest number of patterns, are not optimal in the sense of decreasing the run-times per pattern. This is probably because the substrings that occur commonly in the set of patterns also occur commonly in the text, resulting in a large number of excess occurrences to be checked with the exact matching. However, regardless of the mentioned flaw, the proposed method was able to improve the run-times of the searches remarkably by reducing the total sum of the lengths of the patterns and substrings to be searched with the locate functionality of the CSA.

4.1 Discussion and Future Work

The current methods for choosing and sorting the substrings to be used are relatively straight-forward. As the substrings that head the list currently in use are clearly less than perfect, it is also clear there is room for improvement. This may be because the data is not independently and identically distributed, which means that longer sequences are not necessarily occurring more rarely in the text, especially so if they occur frequently in the set of patterns.

One approach, that will be studied in the future, is to consider the k-mer distributions of the substrings and compare them to the k-mer distributions of samples of sequences similar to the text to be. This would most likely give better estimates of the probability of a substring to occur in the text.

The scoring and sorting of the substrings that overcome a set threshold is another area of future improvements. Instead of using a simple threshold, it would probably be profitable to take into account both the expected number of occurrences in the text and the number of occurrences in the pattern set and give a score to each substring.

Lastly, when the substrings are being sorted, it should be dynamically taken into account which patterns are already being taken care of by a substring with a higher score.

Acknowledgements

I would like to thank Travis Gagie for comments and suggestions that helped arrive at the ideas presented in this work. The work was supported by the Academy of Finland (grant 134287).

References

1. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in IEEE Symposium on Foundations of Computer Science (FOCS 2000), 2000, pp. 390–398.
2. P. Ferragina and G. Navarro: *Pizza&Chili – compressed indexes and their testbeds*, May 2011.
3. B. Langmead, C. Trapnell, M. Pop, and S. Salzberg: *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.* Genome Biol., 10(3) 2009, p. R25.
4. N. J. Larsson and A. Moffat: *Off-line dictionary-based compression.* Proceedings of the IEEE, 88(11) 2000.
5. H. Li and R. Durbin: *Fast and accurate short read alignment with Burrows-Wheeler transform.* Bioinformatics, 25(14) 2009, pp. 1754–1760.
6. K. Sadakane: *New text indexing functionalities of the compressed suffix arrays.* Journal of Algorithms, 48(2) 2003, pp. 294–313.

Author Index