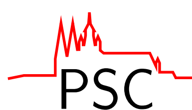
# Prague Stringology Conference 1996–2016

*$20^{th}$ Anniversary*

# Proceedings of the Prague Stringology Conference 2016

*Edited by Jan Holub and Jan Žďárek*

August 2016




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

## Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2016 (PSC 2016) held on August 29–31, 2016 at the Czech Technical University in Prague, which organises the event. The conference focused on stringology, i.e., a discipline concerned with algorithmic processing of strings and sequences, and related topics.

The submitted papers were reviewed by the program committee subject to originality and quality. The fourteen papers in this proceedings made the cut and were selected for regular presentation at the conference. In addition, this volume contains an abstract of the invited talk "The use and usefulness of Fibonacci compression codes" by Shmuel T. Klein.

The Prague Stringology Conference has a long tradition. PSC 2016 is the twentieth event PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2015 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions have been published regularly in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, the International Journal of Foundations of Computer Science, and the Discrete Applied Mathematics.

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.

We would like to thank all those who had submitted papers for PSC 2016 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 2016. Last, but not least, our thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

*In Prague, Czech Republic*
*on August 2016*

Jan Holub and Bruce W. Watson

# Conference Organisation

## Program Committee

| | |
|---|---|
| Amihood Amir | (Bar-Ilan University, Israel) |
| Gabriela Andrejková | (P. J. Šafárik University, Slovakia) |
| Simone Faro | (Università di Catania, Italy) |
| 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, *Honorary 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, *Co-chair* | (FASTAR Group/Stellenbosch University, South Africa) |
| Jan Žďárek | (Czech Technical University in Prague, Czech Republic) |

## Organising Committee

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

## External Referees

Ritu Kundu
Arnaud Lefebvre
Yuto Nakashima
Elise Prieur-Gaston
Mikaël Salson
Steven Watts

# Table of Contents

## Invited Talk

The Use and Usefulness of Fibonacci Codes *by Shmuel T. Klein* . . . . . . . . . . . . . 1

## Contributed Talks

Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings *by Diptarama, Ryo Yoshinaka, and Ayumi Shinohara* . . . . . . . . . . . . . . . . . . . . . . . . 7

Using Human Computation in Dead-zone based 2D Pattern Matching *by Kamil Awid, Loek Cleophas, and Bruce W. Watson* . . . . . . . . . . . . . . . . . . . . . . . . 22

Generating All Minimal Petri Net Unsolvable Binary Words *by Evgeny Erofeev, Kamila Barylska, Łukasz Mikulski, and Marcin Piątkowski* . . . . . . . . . 33

Interpreting the Subset Construction Using Finite Sublanguages *by Mwawi Msiska and Lynette van Zijl* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

Accelerated Partial Decoding in Wavelet Trees *by Gilad Baruch, Shmuel T. Klein, and Dana Shapira* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

A Family of Data Compression Codes with Multiple Delimiters *by Igor O. Zavadskyi and Anatoly V. Anisimov* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

A Resource-frugal Probabilistic Dictionary and Applications in (Meta)Genomics *by Camille Marchet, Antoine Limasset, Lucie Bittner, and Pierre Peterlongo* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

The String Matching Algorithms Research Tool *by Simone Faro, Thierry Lecroq, Stefano Borzì, Simone Di Mauro, and Alessandro Maggio* . . . . . . . . . . . . 99

Jumbled Matching with SIMD *by Sukhpal Singh Ghuman and Jorma Tarhio* . . 114

Forced Repetitions over Alphabet Lists *by Neerja Mhaskar and Michael Soltys* 125

Computing Smallest and Largest Repetition Factorizations in $O(n \log n)$ Time *by Hiroe Inoue, Yoshiaki Matsuoka, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda* . . . . . . . . . . . . . . . . . . . . . . . . . . . 135

Computing All Approximate Enhanced Covers with the Hamming Distance *by Ondřej Guth* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146

Dynamic Index and LZ Factorization in Compressed Space *by Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158

Algorithms to Compute the Lyndon Array *by Frantisek Franek, A. S. M. Sohidull Islam, M. Sohel Rahman, and William F. Smyth* . . . . . . . . . . . . . . . . . 172

*Author Index* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185

# The Use and Usefulness of Fibonacci Codes
## (*Invited talk*)

Shmuel T. Klein

Computer Science Department, Bar Ilan University, Israel
tomi@cs.biu.ac.il

## 1 Introduction

Contrary to our intuition led by the knowledge that the price for digital storage is constantly dropping, compression techniques are not becoming obsolete, and in fact research in data compression is flourishing as can be seen by the large number of papers published constantly on the topic. For instance, very large textual databases as those found in large Information Retrieval Systems, could contain hundreds of millions of words, which should be compressed by some method giving, in addition to good compression performance, also very fast decoding and the ability to search for the appearance of some strings directly in the compressed text.

Classical Huffman coding, when applied to individual characters, gives relatively poor compression, but when every word of a large textual database is considered as an atomic element to be encoded, this so-called Huffword variant may compete with the best other compression methods [11]. Yet the codewords of a binary Huffman code are not necessarily aligned on byte boundaries, which complicates both the decoding process and the ability to perform searches in the compressed file. The next step was therefore to pass to 256-ary Huffman coding, in which every codeword consists of an integral number of 8-bit bytes [4]. The loss incurred in the compression efficiency, which is only a few percent for large enough alphabets, is compensated for by the advantages of the easier processing.

When searches in the compressed text should also be supported, Huffman codes suffer from a problem of synchronization: denoting by $\mathcal{E}$ the encoding function, the compressed form $\mathcal{E}(x)$ of an element $x$ may appear in the compressed text $\mathcal{E}(T)$, without corresponding to an actual occurrence of $x$ in the text $T$, because the occurrence of $\mathcal{E}(x)$ is not necessarily aligned on codeword boundaries. This problem has been overcome in [8], relying on the tendency of Huffman codes to resynchronize quickly after errors, but the suggested solution is probabilistic and may produce wrong results. As alternative, [4] propose to reserve the first bit of each byte as *tag*, which is used to identify the last byte of each codeword, thereby reducing the order of the Huffman tree to 128-ary. These *Tagged Huffman codes* have then been replaced by *End-Tagged Dense codes* (ETDC) in [3] and by $(s, c)$-*Dense codes* (SCDC) in [2]. The two last mentioned codes consist of fixed codewords which do not depend on the probabilities of the items to be encoded. Thus their construction is simpler than that of Huffman codes: all one has to do is to sort the items by non-increasing frequency and then assign the codewords accordingly, starting with the shortest ones.

We show here that similar properties, and in fact some interesting others, can be obtained by *Fibonacci codes* [7], which have been suggested in the context of compression codes for the unbounded transmission of strings [1] and because of their robustness against errors in data communication applications [5]. They are also studied as a simple alternative to Huffman codes in [12]. The properties of representing

Shmuel T. Klein: The Use and Usefulness of Fibonacci Codes, pp. 1–5.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

integers in a Fibonacci based numeration system have been known long before the codes were suggested [13], and the following challenging quote appeared on page 211 in the book *Computer Number Systems and Arithmetic* by N.R. Scott in 1985:

The Fibonacci number system (so-called by Knuth)
is another remarkable and
remarkably useless number system.

This obviously has been taken as an incentive to look for ever more useful applications of Fibonacci codes, not only as alternatives to dense codes for large textual word-based compression systems. They are in particular mentioned: in [10] as a good choice for compressing a set of small integers; in [6] to improve modular exponentiation; and in [9] as a basis to devise a rewriting code to enhance the repeated use of flash memory.

In the next section, we review the relevant features of Fibonacci codes of order $m \geq 2$. Examples of several applications will then be given in the talk itself.

## 2 Fibonacci codes

Fibonacci numbers of order $m \geq 2$, denoted by $F_i^{(m)}$, are defined by the following recurrence relation:

$$F_n^{(m)} = F_{n-1}^{(m)} + F_{n-2}^{(m)} + \cdots + F_{n-m}^{(m)} \qquad \text{for } n > 0,$$

and the boundary conditions

$$F_0^{(m)} = 1 \qquad \text{and} \qquad F_n^{(m)} = 0 \quad \text{for } -m < n < 0.$$

For fixed order $m$, the number $F_n^{(m)}$ can be represented as a linear combination of the $n$th powers of the roots of the corresponding polynomial $P(m) = x^m - x^{m-1} - \cdots - x - 1$. $P(m)$ has only one real root that is larger than 1, which we shall denote by $\phi_{(m)}$, the other $m-1$ roots are complex numbers with norm $< 1$ (for $m = 2$, the second root is also real and its absolute value is $< 1$). Therefore, when representing $F_n^{(m)}$ as such a linear combination, the term with $\phi_{(m)}^n$ will be the dominant one, and the others will rapidly become negligible for increasing $n$.

For example, $m = 2$ corresponds to the classical Fibonacci sequence and $\phi_{(2)} = \frac{1+\sqrt{5}}{2} = 1.6180$ is the well-known golden ratio. As a matter of fact, the entire Fibonacci sequence can be obtained by $F_n^{(m)} = [a_{(m)}\phi_{(m)}^n]$, where $a_{(m)}$ is the coefficient of the dominating term in the above mentioned linear combination, and $[x]$ means that the value of the real number $x$ is rounded to the closest integer. Table 1 lists the first few elements of the Fibonacci sequences of order up to 6. The column headed *General Term* brings the values of $a_{(m)}$ and $\phi_{(m)}$. For larger $n$, the numbers $a_{(m)}\phi_{(m)}^n$ are usually quite close to integers.

The standard representation of an integer as a binary string is based on a numeration system whose basis elements are the powers of 2. If $B$ is represented by the $k$-bit string $b_{k-1}b_{k-2}\cdots b_1b_0$, then $B = \sum_{i=0}^{k-1} b_i 2^i$. But many other possible binary representations do exist, and those using the Fibonacci sequences as basis elements have some interesting properties. Let us first consider the standard Fibonacci numbers of order 2.

| $F_n^{(m)}$ | General Term | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m=2$ | $0.7236\ (1.6180)^n$ | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 |
| $m=3$ | $0.6184\ (1.8393)^n$ | 1 | 2 | 4 | 7 | 13 | 24 | 44 | 81 | 149 | 274 | 504 | 927 | 1705 |
| $m=4$ | $0.5663\ (1.9275)^n$ | 1 | 2 | 4 | 8 | 15 | 29 | 56 | 108 | 208 | 401 | 773 | 1490 | 2872 |
| $m=5$ | $0.5379\ (1.9659)^n$ | 1 | 2 | 4 | 8 | 16 | 31 | 61 | 120 | 236 | 464 | 912 | 1793 | 3525 |
| $m=6$ | $0.5218\ (1.9836)^n$ | 1 | 2 | 4 | 8 | 16 | 32 | 63 | 125 | 248 | 492 | 976 | 1936 | 3840 |

**Table 1.** Fibonacci numbers of order $m = 2, 3, 4, 5, 6$

Any integer $B$ can be represented by a binary string of length $r$, $c_r c_{r-1} \cdots c_2 c_1$, such that $B = \sum_{i=1}^{r} c_i F_i^{(2)}$. The representation will be unique if one uses the following procedure to produce it: given the integer $B$, find the largest Fibonacci number $F_r^{(2)}$ smaller or equal to $B$; then continue recursively with $B - F_r^{(2)}$. For example, $45 = 34 + 8 + 3$, so its binary Fibonacci representation would be 10010100. As a result of this encoding procedure, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s.

This property can be exploited to devise an infinite code whose set of codewords consists of the Fibonacci representations of the integers: to assure the code being uniquely decipherable (UD), each codeword is prefixed by a single 1-bit, which acts like a *comma* and permits to identify the boundaries between the codewords. The first few elements of this code would thus be $\{u_1, u_2, \ldots\}$ = {**1**1, **1**10, **1**100, **1**101, **1**1000, **1**1001,...}, where the separating 1 is put in boldface for visibility. A typical compressed text could be **1**1001**1**1001**1**101**1**1**1**101, which is easily parsed as $u_6 u_3 u_4 u_1 u_4$. Though being UD, this is not a prefix code, so decoding may be somewhat more involved. In particular, the first codeword 11, which is the only one containing no zeros, complicates the decoding, because if a run of several such codewords appears, the correct decoding of the codeword preceding the run depends on the parity of the length of the run. Consider for example the encoded string 11011111110: a first attempt to parse it as 110 | 11 | 11 | 11 | 10 $= u_2 u_1 u_1 u_1 10$ would fail, because the tail 10 is not a codeword; hence only when trying to decode the fifth codeword do we realize that the first one is not correct, and that the parsing should rather be 1101 | 11 | 11 | 110 $= u_4 u_1 u_1 u_2$.

To overcome this problem, [1,5] suggest to reverse all the codewords, yielding the set $\{v_1, v_2, \ldots\}$ = {11, 011, 0011, 1011, 00011, 10011, ...}, which is a prefix code, since all codewords are terminated by 11 and this substring does not appear anywhere in any codeword, except at its suffix. In addition, we show below that having a reversed representation, with the bits corresponding to increasing basis elements running from left to right rather than as usual, is advantageous for fast decoding. Table 2 brings a larger sample of this set of codewords in the column headed Fib2. Note that the order of the elements is not lexicographic, e.g., 10011 precedes 01011.

The generalization to higher order seems at first sight straightforward: any integer $B$ can be uniquely represented by the string $d_s d_{s-1} \cdots d_2 d_1$ such that $B = \sum_{i=1}^{s} d_i F_i^{(m)}$ using the iterative encoding procedure mentioned above. In this representation, there are no consecutive substrings of $m$ 1s. For example, the representations of the integers 10, 11, 12 and 13 using $F^{(3)}$ are, respectively, 1011, 1100, 1101 and 10000. But simply adding now $m-1$ 1's as commas and reversing the strings does not yield a prefix

code for $m > 2$, and in fact the code so obtained is not even UD. For example, for $m = 3$, the above numbers would give the codewords $\{v_{10}, \ldots, v_{13}\} = \{$1101**11**, 0011**11**, 1011**11**, 00001**11**$\}$, but the encoding of the fourth element of the sequence would be $v_4 =$ 001**11**, which is a prefix of $v_{11}$. The string 0011110111 could be parsed both as 00111 | 10111 $= v_4 v_5$ and as 001111 | 0111 $= v_{11} v_2$. The problem stems from the fact that for $m > 2$, there can be more than one leading 1 in the representation of an integer, so adding $m - 1$ 1s may give a string of up to $2m - 2$ consecutive 1s. The fact that a string of $m$ 1s appears only as a suffix is thus only true for $m = 2$. To turn the sequence into a prefix code, the definition has to be amended as follows: the set Fib$m$ will be defined as the set of binary codewords of lengths $\geq m$, such that every codeword contains exactly one occurrence of the substring consisting of $m$ consecutive 1s, and this occurrence is the suffix of every codeword. The first elements of these codes for $m \leq 4$ are given in Table 2. For $m = 2$, this last definition is equivalent to the one above based on the representation with basis elements $F_n^{(2)}$; for $m > 2$, only a subset of the corresponding numbers is taken. There is nevertheless a connection between the codewords and the higher order Fibonacci numbers: for $m \geq 2$, and $n \geq 0$, the code Fib$m$ consists of

$$F_n^{(m)} \quad \text{codewords of length} \quad n + m.$$

| index | Fib2 | Fib3 | Fib4 |
|---|---|---|---|
| 1 | 11 | 111 | 1111 |
| 2 | 011 | 0111 | 01111 |
| 3 | 0011 | 00111 | 001111 |
| 4 | 1011 | 10111 | 101111 |
| 5 | 00011 | 000111 | 0001111 |
| 6 | 10011 | 100111 | 1001111 |
| 7 | 01011 | 010111 | 0101111 |
| 8 | 000011 | 110111 | 1101111 |
| 9 | 100011 | 0000111 | 00001111 |
| 10 | 010011 | 1000111 | 10001111 |
| 11 | 001011 | 0100111 | 01001111 |
| 12 | 101011 | 1100111 | 11001111 |
| 13 | 0000011 | 0010111 | 00101111 |
| 14 | 1000011 | 1010111 | 10101111 |
| 15 | 0100011 | 0110111 | 01101111 |
| 16 | 0010011 | 00000111 | 11101111 |
| 17 | 1010011 | 10000111 | 000001111 |
| 18 | 0001011 | 01000111 | 100001111 |
| 19 | 1001011 | 11000111 | 010001111 |
| 20 | 0101011 | 00100111 | 110001111 |
| 21 | 00000011 | 10100111 | 001001111 |
| 22 | 10000011 | 01100111 | 101001111 |
| 23 | 01000011 | 00010111 | 011001111 |
| 24 | 00100011 | 10010111 | 111001111 |
| 25 | 10100011 | 01010111 | 000101111 |
| 26 | 00010011 | 11010111 | 100101111 |
| 27 | 10010011 | 00110111 | 010101111 |
| 28 | 01010011 | 10110111 | 110101111 |
| 29 | 00001011 | 000000111 | 001101111 |
| 30 | 10001011 | 100000111 | 101101111 |
| 31 | 01001011 | 010000111 | 011101111 |
| 32 | 00101011 | 110000111 | 0000001111 |
| 33 | 10101011 | 001000111 | 1000001111 |
| 34 | 000000011 | 101000111 | 0100001111 |
| 35 | 100000011 | 011000111 | 1100001111 |

**Table 2.** Fibonacci codes of order $m = 2, 3, 4$

This is visualized in Table 2, where for each code, blocks of codewords of the same length are separated by horizontal lines. Within each such block of lengths $\geq m + 2$ for Fib$m$, the prefixes of the codewords obtained by removing the terminating string of 1s correspond to consecutive integers in the representation based on $F^{(m)}$. For decoding, the Fibonacci representation will thus be used to get the relative index within the block, to which the starting index of the given block has to be added.

Many of the features of Fibonacci codes are based on the following facts. To represent an integer $n$, more bits are needed in the Fibonacci than in the standard representation, since it is less dense. In fact, it can be shown that the number of bits needed for $m = 2$ is $\lfloor \log_{\phi_2}(\sqrt{5}n) - 1 \rfloor \simeq 1.4404 \log_2 n$. On the other hand, the probability of a 1-bit drops from $\frac{1}{2}$ to only $\frac{1}{2}\left(1 - \frac{1}{\sqrt{5}}\right) = 0.276$, and thus the average number of 1-bits is only $0.389 \log_2 n$ instead of $0.5 \log_2 n$. This can be exploited for many applications.

## References


1. A. Apostolico and A. Fraenkel: *Robust transmission of unbounded strings using Fibonacci representations*. IEEE Trans. Inform. Theory, IT–33 1987, pp. 238–245.
2. N. R. Brisaboa, A. Fariña, G. Ladra, G. Navarro, and M. Esteller: *(s,c)-dense coding: an optimized compression code for natural language text databases*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'03, vol. 2857, LNCS, 2010, pp. 122–136.
3. N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá: *An efficient compression code for text databases*, in Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16, 2003, Proceedings, 2003, pp. 468–481.
4. E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates: *Fast and flexible word searching on compressed text*. ACM Trans. Inf. Syst., 18(2) 2000, pp. 113–139.
5. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression*. Discrete Applied Mathematics, 64(1) 1996, pp. 31–55.
6. S. T. Klein: *Should one always use repeated squaring for modular exponentiation?* Inf. Process. Lett., 106(6) 2008, pp. 232–237.
7. S. T. Klein and M. K. Ben-Nissan: *On the usefulness of Fibonacci compression codes*. Comput. J., 53(6) 2010, pp. 701–716.
8. S. T. Klein and D. Shapira: *Pattern matching in Huffman encoded texts*. Inf. Process. Manage., 41(4) 2005, pp. 829–841.
9. S. T. Klein and D. Shapira: *Boosting the compression of rewriting on flash memory*, in Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 193–202.
10. D. A. Lelewer and D. S. Hirschberg: *Data compression*. ACM Comput. Surv., 19(3) 1987, pp. 261–296.
11. A. Moffat: *Word-based text compression*. Softw., Pract. Exper., 19(2) 1989, pp. 185–198.
12. R. Przywarski, S. Grabowski, G. Navarro, and A. Salinger: *FM-KZ: an even simpler alphabet-independent FM-index*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 28-30, 2006, 2006, pp. 226–241.
13. E. Zeckendorf: *Représentation des nombres naturels par une somme des nombres de Fibonacci ou de nombres de Lucas*. Bull. Soc. Roy. Sci. Liège, 41 1972, pp. 179–182.

# Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings

Diptarama, Ryo Yoshinaka, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University
6-6-05 Aramaki Aza Aoba, Aoba-ku, Sendai, Japan
{diptarama@shino., ry@, ayumi@}ecei.tohoku.ac.jp

**Abstract.** A multi-track string is a tuple of strings of the same length. The full permuted pattern matching problem is, given two multi-track strings $\mathbb{T} = (t_1, t_2, \ldots, t_N)$ and $\mathbb{P} = (p_1, p_2, \ldots, p_N)$ such that $|p_1| = \cdots = |p_N| \leq |t_1| = \cdots = |t_N|$, to find all positions $i$ such that $\mathbb{P} = (t_{r_1}[i : i+m-1], \ldots, t_{r_N}[i : i+m-1])$ for some permutation $(r_1, \ldots, r_N)$ of $(1, \ldots, N)$, where $m = |p_1|$ and $t[i : j]$ denotes the substring of $t$ from position $i$ to $j$. We propose new algorithms that perform full permuted pattern matching practically fast. The first and second algorithms are based on the Boyer-Moore algorithm and the Horspool algorithm, respectively. The third algorithm is based on the Aho-Corasick algorithm where we use a multi-track character instead of a single character in the so-called *goto* function. The fourth algorithm is an improvement of the multi-track Knuth-Morris-Pratt algorithm that uses an automaton instead of the failure function of the original algorithm. Our experiment results demonstrate that those algorithms perform permuted pattern matching faster than existing algorithms.

**Keywords:** permuted pattern matching, multi-track string, Boyer-Moore algorithm, Horspool algorithm, AC-automaton

## 1 Introduction

The pattern matching problem on strings is to find all occurrences of a pattern string in a text string. Pattern matching algorithms such as the Knuth-Morris-Pratt (KMP) algorithm [8], Boyer-Moore algorithm [2], and Horspool algorithm [5], perform pattern matching fast by preprocessing the pattern. On the other hand, pattern matching can be also performed by preprocessing the text into some data structure such as a suffix tree [12], a suffix array [10], and a position heap [4].

The *permuted pattern matching problem*, proposed by Katsura *et al.* [6,7], is a generalization of the pattern matching problem, where we compare tuples of strings. Tuples of strings can model various types of real data such as multiple-sensor data, polyphonic music data, and multiple genomes. We call a tuple of strings of the same length a *multi-track string*. The permuted pattern matching problem is, given two multi-track strings $\mathbb{T} = (t_1, t_2, \ldots, t_N)$ and $\mathbb{P} = (p_1, p_2, \ldots, p_M)$ where $M \leq N$ and $|t_1| = \cdots = |t_N| = n \geq |p_1| = \cdots = |p_M| = m$, to find all positions $i$ such that $\mathbb{P}$ is a permutation of a sub-tuple of $(t_1[i : i+m-1], \ldots, t_N[i : i+m-1])$, where $w[i : j]$ denotes the substring of $w$ from position $i$ to $j$. As an example, data from multiple sensors such as a three dimensional accelerometer can be considered as a multi-track string. This problem can be solved by constructing some data structure from the text such as a multi-track suffix tree [6] and a multi-track position heap [7], or by preprocessing the pattern like the Aho-Corasick (AC) automaton based algorithm [6] and KMP based algorithm [3] do.

In this paper, we focus on the *full* permuted matching problem, which is a special case of the permuted matching problem where we have $M = N$. We propose several

Diptarama, Ryo Yoshinaka, Ayumi Shinohara: Fast Full Permuted Pattern Matching Algorithms on Multi-track Strings, pp. 7–21.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

| Algorithm | Preprocessing time | Matching time | Online |
|---|---|---|---|
| AC-automaton based [6] | $O(mM \log \sigma)$ | $O(nN \log \sigma)$ | yes |
| Multi-track KMP* [3] | $O(mM)$ | $O(nN)$ | no |
| Filter-MTKMP [3] | $O(m(M + \sigma))$ | $O(n(mN + \sigma))$ | yes |
| MT-BM* | $O(m(M \log \sigma + \sigma))$ | $O(nN(m + \log \sigma) + n(N + \sigma))$ | yes |
| MT-H* | $O(m(M \log \sigma + \sigma))$ | $O(nN(m + \log \sigma) + n(N + \sigma))$ | yes |
| MT AC-automaton* | $O(dM \log \sigma)$ | $O(nN \log \sigma)$ | no |
| MT permuted matching automaton* | $O(mM \log \sigma)$ | $O(nN \log \sigma)$ | yes |

**Table 1.** Comparison of the algorithms for permuted pattern matching. Multi-track AC-automaton can find occurrences of *multiple* patterns. Algorithms with an asterisk are for full permuted pattern matching ($M = N$).

new algorithms that perform full permuted pattern matching practically fast. The first algorithm, *MT-BM*, is based on the Boyer-Moore algorithm [2] and the second one, *MT-H*, is based on the Horspool algorithm [5], on which we made a significant improvement by using a data structure called *track trie*. The third algorithm, *multi-track AC-automaton*, is an algorithm for dictionary matching on multi-track strings based on the AC-algorithm [1], where we use a multi-track character instead of a single character in the so-called *goto* function. The fourth algorithm, *multi-track permuted matching automaton*, is an improvement of multi-track KMP algorithm [3] that uses an automaton instead of the failure function in the KMP algorithm. Moreover, we conduct experiments and show that our algorithms perform permuted pattern matching faster than existing algorithms. The worst case running time of proposed algorithms and existing algorithms are summarized in Table 1, where $d$ is the total length of the patterns and $\sigma$ is the size of the alphabet.

## 2 Preliminaries

Let $w \in \Sigma^n$ be a string of length $n$ over an alphabet $\Sigma$ and $\sigma = |\Sigma|$ be the alphabet size. The length $n$ of $w$ is denoted by $|w|$. The *empty string*, denoted by $\varepsilon$, is a string of length 0. By $w[i]$ we denote the $i$-th character of $w$ for $i \in \{1, \ldots, n\}$. The substring of $w$ that begins at position $i$ and ends at position $j$ is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. We abbreviate $w[1 : i]$ to $w[: i]$ and $w[i : n]$ to $w[i :]$, which are called a *prefix* and a *suffix* of $w$, respectively. The reverse string of $w$ is denoted by $w^R$: that is, $w^R = w[n]w[n-1] \ldots w[2]w[1]$. For two strings $x$ and $y$, we denote by $x \prec y$ that $x$ is lexicographically smaller than $y$, and by $x \preceq y$ that either $x = y$ or $x \prec y$.

A *multi-track string* (or *multi-track* for short) $\mathbb{W} = (w_1, w_2, ..., w_N)$ is an $N$-tuple of strings $w_i \in \Sigma^n$, and each $w_i$ is called the *i-th track* of $\mathbb{W}$. A *multi-track character* $\mathbb{C} = (c_1, c_2, ..., c_N)$ is an $N$-tuple of characters $c_i \in \Sigma$. The length $n$ of strings in $\mathbb{W}$ is called the *length* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{len}$. The number $N$ of tracks in $\mathbb{W}$ is called the *track count* of $\mathbb{W}$ and denoted by $|\mathbb{W}|_{num}$. The multi-track character $(w_1[i], w_2[i], ...w_N[i])$ is denoted by $\mathbb{W}[i]$ and the multi-track $\mathbb{W}[i : j]$ is $(w_1[i : j], w_2[i : j], \ldots, w_N[i : j])$ for $1 \leq i \leq j \leq |\mathbb{W}|_{len}$. Similarly to the notation for strings, $\mathbb{W}[: i]$ and $\mathbb{W}[i :]$ mean $\mathbb{W}[1 : i]$ and $\mathbb{W}[i : |\mathbb{W}|_{len}]$ and called a prefix and a suffix of $\mathbb{W}$, respectively. Moreover, $\mathbb{W}[i][j]$ denotes $w_j[i]$.

Let $\mathbf{r} = (r_1, r_2, \ldots, r_N)$ be a permutation of $(1, 2, \ldots, N)$. For a multi-track $\mathbb{W} = (w_1, w_2, \ldots, w_N)$, $\mathbb{W}\langle \mathbf{r} \rangle = \mathbb{W}\langle r_1, r_2, \ldots, r_N \rangle = (w_{r_1}, \ldots, w_{r_N})$ is called a *permuted multi-track* of $\mathbb{W}$. The *sorted index* $SI(\mathbb{W})$ of a multi-track $\mathbb{W}$ is a permutation

$(r_1, \ldots, r_N)$ such that $w_{r_i} \preceq w_{r_{i+1}}$ for any $1 \le i < N$, where we assume $r_i < r_{i+1}$ in the case $w_{r_i} = w_{r_{i+1}}$. The sorted multi-track $sort(\mathbb{W})$ is defined as $\mathbb{W}\langle SI(\mathbb{W})\rangle$. The *reverse* of a multi-track $\mathbb{W} = (w_1, \ldots, w_N)$ is $\mathbb{W}^R = (w_1^R, \ldots, w_N^R)$. The sorted index of the reverse multi-track, denoted by $RI(\mathbb{W})$, is a permutation $(r_1, \ldots, r_N)$ such that $w_{r_i}^R \preceq w_{r_{i+1}}^R$ for any $1 \le i < N$. Note that $SI(\mathbb{W}[i :])$ and $RI(\mathbb{W}[: i])$ for $1 \le i \le n$ can be computed in $O(nN \log \sigma)$ time *offline* by using a suffix tree [6,11] or a suffix array [9], and $RI(\mathbb{W}[: i])$ for $1 \le i \le n$ can be computed in $O(n(N + \sigma))$ time *online* by using radix sort.

For two multi-tracks $\mathbb{X} = (x_1, x_2, \ldots, x_N)$ and $\mathbb{Y} = (y_1, y_2, \ldots, y_N)$, $\mathbb{X}$ *permuted-matches* $\mathbb{Y}$, denoted by $\mathbb{X} \stackrel{\bowtie}{=} \mathbb{Y}$, if $\mathbb{X} = \mathbb{Y}\langle \mathbf{r}\rangle$ for some permutation $\mathbf{r}$.

Throughout the paper, we assume that $\mathbb{P}$ is a pattern with $|\mathbb{P}|_{num} = M$ and $|\mathbb{P}|_{len} = m$, and $\mathbb{T}$ is a text with $|\mathbb{T}|_{num} = N = M$ and $|\mathbb{T}|_{len} = n \ge m$. The pattern matching problem on multi-tracks is defined as follows.

**Definition 1 (Full permuted pattern matching).** *Given a multi-track text $\mathbb{T}$ and a multi-track pattern $\mathbb{P}$, compute all positions $i$ that satisfy $\mathbb{P} \stackrel{\bowtie}{=} \mathbb{T}[i : i + m - 1]$.*

For example, given a text $\mathbb{T} = \begin{pmatrix} \texttt{aabaaaaa,} \\ \texttt{abaabbaa,} \\ \texttt{baaababa} \end{pmatrix}$ and a pattern $\mathbb{P} = \begin{pmatrix} \texttt{aba,} \\ \texttt{baa,} \\ \texttt{aaa} \end{pmatrix}$, we can see that the pattern matches at $\mathbb{T}[2 : 4] = \mathbb{P}$. Moreover, the pattern permuted matches with $T[6 : 8]$, since $\mathbb{P}\langle 3, 2, 1\rangle = \begin{pmatrix} \texttt{aaa,} \\ \texttt{baa,} \\ \texttt{aba} \end{pmatrix} = T[6 : 8]$. Therefore, we should output $\{2, 6\}$ in this case.

We remark that Katsura *et al.* defined a more general problem, where we have $|\mathbb{T}|_{num} = N \ge |\mathbb{P}|_{num} = M$ and our task is to find a subsequence $(r_1, \ldots, r_M)$ of $(1, \ldots, N)$ and a position $i$ for which $\mathbb{P} \stackrel{\bowtie}{=} \mathbb{T}\langle t_{r_1}, \ldots, t_{r_M}\rangle[i : i + m - 1]$ holds.

## 3 Boyer-Moore and Horspool algorithms for multi-track strings

In this section, we propose two permuted pattern matching algorithms that are based on the Boyer-Moore algorithm and the Horspool algorithm, which we call MT-BM and MT-H, respectively.

### 3.1 Multi-track Boyer-Moore algorithm

The original Boyer-Moore algorithm uses two failure functions $GS$ (good suffixes) and $BC$ (bad characters) to determine how much the position of a substring to compare should be shifted when a mismatch is found between the input patten and the substring of the text. Those functions are defined as follows on multi-tracks.

**Definition 2 (Suffixes).** *For a multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$, $suf[i]$ is the maximum value of $l$ such that $\mathbb{P}[i - l + 1 : i] \stackrel{\bowtie}{=} \mathbb{P}[m - l + 1 : m]$ for $1 \le i \le m$.*

**Definition 3 (Good suffixes).** *For multi-track $\mathbb{P}$ of length $|\mathbb{P}|_{len} = m$, $GS[m] = 1$ and $GS[i] = \min A$ for $0 \le i < m$, where*

$$A = \left\{\, 0 < s < i \;\middle|\; \mathbb{P}[i - s + 1 : m - s] \stackrel{\bowtie}{=} \mathbb{P}[i + 1 : m],\, \mathbb{P}[i - s : m - s] \not\stackrel{\bowtie}{=} \mathbb{P}[i : m] \,\right\} \\ \cup \left\{\, i \le s < m \;\middle|\; \mathbb{P}[1 : m - s] \stackrel{\bowtie}{=} \mathbb{P}[s + 1 : m] \,\right\} \cup \{m\}.$$

**Algorithm 1:** MT-BM and MT-H preprocessing functions

```
1  Function ComputeSuf(ℙ)
2      compute RI(ℙ[: i]) for 1 ≤ i ≤ m;
3      suf[m] ← m, j ← m, k ← m;
4      for i ← m − 1 to 1 do
5          if i > k and suf[m − (j − i)] < i − k then suf[i] ← suf[m − (j − i)] ;
6          else
7              if i < k then k ← i ;
8              j ← i;
9              while k > 0 and ℙ[k]⟨RI(ℙ[: j])⟩ = ℙ[k + m − j]⟨RI(ℙ[: m])⟩ do k ← k − 1 ;
10             suf[i] ← j − k ;
11     return suf;
12 Function ComputeGS(ℙ)
13     suf ← ComputeSuf(ℙ);
14     for i ← 1 to m do GS[i] ← m ;
15     j ← 1;
16     for i ← m to 1 do
17         if suf[i] = i then
18             while j ≤ m − i do
19                 if GS[j] = m then GS[j] ← m − i ;
20                 j ← j + 1;
21     for i ← 1 to m − 1 do GS[m − suf[i]] ← m − i ;
22     return GS;
23 Function ComputeBC(ℙ)
24     compute RI(ℙ[: i]) for 1 ≤ i ≤ m;
25     for i ← 1 to m − 1 do
26         if BC(ℙ[i]⟨RI(ℙ[: i])⟩) = m then BC.add(ℙ[i]⟨RI(ℙ[: i])⟩, m − i) ;
27         else BC(ℙ[i]⟨RI(ℙ[: i])⟩) ← m − i ;
28     return BC;
```

**Algorithm 2:** MT-BM

```
   Input: Multi-track 𝕋, Multi-track ℙ
   Output: match positions
1  compute RI(𝕋[: i]) for 1 ≤ i ≤ n;
2  compute RI(ℙ[: i]) for 1 ≤ i ≤ m;
3  BC ← ComputeBC(ℙ);
4  GS ← ComputeGS(ℙ);
5  j ← 0;
6  while j ≤ n − m + 1 do
7      i ← m;
8      while i > 0 and 𝕋[i + j]⟨RI(𝕋[: j + m])⟩ = ℙ[i]⟨RI(ℙ[: m])⟩ do i ← i − 1 ;
9      if i ≤ 0 then
10         output j + 1;
11         j ← j + GS[0];
12     else j ← j + max(GS[i], BC(𝕋[i + j]⟨RI(𝕋[: i + j])⟩) − (m − i)) ;
```

**Definition 4 (Bad character).** *For multi-track* $\mathbb{P}$ *of length* $|\mathbb{P}|_{len} = m$ *and a multi-track character* $\mathbb{C}$, $BC(\mathbb{C})$ *is the first occurrence position of* $sort(\mathbb{C})$ *in* $\mathbb{P}^R[2:]$*. The function* $BC(\mathbb{C})$ *returns* $m$ *if there is no occurrence of* $sort(\mathbb{C})$ *in* $\mathbb{P}^R[2:]$.

**Algorithm 3:** MT-H

**Input:** Multi-track $\mathbb{T}$, Multi-track $\mathbb{P}$
**Output:** match positions
1 compute $RI(\mathbb{T}[:i])$ for $1 \le i \le n$;
2 compute $RI(\mathbb{P}[:i])$ for $1 \le i \le m$;
3 $BC \leftarrow$ ComputeBC($\mathbb{P}$);
4 $j \leftarrow 0$;
5 **while** $j \le n - m + 1$ **do**
6 $i \leftarrow m$;
7 **while** $i > 0$ **and** $\mathbb{T}[i+j]\langle RI(\mathbb{T}[:j+m])\rangle = \mathbb{P}[i]\langle RI(\mathbb{P}[:m])\rangle$ **do** $i \leftarrow i - 1$ ;
8 **if** $i \le 0$ **then output** $j + 1$ ;
9 **else** $j \leftarrow j + BC(\mathbb{T}[j+m]\langle RI(\mathbb{T}[:j+m])\rangle))$ ;

In the implementation, *suf* and $GS$ can be represented as arrays, while $BC$ can be realized in a trie of the multi-track characters. We perform permuted-match instead of exact match when computing $GS$. Algorithm 1 shows how to construct $GS$ and $BC$. The array $GS$ is computed by ComputeGS, which uses array *suf* computed by ComputeSuf. Note that we compute $RI$ at the beginning (Lines 2 and 24) of the algorithm and will not recompute them when we use the values later.

**Lemma 5.** *The function ComputeSuf computes the array suf in $O(m(M+\sigma))$ time.*

*Proof.* First, $RI(\mathbb{P}[:i])$ can be computed in $O(m(M+\sigma))$ time by using radix sort. The **for** loop is executed $m-1$ times and the **while** loop at line 9 is executed at most $m$ times through the whole run, because $k$ is always reduced in each loop. Comparison of two multi-track characters of the pattern that executed in each loop can be computed in $O(M)$ time. ☐

**Lemma 6.** *The function ComputeGS computes GS in $O(m)$ time.*

*Proof.* All the **for** loops are executed at most $m$ times. The **while** loop is executed at most $m$ times through the whole execution of the algorithm, since $j$ is always increased and does not exceed $m$. ☐

**Lemma 7.** *The function ComputeBC computes BC in $O(m(M\log\sigma+\sigma))$ time.*

*Proof.* $RI(\mathbb{P}[:i])$ can be computed in $O(m(M+\sigma))$ time by using radix sort. Each edge in the trie of $BC$ can be accessed in $O(\log\sigma)$ time by using binary search. Since the depth of the trie is at most $M$, each $BC(\mathbb{P}[i])$ for $1 \le i \le m$ can be added and accessed in $O(M\log\sigma)$ time. ☐

By using both $GS$ and $BC$, MT-BM outputs the positions of the text that are permuted-matched with the pattern. The matching algorithm of MT-BM is shown in Algorithm 2.

**Theorem 8.** *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, MT-BM outputs the positions of the text that permuted-match with the pattern online in $O(nN(m+\log\sigma)+n(N+\sigma))$ time in the worst case with $O(m(M\log\sigma+\sigma))$ time preprocessing.*

*Proof.* From Lemmas 5, 6, and 7, Algorithm 2 needs $O(m(M\log\sigma+\sigma))$ time for preprocessing. Next, $RI(\mathbb{T}[:i])$ can be computed in $O(n(N+\sigma))$ time by using radix sort. In the outer **while** loop starting at line 6, the value of $j$ is increased by at least 1,

**Algorithm 4:** Track-trie construction algorithm (constructTrackTrie($\mathbb{P}$))

**Input:** Multi-track $\mathbb{P}$
**Output:** *trackTrie*

1. $newNode \leftarrow rootNode$;
2. $weight(rootNode) \leftarrow M$;
3. **for** $i \leftarrow 1$ **to** $M$ **do**
4.     $activeNode \leftarrow rootNode$;
5.     **for** $j \leftarrow m$ **to** $1$ **do**
6.         **if** $goto(activeNode, \mathbb{P}[j][i]) =$ **Null then**
7.             $newNode \leftarrow newNode + 1$;
8.             $weight(newNode) \leftarrow 1$;
9.             $goto(activeNode, \mathbb{P}[j][i]) \leftarrow newNode$;
10.             $activeNode \leftarrow newNode$;
11.         **else**
12.             $activeNode \leftarrow goto(activeNode, \mathbb{P}[j][i])$;
13.             $weight(activeNode) \leftarrow weight(activeNode) + 1$;

**Algorithm 5:** Track-trie matching algorithm (matchTrackTrie($\mathbb{T}, j$))

**Input:** Multi-track $\mathbb{T}$, index $j$, *trackTrie*
**Output:** mismatch position

1. $activeNode[k] \leftarrow rootNode$ for $1 \leq k \leq M$;
2. $temp(node) \leftarrow 0$ for all $node$ in $trackTrie$;
3. **for** $i \leftarrow m$ **to** $1$ **do**
4.     **for** $k \leftarrow 1$ **to** $M$ **do**
5.         **if** $goto(activeNodes, \mathbb{T}[i+j][k]) =$ **Null then return** $i$ ;
6.         **else**
7.             $activeNodes[k] \leftarrow goto(activeNodes[k], \mathbb{T}[i+j][k])$;
8.             $temp(activeNodes[k]) \leftarrow temp(activeNodes[k]) + 1$;
9.             **if** $temp(activeNodes[k]) > weight(activeNodes[k])$ **then return** $i$ ;
10. **return** 0;

so the loop is executed at most $n - m + 2$ times. In each execution of the outer loop, the inner **while** loop is executed at most $m$ times, where multi-track characters of the pattern and the text can be compared in $O(N)$ time. $BC$ can be accessed in $O(N \log \sigma)$ time and $GS$ can be executed in $O(1)$ time. □

### 3.2 Multi-track Horspool algorithm

MT-H in Algorithm 3 uses $BC$ to shift the pattern that can be computed in the same way as $BC$ of MT-BM shown in Algorithm 1.

**Theorem 9.** *Given a multi-track text $\mathbb{T}$ and a pattern $\mathbb{P}$, MT-H outputs the positions of the text that are permuted-matched with the pattern in $O(nN(m+\log \sigma)+n(N+\sigma))$ time in the worst case with $O(m(M \log \sigma + \sigma))$ time preprocessing.*

*Proof.* Similar to the proof of Theorem 8, beside MT-H uses $BC$ only. □

### 3.3 Boyer-Moore and Horspool matching algorithms with track-trie

The two algorithms presented in the previous subsections decide if two multi-tracks permuted-match by sorting them. In this subsection, we present another idea for this

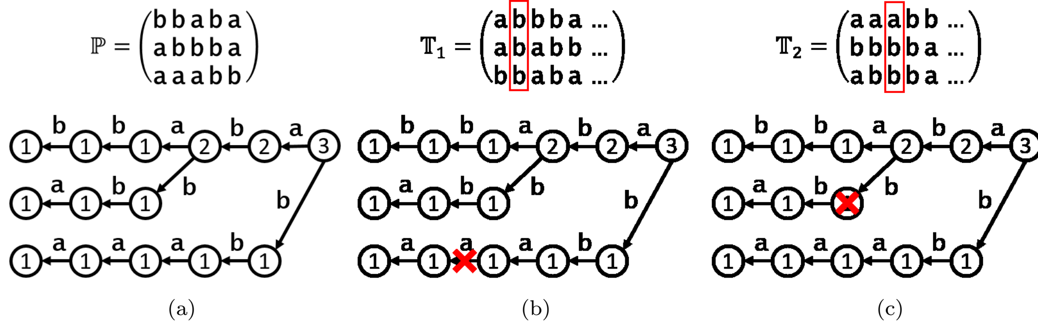



**Figure 1.** (a) Track trie of $\mathbb{P} = (\texttt{bbaba}, \texttt{abbba}, \texttt{aaabb})$, (b) example of mismatch when the track trie cannot find the transition, (c) example of mismatch when the number of tracks is more than the weight of the node.

task using a data structure called a *track trie*. The track trie of a multi-track $\mathbb{P}$ stores all the reversed strings of the tracks of $\mathbb{P}$, that is, $\{p_1^R, p_2^R, \ldots, p_M^R\}$. Fig. 1(a) shows the track trie of a multi-track pattern $\mathbb{P} = (\texttt{aaabb}, \texttt{abbba}, \texttt{bbaba})$.

Algorithm 4 is the construction algorithm for the track-trie of $\mathbb{P}$. For a node $s$ of the track trie and a character $c \in \Sigma$, the goto function $goto(s, c)$ returns the child of $s$ that has an edge labeled $c$. We naturally extend it to the domain $\Sigma^*$ by $goto(s, \varepsilon) = s$ and $goto(s, aw) = goto(goto(s, a), w)$ for any $a \in \Sigma$ and $w \in \Sigma^*$. We also associate a weight with each node to find mismatch on a text, as we will explain later.

**Theorem 10.** *Algorithm 4 constructs the track-trie of* $\mathbb{P}$ *in* $O(mM \log \sigma)$ *time.*

*Proof.* The function *goto* can be calculated in $O(\log \sigma)$ time by binary search. On each execution of the inner **for** loop (line 5), Algorithm 4 executes *goto* to check child nodes of *activeNode*. If there is no node with an edge labeled $\mathbb{P}[j][i]$, then a new node is constructed, which can be done in $O(1)$ time. On the other hand, if there is a node with an edge labeled $\mathbb{P}[j][i]$, Algorithm 4 accesses the child node and then increases its weight by one. The total number of iterations of the inner loop is $mM$. □

For a given multi-track text $\mathbb{T}$ and a position $i$, Algorithm 5 finds a mismatch position in two cases; (1) when a track cannot find its *goto* destination, and (2) when the number of tracks that have the same string $w$ is more than the weight of the node that represents the string $goto(root, w)$. Those mismatch conditions are illustrated in Fig. 1 (b) and (c), respectively. Fig. 1 (b) shows that the track trie cannot find a transition for the second character b of the third track. On the other hand, Fig. 1 (c) shows that $\mathbb{T}_2[3:]$ has two 'bba' on its track, however the $\mathbb{P}[3:]$ has only one 'bba' on its track, i.e. the node that represents 'bba' has one on its weight.

**Theorem 11.** *Given a multi-track text* $\mathbb{T}$ *and a position* $j$*, Algorithm 5 finds a mismatch position in the pattern in* $O(mM \log \sigma)$ *time.*

*Proof.* For each position $i+j$ on the text, Algorithm 5 executes *goto* to check whether *activeNodes*$[k]$ has a child node with an edge labeled $\mathbb{T}[i+j][k]$ for $1 \leq k \leq M$. If there is no child node with an edge labeled $\mathbb{T}[i+j][k]$, then Algorithm 5 considers it as mismatch and returns the mismatch position. On the other hand, if there is such a child node, Algorithm 5 changes *activeNodes*$[k]$ to the child node, and then check whether the number of tracks of $\mathbb{T}[i+j:]$ that contain $\mathbb{T}[k][i+j:i+m]$ as a prefix is more than the weight of the child node. If the number of tracks exceeds the weight, then Algorithm 5 treats it as mismatch and returns the mismatch position. The total number of iterations of the inner loop is at most $mM$. □

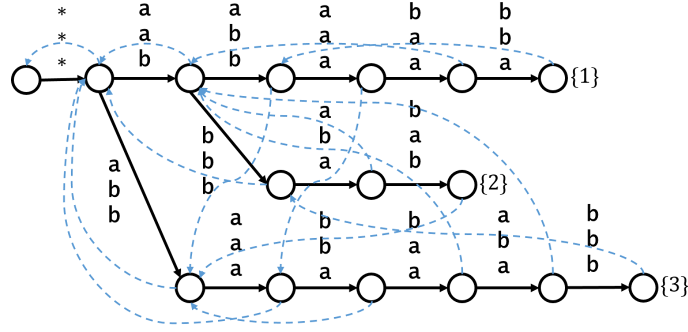



**Figure 2.** Multi-track AC-automaton of $D = \{\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3\}$, where $\mathbb{P}_1 = (\mathtt{aaabb}, \mathtt{abaab}, \mathtt{bbaaa})$, $\mathbb{P}_2 = (\mathtt{abab}, \mathtt{abba}, \mathtt{bbab})$, and $\mathbb{P}_3 = (\mathtt{aabbab}, \mathtt{bababb}, \mathtt{baaaab})$. The asterisk '*' is a special character that matches with any characters in $\Sigma$.

**Algorithm 6:** Multi-track AC-automaton goto function and initial output function construction algorithm

**Input:** Set of multi-track patterns $D = \{\mathbb{P}_1, \mathbb{P}_2, ..., \mathbb{P}_r\}$
**Output:** Goto function and initial output function

1 compute $SI(\mathbb{P}_i[j:])$ for $1 \le i \le r$ and $1 \le j \le m_i$;
2 create states $rootState$ and $\perp$;
3 $goto(\perp, \mathbb{W}) \leftarrow rootState$ for all multi-track character $\mathbb{W} \in \Sigma^M$;
4 $newState \leftarrow rootState$;
5 **for** $i \leftarrow 1$ **to** $r$ **do**
6   $activeState \leftarrow rootState$;
7   **for** $1 \le j \le m_i$ **do**
8     **if** $goto(activeState, \mathbb{P}_i[j]\langle SI(\mathbb{P}_i[1:])\rangle) \ne$ **fail then**
9       $activeState \leftarrow goto(activeState, \mathbb{P}_i[j]\langle SI(\mathbb{P}_i[1:])\rangle)$;
10     **else**
11       $newState \leftarrow newState + 1$;
12       $goto(activeState, \mathbb{P}_i[j]\langle SI(\mathbb{P}_i[1:])\rangle) \leftarrow newState$;
13       $label(newState) \leftarrow i$;
14       $activeState \leftarrow newState$;
15     **if** $k = m_i$ **then**
16       $output(activeState) \leftarrow output(activeState) \cup \{\mathbb{P}_i\}$;

Although the worst case time complexity remains the same, by using track-trie, both MT-BM and MT-H can match the pattern to the text practically faster, because we do not need to compute the reverse sorted index of the text. First, we construct the track-trie of the pattern by using constructTrackTrie($\mathbb{P}$). Then, we replace line 8 (resp. line 7) of Algorithm 2 (resp. Algorithm 3) by matchTrackTrie($\mathbb{T}, j$) to find a mismatch position.

## 4 Multi-track AC-automaton

In this section, we will explain a data structure called a multi-track AC-automaton that can perform dictionary matching on multi-tracks. Given a set $D = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_r\}$ of multi-track patterns called a *dictionary* and a multi-track text $\mathbb{T}$, by preprocessing the patterns, the multi-track AC-automaton can find all occurrence positions of each pattern in the text. Let $d = \Sigma_{i=1}^{r} m_i$ be the total length of the patterns in $D$, where

**Algorithm 7:** Multi-track AC-automaton failure function and output function construction algorithm.

**Input:** Goto function and initial output function
**Output:** Failure and output functions

1 compute $SI(\mathbb{P}_i[j:])$ for $1 \leq i \leq r$ and $1 \leq j \leq m_i$;
2 $failure(rootState) \leftarrow \bot$;
3 push $rootState$ to $queue$;
4 **while** $q \neq empty$ **do**
5 pop $activeState$ from $queue$;
6 **for** $a$ such that $goto(activeState, a) = s \neq$ **fail do**
7 push $s$ to $queue$;
8 $state \leftarrow failure(activeState)$;
9 $j \leftarrow label(s)$;
10 **while** $goto(state, \mathbb{P}_j[depth(s)]\langle SI(\mathbb{P}_i[depth(s) - depth(state):])\rangle) =$ **fail do**
11 $state \leftarrow failure(state)$;
12 $failure(s) \leftarrow goto(state, \mathbb{P}_j[depth(s)]\langle SI(\mathbb{P}_i[depth(s) - depth(state):])\rangle)$;
13 $output(s) \leftarrow output(s) \cup output(failure(s))$;

**Algorithm 8:** multi-track AC-automaton matching algorithm

**Input:** Goto, failure and output functions
**Output:** Set of (pattern,position) tuple $\{(\mathbb{P}_{k_1}, pos_1), (\mathbb{P}_{k_2}, pos_2), ...\}$

1 compute $SI(\mathbb{T}[i:])$ for $1 \leq i \leq n$;
2 $activeState \leftarrow rootState$;
3 **for** $i = 1$ **to** $n$ **do**
4 **while** $goto(activeState, \mathbb{T}[i]\langle SI(\mathbb{T}[i - depth(activeState) + 1:])\rangle) =$ **fail do**
5 $activeState \leftarrow failure(activeState)$;
6 $activeState \leftarrow goto(activeState, \mathbb{T}[i]\langle SI(\mathbb{T}[i - depth(activeState) + 1:])\rangle)$;
7 **for** $k \in output[activeState]$ **do output** $(k, i - m_k + 1)$ ;

$m_i = |\mathbb{P}_i|_{len}$. The multi-track AC-automaton of $D$, denoted by $MTAC(D)$, consists of three functions; *goto*, *failure*, and *output* functions.

Unlike the original AC-automaton, the multi-track AC-automaton uses a multi-track character, instead of a single character to define *goto*. The states and *goto* in $MTAC(D)$ construct a trie of $sort(\mathbb{P}_i)$ for all $\mathbb{P}_i \in D$. Each state in $MTAC(D)$ represents a prefix of $sort(\mathbb{P}_i)$, thus each state can be denoted by $S(\mathbb{W})$, where $\mathbb{W}$ is the string obtained by concatenating the labels of the edges from the root to the state. Therefore, we can define $goto(S(\mathbb{P}_i[: j]), \mathbb{P}_i[j+1]) = S(\mathbb{P}_i[: j+1])$ for $1 \leq i \leq r$ and $1 \leq j < m_i$. For convenience, we denote $goto(goto(s, \mathbb{P}_i[i]), \mathbb{P}_i[i+1])$ as $goto(s, \mathbb{P}_i[i : i+1])$, and $goto(goto(s, \mathbb{P}_i[j : k-1]), \mathbb{P}_i[k])$ as $goto(s, \mathbb{P}_i[j : k])$. For a state $s$ and a multi-track character $\mathbb{C}$, $goto(s, \mathbb{C})$ can be implemented by using multi-track character trie of depth at most $M$ nodes, thus $goto(s, \mathbb{C})$ can be executed in $O(M \log \sigma)$ time. The function *goto* can be constructed by using Algorithm 6.

Next, the failure function of a state $S(\mathbb{P}_i[: j])$ is defined as $flink(S(\mathbb{P}_i[: j])) = S(sort(\mathbb{P}_i[k : j]))$, where $\mathbb{P}_i[k : j]$ is the longest proper suffix of $\mathbb{P}_i[: j]$ such that $\mathbb{P}_i[k : j]$ is a prefix of some $sort(\mathbb{P}_\ell)$ with $\mathbb{P}_\ell \in D$. Algorithm 7 shows a construction algorithm for the failure function of a multi-track AC-automaton.

Finally, the output function of the multi-track AC-automaton is similar to the original AC-Automaton. For a state $S(\mathbb{P}_i[: j])$, the output of the state

$output(S(\mathbb{P}_i[: j]))$ is the set of patterns $\mathbb{P}_\ell \in D$ such that $\mathbb{P}_\ell \cong \mathbb{P}_i[k : j]$ for some $1 \le k \le j$. The initial output function is constructed by Algorithm 6, and then updated by Algorithm 7 to get the final output function. Fig. 2 shows an example of $MTAC(D)$. In order to simplify the construction algorithm, we use a special state that reads any multi-track character to get to the root state.

**Theorem 12.** *Algorithm 6 constructs the goto and initial output functions of a set* $D = \{\mathbb{P}_1, \mathbb{P}_2, \dots, \mathbb{P}_r\}$ *of multi-track patterns in* $O(dM \log \sigma)$ *time.*

*Proof.* The total number of executions of *goto* is $d = \Sigma_{i=1}^r m_i$ and $goto(s, \mathbb{C})$ is executed in $O(M \log \sigma)$ time. □

**Theorem 13.** *Algorithm 7 constructs the failure and output functions of a set* $D = \{\mathbb{P}_1, \mathbb{P}_2, \dots, \mathbb{P}_r\}$ *of multi-track patterns in* $O(dM \log \sigma)$ *time.*

*Proof.* Updating the output function can be performed in $O(1)$ time by using list to save the output function, and update it by concatenate the list. Therefore, we can bound the running time of Algorithm 7 by counting the number of executions of *goto*. For each pattern $\mathbb{P}_i$, let $s_{i,j}$ be a state such that $s_{i,j} = goto(root, \mathbb{P}_i[: j])$ for $1 \le j \le m_i$. Let $f_{i,j}$ be the number of executions of *failure* when finding $failure(s_{i,j})$. The maximum value of $f_{i,j}$ is bounded by $depth(failure(s_{i,j-1})) + 1$. Because the depth of $failure(s_{i,j})$ is at most $depth(failure(s_{i,j-1})) - f_{i,j} + 1$, we get $f_{i,j} \le depth(failure(s_{i,j-2})) - f_{i,j-1} + 2$ recursively. By solving this formula, we get $\Sigma_{j=1}^{m_i} f_{i,j} \le 2m_i$, and $\Sigma_{i=1}^r \Sigma_{j=1}^{m_i} f_{i,j} \le \Sigma_{i=1}^r 2m_i = 2d$. Moreover, each *goto* is executed in $O(M \log \sigma)$ time. □

By using the goto, output, and failure functions, the multi-track AC-automaton can perform permuted pattern matching on a text $\mathbb{T}$ as shown in Algorithm 8. Let *activeState* be the current state of the multi-track AC-automaton and $d$ be the depth of *activeState*. For each position $i$ on $\mathbb{T}$, Algorithm 8 uses the sorted index of $\mathbb{T}[i-d :]$ to determine permutation of $\mathbb{T}[i]$ used in the goto function.

**Theorem 14.** *Algorithm 8 performs permuted pattern matching on a multi-track text* $\mathbb{T}$ *in* $O(nN \log \sigma)$ *time.*

*Proof.* The running time of Algorithm 8 can be evaluated by counting the number of executions of *goto*. First, for each $i$, *goto* is executed at least once on *activeState* transition. Next, *goto* is executed to check whether the transition is **fail** or not. In this case, the number of executions of *goto* is the same as that of *failure*. The latter is at most $n$, because whenever *goto* is executed, the depth of *activeState* is increased by one, and whenever *failure* is executed, the depth of *activeState* is decreased by at least one. Therefore, the number of executions of *goto* is $O(n)$. □

## 5 Multi-track permuted matching automaton

In this section, we will describe a data structure called a multi-track permuted matching automaton that can perform permuted pattern matching on a multi-track text $\mathbb{T}$ online, by preprocessing a multi-track pattern $\mathbb{P}$. A multi-track permuted matching automaton is constructed by two functions, *goto* and *failure*. In addition, similarly to a track-trie, each state of the multi-track permuted matching automaton has a

**Algorithm 9:** Multi-track permuted matching automaton goto function construction algorithm

**Input:** Multi-track $\mathbb{P}$
**Output:** Goto function
1. create states *rootState* and $\bot$;
2. $goto(\bot, w) \leftarrow rootState$ for all character $w \in \Sigma$;
3. $newState \leftarrow rootState$;
4. $weight(\bot) \leftarrow weight(rootState) \leftarrow M$;
5. **for** $i \leftarrow 1$ **to** $M$ **do**
6.     $activeState \leftarrow rootState$;
7.     **for** $1 \leq j \leq m$ **do**
8.         **if** $goto(activeNode, \mathbb{P}[j][i]) =$ **Null then**
9.             $newState \leftarrow newState + 1$;
10.             $weight(newState) \leftarrow 1$;
11.             $activeState \leftarrow newState$;
12.         **else**
13.             $activeState \leftarrow goto(activeState, \mathbb{P}[j][i])$;
14.             $weight(activeState) \leftarrow weight(activeState) + 1$;
15.         **if** $k = m$ **then** set *activeState* as an accept state ;

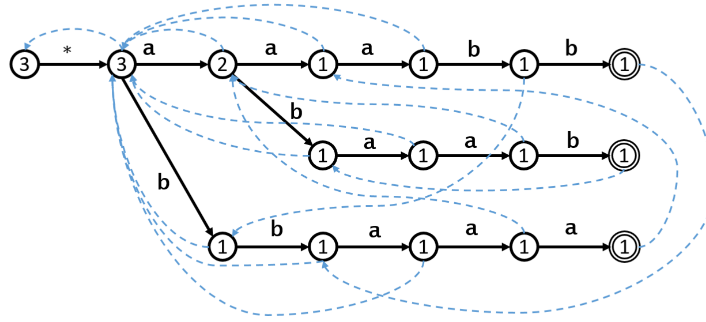


**Figure 3.** Multi-track permuted matching automaton of $\mathbb{P} = (\texttt{aaabb}, \texttt{abaab}, \texttt{bbaaa})$. The asterisk '*' is a special character that matches with any characters in $\Sigma$.

weight in order to determine whether *failure* should be executed or not. Fig. 3 shows an example of a multi-track permuted matching automaton.

For a multi-track pattern $\mathbb{P} = (p_1, p_2, ..., p_m)$, the multi-track permuted matching automaton of the pattern is denoted by $MTPMA(\mathbb{P})$. The goto function of the multi-track permuted matching automaton is similar to that of an AC-automaton, thus, each state in $MTPMA(\mathbb{P})$ represents a prefix of $p_i$, which is denoted by $S(w)$, where $w$ is the string obtained by concatenating the labels of the edges from the root to the state. Each state $S(w)$ has a weight, which is a number of tracks containing $w$ as a prefix. Moreover, a state $S(w)$ is called an *accept state* if $w = p_i$ for some $i$. Algorithm 9 constructs the goto function of a multi-track permuted matching automaton.

**Theorem 15.** *Algorithm 9 constructs the goto function of a multi-track pattern* $\mathbb{P}$ *in* $O(mM \log \sigma)$ *time.*

*Proof.* For each track, the number of executions of *goto* is $m$ and there are $M$ tracks in a pattern $\mathbb{P}$. Moreover, *goto* can be executed in $O(\log \sigma)$ time. □

Next, we will define the failure function of a multi-track permuted matching automaton. Let $S_j$ be the set of states that have depth $j$ and $S(p_i[: j]) \in S_j$ be a state

**Algorithm 10:** Multi-track permuted matching automaton failure function construction algorithm

```
   Input: Multi-track ℙ, goto function
   Output: Failure function
 1 activeStates[i] ← rootState for 1 ≤ i ≤ M;
 2 failure(rootState) ← ⊥;
 3 for i ← 1 to m do
 4     tempStates ← activeStates;
 5     failFlag ← true;
 6     while failFlag = true do
 7         failFlag ← false;
 8         vector failStates;
 9         for j ← 1 to |tempStates| do failStates[j] ← failure(tempStates[j]) ;
10         failFlag ← isFail(activeStates, failStates);
11         if failFlag = true then tempStates ← failStates ;
12         else
13             for j ← 1 to |activeStates| do
14                 for a such that goto(activeStates[j], a) = s ≠ fail do
15                     failure(s) ← goto(failStates[j], a);
16     clear tempStates;
17     for j ← 1 to |activeStates| do
18         for a such that goto(activeStates[j], a) = s ≠ fail do tempStates.add(s) ;
19     activeStates ← tempStates;
20 Function isFail(activeStates, failStates)
21     for j ← 1 to |activeStates| do
22         for a such that goto(activeStates[j], a) = s ≠ fail do
23             if goto(failStates[j], a) = fail then return true ;
24             else
25                 nextState ← goto(failStates[j], a);
26                 temp(nextState) ← temp(nextState) + weight(s);
27                 if temp(nextState) > weight(nextState) then return true ;
28     return false;
```

of depth $j$. The failure function of the state is $failure(S(p_i[: j])) = S(p_k[: \ell]) \in S_\ell$ such that $p_k[: \ell]$ is a proper suffix of $p_i[: j]$ and $\mathbb{P}[: \ell]$ permuted matches with a suffix of $\mathbb{P}[: j]$. Note that the definition of this failure function is similar to that of the multi-track KMP algorithm introduced in [3].

Algorithm 10 constructs the failure function of a multi-track permuted matching automaton. We use a state pointer for each track in the pattern. Similarly to a track-trie, there are two conditions that are considered as failure in a multi-track permuted matching automaton. The first condition is when it cannot find the goto transition, and the second condition is when the number of state pointers in the state is more than the weight of the state.

**Theorem 16.** *Algorithm 10 constructs the failure function of a multi-track permuted matching automaton in $O(mM \log \sigma)$ time.*

**Algorithm 11:** Multi-track permuted matching automaton matching algorithm

```
Input: goto and failure functions
Output: Permuted match positions
1  activeStates[i] ← rootState for 1 ≤ i ≤ N;
2  for 1 ≤ i ≤ n do
3      failFlag ← true;
4      while failFlag = true do
5          failFlag ← false;
6          failFlag ← isFail(activeStates, 𝕋, i);
7          if failFlag = true then
8              for j = 1 to N do activeStates[j] ← failure(activeStates) ;
9          else
10             for j = 1 to |activeStates| do activeStates[j] ← goto(activeStates[j], 𝕋[i][j]) ;
11     if activeStates[1] is an accept state then output i − m + 1 ;
12 Function isFail(activeStates, 𝕋, i)
13     for j = 1 to N do
14         if goto(activeStates[j], 𝕋[i][j]) = fail then return true ;
15         else
16             nextState = goto(activeStates[j], 𝕋[i][j]);
17             temp(nextState) = temp(nextState) + 1;
18             if temp(nextState) > weight(nextState) then return true ;
19     return false;
```

*Proof.* Similar to the proof of Theorem 13, the failure and goto functions are executed $O(mM)$ times. Moreover, execution time of the failure function is $O(1)$ and that of the goto function is $O(\log \sigma)$. □

Finally, by using the goto and failure functions, Algorithm 11 can perform permuted pattern match on a multi-track text $\mathbb{T}$. Algorithm 11 uses $N$ pointers *activeStates* to point the current states. Note that all *activeStates* always have the same depth. Similarly to Algorithm 10, Algorithm 11 also uses two conditions to determine whether it should execute the failure function or not. If any of the *activeStates* is fail, then all of the *activeStates* execute the failure function, otherwise *activeStates* execute the goto function.

**Theorem 17.** *Algorithm 11 performs permuted pattern match on a multi-track string* $\mathbb{T}$ *in* $O(nN \log \sigma)$ *time.*

*Proof.* Similarly to the proof of Theorem 14, the number of executions of the failure and goto function is $O(nN)$. Since the execution time of the failure function is $O(1)$ and the goto function is $O(\log \sigma)$, Algorithm 11 runs in $O(nN \log \sigma)$ time. □

## 6 Experiments

We evaluate performance of our algorithms by conducting experiments on full-permuted pattern matching. We compared the running time of our algorithms with existing algorithms, AC automaton based algorithm [6] and KMP based algorithm [3]. We ran the algorithms on a computer with Intel Xeon CPU E5-2609 8 cores 2.40 GHz, 256 GB memory, and Debian Wheezy operating system.

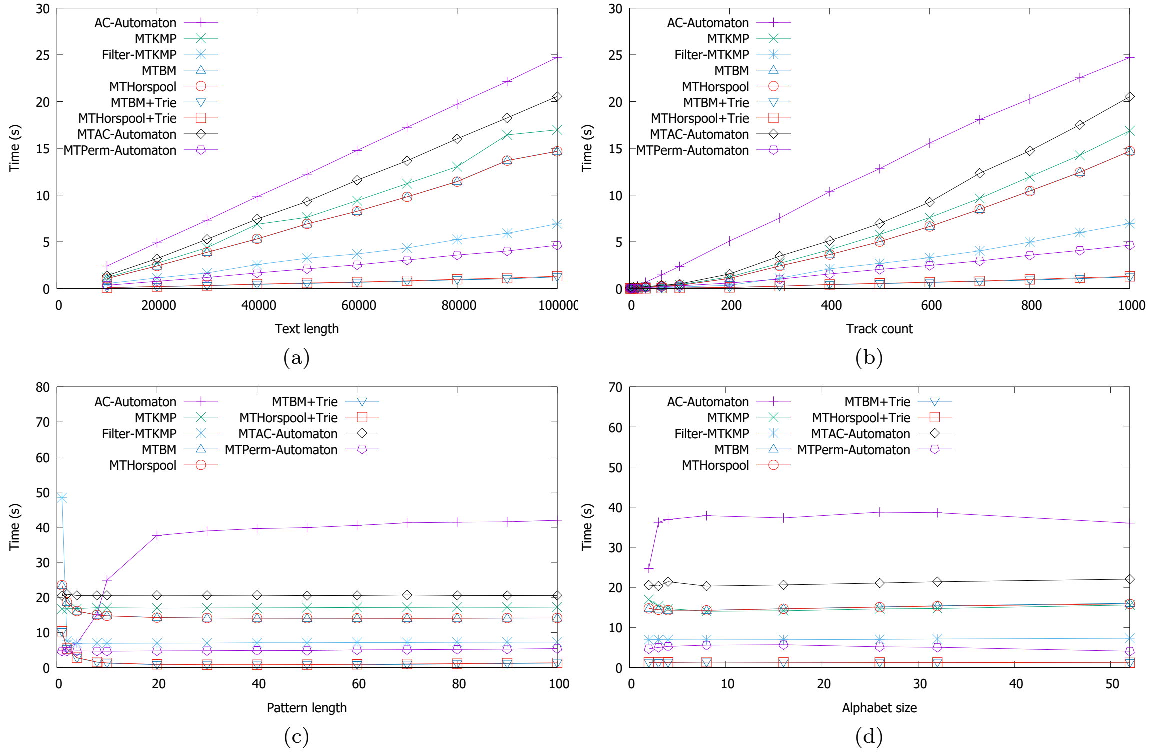



**Figure 4.** Running time of the algorithms on full-permuted pattern matching with respect to (a) text length, (b) track count, (c) pattern length, and (d) alphabet size.

We set the parameter values as follows, $n = 100000$, $m = 10$, $N = M = 1000$, and $\sigma = 2$, and changed one of the parameters in each experiment to see the running time of the algorithms with respect to the parameters. We used randomly generated texts and patterns, and inserted 50 occurrences of a pattern into each text to make sure that there are occurrences of the pattern in the text.

The result of the experiments are shown in Fig. 4 (a)–(d), where one of the parameters $n$, $N$, $m$, and $\sigma$ is changed respectively. First, we can see that the running time of the algorithms increase linearly with respect to the length and track count of the text, and is not much affected by the pattern length or the alphabet size. The running times of MT-BM and MT-H are almost the same, and the running times of these algorithms are faster when a track-trie is used.

Multi-track AC-automaton is slower than the MTKMP algorithm on a single pattern matching, although it can support dictionary matching on multi-track strings. We can also see that multi-track permuted matching automaton runs faster than the MTKMP and Filter-MTKMP algorithms, as it is an improvement of MTKMP algorithm.

# 7 Concluding remarks

In this paper, we focused on *full* permuted pattern matching problems, where the track count $N$ of a text equals to the track count $M$ of a pattern. In general, the permuted pattern matching problem is more difficult if $N > M$. For example, when we construct $GS[i]$ for the full-permuted pattern matching problem, we compute the minimum value of $s$ such that $\mathbb{P}[i - s + 1 : m - s] \stackrel{\bowtie}{=} \mathbb{P}[i + 1 : m]$, because we know that if a substring $\mathbb{T}[j : j + m - i - 1]$ of the text does not match with $\mathbb{P}[i + 1 : m]$,

then $\mathbb{T}[j : j + m - i - 1] \ncong \mathbb{P}[i - k + 1 : m - k]$ for $0 < k < s$. However, in the case where $N > M$, there is a possibility that $\mathbb{P}[i - k + 1 : m - k]$ matches with the $\mathbb{T}[j : j + m - i - 1]$, and we might miss the occurrences of the pattern if we use the same shift as in the case of full permuted pattern matching. This problem is also arises in multi-track AC-automaton and multi-track permuted matching automaton when we try to construct the failure function. We should find another condition to define the failure function for these algorithms.

**Acknowledgments** This work is supported by Tohoku University Division for Interdisciplinary Advance Research and Education, JSPS KAKENHI Grant Numbers JP15H05706, JP24106010, and ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan).

## References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
3. Diptarama, Y. Ueki, K. Narisawa, and A. Shinohara: *KMP based pattern matching algorithms for multi-track strings*, in Proceedings of Student Research Forum Papers and Posters at SOFSEM 2016, 2016, pp. 100–107.
4. A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo: *Position heaps: A simple and dynamic text indexing data structure.* Journal of Discrete Algorithms, 9(1) 2011, pp. 100–121.
5. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
6. T. Katsura, K. Narisawa, A. Shinohara, H. Bannai, and S. Inenaga: *Permuted pattern matching on multi-track strings*, in SOFSEM, 2013, pp. 280–291.
7. T. Katsura, Y. Otomo, K. Narisawa, and A. Shinohara: *Position heaps for permuted pattern matching on multi-track strings*, in Proceedings of Student Research Forum Papers and Posters at SOFSEM 2015, 2015, pp. 41–531.
8. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
9. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in CPM, 2003, pp. 200–210.
10. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
11. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
12. P. Weiner: *Linear pattern matching algorithms*, in SWAT, 1973, pp. 1–11.

# Using Human Computation in Dead-zone based 2D Pattern Matching

Kamil Awid[1,2], Loek Cleophas[1,3], and Bruce W. Watson[1,2]

[1] FASTAR Research Group, Department of Information Science
Stellenbosch University, Republic of South Africa
[2] Centre for Artificial Intelligence Research
CSIR Meraka Institute, Republic of South Africa
[3] Natural and Formal Languages Group, Department of Computer Science
Umeå University, Sweden
{kamil, loek, bruce}@fastar.org

**Abstract.** This paper examines the application of human computation (HC) to two-dimensional image pattern matching. The two main goals of our algorithm are to use *turks* as the processing units to perform an efficient pattern match attempt on a subsection of an image, and to divide the work using a version of *dead-zone* based pattern matching. In this approach, human computation presents an alternative to machine learning by outsourcing computationally difficult work to humans, while the dead-zone search offers an efficient search paradigm open to parallelization—making the combination a powerful approach for searching for patterns in two-dimensional images.

**Keywords:** image pattern matching, dead-zone pattern matching, human computation

## 1 Introduction, motivation, and related work

In human computation, humans are used for tasks for which humans are more suitable than computers, i.e. they are human processing units, typically called *turks* in this context. We consider the problem of utilizing turks in the search for an object inside of a matrix of objects. To be more precise, this paper examines the utilization of turks in the search for occurrences of an image (the *pattern image* or *pattern*) in a larger image (the *subject image* or *subject*). This presents a two-fold problem: 1) efficiently exploring and dividing the search space (i.e. the image); and 2) utilizing a turk to check whether the pattern actually occurs in a specific area of the image. The algorithm that is to be used for the search is an adaptation of a dead-zone based pattern matching algorithm [7]; the new algorithm generalizes from this in that it performs a search on a *two-dimensional* matrix instead of on a *one-dimensional* array of symbols. The use of human computation allows more powerful searching that otherwise may be very difficult to solve with pure computational algorithms since humans easily recognize images that have been rotated, scaled, sheared, images with alternative colours, and vague patterns.

### 1.1 Human Computation

Human computation (HC) provides a mechanism for solving problems using humans as an alternative to using concepts of machine learning (ML) and artificial intelligence (AI). Human computation relies on a series of so-called *turks* which juxtapose computer processing units for processing information. Using HC it is possible to solve a

Kamil Awid, Loek Cleophas, Bruce W. Watson: Using Human Computation in Dead-zone based 2D Pattern Matching, pp. 22–32.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

myriad of problems that are trivial for humans yet baffle sophisticated programs [4]. The majority of these problems lie outside of what ML and AI can currently solve. Captcha [11] and more generally the Amazon Turk [6] service both attempt to solve problems that would be otherwise difficult to solve computationally.

The goal of this paper is to provide an alternative rather than a replacement for machine learning in the development of a 2-dimensional search algorithm. The two concepts are orthogonal and can be used in conjunction with each other, however, that is outside the scope of this paper. Due to the multidisciplinary nature of human computation, a number of considerations have to be taken into account at various tiers including high level design, algorithms, and human-computer interaction and other human aspects.

The work in 2-dimensional image search can be applied to satellite imaging data where turks can classify objects on a map and perhaps even train machine learning models. Letting a single turk scan such imaging data (possibly gigabytes of imaging data) may be quite cumbersome and prevents parallelism and verification. This work in this paper is motivated by such examples. The aim is to use humans and the computer to distribute tasks based on their respective strengths efficiently.

Work in HC has been approached from multiple directions. Researchers from MIT CSAIL have developed a javascript library to deal with the intricacies of HC [8] enabling easy parallelism, crash-and-rerun programming, and ease of implementation. In case a human computation program encounters an error, it is important to recover since external calls are expensive (i.e. re-running the entire task with a turk may cost time or money); a crash-and-rerun program mitigates this problem by recording computationally expensive information and allowing the use of this information to rerun from the last known point. Well known functions for parallelism such as fork and join are available through this toolkit.

Luis Van Ahn dealt with problems in motivation (i.e. monetary motivation and game theory), interfaces, and algorithms in his PhD thesis [4]. Other research in HC includes task routing [1], combining human and machine intelligence [2], parallelization and design patterns [3]. Additionally, Amazon has developed an environment where the developers can utilize human computation through their Amazon Turk service [6]. While Amazon provides an interface for connecting with turks, consideration still has to be given to development of efficient and coherent algorithms optimized for processing by humans (i.e. turks).

HC search algorithms allow the matching flexibilities of a human, and therefore the inputs can range across many object types, including sounds, pictures, or strings. With such flexibility, however, the fuzziness of the output increases. This may have beneficial and adverse effects depending on the problem at hand. In the case of a search algorithm, the uncertainty revolves about incorrect pattern matching and not completing assigned tasks. These issues can be mitigated through parallelizing [9] the work through different turks and using voting [10] or statistical methods to decide whether the answer is correct. There are other ways to mitigate output errors by rephrasing the problem in a way that results in the capture of natural human instincts in solving a problem [5]. Parallelization, however, lets us measure the degree of confidence of the final result by taking a sample of human outputs. Additionally, parallelization reduces a dependency on a single turk, reducing the amount of time it takes to solve a problem e.g. if a particular turk is unavailable at the moment.

### 1.2 Dead-zone pattern matching

Dead-zone (DZ) pattern matching [7] is an approach for string pattern matching—finding all occurrences or matches of a pattern string $p$ in a larger string or text $S$. In a nutshell, DZ algorithms start from a situation in which a single *live-zone*—the entire text $S$—exists, and select a pivot in such a live-zone. They then proceed in checking whether an actual match of $p$ occurs there—if so, this match is reported. Based on the information gathered during this checking, the algorithm can *dead-zone* particular areas to the right and left of the pivot—preventing unnecessary further match attempts in these areas, and splitting the live-zone into two separate smaller ones. It repeatedly processes such a live-zone (or multiple ones in parallel), until a situation is reached where no live-zone remains, and all of the text is dead, with all pattern occurrences having been reported.

Each algorithm from the DZ family is easily parallelized and therefore especially useful in the field of human computation where parallelization is necessary to have the same tasks processed by multiple turks in order to deal with uncertainty.

*2-dimensional pattern matching* is about finding occurrences or *matches* of a 2-dimensional pattern $p$ in a 2-dimensional symbol matrix $S$. Figure 1 is a representation of what a 2-dimensional search algorithm tries to accomplish. The dead-zone algorithm starts a pattern match attempt in the middle of $S$. If a match is not found, the algorithm proceeds to shift in 4 directions using the data obtained during the match attempt. Once the shifting is complete and the dead-zones have been determined, the algorithm divides the matrix into 8 areas and recurses into each zone.

auvwxyz1234
babc~~de12~~345
cfgh~~i~~j5~~6~~789
dklm~~n~~o1~~2~~345
epqr~~st67~~891
fuvwxyz1234

**Figure 1.** Symbol matrix **S** with occurrence of a 2x2 square pattern $p$ (j5o1), dead-zone drawn around (struck-through text), and 8 areas created subsequently from the dead-zones (top, top-left, top-right, middle-left, etc.)

## 2 An algorithm for 2-dimensional dead-zone matching using human computation

Our new algorithm processes 2-dimensional data in the form of an image. The algorithm below matches an image contained in a larger image. Humans are best utilized in the processing of generalized problems i.e. problems that avoid detailed information, since the end result may be an approximation - while a turk may have a hard time recognizing pixels on the screen, he or she can certainly discern whether pictures made out of these pixels are similar. Therefore, the 2-dimensional data used for our algorithm is in the form of an image instead of other symbols. Nonetheless, the algorithm can be applied to any matrix of symbols.

The algorithm relies on dead-zoning a part of the image, then proceeding to shifting, dividing and delegating the remaining work to turks. This approach allows the smaller instances of the problem (smaller live-zones) to be easily parallelized between numerous turks.

The algorithm uses the TurKit algorithms developed at MIT CSAIL [8]. Namely, we are using crash-and-rerun concepts, fork (to allow parallel processing), createHIT (to create our task for the turk), and voting (determine whether the turks agree on the results). TurKit utilizes HTML to generate interfaces.

```
1
2  function human_2d_dz(live_low, live_high)
3  {
4    if(<pattern larger than livezone>)
5      return null;
6
7    draw_viewport(live_low, live_high);
8    pattern_found =human_search(); // Turk's work
9
10   if(pattern_found)
11   {
12     console.log("Match at " + live_low + " " + live_high);
13     fork(function(){
14       // vote on the correctness of the result
15       vote_result =mturk_vote(pattern_found, ..);
16     })
17
18   }
19
20
21   //Expand zone and let user estimate whether there is a possibility of the
         pattern occurring on any of the sides of the viewport.
22   //Let the user indicate how far they shift in the image to create deadzones.
23   new_deadzone =expand_deadzone(live_low, live_high);
24
25   live_zones =create_live_zones(new_deadzone)
26   for(zone in live_zones)
27   {
28     fork(function(){
29         human_2d_dz(
30           live_zones[zone].live_low,
31           live_zones[zone].live_high
32         )
33     });
34     join() ;
35   }
36
37 }
38
39 //Example of a human search function utilizing TurKit
40 function human_search()
41 {
42   var hitId =mturk.createHIT({
43           title : "Find possible image zone",
44           desc : "Indicate whether the pattern is found in the viewport.",
45           url : votePage,
46           height : 800,
```

```
47          reward : 0.01,
48    })
49
50    return mturk.waitForHIT(hitId).assignments[0].answer.found;
51  }
```

**Listing 1.1.** 2D DZ algorithm using TurKit

## 2.1 Pattern Match

The algorithm utilizes a viewport (Figure 2) to deal with a particular subsection of the image to be searched. The viewport is a rectangular region, just like the image. The procedure *draw_viewport* (Line 7) takes the entire image and the live zone (indicated by 2D coordinates *live_low* and *live_high*) and creates a viewport that is to be processed by *human_search* (Line 8). After processing by a turk, *human_search* returns a boolean flag to indicate whether a pattern occurrence was found.

**Figure 2.** Viewport displayed to the user.

The turk is initially presented with the viewport centred in the middle of the original image $s$. In the case that the turk finds the pattern $p$ completely inside the viewport, the algorithm returns the position of the image inside $s$. In any case, the program continues to search for other occurrences of the pattern.

The viewport has size $p$ to keep the result of the turk operations consistent with the problem: the pattern found must fit within the viewport i.e. patterns larger than the viewport will be ignored.

When the viewport completely overlays existing dead-zones, we can be confident that the pattern of size of $p$ cannot be found inside this viewport. In the case of scaled patterns, we cannot confidently state that the pattern does not reside in a viewport partially obscured by a dead-zone, preventing the algorithm from skipping the area to be examined by the turk.

## 2.2 Expanding the dead-zone and shifting the viewport

The next step in the algorithm, *expand_deadzone* (Line 22), draws a zone around the viewport for the turk to indicate the dead-zones (Figure 3). The turk is then asked by

**Figure 3.** Dead-zones drawn around viewport.

**Figure 4.** In the case that a partial match is found (ex. tower in the image), the dead-zones will be indicated, and the viewport will be shifted accordingly based on user distance indicated.

the algorithm to estimate the appearance of the pattern next to the current viewport. Once the turk indicates where the pattern could possibly occur, we can shift and infer new dead-zones.

In the case that a partial match is found (i.e. $p$ is overlapping zones), the turk indicates the dead-zoned areas, and the algorithm moves the viewport based on the shift distance indicated. (Figure 4)

### 2.3 Pattern not found/Slicing

If the user has indicated that there is no possibility of the pattern in the extended dead-zone the algorithm proceeds to slice $s$ into 8 zones starting from each corner (Figure 5) by using the procedure *create_live_zones* (Line 24) which returns the aforementioned 8-zones with live-zone data. The algorithm recurses into each of the rectangular zones created and performs all the steps outlined above until a pattern is found or all zones the size of $p$ are dead-zoned. The algorithm utilizes parallelism introduced in TurKit. The TurKit *fork* function works on the same principles as the machine equivalent of the function i.e. it creates a new process. The function passed into the fork further divides the problem and eventually returns the result. The different return values are synchronized through the *join* function.

### 2.4 Completion

A critical step to completion of the algorithm is resolving issues with fuzziness. In the process above, fuzziness is mitigated by having the turks vote using the *mturk_vote* (Line 14) function from the TurKit library. Alternatively, taking multiple samples of the results processed by turks and performing regression analysis is possible (not shown above).

## 3 Expected Case

While there are no experimental results at this stage, we can reason about the expected case of the algorithm. Figure 6 presents a typical search case in the algorithm using 3 turks as an example. Parallelization of the task has not been shown, however,

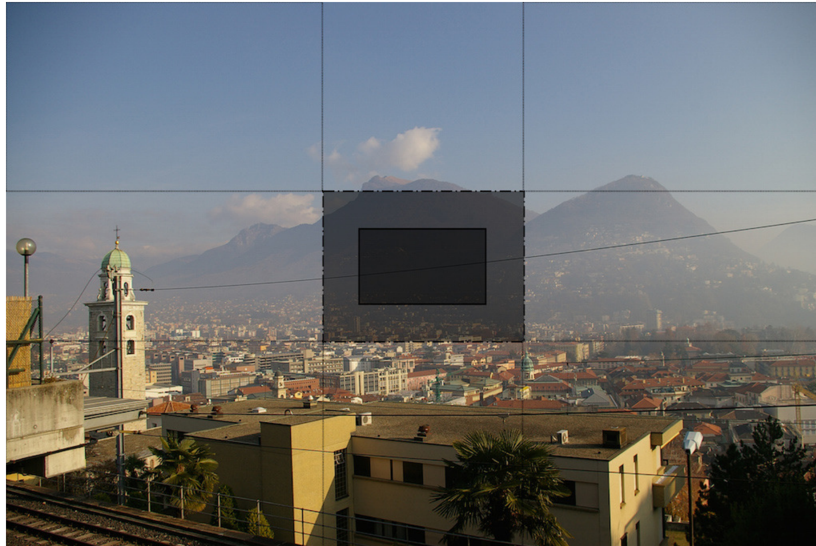


**Figure 5.** New zones to be analyzed.

it is feasible to send the initial task to all three turks starting at different locations of the viewport.

At the start of the algorithm, the viewport is generated and, along with the pattern, sent to Turk 1. Turk 1 indicates that the pattern is not in the viewport, and not likely to be near i.e. the dead-zones are inferred. The algorithm will make a number of shifts from areas without possible matches when the dead-zone is indicated by the user. This yields an advantage over scanning areas sequentially since the algorithm does not have to check every area of $S$.

Subsequently, the algorithm then slices an image into new live-zones and sends the data to Turk 2 which indicates the pattern is near. The algorithm then shifts an amount indicated by the user and sends new live-zones to Turk 3 where a match is finally indicated.

## 4 Robustness

As stated before, due to the parallelized nature of the algorithm we are able to reduce the error in computing problems. In the case of the DZ algorithm, chunks of the images can be sent to multiple turks for verification. Parallelized results are then compared to verify with a lower degree of error that the final result is valid. This mitigates mistakes and inherent change blindness in the turk. In the current version of the algorithm, voting is used to verify results.

The development of a natural algorithm heavily relies on concepts from the field of Human Computer Interfaces. A search algorithm must consider human memory principles for processing data i.e. the short term memory is limited to seven chunks at a given time while the long term memory is useful for seeing larger patterns [12]. Chunking is designed to deal with human memory limitations, namely limiting the number of artifacts on the screen by splitting data into meaningful pieces (a well known example of that are phone numbers where the area code is separated from the rest of the number). Chunking the information for the search algorithm can decrease the processing time and reduce errors in turk processing. The algorithm

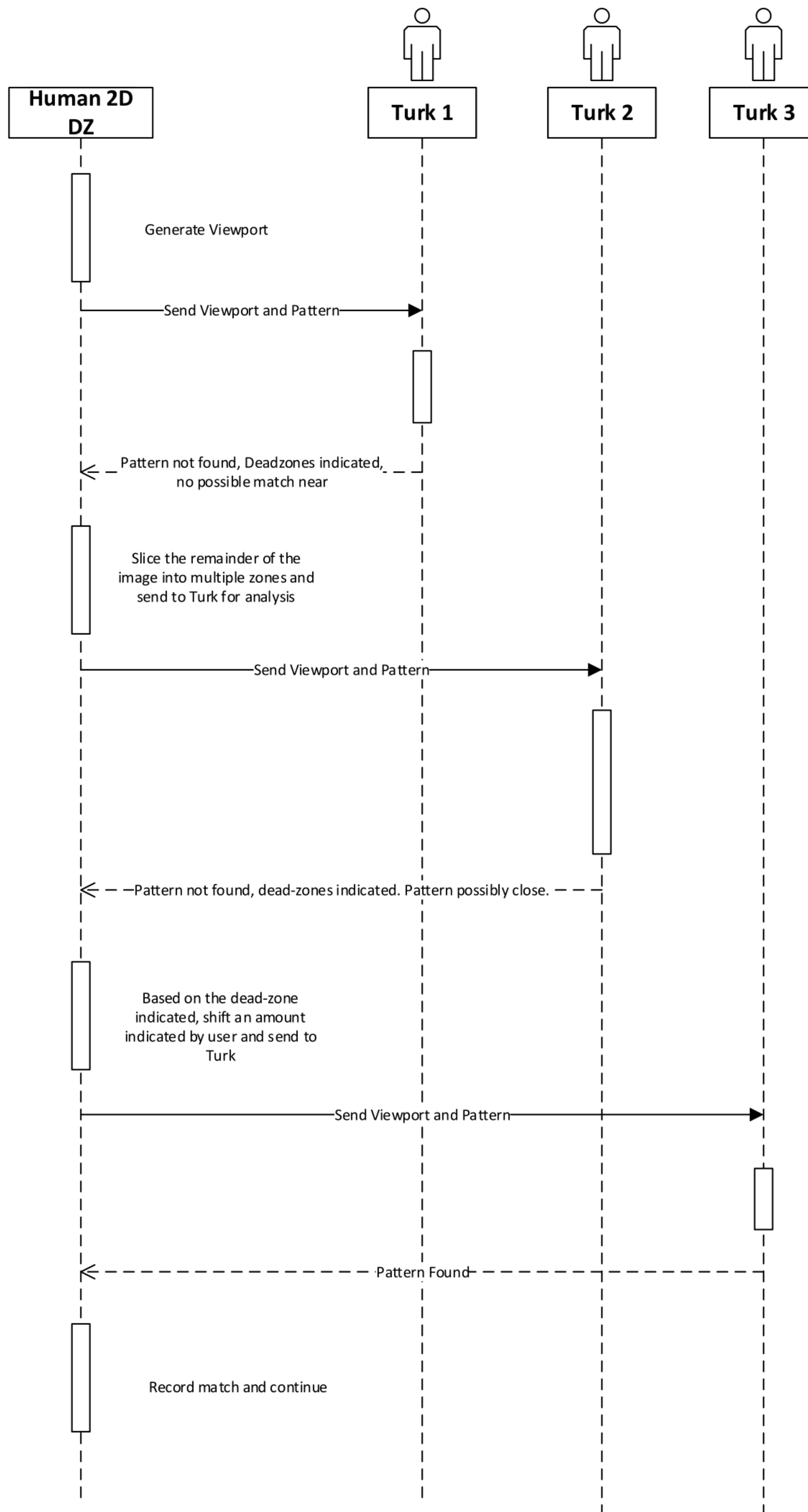



**Figure 6.** Sequence of Human 2D DZ.

utilizes chunking to help the user examine smaller sections of data to find a pattern match.

A problem the algorithm runs into is the definition of problem bounds. If the turk is asked to detect an object on a 2D surface, the turks may bring their own biases. An example of such a bias could be an object that is differently coloured or sized, but otherwise the same. Limiting the scope of the problem is a critical part of the algorithm. The algorithm does not deal with this problem, it is left for the user to provide a sufficient scope to the turks.

## 5 Performance

The greatest cost savings this algorithm provides is in dead-zones indicated by each turk. The entire image does not have to be scanned if there is no chance for a match.

The performance of the algorithm largely varies from turk to turk; for example adults may take more care in scanning the viewport while children may haphazardly scan the same area. Research outlined in [8] has shown a number of results. The experiments considered a number of problems dealing with blurry text, iterative writing and photo sorting. Part of the time is spent on waiting for turks to accept tasks and waiting for turks to perform the work, which is to be expected from a natural system. As mentioned before, effective chunking increases the performance of the algorithm, but there may be cases where chunking is not possible. Additionally, the time to compute with a turk will be significantly different than using a machine. For this reason, it is necessary to separate "human time" from "computing time". For example, the time to slice and shift is largely dependent on computing resources, while the time to find a pattern is dependent on the turk recognizing the object. For this reason we introduce $O(ht)$ for the asymptotic notation of human time. The 2D algorithm's time complexity would therefore be $T(n) = 8T(\frac{n}{8}) + ht \times n$, where the work done outside the recursion is indicating dead-zones and shifting. In the case of this algorithm, the running time will depend on the marking of the potential dead-zone and shift performed by the user. The worst case for machine computation, where major shifts do not occur, being $O(n \log n)$.

The dead-zone algorithm has a number of advantages for searching text. In the case of most algorithms, the worst case scenario is quadratic $O(|S|^2)$ while the best case scenario is $O(\frac{|S|}{|p|})$. In the case of a DZ algorithm, the worst case scenario remains the same, however, the best case scenario is significantly improved. The best case scenario yielded by the DZ algorithm is $O(\frac{|S|-|p|+1}{2|p|-1})$. In practice the improvement is significant since the algorithm performs half the match attempts. Additionally, the algorithm is easily parallelisable which is a key in battling the latency presented by human computation.

A large part of the performance will depend on the Human-Computer Interfacing due to the high amount of interactions and latency between the turk and the machine. A well designed interface will make the process seamless by removing obstructions for new turks in the process. As mentioned above, the time it takes the human to perform the task will be vastly different from the time it takes a machine to perform the same task. In the case of a human, and additionally, the time to process will differ from turk to turk. A well designed interface will optimize the processing and reduce the average time spent by the turk. The experimentation which remains to be done will test this algorithm against other alternatives in the paper to gauge which performs

fastest in a real life scenario. The alternatives will include classic divide and conquer search algorithms and displaying the entire matrix to the user for their peruse.

Furthermore, the cost of running turks will have to be tweaked and tested to determine the best balance in cost to difficulty of task ratio.

### 5.1 Parallelization

In order to create a robust algorithm the final result validity has to be measured to be reasonably accurate within a confidence interval. Parallelization occurs in two levels. The first level poses the problem multiple times to an array of turks. The second level parallelizes the work that needs to be completed in a single run by multiple turks. The chunking of the pieces to be found yields work that can be performed by multiple turks at the same time, asynchronously. This paves way for distributed human computation.

## 6 Conclusions and future work

Human computation is just beginning to scratch the surface with the introduction of such applications as Captcha and Amazon Turk. A human computation DZ algorithm can be used in various fields dealing with imaging. Some examples discussed before were dealing with pictures and sounds, however, more concrete examples of such pattern matching could include geotagging locations (human turks indicate where various locations on a picture are), screening for cancers (determining cancerous patterns on a photo) etc.

With the proven efficiency of a 1-dimensional DZ algorithm we are expecting a more efficient matrix search using the pattern recognition of a human turk. Additionally, using a human turk gives us the possibility of performing flexible image processing while keeping the cost and time of the turk down.

While most modern algorithms tend to examine machine learning and artificial intelligence, human computation departs from this concept by utilizing human turks to perform simple work in order to solve a bigger problem. Humans are currently inherently better at recognizing patterns and with continued expansion of social networks we are given more access to resources. Utilizing the power of distributed networks, human computation can lead to results faster with the help of traditional algorithms.

The next steps in the algorithm is to measure the running time and cost. Due to approximate nature of human computation, a sample of data comparing the two algorithms above will be taken. The data will measure the number of steps that are taken to find the needle in a haystack in order to get a more accurate cost. Additionally, the experiment will measure the time taken to find and the amount of false positives and negatives yielded by both algorithms. The goal of the algorithms is to optimize robustness, running time and user experience. Furthermore, human computation sorting and classification algorithms need to be examined and expanded. Humans have tendencies and biases, and therefore it is important to adapt algorithms to work more naturally with a human.

## References

1. Haoqi Zhang, Eric Horvitz, Yiling Chen, and David C. Parkes: Task Routing for Prediction Tasks. Proceedings AAMAS 2012, pp. 889–896.
2. Ece Kamar, Severin Hacker, and Eric Horvitz: Combining Human and Machine Intelligence in Large-scale Crowdsourcing. Proceedings AAMAS 2012, pp. 467-474.
3. Greg Little: Programming with Human Computation. MIT, 2007.
4. Luis von Ahn: Human computation. Carnegie Mellon University, 2nd edition, 2005.
5. Daniel Villatoro, Jordi Sabater-Mir, Jaime Simo Sichman: Validation of Agent-Based Simulation through Human Computation: An Example of Crowd Simulation. School of Computer Engineering, Nanyang Technological University, Singapore 2012.
6. Amazon MTurk: Amazon MTurk FAQ. https://www.mturk.com/mturk/help?helpPage=overview
7. Bruce W. Watson, Derrick G. Kourie, and Tinus Strauss: A Sequential Recursive Implementation of Dead-Zone Single Keyword Pattern Matching. IWOCA 2012, LNCS 7643, Springer-Verlag 2012, pp. 236–248.
8. Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller: TurKit: Human Computation Algorithms on Mechanical Turk. Proceedings UIST 2010, pp. 57–66.
9. Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller: Exploring Iterative and Parallel Human Computation Processes. Proceedings HCOMP 2010, pp. 68–76.
10. Andrew Mao, Ariel D. Procaccia, and Yiling Chen: Better Human Computation Through Principled Voting. Proceedings AAAI Conference on Artificial Intelligence 2013.
11. Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford: CAPTCHA: Using Hard AI Problems For Security. Eurocrypt 2003, LNCS 2656, pp. 294–311.
12. George A. Miller: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. The Psychological Review vol. 63, no. 2, 1959, pp. 81–97.

# Generating All Minimal Petri Net Unsolvable Binary Words*

Evgeny Erofeev[1], Kamila Barylska[2], Łukasz Mikulski[2], and Marcin Piątkowski[2]

[1] Parallel Systems, Department of Computing Science
Carl von Ossietzky Universität, D-26111 Oldenburg, Germany
evgeny.erofeev@informatik.uni-oldenburg.de
[2] Faculty of Mathematics and Computer Science
Nicolaus Copernicus University, 87-100 Toruń, Poland
{kamila.barylska,lukasz.mikulski,marcin.piatkowski}@mat.umk.pl

**Abstract.** Sets of finite words, as well as some infinite ones, can be described using finite systems, e.g. automata. On the other hand, some automata may be constructed with the use of even more compact models, like Petri nets. We call such automata Petri net solvable. In this paper we consider the solvability of singleton languages over a binary alphabet (i.e. binary words). An unsolvable (i.e. not solvable) word $w$ is called minimal if each proper factor of $w$ is solvable. We present a complete language-theory characterisation of the set of all minimal unsolvable binary words. The characterisation utilises morphic-based transformations which expose the combinatorial structure of those words, and allows to introduce a pattern matching condition for unsolvability.

**Keywords:** binary words, labelled transition systems, generations, Petri nets, synthesis

## 1 Introduction

To deal with infinite sets of words we need to specify them in a finite way. Finite automata which are known as a classical model for describing regular languages, are equivalent to finite labelled transition systems [9]. Some sets may be expressed with use of even more compact system models.

In this paper we investigate the synthesis problem with a specifications given in the form of labelled transition systems. The sought system model is a free-labelled place/transition Petri net [12], with its reachability graph as a natural bridge between specification and implementation. Namely, we are concerned with finding a net, whose reachability graph is isomorphic to a given labelled transition system. Labelled Petri nets are known to be more powerful than finite automata, and hence labelled transition systems [10]. On the other hand, the class of free-labelled Petri net languages is a subset of the class of all Petri net languages. In the present paper we draw attention to the following question: what classes of automata can or cannot be generated by free-labelled Petri nets.

To address this issue one may use the theory of regions [1]. For a given labelled transition system, the solution of a number of linear inequations systems provided by the theory of regions exists if and only if there exists an implementation in a net form.

* This research has been partially supported by the Polish grant No.2013/09/D/ST6/03928, and by DFG (German Research Foundation) through grant Be 1267/14-1 CAVER (Design and Analysis Methods for Real-Time Systems) and Graduiertenkolleg GRK-1765 SCARE (System Correctness under Adverse Conditions).

Evgeny Erofeev, Kamila Barylska, Lukasz Mikulski, Marcin Piątkowski: Generating All Minimal Petri Net Unsolvable Binary Words, pp. 33–47.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

Moreover, solutions of such linear inequations systems are usually utilised during the synthesis of the resulting system (see Synet [5] and APT [13]).

Our aim is to suggest a combinatorial approach and to provide a complete characterisation of a generative nature for a special kind of labelled transition systems – non-branching and acyclic transition systems having at most two labels (i.e. binary words) [2]. More precisely, we characterise all minimal unsolvable binary words.

The paper is organized as follows. First we give some basic notions and notations concerning labelled transition systems, Petri nets and theory of regions. After that we present a necessary condition for minimal unsolvability in the form of extended regular expressions [6]. It allows to formulate possible shapes of minimal unsolvable words. In section 4 we introduce the notion of (base) extendable and non-extendable binary unsolvable words. In the following sections we provide the main results of this paper: a generic characterisation of all minimal unsolvable binary words (section 5) and its utilization for an efficient verifying procedure (section 6). We conclude the paper with a short section containing some directions for further research.

Due to the page limitation, most of technical proofs were omitted. The extended version of this paper containing all the proofs and a detailed argumentation is available for more inquisitive readers (see: [3]).

## 2 Basic notions

In this section we introduce notions used throughout the paper.

**Words**

A *word* (or a *string*) *over alphabet* $T$ is a finite sequence $w \in T^*$, and it is *binary* if $|T| = 2$. For a word $w$ and a letter $t$, $\#_t(w)$ denotes the number of times $t$ occurs in $w$. A word $w' \in T^*$ is called a *subword* (or *factor*) of $w \in T^*$ if $\exists u_1, u_2 \in T^* \colon w = u_1 w' u_2$. In particular, $w'$ is called a *prefix* of $w$ if $u_1 = \varepsilon$, a *suffix* of $w$ if $u_2 = \varepsilon$, and an *infix* of $w$ if $u_1 \neq \varepsilon$ and $u_2 \neq \varepsilon$. For a word $w = x_1 x_2 \cdots x_n$ we use a notation for a factor $w[i..j] = x_i \cdots x_j$ and for a single letter $w[i] = x_i$.

A mapping $\phi : \Sigma_1^* \to \Sigma_2^*$ is called a *morphism* if we have $\phi(u \cdot v) = \phi(u) \cdot \phi(v)$ for every $u, v \in \Sigma_1^*$ whenever all operations are defined. A morphism $\phi$ is uniquely determined by its values on the alphabet. Moreover, $\phi$ maps the neutral element of $\Sigma_1^*$ into the neutral element of $\Sigma_2^*$.

**Transition systems**

A *finite labelled transition system* (or simply *lts*) with an initial state is a tuple $TS = (S, T, \to, s_0)$ with nodes $S$ (a finite set of states), edge labels $T$ (a finite set of letters), edges $\to \subseteq (S \times T \times S)$, and an initial state $s_0 \in S$.[1] A label $t$ is enabled at $s \in S$, denoted by $s[t\rangle$, if $\exists s' \in S \colon (s, t, s') \in \to$. A state $s'$ is reachable from $s$ through the execution of $\sigma \in T^*$, denoted by $s[\sigma\rangle s'$, if there is a directed path from $s$ to $s'$ which edges are labelled consecutively by $\sigma$. The set of states reachable from $s$ is denoted by $[s\rangle$. A sequence $\sigma \in T^*$ is enabled, or firable, at a state $s$, denoted by $s[\sigma\rangle$, if there is some state $s'$ such that $s[\sigma\rangle s'$.[2] Two labelled transition systems $TS_1 = (S_1, \to_1, T, s_{0_1})$ and $TS_2 = (S_2, \to_2, T, s_{0_2})$ are isomorphic if there is a bijection $\zeta \colon S_1 \to S_2$ with $\zeta(s_{0_1}) = s_{0_2}$ and $(s, t, s') \in \to_1 \Leftrightarrow (\zeta(s), t, \zeta(s')) \in \to_2$, for all $s, s' \in S_1$.

[1] Note that an lts may be considered as a finite automata with no specified set of accepting states.

[2] For compactness, in case of long formulas we write $|_r\, \alpha \,|_s\, \beta \,|_t$ instead of $r\ [\alpha\rangle\ s\ [\beta\rangle\ t$.

A word $w = t_1 t_2 \cdots t_n$ of length $n \in \mathbb{N}$ uniquely corresponds to a finite transition system $TS(w) = (\{0, \ldots, n\}, \{(i-1, t_i, i) \mid 0 < i \leq n \wedge t_i \in T\}, T, 0)$.

**Petri nets**

An *initially marked (free labelled) Petri net* is denoted as $N = (P, T, F, M_0)$ where $P$ is a finite set of places, $T$ is a finite set of transitions, $F$ is the flow function $F\colon ((P \times T) \cup (T \times P)) \to \mathbb{N}$ specifying the arc weights, and $M_0$ is the initial marking (where a marking is a mapping $M\colon P \to \mathbb{N}$, indicating the number of tokens in each place). A transition $t \in T$ is enabled at a marking $M$, denoted by $M[t\rangle$, if $\forall p \in P\colon M(p) \geq F(p,t)$. The firing of $t$ at marking $M$ leads to $M'$, denoted by $M[t\rangle M'$, if $M[t\rangle$ and $M'(p) = M(p) - F(p,t) + F(t,p)$ for every $p \in P$. This can be naturally extended to $M[\sigma\rangle M'$ for sequences $\sigma \in T^*$, and $[M\rangle$ denotes the set of all markings reachable from $M$. The reachability graph $RG(N)$ of a bounded (such that the number of tokens in each place does not exceed a certain finite number) Petri net $N$ is the labelled transition system with the set of vertices $[M_0\rangle$, labels set $T$, set of edges $\{(M, t, M') \mid M, M' \in [M_0\rangle \wedge M[t\rangle M'\}$, and initial state $M_0$,. If a labelled transition system $TS$ is isomorphic to the reachability graph of a Petri net $N$, we say that $N$ *PN-solves* (or simply *solves*) $TS$, and that $TS$ is *synthesisable* to $N$. We say that $N$ solves a word $w$ if it solves $TS(w)$. A word $w$ is then called *solvable*, otherwise it is called *unsolvable*.

**Solvability**

Theory of regions constitutes the most common tool for proving solvability of labelled transition systems. Let $(S, T, \rightarrow, s_0)$ be an lts and $N = (P, T, F, M_0)$ be a Petri net, which we hope to synthesise. The synthesis comprises solving systems of linear inequalities in integer numbers. Those inequalities guaranty satisfiability of the following properties:

**State separation property** (*ssp* in short)
: For every pair $s, s' \in S$ of distinct states ($s \neq s'$) there exists a place $p \in P$ such that $M(p) \neq M'(p)$ for markings $M, M' \in [M_0\rangle$ corresponding to $s$ and $s'$.

**Event/state separation property** (*essp* in short)
: For every state-transition pair $s \in S$ and $t \in T$ with $\neg(s[t\rangle)$ there exists a place $p \in P$ such that $M(p) < F(p,t)$ for the marking $M \in [M_0\rangle$ corresponding to $s$.

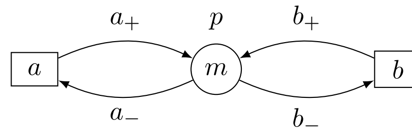



**Figure 1.** A general form of a place $p$ containing initially $m$ tokens and preventing a transition ($a$ or $b$) to satisfy essp.

Note that if the lts is defined by a word $w$ then the state separation property is easy to satisfy by introducing a counter place. On the other hand, satisfiability of event/state separation property, for every state-transition pair $s \in S$ and $t \in T$ with $\neg(s[t\rangle)$, requires a place preventing $t$ at $s$. In the case of binary word $w \in \{a, b\}^*$ such a place $p \in P$ is of the form depicted in figure 2.

The labelled transition systems $TS_1$ and $TS_2$ depicted in figure 2 correspond to the words *aabba* and *abbaa*, respectively. The former is PN-solvable, since the reachability

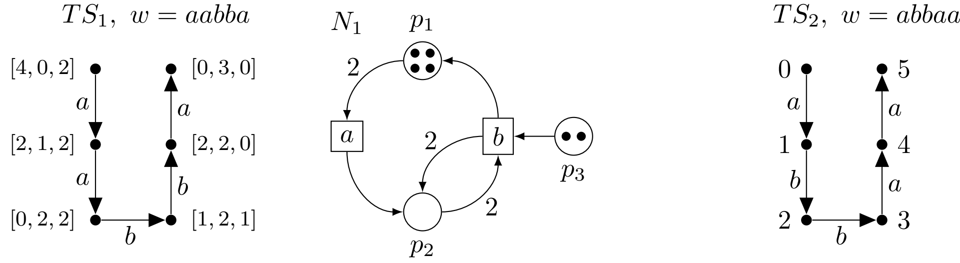



**Figure 2.** $N_1$ solves $TS_1$. No solution of $TS_2$ exists.

graph of $N_1$ is isomorphic to $TS_1$, while the latter contains an unsolvable event/state separation problem represented by event $a$ and state 2 (see [2] for detailed explanation). Note that word $abbaa$, isomorphic to $TS_2$, is the shortest binary word (modulo swapping $a/b$) which is not PN-solvable. However, its reverse ($aabba$) is solvable.

**Minimal unsolvable words**

If a word $w$ is PN-solvable, then all of its subwords $w'$ are. To see this, let the Petri net solving $w$ be executed up to the state before $w'$, take this as the new initial marking, and add a pre-place with $\#_a(w')$ tokens to $a$ and a pre-place with $\#_b(w')$ tokens to $b$. Thus, the unsolvability of any proper subword of $w$ entails the unsolvability of $w$. For this reason, the notion of a *minimal unsolvable word* (*muw* in short) is well-defined, namely, as an unsolvable word all of which proper subwords are solvable. A complete list of minimal unsolvable words up to length 110 can be found, amongst some other lists, in [11].

## 3 Structural classification of minimal unsolvable words

In [2,4] some properties of solvable and of unsolvable words have already been described. In this section we shall indicate some important restrictions which grant all possible shapes of minimal unsolvable words.

Basing on [2] we can state the following proposition which provides a sufficient condition for unsolvability:

**Proposition 1.** SUFFICIENT CONDITION FOR UNSOLVABILITY *If a word over* $\{a, b\}$ *has a subword of the form* (1)*, then it is not PN-solvable.*

$$\boxed{(\ a\ b\ \alpha\ )\ b^*\ (\ b\ a\ \alpha\ )^+\ a\ , \qquad \textit{with } \alpha \in T^*} \tag{1}$$

Further in this paper we show that it is also a necessary condition.

**Remark:** Let us notice that for an arbitrary $\alpha$ the language described by the expression $(ab\alpha)b^*(ba\alpha)^+a$ is not regular, not even context free.

It can be shown ([3]) that, up to swapping $a/b$, all minimal unsolvable words match one of the following three general patterns:

$$\boxed{\begin{array}{l} ab^{x+k}ab^xa, \text{ with } x > 0,\ k > 2 \quad \text{ or} \\ ab^{x+2}(ab^{x+1})^*ab^xa, \text{ with } x > 0 \quad \text{ or} \\ ab^{x_1}ab^{x_2}a \cdots ab^{x_n}a, \text{ with } x_1 = x+1,\ x_n = x,\ x_i \in \{x, x+1\} \text{ for } x > 0,\ n \geq 3 \end{array}} \tag{2}$$

$$\boxed{\begin{array}{l} bab^x(ab^{x+1})^*ab^{x+2}, \text{ with } x>0 \quad \text{or} \\ bab^{x_2}ab^{x_3}a\cdots ab^{x_n}, \text{ with } x_2=x,\ x_n=x+1,\ x_i\in\{x,x+1\} \text{ for } x>0,\ n\geq 3 \end{array}} \tag{3}$$

$$\boxed{ab^xaa, \text{ with } x>2 \quad \text{or} \quad abb(ab)^kaa, \text{ with } k\geq 0} \tag{4}$$

**Remark:** Let us notice that words of the form (3) start and end with $b$, while the other start and end with $a$. For some technical purpose, let us concentrate on words containing not less $b$'s than $a$'s. In the case of equal numbers of $a$'s and $b$'s we concentrate on words starting with $a$.

Note that both last forms of patterns (2) and (3) do not satisfy (1). In order to prove the necessity of the condition from proposition 1 we restrict them even more, obtaining as a side effect complete characterisation and the compatibility with (1). Moreover, the sets of words generated by all the patterns listed above are mutually disjoint. In the following section we divide them into classes of extendable and non-extendable words.

## 4 Generative nature of minimal unsolvable binary words

In this section we provide a complete characterisation of minimal unsolvable binary words. The general idea is to split the whole set into two classes: extendable (which are origins for more complex minimal unsolvable words) and non-extendable (which might be also seen as origins of more complex unsolvable, but not minimal, binary words). In the former class we distinguish the simplest extendable muw's, i.e. the words in which the factor $\alpha$ from (1) is of the form $a^i$ or $b^i$. Such words are called base extendable. After introducing the class of base extendable words, we provide an extension operation based on simple morhisms, which are prefix codes. The code nature is used in subsequent section, where we define the converse operation, called compression.

### 4.1 Base extendable and non-extendable words

The following definitions must be understood modulo swapping $a/b$.

**Definition 2.** BASE EXTENDABLE WORDS

A word $u \in \{a,b\}^*$ is called *base extendable* if it is of the form

$abw(baw)^ka$ with $w=b^j,\ j>0,\ k\geq 1,$ or

$baw(abw)^kb$ with $w=b^j,\ j\geq 0,\ k\geq 1.$

The class of base extendable words is denoted by $\mathcal{BE}$. □ 2

**Definition 3.** NON-EXTENDABLE WORDS

A word $u \in \{a,b\}^*$ is called *non-extendable* if it is of the form

$abb^jb^kbab^ja$ with $j\geq 0,\ k\geq 1.$

The class of all non-extendable words is denoted by $\mathcal{NE}$. □ 3

We now establish that all words from classes $\mathcal{BE}$ and $\mathcal{NE}$ are minimal unsolvable.

**Lemma 4.** Minimal unsolvability of base extendable and non-extendable words *If $w$ belongs to class $\mathcal{BE}$ or $\mathcal{NE}$, then it is unsolvable and minimal with that property.*

**Proof:** Let us notice that a word $w$ is a muw if and only if $w$ is unsolvable and every proper prefix and every proper suffix of $w$ is solvable. Every word $w$ from $\mathcal{BE} \cup \mathcal{NE}$ is of the form (1), hence unsolvable. We shall prove the minimality of $w$ by indicating Petri nets solving its proper prefix and suffix.

**CASE 1 (base extendable words):**

**(a)** $w = abb^j(bab^j)^k a$

Consider first an arbitrary (modulo swapping $a/b$) base extendable word of the form $w = abb^j(bab^j)^k a$ with $j \geq 0$ and $k \geq 1$. This form satisfies (1) with $\alpha = b^j$, the star $^*$ being repeated zero times, and the plus $^+$ being repeated $k$ times. Due to proposition 1, all binary words of this form are unsolvable.

The maximal proper prefix $abb^j(bab^j)^k$ of this word can be solved by Petri net $N_1$ in figure 4.1. Place $q$ in this net enables the initial $a$, and then disables it unless $b$ has been fired $j+2$ times. After the execution of block $bb^jb$ there are $k-1$ tokens more than $a$ needs to fire on place $q$. These surplus tokens allow $a$ to be fired after each sequence $b^jb$, but not earlier. Place $p$ has initially 1 token on it, which is necessary to execute block $bb^jb$ after the first $a$, and this place has only $j+1$ tokens after each next $a$, preventing $b$ at states where $a$ must occur. Places $d$ and $c_b$ prevent undesirable occurrences of $b$ at the very beginning and at the very end of the prefix, respectively.

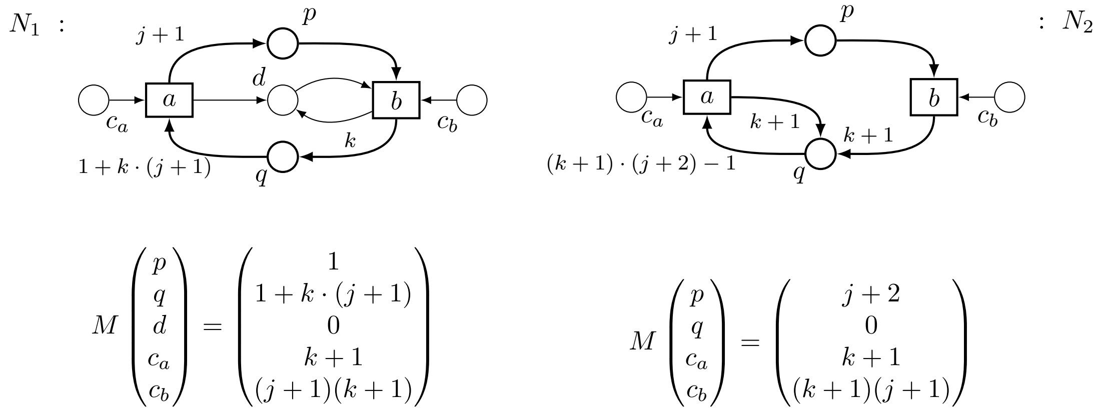



**Figure 3.** $N_1$ solves the prefix $abb^j(bab^j)^k$. $N_2$ solves the suffix $bb^j(bab^j)^k a$.

For the general form of maximal proper suffix $bb^j(bab^j)^k a$ of $w$, one can consider Petri net $N_2$ on the right-hand side of figure 4.1 as a possible solution. Indeed, place $q$ prevents premature occurrences of $a$ in the first block $bb^jb$, and enables $a$ only after this and each next block $b^jb$. Doing so, it collects one additional token after each $b^jb$, which allows this place to enable the very last $a$ after sequence $b^j$. The initial marking allows to execute the sequence $bb^jb$ at the beginning, and at most $j+1$ $b$'s in a row after that, thanks to place $p$. Place $c_b$ restricts the total number of $b$'s allowing only block $b^j$ at the end. Thus we deduce that any word of the form $abb^j(bab^j)^k a$ with $j > 0$ and $k \geq 1$ is a muw.

**(b)** $w = bab^j(abb^j)^k b$

We can similarly examine arbitrary (modulo swapping $a/b$) base extendable word of another form $w = bab^j(abb^j)^k b$ with $j \geq 0$ and $k \geq 1$. The word $w$ satisfies (1) with $\alpha = b^j$, the star $^*$ being repeated zero times, the plus $^+$ being repeated $k$ times, and $a$ and $b$ swapped. Due to proposition 1, all binary words of this form are unsolvable. Petri nets $N_1$ and $N_2$ in figure 4 are possible solutions for maximal proper prefix and for maximal proper suffix of $w$, respectively.

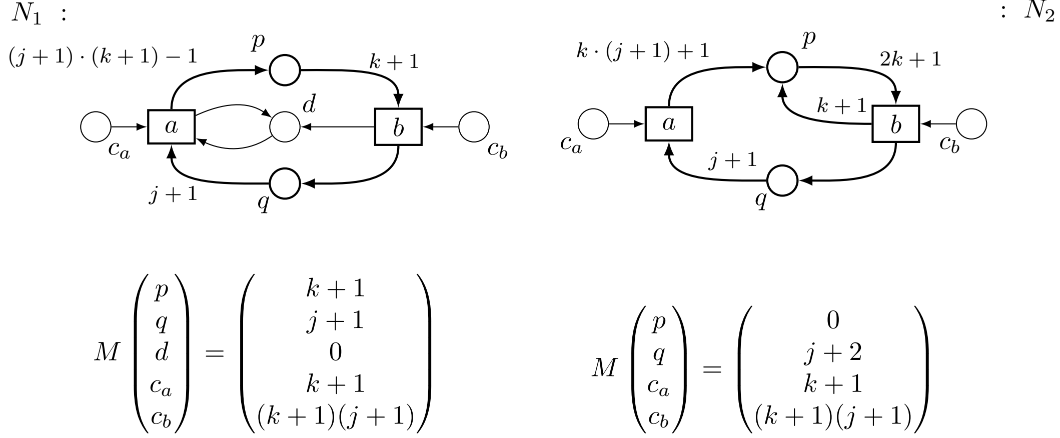



**Figure 4.** $N_1$ solves the prefix $bab^j(abb^j)^k$. $N_2$ solves the suffix $ab^j(abb^j)^k b$.

**CASE 2 (non-extendable words):**

We now demonstrate that any (modulo swapping $a/b$) binary word of the form $w = abb^j b^k bab^j a$ with $j \geq 0$ and $k \geq 1$ from class $\mathcal{NE}$ is minimal unsolvable. The word $w$ satisfies (1) with $\alpha = b^j$, the star $^*$ being repeated $k$ times, and the plus $^+$ being repeated only once. Due to proposition 1, $w$ is unsolvable. To show minimality of $w$, we provide Petri nets $N_1$ and $N_2$ (see figure 4.1) solving its maximal proper prefix and maximal proper suffix, respectively. □ 4

**Remark** (*On special structure of Petri nets which solve prefixes and suffixes*):
Petri net $N_1$ in figure 4.1, which solves maximal proper prefix $abb^j(bab^j)^k$ of word $w = abb^j(bab^j)^k a$ from class $\mathcal{BE}$, has a special structure. Place $d$ serves for preventing undesirable $b$ in the very beginning of $w$, and places $c_a$ and $c_b$ restrict the total number of $a$'s and $b$'s, correspondingly. So, the internal structure of the word, being executed by $N_1$, is determined by two places $p$ and $q$, which prevent $b$ and $a$, respectively, whenever it is necessary. In what follows, we will call the part of $N_1$ consisting of these two places (and transitions) the *core part* . So, Petri net $N_2$ if figure 4.1 has the core part made of places $p$ and $q$. Similarly, such parts are formed by places $p$ and $q$ for both nets in figure 4 as well as both nets in figure 4.1. In future consideration we shall sometimes concentrate only on such core parts, as the other necessary places may be easily added and does not influence the main behaviour of the nets.

*Example 5.* Let us consider a word $w = abbbaba$, which is of the form (1), with $\alpha = b$, the star $^*$ being repeated zero times, and the plus $^+$ being repeated just once. By definition 2, $w$ is a base extendable word with $j = 1$ and $k = 1$. The word $w$ is unsolvable (by proposition 1) and minimal with that property. We show the minimality by introducing Petri nets solving a proper prefix $abbbab$ and a proper

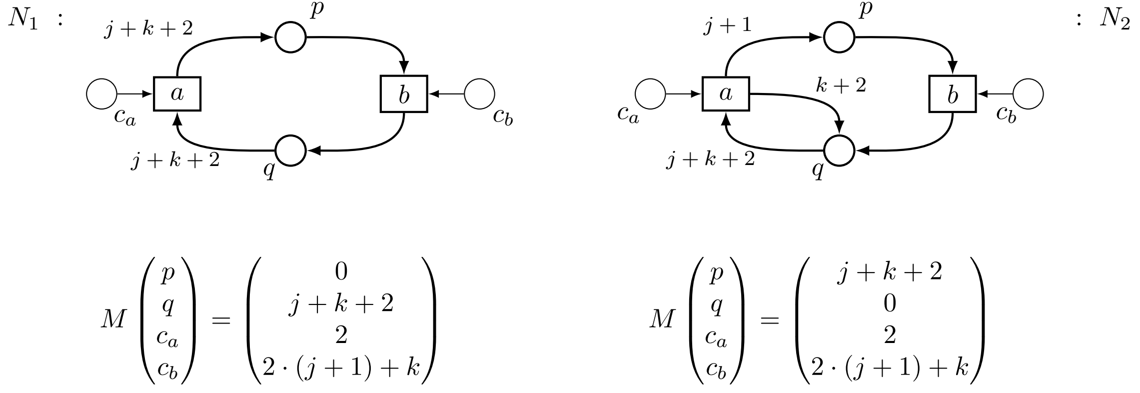



**Figure 5.** $N_1$ solves the prefix $abb^jb^kbab^j$. $N_2$ solves the suffix $bb^jb^kbab^ja$.

suffix *bbbaba* of $w$. Those Petri nets, constructed on the basis of the proof of lemma 4, are depicted in figure 5.

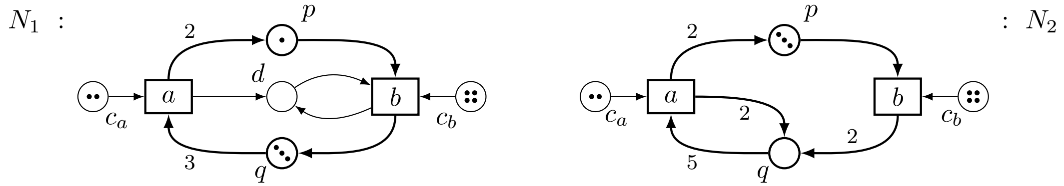



**Figure 6.** $N_1$ solves the prefix *abbbab*. $N_2$ solves the suffix *bbbaba*.

Notice that both Petri nets contain core parts consisting of places $p$ and $q$, which are responsible for the required behaviour of the nets, as well as auxiliary places – a delay place $d$ and counter places $c_a$ and $c_b$.

## 4.2 Extension operation and extendable words

Let us now explain how some minimal unsolvable words can be obtained from other minimal unsolvable words. For this purpose we use the following notion of *extension operation*:

**Definition 6.** Extension operation

For a word $u = xwx$ ($w \in \{a, b\}^*$, $x \in \{a, b\}$) an extension operation $E$ is defined as follows:

$$E(awa) = \bigcup_{i=1}^{\infty} \left\{abM_{a,i}(w)a^{i+1},\ aM_{b,i}(wa)\right\},$$

$$E(bwb) = \bigcup_{i=1}^{\infty} \left\{baM_{b,i}(w)b^{i+1},\ bM_{a,i}(wb)\right\},$$

where $M_{a,i}$ and $M_{b,i}$ are morphisms defined as follows

$$M_{a,i} = \begin{cases} a \mapsto a^{i+1}b \\ b \mapsto a^ib \end{cases} \qquad \text{and} \qquad M_{b,i} = \begin{cases} a \mapsto b^ia \\ b \mapsto b^{i+1}a \end{cases}.$$

□ 6

In what follows, for a given $w \in \{a,b\}^*$, we shall call $u \in E(w)$ an *extension* of $w$.

We are now ready to define the class of *extendable words*.

**Definition 7.** (Derivative) extendable words

For a word $w \in \{a,b\}^*$

1. if $w \in E(v)$ for some base extendable $v$, then $w$ is (derivative) extendable,
2. if $w \in E(v)$ for some extendable $v$, then $w$ is (derivative) extendable,
3. there are no other (derivative) extendable words.

The class of all (derivative) extendable words is denoted by $\mathcal{E}$. In what follows we call them simply *extendable words*. □ 7

The following lemmata constitute unsolvability and minimality of all extendable words.

**Lemma 8.** Unsolvability of extendable words *If $u \in \{a,b\}^*$ is of the form $abv(bav)^k a$ ($k > 0$), then every $w \in E(u)$ is unsolvable.*

**Proof:** It follows directly by definitions 2 and 7, and proposition 1. □ 8

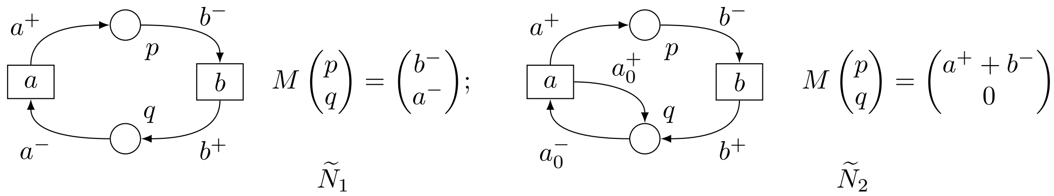



**Figure 7.** Core parts of Petri nets: $\widetilde{N}_1$ for a net solving prefix, $\widetilde{N}_2$ for a net solving suffix.

**Transformations of core part w.r.t. morphisms**

As it has been demonstrated above, for every base extendable word $w$ there are Petri nets $N_1$ and $N_2$, which solve maximal proper prefix $w_1$ and maximal proper suffix $w_2$ of $w$, respectively. Recall that the nets $N_1$ and $N_2$ have a special structure: so called "core" parts $\widetilde{N}_1$ and $\widetilde{N}_2$ (general patterns of $\widetilde{N}_1$ and $\widetilde{N}_2$ are depicted in figure 4.2) determining internal order of firings of $a$'s and $b$'s during execution of $w_1$ and $w_2$, while the remaining parts of $N_1$ and $N_2$ take responsibility for correct implementation of the beginnings and the ends of $w_1$ and $w_2$. Applying operation $E$ to $w$, one can easily obtain new minimal unsolvable word $w'$. Moreover, applying appropriate transformation (which is determined by the particular morphism that has been used to gain $w'$ from $w$) to $\widetilde{N}_1$ or to $\widetilde{N}_2$, one derives new core part $\widetilde{N}'_1$ or $\widetilde{N}'_2$, which correctly implements the internal structure of maximal proper prefix $w'_1$ or maximal proper suffix $w'_2$ of $w'$, respectively. In table 1 the correspondence between morphisms from definition 6 and such transformations of nets is provided for general forms of $\widetilde{N}_1$ and $\widetilde{N}_2$. This fact is confirmed throughout the proof of the following lemma

**Lemma 9.** Minimality of extendable words *If $w \in \mathcal{E}$, then $w$ is minimal unsolvable.*

| | $M_{a,i}$ | $M_{b,i}$ |
|---|---|---|
| $\widetilde{N}_1$ | $a^+ \longmapsto a^+ + b^-$ | $a^+ \longmapsto a^+ + i \cdot (a^+ + b^-)$ |
| | $b^- \longmapsto b^- + i \cdot (a^+ + b^-)$ | $b^- \longmapsto a^+ + b^-$ |
| | $b^+ \longmapsto b^+ + i \cdot (a^- + b^+)$ | $b^+ \longmapsto a^- + b^+$ |
| | $a^- \longmapsto a^- + b^+$ | $a^- \longmapsto a^- + i \cdot (a^- + b^+)$ |
| | $M(p) \longmapsto b^- + i \cdot (a^+ + b^-)$ | $M(p) \longmapsto a^+ + b^-$ |
| | $M(q) \longmapsto a^- + b^+$ | $M(q) \longmapsto a^- + i \cdot (a^- + b^+)$ |
| $\widetilde{N}_2$ | $a^+ \longmapsto a^+ + b^-$ | $a^+ \longmapsto a^+ + i \cdot (a^+ + b^-)$ |
| | $b^- \longmapsto b^- + i \cdot (a^+ + b^-)$ | $b^- \longmapsto a^+ + b^-$ |
| | $b^+ \longmapsto b^+ + i \cdot (a_0^- + b^+ - a_0^+)$ | $b^+ \longmapsto b^+ + a_0^- - a_0^+$ |
| | $a_0^- \longmapsto a_0^- + b^+$ | $a_0^- \longmapsto a_0^- + i \cdot (b^+ + a_0^- - a_0^+)$ |
| | $a_0^+ \longmapsto a_0^+$ | $a_0^+ \longmapsto a_0^+$ |
| | $M(p) \longmapsto b^- + (i+1) \cdot (a^+ + b^-)$ | $M(p) \longmapsto a^+ + (i+1) \cdot (a^+ + b^-)$ |
| | $M(q) \longmapsto 0$ | $M(q) \longmapsto 0$ |

**Table 1.** Correspondence between morphisms and transformations

**Proof:** *(Sketch)* Unsolvability follows from lemma 8. By definition 7, for every $w \in \mathcal{E}$ there is a sequence $w_0, w_1, \ldots, w_r$ such that $w_0 \in \mathcal{BE}$, $w_j \in \mathcal{E}$ and $w_j \in E(w_{j-1})$ for $1 \leq j \leq r$, and $w_r = w$. With induction on $r$ and using table 1, one can construct the core parts of Petri nets, solving maximal proper prefix and suffix of $w$. Additional parts of these nets can be implemented in an uncompliceted way. ☐ 9

Let us note that the extension operation being applied to an extendable word, produces another extendable word which is unsolvable and minimal. On the other hand, from a non-extendable word this operation derives unsolvable but not minimal words.

**Lemma 10.** Unsolvability of extensions of non-extendable words
*If $w \in \mathcal{NE}$, then extension $u \in E(w)$ is unsolvable but not minimal.*

**Proof:**
Follows from definition 3 and 6, and decomposition of the result using proposition 1.
☐ 10

*Example 11.* Observe again the word $w = abbbaba$. From the previous considerations (see example 5) we know that this word is base extendable, and therefore is a muw. By the application of the extension operation, using the morphism $M_{a,1} = \begin{cases} a \mapsto aab \\ b \mapsto ab \end{cases}$ we obtain word $w_{a,1} = ab\,ababa\,ba\,ababa\,a$, which is of the form (1) with $\alpha = ababa$, the star $^*$ being repeated zero times, and the plus $^+$ being repeated just once, hence – by proposition 1 – unsolvable. On the basis of the Petri nets of figure 5, and according to table 1 we construct Petri nets (depicted in figure 11) solving the maximal proper prefix *abababababaababa* and the maximal proper suffix *babababaababaa* of $w_{a,1}$. Thus, $w_{a,1}$ is a minimal unsolvable word.

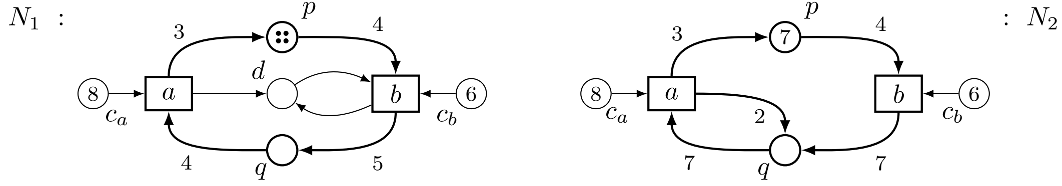



**Figure 8.** $N_1$ solves the prefix *ababababaababa* and $N_2$ solves the suffix *babababaababaa* of $w_{a,1} = ababababaababaa$.

# 5 Generation-based classification of minimal unsolvable binary words

Regard minimal unsolvable words w.r.t. the classification obtained earlier. All possible patterns from (2)–(4), can be distinguished into base extendable

$ab(ba)^{k+1}a$, with $k \geq 0$, for the second pattern from (4),

$abb^x(bab^x)^k a$, with $x > 0, k > 0$, for the second pattern from (2),

$bab^x(abb^x)^k b$, with $x > 0, k > 0$, for the first pattern from (3),

non-extendable

$abb^{x-1}baa$, with $x > 2$ for the first pattern from (4),

$abb^x b^{k-1} bab^x a$, with $x > 0$, $k > 2$ for the first pattern from (2),

and the rest, which we call $\mathcal{C}$ (*compressible*)

$ab^{x_1}ab^{x_2}a \cdots ab^{x_n}a$, with $x_1 = x+1$, $x_n = x$, $x_i \in \{x, x+1\}, x > 0$, $n \geq 3$,
for the third pattern from (2),

$bab^{x_2}ab^{x_3}a \cdots ab^{x_n}$, with $x_2 = x$, $x_n = x+1$, $x_i \in \{x, x+1\}, x > 0$, $n \geq 3$,
for the second pattern from (3).

From this classification we derive that the class of all minimal unsolvable words $\mathcal{MUW} = \mathcal{BE} \cup \mathcal{NE} \cup \mathcal{C}$, where $\mathcal{BE}$, $\mathcal{NE}$ and $\mathcal{C}$ are mutually disjoint classes. Note, that since all words from class $\mathcal{E}$ are unsolvable and minimal with that property, and $\mathcal{E}$ is disjoint with $\mathcal{BE}$ and $\mathcal{NE}$, we have $\mathcal{E} \subseteq \mathcal{C}$.

## 5.1 Morphic compression and reducibility

In the previous section we showed how to construct new minimal unsolvable words on the basis of extendable words. The purpose of this section is to introduce an inverse transformation,which allows to compress longer minimal unsolvable words into shorter ones.

**Definition 12.** Compression function

For a word $v = xux$ ($u \in \{a,b\}^*$, $x \in \{a,b\}$) a *compression function* $C$ is defined as follows :

$$\begin{aligned} C(abua^{i+1}) &= aM_{a,i}^{-1}(u)a, \quad C(baub^{i+1}) = bM_{b,i}^{-1}(u)b, \\ C(auba) &= aM_{b,i}^{-1}(uba), \qquad C(buab) = bM_{a,i}^{-1}(uab), \end{aligned} \tag{5}$$

where $i \geq 1$ and $M_{a,i}^{-1}$, $M_{b,i}^{-1}$ are functions defined as follows:

$$M_{a,i}^{-1} : \begin{cases} a^{i+1}b & \mapsto a \\ a^i b & \mapsto b \end{cases} \qquad \text{and} \qquad M_{b,i}^{-1} : \begin{cases} b^i a & \mapsto a \\ b^{i+1}a & \mapsto b. \end{cases}$$

☐ 12

It is easy to see that among all possible forms from the classification of minimal unsolvable words, function $C$ can only be applied to patterns from class $\mathcal{C}$. Moreover, the form of a given word from $\mathcal{C}$ explicitly defines the particular function $M_{x,i}^{-1}$ which is used when applying $C$ to the word. Let us also notice that since $\mathcal{E} \subseteq \mathcal{C}$, all words from class $\mathcal{E}$ are compressible with function $C$.

From definitions 6 and 12 it is clear that the morphisms $M_{x,i}$ are reciprocal to the functions $M_{x,i}^{-1}$ for $x \in \{a,b\}$, $i \geq 1$. The following lemma establishes that the extension operation $E$ and the application of compression function $C$ are complement to each other in the following sense.

**Lemma 13.** Compression and extension operations

1. *If $v \in \mathcal{BE} \cup \mathcal{E}$ and $u \in E(v)$, then $C(u) = v$;*
2. *If $u \in \mathcal{C}$ and $v = C(u)$, then $u \in E(v)$.*

**Proof:** Can be ascertained by consecutive application of extension and compression operations, according to definition 6 and 12. ☐ 13

### 5.2 Compression of a muw is an unsolvable word

By use of lemma 13, it can be shown that $\mathcal{C} \subseteq \mathcal{E}$, implying that classes of extendable and compressible words coincide. This fact completes the characterisation of all minimal unsolvable words regarding their generative nature, and allows us introduce one of the main results of the paper:

**Theorem 14.** Generative nature of minimal unsolvable binary words
*Let $w$ be a minimal Petri net unsolvable binary word. Then we have the following exclusive alternatives:*

- *$w$ is a non-extendable word ($w \in \mathcal{NE}$), or*
- *$w$ is a base extendable word ($w \in \mathcal{BE}$), or*
- *$w$ is an extendable word ($w \in \mathcal{E}$).*

Basing on theorem 14 and proofs of lemmata 4 and 8 we can formulate the following

**Corollary 15** (The necessary condition for unsolvability).
*If a word over $\{a,b\}$ is not PN-solvable, it has a subword of the form (1).*

### Generation of maximal partial solutions of minimal unsolvable words

In the last case of the alternative from theorem 14 (case $w \in \mathcal{E}$), applying function $C$ to $w$ consecutively, we can recover a sequence of minimal unsolvable words $w_0, w_1, \ldots, w_r$, such that $w_0 \in \mathcal{BE}$, $w_r = w$, $w_i \in \mathcal{E}$ and $w_{i-1} = C(w_i)$ for $1 \leq i \leq r$.

Moreover, starting from a word $w_0$, its maximal proper prefix and maximal proper suffix, and Petri nets solving them (in special forms, that have been provided in the paper), using appropriate transformations, we can derive Petri nets solving maximal proper prefix and maximal proper suffix of $w_i$ for all $1 \leq i \leq r$. We now demonstrate this with the following example:

*Example 16.* Let us consider word $v = ba\,aabaaabaa\,ab\,aabaaabaa\,b$. It is unsolvable by proposition 1, because it is of the form $ba\alpha\, a^*\, (ab\alpha)^+\, b$ (which is exactly the form (1) – modulo swapping $a/b$) with $\alpha = aabaaabaa$, the star $^*$ being repeated zero times, and the plus $^+$ being repeated just once. Due to theorem 14, if $v$ is minimal, then it belongs to one of the classes $\mathcal{BE}, \mathcal{NE}, \mathcal{E}$. Since it does not fit the patterns of classes $\mathcal{BE}, \mathcal{NE}$, we now aim to check whether $v \in \mathcal{E}$. In order to do this we compress $v$ with function $C$. It can be easily seen that the word could be written in the form

$$v = b(aaab)(aaab)(aaab)(aab)(aaab)(aab),$$

hence we need to consider the function $M_{a,2}^{-1} : \begin{cases} aaab & \mapsto a \\ aab & \mapsto b \end{cases}$, and by the compression we obtain word $v_{a,2}^{-1} = baaabab$. Let us notice that $v_{a,2}^{-1}$ is dual to the word $w = abbbaba$ (see example 5), up to swapping $a/b$, hence it is a minimal unsolvable word. Function $C$ cannot be applied to $w = C(v)$, which accord with the fact that $w \in \mathcal{BE}$.

Moreover, starting with the word $w = abbbaba$, together with Petri nets solving its proper prefix and suffix (see figure 5) and applying the morphism $M_{b,2} : \begin{cases} a & \mapsto bba \\ b & \mapsto bbba \end{cases}$ we obtain the word $w_{b,2} = ab\,bbabbbabb\,ba\,bbabbbabb\,a$ which is dual to $v$ up to swapping $a/b$. By the previous considerations we can easily construct Petri nets solving the maximal proper prefix and the maximal proper suffix of $w_{b,2}$, hence, by swapping letters we can obtain Petri nets for a proper prefix and a proper suffix of $v$. Such nets are depicted in figure 16. Now we can state that the word $v$ is not only unsolvable, but also minimal with that property.

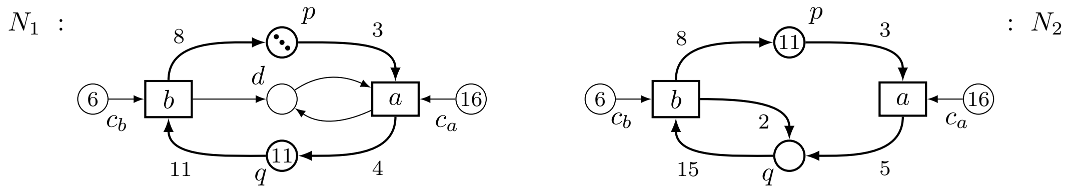



**Figure 9.** $N_1$ solves the prefix $baaabaaabaaabaabaaabaa$ and $N_2$ solves the suffix $aaabaaabaaabaabaaabaab$ of $v = baaabaaabaaabaabaaabaab$ .

## 6 Algorithm for checking unsolvability

The classification of minimal unsolvable words presented in sections 3 and 4 leads to an efficient algorithm for verifying solvability/unsolvability of a binary word. By definition 3 all non-extendable words are of the form (Ia) $ab^x ab^y a$ or (Ib) $ba^x ba^y b$, where $x > y + 2$, $y \geq 0$, and by definition 2 and 6 all extendable words (including base extendable ones) are of the form (IIa) $abw(baw)^k a$ or (IIb) $baw(abw)^k b$, where $k \geq 1$ and $w \in \{a, b\}^*$.

Recall that a word $v \in \{a,b\}^*$ containing a minimal unsolvable word as a factor is also unsolvable. Moreover, due to theorem 14, $v$ is unsolvable if it contains at least one of the patterns (Ia) (Ib), (IIa) or (IIb). Therefore, checking the solvability of a binary word can be reduced to a pattern-matching problem.

The algorithm described below takes a binary word $v$ as an input and returns true if $v$ is solvable and false otherwise (i.e. any of the above mentioned patterns was found inside $v$).

As the first step we search for the patterns (Ia) and (Ib). We scan the input word from left to right comparing the sizes of the two blocks of consecutive $b$'s between any three consecutive occurrences of $a$ and the sizes of the two blocks of consecutive $a$'s between any three consecutive occurrences of $b$. This can be done in $O(n)$ time and $O(1)$ space.

The second step is to search for the patterns (IIa) and (IIb). It utilizes the Knuth-Morris-Pratt failure function called also the border table (see [7]). For any position $i$ in $v$ it contains the length of the longest factor $u$, which is at the same time a proper prefix and a proper suffix of $v[1..i]$. Such a factor is called a border of $v[1..i]$. For the relation between borders and periods of a word see for instance [8].

The search for the patterns (IIa) and (IIb) is performed as follows. For any possible pair of letters $v[i..i+1] = ab$ ($v[i..i+1] = ba$ respectively) we temporarily swap $v[i]$ with $v[i+1]$ and then build the border table for the suffix of $v$ starting at position $i$. After discovering a repetition $v[i..j]$ (i.e. difference between $j$ and the lenght of the border divides $j-i+1$) we check whether it is followed by $a$ ($b$ respectively) and report the occurrence of the pattern if needed.

The border table for a single suffix of the input word $v$ can be constructed in $O(n)$ time and $O(n)$ space (see [7]). We have to process at most $O(n)$ suffixes of $v$, therefore the second step and the whole algorithm runs in $O(n^2)$ time and $O(n)$ space.

## 7 Conclusions and future work

In this paper we studied the class of binary words which can not be generated by any injectively-labelled Petri net, and which are minimal with that property. We examined in detail all possible shapes of such words. The presented classification of minimal unsolvable words results in the construction of a pattern-matching based algorithm for checking the solvability/unsolvability for binary words. The implementation could be found at [11]. Moreover, we introduced the extension and compression functions, which can be foundations of a fixed-point procedure for the generation of the set of all minimal unsolvable binary words. The non-extendable and base extendable words are defined by simple parametrized formulas (see definitions 3 and 2). Choosing all possible values of the parameters $j$ and $k$ we can generate all non-extendable and base extendable words of a given length. Then by using recursive calls of extension and compression function we can generate all extendable words of a given length.

It would be interesting to examine larger alphabets in the hope of finding analogous regularities. The present work can also be of interest in a wider context – a natural extension of this work would consist in analyzing more complex labelled transition systems in terms of their solvability, utilizing the presented results. For instance, for an unsolvable word $w$, we might find a net $N$ whose reachability graph consists of only two maximal branches labelled by $w$ and $w'$, for some $w'$. Then we can deliberate over "approximate solvability" of $w$.

## References


1. E. Badouel, L. Bernardinello, and P. Darondeau: *Petri Net Synthesis*, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2015.
2. K. Barylska, E. Best, E. Erofeev, Ł. Mikulski, and M. Piątkowski: *On binary words being Petri net solvable*, in Proceedings ATAED 2015, vol. 1371, CEUR-WS.org, 2015, pp. 1–15.
3. K. Barylska, E. Erofeev, Ł. Mikulski, and M. Piątkowski: *Generating all minimal Petri net unsolvable binary words - full version*, 2016, http://folco.mat.umk.pl/papers/generating-binary-muws.pdf.
4. E. Best, E. Erofeev, U. Schlachter, and H. Wimmel: *Characterising petri net solvable binary words*, vol. 9698 of Lecture Notes in Computer Science, 2016, pp. 39–58.
5. B. Caillaud: 2002, http://www.irisa.fr/s4/tools/synet.
6. C. Câmpeanu, K. Salomaa, and S. Yu: *A formal study of practical regular expressions.* International Journal of Foundations of Computer Science, 14(6) 2003, pp. 1007–1018.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, eds., *Introduction to algorithms*, MIT Press, third ed., 2009.
8. M. Crochemore, L. Ilie, and W. Rytter: *Repetitions in strings: Algorithms and combinatorics.* Theoretical Computer Science, 410(50) 2009, pp. 5227–5235.
9. M. Droste and R. M. Shortt: *From petri nets to automata with concurrency.* Applied Categorical Structures, 10(2) 2002, pp. 173–191.
10. J. L. Peterson: *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, 1981.
11. M. Piątkowski et al.: 2015, http://folco.mat.umk.pl/unsolvable-words.
12. W. Reisig: *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*, Springer, 2013.
13. U. Schlachter et al.: 2013, http://github.com/CvO-Theory/apt.

# Interpreting the Subset Construction Using Finite Sublanguages

Mwawi Msiska* and Lynette van Zijl

Dept of Mathematical Sciences, Computer Science Division,
Stellenbosch University, Private Bag X1, 7602 Matieland, South Africa
mfmsiska@gmail.com
lvzijl@cs.sun.ac.za
http://www.cs.sun.ac.za

**Abstract.** We present a language-based approach to the well-known problem of the conversion between finite automaton (FA) types. We base our approach on the existence, for any FA, of a finite subset of the language of the FA, which we call the finite exhaustive language (FEL). An FA uses all its *reachable* transitions after computing all strings in its FEL. We convert the FA by *summarizing* its computations on strings from the FEL of its equivalent FA. We illustrate our approach using the well known nondeterministic finite automaton (NFA) to deterministic finite automaton (DFA) conversion. We describe a method to calculate the FEL of the DFA through graph traversals of the NFA, without first converting the NFA into a DFA. Using the FEL, we construct a DFA that has neither dead nor unreachable states. For an $n$-state NFA, we show that $O(e^{\sqrt{n \log n}})$ is an upper bound on the length of strings in the FEL of its equivalent DFA.

## 1 Introduction

For most finite automata, there are algorithms to simulate one type of automaton by another. For example, a deterministic finite automaton (DFA) can simulate a nondeterministic finite automaton (NFA) using the well-known subset construction [8]. In general, consider a simulation of finite automaton (FA) class $A$ by FA class $B$. Most simulation algorithms are based on relating each state of an equivalent class $B$ FA to some collection of states of the given class $A$ FA. Transitions of the class $B$ FA are constructed by relating properties of the state collections to the transition function of the given class $A$ FA. Typically, arguments on the structure of these state collections in the simulations are fairly easy; however, deriving simulated transition functions describing the associations amongst these state collections can be quite complex. For example, see the derivation of transitions among *crossing sequences* in [4].

In contrast, we demonstrate an algorithm that is language-based, rather than state-based, for such simulations. Consider a simulation of a class $A$ FA, $M$, by a class $B$ FA, $M'$, accepting a language $\mathcal{L}(M)$. Our approach constructs a finite sublanguage $\mathcal{L}'(M)$ of $\mathcal{L}(M)$ (called the finite exhaustive language, or FEL), by conducting restricted walks through the digraph (that is, the state diagram) of $M$. Our claim is that the equivalent class $B$ FA, $M'$, will then exhaust (visit) all its states and transitions to recognize $\mathcal{L}'(M)$. In our approach, we only need to know the structure of the collection of states of $M$ relative to a single state of $M'$. The transitions are

* The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Mwawi Msiska, Lynette van Zijl: Interpreting the Subset Construction Using Finite Sublanguages, pp. 48–62.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

implied from actual runs of $M$ on members of $\mathcal{L}'(M)$. We illustrate the approach for the simplest case, namely, the simulation of NFAs by DFAs.

In the literature, the construction of finite automata based on a finite language has previously been considered [2,3,9]. However, the context there is typically applications such as dictionary building, and not simulations between automata types per se. The interested reader may also note the similarities between the construction of the FEL based on walks through the digraph, and other rule-based descriptions of infinite languages such as regular expressions [4] and language equations [1]. The FEL corresponds to all strings that can be obtained by such formalisms, but with a bound on the length of the strings. The contribution of this paper is therefore the proof of the upper bound for the length of the strings in the FEL, and the algorithms to simulate one automaton class by another based on the FEL.

Section 2 recalls the classical one-way FA (NFAs and DFAs), and sets out the notation that we use throughout the paper. We introduce the notion of finite exhaustive languages for FAs in Sect. 3, where we also present a generalized simulation algorithm. In Sect. 4, we illustrate the generalized algorithm in the context of the well known NFA to DFA conversion. Specifically, we describe a method for calculating the FEL for the equivalent DFA by using specialized walks through the NFA digraph (state diagram), which we call simple computations.

## 2 Preliminaries

Let $\Sigma$ be a nonempty finite set of symbols. We denote the free monoid over $\Sigma$ by $\Sigma^*$. If $w \in \Sigma^*$, then we call $w$ a string or a word. The length of $w$ is denoted by $|w|$, and $w_i$ denotes the $i$-th symbol of $w$, where $1 \leq i \leq |w|$. If $\mathcal{A}$ is a set, then $|\mathcal{A}|$ denotes the number of elements in $\mathcal{A}$, and $2^{\mathcal{A}}$ denotes the powerset of $\mathcal{A}$. We denote the set $\{0, 1, \ldots\}$ by $\mathbb{N}$.

We define a generic FA as a 5-tuple $(Q, \Sigma, \delta, S, F)$, where $Q$ is a finite non-empty set of states and $\delta$ is a state transition function. Here, $S \subseteq Q$ and $F \subseteq Q$ denote the set of start states and the set of accept states, respectively, where $S \neq \emptyset$. Specific classes of FA are defined in the literature by placing restrictions on the last four members of the 5-tuple. For example, a DFA has a single start state, therefore we can replace $S$ by a single state $q_0 \in Q$.

A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where $q_0 \in Q$ is called the start state, and $\delta : Q \times \Sigma \to Q$ is the transition function, which may be a partial function. The transition function $\delta$ can be extended to strings, so that $\widehat{\delta} : Q \times \Sigma^* \to Q$, such that $\widehat{\delta}(q_i, w) = q_{i+|w|}$ if there exists a sequence $q_i, q_{i+1}, \ldots, q_{i+|w|}$, where $q_{t+1} = \delta(q_t, w_{(1+t-i)}),\ i \leq t < (i + |w|)$. The DFA accepts a string $w$ iff $\widehat{\delta}(q_0, w) = q_f$, where $q_f \in F$. Similarly, an NFA is a 5-tuple $(Q, \Sigma, \delta, S, F)$, where $\delta : Q \times \Sigma \to 2^Q$ is the transition function. Again, $\delta$ extends to $\widehat{\delta} : 2^Q \times \Sigma^* \to 2^Q$ where $\widehat{\delta}(\mathcal{A}, a) = \bigcup_{q \in \mathcal{A}} \delta(q, a)$ and $\widehat{\delta}(\mathcal{A}, aw) = \widehat{\delta}(\widehat{\delta}(\mathcal{A}, a), w)$ for some $\mathcal{A} \subseteq Q,\ a \in \Sigma$ and $w \in \Sigma^*$. An NFA accepts a string $w$ iff $\widehat{\delta}(S, w) \cap F \neq \emptyset$. If $M$ is an FA, then $\mathcal{L}(M)$ denotes *the language of* $M$, that is, the set of all the strings accepted by $M$.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA. By *transition*, we refer to the triple $(p, a, q)$, such that $q \in \delta(p, a)$, or $q = \delta(p, a)$ in the case of a DFA, for some $p, q \in Q$ and $a \in \Sigma$. A *computation* of $M$ on a string $w = w_1 w_2 \ldots w_k$ in $\Sigma^*$, starting from state $q_0$, is a sequence $q_0, q_1, \ldots, q_k$ of states in $Q$ such that $(q_i, w_{i+1}, q_{i+1})$ is a transition for all $0 \leq i < k$. We denote this computation by $\vec{\delta}_{w,q_0}$, and if $q_0 \in S$, we simply

write $\overrightarrow{\delta}_w$ . The computation is accepting if $q_0 \in S$ and $q_k \in F$. A state $q \in Q$ is *reachable* from state $p \in Q$ if there exists a computation from $p$ to $q$, otherwise the state $q$ is *unreachable* from $p$. A state $q \in Q \setminus F$ is *dead* if none of the states in $F$ is reachable from $q$. The FA $M$ is *trim* if no state in $Q$ is dead or unreachable. The function $\textsc{trim}(M)$ removes, from $M$, all unreachable and dead states. Note that $\mathcal{L}(M) = \mathcal{L}(\textsc{trim}(M))$.

## 3 Finite Exhaustive Languages and Generalized Simulations

Given any FA $M = (Q, \Sigma, \delta, S, F)$, it is possible to find a finite sublanguage $\mathcal{L}'(M) \subseteq \mathcal{L}(M)$ such that $\textsc{trim}(M)$ exhausts all its transitions (and states) after computing all strings in $\mathcal{L}'(M)$, since $|Q|$ is finite. We call $\mathcal{L}'(M)$ the finite exhaustive language (FEL) of $M$, and formally define it as:

**Definition 1 (Finite exhaustive language).** *Let $M = (Q, \Sigma, \delta, S, F)$ be an FA. Let $\mathcal{T}$ be the set of all transitions in $\textsc{trim}(M)$. Let $f(\overrightarrow{\delta}_w) = \{(p_i, a, p_{i+1}) \mid (p_i, a, p_{i+1})$ is a transition in $\overrightarrow{\delta}_w = p_0, \ldots, p_i, \ldots, p_{|w|}\}$. Then a finite sublanguage $\mathcal{L}'(M) \subseteq \mathcal{L}(M)$ is exhaustive if $\left(\bigcup_{w \in \mathcal{L}'(M)} f\left(\overrightarrow{\delta}_w\right)\right) = \mathcal{T}$.*

Note that the FEL for a given FA is not unique, unless the FA recognizes a finite language. Generally, we seek an FEL with the shortest strings, which ensures the coverage of all accept states.

When simulating a finite automaton $M_{\mathrm{A}}$ by a finite automaton $M_{\mathrm{B}}$, we proceed as follows. First build the FEL $\mathcal{L}'(M_B)$ for $M_B$, using a finite set of finite walks on the digraph of $M_A$. Then run $M_A$ on all strings in $\mathcal{L}'(M_B)$. Assuming $M_{\mathrm{B}}$ processes a string $w \in \mathcal{L}'(M_{\mathrm{B}})$ in the left-to-right order, we can write down the computation of $M_{\mathrm{A}}$ on $w$ as a sequence $C_{\mathrm{qB}} = q_{B_0}, w_1, q_{B_1}, w_2, \ldots, w_{|w|}, q_{B_{|w|}}$, such that a triple $\rho_i = (q_{B_i}, w_{i+1}, q_{B_{i+1}})$ appears as subsequence whenever $q_{B_i}$ is a collection of states of $M_{\mathrm{A}}$ that $M_{\mathrm{A}}$ is in whenever $w_{i+1}$ is the next symbol, when the input head is moving right; $q_{B_{i+1}}$ is a collection of $M_{\mathrm{A}}$ states that $M_{\mathrm{A}}$ enters whenever the input head moves right, past $w_{i+1}$, to the symbol $w_{i+2}$. The collections $q_{B_i}$ in $C_{\mathrm{qB}}$ are states of $M_{\mathrm{B}}$ and the triples $\rho_i$ are transitions of $M_{\mathrm{B}}$. The state $q_{B_{|w|}}$ is an accept state in $M_{\mathrm{B}}$.

*Example 2.* Figure 1(a) shows an NFA, $N_1$, whose simulating DFA, $M_1$, is shown in Fig. 1(b). A candidate FEL for $M_1$ is {aa, ab, aaa, aab, baa, bab, abaa, abbaa}. Figure 1(c) shows a partially constructed $M_1$ (bottom) from a run (top) of $N_1$ on the string aa. We determine the accept state in the partial $M_1$ as the collection of $N_1$ states in which the run of $N_1$ on aa halts, since aa $\in \mathcal{L}(M_1)$.

The next section describes how the general concept of an FEL can be used in the specific case of a DFA that simulates an NFA.

## 4 NFA to DFA Conversion

We introduce the concept of *simple computations* in Sect. 4.1. We use simple computations to calculate the FEL of a DFA from an equivalent NFA, for NFAs with restrictions on their cycle structure. In Sect. 4.2 we show that simple computations do not always yield the FEL of the equivalent [1] DFA, for some subclass of NFAs. We then extend the simple computations to cover the general class of NFAs.

[1] Henceforth, an *equivalent* DFA refers to the subset-construction equivalent DFA.

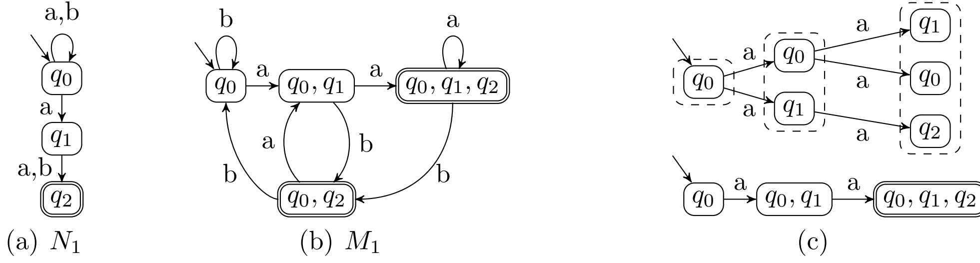



**Figure 1.** (a) An NFA $N_1$. (b) A DFA simulating $N_1$. (c) A run of $N_1$ on the string aa (top) giving part of the simulating DFA (bottom).

### 4.1 Simple Computations

**Definition 3 (Computation path).** *Let $N = (Q, \Sigma, \delta, S, F)$ be an NFA. Then the sequence $C_p = p_0, p_1, \ldots, p_k$ of states in $Q$ is a* computation path *of $N$ if there exists $a_{i+1} \in \Sigma$ such that $p_{i+1} \in \delta(p_i, a_{i+1})$ for all $0 \leq i < k$.*

The reader should note that we use the terms *computation* and *computation path* interchangeably. When the word $w$ which plays a role in the sequence of states is to be emphasized, we use the former, and when we are more interested in the states that can occur in the state sequence, we use the latter.

A computation path $C_\text{p} = p_0, p_1, \ldots, p_k$ is cyclic if there exist $i, j \in \{0, 1, \ldots, k\}$ such that $i < j$ and $p_i = p_j$. The subsequence $p_i, \ldots, p_j$ is a cycle. The cycle is simple if $p_u \neq p_v$ for all $u < v$ in $\{i, \ldots, (j-1)\}$. If a cycle is not simple, then Algorithm 1 can be used to recursively remove all the nested cycles, and return a simple cycle.

In the algorithm, we assume that $C_\text{cycle}$ has multiple nested cycles. The algorithm explores all possible orders of removing the nested cycles, removing one cycle at a time, until a simple cycle remains. Different orders of removing nested cycles may result in different simple cycles. We therefore collect the resulting simple cycles into a set $R$. For example, $\textsc{simplify}([q_0, q_1, q_2, q_3, q_1, q_3, q_0]) = \{[q_0, q_1, q_2, q_3, q_0], [q_0, q_1, q_3, q_0]\}$. We obtain the first simple cycle by deleting the nested cycle $[q_3, q_1, q_3]$; and the second by deleting the nested cycle $[q_1, q_2, q_3, q_1]$.

We now define a *simple computation*, which is a computation path that does not traverse any cycle more than once.

**Definition 4 (Simple computation).** *Let $C_p = p_0, p_1, \ldots, p_k$ be a computation path of an NFA $N = (Q, \Sigma, \delta, S, F)$. $C_p$ is a simple computation if either*

1. *$C_p$ has at most one cycle; or*
2. *if $C_p$ has multiple cycles, then any two cycles $C_{y_1}$ and $C_{y_2}$ in $C_p$ are such that $\textsc{simplify}(C_{y_1}) \cap \textsc{simplify}(C_{y_2}) = \emptyset$.*

A simple computation is *non-cyclic* if it does not include any cycle; otherwise, it is *cyclic.* If $p_0 \in S$ and $p_k \in F$, then the simple computation is *accepting.* We say that a simple computation $C_\text{p} = p_0, p_1, \ldots, p_k$ *yields* a string $w = w_1 w_2 \ldots w_k$ if $p_{i+1} \in \delta(p_i, w_{i+1})$ for all $0 \leq i < k$. Note that a simple computation may actually yield a non-empty finite set of strings (see Example 5). Also note that if $C_\text{p}$ is accepting, then its yield is in $\mathcal{L}(N)$. Henceforth, we denote by $W_\delta$, the union of yields of all the accepting simple computations of an NFA $N = (Q, \Sigma, \delta, S, F)$.

**Algorithm 1.** Simplify

**Require:** $C_{\text{cycle}} = p_0, p_1, \ldots, p_m$ where $p_0 = p_m$
  **function** SIMPLIFY($C_{\text{cycle}}$)
    $R \leftarrow \emptyset$
    SIMPLIFY2($C_{\text{cycle}}, R$)
    **return** $R$
  **end function**
  **procedure** SIMPLIFY2($C_{\text{cycle}}, \&R$)    ▷ $R$ is a reference parameter.
    $m \leftarrow |C_{\text{p}}| - 1$
    $\eta \leftarrow 0$
    **for all** $(i, j) \mid (i < j) \;\wedge\; (p_i = p_j) \;\wedge\; \neg(i = 0 \;\wedge\; j = m)$ **do**
      $\eta \leftarrow \eta + 1$
      $C_{\text{temp}} \leftarrow p_0, \ldots, p_{i-1}, p_j, \ldots, p_m$    ▷ Delete the cycle $p_i, \ldots, p_j$ .
      SIMPLIFY($C_{\text{temp}}, R$)
    **end for**
    **if** $\eta = 0$ **then**    ▷ $C_{\text{cycle}}$ is simple.
      $R \leftarrow R \cup \{C_{\text{cycle}}\}$
    **end if**
  **end procedure**

*Example 5.* Table 1 lists some of the simple computations and their corresponding yields for the NFA $N_3$, in Fig. 2. To illustrate Definition 4, note that $q_0, q_1, q_2, q_2$ is a simple computation, since it has only one cycle. Also, $q_0, q_1, q_2, q_2, q_1$ is a simple computation – although it has two cycles, it is easy to see that SIMPLIFY($[q_1, q_2, q_2, q_1]$) $\cap$ SIMPLIFY($[q_2, q_2]$) $= \{[q_1, q_2, q_1]\} \cap \{[q_2, q_2]\} = \emptyset$ . On the other hand, the computation path $D_{\text{p}} = q_0, q_1, q_2, q_1, q_2, q_2, q_1, q_3$ is not a simple computation, since SIMPLIFY($[q_1, q_2, q_1]$) $\cap$ SIMPLIFY($[q_1, q_2, q_2, q_1]$) $= \{[q_1, q_2, q_1]\} \neq \emptyset$ .

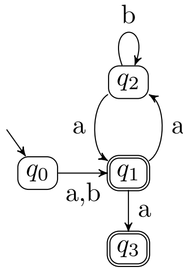



**Figure 2.** $N_3$.

**Table 1.** Some simple computations of $N_3$, and their yields.

| Simple computation | Type | Yield |
|---|---|---|
| $q_0, q_1, q_2$ | non-cyclic | {aa, ba} |
| $q_0, q_1, q_2, q_2$ | cyclic | {aab, bab} |
| $q_0, q_1$ | accepting non-cyclic | {a, b} |
| $q_0, q_1, q_3$ | accepting non-cyclic | {aa, ba} |
| $q_0, q_1, q_2, q_1$ | accepting cyclic | {aaa, baa} |
| $q_0, q_1, q_2, q_2, q_1$ | accepting cyclic | {aaba, baba} |
| $q_0, q_1, q_2, q_2, q_1, q_3$ | accepting cyclic | {aabaa, babaa} |

It is known that there are $n$-state NFAs for which the smallest equivalent minimal DFA has $O(2^n)$ states. One therefore has to consider whether all simple computations of length up to $2^n$ must be considered when the FEL is constructed. To that end, we first show an upper bound on the union of the yields of all simple computations in an NFA.

**Lemma 6.** *If $N = (Q, \Sigma, \delta, S, F)$ is an NFA with $|Q| = n$ states and $w$ is the longest string in $W_\delta$, then $|w|$ is $O(n^3)$.*

*Proof.* Let $N = (Q, \Sigma, \delta, S, F)$ be an NFA. Let $G_{\text{N}} = (Q, E)$ be a digraph representing the transitions of $N$, where $E = \{(p, q) \mid q \in \delta(p, a)$ for some $a \in \Sigma\}$. Note that the edges of $G_{\text{N}}$ are not labelled with transition symbols. Let $C_{\text{p}}$ be an accepting

simple computation of the NFA $N$. If $|Q| = n$, then the maximum number of edges in $G$ is $\left(2 \cdot \binom{n}{2} + n\right) = n^2$. If $CY$ is a simple cycle in $C_\mathrm{p}$, then at least one edge of $CY$ is unique in $C_\mathrm{p}$, otherwise the cycle would have been traversed more than once, thus contradicting Definition 4. Therefore the maximum number of simple cycles in a simple computation is $n^2$. Note that if $C_\mathrm{p}$ has $n^2$ simple cycles, then none of the edges has been repeated, hence the length of $C_\mathrm{p}$ is $(n^2 + 1)$, and the length of each string in the yield of $C_\mathrm{p}$ is $n^2$.

By the pigeon hole principle, any subsequence of $C_\mathrm{p}$ having $(n+1)$ states contains a simple cycle. If $|C_\mathrm{p}| = n(n+1) = n^2 + n$, then $C_\mathrm{p}$ has at least $n$ cycles. Since the number of simple cycles may be less than $n^2$, it may be possible to add more states to $C_\mathrm{p}$ while keeping it a simple computation. However if $|C_\mathrm{p}| = n^2(n+1) = n^3+n^2$, then $C_\mathrm{p}$ has at least $n^2$ simple cycles. Since $|C_\mathrm{p}| > (n^2+1)$, it follows that some edge in $C_\mathrm{p}$ is not unique, therefore $C_\mathrm{p}$ is not a simple computation. Note that when $|C_\mathrm{p}| = n^3$, we cannot be certain that $C_\mathrm{p}$ has at least $n^2$ cycles. Therefore, the maximum length of a simple computation, and thus the longest string in $W_\delta$, is $O(n^3)$. □

**Proposition 7.** *Let $N = (Q, \Sigma, \delta, S, F)$ be an NFA with no accepting cyclic simple computations. Let $M$ be the subset-construction equivalent DFA. Then* TRIM$(M)$ *uses all its transitions after computing all the strings in $W_\delta$.*

*Proof.* Since $N$ has no accepting cyclic simple computations, it follows that all cycles of $N$ (if any) are outside TRIM$(M)$. Thus TRIM$(M)$ is a directed acyclic graph. Therefore, $\mathcal{L}(N)$ is finite and $\mathcal{L}(N) = W_\delta$. □

Proposition 7 informally states that an accepting non-cyclic computation in an NFA has an isomorphic accepting non-cyclic computation in an equivalent DFA derived by the subset construction. The next section compares cycles in an NFA and its subset-construction equivalent DFA.

### 4.2 Extending Simple Computations

If an NFA contains cycles in its trim, then there is no guarantee that $W_\delta$ uses all transitions in any equivalent DFA, as accepting cyclic computations of an NFA are generally not isomorphic to accepting cyclic computations of its equivalent DFA. For example, $W_\delta$ for the NFA $N_4$ in Fig. 3 does not use some transitions of $M_4$, the minimal equivalent DFA.

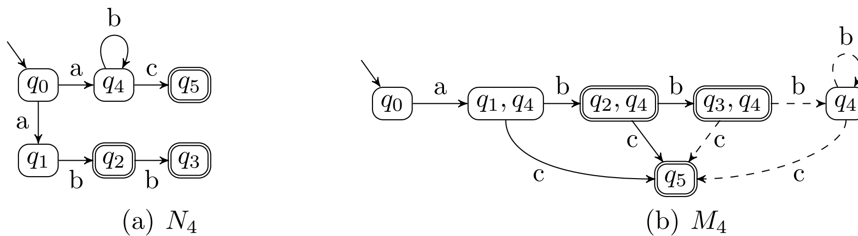



**Figure 3.** (a) An NFA $N_4$, with $W_\delta = \{\text{ab, abb, abc, ac}\}$. (b) The trim of the subset-construction equivalent DFA, $M_4$. No computation of $M_4$ on any string in $W_\delta$ uses any of the dashed transitions. Notice that the cyclic computation $q_0, q_4, q_4, q_5$ in $N_4$ is converted to the non-cyclic computation $\{q_0\}, \{q_1, q_4\}, \{q_2, q_4\}, \{q_5\}$ in $M_4$. Also notice that $M_4$ is minimal.

The transitions of an NFA's equivalent DFA missed by $W_\delta$ can be accounted for by extending computation paths that yield $W_\delta$, by allowing them to traverse cycles

more than once. An upper bound on the number of times a cycle must be traversed to cover the missed transitions must exist, since the equivalent DFA has a finite number of transitions.

The lack of isomorphism between a cyclic computation of an NFA and a cyclic computation of its equivalent DFA is not surprising. A DFA's computation is generally a parallel composition of several computations of its equivalent NFA. Thus to replicate the cyclic behaviour of an NFA's cycle in its equivalent DFA, we consider the interaction between a cyclic simple computation and other computation paths of the NFA. We present the interactions, in order of increasing complexity, in Lemmas 9, 12 and 13. We first set out some new notation.

Let $C_\mathrm{p} = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA $N$, which includes a cycle $CY = p_i, \ldots, p_j$. Let $x$ be a string in the yield of $C_\mathrm{p}$. Let $W'_{CY}$ be the union of the yields of all accepting computation paths of $N$ that do not include the cycle $CY$. The largest integer, $r$, such that $w_1 w_2 \ldots w_u = x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^r$, for all $w \in W'_{CY}$, is the *wrap value* of the cycle $CY$. If $r = \infty$, then the cycle $CY$ *overlaps* some other cycle in $N$. We formally define exactly when two cycles overlap in Definition 11.

**Definition 8 (Free cycle).** *Let $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA $N$, which includes a cycle $CY = p_i, \ldots, p_j$. Let $C_{p_i} = p_0, p_1, \ldots, p_i$ be a prefix of $C_p$ such that $|C_{p_i}| = i + 1$. Let $C_{t_i} = t_0, t_1, \ldots, t_i$ be any computation path, where $t_0 \in S$ and $C_{p_i} \neq C_{t_i}$. Let $W_{p_i}$ and $W_{t_i}$ be the yields of $C_{p_i}$ and $C_{t_i}$, respectively. Then the cycle $CY$ is* free *if*

1. *The wrap value of $CY$ is 0 and*
2. *$W_{p_i} \cap W_{t_i} = \emptyset$.*

**Lemma 9.** *Let $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA $N$, which includes a* free *cycle $CY = p_i, \ldots, p_j$. Let $M$ be a DFA equivalent to $N$. Then $M$ has a computation path isomorphic to $C_p$.*

*Proof.* Let $N = (Q, \Sigma, \delta, S, F)$. Let $x$ be a string in the yield of $C_\mathrm{p}$. Let $C_\mathrm{t} = t_0, t_1, \ldots, t_l$ be an accepting computation path yielding a string $y$. For brevity, we ignore all other computation paths besides $C_\mathrm{p}$ and $C_\mathrm{t}$. Consider the prefixes $C_{p_i} = p_0, p_1, \ldots p_i$ and $C_{t_i} = t_0, t_1, \ldots t_i$ of $C_\mathrm{p}$ and $C_\mathrm{t}$, respectively, where $|C_{p_i}| = |C_{t_i}| = i + 1$. If $C_{p_i} = C_{t_i}$, then the lemma holds true trivially, since Definition 8 is violated. Assume $C_{p_i} \neq C_{t_i}$. Since $CY$ is a free cycle in $N$, then $y_\nu \neq x_\nu$, for some $1 \leq \nu \leq i$. In the worst case, when $y_1 \ldots y_{i-1} = x_1 \ldots x_{i-1}$, one of the branches of the parallel composition of $C_\mathrm{p}$ and $C_\mathrm{t}$ results in the simple computation path $C_\mathrm{pt} = \{p_0, t_0\}, \{p_1, t_1\}, \ldots, \{p_{i-1}, t_{i-1}\}, \{p_i\}, \ldots, \{p_j\}, \ldots, \{p_k\}$ in $M$. It is immediate that $C_\mathrm{p}$ is isomorphic to $C_\mathrm{pt}$. □

**Corollary 10.** *If all cycles of an NFA $N = (Q, \Sigma, \delta, S, F)$ are free, then $W_\delta$ is the FEL of a DFA equivalent to $N$.*

*Proof.* Let $M$ be the DFA equivalent to $N$. By Lemma 9, $M$ has an accepting cyclic simple computation isomorphic to each accepting cyclic simple computation of $N$. Thus each accepting cyclic computation of $N$ is isomorphic to some accepting cyclic computation of $M$. By Proposition 7, $M$ has an accepting non-cyclic computation isomorphic to each accepting non-cyclic computation of $N$. □

**Definition 11 (Cycle overlap).** *Let $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ and $C_t = t_0, t_1, \ldots, t_u, \ldots, t_v, \ldots, t_l$ be accepting cyclic simple computations of an NFA $N$, which include cycles $CX = p_i, \ldots, p_j$, and $CY = t_u, \ldots, t_v$. The cycles $CX$ and $CY$* overlap *if there exists a string $w$ such that both $p_0, p_1, \ldots, p_i, (p_{i+1} \ldots, p_j)^\gamma$ and $t_0, t_1, \ldots, t_u$, $(t_{u+1} \ldots, t_v)^\chi$ contain $w$ in their yields, for some $\chi, \gamma \in \{1, 2, \ldots\}$.*

**Lemma 12.** *Let $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA $N = (Q, \Sigma, \delta, S, F)$, which includes a cycle $CY = p_i, \ldots, p_j$. Let $M = (Q', \Sigma, \delta', q_0, F')$ be the equivalent DFA. Let $r$ be the wrap value of $CY$. Assume the cycle $CY$ is not free. Let $W^\lambda$ be the yield of a computation path of $N$ that completes the cycle $CY$ $\lambda$ times. If $CY$ does not overlap with any other cycle, then $W^{r+2}$ exhaust all transitions of $M$ that $M$ would use to recognize $W^{r+\phi}$, for all $\phi > 2$.*

*Proof.* Let $x = x_1 x_2 \ldots x_i \ldots x_j \ldots x_k$ be a string in the yield of the accepting simple computation $C_\mathrm{p}$. Let $x' = x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^{r+1}$. Let $C_\mathrm{t} = t_0, t_1, \ldots, t_l$ be an accepting computation path of $N$. Let $y$ be a string in the yield of $C_\mathrm{t}$ such that $y_1 y_2 \ldots y_u = x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^r$. For brevity, we ignore all other computation paths besides $C_\mathrm{p}$ and $C_\mathrm{t}$. Consider a computation path in $M$ resulting from the parallel composition of $C_\mathrm{p}$ and $C_\mathrm{t}$, $C_\mathrm{pt} = \{p_0, t_0\}, \{p_1, t_1\}, \ldots, \{p_i, t_i\}, \ldots, \{p_j, t_j\}$. Suppose $p_i = t_i$. Since $x' \neq y_1 y_2 \ldots y_{[i+(r+1)(j-i)]}$, it follows that if $\vec{\delta}\,(t_0, x') = t_0, t_1, \ldots, t_{[i+(r+1)(j-i)]}$ then

$$\vec{\delta}\,(t_0, x') = \emptyset. \tag{1}$$

But, $\delta'(\{p_i, t_i\}, x_{i+1}) = \delta'(\{p_i\}, x_{i+1}) = \{p_{i+1}, t_{i+1}\}$. Therefore

$$\begin{aligned}\delta'(\{p_{[i+(r+1)(j-i)]}, t_{[i+(r+1)(j-i)]}\}, x_{[i+(r+1)(j-i)+1]}) = {} & \delta'(\{p_i, t_{[i+(r+1)(j-i)]}\}, x_{i+1}) = \\ & \{p_{i+1}, t_{i+1}\}\ .\end{aligned}$$

Thus the computation of $M$ on $x' x_{i+1}$ is cyclic. Since $M$ is deterministic, if follows that its computation on $x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^{r+\eta}$, for all $\eta > 1$, repeats the cycle $\{p_{i+1}, t_{i+1}\}, \{p_{i+2}, t_{i+2}\} \ldots, \{p_{i+1}, t_{i+1}\}$, and does not use any new transitions.

Now suppose $p_i \neq t_i$. Consider the state

$$\{p_{[i+(r+1)(j-1)]}, t_{[i+(r+1)(j-1)]}\} = \{p_i, t_{[i+(r+1)(j-1)]}\} = \widehat{\delta'}(q_0, x')\ .$$

If $t_{[i+(r+1)(j-1)]} = t_i$, then we have a cycle, and we are done. If $t_{[i+(r+1)(j-1)]} \neq t_i$ then, because of (1), $t_{[i+(r+1)(j-1)]} \in \delta(p_{i-1}, x_i)$. Consequently $\{p_i, t_i\}$ includes the state $t_{[i+(r+1)(j-1)]}$, which contradicts $t_{[i+(r+1)(j-1)]} \neq t_i$. The only way to resolve this is by fixing $\widehat{\delta'}(q_0, x') = \{p_i\}$. Since the state $\{p_i\}$ has appeared in $M$, the subset construction dictates the existence of the cycle $CY' = \{p_i\}, \{p_{i+1}\}, \ldots, \{p_j\}$ in $M$. Extending $x'$ to $x'' = x_i (x_{i+1} \ldots x_j)^{r+2}$ ensures that the computation of $M$ on $w''$ completes this cycle. Again the computation of $M$ on $x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^{r+\phi}$, for all $\phi > 2$, repeats the cycle $CY'$ and does not use any new transitions. □

**Lemma 13.** *Let $C_p = p_0, \ldots, p_i, \ldots, p_j, \ldots, p_k$ and $C_t = t_0, \ldots, t_u, \ldots, t_v, \ldots, t_l$ be accepting cyclic simple computations of an NFA $N$, which include cycles $CX = p_i, \ldots, p_j$, and $CY = t_u, \ldots, t_v$. Let $l_{CX} = (j - i)$ and $l_{CY} = (v - u)$, and $g$ be the least common multiple (lcm) of $l_{CX}$ and $l_{CY}$. Let $M$ be a DFA equivalent to $N$. If $CX$ and $CY$ are the only overlapping cycles of $N$, then $M$ has a cycle $C_{XY}$ such that*

$|C_{XY}| = (g+1)$. *The shortest strings that can traverse the cycle* $C_{XY}$ *is in the yield of the computation paths*

$$C_p^g = p_0, p_1, \ldots, p_i, (p_{i+1}, \ldots, p_j)^{g/l_{CX}}, p_{j+1}, \ldots, p_k \ \ or$$
$$C_t^g = t_0, t_1, \ldots, t_u, (t_{u+1}, \ldots, t_v)^{g/l_{CY}}, t_{v+1}, \ldots, t_l \ \ .$$

*Proof.* Without loss of generality, assume the worst case, when $i = v + \lambda(v-u)$, for some integer $\lambda \geq 0$. Then the first state in the parallel composition of $C_\mathrm{p}$ and $C_\mathrm{t}$, $C_\mathrm{pt}$, in which states from both cycles appear is $\{p_i, t_{v+\lambda(v-u)}\}$. Let $\{p_i, t_{v+\lambda(v-u)}\}$ be the first state in the cycle $C_\mathrm{XY}$ of $C_\mathrm{pt}$. To complete the cycle in $C_\mathrm{pt}$, we must encounter the state $\{p_i, t_{v+\lambda(v-u)}\} = \{p_j, t_{v+\lambda(v-u)}\}$ again. Going through the two cycles simultaneously, starting from the first occurrence of $\{p_i, t_{v+\lambda(v-u)}\}$, the $h^\mathrm{th}$ occurrence of $p_i$ is $p_{i+hl_\mathrm{CX}}$. The $m^\mathrm{th}$ occurrence of $t_{v+\lambda(v-u)}$ is $t_{v+\lambda(v-u)+ml_\mathrm{CY}}$. The state $\{p_i, t_{v+\lambda(v-u)}\}$ appears again when $i + hl_\mathrm{CX} = v + \lambda(v-u) + ml_\mathrm{CY}$. Since $i = v + \lambda(v-u)$, it follows that

$$hl_\mathrm{CX} = ml_\mathrm{CY} \ \ . \tag{2}$$

Since $g$ is the lcm of $l_\mathrm{CX}$ and $l_\mathrm{CY}$, the smallest values of $h$ and $m$ satisfying (2) are $h = g/l_\mathrm{CX}$ and $m = g/l_\mathrm{CY}$. □

**Corollary 14.** *If an NFA has* $\eta$ *overlapping cycles with lengths*[2] $l_1, l_2, \ldots, l_\eta$, *and* $g$ *is the lcm of the* $l_i$*'s, then the FEL for an equivalent DFA must include yield from accepting computation paths involving* $2, 3, \ldots, (g/l_i)$ *repetitions of the* $i^{th}$ *overlapping cycle, for all* $1 \leq i \leq \eta$.

*Remark 15.* By Corollary 14, the longest cycle in a DFA simulating an $n$-state NFA occurs when all the overlapping cycles have relatively prime lengths and the states in any two cycles are disjoint. In this case $\sum_{i=1}^{\eta} l_i \leq n$. From the upper bound to Landau's function in [7,5], Corollary 16 follows.

**Corollary 16.** *The* $g$ *in Corollary 14 is in* $O(e^{\sqrt{n \log n}})$.

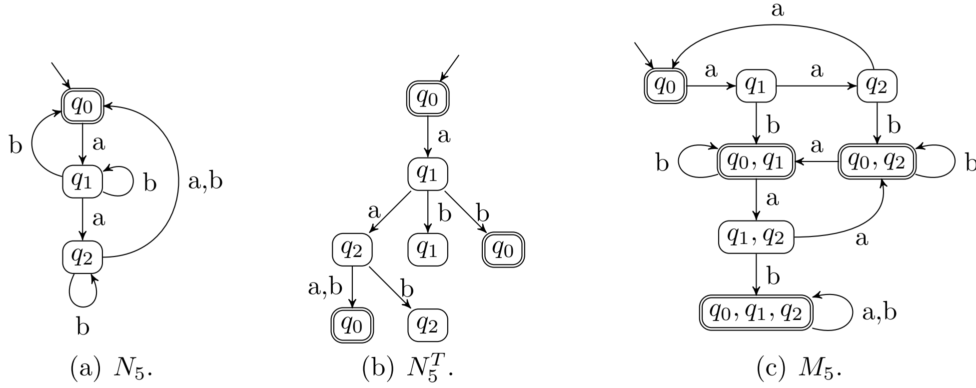
Note than although a DFA may require exactly $2^n$ states in order to simulate an $n$-state NFA, the longest cycle in the simulating DFA has only $O(e^{\sqrt{n \log n}})$ states. Furthermore, each simple cycle in an NFA can be traversed by an (n+1)-state computation path. Therefore the FEL of the equivalent DFA does not require strings longer than $O(e^{\sqrt{n \log n}})$.

We now consider the algorithm to construct an FEL for a DFA equivalent to a given NFA.

### 4.3 Construction Details

**Representing NFAs using Cycle Trees** From the discussion in Sect. 4.2, it is clear that we need to identify all cycles in a given NFA. We thus represent an NFA having a single start state by a tree, which we call a *cycle tree*. We create a cycle tree by performing depth first search from a start state, adding a tree node for each state encountered. We add an edge labelled $a$ from a parent node $q_p$ to a child node $q_c$ if the NFA has a transition $(q_p, a, q_c)$. A branch of the traversal stops when either it encounters a previously seen state, along the current path from the start state, or

[2] Here the length is the number of states in the cycle minus one.




(a) $N_5$. (b) $N_5^T$. (c) $M_5$.

**Figure 4.** (a) A 3-state NFA example from the set of NFAs in [6], for which an $n$-state NFA requires the simulating DFAs to have exactly $(2^n - 1)$ states. (b) A cycle tree representation of $N_5$. (c) The DFA simulating $N_5$.

when the current state does not have transitions. Algorithm 2 creates a cycle tree from a single start state of an NFA $N = (Q, \Sigma, \delta, S, F)$. [3] In the algorithm, $q_0 \in S$. The function CREATETREE($q_0$) creates the cycle tree and returns a pointer to the root of the tree.

**Algorithm 2.** Creating a cycle tree

```
function cycleTree(q0)
    root ← q0
    build(root, ∅)
    return root
end function
procedure build(p, V_path)
    V_path ← {p} ∪ V_path
    for all a ∈ Σ do
        for all q ∈ δ(p, a) do
            edge(p, a, q)
            if q ∉ V_path then
                build(q, V_path)
            end if
        end for
    end for
end procedure
```

In the procedure BUILD($p, V_{\text{path}}$), $p$ is a tree node corresponding to the current NFA state, and $V_{\text{path}}$ is a set of tree node labels (these also correspond to NFA state labels) encountered along the current path from the root. The procedure EDGE($p, a, q$) creates a directed edge from a tree node labelled $p$ to a new tree node labelled $q$. The directed edge is labelled with the symbol $a$.

*Example 17.* Applying Algorithm 2 to the NFA $N_5$ in Fig. 4(a) results in the cycle tree $N_5^T$ in Fig. 4(b).

Properties of NFAs correspond to properties of cycle trees as follows. If an NFA $N = (Q, \Sigma, \delta, S, F)$ has a cyclic computation path $C_{\text{p}} = p_0, p_1, \ldots, p_i, \ldots, p_j$, having

[3] If the NFA has multiple start states, we run the algorithm for each start state, resulting in a forest of cycle trees.

a single simple cycle $p_i, \ldots, p_j$ and $p_0 \in S$, then the path $C_\mathrm{p}$ also appears as a path in some cycle tree, where $p_0$ is the root and $p_j$ is a leaf. If $N$ has an accept state $q_f \in F$, then all occurrences of nodes labelled $q_f$ in the cycle tree are marked as accepting. If the NFA is trim, then, in its cycle trees, every non-accepting leaf node represents an NFA state in a cycle. Note that an accepting leaf node may also indicate a cycle.

The advantage of representing an NFA by a cycle tree is that if the NFA is not trim, then the depth first search avoids all unreachable states. If the resulting cycle tree has a node representing a dead NFA state, then we can identify it as a non-cyclic non-accepting leaf node. Thus we can delete dead states in $O(h)$ time, where $h$ is the depth of the cycle tree.

The computation of a cycle tree is similar to that of an NFA, except when the computation reaches a *cyclic leaf node* $v_\mathrm{cy}$ having the same label as some internal node $v_\mathrm{in}$, along the current path from the root. In this case, the computation jumps, without consuming any symbol, to the node $v_\mathrm{in}$.

**Algorithm 3.** Simple computation

```
function allYield(N^T)
    V_an ← acceptNodes(N^T)
    W_δ ← ∅
    n_c ← cycles(N^T)
    for all 0 ≤ i ≤ n_c do
        for all v_an ∈ V_an do
            V_s ← ∅
            W ← ∅
            trace(W, v_an, V_s, i)
            W_δ ← W_δ ∪ W
        end for
    end for
    return reverse(W_δ)
end function
```

**Simple Computations from a Cycle Tree** We generate the yield of all accepting simple computations of a cycle tree using Algorithm 3. In the algorithm, the function allYield$(N^\mathrm{T})$ returns the union of the yield of all simple computations of the cycle tree $N^\mathrm{T}$; $V_\mathrm{an}$ is the set of accept nodes of $N^\mathrm{T}$; $W_\delta$ is a set of strings, which becomes the union of yields of all accepting simple computations in the end; cycles$(N^\mathrm{T})$ returns the number of simple cycles in $N^\mathrm{T}$; and reverse$(W_\delta)$ reverses every string in $W_\delta$. The procedure trace$(W, v_\mathrm{an}, V_\mathrm{s}, i)$ traces reversed simple computations, with at most $i$ cycles, from the accept node $v_\mathrm{an}$ to the root of $N^\mathrm{T}$, and builds the reverse of strings in the yield. A more detailed description of trace is given in Algorithm 4.

In Algorithm 4, trace$(\&W_\delta, v_\mathrm{b}, V_\mathrm{s}, n_\mathrm{c})$ creates the reversed yields of all accepting simple computations that halt at the accept node, $v_\mathrm{b}$. The parameter $W_\delta$ is a set of reversed strings; $V_\mathrm{s}$ is the set of cyclic leaf nodes visited so far; and $n_\mathrm{c}$ is the maximum number of cycles that the simple computation may go through. The procedure traces

**Algorithm 4.** Trace

**procedure** TRACE($\&W_\delta, v_b, V_s, n_c$) ▷ $R$ is a reference parameter.
  $v_{cd} \leftarrow v_b$, $v_{pt} \leftarrow$ parent($v_{cd}$)
  **while** $v_{cd} \neq$ root **do**
    $V_{lf} \leftarrow$ JUMPTOSET($v_{cd}$)
    **if** $V_{lf} \neq \emptyset$ **then**
      $W'_\delta \leftarrow W_\delta$, $W_\delta \leftarrow \emptyset$
      **for all** $v_{lf} \in V_{lf}$ **do**
        **if** $v_{lf} \notin V_s,\ |V_s| > n_c$ **then**
          $W_{nw} \leftarrow W'_\delta$
          $V_{sn} \leftarrow V_s \cup \{v_{lf}\}$
          TRACE($W_{nw}, v_{lf}, V_{sn}, n_c$)
          $W_\delta \leftarrow W_\delta \cup W_{nw}$
          $V_{sn} \leftarrow V_s$, $W_{nw} \leftarrow W'_\delta$
          EXTEND($W_{nw}, v_{ch}, v_{pt}$) ▷ $W_{nw}$ is a reference parameter.
          TRACE($W_{nw}, v_{pt}, V_{sn}, n_c$)
          $W_\delta \leftarrow W_\delta \cup W_{nw}$
        **else**
          EXTEND($W'_\delta, v_{ch}, v_{pt}$) ▷ $W'_\delta$ is a reference parameter.
          TRACE($W'_\delta, v_{pt}, V_s, n_c$)
          $W_\delta \leftarrow W_\delta \cup W'_\delta$
        **end if**
      **end for**
      **return**
    **else**
      EXTEND($W_\delta, v_{cd}, v_{pt}$) ▷ $W_\delta$ is a reference parameter.
      $v_{pt} \leftarrow$ parent($v_{pt}$)
      $v_{cd} \leftarrow$ parent($v_{cd}$)
    **end if**
  **end while**
  $V_{lf} \leftarrow$ JUMPTOSET($v_{cd}$)
  **if** $(V_{lf} \neq \emptyset) \ \wedge \ (|V_s| > n_c)$ **then**
    **for all** $v_{lf} \in V_{lf}$ **do**
      $W_{nw} \leftarrow W'_\delta$
      **if** $v_{lf} \notin V_s$ **then**
        $V_{sn} \leftarrow V_s \cup \{v_{lf}\}$
      **else if** $|V_s| > n_c$ **then**
        $V_{sn} \leftarrow \{v_{lf}\}$
      **end if**
      TRACE($W_{nw}, v_{lf}, V_{sn}, n_c$)
      $W_\delta \leftarrow W_\delta \cup W_{nw} \cup W'_\delta$
    **end for**
  **end if**
**end procedure**

the reverse of all simple computations from an accept node $v_b$ to the root node, while building the reverse of strings in the union of the yields of the reversed simple computations. When TRACE encounters an internal node $v_{in}$ having the same label as a cyclic leaf node $v_{lf} \notin V_s$, such that a forward path exists from $v_{in}$ to $v_{lf}$, the reversed simple computation is duplicated. One copy proceeds with the parent node of the node $v_{in}$. The other copy proceeds from the leaf node $v_{lf}$. The procedure EXTEND($\&W_\delta, v_{cd}, v_{pt}$) collects all transition symbols on the edge $(v_{pt}, v_{cd})$ into a set $A$, and computes $W_\delta \leftarrow W_\delta \times A$. The function JUMPTOSET($v_{cd}$) returns a set of all

cyclic leaf nodes having the same label as an internal node labelled $v_{\text{cd}}$, if there is a forward path from $v_{\text{cd}}$ to each of the cyclic leaf nodes.

*Example 18.* Algorithm 3 applied to $N_5^{\text{T}}$, in Fig. 4(b) yields the set {$\epsilon$, aaa, aab, abaaa, abaab, abbaabb, abbaaba, aabbab, aaaabb, aabaab, aabb, aababb, aabbabb, abbaab, abaaba, abbaaa, abaabb, ababa, abb, ab, abababa, abababb, ababb, aaaab, aabab, abaa, abab, aaba, aabaabb, abaaab, ababaa, ababab, ababaab, ababbab}, which exhausts all transitions of $M_5$ in Fig. 4(c).

**Extending simple computations** The algorithm for generating the yield of extended computations is similar to Algorithm 3, except that a cycle $CY$ is repeated $2, 3, \ldots, \lambda$, where $\lambda = \max([r+2], g/l_{\text{CY}})$, if $r$ is the wrap value of the cycle, $l_{\text{CY}}$ is the length of $CY$ (measured in terms of the number of transitions) and $g$ is the lcm of lengths of all cycles that overlap with $CY$, including $CY$. In place of the set $V_s$, we use a table that maps each cyclic leaf to the number of times it has been encountered.

We determine the wrap value of a cycle from the cycle tree using Algorithm 5. In the algorithm, $V_{\text{lf}}$ is the set of all leaves and $V_{\text{cy}}$ is the set of all cyclic leaves. The variable `map` is a table that associates each node in $V_{\text{cy}}$ to an integer, representing the maximum number of times a cycle is used in a computation. The function $\textsc{pathYield}(\text{root}, v_{\text{lf}})$ returns the yield of the computation path from the root to the leaf $v_{\text{lf}}$. The procedure $\textsc{run}(\text{map}, v_{\text{lf}}, w)$ considers all possible runs of the string $w$ on the NFA represented by the forest of cycle trees. Each run avoids the state $v_{\text{lf}}$ because the string $w$ is already known to be in the yield of a computation path $\text{root}, \ldots, V_{\text{lf}}$. Each run also keeps track of the number of times each state in $V_{\text{cy}}$ is encountered. At the end of each run, a table entry $(v_{\text{cy}}, \mu) \in$ `map` is updated to $(v_{\text{cy}}, \nu)$ if $\mu < \nu$ where $\nu$ is the number of times the node $v_{\text{cy}}$ was encountered and $v_{\text{cy}} \in V_{\text{cy}}$. In the end, the table `map` associates each cyclic leaf node with its wrap value.

**Algorithm 5.** Wrap value

```
map ← V_cy × {0}
for all v_lf ∈ V_lf do
    W ← pathYield(root, v_lf)
    for all w ∈ W do
        run(map, v_lf, w)
    end for
end for
return map
```

We determine overlapping cycles using Algorithm 6. The algorithm groups cyclic leaf nodes that represent overlapping cycles into a set. Note that the equality of cyclic leaf nodes is not based on labels. In the algorithm, the function $\textsc{getPath}(\text{root}, v_{\text{cy}})$ returns a computation path from the root to the node $v_{\text{cy}}$. Therefore $C_{\text{PATH}}$ is the set of computation paths from the root to every cyclic leaf node. The function $\textsc{stretchYield}(C_{\text{p}})$ extends the computation path $C_{\text{p}}$ by going through the cycle several times until the length of the extended computation path is at least $2|Q|+1$. The function then returns the yield of this extended computation path. The function $\textsc{pathRun}(C_{\text{q}}, w)$ runs the string $w$ on the sub-NFA defined by $C_{\text{p}}$. The function returns `true` if the computation does not hang, otherwise it returns `false`. The function $\textsc{lastState}(C_{\text{q}})$ return the last state in the computation path $C_{\text{q}}$.

**Algorithm 6.** Cycle overlap

$C_{\text{PATH}} \leftarrow \bigcup_{v_{\text{cy}} \in V_{\text{cy}}} \text{GETPATH}(\text{root}, v_{\text{cy}})$
$V_{\text{overlap}} \leftarrow \emptyset$
**for all** $C_{\text{p}} \in C_{\text{PATH}}$ **do**
  $W \leftarrow \text{STRETCHEDYIELD}(C_{\text{p}})$
  **for all** $w \in W$ **do**
    $R_{\text{overlap}} \leftarrow \emptyset$
    **for all** $C_{\text{q}} \in C_{\text{PATH}} \setminus \{C_{\text{p}}\}$ **do**
      **if** $\text{PATHRUN}(C_{\text{q}}, w)$ **then**
        $R_{\text{overlap}} \leftarrow R_{\text{overlap}} \cup \text{LASTSTATE}(C_{\text{q}})$
      **end if**
    **end for**
    $V_{\text{overlap}} \leftarrow V_{\text{overlap}} \cup R_{\text{overlap}}$
  **end for**
**end for**

**Constructing the DFA using FEL** Given an NFA and its cycle trees, Algorithm 7 summarizes the construction of its simulating DFA.

**Algorithm 7.** DFA construction

$W \leftarrow \text{YIELD}(N^{\text{FT}})$
$q_0 \leftarrow S$, $Q' \leftarrow \emptyset$
$Q' \leftarrow \{q_0\} \cup Q'$
**for all** $w \in W$ **do**
  $q \leftarrow q_0$
  **for all** $0 < i \leq |w|$ **do**
    $q' \leftarrow \delta'(q, w_i)$
    **if** $q' = \emptyset$ **then**
      $q' = \bigcup_{r \in q} \delta(r, w_i)$
      $Q' \leftarrow \{q'\} \cup Q'$
    **end if**
    $\text{TRANS}(q, q', w_i)$
    $q \leftarrow q'$
  **end for**
  $\text{ACCEPT}(q)$
**end for**

In the algorithm, $N^{\text{FT}}$ is an NFA represented as a forest of cycle trees. The given NFA is $N = (Q, \Sigma, \delta, S, F)$, and the equivalent DFA to be constructed is $M = (Q', \Sigma, \delta', q'_0, F')$. The function $\text{YIELD}(N^{\text{FT}})$ creates simple and extended computation yields from all cycle trees in $N^{\text{FT}}$ and returns $\mathcal{L}'(M)$, the FEL of the DFA $M$. The function $\text{TRANS}(q, q', w_i)$ creates a transition on the symbol $w_i$ from the state $q$ to the state $q'$. The function $\text{ACCEPT}(q)$ marks the state $q$ as an accept state. The algorithm incrementally creates the equivalent DFA. The partial DFA runs each string in $W$ either to its end or until the computation dies in the middle of the string. If the partial DFA consumes the entire string, we mark the last state reached as an accept state. If the computation dies in state $q$, in the middle of a string, we determine the next DFA state, $q'$, as the set of states that the NFA would be in after consuming the next symbol, $w_i$. If $q'$ is not already a state in the partial DFA, we add it. Then we add transition $(q, w_i, q')$, and continue the computation from $q'$.

## 5 Conclusion

We argued that if a finite automaton $M'$ in finite automaton class $B$ simulates another finite automaton $M$ in class $A$, then it is possible to generate the finite exhaustive language, $\mathcal{L}'(M)$, of $M'$ from walks through the diagraph of $M$. Furthermore, if we know the structure of a collection of states of $M$ relative to a single state of $M'$, we can then construct $M'$ by summarizing the behaviour of actual runs of $M$ on all strings in $\mathcal{L}'(M)$. We presented algorithms to construct an FEL from an NFA, and to use the FEL to build a simulating DFA. We noted that the length of the strings in the FEL has an upper bound of $O(e^{\sqrt{n \log n}})$.

The algorithms to construct the FEL in the NFA to DFA conversion problem are somewhat intricate compared to the subset construction. Nevertheless, our finite exhaustive language approach is more general, and we claim that it applies to all simulations of one finite automata class by another. Moreover, it provides a string-based approach to the problem, as opposed to the traditional state-based approach.

For future work, we are in the process of illustrating the concept for two-way finite automata.

## References

1. J. A. Brzozowski and E. Leiss: *On equations for regular languages, finite automata, and sequential networks*. Theoretical Computer Science, 10(1) 1980, pp. 19–35.
2. M. Crochemore, L. Giambruno, and A. Langiu: *On-line construction of a small automaton for a finite set of words*. International Journal of Foundations of Computer Science, 23(2) 2012, pp. 281–301.
3. J. Daciuk: *Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings*, in Lecture Notes in Computer Science, J.-M. Champarnaud and D. Maurel, eds., vol. 2608, 2003, pp. 255 – 261.
4. J. Hopcroft and J. Ullman: *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.
5. E. Landau: *Handbuch der Lehre von der Verteilung der Primzahlen*, vol. 1, Chelsea, 2 ed., 1953.
6. A. Meyer and M. Fischer: *Economy of description by automata, grammars, and formal systems*, in 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 188–191.
7. J.-L. Nicolas: *On Landau's function g(n)*, in The Mathematics of Paul Erdös I, Springer, 1997, pp. 228–240.
8. M. O. Rabin and D. Scott: *Finite automata and their decision problems*. IBM Journal of Research and Development, 3(2) 1959, pp. 114–125.
9. B. Watson: *A new algorithm for the construction of minimal acyclic DFAs*. Science of Computer Programming, 48(2) 2003, pp. 81–97.

# Accelerated Partial Decoding in Wavelet Trees

Gilad Baruch[1], Shmuel T. Klein, and Dana Shapira[2]

[1] Computer Science Department, Bar Ilan University, Israel
[2] Department of Computer Science, Ariel University, Israel
gilad.baruch@gmail.com, tomi@cs.biu.ac.il, shapird@g.ariel.ac.il

**Abstract.** A Wavelet Tree (WT) is a compact data structure which is used in order to perform various well defined operations directly on the compressed form of a file. As *random access* is one of these operations, the underlying file is not needed anymore, and is often discarded because it can be restored, when necessary, by repeated accesses. This paper concentrates on cases in which partial decoding of a contiguous portion of the file, or even its full decoding, is still needed. We show how to accelerate the decoding relative to repeatedly performing random accesses on the consecutive indices. Preliminary experiments on full decoding support the effectiveness of our approach, and present an improvement of about 60% of the run-time.

## 1 Introduction

Research in Lossless Data Compression evolved in different directions over the years as a result of switching the focus between several of its objectives. Initially, the traditional goal was representing the data as compactly as possible, and decompressing the involved data whenever some further processing was desired. In the second stage, the concern was extended to finding a good balance between storage efficiency of the encoded input file and the processing time of the underlying data. This was often achieved using various auxiliary data structures which were especially suitable for obtaining the outcome of well defined specific operations known in advance. One of such common operations is *random access*, which enables direct access to any element of the encoded text. The support of random access causes the compressed text itself to be redundant, so that the compressed text is not needed any more and may be discarded. In case further operations are desired, the original file, or its relevant parts, is reconstructed using random access repeatedly.

One of the compact data structure suggested to cope with operations performed on a compressed file, is the well known *Wavelet Tree* (WT), defined by Grossi et al. [8]. A Wavelet tree $T$ for a text file of $n$ elements drawn from an alphabet $\Sigma$, is a binary tree whose leaves are labeled by the elements of $\Sigma$, and the internal nodes store bitmaps. The contents of the bitmaps is described below. Balanced Wavelet trees can be constructed in $O(n \log |\Sigma|)$ time and require $n \log |\Sigma|(1 + O(1))$ bits.

The data structures associated with a WT for general prefix codes require some amount of additional storage to the memory usage of the compressed file itself. Given a text string of length $n$ over an alphabet $\Sigma$, the space required by Grossi et al.'s implementation can be bounded by $nH_h + O(\frac{n \log \log n}{\log_{|\Sigma|} n})$ bits, for all $h \geq 0$, where $H_h$ denotes the $h$th-order empirical entropy of the text, which is at most $\log |\Sigma|$; processing time is just $O(m \log |\Sigma| + \text{polylog}(n))$ for searching any pattern sequence of length $m$. Multiary Wavelet trees replace the bitmaps by sequences over sublogarithmic sized alphabets in order to reduce the $O(\log |\Sigma|)$ height of binary Wavelet trees, and obtain the same space as the binary ones, but their times are reduced by an

Gilad Baruch, Shmuel T. Klein, Dana Shapira: Accelerated Partial Decoding in Wavelet Trees, pp. 63–70.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

$O(\log\log n)$ factor. If the alphabet $\Sigma$ is small enough, say $|\Sigma| = O(\text{polylog}(n))$, the tree height is a constant and so are the query times.

Various manipulations on the bitmaps of the WT are based on fast implementations of operations known as rank and select. These are defined for any bit vector $B$ and bit $b \in \{0,1\}$ as:

$\mathsf{rank}_b(B,i)$ – **number** of occurrences of $b$ up to and including position $i$; and
$\mathsf{select}_b(B,i)$ – **position** of the $i$th occurrence of $b$ in $B$.

Jacobson [9] showed that rank, on a bit-vector of length $n$, can be computed in $O(1)$ time using $n + O(\frac{n\log\log n}{\log n}) = n + o(n)$ bits. Other efficient implementations for rank and select are due to Raman et al. [18], Okanohara and Sadakane [17], Barbay et al. [1] and Navarro and Providel [16], to list only a few.

Klein and Shapira [11] applied a pruning strategy to WTs based on Fibonacci Codes, so that in addition to supporting improved rank, select and random access to the corresponding Fibonacci encoded file, the size of the Fibonacci based WT is reduced. However, for any finite probability distribution, the compression by a prefix of the Fibonacci code will always be inferior to what can be achieved by a Huffman code.

We therefore suggested in previous research [2] a different method based on pruning a Huffman tree shaped WT according to the underlying *skeleton* Huffman tree [10]. The resulting smaller WT is especially designed to support faster random access for a single index and save memory storage, at the price of less effective rank and select operations, when compared to the original Huffman shaped WTs. The general idea is to apply some cut-off strategy on the internal nodes of the WTs, so that the overhead of the additional storage, used by the data structures for processing the stored bitmaps, is reduced. Moreover, the average path lengths corresponding to the codewords is also decreased, and so is also the average time spent for traversing the paths from the root to the desired leaf, which is the basic processing component used to evaluate random access.

Given a WT, we suggest in this paper to enhance random access for a sequence of consecutive indices, possibly the entire file, via *range* decoding, unlike the acceleration of a single random access in [2]. The main idea is using the dependency between the consecutive indices, rather than repeatedly performing random access on each one independently. Preliminary experiments support the effectiveness of our approach, and present an improvement of about 60% of the run-time.

There are obviously many scenarios in which the partial decoding of a large compressed file is needed, and we shall mention only one example. Searches in large full text retrieval systems are generally not performed by direct pattern matching, but are rather based on so-called *inverted files*, dictionaries and concordances that have been built in a pre-processing stage. To answer a query, rather than scanning the text for the occurrences of the requested terms, the sorted lists of their locations are retrieved from the auxiliary files and are processed according to the Boolean query at hand. This results in a series of pointers $\ell_1, \ldots, \ell_r$ into the given text, and it is only at this stage that the compressed file is accessed, directly at $\ell_1$, then at $\ell_2$, etc. For each of these locations, the user is generally interested in seeing not just the requested terms of the query, but also some of their local contexts. These contexts are known as *snippets* or *KWICs* (KeyWord In Context) [14], and consist of about one or two lines of text. If according to this short excerpt, the user judges that the occurrence is relevant, a larger portion of the file may be decoded. In any case, for a single query,

the compressed file may be accessed at many different locations, and the possibility of partial decoding is critical for this application.

Our paper is organized as follows. Section 2 discusses previous research dealing with random access to files encoded using variable length codes. Section 3 recalls the details of WTs and presents our proposed algorithm for accelerating partial decoding. Section 4 then empirically compares our suggested algorithm to the traditional one. Finally, Section 5 concludes.

## 2 Previous Research

If the text is encoded by using some standard fixed length code, such as ASCII, random access to the $i$th codeword is straightforward for any $i$. However, fixed length codes are wasteful from the storage point of view, and have therefore been replaced in many applications by variable length codes. This may improve the compression performance, but at the price of losing the simple random access, because the beginning position of the $i$th codeword is the sum of the lengths of all the preceding ones.

A possible solution to allow random access is to divide the encoded file into blocks of size $b$ codewords, and to use an auxiliary vector to indicate the beginning of each block. The time complexity of random access depends on the size $b$, as we can begin from the sampled bit address of the $\frac{i}{b}$th block to retrieve the $i$th codeword. This method, known as *sampling*, thus suggests a processing time vs. memory storage tradeoff, since direct access requires decoding $i - \lfloor\frac{i}{b}\rfloor b$ codewords, i.e., less than $b$.

Ferragina and Venturini [5] replace every block of a fixed number $\ell$ of symbols by a single codeword of a Huffman code built according to the frequency of occurrence of the blocks. Their idea is to represent $T$ as a sequence of $\lceil\frac{n}{\ell}\rceil$ macro-symbols over the macro-alphabet $\Sigma^\ell$, where $\ell$ is chosen as $\left\lceil\frac{\log_{|\Sigma|} n}{2}\right\rceil$. To guarantee constant time direct access to the encoding of the blocks, they use a two level storage scheme for the starting positions: absolute ones every $\Theta(\log n)$ contiguous blocks, and relative ones for the rest. Their representation uses $O\left(\frac{n \log\log n}{(\log_{|\Sigma|} n}\right)$ bits.

Teuhola [19] extends Moffat and Stuiver's work [15] on *Interpolative coding*, so that direct access, as well as finding the position in which the prefix sum exceeds some threshold, is achieved in $O(\log n)$ time. They consider the successive gaps in the sequence as basic elements, and build a complete binary tree of pairwise sums with the elements as leaves.

Brisaboa et al. [3] use a variant of a Wavelet tree on Byte-Codes. This induces a 128 or 256-ary tree, rather than a binary one, and the root of the Wavelet tree contains the first *byte*, rather than the first bit, of all the codewords, in the same order as they appear in the original text. The second level nodes then store the second byte of the corresponding codewords, and so on. The reordering of the compressed text bits becomes an implicit index representation of the text, which is empirically shown to be better than explicit main memory inverted indexes, built on the same collection of words, when using the same amount of space.

In another work, Brisaboa et al. [4] introduced directly accessible codes (DACs), in which the codewords represent integers. The corresponding Wavelet tree is similar to the one constructed for the byte codes, working with blocks instead of bytes.

Külekci [13] suggested the usage of Wavelet trees for *Elias* and *Rice* variable length codes. The method is based on handling separately the unary and binary parts of the

codeword in different strings so that random access is supported in constant time. As an alternative, the usage of a WT over the lengths of the unary section of each Elias or Rice codeword is proposed, while storing their binary section, allowing direct access in time $\log r$, where $r$ is the number of distinct unary lengths in the file.

## 3 Accelerating partial decoding for WTs

The binary tree corresponding to a prefix code $C$ is defined as follows: we imagine that every edge pointing to a left child is labeled 0 and every edge pointing to a right child is labeled 1; each node $v$ is associated with the bit string obtained by concatenating the labels on the edges on the path from the root to $v$; finally, the tree is defined as the binary tree for which the set of bit strings associated with its leaves is the code $C$.

As mentioned above, the nodes of the WT are annotated by bitmaps. The WT reorders the bits of the compressed file into an alternative form, thereby enabling direct access, as well as rank and select. Wavelet trees can be defined for any prefix code, and the tree structure associated with this code is inherited by the WT. A WT $T$ for an array $A = A[1]A[2]\cdots A[n]$ of $n$ elements, drawn from an alphabet $\Sigma$, is a binary tree whose leaves are labeled by the elements of $\Sigma$, and the internal nodes store bitmaps. The bitmap at the root contains $n$ bits, in which the $i^{th}$ bit is set to 0 or 1 depending on whether $A[i]$ is the label of a leaf that is stored in the left or right subtree of $T$. Each internal node $v$ of $T$, is itself the root of a Wavelet tree $T_v$ for the *subarray* of $A$ consisting only of the labels of the leaves of $T_v$, which are not necessarily consecutive elements of the array $A$.

These bitmaps can be stored as a single bit stream by concatenating them in order of any predetermined top-down tree traversal, such as depth-first or breadth-first. No delimiters between the individual bitmaps are required, since we can restore the tree topology along with the bitmaps lengths at each node once the size $n$ of the text is given in the header of the file. Figure 1 depicts the WT induced by the Huffman tree for the example text $T$ =A--HUFFMAN--WAVELET--TREE--MATTERS. The WT is the entire figure including the annotating bitmaps.

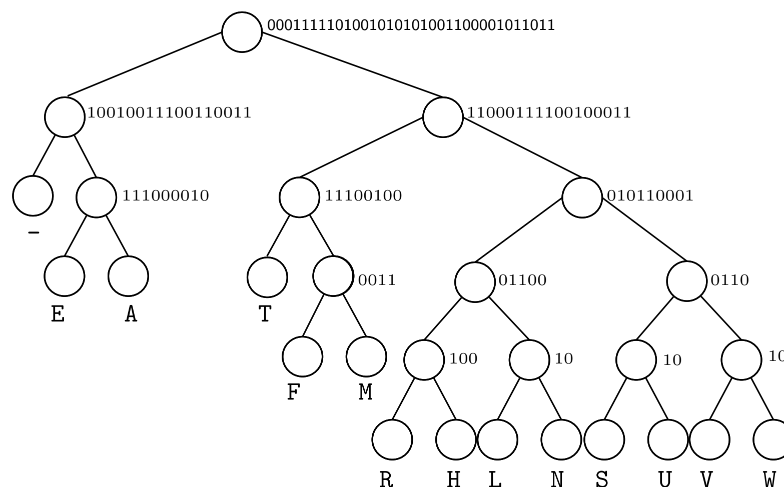



**Figure 1.** The WT induced by the Huffman tree corresponding to the frequencies {8,5,4,4,2,2,2,1,1,1,1,1,1,1} of {-,E,A,T,F,M,R,H,L,N,S,U,V,W}, respectively, assigned to the leaves, left to right.

The algorithm for extracting the $i$-th element of the text $T$ by means of a WT rooted by $v_{root}$ is given in Figure 2, using the function call extract($v_{root}$,$i$). $B_v$ denotes the bitmap belonging to vertex $v$ of the WT, and $\cdot$ denotes concatenation. Computing the new index in the following bitmap is done by the rank operation in lines 2.1.2 and 2.2.2. The decoding of the codeword $cw$ in line 3 by means of the decoding function $\mathcal{D}$ can be done by a preprocessed lookup table.

```
extract(v, i)
1       cw ⟵ ε
2       while v is not a leaf
2.1         if B_v[i] = 0 then
2.1.1           cw ⟵ cw · 0
2.1.2           i ⟵ rank_0(B_v, i)
2.1.3           v ⟵ left(v)
2.2         else
2.2.1           cw ⟵ cw · 1
2.2.2           i ⟵ rank_1(B_v, i)
2.2.3           v ⟵ right(v)
3       return D(cw)
```

**Figure 2.** Extracting the $i$-th element of $T$ from a WT rooted at $v$.

The straightforward decoding algorithm works on successive indices *independently*, starting each time at the root, and working its way down the WT until a leaf is reached, where the information for that index is extracted. The formal algorithm for partial decoding of a range of elements with indices between $i$ and $j$ is given in Figure 3, where the decoding is output to an array $A$.

```
range_decoding(i, j)
1       for k = i to j
1.1         A[k − i] ⟵ extract(root, k)
2       return A
```

**Figure 3.** Traditional range decoding.

Unlike the traditional approach, the proposed algorithm takes advantage of the fact that partial decoding is applied on a strictly monotonic increasing series of indices. During runtime, partial calculations are stored so that the same computations are not done more than once. A similar idea is performed in the well known KMP algorithm [12] for pattern matching, in which the algorithm makes sure that it does not match any character more than once.

In spite of the fact that there exist constant time solutions for rank and select that require sublinear extra space, in many practical cases, simple solutions are better in terms of time and space [7]. Thus, in order to save space, the rank operation in lines 2.1.2 and 2.2.2 is not necessarily done in $O(1)$ time. In either case, it can even be done faster using the fact that the ranks of consecutive zeros or consecutive ones in a given bitvector differ only by one. More precisely, if for indices $i$ and $j$, it holds that $B_v[i] = 0$ and $B_v[j] = 0$, but for each index $k$ between $i$ and $j$, $B_v[k] \neq 0$, then

$$\mathsf{rank}_0(B_v, j) = \mathsf{rank}_0(B_v, i) + 1.$$

For this reason the rank results are maintained for each internal node of the WT which has already been visited during the production of the solution of the current range decoding query.

Each time a node is visited for the first time, the rank queries in lines 2.1.2 and 2.2.2 of Figure 3 are fully computed for the corresponding bit using the rank/select data structures. The resulting value is then stored at the node for future use. If, during the computation of extract($i$), a node is reached that has already been visited, the stored value is extracted and incremented, rather than recalculated from scratch by the rank operation, as done for the first time. In the special case of full decoding, the rank results for all nodes are initialized by zero and none of them are obtained by means of the rank/select data structure.

Since $\mathsf{rank}_{1-b}(B,i) = i - \mathsf{rank}_b(B,i)$, only one of the two, say, $\mathsf{rank}_0(B,i)$ needs to be stored. We denote the stored value in node $v$ by $rnk(v)$. At allocation time of a new node, its $rnk$ value will be initialized by -1. The line $i \longleftarrow \mathsf{rank}_0(B_v,i)$ in Figure 2 is replaced by the top half of the code of Figure 4 indicated by 2.1.2, whereas the line $i \longleftarrow \mathsf{rank}_1(B_v,i)$ in Figure 2 is replaced by the bottom half of the code, indicated by 2.2.2.

```
2.1.2   if rnk(v) < 0   // first visit at v
            i ⟵ rank_0(B_v, i)
        else
            i ⟵ rnk(v) + 1
        rnk(v) ⟵ i

2.2.2   if rnk(v) < 0   // first visit at v
            rnk(v) ⟵ rank_0(B_v, i)
            i ⟵ i − rnk(v)
        else
            i ⟵ (i − rnk(v)) + 1
```

**Figure 4.** Partial decoding acceleration.

Note that the two parts are not completely symmetrical. The upper part, 2.1.2, corresponds to a 0-bit, so the assignment to $rnk(v)$ is excluded from the if-clause, as it has to be performed on any visit to the node $v$. The lower part, 2.2.2, corresponds to a 1-bit, thus the value of $rnk(v)$ is only set at the first visit to the node, since it does not change on recurring visits: $\mathsf{rank}_0(B_v,i) = \mathsf{rank}_0(B_v,i-1)$ when the $i$th bit is 1.

## 4 Experimental Results

We considered four texts of different languages and alphabet sizes. *ftxt* is the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [20]; *ebib* is the Bible (King James version) in English, in which the text was stripped of all punctuation signs; *English* is the concatenation of English text files selected from the Gutenberg Project; and *dblp* is an XML file providing bibliographic information on major Computer Science journals and proceedings, obtained from `dblp.uni-trier.de`. Our implementation used the *Succinct Data Structure Library* [6].

Table 1 presents some information on the data files involved. The second column presents the original file sizes in MB, and the third column gives the sizes of the alphabets.

| File | size (MB) | $\|\Sigma\|$ |
|---|---|---|
| *ftxt* | 7.6 | 132 |
| *ebib* | 3.5 | 53 |
| *English* | 200.0 | 225 |
| *dblp* | 200.0 | 96 |

**Table 1.** Information about the used datasets

For our preliminary experiments we considered several variants of the WT with different topology and different rank data structure implementations. As all variants produced basically the same results for full decoding, we present here the ones for the Huffman based WT and rank implementation of Vigna [21], and leave the other reports for the full version of the paper.

Table 2 compares the processing times of full decoding of the traditional approach to that of our algorithm. The second and third columns give the processing time, in seconds, of the traditional and the proposed algorithm, respectively, and the fourth column is the ratio of the latter to the former. The experiments were conducted on a machine running 64 bit Linux Ubuntu with an Intel Core i7-4720 at 2.60GHz processor, 6144K L3 cache size of the CPU, and 4GB of main memory.

| File | traditional | proposed method | ratio |
|---|---|---|---|
| ftxt | 1.87 | 0.72 | 0.38 |
| ebib2 | 0.66 | 0.27 | 0.4 |
| english | 28.21 | 12.32 | 0.44 |
| dblp | 36.52 | 16.42 | 0.45 |

**Table 2.** Full decoding processing time comparison.

As can be seen, our method is about 60% faster, and consistently achieves a significant processing time improvement relative to the traditional approach,

## 5 Conclusion

We have presented an enhanced range decoding especially designed for WTs, and gave empirical evidence that the running time performance of full decoding is significantly improved as compared to the running time of the traditional decoding. Our improvement can be implemented without any additional storage: the *rnk* values are generated and used only during run time and needs only $O(\Sigma)$ bytes of RAM, which is independent of the size of the text.

For partial range decoding, the suggested approach might be a bit slower, because of the extra if checking for the existence of the rank result in cache for every visited node, rather then computing the rank completely using the rank/select data structure straight away. However, we believe that in case of revisiting many nodes, the overhead of this extra if will vanish in the more time consuming rank computation which is then avoided.

## References

1. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich: *Alphabet partitioning for compressed rank/select and applications*. Algorithms and Computation, Lecture Notes in Computer Science, 6507 2010, pp. 315–326.
2. G. Baruch, S. T. Klein, and D. Shapira: *A space efficient direct access data structure*, in Proc. of Data Compression Conference DCC–2016, 2016, pp. 63–72.
3. N. R. Brisaboa, A. Fariña, G. Ladra, and G. Navarro: *Reorganizing compressed text*, in Proc. of the 31th Annual Internetional ACM SIGIR Conference on Research and Developing in Information Retrieval (SIGIR), 2008, pp. 139–146.
4. N. R. Brisaboa, S. Ladra, and G. Navarro: *DACs: Bringing direct access to variable length codes*. Information Processing and Management, 49(1) 2013, pp. 392–404.
5. P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds*. Theoretical Computer Science, 372 2007, pp. 115–121.
6. S. Gog, T. Beller, A. Moffat, and M. Petri: *From theory to practice: plug and play with succinct data structures*, in International Symposium on Experimental Algorithms (SEA 2014), 2014, pp. 326–337.
7. R. González, S. Grabowski, V. Mäkinen, and G. Navarro: *Practical implementation of rank and select queries*, in Poster Proceedings of 4th Workshop on Efficient and Experimental Algorithms (WEA05), 2005, pp. 27–38.
8. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.
9. G. Jacobson: *Space efficient static trees and graphs*, in Proceedings of FOCS, 1989, pp. 549–554.
10. S. T. Klein: *Skeleton trees for the efficient decoding of Huffman encoded texts*. in the Special issue on Compression and Efficiency in Information Retrieval of the Kluwer Journal of Information Retrieval, 3 2000, pp. 7–23.
11. S. T. Klein and D. Shapira: *Random access to Fibonacci codes*, in The Prague Stringology Conference PSC-2014, 2014, pp. 96–109.
12. D. E. Knuth, J. H. Morris, and V. Pratt: *Fast pattern matching in string*. SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
13. M. O. Külekci: *Enhanced variable-length codes: Improved compression with efficient random access*, in Proc. Data Compression Conference DCC–2014, Snowbird, Utah, 2014, pp. 362–371.
14. H. P. Luhn: *Keyword-in-context index for technical literature*. American Documentation, 11(4) 1960, pp. 288–295.
15. A. Moffat and L. Stuiver: *Binary interpolative coding for effective index compression*. Information Retrieval, 3(1) 2000, pp. 25–47.
16. G. Navarro and E. Providel: *Fast, small, simple rank/select on bitmaps*. Experimental Algorithms, LNCS, 7276 2012, pp. 295–306.
17. D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proc. ALENEX, SIAM, 2007.
18. R. Raman, V. Raman, and S. Rao Satti: *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets*. Transactions on Algorithms (TALG), 2007, pp. 233–242.
19. J. Teuhola: *Interpolative coding of integer sequences supporting log-time random access*. Information Processing and Management, IP & M, 47(5) 2011, pp. 742–761.
20. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The arcade project*. Parallel Text Processing, 2000, pp. 369–388.
21. S. Vigna: *Broadword implementation of rank/select queries*, in Proc. of 7th Workshop on Experimental Algorithms (WEA), 2008, pp. 154–168.

# A Family of Data Compression Codes with Multiple Delimiters

Igor O. Zavadskyi and Anatoly V. Anisimov

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorza@gmail.com

**Abstract.** A new family of perspective variable length self-synchronazable binary codes with multiple pattern delimiters is introduced. Each delimiter consists of a run of consecutive ones surrounded by zero brackets. These codes are complete and universal. A simple bijective correspondence between natural numbers and any multi-delimiter code set is established. A fast byte aligned decoding algorithm is constructed. Comparisons of text compression rate and decoding speed for different multi-delimiter codes, the Fibonacci code Fib3 and $(s, c)$-dense codes are also presented.

**Keywords:** prefix code, Fibonacci code, data compression, robustness, completeness, universality, density, multi-delimiter

## 1 Introduction

For the last few decades, the data compression technology has accumulated a substantial arsenal of powerful string processing methods. For details, we refer to the book by D. Salomon [11]. The present period of the information infrastructure development actualizes the demand for efficient data compression methods that on one hand provide satisfactory compression rate, and, on the other, support fast encoding, decoding and search in compressed data. Along with this the need for a code robustness in the sense of limiting possible error propagations has also been strengthened.

As is known, the classical Huffman codes provide good compression efficiency approaching to the theoretically best [8]. Unfortunately, Huffman's encoding does not allow the direct search in compressed data by a given compressed pattern. At the expense of losing some compression efficiency this was amended by introducing byte aligned tagged Huffman codes: Tagged Huffman Codes [5], End-Tagged Dense Codes (ETDC) [4] and (s,c)-dense codes (SCDC) [3]. In these methods codewords are represented as sequences of bytes, which along with encoded information incorporate flags for the end of a codeword.

The alternative approach for coding stems from using the Fibonacci numbers of higher orders. The mathematical study of Fibonacci codes was started in the pioneering paper [2]. The authors first introduced families of Fibonacci codes of higher orders with the emphasis on their robustness. Also, they proved completeness of these codes and their universality in the sense of [6].

The most strong argumentation for the use of Fibonacci codes of higher orders in data compression was given in [10]. For these codes, the authors developed fast byte aligned algorithms for decoding and search in the compressed text [9]. They also showed that Fibonacci codes have better compression efficiency comparing with ETDC and SCDC codes for middle size text corpora while still being inferior on decompression and search speed.

Igor O. Zavadskyi, Anatoly V. Anisimov: A Family of Data Compression Codes with Multiple Delimiters, pp. 71–84.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

Another advantage of Fibonacci codes over ETDC, SCDC and Huffman codes is their robustness in the sense of limiting possible error propagations. Although SCDC codes may limit the propagation of errors coursed by bit erroneous inversions, they are completely not resistant to insertions or deletions of bits. Huffman's codes are vulnerable to any of these errors. Whereas in Fibonacci codes errors coursed by a single bit inversion, deletion or insertion cannot propagate over more than two adjacent codewords. In other words, they are synchronizable with synchronization delay at most one codeword.

In this presentation, we study a new family of binary codes with multiple suffix delimiters. These codes were first introduced in [1]. Each delimiter consists of a run of consecutive ones surrounded with zero brackets. Thus, delimiters have the form $01\cdots10$. A number of ones in delimiters is defined by a given fixed set of positive integers $m_1, \ldots, m_t$. The multi-delimiter code $D_{m_1,\ldots,m_t}$ consists of $t$ words $11\cdots10$ with $m_1, \ldots, m_t$ ones and all other words in which delimiters occur only as a suffix. For example, the multi-delimiter code $D_{2,3}$ consists of words 110, 1110 and all other words in which 110 or 1110 occurs only as a suffix, e.g. 0110, 01110, 10110, etc.

By their properties, the multi-delimiter codes are close to the Fibonacci codes of higher orders. Due to robust delimiters, multi-delimiter codes are synchronizable with synchronization delay at most one codeword, as well as Fibonacci codes. We prove completeness and universality of such codes. There also exists a bijection between any code $D_{m_1,\ldots,m_t}$ and the set of natural numbers. This bijection is implemented by very simple encoding and decoding procedures. For practical use we present a byte aligned decoding algorithm with better computational complexity than that of Fibonacci codes.

Each of ETDC, SCDC, Fibonacci and multi-delimiter codes is well suited for natural language text compression if words of a text are considered as atomic symbols. As shown in [10], the Fibonacci code of order three, denoted by Fib3, has the best compression rate when applied to this kind of data. From our study, it follows that the simple code $D_2$ with one delimiter 0110 has asymptotically higher density as against Fib3, although it is slightly inferior in compression rate for realistic alphabet sizes of natural language texts.

We also note that by varying delimiters for better compression we can adapt multi-delimiter codes to a given probability distribution and an alphabet size. Thus, for example, we compare the codes $D_{2,3}$, $D_{2,3,5}$ and $D_{2,4,5}$ with the code Fib3. Those multi-delimiter codes are asymptotically less dense than Fib3. Nevertheless, in practice the alphabet size of a text is often relatively small, from a few thousand up to a few million words. For such texts the aforementioned multi-delimiter codes outperform the Fib3 code in compression rate by $2-3\%$, while both Fibonacci and multi-delimiter codes significantly outperform the ETDC/SCDC codes.

The structure of the presentation is as follows. In Section 2 we define the family of multi-delimiter codes and discuss their density. A bijective correspondence between the set of natural numbers and codewords of any code $D_{m_1,\ldots,m_t}$ is established in the next section. Also, herein we present the bitwise encoding/decoding algorithms. The completeness and universality of multi-delimiter codes is proven in Section 4. The fast byte aligned decoding algorithm for the multi-delimiter code $D_{2,3,5}$ is given in Section 5. This code appears to be the most efficient in compression among all multi-delimiter codes when applied to small or mid-size texts. In Section 6 we present the results of computational experiments to compare the compression rate and decoding time of

| Index | Fib2 | $D_1$ | $D_{1,2}$ | Fib3 | $D_2$ | $D_{2,3}$ | $D_{2,3,4}$ |
|---|---|---|---|---|---|---|---|
| 1 | 11 | 10 | 10 | 111 | 110 | 110 | 110 |
| 2 | 011 | 010 | 010 | 0111 | 0110 | 0110 | 0110 |
| 3 | 0011 | 0010 | 110 | 00111 | 00110 | 1110 | 1110 |
| 4 | 1011 | 00010 | 0010 | 10111 | 10110 | 00110 | 00110 |
| 5 | 00011 | 11010 | 0110 | 000111 | 000110 | 10110 | 10110 |
| 6 | 01011 | 000010 | 00010 | 010111 | 010110 | 01110 | 01110 |
| 7 | 10011 | 011010 | 00110 | 100111 | 100110 | 000110 | 11110 |
| 8 | 000011 | 110010 | 000010 | 110111 | 0000110 | 010110 | 000110 |
| 9 | 001011 | 111010 | 000110 | 0000111 | 0010110 | 100110 | 010110 |
| 10 | 010011 | 0000010 | 111010 | 0010111 | 0100110 | 001110 | 100110 |
| 11 | 100011 | 0011010 | 0000010 | 0100111 | 1000110 | 101110 | 001110 |
| 12 | 101011 | 0110010 | 0000110 | 1000111 | 1010110 | 0000110 | 101110 |
| 13 | 0000011 | 1100010 | 0111010 | 1010111 | 1110110 | 0010110 | 011110 |
| 14 | 0001011 | 0111010 | 1110010 | 0110111 | | 0100110 | 0000110 |
| 15 | 0010011 | 1110010 | 1110110 | 1100111 | | 1000110 | 0010110 |
| 16 | 0100011 | 1111010 | 1111010 | | | 1010110 | 0100110 |
| 17 | 1000011 | | | | | 0001110 | 1000110 |
| 18 | 0101011 | | | | | 0101110 | 1010110 |
| 19 | 1001011 | | | | | 1001110 | 0001110 |
| 20 | 1010011 | | | | | | 0101110 |
| 21 | | | | | | | 1001110 |
| 22 | | | | | | | 0011110 |
| 23 | | | | | | | 1011110 |

**Table 1.** Sample codeword sets of multi-delimiter and Fibonacci codes

SCDC, Fibonacci and multi-delimiter codes. And in the last section we summarize the advantages of multi-delimiter codes.

## 2 Definition of multi-delimiter codes

Let $\mathcal{M} = \{m_1, \ldots, m_t\}$ be a set of integers, given in ascending order, $0 < m_1 < \cdots < m_t$.

**Definition 1** *The multi-delimiter code $D_{m_1,\ldots,m_t}$ consists of all the words of the form $1^{m_i}0, i = 1, \ldots, t$ and all other words that meet the following requirements:*

*(i) for any $m_i \in \mathcal{M}$ a word does not start with a sequence $1^{m_i}0$;*
*(ii) a word ends with the suffix $01^{m_i}0$ for some $m_i \in \mathcal{M}$;*
*(iii) for any $m_i \in \mathcal{M}$ a word cannot contain the sequence $01^{m_i}0$ anywhere, except a suffix.*

The given definition implies that code delimiters in $D_{m_1,\ldots,m_t}$ are sequences of the form $01^{m_i}0$. However, the code also contains shorter words of the form $1^{m_i}0$, which form a delimiter together with the ending zero of a preceding codeword.

The sample of codewords of the length $\leq 7$ for some multi-delimiter codes and, for comparison, some Fibonacci codes is given in Table 1. As is seen, the codes $D_{2,3}$ and $D_{2,3,4}$ with 2 and 3 delimiters respectively contain many more short codewords than both the Fibonacci code Fib3 and the one-delimiter code $D_2$. This is an important factor when considering the compression efficiency.

We calculate the number of short codewords for several multi-delimiter codes that are potentially suitable for natural language text compression. Also, we calculate

| Code | The number of codewords of length $\leq n$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Asymptotic | $n=2$ | $n=3$ | $n=4$ | $n=5$ | $n=6$ | $n=7$ | $n=8$ | $n=15$ |
| The codes with the shortest codeword of length 2 | | | | | | | | | |
| Fib2 | $1.618^n$ | 1 | 2 | 4 | 7 | 12 | 20 | 33 | 986 |
| $D_1$ | $1.755^n$ | 1 | 2 | 3 | 5 | 9 | 16 | 28 | 1432 |
| $D_{1,2}$ | $1.618^n$ | 1 | 3 | 5 | 7 | 10 | 16 | 27 | 799 |
| $D_{1,3}$ | $1.674^n$ | 1 | 2 | 4 | 7 | 11 | 18 | 30 | 1106 |
| The codes with the shortest codeword of length 3 | | | | | | | | | |
| Fib3 | $1.839^n$ | 0 | 1 | 2 | 4 | 8 | 15 | 28 | 2031 |
| $D_2$ | $1.867^n$ | 0 | 1 | 2 | 4 | 7 | 13 | 24 | 1906 |
| $D_{2,3}$ | $1.785^n$ | 0 | 1 | 3 | 6 | 11 | 19 | 33 | 1874 |
| $D_{2,4}$ | $1.823^n$ | 0 | 1 | 2 | 5 | 9 | 17 | 30 | 1998 |
| $D_{2,5}$ | $1.844^n$ | 0 | 1 | 2 | 4 | 8 | 15 | 28 | 1999 |
| $D_{2,3,4}$ | $1.731^n$ | 0 | 1 | 3 | 7 | 13 | 23 | 39 | 1721 |
| $D_{2,4,5}$ | $1.796^n$ | 0 | 1 | 2 | 5 | 10 | 19 | 34 | 2019 |
| $D_{2,4,6}$ | $1.809^n$ | 0 | 1 | 2 | 5 | 9 | 18 | 32 | 2032 |
| The codes with the shortest codeword of length 4 | | | | | | | | | |
| Fib4 | $1.928^n$ | 0 | 0 | 1 | 2 | 4 | 8 | 16 | 1606 |
| $D_3$ | $1.933^n$ | 0 | 0 | 1 | 2 | 4 | 8 | 15 | 1510 |

**Table 2.** The number of codewords of multi-delimiter and Fibonacci codes

the asymptotic densities of these codes using the standard technique of generating functions. The results are presented in Table 2.

In general, codes with more delimiters contain more short words, although they have worse asymptotic density. This regularity is also related to lengths of delimiters: the shorter they are the larger quantity of short words a code contains. Considering the application for text compression, the most efficient seems to be the codes $D_{2,\ldots}$, which we thoroughly investigate.

## 3 Encoding integers

We define a multi-delimiter code as a set of words. There exists a simple bijection between this set and the set of natural numbers. This bijection allows us to encode integers.

Let $\mathcal{M} = \{m_1, \ldots, m_t\}$ be the set of parameters of the code $D_{m_1,\ldots,m_t}$. By $N_{\mathcal{M}} = \{j_1, j_2, \ldots\}$ denote the ascending sequence of all natural numbers that do not belong to $\mathcal{M}$.

By $\varphi_{\mathcal{M}}(i)$ denote the function $\varphi_{\mathcal{M}}(i) = j_i$, $j_i \in N_{\mathcal{M}}$ as defined above.

It is easy to see that the function $\varphi_{\mathcal{M}}$ is a bijective mapping of the set of natural numbers onto $N_{\mathcal{M}}$. Evidently, this function and the inverse function $\varphi_{\mathcal{M}}^{-1}$ can be constructively implemented by simple constant time procedures.

A run of consecutive ones in a word $w$ is called *isolated* if it is a prefix of this word ending with zero, or it is its suffix starting with zero, or it is a substring of $w$ surrounded with zeros, or it coincides with $w$.

The main idea of encoding integers by the code $D_{m_1,\ldots,m_t}$ is as follows. We scan the binary representation of an integer from left to right. During this scan each isolated group of $i$ consecutive $1s$ is changed to $\varphi_{\mathcal{M}}(i)$ isolated $1s$. This way we exclude the appearance of delimiters inside a codeword. In decoding, we change internal isolated groups of $j$ consecutive $1s$ to groups of $\varphi_{\mathcal{M}}^{-1}(j)$ ones. The detailed description of the encoding procedure is as follows.

***Bitwise Integer Encoding Algorithm***

*Input*: an integer $x = x_n x_{n-1} \cdots x_0$, $x_i \in \{0,1\}, x_n = 1$;
*Result*: the corresponding codeword from $D_{m_1,\ldots,m_t}$.

1. $x \leftarrow x - 2^n$, i.e. extract the most significant bit of the number $x$, which is always 1.
2. If $x = 0$, append the sequence $1^{m_1}0$ to the string $x_{n-1} \cdots x_0$, which contains only zeros or empty. *Result* $\leftarrow x_{n-1} \cdots x_0 1^{m_1}0$. Stop.
3. If the binary representation of $x$ takes the form of a string $0^r 1^{m_i} 0, r \geq 0, m_i \in \mathcal{M}, i > 1$, then *Result* $\leftarrow x$. Stop.
4. In the string $x$ replace each isolated group of $i$ consecutive $1s$ with the group of $\varphi_{\mathcal{M}}(i)$ consecutive $1s$ except its occurrence as a suffix of the form $01^{m_i}0, i > 1$. Assign this new value to $x$.
5. If the word ends with a sequence $01^{m_i}0, i > 1$, then *Result* $\leftarrow x$. Stop.
6. Append the string $01^{m_1}0$ to the right end of the word. Assign this new value to $x$. *Result* $\leftarrow x$. Stop.

According to this algorithm, if $x \neq 2^n$, the delimiter $01^{m_1}0$ with $m_1$ ones does not contain information bits, and therefore it should be deleted during the decoding. However, delimiters of the form $01^{m_i}0, i > 1$ are informative parts of codewords, and they must be processed during the decoding. If $x = 2^n$, the last $m_1 + 1$ bits of the form $1^{m_1}0$ must be deleted.

***Bitwise Decoding Algorithm***

*Input*: a codeword $y \in D_{m_1,\ldots,m_t}$.
*Result*: the integer given in the binary form which encoding results in $y$.

1. If the codeword $y$ is of the form $0^p 1^{m_1} 0$, where $p \geq 0$, extract the last $m_1 + 1$ bits and go to step 4.
2. If the codeword $y$ ends with the sequence $01^{m_1}0$, extract the last $m_1 + 2$ bits. Assign this new value to $y$.
3. In the string $y$ replace each isolated group of $i$ consecutive $1s$, where $i \notin \mathcal{M}$, with the group of $\varphi_{\mathcal{M}}^{-1}(i)$ consecutive $1s$. Assign this new value to $y$.
4. Prepend the symbol 1 to the beginning of $y$. *Result* $\leftarrow y$. Stop.

Let us give the example. We encode the number $14 = 1110_2$ using the code $D_{2,3}$. For this code, $\mathcal{M} = \{2,3\}$, $N_{\mathcal{M}} = \{1,4,5,\ldots\}$, $\varphi_{\mathcal{M}}(2) = 4$, $\varphi_{\mathcal{M}}(3) = 5$, $\varphi_{\mathcal{M}}(4) = 6$ etc.

1. Extracting the most significant bit we obtain the number 110.
2. 110 is the isolated group of ones. Replace it with the isolated group of $\varphi_{\mathcal{M}}(2) = 4$ ones, i.e. 11110.
3. Appending the string $01^{m_1}0$ to the right end of the word we get the result 111100110.

Now let us decode the codeword 111100110.

1. Extracting the last $m_1 + 2$ bits we obtain the number 11110.
2. Replace the isolated group of 4 ones in the beginning of the codeword with the isolated group of $\varphi_{\mathcal{M}}^{-1}(4) = 2$ ones: 110.
3. Prepend the symbol 1 to the beginning of the word: 1110.

## 4 Some general properties of multi-delimiter codes

Evidently, any multi-delimiter code is prefix-free and thus uniquely decodable (UD). However, this fact can be proved formally by checking the Kraft inequality. If it holds as the equality, the code is also complete, which means that no codeword can be added to a code in a way that preserves the UD property.

**Theorem 1.** *Each multi-delimiter code $D_{m_1,\ldots,m_t}$ is uniquely decodable, complete and universal.*

*Proof.* Completeness and UD property of any multi-delimiter code $D_{m_1,\ldots,m_t}$ is proved in [1]. The proof is based on checking the Kraft equality.

Let us prove the universality of multi-delimiter codes. The notion of universality was introduced by P. Elias [6] to reflect the property of a code to be nearly optimal for data sources with any given probability distribution. Formally this means that there exists a constant $K$ such that for any finite distribution of probabilities $P = (p_1, \ldots, p_n)$, where $p_1 \geq p_2 \geq \ldots$, the inequality $\sum_{i=1}^{n} l_i p_i \leq K \cdot \max(1, E(P))$ holds true, where $E(P) = -\sum_{i=1}^{n} p_i \log_2 p_i$ is entropy of distribution $P$ and $l_i$ are codeword lengths.

Note that encoding procedure that transforms a number $x$ into the corresponding codeword of the code $D_{m_1,\ldots,m_t}$ can enlarge each internal isolated group of sequential 1s in the binary representation of $x$ to a maximum of $t$ ones. The quantity of such groups does not exceed $\frac{1}{2}\log_2 x$. To some binary words the delimiter $01^{m_1}0$ could be externally appended, while the leftmost 1 is always deleted. Therefore the length of the codeword is upper bounded by the value $(\frac{1}{2}t + 1)\log_2 x + m_1 + 1$.

Let us sort codewords from $D_{m_1,\ldots,m_t}$ in ascending order of their bit lengths: $a_1, a_2, \ldots$ We map them to symbols of the input alphabet sorted in descending order of their probabilities. Evidently, the correspondence between an integer and the length of its codeword is not monotonic. Nevertheless, there are at least $i$ words of lengths that do not exceed the upper bound for $i$, which is equal to $(\frac{1}{2}t + 1)\log_2 i + m_1 + 1$. Thus, the length of $a_i$ does not exceed this bound too. To conclude the proof it only remains to apply general Lemma 6 by Apostolico and Fraenkel taken from [2]. "Let $\psi$ be a binary representation such that $|\psi(k)| \leq c_1 + c_2 \log k$ $(k \in \mathbb{Z}^+)$, where $c_1$ and $c_2$ are constants and $c_2 > 0$. Let $p_k$ be the probability to meet $k$. If $p_1 \geq p_2 \geq \cdots \geq p_n$, $\Sigma p_i \leq 1$ then $\psi$ is universal." □

## 5 Fast decoding

The value of a code depends not only on compression rate, but on a number of other properties. And not least of all it concerns the time of compression and decompression. The decompression time is more critical than the time of compression. That is why in this presentation we only concentrate on the accelerating of decoding.

The aforementioned encoding and decoding algorithms are bitwise, and thus, they are quite slow. To accelerate them we construct a byte aligned lookup table method, which performs the same mapping as the bitwise decoding algorithm. The main idea of the proposed method is similar to that developed in [10]. At each iteration, the algorithm processes some parameters from a table row, which examples are given in Table 3. The choice of a row depends on two parameters listed in the left two columns of the table. They are a byte read from the input (column 2) and a value $r$

which depends on bits left unprocessed at the previous iteration (column 1). These two parameters can be considered as indices of the two-dimensional array $TAB$ containing all decoded numbers which can be extracted from a current byte and also some other parameters.

As shown, the code $D_{2,3,5}$ has one of the best compression rates comparing with other multi-delimiter codes. Therefore, for this code we give the detailed description of the decoding algorithm. We consider the simplest one byte variant, i.e. processing 8 bits per iteration. It is not difficult to extend considered constructions to any other multi-delimiter code and to other number of bytes. The table-driven decoding algorithm is given below. Its parameters have the following meanings:

$w_1, w_2, w_3, w_4$ - decoded numbers that can be extracted at the current iteration.
$l_1$ - the bit length of a number $w_1$.
$g$ - the number of codewords for which decoding is finished at the current iteration.
$w$ - a partially decoded number. We use this variable to transfer decoded bits from iteration to iteration when some codeword is split among bytes.
$r_{prev}$ - an index, which depends on the bits left unprocessed at the previous iteration.
$r$ - an index, which depends on the bits left unprocessed at the current iteration.
$Text$ - a coded text.
$Dict$ - the dictionary that maps the decoded numbers to the words of the input text.
$TAB$ - the array containing values dependent on the remainder $r_{prev}$ and the next byte of the code.

***Fast Byte Aligned Decoding Algorithm***

*Input*: a coded $Text$.
*Result*: the sequence of integers.

1. $w \leftarrow 1$, $r_{prev} \leftarrow 0$, $i \leftarrow 0$
2. **while** $i <$ *length of encoded text*
3. $\quad (g, w_1, w_2, w_3, w_4, r, l_1) \leftarrow TAB[r_{prev}][Text[i]]$
4. $\quad w \leftarrow (w << l_1)|w_1$ // append the $w_1$ bits to the right of $w$
5. $\quad$ **if** $g > 0$
6. $\quad\quad$ **output** $Dict(w)$
7. $\quad\quad$ **if** $g > 1$
8. $\quad\quad\quad$ **output** $Dict(w_2)$
9. $\quad\quad\quad$ **if** $g > 2$
10. $\quad\quad\quad\quad$ **output** $Dict(w_3)$
11. $\quad\quad\quad\quad$ $w \leftarrow w_4$
12. $\quad\quad\quad$ **else**
13. $\quad\quad\quad\quad$ $w \leftarrow w_3$
14. $\quad\quad$ **else**
15. $\quad\quad\quad$ $w \leftarrow w_2$
16. $\quad r_{prev} \leftarrow r$
17. $\quad i \leftarrow i + 1$

Let us explain how this algorithm works. A byte being processed is divided into two parts: the left one contains bits, which can be decoded unambiguously, and the right part contains the rest of the byte. The result of decoding of the left part is

| $r_{prev}$ | $Text[i]$ | $g$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $l_1$ | $r$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 11000 111 | 1 | | 100 | | | 0 | 6 |
| 6 | 01101 011 | 2 | 1110 | 1 | 1 | | 4 | 2 |
| 2 | 11100110 | 2 | 0111110 | 10 | 1 | | 7 | 0 |
| 0 | 10100 011 | 0 | 10100 | | | | 5 | 5 |
| 5 | 01100110 | 3 | | 1 | 10 | 1 | 0 | 0 |

**Table 3.** Rows of the lookup table

assigned to variables $w_1, w_2, w_3$ and $w_4$ (since the length of the shortest codeword of $D_{2,3,5}$ is 3 bits, one byte cannot contain more than 4 adjacent codewords or their parts). If some byte contains parts of $i$ codewords, the first part might contain only the ending of some codeword, while the last one might contain only the beginning of a codeword. This beginning is stored in the column $w_i$ of the table $TAB$ and it is assigned to the variable $w$. At the beginning of the next iteration, we append a new value $w_1$ to the right of the bit representation of $w$ (line 4). This is quite a simple operation if we know the bit length of $w_1$, which is stored in the column $l_1$.

If there are no bits that can be decoded unambiguously in the last number $w_i$ in the byte, we assign 1 to $w_i$, since the decoded number should always be prepended by the leftmost '1' bit (see the last step of the bitwise decoding algorithm). If the ending of the last codeword $w_i$ coincides with the ending of the byte (it implies that $i = g$), we create the fictitious codeword $w_{i+1}$ which is equal to 1. Such situation is illustrated by rows 3 and 5 of Table 3. For the same reason we assign 1 to $w$ at the beginning of the algorithm. The whole Table 3 shows the rows of $TAB$ array used for decoding the text 11000111 01101011 11100110 10100011 01100110. The unambiguously decoded bits are separated from the rest bits with spaces.

Of course, the decoding should be performed with regard to the right part of the previous byte, which contains bits that cannot be decoded unambiguously. That is, if some byte begins with bits 10, it is decoded differently when the previous byte ends with 01 and when it ends with 011. Indeed, in the first case the codeword delimiter 0110 appears, while in the second case we have the sequence 01110, which cannot appear at the end of a codeword. However, it follows from the bitwise decoding algorithm that each zero bit clears the decoding history. More precisely, if we process the code $D_{2,3,5}$ bit-by-bit from left to right and match the sequence 10 or 00, in both cases we can decode the first of these two bits unambiguously regardless the bits right to them. Therefore, all bits of some byte, starting from the left and up to the bit preceding the rightmost zero, can be decoded unambiguously. Regarding the rightmost zero bit, it can be decoded unambiguously in the following cases.

1. Some codeword ends with this zero.
2. This zero belongs to the sequence $0 \cdots 0$, which is the prefix part of some codeword.
3. The byte contains 3 or more ones after this zero.

In all other cases, the rightmost zero either might belong or not belong to the delimiter 0110. If it belongs to this delimiter, it should be discarded together with the whole delimiter. Otherwise, it should be present in the decoded number. These two cases can be distinguished only at the next iteration.

Also, we note that if a byte ends with the run of 6 ones, the first two bits of this byte do not affect the next ensuing decoding since any of these ones cannot belong to a delimiter.

| Value of $r$ (type) | Number of ones in the end of a byte | Is the rightmost zero bit decoded? |
|---|---|---|
| 0 | 0 | yes |
| 1 | 0 | no |
| 2 | 1 | yes |
| 3 | 1 | no |
| 4 | 2 | yes |
| 5 | 2 | no |
| 6 | 3 | yes |
| 7 | 4 | yes |
| 8 | 5 | yes |
| 9 | $\geq 6$ | yes |

**Table 4.** The types of the byte endings in the fast decoding

Thus, we have 10 types of byte endings, which differently affect the next byte decoding. These types are listed in Table 4 and correspond to 10 possible values of $r$.

Now we can calculate the space complexity of the byte aligned decoding algorithm. It is easy to show that the value $w_1$ cannot be longer than 11 bits and each of the other values $w_i$ fit into one byte. Thus, if for each value we use a whole number of bytes, then one row of the array $TAB$ could be stored in 8 bytes and the whole array requires $8 \times 10 \times 256$ bytes =20K memory. However, on the bit level each row of the array $TAB$ can be packed only into 4 bytes. For such representation, we have built more sophisticated, but several times faster implementation of the table decoding algorithm in assembly language (its details are out of the scope of this presentation). In such case 10K memory needed to store the array $TAB$. For comparison, the fastest one-byte table decoding algorithm for Fib3 code reported in [10] requires 21,4K memory for precomputed arrays.

If the code is applied to represent a sequence of numbers, one need only store the array $TAB$. However, if it is used for compressing many other data types, the dictionary also should be stored. The application of multi-delimiter codes to natural language text compression is discussed in the next section. Also, the experimental estimates of the time complexity of the fast decoding algorithm are presented.

## 6 Data compression by multi-delimiter codes

To determine the data compression efficiency of a code, first of all it is useful to calculate the number of codewords of the length not greater than $n$. The corresponding results are presented in Table 2. As is seen, one-delimiter codes $D_{m-1}$ are asymptotically denser than Fibonacci codes Fib$m$ although they contain less short codewords. As we add other delimiters to a code, the asymptotic density decreases, while the number of short codewords increases. In general, the multi-delimiter codes family is more adaptive as against Fibonacci codes. Choosing appropriate values of $m_1, \ldots, m_t$ allows us to tightly approach the code $D_{m_1,\ldots,m_t}$ to the specific distribution of input symbols and their alphabet size.

For natural language text compression, as noted above, the most efficient seem to be codes with the shortest delimiter 0110. The "champions" are the codes $D_{2,3}$, $D_{2,3,4}$, $D_{2,3,5}$ and $D_{2,4,5}$. However, the code $D_{2,3,4}$ has quite low asymptotic density, which narrows its application to only small alphabets. We investigate more thoroughly the other three codes.

Before presenting the experimental results, let us discuss one specific property of multi-delimiter codes, which relates to use a dictionary in the decoding. In particular, this relates to decoding of natural language texts.

All the encoding/decoding algorithms we discussed fit the following schema. During the encoding, the mapping (*word of text*, *codeword*) is used, where the words of a text are sorted in descending order of frequency, while the codewords are sorted in ascending order of codeword lengths. The decoding process is reverse: one should construct a mapping from the set of codewords to the set of text words. In order to fasten the decoding, a data structure with low access time should be used to store the words of a text. For these purposes, the most efficient data structure is the array with integer indices. It allows us to access the words in a $\mathit{Dictionary}[i]$ style, that is $*(\mathit{Dictionary} + i)$ in C notation. This only requires one addition and three memory readings to obtain a word; however, it also requires constructing a mapping between the set of codewords and the set of integer indices. For Fibonacci codes such mapping can be efficiently performed using some remarkable properties of Fibonacci numbers (the method is developed in [10]); in the SCDC decoding the arithmetic properties of the codes are utilized.

For multi-delimiter codes, we described the encoding and decoding mappings in Section 3. Denote them by $\psi$ and $\psi^{-1}$ respectively. However, they have one essential disadvantage: the codewords $\psi(1), \psi(2), \ldots$ are not sorted in ascending order of their lengths. It follows that the words of a text in the array *Dictionary* could be ordered not in descending order of frequencies $f(w_i)$. This is not a problem since the main ordering principle holds: if $f(w_i) > f(w_j)$, then the length of the codeword $\psi(w_i)$ is equal or less than the length of the codeword $\psi(w_j)$. However, the problem is that the codewords $\psi(1), \ldots, \psi(n)$ do not constitute the set of $n$ shortest codewords. We see three ways to resolve this issue, which represents the trade-off between time, space, and compression efficiency.

1. Encode the text using the codewords $\psi(1), \ldots, \psi(n)$, i.e. not the shortest codewords. As the computational experiment shows, this decreases the compression rate up to 2% but does not increase the time and space complexity of the decoding.
2. Enlarge the size of the dictionary to some value $k > n$ and assign the values to its elements with the indices $\psi^{-1}(c_1), \ldots, \psi^{-1}(c_n)$, where $c_1, \ldots, c_n$ is the set of $n$ shortest codewords, so that $\psi^{-1}(c_i) < k, 1 \leq i \leq n$. The enlarged dictionary is sparse since $k - n$ elements are empty. This requires more memory for the decoding but does not enlarge the size of the dictionary that should be transmitted to a recipient along with the encoded file, because only the set of text words ordered according to their frequencies has to be transmitted. For the code $D_{2,3,5}$, it is enough to increase the *Dictionary* array to four times its original size to achieve the compression less than 0.1% away from the optimal for this code. However, the actual memory consumption increases less than three times, since the enlarged array is sparse. In this case, the decoding time remains optimal.
3. Build the array of some fixed length $t$ for the words with higher frequencies and store the other words in a map (*number*, *word of text*). The non-empty elements of the array have the indices that correspond to shortest codewords. In this case, the access to the map is rather longer, but this data structure is not sparse. If we choose a value of $t$ so that the space complexity is increased by 10%, the time is increased approximately twice. However, the compression rate remains optimal.

| Text | Words | Dictionary size | Entropy bits | SCDC | Fib3 | $D_{2,3,5}$ | $D^L_{2,3,5}$ | $D_{2,3}$ | $D_{2,4,5}$ |
|---|---|---|---|---|---|---|---|---|---|
| Hamlet, Shakespeare | 30 694 | 4 501 | 9.2112 | 10.47<br>13.7% | 10.01<br>8.7% | 9.76<br>6.0% | 10.05<br>9.1% | 9.83<br>6.7% | 9.85<br>6.9% |
| Text in Ukrainian | 90 691 | 14 879 | 10.6455 | 12.04<br>13.1% | 11.4<br>7.1% | 11.26<br>5.8% | 11.61<br>9.1% | 11.35<br>6.6% | 11.33<br>6.4% |
| Robinson Crusoe, D. Defoe | 121 325 | 5 994 | 8.73519 | 10.13<br>16.0% | 9.41<br>7.7% | 9.13<br>4.5% | 9.31<br>6.6% | 9.13<br>4.5% | 9.21<br>5.4% |
| Bible, KJV | 779 079 | 12 452 | 8.6279 | 10.138<br>17.5% | 9.219<br>6.9% | 8.954<br>3.8% | 9.05<br>4.9% | 9.044<br>4.8% | 9.071<br>5.1% |
| Articles from Wikipedia | 19 507 783 | 288179 | 11.0783 | 12.869<br>16.2% | 11.564<br>4.4% | 11.492<br>3.7% | 11.544<br>4.2% | 11.488<br>3.7% | 11.471<br>3.5% |

**Table 5.** Average codeword lengths and excesses over entropy bits for Fib3 and some multi-delimiter codes

In our computational experiments, we follow the second approach by default since usually in natural language texts the size of a vocabulary is never greater than a few megabytes. For modern computers related RAM overheads are quite acceptable. However, some results of the first approach are also presented for comparison.

The results of experiments on compression efficiency of different codes are shown in Table 5. Compression efficiency of SCDC, the Fibonacci code Fib3 and multi-delimiter codes is measured for the texts of different size in two languages: 4 texts in English and 1 in Ukrainian. The largest corpus contains the articles randomly chosen from the English Wikipedia. The punctuation signs in the texts are ignored; lowercase and uppercase symbols are not distinguished. For each text the values of $s$ and $c$ giving the best compression rate of SCDC are determined. Also the "RAM economy" version of the code $D_{2,3,5}$ is tested, which does not enlarge dictionary array (the results are in the column $D^L_{2,3,5}$). The word-level entropy of the texts is calculated. The compression rate is presented as the average codeword length in bits (the first value in a cell) and also as the excess over the entropy bound in percents.

As seen, the multi-delimiter and Fibonacci codes significantly outperform the SCDC codes by compression rate. And codes with 3 delimiters, in its turn, perform $1.2 - 1.8$ times closer to the entropy bound than the Fib3. Among all tested multi-delimiter codes, $D_{2,3,5}$ demonstrates the best compression rate for small and mid-size texts (up to 1M words), but for the large text of 19M words the code $D_{2,4,5}$ becomes slightly better. That is to be expected, since $D_{2,4,5}$ has the better asymptotic density as shown in Table 2. The code $D_{2,3}$ is superior to Fib3, and this shows the usefulness of a second delimiter, but it is inferior to $D_{2,3,5}$ or $D_{2,4,5}$, which demonstrates the benefit of a third delimiter. The rest of multi-delimiter codes presented in Table 2 are inferior at least to one of these three tested codes for all five texts. However, the code $D_{2,3,4}$ shows the best performance on some extremely small texts, which alphabets are less than 2000 words, but, in general, this is too small size for natural language text compression.

The excesses over entropy bounds for the codes Fib3, $D_{2,3}$, $D_{2,4,5}$ and $D_{2,3,5}$ are also shown in Fig. 1.

We also compared the decompression time for three codes: the optimal $(s, c)$-codes, Fib3 and $D_{2,3,5}$. We applied the one-byte variant of the fastest byte-aligned decoding method described in [10] (with two precomputed tables and no multiplications) for Fib3, the bitwise and byte-aligned decoding algorithms for $D_{2,3,5}$ described

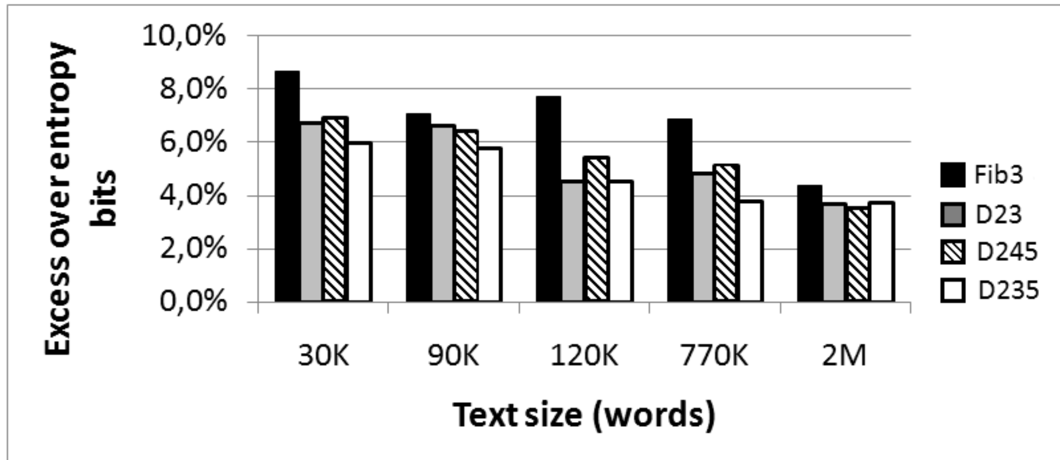



**Figure 1.** Excesses over entropy bits for the codes Fib3, $D_{2,3}$, $D_{2,4,5}$, $D_{2,3,5}$.

| Text | SCDC algorithm | $D_{2,3,5}$, byte-aligned algorithm | $D_{2,3,5}$, bitwise algorithm | Fib3, byte-aligned algorithm |
|---|---|---|---|---|
| Robinson Crusoe, D. Defoe | 15.1 | 17.3<br>14.6% | 48.2<br>219.2% | 31.2<br>106.6% |
| Bible, KJV | 103 | 111<br>7.8% | 270<br>162.1% | 200<br>94.2% |

**Table 6.** Empirical comparison of decoding time, in milliseconds

in sections 3 and 5, respectively. The values are averaged over 1000 runs of decoding on a PC with AMD Athlon II X2 245 2.9 GHz processor, 4 GB RAM, running Windows 7 32-bit operating system. The result of decompression is stored in RAM as the array of words; the time needed to write this array to file is excluded since this is too expensive operation and it dissolves the differences between decompression methods themselves. The results for two texts in English are presented in Table 6. Values are given in milliseconds and the overrun comparing to SCDC in percents is also presented.

As seen, for the code $D_{2,3,5}$ the fast byte-aligned decoding performs significantly faster than that of the Fib3 code. This is expected, since the fast decoding algorithm for $D_{2,3,5}$ performs on average many fewer operations to obtain the index of a word in the dictionary (lines $4-15$), while the reading from the precomputed array (line 3) is roughly of the same time as the similar operation in the fast Fib3 decoding. However, the byte-aligned decoding algorithm for $D_{2,3,5}$ remains slightly inferior to SCDC decoding.

The code Fib3, in comparison with the multi-delimiter codes, also has a drawback, which refers to the characteristic of the instantaneous separation that is important for searching a word in a compressed file without its decompression. As Fib3, so multi-delimiter codes as well as many other codes used for text compression are characterized by the following: for any codeword $w$, if a bit sequence $w$ occurs in a compressed file, we can not guarantee that it truly corresponds to the occurrence of the whole codeword $w$. It could be a suffix of another codeword or it could contain another word as a suffix. In multi-delimiter codes, to check if $w$ is truly a separate codeword, it is enough to consider the fixed number of bits that precede $w$. For example, it is enough to check four bits for the code $D_2$. If they turn out to be 0110,

then $w$ is a codeword, otherwise it is not. However, it is not enough to check any fixed number of bits preceding a codeword in the code Fib3, since the delimiter and the shortest word in this code is 111. Several such codewords can "stick together" if they are adjacent.

This property of multi-delimiter codes allows to perform the pattern search in a compressed file a bit faster. However, we do not discuss the search problem in this presentation in details. General binary search methods (e.g., [9], [7]) can be applied to multi-delimiter codes as well.

## 7 Conclusion

We introduce a new family of variable length prefix multi-delimiter codes. They possess all properties known for the Fibonacci codes such as completeness, universality, simple vocabulary representation, and strong robustness. But also they have some more advantages:

(i) Adaptability. Varying delimiters we can adapt a multi-delimiter code to a given source probability distribution and an alphabet size.
(ii) The better compression rate for natural language text compressing.
(iii) The faster byte aligned decoding method.
(iv) Instantaneous separation of codewords allowing faster compressed search.

The multi-delimiter codes seem to be preferable over $(s, c)$ dense codes in the compression of small and mid-size natural language texts, since they have significantly better compression rate but only slightly greater decompression time. These codes together with the Fibonacci codes can be useful in many practical applications.

## References

1. A. Anisimov and I. Zavadskyi: *Variable length prefix $(\delta, k)$-codes*, in Proc. IEEE International Black Sea Conference on Communications and Networking, BlackSeaCom'15, Constanta, Romania, 2015, pp. 43–47.
2. A. Apostolico and A. S. Fraenkel: *Robust transmission of unbounded strings using fibonacci representations*. IEEE Transactions Information Theory, 33 1987, pp. 238–245.
3. N. Brisaboa, A. Farina, G. Navarro, and M. Esteller: *(s,c)-dense coding: an optimized compression code for natural language text databases*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'03, no. 2857 in Lecture Notes in Computer Science, Manaus, Brazil, 2003, Springer-Verlag, Berlin, pp. 122–136.
4. N. Brisaboa, E. Iglesias, G. Navarro, and J. Parama: *An efficient compression code for text databases*, in 25th European Conference on IR Research, no. 2633 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 468–481.
5. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text*. ACM Transactions on Information Systems, 18(2) 2000, pp. 113–119.
6. P. Elias: *Universal codeword sets and representations of the integers*. IEEE Transactions Information Theory, 21 1975, pp. 194–203.
7. S. Faro and T. Lecroq: *An efficient matching algorithm for encoded dna sequences and binary strings*, in Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching, CPM'09, no. 5577 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2009, pp. 106–115.
8. D. Huffman: *A method for the construction of minimum-redundancy codes*. Proc. IRE, 40 1952, pp. 1098–1101.

9. S. T. Klein and M. Ben-Nissan: *Accelerating boyer moore searches on binary texts*, in Proc. Intern. Conf. on Implementation and Application of Automata, CIAA'07, no. 4783 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 130–143.
10. S. T. Klein and M. Ben-Nissan: *On the usefulness of fibonacci compression codes.* Computer Journal, 53(6) 2010, pp. 701–716.
11. D. Salomon: *Variable-Length Codes for Data Compression*, Springer-Verlag, London, U.K., 2007.

# A Resource-frugal Probabilistic Dictionary and Applications in (Meta)Genomics

Camille Marchet[1], Antoine Limasset[1], Lucie Bittner[2], and Pierre Peterlongo[1]

[1] IRISA Inria Rennes Bretagne Atlantique, GenScale team
[2] Sorbonne Universités, Université Pierre et Marie Curie (UPMC), CNRS, Institut de Biologie Paris-Seine (IBPS), Evolution Paris Seine, F-75005 Paris, France
Corresponding author pierre.peterlongo@inria.fr

**Abstract.** Genomic and metagenomic fields, generating huge sets of short genomic sequences, brought their own share of high performance problems. To extract relevant pieces of information from the huge data sets generated by current sequencing techniques, one must rely on extremely scalable methods and solutions. Indexing billions of objects is a task considered too expensive while being a fundamental need in this field. In this paper we propose a straightforward indexing structure that scales to billions of element and we propose two direct applications in genomics and metagenomics.
We show that our proposal solves problem instances for which no other known solution scales up. We believe that many tools and applications could benefit from either the fundamental data structure we provide or from the applications developed from this structure.

**Keywords:** bioinformatics; sequences comparison; genomics; metagenomics; data structures; minimal perfect hash functions; indexing

## Introduction

A genome or a chromosome can be seen as a word of millions characters long, written in a four letters (or *bases*) alphabet. Modern molecular genome biology relies on sequencing, where the information contained in a genome is chopped into small sequences (around one hundred bases), called reads. By providing millions of short genomic reads along with reasonable sequencing costs, high-throughput sequencing technologies [31] (HTS) introduced an era where data generation is no longer a bottleneck while data analysis is, as this amount of sequences needs to be pulled together in a coherent way. Thanks to HTS improvements, it is possible to sequence hundreds of single genomes and RNA molecules, giving insight to diversity and expression of the genes. HTS even allow to go beyond the study of an individual by sequencing different species/organisms from the same environment at once, going from genomics to metagenomics. This massive sequencing represents a breakthrough: for instance one now can access and directly investigate the majority of the microbial world, which cannot be grown in the lab [17]. However, because of the diversity and complexity of microbial communities, such experiments produce tremendous volumes of data, which represent a challenge for bioinformaticians to deal with. The fragmented nature of genomic information, shredded in reads, craves algorithms to organize and make sense of the data.

A fundamental algorithmic need is to be able to index read sets for a fast information retrieval. In particular, given the amount of data an experiment can produce, methods that scale up to large data sets are needed. In this paper we propose a novel indexation method, called the quasi-dictionary, a probabilistic data structure based

Camille Marchet, Antoine Limasset, Lucie Bittner, Pierre Peterlongo: A Resource-frugal Probabilistic Dictionary and Applications in (Meta)Genomics, pp. 85–98.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

on Minimal Perfect Hash Functions (MPHF). This technique provides a way to associate any kind of data to any piece of sequence from a read set, scaling to very large (billions of elements) data sets, with a low and controlled false positive rate.

A number of studies have focused on optimizing non-probabilistic text indexation, using for instance FM-index [13], or hash tables. However, except the Bloomier filter [9], to the best of our knowledge, no probabilistic dictionary has yet been proposed for which the false positive or wrong answer rates are mastered and limited. The quasi-dictionary mimics the Bloomier filter solution as it enables to associate a value to each element from a set, and to obtain the value of an element with a mastered false positive probability if the element was not indexed. Existing published results in [9] indicate that the Bloomier filter and the quasi-dictionary have similar execution times, while our results tend to show that the quasi-dictionary uses approximately ten times less memory. Moreover, there are no available/free Bloomier filter yet implemented.

We propose two applications that use quasi-dictionary for indexing $k$-mers, enabling to scale up large (meta)genomic instances. As suggested by their names (*short read connector counter* and *short read connector linker*, as presented below), these applications have the ability to connect any read to either its estimated abundance in any read set or to a list of reads in any read set. A key point of these applications is to estimate read similarity using $k$-mers diversity only. This alignment-free approach is widely used and is a good estimation of similarity measure [12].

Our first application, called *short read connector counter* (SRC_counter), consists in estimating the number of occurrences of a read (i.e. its abundance) in a read set. This is a central point in high-throughput sequencing studies. Abundance is first very commonly used as indicator value for reads trimming: i.e. reads with relatively low abundance value are considered as amplification errors and/or sequencing errors, and these rare reads are generally removed before thorough analyses [21,30]. The abundance of reads is then interpreted as a quantitative or semi-quantitative metric: i.e. reads abundance is used as a measure of genic or taxon abundance, themselves very commonly used for comparisons of community similarity [2,19].

The second proposed application in this work, called *short read connector linker* (SRC_linker), consists in providing a list of similar reads between read sets. We define the read set similarity problem as follows. Given a read set *bank* and a read set *query*, provide a similarity measure between each pair of reads $b_i \times q_j$, with $b_i$ a read from the bank set and $q_j$ a read from the query set. Note that the bank and the query sets may refer to the same data set. Computing read similarity intra-read set or inter-read sets can be performed by a general purpose tool, such as those computing similarities using dynamic programming, and using heuristic tools such as BLAST [1]. However, comparing all versus all reads requires a quadratic number of read comparisons, leading to prohibitive computation time, as this is shown in our proposed results. There exist tools dedicated to the computation of distances between read sets [7,25,26], but none of them can provide similarity between each pair of reads $b_i \times q_j$. Otherwise, some tools such as starcode [35] are optimized for pairwise sequence comparisons with mainly the aim of clustering DNA barcodes. As shown in results, such tools also suffer from quadratic computation time complexity and thus do not scale up data sets composed of numerous reads.

*Availability and license* Our proposed tools SRC_counter and SRC_linker were developed using the GATB library [11]. They may be used as stand alone tools or as li-

braries. They are licensed under the GNU Affero General Public License version 3 and can be downloaded from `http://github.com/GATB/short_read_connector`. Also licensed under the GNU Affero General Public License version 3, the quasi-dictionary can be downloaded from `http://github.com/pierrepeterlongo/quasi_dictionary`.

# 1 Methods

We first recall basic notations: a $k$-mer is a word of length $k$ on an alphabet $\Sigma$. Given a read set[1] $\mathcal{R}$, a $k$-mer is said *solid* in $\mathcal{R}$ with respect to a threshold $t$ if its number of occurrences in $\mathcal{R}$ is bigger or equal to $t$. Let $|w|$ denote the length of a word $w \in \Sigma^*$ and $|\mathcal{R}|$ denote the number of elements contained in $\mathcal{R}$.

## 1.1 Quasi-dictionary index

In the following, we present our indexing solution. Based on this solution, two applications are proposed sections 1.3 and 1.4.

The index we propose associates each solid $k$-mer from a read set $\mathcal{R}$ to a unique value in $[0, N-1]$, with $N$ being the total number of solid $k$-mers in $\mathcal{R}$. Ideally, when querying a non indexed $k$-mer (i.e. a non solid $k$-mer or a $k$-mer absent from $\mathcal{R}$) the index returns no value. In our proposal, a non indexed $k$-mer may be associated to a value in $[0, N-1]$ with a probability $p > 0$. This is why we refer to our index as the ***quasi***-*dictionary*, since it is a probabilistic index. However, note that querying any indexed $k$-mer provides a unique and deterministic answer.

We define the quasi-dictionary as follows :

Given a static set composed of $N$ distinct elements, a quasi-dictionary is composed of two structures: a minimal perfect hash function MPHF (see for instance [5]) and a table of fingerprints $FingerPrints$.

The MPHF for $S$ is a function such that:

$$\forall e \in S,\ \text{MPHF}(e) = i \in [0, N-1]$$
$$\forall e_1, e_2 \in S,\ (\text{MPHF}(e_1) = \text{MPHF}(e_2)) \Leftrightarrow (e_1 = e_2)$$

The fingerprint table for $S$ is composed of $N$ elements. It assigns to each element form $S$ an integer value in $[0, 2^f - 1]$, with $f$ the size of the fingerprint in bits. This table is used to verify the membership of an element $e$ to the indexed set of elements using the MPHF. When $S$ represents a set of $k$-mers, the fingerprint table uses $f \leq 2k$ since a $k$-mer can be coded as a word of $2k$ bits. The false positive rate is then

$$\frac{2^{(2k-f)} - 1}{2^{2k}} \approx \frac{1}{2^f}.$$

## 1.2 Indexing solid $k$-mers using a quasi-dictionary

Algorithm 1 presents the construction of the quasi-dictionary. The set of solid $k$-mers (algorithm 1, line 1) is obtained using the DSK [28] method. The MPHF (algorithm 1, line 2) is computed using the MPHF library[2].

[1] Note that formally $\mathcal{R}$ should be denoted as a *"collection"* instead of a *"set"* as a read may appear twice or more in $\mathcal{R}$. However, to make the reading easier, we use in this manuscript the term "*set*" usually employed for describing HTS outputs.

[2] `https://github.com/rizkg/BooPHF`, commit number 852*cda*2

**Algorithm 1:** *create_quasidictionary*

**Data**: Read set $\mathcal{R}$, $k \in \mathbb{N}, t \in \mathbb{N}, f \in \mathbb{N}$
**Result**: A quasi-dictionary QD
1 $k$-mer set $\mathcal{K}$ = get_solid_kmers($\mathcal{R}, k, t$) ;
2 $QD.MPHF = create_MPHF(\mathcal{K})$ ;
3 **foreach** *$k$-mer $w$ in $\mathcal{K}$* **do**
4     $index = QD.MPHF(w)$;
5     $QD.FingerPrints[index] = create_fingerprint(w, f)$;
6 **return** QD;

The fingerprint of a word $w$ (algorithm 1, line 5) is obtained thanks to a hashing function

$$create_fingerprint : \Sigma^{|w|} \rightarrow [0, 2^f - 1],$$

with $f \leq 2k$. In practice we chose to use a xor-shift [27] for its efficiency in terms of throughput and hash distribution.

**Algorithm 2:** *query_quasidictionary*

**Data**: Quasi-dictionary $QD$, word $w$
**Result**: A unique value in $[0, N-1]$ (with $N$ the number of indexed elements) or -1 if $w$ detected as non indexed
1 $index = QD.MPHF(w)$;
2 **if** $index \geq 0$ ***and*** $QD.FingerPrints[index] = create_fingerprint(w)$ **then**
3     **return** $index$;
4 **return** $-1$;

The querying of a quasi-dictionary with a word $w$ is straightforward, as presented in Algorithm 2. The *index* of $w$ is retrieved using the MPHF. Then the fingerprint stored for this *index* is compared to the fingerprint of $w$. If they differ, then $w$ is not indexed and the $-1$ value is returned. If they are equal, the value $index \geq 0$ is returned. Note that two distinct words have the same fingerprint with a probability $\approx \frac{1}{2^f}$. It follows that there is a probability $\approx \frac{1}{2^f}$ that the quasi-dictionary returns a false positive value despite the fingerprint checking, *i.e.* an $index \neq -1$ for a non indexed word. On the other hand, the *index* returned for an indexed word is the correct one. In practice we use $f = 12$ that limits the false positive rate to $\approx 0.02\%$. Note that our implementation authorizes any value $f \leq 64$.

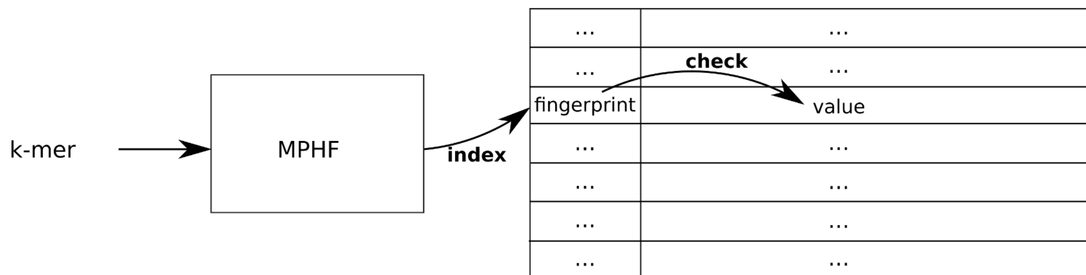



**Figure 1.** Quasi dictionary structure composed of a MPHF and a table of fingerprints. The fingerprints are obtained by hashing the corresponding $k$-mers. Thanks to the MPHF a unique index is associated to each $k$-mer where the fingerprint is stored. The fingerprint is associated to the value we want to link to the $k$-mer. When a $k$-mer is queried the fingerprint obtained is checked against the fingerprint stored to limit false positives.

*DNA strands* DNA molecules are composed of two strands, each one being the reverse complement[3] of the other. As current sequencers usually do not provide the strand of each sequenced read, each indexed or queried $k$-mer should be considered in the forward or in the reverse complement strand. This is why, in the proposed implementations, we index and query only the canonical representation of each $k$-mer, which is the lexicographically smaller word between a $k$-mer and its reverse complement.

*Time and memory complexities* Our MPHF implementation has the following characteristics. The structure can be constructed in $O(N)$ time and uses $\approx 4$ bits by elements. We could use parameters limiting memory fingerprint to less than 3 bits per element, but we chose parameters to greatly speed up MPHF construction and query. The fingerprint table is constructed in $O(N)$ time, as the $create_fingerprint$ function runs in $O(1)$. This table uses exactly $N \times f$ bits. Thus the overall quasi-dictionary size, with $f = 12$ is $\approx 16$ bits per element. Note that this value does not take into account the size of the values associated to each indexed element.

The querying of an element is performed in constant time and does not increase memory complexity.

### 1.3 Approximating the number of occurrences of a read in a read set

**Algorithm 3:** SRC_counter: Quasi-dictionary used for counting $k$-mers

**Data**: Read set $\mathcal{B}$, read set $\mathcal{Q}$, $k \in \mathbb{N}, t \in \mathbb{N}, f \in \mathbb{N}$
**Result**: For each read from $\mathcal{Q}$, its $k$-mer similarity with set $\mathcal{B}$

1. quasi-dictionary $QD = create_quasidictionary(\mathcal{B}, k, t, f)$ ;
2. create a table *count* composed of $N$ integers[a];
3. **foreach** *Solid $k$-mer $w$ from $\mathcal{B}$* **do**
4.     $count[query_quasidictionary(w)]$ = number of occurrences of $w$ in $\mathcal{B}$;
5. **foreach** *read $q$ in $\mathcal{Q}$* **do**
6.     create an empty vector *count_q*;
7.     **foreach** *$k$-mer $w$ in $q$* **do**
8.         **if** $query_quasidictionary(w) \geq 0$ **then**
9.             add $count[query_quasidictionary(w)]$ to *count_q* ;
10.     Output the $q$ identifier, and (mean, median, min and max values of *count_q*);

[a] with $N$ the number of solid $k$-mers from $\mathcal{B}$

As presented in Algorithm 3, we propose a first straightforward application using the quasi-dictionary. This application is called SRC_counter for *short read connector counter*. It approximates the number of occurrences of reads in a read set.

Two potentially equal read sets $\mathcal{B}$ and $\mathcal{Q}$ are considered. The indexation phase works as follows. Each solid $k$-mer of $\mathcal{B}$ is indexed using a quasi-dictionary. A third-party table named *count* stores the counts of indexed $k$-mers. Elements of this table are accessed via the quasi-dictionary *index* value of indexed items (Algorithm 3 lines 4 and 9). The number of occurrences of each solid $k$-mer from $\mathcal{B}$ (line 4) is obtained from DSK output, used during the quasi-dictionary creation (line 1). Then starts the query phase. Once the *count* table is created, for each read $q$ from set $\mathcal{Q}$, the count of all its $k$-mers indexed in the quasi-dictionary are recovered and stored in a vector

[3] The reverse complement of a DNA sequence is the palindrome of the sequence, in which $A$ and $T$ are swapped and $C$ and $G$ are swapped. For instance the reverse complement of $ACCG$ is $CGGT$.

(lines 8 and 9). Finally, collected counts from $k$-mers from $q$ are used to output an estimation of its abundance in read set $\mathcal{B}$. The abundance is approximated using the mean number of occurrences of $k$-mers from $q$, to supplement we output the median, the min and the max number of occurrences of $k$-mers from $q$. In rare occasions, false positives of the method can lead to an over-estimation of the count.

This algorithm is extremely simple. In addition to the quasi-dictionary creation time and memory complexities, it has a constant memory overhead (8 bits by element in our implementation) and it has an additional $O(\sum_{Q\in\mathcal{Q}}|Q|)$ time complexity.

### 1.4 Identifying similar reads between read sets or inside a read set

**Algorithm 4:** SRC_linker: Quasi-dictionary used for identifying read similarities

**Data**: Read set $\mathcal{B}$, read set $\mathcal{Q}$, $k\in\mathbb{N}, t\in\mathbb{N}, f\in\mathbb{N}$
**Result**: For each read from $\mathcal{Q}$, its $k$-mer similarity with each read from set $\mathcal{B}$

```
1  quasi-dictionary QD = create_quasidictionary(B, k, t, f) ;
2  create a table ids composed of N vectors of integers^a ;
3  foreach read b in B do
4  |  foreach k-mer w in b do
5  |  |  index = query_quasidictionary(w);
6  |  |  if index ≥ 0 then
7  |  |  |  add identifier of b to vector ids[index] ;
8  foreach read q in Q do
9  |  create a hash table associating targets (target_read_id) to couple(next_free_position,
   |  count);
10 |  foreach i in [0, |q| − k] do
11 |  |  w = k-mer occurring position i in q;
12 |  |  index = query_quasidictionary(w);
13 |  |  if index ≥ 0 then
14 |  |  |  foreach tg_id in vector ids[index] do
15 |  |  |  |  if targets[tg_id] is empty then
16 |  |  |  |  |  targets[tg_id].next_free_position = 0
17 |  |  |  |  targets[tg_id].count+ = max (k, i + k − targets[tg_id].next_free_position)
   |  |  |  |  targets[tg_id].next_free_position = i + k
18 |  Output the id of q and each^b tg_id associate to its count from targets table;
```

[a] with $N$ the number of solid $k$-mers from $\mathcal{B}$
[b] In practice only *tg_id* whose *count* value is higher or equal to a user defined threshold are output

Our second proposal, called SRC_linker for *short read connector linker*, compares reads from two potentially identical read sets $\mathcal{B}$ and $\mathcal{Q}$. For each read $q$ from $\mathcal{Q}$, a similarity measure with reads from $\mathcal{B}$ is provided.

The similarity measure we propose for a couple of reads $q\times b$ is the number of positions on $q$ that is covered by at least a $k$-mer that also occur on $b$. Note that this measure is not symmetrical as one does not verify that the $k$-mers do not overlap on $b$.

The indexation phase of SRC_linker works as follows. A quasi-dictionary is created and a third-party table *ids* of size $N$ is created. Each element of this table stores for a solid $k$-mer $w$ from $\mathcal{B}$ a vector containing the identifiers of reads from $\mathcal{B}$ in which $w$ occurs. See lines 2 to 7 of Algorithm 4.

The query phase (lines 8 to the end of Algorithm 4) is straightforward. In practice, for each targeted read $b_j$ in $\mathcal{B}$ we remind the ending position of the last shared $k$-mer

on $q$ with $b_j$ denoted by $next_free_position$ in the Algorithm 4. Given a new shared $k$-mer, the number of positions that was not already covered by another shared $k$-mer is added to the similarity measure (line 17 of Algorithm 4).

Once all $k$-mers of a read $q$ are treated, the identifier of $q$ is output and for each read $b_j$ from $\mathcal{B}$ its identifier is output together with the number of shared $k$-mers with $q$. In practice, in order to avoid quadratic output size and to focus on similar reads, only reads sharing a number of $k$-mers higher or equal to a user defined threshold are output.
In addition to the quasi-dictionary data structure creation, considering a fixed read size, Algorithm 4 has $O(N \times \overline{m})$ memory complexity and a $O(N + \sum_{Q \in \mathcal{Q}} |Q| \times \overline{m})$ time complexity, with $\overline{m}$ the average number of distinct reads from $\mathcal{B}$ in which a $k$-mer from $\mathcal{Q}$ occurs. In the worst case $\overline{m} = N$, for instance with $\mathcal{B} = \mathcal{Q} = \left\{A^{|\text{read}|}\right\}^N$. In practice, in our tests as well as for real sets composed of hundred of million reads, $\overline{m}$ is limited to $\approx 2.22$.

**Storing read identifiers on disk** Storing the read identifiers as proposed in Algorithm 4 presents important drawbacks as it requires a large amount of RAM. In order to get rid of this limitation we propose a disk version of this algorithm, in which the table *ids* is stored on disk. As shown in Algorithm 5 (see Appendix), the algorithmic solution is not straightforward as one needs to know for each indexed $k$-mer $w$ its number of occurrences in the read set $\mathcal{B}$ plus the number of occurrences of $k$-mers $\neq w$ from $\mathcal{B}$ (false positives) that have the same quasi-dictionary $index$.

This disk based solution enables to scale up very large instances with frugal RAM needs, at the price of a longer computation time, as show in results.

## 2 Results

This section presents results about the fundamental quasi-dictionary data structure and about potential applications derived from its usage. To this end, we use a metagenomic *Tara* Oceans [18] read set ERR59928[4] composed of 189,207,003 reads of average size 97 nucleotides. From this read set, we created six sub-sets by selecting first 10K, 100K, 1M, 10M, 50M and 100M reads (with K meaning thousand and M meaning million).

Tests were performed on a linux 20-CPU nodes running at 2.60 GHz with an overall of 252 GBytes memory.

### 2.1 SRC_counter tests and performances

We first provide SRC_counter results enabling to evaluate the gain of our proposed data structure when compared to a classical hash table. Secondly we provide results that enable to estimate the impact of false positives on results.

**SRC_counter performances compared to standard hash table index** We tested the SRC_counter performances by indexing iteratively the six read subsets plus the full ERR59928 set, each time querying reads from set 10M. We compared our solution performances with a classical indexation scheme done using the C++

[4] `http://www.ebi.ac.uk/ena/data/view/ERR599280`

| Indexed Dataset (nb solid $k$-mers) | $k$-mer count time (s) | Construc. time (s) | | Memory (GB) | | | Query Time(s) | |
|---|---|---|---|---|---|---|---|---|
| | | QD | Hash | QD | QD62 | Hash | QD | Hash |
| 1M (64,321,167) | 2 | 1 | 106 | 0.25 | 2.45 | 2.46 | 10 | 13 |
| 10M (621,663,812) | 15 | 7 | 1091 | 1.80 | 5.45 | 23.58 | 11 | 17 |
| 50M (2,812,637,134) | 72 | 77 | 5027 | 8.00 | 16.37 | 106.25 | 11 | 19 |
| 100M (5,191,190,377) | 196 | 220 | 9335 | 14.71 | 44.93 | 202.91 | 13 | 19 |
| Full (8,783,654,120) | 486 | 532 | | 24.83 | 75.96 | | 15 | |

**Table 1.** Wallclock time and memory used by the SRC_counter algorithm for creating and for querying the quasi-dictionary using the default fingerprint size $f = 8$ (denoted by "QD") and the C++ *unordered_map*, denoted by "Hash". Column "$k$-mer count time" indicates the time DSK spent counting $k$-mers. Tests were performed using $k = 31$ and $t = 1$ (all $k$-mers are solid). The query read set was always the 10M set. We additionally provide memory results using the quasi-dictionary with a fingerprint size $f = 62$ (denoted by "QD62"). Construction and query time for QD62 are not shown as they are almost identical to the QD ones. On the full data set, using a classical hash table, the memory exceeded the maximal authorized machine limits (252 GB).

*unordered_map* hash table. Results are presented in Table 1. These results show that the quasi-dictionary is much faster to compute than this hash table solution, in particular because of parallelisation. Moreover, the quasi-dictionary memory footprint is $\approx 13$ times smaller on large enough instances (10 million indexed reads or more). These results show that the hash table is not a viable solution scaling up current read sets composed of several billions $k$-mers. Results also highlight the fact that the query is fast and only slightly depends on the number of indexed elements.

Importantly, using a fingerprint large enough (here $f = 62$ for $k$-mers of length $k = 31$), we can force the quasi-dictionary to avoid false positives. As expected, the quasi-dictionary data structure size increases with the size of $f$ but interestingly, on this example and as shown in Table 1, the size of the quasi-dictionary with $f = 62$ remains in average 4 times smaller than the size of the hash-table on large problem instances. Keeping in mind that the quasi-dictionary is faster to construct and to query, the usage of this data structure avoiding false positives presents only advantages compared to the hash table usage for indexing a static set. However, one should recall that this is true because we are using an alphabet of size four, so any 31-mer on the alphabet $\{A, C, G, T\}$ can be assigned to a unique value in $[0, 2^{62} - 1]$ and *vice versa*. With larger alphabets such as the amino-acids or the Latin ones, the usage of a hash table is recommended if false positives are not tolerated.

**Approximating false positives impact** We propose an experiment to assess the impact on result quality when using a probabilistic data structure instead of a deterministic one for estimating read abundances.

We used the read set 100M both for the indexation and for the querying, thus providing an estimation of the abundance of each read in its own read set. We made the indexation using $k = 31, c = 2$ and $f = 8$. Note that, with $c = 2$ only $k$-mers seen twice or more in the set are solid and thus are indexed. In this example only 756,804,245 $k$-mers are solid among the 5,191,190,377 distinct $k$-mers present in the read set. This means that during the query, 85.4% of queried $k$-mers are not indexed. This enables to measure the impact of the quasi-dictionary false positives. We applied the count algorithm as described in Algorithm 3, and the tuned version using a hash table instead of a quasi-dictionary. We analyzed the count output composed of the average number of occurrences of $k$-mers of each read in the 100M read set.

Because of the quasi-dictionary false positives, results obtained using this structure are an over-estimation of the real result. Thus, we computed for each read the observed difference in the counts between results obtained using the quasi-dictionary implementation and the hash table implementation. The max over-approximation is 26.9, and the mean observed over-approximation is $7.27 \times 10^{-3}$ with a $3.59 \times 10^{-3}$ standard deviation. Thus, as the average estimated abundance of each read which is $\approx 2.22$, the average count over-estimation represents $\approx 0.033\%$ of this value. Such divergences are negligible.

### 2.2 Identifying similar reads

We set a benchmark of our method with comparisons to state of the art tools that can be used in current pipelines for the read similarity identification presented in this paper. We compared our tool with the classical method BLAST [1] (version 2.3.0), with default parameters. BLAST is able to index big data sets, and consumes a reasonable quantity of memory, but the throughput of the tool is relatively low and only small data sets were treated within the timeout (10h, wallclock time). We also included two broadly used mappers in the comparison. We used Bowtie2 [22] (version 2.2.7), and BWA [23] (version 0.7.10) in any alignment mode (`-a` mode in Bowtie2, `-N` for BWA) in order to output all alignment found instead of the best ones only. Both tools are not well suited to index large set of short sequences nor to find all alignments and therefore use considerably more resources than their standard usage.

We also compared SRC_linker to starcode (1.0), that clusters DNA sequences by finding all sequences pairs below a Levenshtein distance metric. One should notice that benchmark comparisons with tools as starcode is unfair as such tool provides much more precise distance information between pair of reads than SRC_linker and performs additional task as clustering. However, our benchmark highlights the fact that such approaches suffer from intractable number of read comparisons, as demonstrated by presented results.

| | Time(s) | | | | | Memory(GB) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Indexed Dataset | Blast | Bowtie2 | BWA | starcode | SRC_linker | Blast | Bowtie2 | BWA | starcode | SRC_linker |
| 10K | 4 | 3 | 6 | 2 | 1 | 0.7 | 0.29 | 0.04 | 11.36 | 1.01 |
| 100K | 52 | 51 | 106 | 29 | 5 | 18.5 | 0.77 | 0.49 | 12.06 | 1.07 |
| 1M | **795** | **10,644** | **3,155** | **1,103** | **45** | **24.5** | **5.54** | **3.4** | **18.18** | **1.28** |
| 10M | | | 62,912 | 131,139 | 587 | | | 5.9 | 73.5 | 3.61 |
| 100M | | | | | 14,748 | | | | | 44.37 |
| Full | | | | | 40,828 | | | | | 110.84 |

**Table 2.** CPU time and memory consumption for indexing and querying a data set versus itself. We set a timeout of 10h. BLAST crashed for 10M data set, Bowtie2 reached the timeout we set with more than 200h (CPU) for 10M reads. BWA performs best among the mappers, reaching the timeout for 100M reads (more than 200h (CPU) on this data set). On the 100M data set, starcode reached the timeout. Only SRC_linker finished on all data sets. On the full data set, it lasted an order of magnitude comparable to what BWA performed on only 10M.

We focused on a practical use case for which our method could be used, namely retrieving similarities in a read set against itself. We used default SRC_linker parameters ($k = 31$, $f = 12$, $c = 2$). Because of the limitations of the methods we used for the benchmark, reported in Table 2, we could compare against all methods only up to 1M reads. BWA performed better than the two other tools in terms of memory,

being able to scale up to 10M reads, while Bowtie2 and BLAST could only reach 1M reads comparison. On this modest size of read set, we show that we are already ahead both in terms of memory and time. However the gap between our approach and others increases with the amount of data to process. Dealing with the full Tara data set reveals the specificity of our approach (Table 2) that requires low resources in comparison to others and is able to deal with bigger data sets.

| | Indexation Time (s) | | Query Time (s) | | Memory |
|---|---|---|---|---|---|
| | One thread | 20 threads | One thread | 20 threads | (GB) |
| RAM Full | 18,067 | 1,768 | 17,558 | 992 | 110 |
| Disk Full | 106,766 | 28,471 | 24,873 | 1,736 | 19 |

**Table 3.** Multithreading and disk performances. The full read set was used to detail the performances of the RAM and Disk algorithm on a large data set. We used default parameters $k = 31$, $f = 12$, $c = 2$. Times are wallclock times.

Finally, we highlight that we provide a parallelised tool (10× speedup for the index and 17× speedup for query for RAM algorithm as shown in Table 3) on the contrary to classical methods that are partly-parallelised as only the alignment step is well suited for parallelisation. The disk version does not fully benefit from multiple cores since the bottleneck is disk access. The main interest of this technique is a highly reduced memory usage at the price of an order of magnitude lower throughput, as presented Table 3.

## 3 Discussions and conclusion

In this contribution, we propose a new indexation scheme based on a Minimal Perfect Hash Function (MPHF) together with a fingerprint value associated to each indexed element. Our proposal is a probabilistic data structure that has similar features than Bloomier filters, with smaller memory fingerprint. This solution is resource-frugal (we have shown experiments on sets containing more than eight billion elements indexed in $\approx 3$ hours and using less than 25 GB RAM) and opens the way to new (meta)genomic applications. As proofs of concept, we proposed two novel applications: SRC_counter and SRC_linker. The first estimates the abundance of a sequence in a read set. The second detects similarities between pair of reads inter or intra-read sets. These applications are a start for broader uses and purposes.

Two main limitations of our proposal due to the nature of the data structure can be pointed out. Firstly, compared to standard hash tables, our indexing data structure presents an important drawback: the exact set of keys to index has to be defined during the data structure creation and it has to be static. This may be a limitation for non fixed set of keys. Moreover, our data structure can generate false positives during query. Even with the proposed false positive ratio limited to $\approx 10^{-2}\%$ with defaults parameters, this may be incompatible with some applications. However we can force our tools to avoid false positives by using as a fingerprint the key itself. Interestingly, this still provides better time and memory performances than using a standard hash table in the DNA $k$-mer indexing context, with $k = 31$, which is a very common value for read comparisons [7]. Secondly, one should notice that our indexation proposal saves space regarding the association between an element and a specific array offset (if the element was indexed). However, our proposal does not

limit the space needed for storing the value associated to each indexed element. Thus, with respect to classical hash tables, the memory gain is limited in problem instances in which large values are associated to each key. Indeed, in this case, the memory footprint is mainly due to the value over the indexing scheme. In order to benefit from our proposal even in such cases we proposed an application example in which the values are stored on disk. However, our approach is namely designed for problems where a huge number of elements to index are at stake, along with a small quantity of information to match with.

We could improve our technique to recognize key from the original set, using a technique from the hashing field [20] or from the set representation field [6]. In such framework, a set can be represented with less memory than the sum of the memory required by the keys. We could thus hope being able to represent a non-probabilistic dictionary without storing keys. Otherwise, we could use the hashing information to achieve a smaller false positive rate with the same or a reduced memory usage. The main challenge will be to keep fast query operation for such complex data structure.

The results we provided show that alignment-based approaches do not scale when it comes to find similar reads in data sets composed of millions of sequences. The fact that HTS data count rarely less than millions reads justifies our approach based on $k$-mer similarity. Moreover our approach is more straightforward and requires less parameters and heuristics than mapping approaches, that can sometimes turn them into blackboxes. However, such an approach remains less precise than mapping, since the $k$-mer order is not taken into account and is less sensitive because of the fixed size of $k$. An important future work will be to evaluate the differences between matches of our pseudo-alignment and matches of well-known and widely used tool as BLAST.

Our tools property of enabling the test of a read set against itself opens the door to applications such as read clustering. Latest sequencing technologies, called Third Generation Sequencers (TGS), provide longer reads [32,33] (more than a thousand bases instead of a few hundreds for HTS). With previous HTS short reads, *de novo* approaches to reconstruct DNA or RNA molecules were using assembly [16,29], based on de Bruijn graphs. For RNA, these TGS long reads mean a change of paradigm as assembly is no more necessary, as one read is long enough to represent one full-length molecule. The important matter becomes to segregate families of RNA molecules within a read set, a purpose our approach could be designed for.

Furthermore, the methods we provide have straightforward applications examples in biology, such as the building of sequences similarity networks (SSN) [3] using SRC_linker. SSN are extremely useful for biologists because, in addition to allowing a user-friendly visualization of the genetic diversity from huge HTS data sets, they can be studied analytically and statistically using graph topology metrics. SSN have recently been adapted to address an increasing number of biological questions investigating both patterns and processes: e.g. population structuring [15,14]; genomes heterogeneity [8]; microbial complexity and evolution [10]; microbiome adaptation [4,34] or to explore the microbial dark matter [24]. In metagenomic microbial studies, SSN offer an alternative to classical and potentially biased methods, and thus facilitate large-scale analyses and hypotheses generation, while notably including unknown/dark matter sequences in the global analysis [15,24]. Currently SSN are built upon general purposes tools such as BLAST. They thus hardly scale up large data sets. A future work will consist in checking the feasibility of applying SRC_linker for constructing SSN and, in case of success, to use it on large SSN problem instances on which other classical tools cannot be applied.

## Acknowledgments

This work was funded by French ANR-12-BS02-0008 Colib'read project. We thank the GenOuest BioInformatics Platform that provided the computing resources necessary for benchmarking. We warmly thank Guillaume Rizk and Rayan Chikhi for their work on the MPHF and for their feedback on the preliminary version of this manuscript.

## References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: *Basic local alignment search tool*. Journal of molecular biology, 215(3) 1990, pp. 403–410.
2. A. S. Amend, K. A. Seifert, and T. D. Bruns: *Quantifying microbial communities with 454 pyrosequencing: does read abundance count?* Mol. Ecol., 19(24) Dec 2010, pp. 5555–5565.
3. H. J. Atkinson, J. H. Morris, T. E. Ferrin, and P. C. Babbitt: *Using sequence similarity networks for visualization of relationships across diverse protein superfamilies*. PLoS ONE, 4(2) 2009, p. e4345.
4. E. Bapteste, C. Bicep, and P. Lopez: *Evolution of genetic diversity using networks: the human gut microbiome as a case study*. Clin. Microbiol. Infect., 18 Suppl 4 Jul 2012, pp. 40–43.
5. D. Belazzougui, P. Boldi, G. Ottaviano, R. Venturini, and S. Vigna: *Cache-oblivious peeling of random hypergraphs*, in Data Compression Conference Proceedings, 2014, pp. 352–361.
6. D. Belazzougui and R. Venturini: *Compressed static functions with applications*, in Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '13, Philadelphia, PA, USA, 2013, Society for Industrial and Applied Mathematics, pp. 229–240.
7. G. Benoit, P. Peterlongo, M. Mariadassou, E. Drezen, S. Schbath, D. Lavenier, and C. Lemaitre: *Multiple Comparative Metagenomics using Multiset k-mer Counting*. apr 2016, pp. 1–17.
8. E. Boon, S. Halary, E. Bapteste, and M. Hijri: *Studying genome heterogeneity within the arbuscular mycorrhizal fungal cytoplasm*. Genome Biol Evol, 7(2) Feb 2015, pp. 505–521.
9. D. Charles and K. Chellapilla: *Bloomier filters: A second look*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 5193 LNCS, 2008, pp. 259–270.
10. E. Corel, P. Lopez, R. Meheust, and E. Bapteste: *Network-Thinking: Graphs to Analyze Microbial Complexity and Evolution*. Trends Microbiol., 24(3) Mar 2016, pp. 224–237.
11. E. Drezen, G. Rizk, R. Chikhi, C. Deltel, C. Lemaitre, P. Peterlongo, and D. Lavenier: *GATB: Genome Assembly & Analysis Tool Box*. Bioinformatics (Oxford, England), jul 2014, pp. 1–3.
12. V. B. Dubinkina, D. S. Ischenko, V. I. Ulyantsev, A. V. Tyakht, and D. G. Alexeev: *Assessment of k-mer spectrum applicability for metagenomic dissimilarity analysis*. BMC Bioinformatics, 17(1) dec 2016, p. 38.
13. P. Ferragina and G. Manzini: *Indexing Compressed Text*. Journal of the ACM, 52(4) 2000, pp. 552–581.
14. M. Fondi, A. Karkman, M. Tamminen, E. Bosi, M. Virta, R. Fani, E. Alm, and J. McInerney: *Every gene is everywhere but the environment selects: Global geo-localization of gene sharing in environmental samples through network analysis*. Genome Biology and Evolution, 2016.
15. D. Forster, L. Bittner, S. Karkar, M. Dunthorn, S. Romac, S. Audic, P. Lopez, T. Stoeck, and E. Bapteste: *Testing ecological theories with sequence similarity networks: marine ciliates exhibit similar geographic dispersal patterns as multicellular organisms*. BMC Biol., 13 2015, p. 16.
16. M. G. Grabherr, B. J. Haas, M. Yassour, J. Z. Levin, D. A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng, et al.: *Full-length transcriptome assembly from RNA-Seq data without a reference genome*. Nature biotechnology, 29(7) 2011, pp. 644–652.
17. L. A. Hug, B. J. Baker, K. Anantharaman, C. T. Brown, A. J. Probst, C. J. Castelle, C. N. Butterfield, A. W. Hernsdorf, Y. Amano, K. Ise, Y. Suzuki, N. Dudek, D. A. Relman, K. M. Finstad, R. Amundson, B. C. Thomas, and J. F. Banfield: *A new view of the tree of life*. Nature Microbiology, 1 Apr 2016, pp. 16048 EP –, Letter.

18. E. Karsenti, S. G. Acinas, P. Bork, C. Bowler, C. de Vargas, J. Raes, M. Sullivan, D. Arendt, F. Benzoni, J. M. Claverie, M. Follows, G. Gorsky, P. Hingamp, D. Iudicone, O. Jaillon, S. Kandels-Lewis, U. Krzic, F. Not, H. Ogata, S. Pesant, E. G. Reynaud, C. Sardet, M. E. Sieracki, S. Speich, D. Velayoudon, J. Weissenbach, and P. Wincker: *A holistic approach to marine Eco-systems biology.* PLoS Biology, 9 2011.
19. S. W. Kembel, M. Wu, J. A. Eisen, and J. L. Green: *Incorporating 16s gene copy number information improves estimates of microbial diversity and abundance.* PLoS Comput Biol, 8(10) 10 2012, pp. 1–11.
20. A. Kirsch and M. Mitzenmacher: *Less hashing, same performance: Building a better Bloom filter*, in Algorithms–ESA 2006, Springer, 2006, pp. 456–467.
21. V. Kunin, A. Engelbrektson, H. Ochman, and P. Hugenholtz: *Wrinkles in the rare biosphere: pyrosequencing errors can lead to artificial inflation of diversity estimates.* Environ. Microbiol., 12(1) Jan 2010, pp. 118–123.
22. B. Langmead and S. L. Salzberg: *Fast gapped-read alignment with Bowtie 2.* Nature methods, 9(4) 2012, pp. 357–359.
23. H. Li and R. Durbin: *Fast and accurate short read alignment with Burrows–Wheeler transform.* Bioinformatics, 25(14) 2009, pp. 1754–1760.
24. P. Lopez, S. Halary, and E. Bapteste: *Highly divergent ancient gene families in metagenomic samples are compatible with additional divisions of life.* Biol. Direct, 10 2015, p. 64.
25. N. Maillet, G. Collet, T. Vannier, D. Lavenier, and P. Peterlongo: *COMMET: comparing and combining multiple metagenomic datasets*, in Bioinformatics and Biomedicine (BIBM), 2014 IEEE International Conference on, IEEE, 2014, pp. 94–98.
26. N. Maillet, C. Lemaitre, R. Chikhi, D. Lavenier, and P. Peterlongo: *Compareads: comparing huge metagenomic experiments.* BMC Bioinformatics, 13(19) 2012, pp. 1–10.
27. G. Marsaglia: *Xorshift rngs.* Journal of Statistical Software, 8(14) 2003, pp. 1–6.
28. G. Rizk, D. Lavenier, and R. Chikhi: *DSK: K-mer counting with very low memory usage.* Bioinformatics, 29(5) 2013, pp. 652–653.
29. G. Robertson, J. Schein, R. Chiu, R. Corbett, M. Field, S. D. Jackman, K. Mungall, S. Lee, H. M. Okada, J. Q. Qian, et al.: *De novo assembly and analysis of RNA-seq data.* Nature methods, 7(11) 2010, pp. 909–912.
30. M. Schirmer, U. Z. Ijaz, R. D'Amore, N. Hall, W. T. Sloan, and C. Quince: *Insight into biases and sequencing errors for amplicon sequencing with the illumina miseq platform.* Nucleic Acids Research, 2015.
31. S. C. Schuster: *Next-generation sequencing transforms todays biology.* Nature, 200(8) 2007, pp. 16–18.
32. D. Sharon, H. Tilgner, F. Grubert, and M. Snyder: *A single-molecule long-read survey of the human transcriptome.* Nature biotechnology, 31(11) 2013, pp. 1009–1014.
33. H. Tilgner, F. Grubert, D. Sharon, and M. P. Snyder: *Defining a personal, allele-specific, and single-molecule long-read transcriptome.* Proceedings of the National Academy of Sciences, 111(27) 2014, pp. 9869–9874.
34. F. Völkel, E. Bapteste, M. Habib, P. Lopez, and C. Vigliotti: *Read networks and k-laminar graphs.* arXiv, 2016, pp. 1–14.
35. E. Zorita, P. Cuscó, and G. J. Filion: *Starcode: sequence clustering based on all-pairs search.* Bioinformatics, 31(12) jun 2015, pp. 1913–1919.

# 4 Appendix

Appendix contains a presentation of the SRC_linker algorithm using disk for storing values (Algorithm 5).

**Algorithm 5:** SRC_linker_Disk: Quasi-dictionary used for identifying read similarities

```
   Data: Read set B, read set Q, k ∈ ℕ, t ∈ ℕ, f ∈ ℕ
   Result: For each read from Q, its k-mer similarity with each read from set B
1  quasi-dictionary QD = create_quasidictionary(B, k, t, f) ;
2  create a table ids composed of N integers^a all valued to 0;
3  foreach read b in B do
4  |  foreach k-mer w in b do
5  |  |  index = query_quasidictionary(w);
6  |  |  if index ≥ 0 then
7  |  |  |  add 1 to ids[index];
8  foreach Solid k-mer w from B do
9  |  index = query_quasidictionary(w);
10 |  if index ≥ 0 then
11 |  |  count = ids[index];
12 |  |  ids[index] = Temporary_File.position;
13 |  |  write count + 1 '0' on Temporary_File;
14 foreach read b in B do
15 |  foreach k-mer w in b do
16 |  |  index = query_quasidictionary(w);
17 |  |  if index ≥ 0 then
18 |  |  |  position = ids[index];
19 |  |  |  Temporary_File.goto(position);
20 |  |  |  write id of b in place of the first 0 found;
21 foreach read q in Q do
22 |  create a hash table targets (target_read_id) → couple(next_free_position, count);
23 |  foreach i in [0, |q| − k] do
24 |  |  w = k-mer occurring position i in q;
25 |  |  index = query_quasidictionary(w);
26 |  |  if index ≥ 0 then
27 |  |  |  position = ids[index];
28 |  |  |  Temporary_File.goto(position);
29 |  |  |  read from Temporary_File and put in vector V all integer until a 0 is found;
30 |  |  |  foreach tg_id in vector V do
31 |  |  |  |  if targets[tg_id] is empty then
32 |  |  |  |  |  targets[tg_id].next_free_position = 0
33 |  |  |  |  targets[tg_id].count+ = max (k, i + k − targets[tg_id].next_free_position)
   |  |  |  |  targets[tg_id].next_free_position = i + k
34 |  Output the id of q and each^b tg_id associate to its count from targets table;
```

[a] with $N$ the number of solid $k$-mers from $\mathcal{B}$

[b] In practice only *tg_id* whose *count* value is higher or equal to a user defined threshold are output

# The String Matching Algorithms Research Tool

Simone Faro[1], Thierry Lecroq[2], Stefano Borzì[1],
Simone Di Mauro[1], and Alessandro Maggio[1]

[1] Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
[2] Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it

**Abstract.** String matching is the problem of finding all occurrences of a given pattern in a given text. It is an extensively studied problem in computer science because of its direct application to several areas such as text, image and signal processing, speech analysis and recognition, data compression, information retrieval, computational biology and chemistry. Since 1970 more than 85 string matching algorithms have been proposed, and more than 50% of them in the last ten years.
In this paper we present Smart, an efficient and flexible tool designed for developing, testing, comparing and evaluating string matching algorithms. It also provides the most comprehensive survey of online exact single string matching algorithms together with a set of corpora available for testing purposes.

**Keywords:** string matching, text processing, design and analysis of algorithms, testing framework, algorithms survey, experimental evaluation

## 1 Introduction

String matching is a very important subject in the wider domain of text processing. It consists in finding *all* occurrences of a given pattern in a given text. It is an extensively studied problem in computer science, mainly due to its direct applications in many areas related with information retrieval and information analysis. String matching algorithms are also basic components used in implementations of practical software existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. Finally they also play an important role in theoretical computer science by providing challenging problems.

Applications require two kinds of solutions depending on which string, the pattern or the text, is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem. In this paper we are interested in this kind of problem, which is generally referred as *online* string matching. Recently Faro and Lecroq presented a comprehensive survey [17] of almost all online string matching algorithms appeared in literature up to 2010.

In 1991 Hume and Sunday presented an efficient framework [20] for testing string matching algorithms. It was developed in the C programming language and has been extensively used in the field during the last few decades. The authors compiled their framework using the stringsearch package[1] including the implementations of 37 string matching algorithms. Although their tool is very useful and simple to be used it presents some questionable points.

[1] `http://hackage.haskell.org/package/stringsearch-0.3.6.4`

Simone Faro, Thierry Lecroq, Stefano Borzì, Simone Di Mauro, Alessandro Maggio: The String Matching Algorithms Research Tool, pp. 99–113.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

First of all, it was designed in order to maintain the preprocessing and the searching phase as separate functions. Although this allows an accurate measurement of the preprocessing time, it needs also the pattern to be copied to some local buffer, affecting the time measurement with an additional overhead which becomes negligible only if the length of the text is large enough. In addition all data related with the pattern are stored in common structure, thus guaranteeing a similar behavior of the algorithms. However it may slightly increase the access time to pattern information during the searching phase.

A useful feature of the tool from Hume and Sunday is that all algorithms are implemented in separate files. However their compilation is not independent from the whole framework. Thus when testing several algorithms there is always a risk that they meddle with each other.

We also noticed that the use of the `signed char` type does not always work properly, because indexing of arrays does not work with negative values. This could be avoided by forcing the cast to the `unsigned char` type all values which are used as indexes.

In this paper we introduce a new version of Smart (String Matching Algorithms Research Tool), an efficient and flexible framework designed for developing, testing, comparing and evaluating string matching algorithms. It includes most of the features which characterize the tool by Hume and Sunday, but includes a lot of other useful and interesting improvements. It allows the user to completely customize the testing environment, adding new algorithms and testing them for correctness, without the need of recompiling it. Moreover it includes the implementation of more than 120 algorithms, divided into more than 300 variants, and a large set of corpora, divided into categories, which can be used inside the testing environment. Finally it has been designed in order to provide a fair comparison between different solutions, thanks also to a large variety of experimental observations.

In May 2010 a preliminary version of Smart was released to the scientific community and referred in a technical report [13] were the authors discussed a comprehensive evaluation of almost all exact string matching algorithms[2] known up to 2010. In the last six years it was used many times to perform experimental evaluation (see for instance [18,2,23,15,10]) and to test the performances of new algorithms.

One of the most interesting features of the new version of Smart is a practical and useful graphical interface which works over the standard framework and allows the use of all functionalities of the tool.

The source file of Smart, together with a detailed description and documentation, can be found at `http://www.dmi.unict.it/~faro/smart/`. It is distributed using GitHub[3] at `https://github.com/smart-tool`.

The paper is organized as follows. In Section 2 we give a brief description of the Smart tool, including its main features and the principles of fair testing which are at the basis of the framework. We give also a list of all implemented algorithms and the corpora which are included in Smart. Then in Section 4 we give a brief survey of the last comprehensive experimental evaluation performed with Smart, and describe the directions of future works in Section 5.

[2] The preliminary version of Smart included 85 different algorithms, divided in 130 variants.

[3] The GitHub web page is accessible at `https://github.com`

## 2 The Smart Tool in Short

Smart is an open source software which 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. The Smart source code can be downloaded at the web page `http://www.dmi.unict.it/~faro/smart/`. It is released under the GNU general public license[4].

Smart is written in the C language and can be compiled in any operating system with a standard `gcc` compiler. The tool uses shared memory for storing the text. Thus Smart requires the system to allow the allocation of shared memory. The default size of the text is 1MB, which is small enough to be supported by any system. However if one wants to use Smart for testing algorithms on larger texts, system settings for shared memory must be checked.

In the following sections we briefly describe Smart's main features.

### 2.1 Implemented Algorithms

In the last 40 years tens of string matching algorithms (and even a larger number of variants thereof) have been proposed. Smart provides the implementation of more than 300 variants and more than 120 different algorithms. This number is on the rise thanks to the continuous contributions of the community.

All implemented algorithms can be divided into four classes (but further classifications are possible): *characters comparison*, *deterministic automata*, *bit parallelism* and *packed string matching*. Classical approaches to the problem make use of *comparisons between characters* or perform transitions on some kinds of *deterministic automata*. However in the last two decades a lot of work has been made in order to exploit the power of the word RAM model of computation to speed-up classical string matching algorithms. In this model, the computer operates on words of length $\omega$, thus blocks of characters are read and processed at once. This means that usual arithmetic and logic operations on the words all take one unit of time. Most of the solutions which exploit the word RAM model are based on the *bit-parallelism* technique or on the *packed string matching* technique.

An almost comprehensive list of all algorithms implemented in the preliminary version of Smart (more than 85) can be found in [13,17]. In the present release of the software we have extended such list introducing some additional variants of previous solutions and the following new algorithms presented between 2010 and 2016. The comprehensive list of algorithms implemented in the new version of Smart (more than 120) can be found in [8].

- Three bit-parallel algorithms for exact searching of long patterns appeared in [7]. Two algorithms are modifications of the BNDM [22] algorithm and the third one is a filtration method which utilizes locations of $q$-grams in the pattern. Two algorithms apply a condensed representation of $q$-grams.
- Two generalizations of the Forward-SBNDM [12] algorithm presented in [23]. The first generalizes the algorithm by using $q$-grams while the second introduces also a $q$-gram lookahead approach.

[4] http://www.gnu.org/licenses/gpl.html

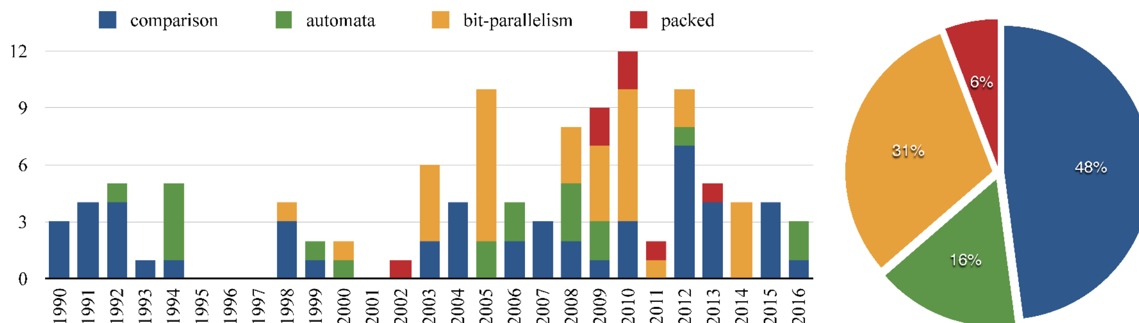



**Figure 1.** (On the left) The temporal distribution of algorithms proposed in the last 26 years (1990-2016) and (on the right) the percentage of all algorithms up to 2016.

- A solution based on a factorization of the pattern and on a particular encoding of the suffix automaton [16], which turns out to lead to longer shifts than that proposed by other known solutions which make use of suffix automata.
- A constant-space $\mathcal{O}(n)$-time packed string matching algorithm [2] which runs in optimal $\mathcal{O}(n/\alpha)$-time, where $\alpha = w/\log\sigma$, and even in real-time.
- Several variants of previous solutions obtained by using a general approach to string matching based on multiple sliding text-windows [14].
- A very fast string matching algorithm [11] for short patterns, which uses specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology.
- Two algorithms, presented in [24], based on a combination of the Boyer-Moore [3] and Horspool [19] algorithms. It takes the maximum shift proposed by the two occurrence heuristics.
- Three improvements of the standard occurrence heuristics [4,5].
- An improvement of Quick-Search algorithm [25] which improves the shift performed by the occurrence heuristics by computing the shift to left performed by the reverse of the pattern at a given fixed distance from the current window.
- A combination of Skip-Search and the Hash$q$ algorithms which computes buckets of positions for the fingerprint of each $q$-gram in the pattern. It was presented in [9].
- Improved versions of the Shift-Or and Shift-And algorithms [1] using a two way scan of the window and $q$-grams. They were presented in [6].

Figure 1 presents the temporal distribution of algorithms proposed in the last 26 years (1990-2016) and the percentage of algorithms belonging to each class, up to 2016. Observe that the number of proposed solutions have doubled in the last ten years, demonstrating the increasing interest in this issue.

The class of algorithms based on comparison of characters is the wider class and consists of almost 50% of all solutions. Also automata play a very important role in the design of efficient string matching algorithms and have been developed to design algorithms which have optimal sub-linear performance on average. Almost 20% of all algorithms in Smart are based on automata.

Bit-parallelism [1] takes advantage of the intrinsic parallelism of the bit operations automata. It is interesting to observe also that almost 50% of solutions in the last ten years (and 31% all along) are based on bit-parallelism, and it seems that such number follows an increasing trend.

| No. | Algorithm | Year |
|---|---|---|
| 1. | Brute-Force (BF) | no date |
| 2. | Deterministic-Finite-Automaton (DFA) | |
| 3. | Morris-Pratt (MP) | 1970 |
| 4. | Knuth-Morris-Pratt (KMP) | 1977 |
| 5. | Boyer-Moore (BM) | |
| 6. | Horspool (HOR) | 1980 |
| 7. | Galil-Seiferas (GS) | 1981 |
| 8. | Apostolico-Giancarlo (AG) | 1986 |
| 9. | Karp-Rabin (KR) | 1987 |
| 10. | Zhu-Takaoka (ZT) | |
| 11. | Shift-Or (SO) | 1989 |
| 12. | Shift-And (SA) | |
| 13. | Quick-Search (QS) | 1990 |
| 14. | Optimal-Mismatch (OM) | |
| 15. | Maximal-Shift (MS) | |
| 16. | Apostolico-Crochemore (AC) | 1991 |
| 17. | Two-Way (TW) | |
| 18. | Tuned-Boyer-Moore (TunBM) | |
| 19. | Colussi (COL) | |
| 20. | Smith (SMITH) | |
| 21. | Galil-Giancarlo (GG) | 1992 |
| 22. | Raita (RAITA) | |
| 23. | S.M. on Ordered ALphabet (SMOA) | |
| 24. | Turbo-Boyer-Moore (TBM) | |
| 25. | Reverse-Factor (RF) | |
| 26. | Not-So-Naive (NSN) | 1993 |
| 27. | Reverse-Colussi (RCOL) | 1994 |
| 28. | Simon (SIM) | |
| 29. | Turbo-Reverse-Factor (TRF) | |
| 30. | Forward-DAWG-Matching (FDM) | |
| 31. | Backward-DAWG-Matching (BDM) | |
| 32. | Skip-Search (SKIP) | 1998 |
| 33. | Alpha-Skip-Search (ASKIP) | |
| 34. | Knuth-Morris-Pratt Skip-Search (KMPS) | |
| 35. | Nondeterministic BDM (BNDM) | |
| 36. | Berry-Ravindran (BR) | 1999 |
| 37. | Backward-Oracle-Matching (BOM) | |
| 38. | Double Forward DAWG Matching (DFDM) | 2000 |
| 39. | BNDM for Long patterns (BNDML) | |
| 40. | Super Alphabet Simulation (SAS) | 2002 |
| 41. | Ahmed-Kaykobad-Chowdhury (AKC) | 2003 |
| 42. | Fast-Search (FS) | |
| 43. | Simplified BNDM (SBNDM) | |
| 44. | Two-Way NDM (TNDM) | |
| 45. | Long patterns BNDM (LBNDM) | |
| 46. | Shift Vector Matching (SVM) | |
| 47. | Forward-Fast-Search (FFS) | 2004 |
| 48. | Backward-Fast-Search (BFS) | |
| 49. | Tailed-Substring (TS) | |
| 50. | Sheik *et al.* (SSABS) | |
| 51. | Wide Window (WW) | 2005 |
| 52. | Linear DAWG Matching (LDM) | |
| 53. | BNDM with loop-unrolling (BNDM2) | |
| 54. | SBNDM with loop-unrolling (SBNDM2) | |
| 55. | BNDM with Horspool Shift (BNDMBMH) | |
| 56. | Horspool with BNDM test (BMHBNDM) | |
| 57. | Forward NDM (FNDM) | |
| 58. | Bit parallel Wide Window (BWW) | |
| 59. | Average Optimal Shift-Or (AOSO) | |
| 60. | Fast Average Optimal Shift-Or (FAOSO) | |
| 61. | Thathoo *et al.* (TVSBS) | 2006 |
| 62. | Horspool using Probabilities (PBMH) | |
| 63. | Improved LDM (ILDM1) | |
| 64. | Improved LDM 2 (ILDM2) | |
| 65. | Franek-Jennings-Smyth (FJS) | 2007 |
| 66. | 2-Block Boyer-Moore (2BLOCK) | |
| 67. | Wu-Manber for Single S.M. (HASHq) | |
| 68. | Horspool with $q$-grams (BMHq) | 2008 |
| 69. | Two Sliding Windows (TSW) | |
| 70. | Extended BOM (EBOM) | |
| 71. | Forward BOM (FBOM) | |
| 72. | Succint BDM (SBDM) | |
| 73. | Forward BNDM (FBNDM) | |
| 74. | Forward Simplified BNDM (FSBNDM) | |
| 75. | Bit-Parallel Length Invariant (BLIM) | |
| 76. | Genomic Rapid Algo for S.M. (GRASPm) | 2009 |
| 77. | Simplified Extended BOM (SEBOM) | |
| 78. | Simplified Forward BOM (SFBOM) | |
| 79. | BNDM with $q$-grams (BNDMq) | |
| 80. | Simplified BNDM with $q$-grams (SBNDMq) | |
| 81. | FNDM with $q$-grams (UFNDMq) | |
| 82. | Small Alphabet Bit-Parallel (SABP) | |
| 83. | Packed String Search (PSS) | |
| 84. | Streaming SIMD Extensions Filter (SSEF) | |
| 85. | Bounded Boyer-Moore (BBM) | 2010 |
| 86. | Bounded Fast-Search (BFS) | |
| 87. | Bounded Forward-Fast-Search (BFFS) | |
| 88. | BNDM with Extended Shifts (BXS) | |
| 89. | BNDMq Long (BQL) | |
| 90. | Q-Gram Filtering (QF) | |
| 91. | Bit-Parallel$^2$ Wide-Window (BP2WW) | |
| 92. | Bit-Parallel Wide-Window$^2$ (BPWW2) | |
| 93. | Factorized Shift-And (KSA) | |
| 94. | Factorized BNDM (KBNDM) | |
| 95. | Packed Belazzougui (PB) | |
| 96. | Packed Belazzougui-Raffinot (PBR) | |
| 97. | FSBNDM with $q$-grams (FSBNDMqf) | 2011 |
| 98. | Packed Crochemore-Perrin (SSECP) | |
| 99. | Fast-Search using Multiple Windows (FSw) | 2012 |
| 100. | TVSBS with Multiple Windows (TVSBSw) | |
| 101. | Max Shift Boyer-Moore (MSBM) | |
| 102. | Max Shift Horspool (MSH) | |
| 103. | Enhanced Two Sliding Windows (ETSW) | |
| 104. | Hash$q$ using Multiple Hashing (MHASHq) | |
| 105. | Enhanced Berry-Ravindran (RSA) | |
| 106. | Backward SNR DAWG Matching (BSDM) | |
| 107. | Multiple Windows SBNDM (SBNDMw) | |
| 108. | Multiple Windows FSBNDM (FSBNDMw) | |
| 109. | Enhanced RS-A (ERSA) | 2013 |
| 110. | Improved Occurrence Heuristics (IOM) | |
| 111. | Worst Occurrence Heuristics (WOM) | |
| 112. | Jumping Occurrence Heuristics (JOM) | |
| 113. | Exact Packed String Matching (EPSM) | |
| 114. | Improved Two-Way Shift-And (TSA) | 2014 |
| 115. | Improved Two-Way Shift-Or (TSO) | |
| 116. | Two-Way Shift-And using $q$-grams (TSAq) | |
| 117. | Two-Way Shift-Or using $q$-grams (TSOq) | |
| 118. | Simple String Matching (SSM) | 2015 |
| 119. | Quantum Leap Quick-Search (QLQS) | |
| 120. | Enhanced ERS-A (EERSA) | |
| 121. | Four Sliding Windows (FSW) | |
| 122. | Skip-Search using $q$-grams (SKIPq) | 2016 |
| 123. | BSDM with $q$-grams (BSDMqx) | |
| 124. | BSDM$qx$ multiple windows (BSDMqxw) | |

**Table 1.** The list of those string matching algorithms implemented in SMART which were published 1970-2016. Each algorithm is associated to its acronym used in Smart.

In packed string matching, multiple characters are packed into one larger word, so that the characters can be compared in bulk rather than individually. In this context, if the characters of a string are drawn from an alphabet of size $\sigma$, then $\lfloor w/\log\sigma\rfloor$ different characters fit in a single word, using $\lfloor\log\sigma\rfloor$ bits per characters. Although algorithms in this class appeared in the last four years they turn out to be among the fastest solutions [21,10], reaching in some cases the optimal $\mathcal{O}(n\log\sigma/w)$ time complexity [2].

### 2.2 Algorithm Testing and Evaluation

The main command provided by the tool, `smart` indeed, is used for running experimental tests. The experimental settings could be almost completely customized. The easiest way to use SMART is to run a single search for a custom pattern and a custom text. To this purpose one should use the `-simple` parameter followed by the pattern and the text. Otherwise it is possible to select the corpus which will be used to compute the experimental results by typing the parameter `-text` followed by the name of the selected corpus (for ex. `smart -text genome`). It is also possible to select more than one corpus by typing the name of the corpora, separated by a dash symbol (for ex. `smart -text genome-protein`). You can also type the parameter `-text all` in order to run experimental tests for all corpora, in which case the corpora will be processed one after another.

For each input file, SMART generates sets of $r$ patterns of fixed length, randomly extracted from the text, where the length of the patterns ranges over the set of values $\{2^k \mid 1 \leq k \leq 12\}$, so that running times can be easily reported in a log-scale plot. The value $r$ is set to 500 by default, but it is allowed to use the parameter `-pset` in order to modify the size of the set of patterns generated by the tool (for ex. you can type `smart -text genome -pset 100`). You can also use the parameter `-short` in order to perform experimental tests on short patterns, whose length ranges over the set of values $\{2+\ell \mid 0 \leq \ell \leq 30\}$. If necessary, it is also allowed to restrict the pattern's length to a given range by using the parameter `-plen` and indicating an upper bound and a lower bound for such lengths (for ex. you can type the command `smart -text genome -plen 8 64`).

Many algorithms are very slow under particular conditions. In order to avoid excessive running times during the experimental evaluation it is possible to set a *time bound* which cannot be exceeded by any single run. This can be done by using the parameter `-tb`, followed by a value expressed in milliseconds. By default such bound is set to 300 ms.

The SMART tool has been developed in order to follow the principles of fair testing in string matching. In particular the experimental testing is based on the following features:

*Algorithm verification*
The tool verifies that all tested algorithms work properly. This verification is done by counting the number of matches returned by the procedure and testing whether the search stops properly at the end of the text. Since all searched patterns are always randomly extracted from the text, it is guaranteed that the number of occurrences is always equal or greater than 1. It is not uncommon that some algorithms do not work for specific input parameters. For instance a $q$-gram based algorithm does not work for patterns shorter than $q$ characters. The framework also provides a control

mechanism able to distinguish a malfunctioning from a not working instance, in which case an error message is returned by the tool.

*Clean time measuring*
The SMART tool has been designed in order to rule out from time measurements all disturbing events. To this purpose the reading of data (text and patterns) belongs to an outer part of the test setting, so that times spent to reading is excluded from time measurements. Moreover printing of matches is not performed during time measurement, since printing produces also an additional overhead, which is partly unsynchronized. Finally the framework has been developed in order that the time measurement itself does not disturb the work of algorithms.

*Fair comparison*
Since in SMART the measuring is focused on performance, the tested algorithms have been implemented in a uniform way, using the same standard for processing string characters, compare them, performing automata transitions, computing matches and for more other common tasks. Then SMART uses the `-O3` level of optimization, which is the highest optimization level. All algorithms are available online and can be analyzed and improved by the community. During time measurement all tested algorithms share the same input data in order to allow a fair comparison. Moreover the SMART tool has been designed in order to ensure that no residual information in cache, during multiple executions of the same algorithm, may be used in the next attempt affecting the time measurement.

*Algorithm preprocessing*
It has never been clear in the literature if preprocessing should be included in the measurements. The work done in preprocessing is only a proportion of the whole task, and it may depend on the pattern length and on the alphabet size. According to Horspool [19] in string matching the timings do not include the work of initializing tables, however for an *online* algorithm, preprocessing time is spent to compute useful informations which are then used for speeding up the search process. Moreover, it is also true that in most cases the longer is the preprocessing time the faster is the searching. This line of reasoning finds its borderline case in an *offline* algorithms, where a preprocessing of the text leads to an extremely fast searching phase. For this reason the SMART tool has been developed in order to include the preprocessing time in the performance measurement.

However the tool can be set up in order to separate time measurements in searching and preprocessing times. This can be done by using the parameter `-pre`. By default SMART produces only a single time measurement which includes preprocessing and searching time.

*Stability measurement*
It is also useful to find out how accurately repeatable the results are. To this purpose using only average running times easily hides important details. It is common, in fact, that for some algorithms running times move away from the mean value. This is what in general we associate with the *stability* of a given algorithm: the smaller is the standard deviation of the running times the superior is its stability. Thus the SMART tool can be set up in order to compute also standard deviation values of running times (use the parameter `-std`). In addition the tool also allows the computation of

the best and worst running times obtained during the experimental evaluation (use the parameter `-dif`).

### 2.3 Output Formats

The Smart tool associates to any experimental test a unique alphanumeric code on 13 characters, beginning with the prefix `EXP` and followed by a string of 10 numbers computed from the timestamp. At the end of the execution of an experimental test Smart stores experimental data in the directory `results/EXPCODE`, where `EXPCODE` is the unique code associated with the experimental test. Files containing experimental data are named with the name of the corpus which has been selected. The system can store experimental data in different formats: simple text, LaTeX, xml, html and php format.

Files in xml format report data in a structured way suitable to be processed or included in other documents, while html files present data in a tabular format. An additional `index.html` file is generated which contains the list of all html pages containing the experimental results computed during the test. Figure 2 shows a portion of the HTML output produced by Smart.

Finally, LaTeX files can be generated to make easy the inclusion of tables containing the experimental results in LaTeX source files.

### 2.4 Text Corpora

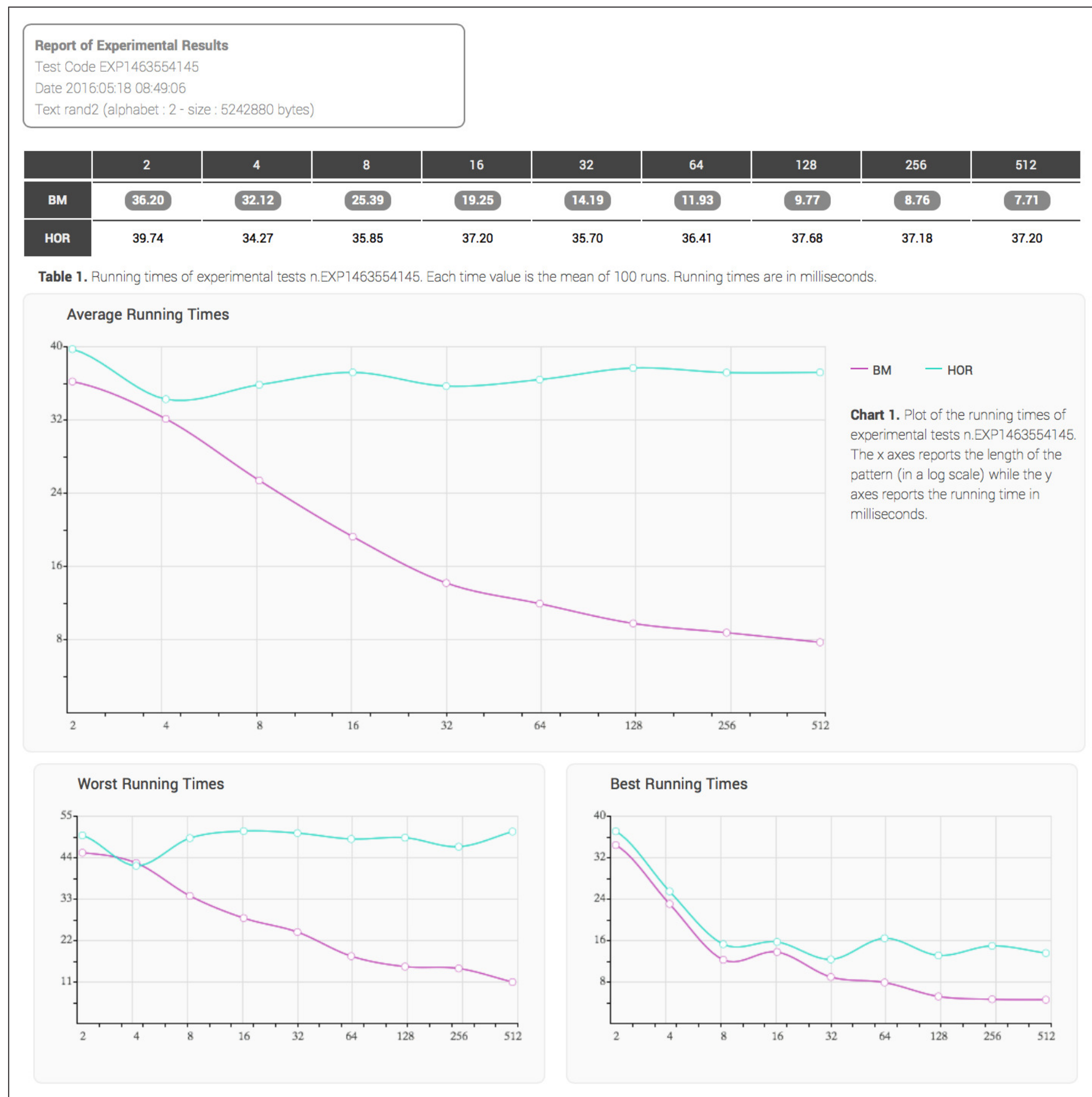
Smart comes with a set of corpora which can be selected in for running experimental results. The corpora are stored in a directory named "`data`", and each corpus consists of a set of texts stored in a sub-directory with the same name of the corpus. Each sub-directory contains and index file (`index.txt`) containing the names and a description of all the files contained in the corpus. It is possible to select the corpus which will be used to compute the experimental results by typing the parameter `-text` followed by the name of the selected corpus.

The Smart tool allows to set an upper bound dimension of the text size used during the experimental results. By default this upper bound dimension is set to 1MB. This means that (at most) the first 1MB of the selected corpus will be used for testing. The default upper bound dimension can be changed by using the parameter `-tsize`, followed by an integer value which indicate the dimension, in Mbytes, which will be used (for ex. `smart -text genome -tsize 5`).

Then all files listed in the index will be loaded in a text buffer, one by one, until the upper bound is reached. If the upper bound is larger than the whole size of the corpus the list of files is processed again in order to fill the whole buffer[5]. In details, Smart provides the following set of 15 corpora.

(i) `englishTexts`, a set of english texts (6.1 MB) over an alphabet of 94 characters. It includes two text of size 3.9 MB and 2.4 MB, respectively.
(ii) `italianTexts`, a set of italian texts (5 MB) over an alphabet of 120 characters. It includes seven texts whose size ranges from 281 KB to 1.5 MB.
(iii) `frenchTexts`, a set of french texts (6.6 MB) over an alphabet of 119 characters. It includes seven texts whose size ranges from 631 KB to 1.2 MB.

[5] Note that during the experimental evaluation the text buffer is stored in shared memory, thus if you set the upper bound to a value $K$ MB it is necessary to ascertain your system allows the allocation of at least $K$ MB of shared memory




| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|---|---|---|
| BM | 36.20 | 32.12 | 25.39 | 19.25 | 14.19 | 11.93 | 9.77 | 8.76 | 7.71 |
| HOR | 39.74 | 34.27 | 35.85 | 37.20 | 35.70 | 36.41 | 37.68 | 37.18 | 37.20 |

**Table 1.** Running times of experimental tests n.EXP1463554145. Each time value is the mean of 100 runs. Running times are in milliseconds.



**Figure 2.** A portion of the HTML output produced by SMART. Experimental data are reported in tabular and in graphical form. Here we observe average running times, worst running times and best running times of Horspool and Boyer-Moore compared on a Rand2 text buffer.

(iv) `chineseTexts`, a set of chinese texts (5.7 MB) over an alphabet of 160 characters. It includes five texts whose size ranges from 745 KB to 2.3 MB.

(v) `genome`, a set of DNA sequences (4.4 MB) over an alphabet of 4 characters. It includes Complete genome of the E. Coli bacterium (4.4 MB).

(vi) `protein`, a set of protein sequences (3.1 MB) over an alphabet of 20 characters. It includes a protein sequence from the Human genome (3.1 MB).

(vii) `midimusic`, a set of midi sequences (2.7 MB) over an alphabet of 117 characters. It includes 206 midi files on Johann Sebastian Bach work (1685-1750) whose size ranges from 4 KB to 205 KB.

(viii) `rand`$\sigma$, random texts (5 MB) over an alphabet of size $\sigma$ with a uniform distribution, where $\sigma$ ranges over the values $\{2, 4, 8, 16, 32, 64, 128, 256\}$.

Files (i) and (v) are from the Large Canterbury Corpus[6], files (ii), (iii) and (iv) are from the Gutenberg project[7] while file (vi) is from the Protein Corpus[8]. Finally files in (vii) are from the Johann Sebastian Bach Midi Page[9].

In addition the Smart tool provides a simple way for adding new corpora to the default set. This can be done by simply introducing a new sub-directory named with the name of the corpus, and containing the set of selected files together with an index file listing their names.

### 2.5 Adding New Algorithms

The Smart tool is not only a framework for testing all known string matching algorithms. It provides also an easy and fast way for assisting researchers to develop and test new efficient algorithms. It is possible to add new string matching algorithms to Smart, testing them for correctness and compare their efficiency against the previous solutions.

The following few requirements must be guaranteed: a new algorithm must be implemented in the `C` programming language and must include the header file "`include/main.h`". The main method must be defined as

```
int search(unsigned char *x, int m, unsigned char *y, int n)
```

where `x` maintains the pattern, `y` maintains the text, while `m` and `n` are their lengths, respectively. The method must return the number of occurrences of the pattern in the text. In addition, if the algorithm does not run under particular conditions (for instance when the length of the pattern is less than a given value), it is required the algorithm to return the value $-1$.

Since preprocessing time is computed separately from searching time, it is required to arrange the code concerning the preprocessing phase between the macros `BEGIN_PREPROCESSING` and `END_PREPROCESSING`, while the code concerning the searching phase must be arranged between the macros `BEGIN_SEARCHING` and `END_SEARCHING`. Figure 3 present the `C` code of the Horspool algorithm in a format suitable for inclusion in Smart.

Before compiling the `C` file, copy the header file `main.h` (which is stored in the folder `source/algos/include`) in the same directory. Then put the compiled binary file in the directory "`source/bin`". Before running a new experimental setting you can test the correctness of your algorithm by executing the command "`./test algoname`", where `algoname` is the name of the binary file of your algorithm. Then it will be possible to include the new algorithm in Smart by typing the command "`./select -add algoname`".

## 3 A Graphical User Interface

The new version of Smart comes with a useful Graphical User Interface (SmartGUI) which could be used for running experimental results. It is implemented in `C++` using the Qt WebKit, one of the major engine to render webpages and execute JavaScript code. Figure 4 shows a screenshot of the SmartGUI where reporting the average

[6] `http://www.data-compression.info/Corpora/CanterburyCorpus/`
[7] `http://www.gutenberg.org`
[8] `http://data-compression.info/Corpora/ProteinCorpus/`
[9] `http://www.bachcentral.com`

```
#include "include/define.h"
#include "include/main.h"

int search(unsigned char *P, int m, unsigned char *T, int n) {
	int i, s, count, hbc[SIGMA];

	BEGIN_PREPROCESSING
	for(i=0;i<SIGMA;i++) hbc[i] = m;
	for(i=0;i<m-1;i++) hbc[P[i]] = m-i-1;
	END_PREPROCESSING

	BEGIN_SEARCHING
	s = 0;
	count = 0;
	while(s <= n-m) {
		i = 0;
		while( i<m && P[i]==T[s+i] ) i++;
		if( i==m ) count++;
		s += hbc[T[s+m-1]];
	}
	END_SEARCHING
	return count;
}
```

**Figure 3.** The C code of the Horspool algorithm for string matching.

running times of the Horspool and Boyer-Moore algorithms when compared on a genome sequence.

The central part of SmartGUI is dedicated to report graphs of running times. The text output is reported next to the graphs, giving a familiar feedback to users which execute Smart using the terminal. When several experimental evaluations are executed, SmartGUI organizes the graphs using tabs.

SmartGUI has been developed to make easier the use of Smart. It allows to view in real-time all the results of the experimental evaluations and to compare the single algorithms of the experiments through the tabs.

In addition SmartGUI makes easy the customization of the tests through input text parameters and checkbox to choose the output format (txt, LaTeX, pdf, etc.) and the text buffers. It also allows to select algorithms or add new algorithms and run tests on it. During any test SmartGUI shows the status through a progress bar. When a single experimental evaluation ends SmartGUI shows more statistics through a web view opening the html format results.

Executable binaries of SmartGUI are available at the Smart web page for Windows, Linux and Mac. It can be downloaded and installed in any folder of your computer, even different from your main Smart folder. At the first execution of SmartGUI it is necessary to link the main Smart folder using the setup procedure available at the menu button `setup smart gui`.

## 4 Experimental Evaluation

In this section we discuss experimental results which could be performed using Smart. The details of the experimental evaluation can be analyzed at the Smart web page (`http://www.dmi.unict.it/~faro/smart/`).

A recent survey [13] already presents an extensive experimental evaluation of almost all string matching algorithms used for searching in different texts. The authors used Smart for computing the experimental results, indeed.

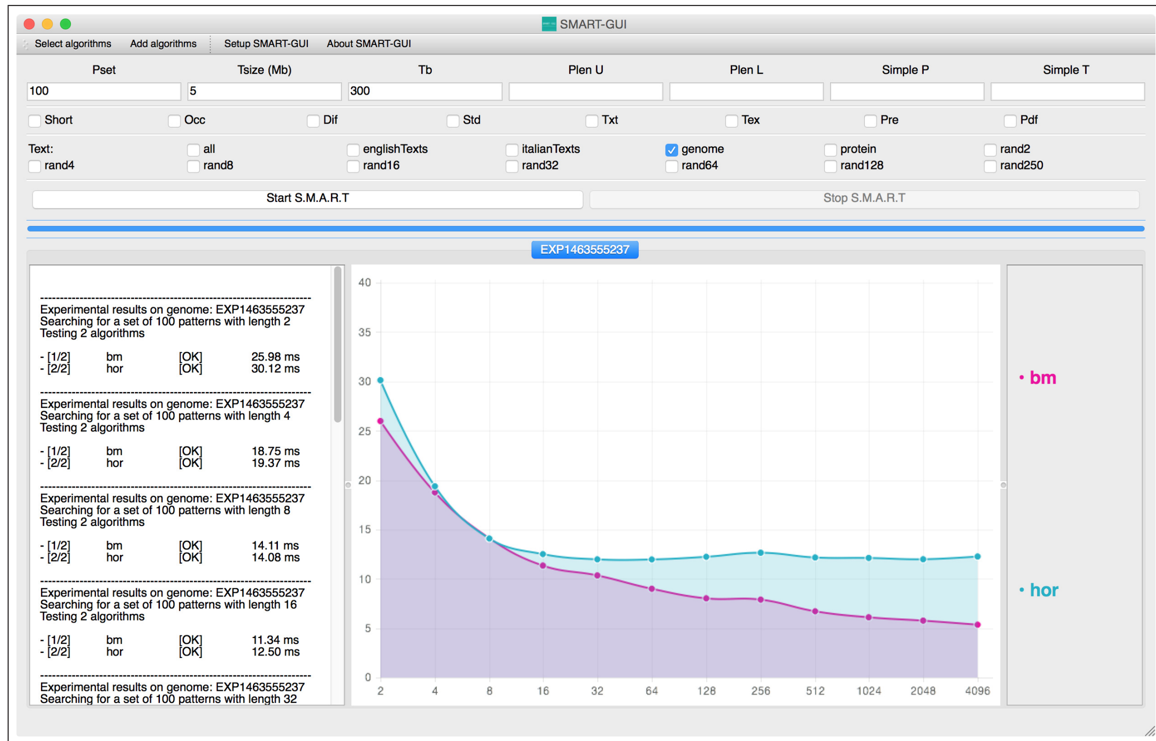



**Figure 4.** A screenshot of the Graphical User Interface of SMART. Here we observe average running times of Horspool and Boyer-Moore algorithms when compared on a genome sequence.

Other more recent experimental evaluations using the SMART tool appeared in [16,14,2,11] where new efficient algorithms were presented.

Thus in this section we do not give another extensive experimental evaluation of string matching algorithms, but we focus our attention on new general experimental observations which SMART allows to do. In particular SMART allows to analyze string matching algorithms from three different points of view: their efficiency, their stability and their flexibility.

*Efficiency Measurement*
The efficiency of an algorithm is evaluated in SMART by computing the mean of running times over a large set of attempts.

Most string matching algorithms are characterized by a performance plot with a decreasing trend. Thus on average the performances increases when the length of the pattern increases. Almost all efficient algorithms, in fact, are based on a sliding window approach whose shift is at most long as the length of the pattern. This behavior is evident, for instance, in Figure 2 and in Figure 4, where we show the experimental results of the well known Horspool [19] and Boyer-Moore [3] algorithms, which are comparison based algorithms based on the occurrence heuristic. Also automata show such decreasing trend, however almost all of them show a decrease in performances in the case of longer patterns. This is due to the size explosion of the underlying automaton and of the correspondent preprocessing time.

Algorithms based on $q$-grams are quite efficient on average. In general their performance increases when the length of the pattern increases or when the value of $q$ increases. However, on the other hand in the case of short patterns their performances drastically decreases when the value of $q$ increases.

*Stability Measurement*
In Smart the stability of an algorithm is computed as the standard deviation of running times observed during the evaluation. Such value shows how much variation exists from the average, i.e. the mean of the running times. A low standard deviation indicates that the running times tend to be very close to the mean, underlying a high stability of the algorithm. On the other hand a high standard deviation indicates that the running times are spread out over a large range of values, thus indicating a low stability. It turns out from our observations that almost all algorithms have a low stability for short patterns while their stability increases when the length of the pattern increases. Such behavior becomes more evident for larger alphabets.

Sometimes an opposite behavior can be observed when searching on texts over a small alphabet like DNA sequences. This is the case, for instance, of some comparison based algorithms based on the occurrence heuristic whose stability decreases when the length of the pattern gets shorter. This trend can be explained by the reduced combination of strings which could be obtained when the pattern is short and the alphabet is small.

*Flexibility Measurement*
Flexibility is used as an attribute of various types of systems. In the field of string matching, it refers to algorithms that can adapt when changes in the input data occur. Thus a string matching algorithm can be considered flexible when, for instance, it maintains good performances for both short and long patterns, or in the case of both small and large alphabets.

As already observed most string matching algorithms obtain good performances only in the case of long patterns sacrificing their performance for short ones. This is a common behavior, for instance, for all algorithm which make use of a sliding window approach. Such approach allows the pattern to slide along the text by performing subsequent shifts. Each shift can be at most as long as the length of the pattern. It turns out that statistically the shift increases when the length of the pattern increases, or when the size of the alphabet increases. Although bit-parallel algorithms are designed to be extremely efficient in the case of long patterns, also this class of algorithms suffers of a lack in flexibility.

Only packed string matching algorithms turn out to have good performances for short patterns. This is the case of the SSECP algorithm [2] whose performances degrade when the length of the pattern increases. It is also the case of the EPSM algorithm [11], whose flexibility is obtained only by combining different algorithms, depending on the length of the pattern.

## 5 Conclusions, Future Works and Acknowledgements

We presented Smart (`http://www.dmi.unict.it/~faro/smart/`) a flexible testing and evaluation tool for single exact string matching algorithms. It contains the implementation of almost all string matching algorithms appeared since 1970 up to 2016. The tool helps researchers in the filed in various way and we encourage them to contribute to the project by providing their own code for testing. Many improvements are possible to enhance Smart, including its adjustment in order to work with 128 bit processors.

We wish to thank Prof. Jorma Tarhio, Dr. M. Oğuzhan Külekci and Dr. Arseny Kapoulkine for helpful comments and suggestions in improving the Smart frame-

work. We wish also to thanks the authors of several string matching algorithms for having provided their codes for inclusion in Smart.

## References

1. R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Commun. ACM, 35(10) 1992, pp. 74–82.
2. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann: *Optimal packed string matching*, in IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India, S. Chakraborty and A. Kumar, eds., vol. 13 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 423–432.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Commun. ACM, 20(10) 1977, pp. 762–772.
4. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics*, in Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2013, pp. 92–106.
5. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics*. J. Discrete Algorithms, 28 2014, pp. 73–84.
6. B. Durian, T. Chhabra, S. S. Ghuman, T. Hirvola, H. Peltola, and J. Tarhio: *Improved two-way bit-parallel search*, in Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 71–83.
7. B. Durian, H. Peltola, L. Salmela, and J. Tarhio: *Bit-parallel search algorithms for long patterns*, in Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings, P. Festa, ed., vol. 6049 of Lecture Notes in Computer Science, Springer, 2010, pp. 129–140.
8. S. Faro: *Exact online string matching bibliography*. CoRR, abs/1605.05067 2016.
9. S. Faro: *A very fast string matching algorithm based on condensed alphabets*, in Algorithmic Aspects in Information and Management - 10th International Conference, AAIM 2016. Proceedings, vol. 9778 of Lecture Notes in Computer Science, Springer, 2016, pp. 65–76.
10. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings, L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, eds., vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
11. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, P. Sanders and N. Zeh, eds., SIAM, 2013, pp. 113–121.
12. S. Faro and T. Lecroq: *Efficient variants of the backward-oracle-matching algorithm*, in Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2008, pp. 146–160.
13. S. Faro and T. Lecroq: *The exact string matching problem: a comprehensive experimental evaluation*. CoRR, abs/1012.2547 2010.
14. S. Faro and T. Lecroq: *Fast searching in biological sequences using multiple hash functions*, in 12th IEEE International Conference on Bioinformatics & Bioengineering, BIBE 2012, Larnaca, Cyprus, November 11-13, 2012, IEEE Computer Society, 2012, pp. 175–180.
15. S. Faro and T. Lecroq: *A fast suffix automata based algorithm for exact online string matching*, in Implementation and Application of Automata - 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings, N. Moreira and R. Reis, eds., vol. 7381 of Lecture Notes in Computer Science, Springer, 2012, pp. 149–158.

16. S. Faro and T. Lecroq: *A multiple sliding windows approach to speed up string matching algorithms*, in Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings, R. Klasing, ed., vol. 7276 of Lecture Notes in Computer Science, Springer, 2012, pp. 172–183.
17. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results*. ACM Comput. Surv., 45(2) 2013, p. 13.
18. F. Hongbo, Y. Nianmin, and M. Haifeng: *A practical and average optimal string matching algorithm based on Lecroq*, in IEEE Internet Computing for Science and Engineering, 2010, pp. 57–63.
19. R. N. Horspool: *Practical fast searching in strings*. Softw., Pract. Exper., 10(6) 1980, pp. 501–506.
20. A. Hume and D. Sunday: *Fast string searching*, in Proceedings of the Summer 1991 USENIX Conference, Nashville, TE, USA, June 1991, USENIX Association, 1991, pp. 221–234.
21. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009, pp. 118–128.
22. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings, M. Farach-Colton, ed., vol. 1448 of Lecture Notes in Computer Science, Springer, 1998, pp. 14–33.
23. H. Peltola and J. Tarhio: *Variations of forward-sbndm*, in Proceedings of the Prague Stringology Conference 2011, Prague, Czech Republic, August 29-31, 2011, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2011, pp. 3–14.
24. M. Sahli and T. Shibuya: *Max-shift BM and max-shift horspool: Practical fast exact string matching algorithms*. JIP, 20(2) 2012, pp. 419–425.
25. D. Sunday: *A very fast substring search algorithm*. Commun. ACM, 33(8) 1990, pp. 132–142.

# Jumbled Matching with SIMD

Sukhpal Singh Ghuman and Jorma Tarhio

Department of Computer Science
Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
firstname.lastname@aalto.fi

**Abstract.** Jumbled pattern matching addresses the problem of finding all permuted occurrences of a substring in a text. We introduce two improved algorithms for exact jumbled matching of short patterns. Our solutions apply SIMD (Single Instruction Multiple Data) computation in order to quickly filter the text. One of them utilizes the equal any operation and the other searches for the least frequent character of the pattern. Our experiments show that the best algorithm is 30% faster than previous algorithms for short English patterns.

## 1 Introduction

Given a text $T = t_0 t_1 \cdots t_{n-1}$ and pattern $P = p_0 p_1 \cdots p_{m-1}$ over a finite alphabet $\Sigma$ of size $\sigma$, the task of exact string matching [30] is to find all the occurrences of $P$ in $T$, i.e. all the positions $i$ such that $t_i t_{i+1} \cdots t_{i+m-1} = p_0 p_1 \cdots p_{m-1}$. In jumbled pattern matching [7,10], the aim is to find all substrings of $T$ which are permutations of $P$. Jumbled matching is also known as permutation matching or Abelian matching. In other words, a substring $u$ of $T$ is a jumbled equivalent to $P$ if the count of each character in $P$ is equal to its count in $u$ and $|P| = |u|$ holds. For example, a permutation $abecd$ of the pattern $P = edcba$ occurs in the text $T = a\underline{abecd}cddee$.

Parikh vectors [33] can be used to identify jumbled substrings. Over a finite ordered alphabet, a Parikh vector $p(S)$ is defined as the vector of multiplicities of the characters of a string $S$. For instance, if $S = baadabbdd$ be a string over $\Sigma = \{a, b, c, d\}$, then the Parikh vector $p(S)$ is (3,3,0,3).

Initially, simple counting solutions [17,25,29] have been presented for jumbled pattern matching. These solutions work in linear time. The main idea of these algorithms is to scan the text from left to right and maintain counts of characters in a sliding alignment window of text. Originally, these counting algorithms were developed as filtration methods for online approximate string matching, but they recognize jumbled patterns as a side-effect when no errors are allowed. Also many other algorithms [6,8,13,16] have been introduced for jumbled pattern matching.

In this article, we introduce new algorithms for exact jumbled matching for short patterns. Our solutions apply the SIMD (Single Instruction Multiple Data) extensions of the SSE technology [18,22,24] which makes possible to process multiple characters at the same time. In this way, we are able to process the text in chunks of 16 characters resulting in faster execution for short patterns. We present two new algorithms based on filtration with SIMD instructions. One of them applies the equal any SIMD operation and the other searches for the least frequent character of the pattern. Our emphasis is on the practical efficiency of the algorithms and we show the competitiveness of the new algorithms by practical experiments. The best one of our algorithms achieves a speed-up of 30% for short English patterns.

Sukhpal Singh Ghuman, Jorma Tarhio: Jumbled Matching with SIMD, pp. 114–124.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

The bit operations represented in pseudocodes are identical to the notations used in the C programming language. The operator << represents the left shift operation and >> corresponds to the right shift operation.

The rest of the paper is organized as follows. Section 2 contains an extensive review on applications of jumbled pattern matching, Section 3 is an introduction to SIMD computation, Section 4 reviews earlier solutions, Sections 5 and 6 describe our new solutions, Section 7 presents the results of practical experiments, and Section 8 concludes the article.

## 2 Applications of Jumbled Matching

In recent past, several variations of jumbled pattern matching have originated. The problem has numerous applications in the field of bioinformatics such as alignment of strings [1], SNP discovery [3], discovery of repeated patterns [14], and the interpretation of mass spectrometry data [2]. Other applications [5] of jumbled pattern matching include string matching with a dyslectic word processor, table rearrangements, anagram checking, scrabble playing, and episode matching.

### 2.1 Gene Clustering

Jumbled matching can be used to find those genes that are closely related to one another. Sequencing of genome has become a regular practice in the last few decades, which in turn has led to the analysis of genomes at gene level [11,36] and the correlation of genes. Genes having similar functionality are correlated to each other. Gene clustering [31,26] helps to find the genes that are closely related to each other irrespective of the order in which they occur. Consistent occurrences of genes in the close proximity across genomes are believed to be functionally related. However, the order of the genes in chromosomes may be different. These kind of gene clusters help in solving the problem of local alignment of genes.

In the case of discovery of repeated patterns [14], jumbled matching algorithms can be used to solve the problem of local alignment of genes. However, the order of the occurrences of genes may not be the same. These can be usually modeled by Parikh fingerprints or character sets [4]. In other words, a group of genes that can appear in different order in genomes may have similarity.

### 2.2 Composition Alignment

In composition matching [1], the main idea is to match among the substrings that have similar composition or order. Composition alignment of two strings $P$ and $T$ having a scoring function $SF(p,t)$ is defined as the composition match between the substrings $p$ and $t$ of $P$ and $T$ as well as the single character match between $P$ and $T$. The task is to find the best scoring alignment.

For example, let $P = GACTGTTATTCCTA$ and $T = GCATGTGGGGATCC$ be two strings over the alphabet $\Sigma = (A, C, G, T)$. One possible composition alignment for $P$ and $T$ is:

$$\underline{GAC}\mathbf{TGT}TATT\underline{CCTA}$$
$$\underline{CGA}\mathbf{TGT}GGGG\underline{ATCC}$$

Here, the characters in bold are used to depict exact single character matches and substring composition matches are represented by underlined substrings. Note that composition matches can occur consecutively in an alignment.

Composition alignment is one of the major applications of jumbled pattern matching. In standard alignment of two strings, each character of one string is matched against every single character of other string. However, in composition alignment matching the substrings that have the same characters are matched against the substring of the other string, even though the order of characters in the string can be different. It is easy to identify the subsequences that contain substrings of similar compositions. In other words, composition alignment is referred as pairing of substrings of exactly matching composition separated by insertions, deletions, or mismatches.

### 2.3 Mass Spectrometery and SNP Discovery

Jumbled pattern matching can also be used in the field of interpretation of mass spectrometer [2]. It is used to find the strings which have the same spectra. Mass spectra are simulated for every potential sequence and the resulting simulated spectra are then compared against the measured mass spectrum.

A single nucleotide polymorphism (SNP) is a variation at a single position in a DNA sequence among individuals. Each SNP represents a difference of a nucleotide in a single DNA. SNPs occur normally throughout a person's DNA. SNPs are believed to contribute strongly to the genetic variability in living beings. A comparatively new method to discover such polymorphisms is based on base-specific cleavage, where resulting cleavage products are analyzed by mass spectrometry. Simulating the mass spectrum that results from a base-specific cleavage experiment is relatively simple [3] and can be compared with simulating the mass spectrum of a protein.

## 3 SIMD

SIMD [22] is a type of parallel architecture that allows one instruction to be operated at the same time on multiple data items. Initially, SIMD has been used in multimedia especially in processing images or audio files. SIMD instructions have also found applications in many other areas like crytography. A detailed review of SIMD and its applications is given in [18,20]. Recently, SIMD instructions have also been applied to string matching [27,28].

SSE (Streaming SIMD Extensions) [24] comprise of SIMD instruction sets supported by modern processors which are capable of parallel execution of operations on multiple data simultaneously through a set of special instructions that work on limited number of special registers. SIMD instructions use sixteen 128-bit registers known as XMM0, XMM1, ..., XMM15. In our algorithms, we use specialized string matching SIMD instructions in addition to standard SIMD instructions.

The SIMD architecture comprises of several aggregation operations that can be applied on strings to process them simultaneously. Some of the aggregation operations that can be used in string processing are equal each, equal any, and ranges. In our approach, we have applied the equal any operation to speed up reading the text. The operation has two input operands which are strings of up to 16 characters. The first string represents a multiset of characters. The second string is the text itself. The output of the operation is a bitvector of 16 bits, where 1 means that the corresponding character in the string belongs to the set and 0 means the opposite.

For instance, let us consider a string *abbc* representing the multiset $\{a, b, b, c\}$ and a string *adcdbeaeebefdbce*. The output of the equal any operation is 1010101001000110 in the reverse order.

We have used the following SIMD instructions in our algorithms. The instruction *simd-load* is formally

$$__m128i_mm_loadu_si128(__m128i\ \ const*\ mem_addr).$$

This load instruction for SSE memory operations loads 16 bytes from the address to an SIMD register of the memory location mentioned as a parameter. The instruction, *simd-equal-any*$(x, y)$ is formally

$$_mm_extract_epi16(_mm_cmpistrm(x, y, _\text{SIDD_CMP_EQUAL_ANY}), 0).$$

The inner instruction has three parameters. The first and second parameters are string fragments with a maximum size of 16 bytes. The third parameter is a constant determining the type of comparison to be performed and the format of value to be returned. In our case, the third parameter is _SIDD_CMP_EQUAL_ANY. The instruction *_mm_extract_epi*16 extracts a selected signed or unsigned 16-bit integer from the first of its parameters.

The instruction *simd-cmpeq*$(x, y)$ is formally

$$_mm_movemask_epi8(__m128i_mm_cmpeq_epi8(__m128i\ x, __m128i\ y)).$$

The instruction *_mm_movemask_epi*8(*__m*128*i z*) creates a mask from the most significant bit of each 8-bit element in the parameter $z$ and stores the result. The instruction *__m*128*i_mm_cmpeq_epi*8(*__m*128*i x*, *__m*128*i y*) compares packed 8-bit integers in $x$ and $y$ bytewise for equality and stores the result.

The performance of SIMD instructions depends on the architecture of the processor. The performance of a single instruction is measured by latency and throughput. Latency is the number of cycles taken by the processor to give the desired outcome form the given input. Throughput refers to the number of cycles between subsequent calls of the same instruction. We used processors Intel i7-860 and i5-4250U in our experiments. Their microarchitectures are Nehalem and Haswell [21], respectively. The latency and throughput of the SIMD instructions used in our algorithms for these processors are given in Table 1. One should observe that string matching instructions are slower than ordinary SIMD instructions. For other processors the difference may be still larger. Therefore it is crucial which SIMD instructions an algorithm designer selects for his code.

| Architecture | SIMD instruction | Latency | Throughput |
|---|---|---|---|
| Nehalem | *_mm_cmpistrm* | 8 | 2 |
| | *_mm_extract_epi*16 | 3 | 1 |
| | *_mm_cmpeq_epi*8 | 1 | 0.5 |
| | *_mm_movemask_epi*8 | 1 | 1 |
| Haswell | *_mm_cmpistrm* | 11 | 3 |
| | *_mm_extract_epi*16 | 3 | 1 |
| | *_mm_cmpeq_epi*8 | 1 | 0.5 |
| | *_mm_movemask_epi*8 | 3 | 1 |

**Table 1.** Latency and throughput of SIMD instructions for Nehalem and Haswelll [23].

## 4 Earlier Solutions to Jumbled Matching

In this section, we present some earlier solutions for jumbled pattern matching. In recent past, many algorithms have been presented in this area. Grossi & Luccio's and Navarro's solutions [17,25,29] are based on the counts of characters occurring in the pattern and in an alignment window. These methods solve this problem in linear time. Navarro's counting algorithm is based on a sliding window approach.

Ejaz [13] proposed several algorithms for jumbled pattern matching. One of them utilizes backward scanning of the alignment window. Moreover, Burcsi et al. [6] introduced a light indexing approach with linear construction time and with sublinear expected query time.

According to the tests by Chhabra et al. [9], BAM, BAM2, and EBL are the fastest algorithms of jumbled matching for short English patterns. We use them as reference methods. Below we explain their main ideas.

**BAM.** The BAM (Bit-parallel Abelian Matcher) algorithm was presented by Cantone and Faro [8]. The algorithm applies backward scanning of the alignment window using bit-parallelism. The central idea of this algorithm is to assign a counter to each distinct character of the pattern and to allocate a bit field of a word for each counter. In addition to that, a common one bit counter is reserved for the remaining characters of the alphabet which do not occur in the pattern.

**BAM2.** Chhabra et al. [9] presented BAM2. BAM2 is a variation of BAM that handles a 2-gram at a time. It processes the whole alignment window with 2-grams (except the leftmost character in the case of odd $m$). This is beneficial because the alignment window is scanned on average further to the left in jumbled matching than in ordinary string matching. Moreover, 2-grams instead of single characters are read in our implementation of BAM2.

**EBL.** EBL (Exact Backward for Large alphabets) presented by Chhabra et al. [9] is based on the SBNDM2 algorithm [12]. EBL works on a sliding window approach. Characters in the window are read from right to left. EBL shifts the text window from left towards right whenever a mismatch occurs. SBNDM2 is a sublinear bit-parallel algorithm for exact string matching. In EBL, an array $B$ corresponding to the incidence vector of SBNDM2 states if the character $c$ is present in the pattern. $B[c]$ is assigned value 1 if $c$ is present, otherwise $B[c]$ is 0. As in SBNDM2, two characters are read before the first test in an alignment window.

## 5 Equal Any Approach

In Sections 5 and 6 we present two new algorithms. Both solutions apply SIMD instructions in order to filter out a significant portion of text. The first algorithm utilizes the equal any SIMD command.

Let us assume that $m < 16$ holds. The width of a test window in the text is 16. The equal any SIMD command returns a bitvector $k$ of 16 bits showing the positions in the test window which hold any character of the pattern. For example, if the test window is *this is a sample* and the pattern *aeiou*, the vector is:

```
elpmas a si siht
1000100100100100
```

Note that the orientation of the bitvector is the opposite of the text. A match candidate is found if the last $m$ bits of the vector are ones. Note that such a case is only a match candidate because the counts of characters are not analyzed. For example, the string *aaaaa* is a match candidate for *abcde*.

**Algorithm 1** **EA**$(P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1})$

```
 1: Place a copy of P after T
 2: Call PP(m)
 3: occ ← 0; x ← simd-load(P)
 4: shift ← 1; i ← 0
 5: while true do
 6:     while shift > 0 do
 7:         y ← simd-load(t_i ··· t_{i+15})
 8:         k ← simd-equal-any(x, y)
 9:         shift ← d[k]
10:         i ← i + shift
11:     occ ← occ + verify(t_i ··· t_{i+m-1})
12:     if i = n then
13:         return occ − 1
14:     i ← i + 1
```

Alg. 1 is the pseudocode of the scanning algorithm EA based on the equal any approach. In EA, we use a table $d$ for shifting the test window. The algorithm applies a skip loop with a stopper, which is a copy of the pattern. A heuristic algorithm called PP to compute $d$ from $m$ is given as Alg. 2. When a block of $m$ ones is found in EA, the window is shifted so that the block is at the right end of the bitvector corresponding to the test window. When a block of $m$ ones is at the right end, a match candidate is found. The entry of $d$ is zero in such a case in order to get out from the skip loop.

Let us study more details of EA. The SIMD register $x$ holds the pattern and the SIMD register $y$ holds a test window of 16 bytes of the text. The registers $x$ and $y$ are processed with the *simd-equal-any* operation (see Sect. 3) resulting a 16-bit integer $k$ on line 8. If $d[k] = 0$ holds, a match candidate is found and the inner loop is exited. On line 11 there is a call of a verification routine which verifies the match candidate. Any previous algorithm for jumbled pattern matching (especially BAM, BAM2, or EBL) can be used as a verification method. The variable *occ* holds the count of matches. The stopper creates a superfluous match which is subtracted from the count on line 13.

Let us consider how the shift table $d$ is computed in PP. The table $d$ is indexed with the 16-bit integer $k$. The computation consists of several subsequent for-loops which are in the decreasing order of shift. This order of computation is essential, because a single entry of $d$ may be assigned several times.

**Algorithm 2 PP**$(m)$

```
1: L ← 2^16 − 1; b ← 1 << 15
2: for i ← 0 to b − 1 do
3:     d[i] ← 16
4: a ← b; b ← 3 << 14
5: for x ← 15 downto 16 − m do
6:     for i ← a to b − 1 do
7:         d[i] ← x
8:     a ← b; b ← b + (1 << (x − 2))
9: s ← (1 << m) − 1; a ← s << (15 − m)
10: b ← 1 << 15; c ← 1 << (14 − m)
11: for x ← 15 − m downto 2 do
12:     for i ← a step b to L do
13:         for j ← 0 to c − 1 do
14:             d[i + j] ← x
15:     a ← a >> 1; b ← b >> 1; c ← c >> 1
16: for i ← a step b to L do
17:     d[i] ← 1
18: for i ← s step s + 1 to L do
19:     d[i] ← 0
```

Let us go through the phases of Alg. PP in detail. Let the 16 bits of $k$ be named by $k_{15}, k_{14}, \ldots, k_0$. In the beginning, the integer $b$ corresponds to a bitvector of one followed by 15 zeros. When the leftmost bit $k_{15}$ is zero, the test window can be shifted 16 positions in the best case (lines 1–3).

On lines 4–8 we consider the case where $k$ starts with $16 - x$ ones followed by a zero for $x = 15, 14, \ldots, 16 - m$. Then the shift is $x$. For example, the shift for 1111010101001100 is 12 for $m > 4$.

On lines 9–15 we consider the case where $k$ holds a block of $m$ ones starting from $k_{14}, k_{13}, \ldots$ or $k_{m+1}$. The loop for $x$ traverses all the possible locations of the block of $m$ ones. The loop for $i$ traverses all bit combinations of the first $16 - x - m$ bits. The loop for $j$ traverses all bit combinations of the last $x - 1$ bits. For example, in the computation of $d[0011111101000100]$ for $m = 6$, $a$ is 0011111100000000, $b$ is 0100000000000000, $c$ is 0000000010000000, and $x$ is 8.

When the block of $m$ ones ends at $k_1$, the shift is one and it is computed in a single loop (lines 16–17). When the block of $m$ ones ends at $k_0$, a match candidate is found, and this is expressed as assigning a zero to all such entries (lines 18–19)

## 6 Least Frequent Character Approach

Our second approach was developed for natural language. We use SIMD instructions to analyze whether a test window of 16 bytes holds the least frequent character of the pattern. The frequency of characters is based on the text or on the language.

Alg. 3 is the pseudocode of the scanning algorithm LF based on the least frequent character approach. On line 11 there is a call of a search routine which searches a block of up to $2m + 14$ characters. Any previous algorithm for jumbled pattern matching (especially BAM, BAM2, or EBL) can be used as a search method. The parameter $R$ is an array containing 16 bytes, each of which holding the least frequent character. The SIMD register $x$ holds $R$ and the SIMD register $y$ holds a test window of 16 bytes of the text. The registers $x$ and $y$ are compared by the *simd-cmpeq* operation (see Sect. 3) on line 6. The algorithm applies a skip loop with a stopper, which is a

**Algorithm 3** **LF**($P = p_0 p_1 \cdots p_{m-1}, T = t_0 t_1 \cdots t_{n-1}, R$)

```
 1: Place a copy of P after T
 2: occ ← 0; i ← 0; f ← 0
 3: x ← simd-load(R)
 4: while true do
 5:     y ← simd-load(t_i ··· t_{i+15})
 6:     while simd-cmpeq(x, y) = 0 do
 7:         i ← i + 16
 8:         y ← simd-load(t_i ··· t_{i+15})
 9:     if f < i − m + 1 then
10:         f ← i − m + 1
11:     occ = occ + search(t_f ··· t_{i+15+m−1})
12:     f ← i + 16
13:     if f ≥ n then
14:         return occ − 1
15:     else
16:         i ← i + 16
```

copy of the pattern. As in Alg. EA, the stopper creates a superfluous match which is subtracted from the count on line 14.

We use an additional variable $f$ to control the starting position of a block. If the previous block has been skipped then the leftmost possible starting position for a match is $i + m - 1$. Otherwise the leftmost possible starting position for a match is $i$. Without such control, we would get a wrong number of matches, because there could be matches which would belong to two blocks.

## 7 Experiments

The tests were run on Intel 2.70 GHz i7-860 Nehalem processor with 16 GB of memory. All the algorithms were implemented in the C programming language and run in the 64-bit mode in the testing framework of Hume and Sunday [19]. We used two types of data for testing the algorithms. The protein text is 3 MB long and English text (KJV Bible) is 4 MB long. Both the texts were taken from the Smart corpus [15]. From both the texts, we picked six sets of 200 patterns with lengths $m = 4, 5, \ldots, 10$.

We tested the EA and LF algorithm schemes with BAM, BAM2, and EBL as the checking subroutine, i.e. six new algorithms. From these we selected LF-BAM2, EA-BAM2, and EA-EBL for further consideration. BAM, BAM2, and EBL were our reference methods. The running time of the PP algorithm for the EA scheme was about 10 ms.

In our tests we applied BAM2 without the bin sharing technique [9]. For patterns having at most 10 characters we do not require to use shared bins in the 64-bit architecture. One bit is reserved for characters which are not present in the pattern and five bits are enough for each bin.

Tables 2 and 3 present the average execution times in seconds for English and protein data, respectively. The results were retrieved as an average of ninety nine runs. The best execution times are highlighted by placing them in a box.

In Table 2 the execution time of the best one of the new algorithms is 17–33 percent less than the best time for earlier algorithms for $4 \leq m \leq 9$. EA-BAM2 is fastest for short patterns of length 4 and 5. LF-BAM2 performs best for the remaining pattern lengths except 10, where BAM2 has best execution time.

| m | BAM | BAM2 | EBL | LF-BAM2 | EA-BAM2 | EA-EBL |
|---|---|---|---|---|---|---|
| 4 | 1.1918 | 1.4221 | 0.7274 | 0.6119 | 0.4995 | 0.5344 |
| 5 | 1.1942 | 0.7561 | 0.7364 | 0.5732 | 0.4903 | 0.5603 |
| 6 | 1.1037 | 0.5473 | 0.6891 | 0.4515 | 0.4859 | 0.5321 |
| 7 | 1.0116 | 0.4207 | 0.6502 | 0.3611 | 0.4809 | 0.5114 |
| 8 | 0.9521 | 0.3711 | 0.6267 | 0.3431 | 0.4761 | 0.5073 |
| 9 | 0.9035 | 0.3217 | 0.6257 | 0.2686 | 0.4791 | 0.5031 |
| 10 | 0.8671 | 0.3005 | 0.6345 | 0.3133 | 0.4803 | 0.4945 |

**Table 2.** Execution times of algorithms (in seconds) for English data on Nehalem.

| m | BAM | BAM2 | EBL | LF-BAM2 | EA-BAM2 | EA-EBL |
|---|---|---|---|---|---|---|
| 4 | 0.6772 | 0. 9947 | 0.4573 | 0.5689 | 0.3307 | 0.3431 |
| 5 | 0.6913 | 0.5511 | 0.4245 | 0.5262 | 0.3203 | 0.3119 |
| 6 | 0.6567 | 0.4255 | 0. 3915 | 0.4629 | 0.3214 | 0.3221 |
| 7 | 0.5942 | 0.3167 | 0.3718 | 0.4065 | 0.3203 | 0.3341 |
| 8 | 0.5732 | 0.2545 | 0.3511 | 0.4115 | 0.3315 | 0.3472 |
| 9 | 0.5512 | 0.2025 | 0.3614 | 0.3405 | 0.3411 | 0.3512 |
| 10 | 0.5441 | 0.1798 | 0.4013 | 0.3792 | 0.3497 | 0.3623 |

**Table 3.** Execution times of algorithms (in seconds) for Protein data on Nehalem.

The results in Table 3 for protein data show that the EA algorithm scheme is competitive in comparison with the previous algorithms. The EA scheme works best in case of short length patterns of length less than 7. EA-BAM2 is fastest for pattern lengths 4 and 6. EA-EBL performs best for $m = 5$. For the remaining pattern lengths BAM2 performs best. The LF algorithm scheme was not competitive for protein data.

| m | BAM | BAM2 | EBL | LF-BAM2 | EA-BAM2 | EA-EBL |
|---|---|---|---|---|---|---|
| 4 | 1.8640 | 2.3047 | 1.0800 | 0.7608 | 0.8158 | 0.8468 |
| 5 | 1.8643 | 1.1992 | 0.9686 | 0.7261 | 0.8082 | 0.9297 |
| 6 | 1.7281 | 0.7698 | 0.9083 | 0.5581 | 0.8082 | 0.9297 |
| 7 | 1.5944 | 0.6541 | 0.8801 | 0.4401 | 0.7945 | 0.9921 |
| 8 | 1.3017 | 0.3906 | 0.7258 | 0.3921 | 0.7192 | 0.8114 |
| 9 | 1.2395 | 0.3886 | 0.7490 | 0.2929 | 0.7294 | 0.8555 |
| 10 | 1.1960 | 0.3356 | 0.7636 | 0.3552 | 0.7247 | 0.8994 |

**Table 4.** Execution times of algorithms (in seconds) for English data on Haswell.

We also performed the same tests on an Haswell processor (i5-4250U) for English and protein data. The results are shown in Tables 4 and 5. In Table 4 the algorithm LF-BAM2 is a clear winner for all the pattern lengths except for $m = 8$ and 10. For certain pattern lengths such as $m = 4, 5, 6$, the speed up is more than twenty percent. For protein data (Table 5), EBL and BAM2 are the winners. However, LF-BAM2 is better than BAM2 for $m = 4, 5$ and better than EBL for $m = 6, \ldots, 10$.

The Haswell processor has the AVX2 support, which enables 32-byte SIMD computation. We compared the 32-byte version of LF-BAM2 with the 16-byte version for pattern lengths $m = 4, 5, \ldots, 16$. In every case, the 16-byte version was slightly faster.

| m | BAM | BAM2 | EBL | LF-BAM2 | EA-BAM2 | EA-EBL |
|---|---|---|---|---|---|---|
| 4 | 0.8944 | 1.0338 | 0.4533 | 0.4776 | 0.4978 | 0.4912 |
| 5 | 0.8112 | 0.6251 | 0.4096 | 0.4519 | 0.4872 | 0.4880 |
| 6 | 0.7776 | 0.3541 | 0.4067 | 0.3870 | 0.4929 | 0.5227 |
| 7 | 0.7430 | 0.3230 | 0.4076 | 0.3467 | 0.5000 | 0.5583 |
| 8 | 0.7228 | 0.2255 | 0.4078 | 0.3549 | 0.5036 | 0.5676 |
| 9 | 0.7066 | 0.2271 | 0.4234 | 0.3103 | 0.5096 | 0.5889 |
| 10 | 0.6952 | 0.1809 | 0.4540 | 0.3478 | 0.5153 | 0.6388 |

**Table 5.** Execution times of algorithms (in seconds) for Protein data on Haswell.

## 8 Concluding Remarks

We introduced improved solutions for exact jumbled pattern matching based on the SIMD architecture. These algorithms are an outcome of a long series of experimentation. We developed and tested also some other algorithms using SIMD instructions but only the best are shown in this paper. Especially, it was hard to develop fast jumbled matching algorithms for $m > 16$. It should be realized that if the latency of the used SIMD instructions would improve in future processors, the running times of the algorithms will respectively change.

## References

1. G. Benson: Composition alignment. In Proceedings of The 3rd International Workshop on Algorithms in Bioinformatics 2003, pp. 447–461.
2. S. Böcker: Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. Journal of Computational Biology 11(6), 2004, pp. 1110–1134.
3. S. Böcker: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics 23(2), 2007, pp. 5–12.
4. S. Böcker, K. Jahn, J. Mixtacki, J. Stoye: Computation of median gene clusters. Journal of Computational Biology 16(8), 2009, pp. 1085-1099.
5. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: On table arrangement, scrabble freaks, and jumbled pattern matching. In Proceedings of the Symposium on Fun with Algorithms 2010, pp. 89–101.
6. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. 23(2), 2012, pp. 357–374.
7. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: On approximate jumbled pattern matching in strings. Theory Comput. Syst. 50(1), 2012, pp. 35–51.
8. D. Cantone, S. Faro: Efficient online Abelian pattern matching in strings by simulating reactive multi-automata. In Proceedings of Prague Stringology Conference 2014, pp. 30–42.
9. T. Chhabra, S.S. Ghuman, J. Tarhio: Tuning algorithms for jumbled matching. In Proceedings of Prague Stringology Conference 2015, pp. 57–66.
10. F. Cicalese, G. Fici, Zs. Lipták: Searching for jumbled patterns in strings. In Proceedings of Prague Stringology Conference 2009, pp. 105–117.
11. E. Domann, T. Hain, R. Ghai, A. Billion, C. Kuenne, K. Zimmermann, T. Chakraborty: Comparative genomic analysis for the presence of potential enterococcal virulence factors in the probiotic enterococcus faecalis strain symbioflor. International Journal of Medical Microbiology 297(7), 2007, pp. 533–539.
12. B. Ďurian, J. Holub, H. Peltola, J. Tarhio: Improving practical exact string matching. Information Processing Letters 110(4), 2010, pp. 148–152.
13. E. Ejaz: Abelian Pattern Matching in Strings. Ph.D. Thesis, Dortmund University of Technology(2010), http://d-nb.info/1007019956.
14. R. Eres, G. M. Landau, L. Parida: Permutation pattern discovery in biosequences. Journal of Computational Biology 11(6), 2004, pp. 1050–1060.

15. S. Faro, T. Leqroc: Smart: String matching algorithms research tool (2015), http://www.dmi.unict.it/~faro/smart/.
16. S. Grabowski, S. Faro, E: Giaquinta: String matching with inversions and translocations in linear average time (most of the time). Information Processing Letters 111(11), 2011, pp. 516–520.
17. R. Grossi, F. Luccio: Simple and efficient string matching with k mismatches. Information Processing Letters 33(3), 1989, pp. 113–120.
18. M. Hassaballah, S. Omran, Y.B. Mahdy: A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput. J. 51(6), 2008, pp. 630–649.
19. A. Hume, D. Sunday: Fast string searching. Software – Practice and Experience 21(11), 1991, pp. 1221–1248.
20. K. Hwang, F.A. Briggs: Computer Architecture and Parallel Processing, McGraw-Hill, 1984.
21. Intel: http://ark.intel.com/products/codename/29896/Lynnfield.
22. Intel: Intel (R) 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Loaded in Jan. 2016).
23. Intel: Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.
24. H. Jeong, S. Kim, W. Lee, S.H. Myung: Performance of SSE and AVX instruction sets. CoRR abs/1211.0820 (2012).
25. P. Jokinen, J. Tarhio, E. Ukkonen: A comparison of approximate string matching algorithms. Software – Practice and Experience 26(12), 1996, pp. 1439–1458.
26. S. Karlin: Detecting anomalous gene clusters and pathogenicity islands in diverse bacterial genomes. Trends in Microbiology 9(7), 2001, pp. 335–343.
27. M.O. Külekci: Filter based fast matching of long patterns by using SIMD instructions. In Proceedings of Prague Stringology Conference 2009, pp. 118–128.
28. S. Ladra, O. Pedreira, J. Duato, N.R. Brisaboa: Exploiting SIMD instructions in current processors to improve classical string algorithms. In Proceedings of The 16th East European Conference on Advances in Databases and Information Systems, vol. 7503 of LNCS, Springer, 2012, pp. 254–267.
29. G. Navarro: Multiple approximate string matching by counting. In Proceedings of 4th South American Workshop on String Processing 1997, pp. 95–111.
30. G. Navarro, M. Raffinot: Flexible Pattern Matching in Strings. Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, New York 2002.
31. R. Overbeek, M. Fonstein, M. D'Souza, G.D. Pusch, N. Maltsev: The use of gene clusters to infer functional coupling. In Proceedings of the National Academy of Sciences 96(6), 1999, pp. 2896–2901.
32. H. Peltola, J. Tarhio: Alternative algorithms for bit-parallel string matching. In Proceedings of SPIRE 2003, the 10th International Symposium on String Processing and Information Retrieval, vol. 2857 of LNCS, Springer, 2003, pp. 80–94.
33. A. Salomaa: Counting (scattered) subwords. Bulletin of the European Association for Theoretical Computer Science 81, 2003, pp. 165–179.
34. T. Schmidt, J. Stoye: 2004. Quadratic time algorithms for finding common intervals in two and more sequences. In Proceedings of 15th Annual Symposium of Combinatorial Pattern Matching, vol. 3109 of LNCS, Springer, 2004, pp. 347–358.
35. T. Uno, M. Yagiura: Fast algorithms to enumerate all common intervals of two permutations. Algorithmica 26(2), 2000, pp. 290–309.
36. A. Wiezer, R. Merkl: A comparative categorization of gene flux in diverse microbial species. Genomics 86(4), 2005, pp. 462–475.

# Forced Repetitions over Alphabet Lists

Neerja Mhaskar[1] and Michael Soltys[2]

[1] McMaster University
Dept. of Computing & Software
1280 Main Street West
Hamilton, Ontario L8S 4K1, Canada
pophlin@mcmaster.ca

[2] California State University Channel Islands
Dept. of Computer Science
One University Drive
Camarillo, CA 93012, USA
michael.soltys@csuci.edu

**Abstract.** Thue [14] showed that there exist arbitrarily long square-free strings over an alphabet of three symbols (not true for two symbols). An open problem was posed in [7], which is a generalization of Thue's original result: given an alphabet list $L = L_1, \ldots, L_n$, where $|L_i| = 3$, is it always possible to find a square-free string, $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \cdots \boldsymbol{w}_n$, where $\boldsymbol{w}_i \in L_i$? In this paper we show that squares can be forced on square-free strings over alphabet lists iff a suffix of the square-free string conforms to a pattern which we term as an offending suffix. We also prove properties of offending suffixes. However, the problem remains tantalizingly open.

**Keywords:** strings, square-free, repetition, Thue morphisms

## 1 Introduction

The study of repetitions in words is an attractive field for both theoretical and applied research. It dates back to the early 20th century and the seminal work of Axel Thue [14,15], who proved the existence of square-free strings over an alphabet of three letters, using iterated morphism. Since his work was not known for a long time, this result was rediscovered by many others independently. For example, the following authors each gave different morphisms to show the existence of a square-free string over a ternary alphabet: [1,10,6,8].

Many different morphisms have been proposed besides Thue's original one. But all these morphisms construct a string over a fixed finite alphabet. A natural generalization of the problem is to allow a (possibly) infinite alphabet of symbols, but to restrict the $i$-th symbol of the word to come from a particular subset. Thus, we are interested in constructing an arbitrarily long square-free string with constraints imposed on the positions. This generalization has been studied by [7,12,9], among others.

The authors of [7] showed that as long as each position is required to be filled with a symbol from a subset of size at least four, then we can always construct a square-free string over such a list (the $i$-th symbol of this string comes from the $i$-th alphabet in the list, and each alphabet is of size at least four). However, the question whether the same holds if the alphabets are restricted to be of size three is still an open question. Note that Thue's original result applies to only a particular case of the problem where all the alphabets in the list are of size three, *and* contain the same elements, but for the generalized case the problem remains open. Also note that if all

Neerja Mhaskar, Michael Soltys: Forced Repetitions over Alphabet Lists, pp. 125–134.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

subsets are $\{a, b\}$, we cannot construct an arbitrarily long square-free string. In this paper, we show that squares can be forced on certain type of square-free strings over an alphabet list and we give a characterization for such strings.

The outline of the paper is as follows: in Section 2, we give a brief introduction to the terminology. In Section 3, we define a pattern "Offending Suffix," and show that having strings over alphabet lists with a suffix conforming to this pattern is a necessary and sufficient condition to force squares: Theorem 1. In section 4, we give a characterization of square-free strings using borders. In Section 5, we show using rudimentary Kolmogorov complexity that given any alphabet list, there always exists a string $\boldsymbol{w}$ without squares longer than $\frac{1}{5}|\boldsymbol{w}|$. In other words, we are able to eliminate a few huge squares in every case.

## 2 Background

An *alphabet* is a set of symbols, and $\Sigma$ is usually used to represent a finite alphabet. The elements of an alphabet are referred to as *symbols* (or *letters*). In this paper, we assume $|\Sigma| \neq 0$. A *string* (or a *word*) over $\Sigma$, is an ordered sequence of symbols from it. Formally, $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \cdots \boldsymbol{w}_n$, where for each $i$, $\boldsymbol{w}_i \in \Sigma$, is a string. In order to emphasize the array structure of $\boldsymbol{w}$, we sometimes represent it as $\boldsymbol{w}[1..n]$. The *length* of a string $\boldsymbol{w}$ is denoted by $|\boldsymbol{w}|$. The set of all finite length strings over $\Sigma$ is denoted by $\Sigma^*$. The empty string is denoted by $\varepsilon$, and it is the string of length zero. The set of all finite strings over $\Sigma$ not containing $\varepsilon$ is denoted by $\Sigma^+$. We denote $\Sigma_k$ to be a fixed generic alphabet of $k$ symbols, and $\Sigma^{\geq j}$ to be the set of all strings over $\Sigma$ of size at least $j$.

A string $\boldsymbol{v}$ is a *subword* (also known as a *substring* or a *factor*) of $\boldsymbol{w}$, if $\boldsymbol{v} = \boldsymbol{w}_i\boldsymbol{w}_{i+1} \cdots \boldsymbol{w}_j$, where $i \leq j$. If $i = 1$, then $\boldsymbol{v}$ is a *prefix* of $\boldsymbol{w}$ and if $j < n$, $\boldsymbol{v}$ is a *proper prefix* of $\boldsymbol{w}$. If $j = n$, then $\boldsymbol{v}$ is a *suffix* of $\boldsymbol{w}$ and if $i > 1$, then $\boldsymbol{v}$ is a *proper suffix* of $\boldsymbol{w}$. We can express that $\boldsymbol{v}$ is a subword more succinctly using array representation as $\boldsymbol{v} = \boldsymbol{w}[i..j]$. A word $\boldsymbol{v}$ is a *subsequence* of a string $\boldsymbol{w}$ if the symbols of $\boldsymbol{v}$ appear in the same order in $\boldsymbol{w}$. Note that the symbols of $\boldsymbol{v}$ do not necessarily appear contiguously in $\boldsymbol{w}$. Hence, any subword is a subsequence, but the reverse is not true.

A string $\boldsymbol{w}$ is said to have a *repetition* if there exists a subword of $\boldsymbol{w}$ consisting of consecutive repeating factors. The most basic type of repetition is a square and we define it as follows: a string $\boldsymbol{w}$ is said to have a *square* if there exists a string $\boldsymbol{v}$ such that $\boldsymbol{vv}$ is a subword of $\boldsymbol{w}$ and it is *square-free* if no such subword exists. A map $h : \Sigma^* \rightarrow \Delta^*$, where $\Sigma$ and $\Delta$ are finite alphabets, is called a *morphism* if for all $x, y \in \Sigma^*$, $h(\boldsymbol{xy}) = h(\boldsymbol{x})h(\boldsymbol{y})$. A morphism is said to be *non-erasing* if for all $\boldsymbol{w} \in \Sigma^*$, $h(\boldsymbol{w}) \geq \boldsymbol{w}$. It is called square-free if $h(\boldsymbol{w})$ is square-free for every square-free word $\boldsymbol{w}$ over $\Sigma$.

An *alphabet list* is an ordered list of finite subsets (alphabets), and in our case all the alphabets have the same cardinality. However for the general case we do not need to impose this condition on alphabet lists. Let $L = L_1, L_2, \ldots, L_n$ , be an ordered list of alphabets. A string $\boldsymbol{w}$ is said to be a word over the list $L$, if $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \cdots \boldsymbol{w}_n$ where for all $i$, $\boldsymbol{w}_i \in L_i$. Note that there are no conditions imposed on the alphabets $L_i$'s: they may be equal, disjoint, or have elements in common. The only condition on $\boldsymbol{w}$ is that the $i$-th symbol of $\boldsymbol{w}$ must be selected from the $i$-th alphabet of $L$, i.e., $\boldsymbol{w}_i \in L_i$. The alphabet set for the list $L = L_1, L_2, \ldots, L_n$ is denoted by $\Sigma_L = L_1 \cup L_2 \cup \cdots \cup L_n$. Given a list $L$ of finite alphabets, we can define the set of strings $\boldsymbol{w}$ over $L$ with a regular expression as follows: $R_L := L_1 \cdot L_2 \cdots L_n$. Let $L^+ := L(R_L)$ be the language

of all the strings over the list $L$. For example, if $L_0 = \{\{a,b,c\},\{c,d,e\},\{a,1,2\}\}$, then

$$R_{L_0} := \{a,b,c\} \cdot \{c,d,e\} \cdot \{a,1,2\},$$

and $ac1 \in L_0^+$, but $2ca \notin L_0^+$. Also, in this case $|L_0^+| = 3^3 = 27$.

Given a square-free string $\boldsymbol{w}$ over a list $L = L_1, L_2, \ldots, L_n$, we say that the alphabet $L_{n+1}$ *forces a square on* $\boldsymbol{w}$ if for all $a \in L_{n+1}$, $\boldsymbol{w}a$ has a square. Note that, this is not to be confused with forcing a square in $\boldsymbol{w}$. For example, if $L = \{a,b,c\}^7$, and $\boldsymbol{w} = abacaba$, then the alphabet $\{a,b,c\}$ forces a square on $\boldsymbol{w}$, as the strings $\boldsymbol{w}a$, $\boldsymbol{w}b$ and $\boldsymbol{w}c$ all have squares.

We introduce the concept of admissibility of lists. We say that an alphabet list $L$ is *admissible* if $L^+$ contains a square-free string. For example, the alphabet list $L = \{\{a,b,c\},\{1,2,3\},\{a,c,2\},\{b,3,c\}\}$, is admissible as the string '$a1c3$' over $L$ is square-free.

Let $\mathcal{L}$ represent a *class* of lists; the intention is for $\mathcal{L}$ to denote lists with a given property. For example, we are going to use $\mathcal{L}_{\Sigma_k}$ to denote the class of lists $L = L_1, L_2, \ldots L_n$, where for each $i$, $L_i = \Sigma_k$, and $\mathcal{L}_k$ will denote the class of all lists $L = L_1, L_2, \ldots, L_n$, where for each $i$, $|L_i| = k$, that is, those lists consisting of alphabets of size $k$. Note that $\mathcal{L}_{\Sigma_k} \subseteq \mathcal{L}_k$. We say that a class of lists $\mathcal{L}$ is admissible if *every* list $L \in \mathcal{L}$ is admissible. An example of admissible class of lists is the class $\mathcal{L}_{\Sigma_3}$ (Thue's result), and $\mathcal{L}_3$ is a class of lists whose admissibility status is unknown, and the subject of investigation in this paper.

A *border* $\beta$ of a string $\boldsymbol{w}$, is a subword that is both a proper prefix and proper suffix of $\boldsymbol{w}$. Note that the proper prefix and proper suffix may overlap. A string can have many borders. The empty string $\varepsilon$ is a border of every string. For example, the string $\boldsymbol{w} = 121324121$ has three borders 1, 121 and the empty string $\varepsilon$. See [13] for many properties of borders.

Given an alphabet $\Sigma$, let $\Delta = \{\boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{X}_3, \ldots, a_1, a_2, a_3, \ldots\}$ be variables, where the $\boldsymbol{X}_i$'s range over $\Sigma^*$, and the $a_i$'s range over $\Sigma$. A *pattern* is a non empty string over $\Delta^*$; for example, $\mathcal{P} = \boldsymbol{X}_1 a_1 \boldsymbol{X}_1$ is a pattern representing all strings where the first half equals the second half, and the two halves are separated by a single symbol. Intuitively, patterns are "templates" for strings. Note that some authors define patterns as being words over variables with no restriction on the size of the variables (see [4]), but we find the definition given here as more amenable to our purpose.

We say that a word $\boldsymbol{w}$ over some alphabet $\Sigma$ *conforms* to a pattern $\mathcal{P}$ if there is a morphism $h : \Delta^* \longrightarrow \Sigma^*$, such that $h(\mathcal{P}) = \boldsymbol{w}$.

We say that a pattern is *avoidable*, if strings of arbitrary length exist, such that no subword of the string conforms to the pattern, otherwise it is said to be *unavoidable*. For example, the pattern $\boldsymbol{XX}$ is unavoidable for all strings in $\Sigma_2^{\geq 4}$, but there exist strings in $\Sigma_3$ of arbitrary length for which it is avoidable (Thue's result, [14]).

The idea of unavoidable patterns was developed independently in [2] and [16]. *Zimin words* (also known as *sesquipowers*) constitute a certain class of unavoidable patterns. The $n$-th Zimin word, $\mathcal{Z}_n$, is defined recursively over the alphabet $\Delta$ of variables of type string as follows:

$$\begin{aligned} \mathcal{Z}_1 &= \boldsymbol{X}_1, \text{and for } n > 1, \\ \mathcal{Z}_n &= \mathcal{Z}_{n-1} \boldsymbol{X}_n \mathcal{Z}_{n-1}. \end{aligned} \tag{1}$$

[16] showed that Zimin words are unavoidable for large classes of words. More precisely, for every $n$, there exists an $N$, so that for every word $\boldsymbol{w} \in \Sigma_n^{\geq N}$ there

exists a morphism $h$ so that $h(\mathcal{Z}_n)$ is a subword of $\boldsymbol{w}$. For instance, the pattern $\mathcal{Z}_3 = \boldsymbol{X}_1\boldsymbol{X}_2\boldsymbol{X}_1\boldsymbol{X}_3\boldsymbol{X}_1\boldsymbol{X}_2\boldsymbol{X}_1$ is unavoidable over $\Sigma_3$ for words of length at least 29, as can be checked with an exhaustive search. See [5] for bounds on Zimin word avoidance. For details on Zimin patterns, see [16,3,11,4,5].

## 3 Offending Suffix Pattern

In this section, we introduce a pattern that we call an "offending suffix", and we show in Theorem 1 that such suffixes characterize in a meaningful way strings over alphabet lists with squares. Let $\mathcal{C}(n)$, an *offending suffix*, be a pattern defined recursively:

$$\begin{aligned} \mathcal{C}(1) &= \boldsymbol{X}_1 a_1 \boldsymbol{X}_1, \text{ and for } n > 1,\\ \mathcal{C}(n) &= \boldsymbol{X}_n \mathcal{C}(n-1) a_n \boldsymbol{X}_n \mathcal{C}(n-1). \end{aligned} \tag{2}$$

To be more precise, given a morphism, $h : \Delta^* \to \Sigma^*$, we call $h(\{a_1, a_2, \ldots, a_n\}) \subseteq \Sigma$ the pivots of $h$. When all the variables in the set $\{\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n\}$ map to $\varepsilon$, we get the pattern for the shortest possible offending suffix for a list $L \in \mathcal{L}_n$. We call this pattern the *shortest offending suffix*, and employ the notation:

$$\mathcal{C}_s(n) = a_1 a_2 a_1 \cdots a_n \cdots a_1 a_2 a_1. \tag{3}$$

Note that $|\mathcal{C}_s(n)| = 2|\mathcal{C}_s(n-1)| + 1$, where $|\mathcal{C}_s(1)| = 1$, and so, $|\mathcal{C}_s(n)| = 2^n - 1$.

As we are interested in offending suffixes for $\mathcal{L}_3$, we consider mainly:

$$\begin{aligned} \mathcal{C}(3) &= \boldsymbol{X}_3\boldsymbol{X}_2\boldsymbol{X}_1 a_1 \boldsymbol{X}_1 a_2 \boldsymbol{X}_2\boldsymbol{X}_1 a_1 \boldsymbol{X}_1 a_3 \boldsymbol{X}_3\boldsymbol{X}_2\boldsymbol{X}_1 a_1 \boldsymbol{X}_1 a_2 \boldsymbol{X}_2\boldsymbol{X}_1 a_1 \boldsymbol{X}_1,\\ \mathcal{C}_s(3) &= a_1 a_2 a_1 a_3 a_1 a_2 a_1, \end{aligned} \tag{4}$$

and observe that $\mathcal{C}_s(3)a_i$, for $i = 1, 2, 3$, all map to strings with squares.

Pattern $\mathcal{C}$ in (2) bears great resemblance to Zimin words (1) discussed at the end of the previous section. Comparing (1) to (2), one can see that mapping $\boldsymbol{X}_i$ to $a_i$ in (1) yields the same string as mapping $\boldsymbol{X}_i$ to $\varepsilon$ in (2). In particular, the shortest offending suffix $\mathcal{C}_s(n)$ can be obtained from the Zimin word $\mathcal{Z}_n$ by mapping $X_i$'s to $a_i$'s. Despite the similarities, we prefer to introduce this new pattern, as the advantage of $\mathcal{C}(n)$ is that it allows for the succinct expression of the most general offending suffix possible.

Given a list $L$, let $h : \Delta^* \to \Sigma_L^*$, be a morphism. We say that $h$ *respects* a list $L = L_1, L_2, \ldots, L_n$, if $h$ yields a string over $L$. So, for example, an $h$ that maps each $\boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{X}_3$ to $\varepsilon$, and also maps $a_1 \mapsto a, a_2 \mapsto b, a_3 \mapsto c$, yields $h(\mathcal{C}(3)) = abacaba$. Such an $h$ respects, for example, a list $L = \{a, e\}, \{a, b\}, \{a, d\}, \{c\}, \{a, e\}, \{b, c, d\}, \{a\}$. In general, papers in the field of string algorithms mix variables over symbols with the symbols themselves, that is, $a$ may stand for both the symbol $a \in \Sigma$, and a variable that takes on values in $\Sigma$. In our case, we need to specify exactly what is a variable and what is a symbol.

The main result of the paper, a characterization of squares in strings over lists in terms of offending suffixes, follows.

**Theorem 1.** *Suppose that $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \cdots \boldsymbol{w}_{i-1}$ is a square-free string over a list $L = L_1, L_2, \ldots, L_{i-1}$, where $L \in \mathcal{L}_3$. Then, the pivots $L_i = \{a, b, c\}$ force a square on $\boldsymbol{w}$ iff $\boldsymbol{w}$ has a suffix conforming to the offending suffix $\mathcal{C}(3)$.*

*Proof.* The proof is by contradiction. We assume throughout that our lists are from the class $\mathcal{L}_3$.
($\Leftarrow$) Suppose $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2\cdots\boldsymbol{w}_{i-1}$ has a suffix conforming to the offending suffix $\mathcal{C}(3)$, where $a, b, c$ are the pivots. Clearly, if we let $L_i = \{a, b, c\}$, then each $\boldsymbol{w}a$, $\boldsymbol{w}b$, $\boldsymbol{w}c$ has a square, and hence by definition $L_i$ forces a square on $\boldsymbol{w}$.
($\Rightarrow$) Suppose, on the other hand, that $L_i = \{a, b, c\}$ forces a square on the word $\boldsymbol{w}$ over $L = L_1, L_2, \ldots, L_{i-1}$. We need to show that $\boldsymbol{w}$ must have a suffix that conforms to the pattern $\mathcal{C}(3)$, with the symbols $a, b, c$ as the pivots. Since $L_i$ forces a square, we know that $\boldsymbol{w}a$, $\boldsymbol{w}b$, $\boldsymbol{w}c$ has a square for a suffix (as $\boldsymbol{w}$ itself was square-free). Let $\boldsymbol{t}a\boldsymbol{t}a$, $\boldsymbol{u}b\boldsymbol{u}b$, $\boldsymbol{v}c\boldsymbol{v}c$ be the squares created by appending $a, b$ and $c$ to $\boldsymbol{w}$, respectively. Here $\boldsymbol{t}$, $\boldsymbol{u}$, $\boldsymbol{v}$ are treated as subwords of $\boldsymbol{w}$.

As all three squares $\boldsymbol{t}a\boldsymbol{t}a$, $\boldsymbol{u}b\boldsymbol{u}b$, $\boldsymbol{v}c\boldsymbol{v}c$ are suffixes of the string $\boldsymbol{w}$, it follows that $\boldsymbol{t}$, $\boldsymbol{u}$, $\boldsymbol{v}$ must be of different sizes, and so we can order them without loss of generality as follows: $|\boldsymbol{t}a\boldsymbol{t}| < |\boldsymbol{u}b\boldsymbol{u}| < |\boldsymbol{v}c\boldsymbol{v}|$. It also follows from the fact that all three are suffixes of $\boldsymbol{w}$, the squares from left-to-right are suffixes of each other. Hence, while $\boldsymbol{t}$ may be empty, we know that $\boldsymbol{u}$ and $\boldsymbol{v}$ are not. We now consider different cases of the overlap of $\boldsymbol{t}a\boldsymbol{t}$, $\boldsymbol{u}b\boldsymbol{u}$, $\boldsymbol{v}c\boldsymbol{v}$, showing in each case that the resulting string has a suffix conforming to the pattern $\mathcal{C}(3)$. Note that it is enough to consider the interplay of $\boldsymbol{u}b\boldsymbol{u}$, $\boldsymbol{v}c\boldsymbol{v}$, as then the interplay of $\boldsymbol{t}a\boldsymbol{t}$, $\boldsymbol{u}b\boldsymbol{u}$ is symmetric and follows by analogy. Also keep in mind that the assumption is that $\boldsymbol{w}$ is square-free; this eliminates some of the possibilities as can be seen below.

1. $\boldsymbol{v} = \boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$ as shown in Figure 1, where $\boldsymbol{p}$ is a proper non-empty prefix of $\boldsymbol{v}$. Since $\boldsymbol{w}$ is square-free, we assume that $\boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$ has no square, and therefore $\boldsymbol{p} \neq \boldsymbol{u}$ and $\boldsymbol{p} \neq b$. From this, we get $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}c\boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$. Therefore, this case is possible.

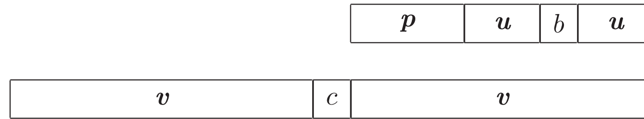



**Figure 1.** $\boldsymbol{v} = \boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$

2. $\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$ as shown in Figure 2. Then, $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}c\boldsymbol{u}b\boldsymbol{u}$. This case is also possible.

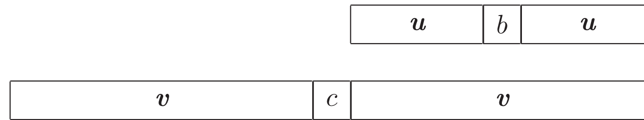



**Figure 2.** $\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$

3. $c\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$ as shown in Figure 3, then $\boldsymbol{u}_1 = c$. Let $\boldsymbol{u} = c\boldsymbol{s}$, where $\boldsymbol{s}$ is a proper non-empty suffix of $\boldsymbol{u}$, then $\boldsymbol{v}c\boldsymbol{v} = c\boldsymbol{s}bc\boldsymbol{s}cc\boldsymbol{s}bc\boldsymbol{s}$. The subword '$cc$' indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

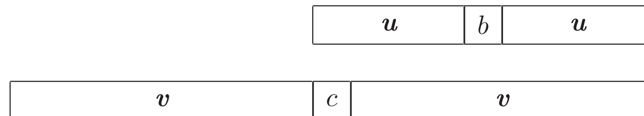



**Figure 3.** $c\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$

4. $\boldsymbol{vcv} = \boldsymbol{qubu}$ and $|c\boldsymbol{v}| < |\boldsymbol{ubu}|$ as shown in Figure 4, where $\boldsymbol{q}$ is a proper prefix of $\boldsymbol{vcv}$. Let $\boldsymbol{u} = \boldsymbol{pcs}$, where $\boldsymbol{p}$, $\boldsymbol{s}$ are proper prefix and suffix of $\boldsymbol{u}$. Therefore $\boldsymbol{v} = \boldsymbol{sbpcs}$. Since $\boldsymbol{p}$ is also a proper suffix of $\boldsymbol{v}$, one of the following must be true:

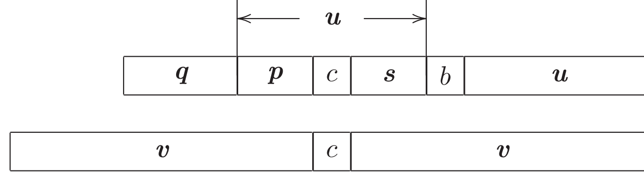



**Figure 4.** $\boldsymbol{vcv} = \boldsymbol{ngbu}$ and $|c\boldsymbol{v}| < |\boldsymbol{ubu}|$

(a) $|\boldsymbol{s}| = |\boldsymbol{p}|$ and so $\boldsymbol{s} = \boldsymbol{p}$. Since $\boldsymbol{s} = \boldsymbol{p}$, $\boldsymbol{v} = \boldsymbol{sbscs}$ and $\boldsymbol{vcv} = \boldsymbol{sbscscsbscs}$. The subword '$\boldsymbol{scsc}$', indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

(b) $|\boldsymbol{s}| > |\boldsymbol{p}|$ and so $\boldsymbol{s} = \boldsymbol{rp}$, where $\boldsymbol{r}$ is a proper non-empty prefix of $\boldsymbol{s}$. Substituting $\boldsymbol{rp}$ for $\boldsymbol{s}$, we have $\boldsymbol{v} = \boldsymbol{sbpcs} = \boldsymbol{rpbpcrp}$ and $\boldsymbol{vcv} = \boldsymbol{rpbpcrpcrpbpcrp}$. The subword '$\boldsymbol{crpcrp}$' indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

(c) $|\boldsymbol{s}| < |\boldsymbol{p}|$ and so $\boldsymbol{p} = \boldsymbol{rs}$, where $\boldsymbol{r}$ is a proper non-empty prefix of $\boldsymbol{p}$. Substituting $\boldsymbol{rs}$ for $\boldsymbol{p}$, we have $\boldsymbol{v} = \boldsymbol{sbpcs} = \boldsymbol{sbrscs}$ and $\boldsymbol{vcv} = \boldsymbol{sbrscscsbrscs}$. The subword '$\boldsymbol{scsc}$' indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

From the above analysis, we can conclude that for $L_i$ to force a square on a square-free string $\boldsymbol{w}$, it must be the case that $\boldsymbol{v} = \boldsymbol{zubu}$, where $\boldsymbol{z}$ is a prefix (possibly empty) of $\boldsymbol{v}$ and $\boldsymbol{z} \neq \boldsymbol{u}$ and $\boldsymbol{z} \neq b$.

Similarly, we get $\boldsymbol{u} = \boldsymbol{ytat}$, where $\boldsymbol{y}$ is a prefix (possibly empty) of $\boldsymbol{u}$ and $\boldsymbol{y} \neq \boldsymbol{t}$ and $\boldsymbol{z} \neq \boldsymbol{y}$. Substituting values of $\boldsymbol{u}$ in $\boldsymbol{v}$, we get $\boldsymbol{v} = \boldsymbol{zytatbytat}$ and $\boldsymbol{vcv} = \boldsymbol{zytatbytatczytatbytat}$. But $\boldsymbol{vcv}$, is a suffix of the square-free string $\boldsymbol{w}$, and it conforms to the offending suffix $\mathcal{C}(3)$ where the elements $a, b, c$ are the pivots.

Therefore, we have shown that if an alphabet $L_i$ forces a square in a square-free string $\boldsymbol{w}$, then $\boldsymbol{w}$ has a suffix conforming to the offending suffix $\mathcal{C}(3)$. ☐

The following Corollary exploits the fact that an alphabet $L_{i+1}$ *forces* a square on a square-free string $\boldsymbol{v}$ of length $i$ iff $\boldsymbol{v}$ has an offending suffix. But, the size of an offending suffix grows exponentially in the size of the alphabets in the list.

**Corollary 2.** *If $L$ is a list in $\mathcal{L}_n$ of length at most $2^n - 1$, then $L$ is admissible.*

*Proof.* Suppose that $L \in \mathcal{L}_n$ and $|L| = 2^n - 1$. We show how to construct a square-free $\boldsymbol{w}$ over $L$. Let $\boldsymbol{w}_1$ be any one of the three symbols in $L_1$. Now, inductively for $i \in [2^n - 2]$, assume that $\boldsymbol{v} = \boldsymbol{w}_1\boldsymbol{w}_2 \cdots \boldsymbol{w}_i$ is square-free. If $L_{i+1}$ forces a square on $\boldsymbol{v}$, then by Theorem 1, $\boldsymbol{v}$ must have an offending suffix. But as the shortest possible offending suffix for $|L_{i+1}| = n$ is $\mathcal{C}_s(n)$ of length $2^n - 1$ (see (3)), we get a contradiction since $|\boldsymbol{v}| \leq 2^n - 2$. Thus $L_{i+1}$, for $i \in [2^n - 2]$, cannot force a square, which means that we can select at least one symbol $\sigma \in L_{i+1}$ so that $\boldsymbol{v}\sigma$ is square free. We proceed this way until $i = 2^n - 2$ and output a square-free string $\boldsymbol{w}$ of length $2^n - 1$ over $L$. Hence $L$ is admissible. ☐

From Theorem 1, we know that an alphabet in a list $L \in \mathcal{L}_3$ can force a square on a square-free string $\boldsymbol{w}$ iff $\boldsymbol{w}$ has a suffix $\boldsymbol{s}$ conforming to the offending suffix $\mathcal{C}(3)$. The question is whether $\boldsymbol{s}$ is unique, that is, does the square-free string $\boldsymbol{w}$ contain more that one suffix that conforms to the offending suffix pattern? In Lemma 3, we show that any square-free string $\boldsymbol{w}$ over $L \in \mathcal{L}_3$ has only one suffix $\boldsymbol{s}$ conforming to the offending suffix (w.r.t fixed pivots) if any, that is $\boldsymbol{s}$ is unique.

**Lemma 3.** *Suppose $\boldsymbol{w}$ is a square-free string over $L = L_1, L_2, \ldots, L_{n-1}$, and $L \in \mathcal{L}_3$. If $\boldsymbol{w}$ has suffixes $\boldsymbol{s}, \boldsymbol{s}'$ conforming to $\mathcal{C}(3)$ with pivots $L_n$ (where $|L_n| = 3$), then $\boldsymbol{s} = \boldsymbol{s}'$.*

*Proof.* The proof is by contradiction. Suppose that the square-free string $\boldsymbol{w}$ over $L \in \mathcal{L}_3$ has two distinct suffixes $\boldsymbol{s}$ and $\boldsymbol{s}'$ conforming to the offending suffix $\mathcal{C}(3)$ with pivots $L_n = \{a, b, c\}$. That is $\exists h, h(\mathcal{C}(3)) = \boldsymbol{s}$ and $\exists h', h'(\mathcal{C}(3)) = \boldsymbol{s}'$, and $\boldsymbol{s} \neq \boldsymbol{s}'$, and both have pivots in $\{a, b, c\}$. Without loss of generality, we assume that $|\boldsymbol{s}| < |\boldsymbol{s}'|$, and since they are suffixes of $\boldsymbol{w}$, $\boldsymbol{s}$ is a suffix of $\boldsymbol{s}'$. We now examine all possible cases of overlap. Note that $\boldsymbol{s}' = h'(\mathcal{C}(3)) = h'(\boldsymbol{X}_2\mathcal{C}(2)a_3\boldsymbol{X}_2\mathcal{C}(2))$ for some morphism $h'$. To examine the cases of overlap, let $\boldsymbol{v} = h'(\boldsymbol{X}_2\mathcal{C}(2))$, then $\boldsymbol{s}' = \boldsymbol{v}h'(a_3)\boldsymbol{v}$, where $h'(a_3)$ represents the middle symbol of $\boldsymbol{s}'$. Similarly, the middle symbol of $\boldsymbol{s}$ is represented by $h(a_3)$ for some morphism $h$. We intentionally use $h'(a_3)$ in $\boldsymbol{s}'$ (and $h(a_3)$ in $\boldsymbol{s}$) as we want to cover all the six different ways in which the variables $a_1, a_2, a_3$ are mapped to pivots $a, b, c$.

1. If $|\boldsymbol{s}| \leq \lfloor |\boldsymbol{s}'|/2 \rfloor$, then $\boldsymbol{v} = \boldsymbol{ps}$ (see Figure 5), where $\boldsymbol{p}$ is a prefix of $\boldsymbol{v}$, and $\boldsymbol{ps}h'(a_3)$ is a prefix of $\boldsymbol{s}'$. Observe that, when $|\boldsymbol{s}| = \lfloor |\boldsymbol{s}'|/2 \rfloor$, $\boldsymbol{p} = \varepsilon$. Since $\boldsymbol{s}$ is an offending suffix, we know that $\boldsymbol{s}h'(a_3)$ has a square and hence $\boldsymbol{s}'$ has a square and it follows that $\boldsymbol{w}$ has a square — contradiction.

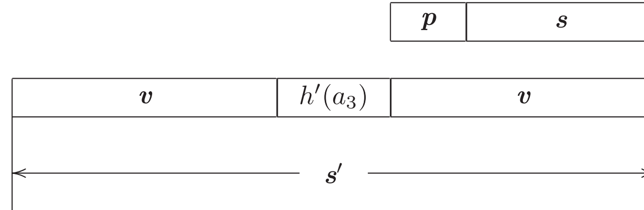



**Figure 5.** $\boldsymbol{v} = \boldsymbol{ps}$

2. If $|\boldsymbol{s}| = \lfloor |\boldsymbol{s}'|/2 \rfloor + 1$, then $\boldsymbol{s} = h'(a_3)\boldsymbol{u}h(a_3)h'(a_3)\boldsymbol{u}$ (see Figure 6), where $\boldsymbol{u}$ is a non-empty subword of $\boldsymbol{s}$, and $\boldsymbol{v} = \boldsymbol{u}h(a_3)h'(a_3)\boldsymbol{u}$. If the morphisms $h$ and $h'$ map $a_3$ to the same element in $\{a, b, c\}$, that is $h'(a_3) = h(a_3)$, then $\boldsymbol{s}$ has a square '$h(a_3)h(a_3)$' and therefore $\boldsymbol{w}$ has a square — contradiction. When $h'(a_3) \neq h(a_3)$, without loss of generality, we assume $h'(a_3) = c$ and $h(a_3) = a$, then $\boldsymbol{v} = \boldsymbol{uacu}$ and $\boldsymbol{s}' = \boldsymbol{vcv} = \boldsymbol{uacucuacu}$ has a square '$\boldsymbol{cucu}$' and it follows that $\boldsymbol{w}$ has a square — contradiction.
3. If $|\boldsymbol{s}| > \lfloor |\boldsymbol{s}'|/2 \rfloor + 1$, then $\boldsymbol{s} = \boldsymbol{p}h'(a_3)\boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$, where $\boldsymbol{p}$ is a non-empty prefix of $\boldsymbol{s}$ and $\boldsymbol{u}$ is a subword (possibly empty) of $\boldsymbol{s}$. Also, $\boldsymbol{v} = \boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$ and $\boldsymbol{s}' = \boldsymbol{v}h'(a_3)\boldsymbol{v} = \boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}h'(a_3)\boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$. We can see that $\boldsymbol{s}'$ has a square '$h'(a_3)\boldsymbol{u}h'(a_3)\boldsymbol{u}$', and it follows that $\boldsymbol{w}$ has a square — contradiction.

This ends the proof. □

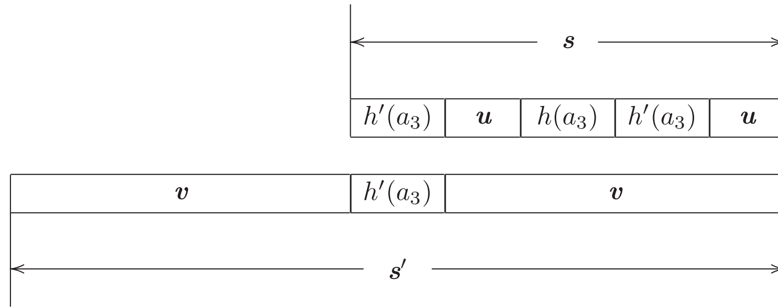



**Figure 6.** $\boldsymbol{v} = \boldsymbol{u}h(a_3)h'(a_3)\boldsymbol{u}$

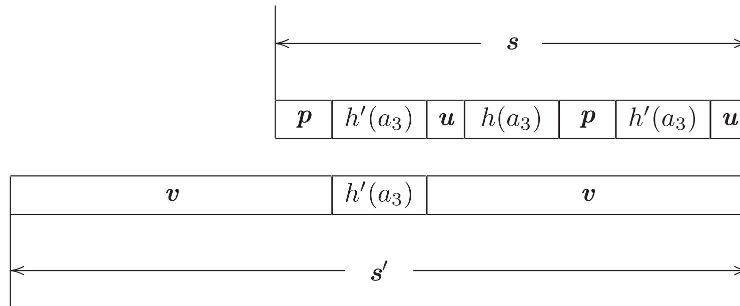



**Figure 7.** $\boldsymbol{v} = \boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$

Suppose the class of lists $\mathcal{L}_3$ is inadmissible, and $L \in \mathcal{L}_3$ is a minimum length list that is inadmissible. By Corollary 2 we know that such a list is of length at least eight. Let $L = L_1, L_2, \ldots, L_{n+1}$, where $n \geq 8$, and let $L' = L_1, L_2, \ldots, L_n$, so that $L = L', L_{n+1}$. Then by Theorem 1 every square-free word over $L'$ has a suffix that conforms to the offending suffix $\mathcal{C}(3)$, where the pivot elements are the symbols of the alphabet $L_{n+1}$. That is, if $\boldsymbol{w}$ is a square-free word over $L'^{+}$, then there is a non empty suffix $\boldsymbol{s}$ of $\boldsymbol{w}$ and a morphism $h$ such that $h(\mathcal{C}(3)) = \boldsymbol{s}$.

If we are able to replace one of the pivots in $\boldsymbol{s}$ with another element from its respective alphabet, such that the new string $\boldsymbol{w'}$ remains square-free and has no suffix conforming to $\mathcal{C}(3)$, then we can show that $L$ is admissible. Simply, use this $\boldsymbol{w'}$ over $L'$, and append to it a symbol from the alphabet $L_{n+1}$, such that the resulting string is square-free. We know that such a symbol exists as $\boldsymbol{w'}$ was square-free with no offending suffix.

## 4 Borders and squares

In this section we relate borders of a string to its squares. There is a vast literature on borders; see for instance [13].

**Lemma 4.** *A string $\boldsymbol{w}$ is square-free if and only if for every subword $\boldsymbol{s}$ of $\boldsymbol{w}$, if $\beta$ is a border of $\boldsymbol{s}$, then $|\beta| < \lceil |\boldsymbol{s}|/2 \rceil$.*

*Proof.* ($\Rightarrow$) Suppose that $\boldsymbol{s}$ is a subword of $\boldsymbol{w}$ and it has a border $\beta$ such that $|\beta| \geq \lceil |\boldsymbol{s}|/2 \rceil$. From Figure 8 we can see that $\beta$ must have a prefix $\boldsymbol{p}$ which yields a square $\boldsymbol{pp}$ in $\boldsymbol{s}$ and hence in $\boldsymbol{w}$, and so $\boldsymbol{w}$ is not square-free — contradiction.
($\Leftarrow$) Suppose $\boldsymbol{w}$ has a square $\boldsymbol{s} = \boldsymbol{uu}$. But $\boldsymbol{s}$ is a subword of $\boldsymbol{w}$ and it has a border $\beta = \boldsymbol{u}$ where $|\beta| \geq \lceil |\boldsymbol{s}|/2 \rceil$ — contradiction. □

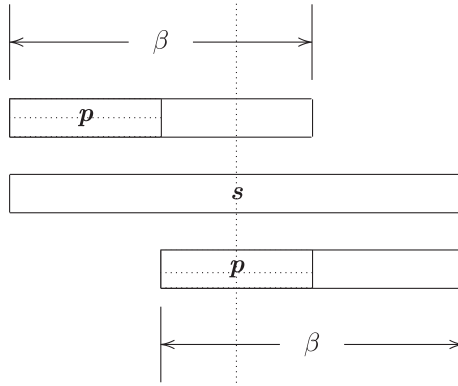



**Figure 8.** "⇒" direction of the proof for Lemma 4

## 5 Repetitions and compression

Suppose that we want to encode the $\boldsymbol{w}$'s, as $\langle \boldsymbol{w} \rangle$, in a way that takes advantage of the repetitions in $\boldsymbol{w}$. The intuition, of course, is that strings with long repetitions can be compressed considerably, and so encoded with fewer bits. We can then use the basic Kolmogorov observation about the existence of incompressible strings to deduce that not all strings can have long repetitions. On the other hand, short repetitions are in some sense local, and so they are easier to avoid. Perhaps we can use this approach to prove the existence of square-free strings in $\mathcal{L}_3$.

Assume that the $L_i$'s are ordered, and since each $L_i$ has three symbols, we can encode the contents of each $L_i$ with 2 bits:

| Encoding | Symbol |
|---|---|
| 00 | 1st symbol |
| 01 | 2nd symbol |
| 10 | 3rd symbol |
| 11 | *separator* |

Suppose now that $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{v}\boldsymbol{v}\boldsymbol{w}_2$, where $|\boldsymbol{w}| = n$, $|\boldsymbol{v}| = \ell$, that is, $\boldsymbol{w}$ is a string over $L$ of length $n$ containing a square of length $\ell$. Then, we propose the following scheme for encoding $\boldsymbol{w}$'s: $\langle \boldsymbol{w} \rangle := \langle \boldsymbol{w}_1 \rangle 11 \langle \boldsymbol{v} \rangle 11 \langle \boldsymbol{w}_2 \rangle$. A given $\boldsymbol{w}$ does not necessarily have a unique encoding, as it may have several squares; but we insist that the encoding always picks a maximal square (in length). Note also that $\langle \boldsymbol{w} \rangle$ encodes $\boldsymbol{w}$ over $L$ as a string over $\Sigma = \{0, 1\}$.

Note that $|\langle \boldsymbol{w} \rangle| = 2(n - 2\ell) + 2\ell + 4$, where the term $2(n - 2\ell)$ arises from the fact that $|\boldsymbol{w}_1| + |\boldsymbol{w}_2|$ have $n - 2\ell$ symbols (as strings over $L$), and each such symbol is encoded with two bits, hence $2(n - 2\ell)$. The two separators $11, 11$ take 4 bits, and the length of $\boldsymbol{v}$ is $\ell$ (as a string over $L$), and so it takes $2\ell$ bits.

It is clear that we can extract $\boldsymbol{w}$ out of $\langle \boldsymbol{w} \rangle$ (uniquely), and so $\langle \cdot \rangle$ is a valid encoding; for completeness, let $f$ decode strings: $f : \Sigma^* \longrightarrow L^+$ work as follows:

$$f(\langle \boldsymbol{w} \rangle) = f(\langle \boldsymbol{w}_1 \rangle 11 \langle \boldsymbol{v} \rangle 11 \langle \boldsymbol{w}_2 \rangle) = \boldsymbol{w}_1\boldsymbol{v}\boldsymbol{v}\boldsymbol{w}_2,$$

and if the input is not a well-formed encoding, say it is 111111, then we let $f$ output, for instance, the lexicographically first string over $L^+$.

On the other hand, $\langle \cdot \rangle : L^+ \longrightarrow \Sigma^*$ encodes strings by finding the longest square $\boldsymbol{v}$ in a given $\boldsymbol{w}$ (if there are several maximal squares, it picks the first one, i.e., the one where the index of first symbol of $\boldsymbol{v}\boldsymbol{v}$ is smallest), yielding $\boldsymbol{w}_1\boldsymbol{v}\boldsymbol{v}\boldsymbol{w}_2$, and outputting $\langle \boldsymbol{w}_1 \rangle 11 \langle \boldsymbol{v} \rangle 11 \langle \boldsymbol{w}_2 \rangle$.

Suppose now that for a given $L = L_1, L_2, \ldots, L_n$ every string has a maximal square of size at least $\ell_0$. We want to bound how big can $\ell_0$ be; to this end, we want to find $\ell_0$ such that:

$$2^{2(n-2\ell_0)+2\ell_0+4} < 3^n. \tag{5}$$

The reason is that the term on the left counts the maximal number of possible encodings given the assumption that every string over $L$ has a square of size at least $\ell_0$, while the term on the right is the size of $|L^+|$. The inequality expresses that if $\ell_0$ is assumed to be too big, then we won't be able to encode all the $3^n$ strings in $L^+$.

Since (5) can be simplified to $2^{2n-2\ell_0+4} < 3^n$, and using $\log_2$ on both sides we obtain: $2n - 2\ell_0 + 4 < \log_2 3n < 1.6n$, which gives us $n - \ell_0 + 2 < 0.8n$, and so $\ell_0 > n - 0.8n + 2 > 0.2n$. Thus, given any $L = L_1, L_2, \ldots, L_n$, there always is a $\boldsymbol{w} \in L^+$ with a square no longer than $\frac{1}{5}n$. Can we strengthen this technique to give a Kolmogorov style proof to prove that $\mathcal{L}_3$ is admissible?

## References


1. S. Arson: *Proof of the existence of asymmetric infinite sequences (russian).* Mat. Sbornik, 2 1937, pp. 769–779.
2. D. R. Bean, A. Ehrenfeucht, and G. F. McNulty: *Avoidable patters in strings of symbols.* Pacific Journal of Mathematics, 85(2) 1979, pp. 261–294.
3. J. Berstel, A. Lauve, C. Reutenauer, and F. V. Saliola: *Combinatorics on Words: Christoffel Words and Repetitions in Words*, American Mathematical Society, 2008.
4. J. Berstel and D. Perrin: *The origins of combinatorics of words.* Electronic Journal of Combinatorics, 28 2007, pp. 996–1022.
5. J. Cooper and D. Rorabaugh: *Bounds on zimin word avoidance.* Electronic Journal of Combinatorics, 21(1) 2014.
6. J. D. Currie: *Which graphs allow infinite non-repetitive walks?* Discrete Mathematics, 87 1991, pp. 249–260.
7. J. Grytczuk, J. Kozik, and P. Micek: *A new approach to nonrepetitive sequences.* arXiv:1103.3809, December 2010.
8. J. Leech: *A problem on strings of beads.* Mathematical Gazette, December 1957, p. 277.
9. N. Mhaskar and M. Soltys: *Non-repetitive string over alphabet list*, in WALCOM: Algorithms and Computation, vol. 8973 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2015, pp. 270–281.
10. M. Morse and G. A. Hedlund: *Unending chess, symbolic dynamics and a problem in semigroups.* Duke Math. J, 11 1944, pp. 1–7.
11. N. Rampersad and J. Shallit: *Repetitions in words.* May 2012.
12. J. Shallit: *A second course in formal languages and automata theory*, Cambridge Univeristy Press, 2009.
13. B. Smyth: *Computing Patterns in Strings*, Pearson Education, 2003.
14. A. Thue: *Über unendliche Zeichenreichen.* Norsek Vid. Selsk. Srk., I Mat. Nat. Kl., 7 1906, pp. 1–22.
15. A. Thue: *Über die gegenseitige lage gleicher teile gewisser Zeichenreihen.* Kra. Vidensk. Selsk. Skrifter., I. Mat. Nat. Kl., 1 1912, pp. 1–67.
16. A. I. Zimin: *Blocking sets of terms.* Mat. Sbornik, 119 1982, pp. 363–375.

# Computing Smallest and Largest Repetition Factorizations in $O(n \log n)$ Time

Hiroe Inoue[1], Yoshiaki Matsuoka[1], Yuto Nakashima[1,2], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Japan
[2] Japan Society for the Promotion of Science (JSPS), Japan
{hiroe.inoue, yuto.nakashima, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp

**Abstract.** A factorization $f_1, \ldots, f_m$ of a string $w$ is called a *repetition factorization* of $w$ if each factor $f_i$ is a repetition, namely, $f_i = x^k x'$ for some non-empty string $x$, an integer $k \geq 2$, and $x'$ being a proper prefix of $x$. Dumitran et al. (Proc. SPIRE 2015) proposed an algorithm which computes a repetition factorization of a given string $w$ in $O(n)$ time, where $n$ is the length of $w$. In this paper, we propose two algorithms which compute smallest/largest repetition factorizations in $O(n \log n)$ time. The first algorithm is a simple $O(n \log n)$ space algorithm while the second one uses only $O(n)$ space.

## 1 Introduction

A *factorization* of a string $w$ is a sequence $f_1, \ldots, f_m$ of non-empty substrings of $w$ such that $w = f_1 \cdots f_m$. The *size* of the factorization is the number $m$ of factors contained in the factorization. Numerous types of factorizations of strings have been considered, of which the most well-studied is the Lempel-Ziv factorizations and its family [22,23,19,21]. Not only do they have an apparent application to data compression, but also they play key roles in other stringology problems [14,13,20]. The Lyndon factorizations [3] are classical subjects in combinatorics on words, and also have some application in data compression, i.e., in a variant of the Burrows-Wheeler transform [15]. It is known that given a string of length $n$, these factorizations for the string can be computed in $O(n)$ time [5,8,18].

Recently, the problems of factorizing a given string into some kinds of "combinatorial" structures have been studied [1,2,12,9]. In this paper, we are particularly interested in the following types of factorizations: A factorization $f_1, \ldots, f_m$ of a string $w$ is said to be a *square factorization* of $w$ if each factor $f_i$ is a square (i.e., $f_i$ is of form $x^2$ for some string $x$), and it is called a *repetition factorization* of $w$ if each factor $f_i$ is a repetition (i.e., $f_i$ is of form $x^k x'$ with $k \geq 2$ and $x'$ begin a proper prefix of $x$). Dumitran et al. showed how to compute a square factorization of an input string of length $n$ in $O(n \log n)$ time, and a repetition factorization in $O(n)$ time [7]. Very recently, Matsuoka et al. [17] proposed an improved algorithm which finds a square factorization in $O(n)$ time, and also proposed algorithms which compute square factorizations of smallest/largest size in $O(n \log n)$ time.

In this paper, we tackle the problems of computing repetition factorizations of smallest/largest sizes of a given string $w$ of length $n$, and show two algorithms for computing such factorizations in $O(n \log n)$ time. The first algorithm, which is based on a reduction of the problem to the classical shortest/longest path problem on a DAG, requires $O(n \log n)$ space as the underlying DAG requires $O(n \log n)$ space. On

Hiroe Inoue, Yoshiaki Matsuoka, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Computing Smallest and Largest Repetition Factorizations in $O(n \log n)$ Time, pp. 135–145.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

the other hand, the second algorithm shaves the space requirement to $O(n)$ but retains $O(n \log n)$ running time, by simulating the first one with dynamic programming.

The rest of the paper is organized as follows. In Section 2, we state some notations and definitions on strings. In Section 3, we show how to find smallest/largest repetition factorizations in $O(n \log n)$ time and space. Section 4 shows an $O(n \log n)$-time and $O(n)$-space solution to the problem. Section 5 concludes and states some future work.

# 2 Preliminaries

## 2.1 Strings

Let $\Sigma$ be a finite *alphabet*. An element of $\Sigma^*$ is called a *string*. The length of a string $w$ is denoted by $|w|$. The empty string $\varepsilon$ is a string of length 0, namely, $|\varepsilon| = 0$. Let $\Sigma^+$ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $w = xyz$, $x$, $y$ and $z$ are called a *prefix*, *substring*, and *suffix* of $w$, respectively. A prefix $x$ is called a *proper prefix*, if $x \neq w$.

The $i$-th character of a string $w$ is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string $w$ and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of $w$ that begins at position $i$ and ends at position $j$. For convenience, let $w[i..j] = \varepsilon$ when $i > j$.

For any string $w$, let $w^1 = w$, and for any integer $k \geq 2$ let $w^k = ww^{k-1}$. A non-empty string $w$ is called *primitive* if there is no string $x$ s.t. $w = x^k$ for some integer $k \geq 2$.

An integer $p \geq 1$ is said to be a *period* of a string $w$ if $w[i] = w[i+p]$ for all $1 \leq i \leq |w| - p$. The following well-known *periodicity lemma* is useful.

**Lemma 1 (Periodicity Lemma [10]).** *If two periods $p, q$ of string $w$ of length $n$ satisfies $p + q - \gcd(p,q) \leq n$, then $\gcd(p,q)$ is also a period of $w$.*

## 2.2 Repetitive structures in strings

We define repetitive structures used in this paper below.

**Definition 2 (Squares).** *A non-empty string $s$ is said to be a* square*, if $s = x^2$ for some string $x$.*

A square $x^2$ is called a *primitively rooted square* if $x$ is primitive.

**Definition 3 (Repetitions).** *A triple $(beg, end, p)$ is said to be a* repetition *of a string $w$, if the smallest period $p$ of the substring $w[beg..end]$ satisfies $|w[beg..end]| \geq 2p$. In other words, the substring $w[beg..end]$ is of form $x^k x'$, where $x = w[beg..beg+p-1]$ is primitive, $k \geq 2$, and $x'$ is a possibly empty proper prefix of $x$.*

We will sometimes identify the triple $(beg, end, p)$ as the corresponding substring $w[beg..fin]$ with the smallest period $p$, and will call the substring $w[beg..fin]$ as a repetition.

**Definition 4 (Runs (maximal repetitions)).** *A repetition $(beg, end, p)$ is said to be a* run *(or a* maximal repetition*) of a string $w$ of length $n$, if $beg = 1$ or $w[beg-1] \neq w[beg+p-1]$, and $end = n$ or $w[end+1] \neq w[end-p+1]$. In other words, the periodicity of the substring $w[beg..end]$ with the smallest period $p$ cannot be extended to the left nor the right.*

Let $Runs(w)$ denote the set of runs of string $w$. The following result is well-known.

**Theorem 5 ([4]).** *For any string $w$ of length $n$, $|Runs(w)| < n$. Also, $Runs(w)$ can be computed in $O(n)$ time for integer alphabets of size $n^{O(1)}$.*

In this paper, we suppose that the runs in $Runs(w)$ are sorted in increasing order of beginning positions and the $i$-th run in $Runs(w)$ is denoted by $r_i = (beg_i, end_i, p_i)$ for any $1 \leq i \leq |Runs(w)|$ (i.e., $beg_i \leq beg_{i+1}$ for any $1 \leq i < |Runs(w)|$). This makes it easy for us to explain our algorithm, and we can sort the runs in $Runs(w)$ in this order in $O(n)$ time by a bucket sort.

The following observation shows a relation between *runs* and *repetitions*.

**Observation 1** *For any repetition $w[i..j]$ of the shortest period $s$ occurring in a string $w$, there exist unique strings $x \in \Sigma^+$ and $y \in \Sigma^*$ s.t. $x$ is a primitively rooted square of length $2s$ and $w[i..j] = xy$. For any repetition $w[i..j]$ of the shortest period $s$ occurring in a string $w$, there exists a unique run $r = (beg, end, p)$ s.t. $beg \leq i < j \leq end$ and $s = p$.*

### 2.3 Problems

In this section, we formally define the problems we consider in this paper.

**Definition 6 (Repetition factorizations).** *A sequence $f_1, \ldots, f_m$ of non-empty strings is said to be a* repetition factorization *of a string $w$, if $w = f_1 \cdots f_m$ and $f_i$ for each $1 \leq i \leq m$ is a repetition of $w$.*

Each $f_i$ in a repetition factorization $f_1, \ldots, f_m$ of a string $w$ is called a *factor* of the factorization. The *size* of the repetition factorization is the number $m$ of factors in the factorization.

The problem we tackle in this paper is the following:

*Problem 7 (Smallest/Largest Repetition Factorization).* Given a string $w$ of length $n$, compute a repetition factorization of smallest/largest size.

*Example 8.* If we are given a string $w =$ `abaabaababaabaababababa`, then we return one of the following as a largest repetition factorization of $w$.

- `abaaba|abab|aabaab|ababa`
- `abaaba|abab|aabaaba|baba`
- `abaaba|ababa|abaaba|baba`
- `abaabaa|baba|abaaba|baba`

We return one of the following as a smallest repetition factorization of $w$.

- `abaabaababaabaab|ababa`
- `abaabaababaabaaba|baba`

### 2.4 Graphs

Our solutions to Problem 7 are based on a certain DAG defined over the runs appearing in a given string.

For any DAG $G$, let $\pi_1 = v_i \cdots v_j$ and $\pi_2 = v_j \cdots v_k$ be any paths on $G$ ($v_\ell$ is a node of $G$ for all $i \leq \ell \leq k$). We denote the concatenated path $v_i \cdots v_k$ by $\pi_1 \oplus \pi_2$.

# 3 $O(n \log n)$-time and space solution

In this section, we show an $O(n \log n)$-time and space solution to Problem 7. Our main idea is to reduce the problem to a shortest/longest path problem on a directed acyclic graph. In Section 3.1, we define a DAG $G$ that is based on the runs of the input string $w$. We will call $G$ as the *repetition graph* of $w$.

In this paper, we focus on the largest repetition factorization problem, but the smallest repetition factorization problem can be solved in a similar way.

## 3.1 Definition of repetition graphs

The repetition graph $G = (V, E)$ of a string $w$ is an edge-weighted directed acyclic graph. First, we define the set $V$ of nodes. It consists of the two mutually disjoint subsets $V'$ and $V''$ of nodes, namely $V = V' \cup V''$ such that

$$V' = \{(0, j) \mid 0 \le j \le n\},$$
$$V'' = \bigcup_{i=1}^{|Runs(w)|} V_i'',$$

where $V_i'' = \{(i, j) \mid beg_i + 2p_i - 1 \le j \le end_i\}$ for each $r_i = (beg_i, end_i, p_i) \in Runs(w)$.

The set $E$ of edges consists of the three mutually disjoint subsets $E'$, $E''$, and $E'''$ of edges, namely $E = E' \cup E'' \cup E'''$, such that

$$E' = \{((i_1, j_1), (i_2, j_2)) \mid j_2 - j_1 = 2p_{i_2}, (i_1, j_1) \in V', (i_2, j_2) \in V''\},$$
$$E'' = \{((i_1, j_1), (i_2, j_2)) \mid i_1 = i_2, j_1 + 1 = j_2, (i_1, j_1), (i_2, j_2) \in V''\},$$
$$E''' = \{((i_1, j_1), (i_2, j_2)) \mid j_1 = j_2, (i_1, j_1) \in V'', (i_2, j_2) \in V'\}.$$

The weight of each edge $e \in E$, denoted $weight(e)$, is defined by

$$weight(e) = \begin{cases} 1 & \text{if } e \in E', \\ 0 & \text{if } e \in E'' \cup E'''. \end{cases}$$

We call the nodes $v_s = (0, 0) \in V'$ and $(0, n) \in V'$ as the *starting node* and *ending node* of the graph $G$, respectively.

For any node $v = (i, j)$, let $run(v) = i$ and $pos(v) = j$. Intuitively, each node $v = (run(v), pos(v))$ in $V''$ corresponds to the $run(v)$-th run in $Runs(w)$ and the position $pos(v)$ in $w$ satisfying $beg_i + 2p_i - 1 \le j \le end_i$. Any node $v \in V'$ does not correspond to any run, but for convenience we let $run(v) = 0$. We also define an auxiliary node $v_s = (0, 0)$ for convenience. We remark that although $pos(v_s) = 0$, the index of any string $w$ begins with 1 throughout this paper.

We describe an intuitive explanation for the repetition graphs (more details will be given in the next subsection). See Figure 1 which shows an example of a repetition graph. Let $(v_1, v_2)$ be an edge in $E'$ (drawn as a diagonal arrow). Edge $(v_1, v_2)$ represents a primitively rooted square $w[pos(v_1) + 1..pos(v_2)]$ of length $2p_{run(v_2)}$. By Observation 1, any repetition $z$ can be decomposed as $xy$, where $x$ is a prefix of $z$ which is the primitively rooted square having the same smallest period as $z$, and $y$ is the remainder (possibly empty). The repetition graph $G$ for string $w$ represents every repetition in $w$ as a path which is a concatenation of a diagonal arrow (i.e. the primitively rooted square part) and some horizontal arrows (i.e. the remainder). For

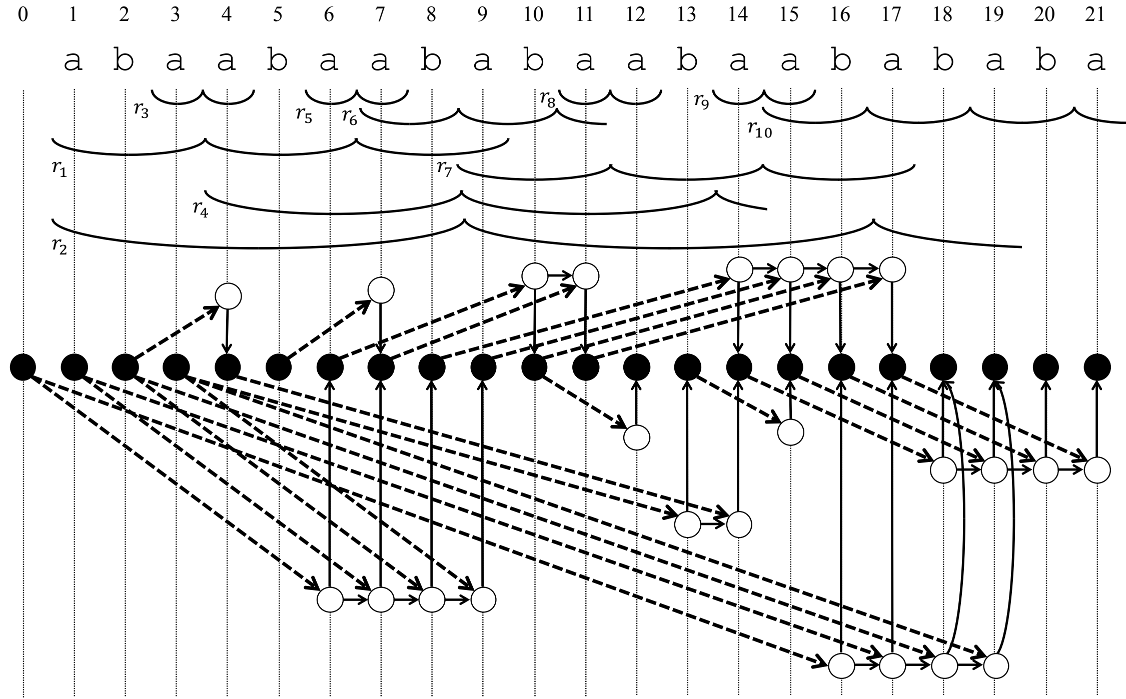



**Figure 1.** The repetition graph $G$ of string abaabaababaabaababababa. Black circles represent the nodes in $V'$ and white circles represent the nodes in $V''$. Dashed arrows represents the edges in $E'$, horizontal solid arrows represent the edges in $E''$, and vertical solid arrow represent the edges in $E'''$.

example, a repetition $w[10..17]$ = baabaaba in Figure 1 is represented by the path which is a concatenation of the diagonal arrow from the black node at position 9 to a white node at position 15, and the horizontal arrows from its white node at position 15 to the upper white node at position 17.

### 3.2 Relations between repetition factorizations and the repetition graph

We explain a correspondence between a path on $G$ and a repetition factorization of $w$. For any integer $1 \leq t \leq n$, a path from node $v_s$ to node $(0,t) \in V'$ is called an *s-t path* of $G$. Let $F = f_1, \ldots, f_M$ be a repetition factorization of $w[1..t]$ and $(0,0) = v_1, \ldots, v_{M+1} = (0,t)$ be the sequence of nodes in $V'$ on an *s-t* path $\pi$. We will say that $F$ corresponds to $\pi$ (or $\pi$ corresponds to $F$), if the sequence $|f_1|, |f_1 f_2|, \ldots, |f_1 \cdots f_M| = t$ of the ending positions of all factors is equal to the sequence $pos(v_2), \ldots, pos(v_{M+1})$. The following lemma states a one-to-one correspondence between paths and repetition factorizations.

**Lemma 9.** *For any string $w$ of length $n$ and integer $1 \leq t \leq n$, there is a one-to-one correspondence between s-t paths on the repetition graph $G$ of $w$, and repetition factorizations of $w[1..t]$.*

*Proof.* First, we show that for any *s-t* path in $G$ there exists a unique repetition factorization of $w[1..t]$. Let $\pi = v_1 \cdots v_k$ be any *s-t* path, where $v_1 = v_s$ and $v_k = (0,t)$. By the definition of $G$, there is a unique decomposition $\pi_1, \ldots, \pi_{M+1}$ of path $\pi$ such that $\pi_i = (v_{j(i)} v_{j(i)+1}) \oplus (v_{j(i)+1} \cdots v_{j(i+1)-1})$, $v_{j(i)} \in V', v_{j(i)+1}, \ldots, v_{j(i+1)-1} \in V''$, and

$\pi_{M+1} = v_k$ (see also Figure 2). We remark that the subpath $(v_{j(i)+1} \cdots v_{j(i+1)-1})$ is possibly empty. By the definition of $G$, if $v_{j(i)}v_{j(i)+1}$ exists on $G$ (i.e. $(v_{j(i)}, v_{j(i)+1}) \in E'$), then $w[pos(v_{j(i)})+1..pos(v_{j(i)+1})]$ is a primitively rooted square of length $2p_{run(v_{j(i)+1})}$. If $v_{j(i)+1} \cdots v_{j(i+1)-1}$ exists on $G$, then $w[pos(v_{j(i)+1})+1..pos(v_{j(i+1)-1})]$ is a substring of a run $r = (beg, end, p)$ such that $beg + 2p \leq pos(v_{j(i)+1}) + 1 \leq pos(v_{j(i+1)-1}) \leq end$. Thus, repetition $w[pos(v_{j(i)})+1..pos(v_{j(i+1)-1})]$ has $p$ as its shortest period if $\pi_i$ exists on $G$. Therefore, $\pi$ corresponds to a unique repetition factorization of $w[1..t]$ since the path decomposition $\pi_1, \ldots, \pi_{M+1}$ is unique and $\pi_i$ corresponds to a repetition.

Second, we show that for any repetition factorization of $w[1..t]$, there exists a corresponding $s$-$t$ path in $G$. Let $F$ be any repetition factorization of $w[1..t]$. By the definition of $G$, there exists an $s$-$t$ path $\pi$ which corresponds to $F$. We show that $\pi$ is the only path on $G$ which corresponds to $F$. On the contrary, suppose that there are two distinct $s$-$t$ paths which correspond to some repetition factorization $F$ of $w[1..t]$. Because of this assumption, some factor $f_i = w[c..d]$ of $F$ with $1 \leq c < d \leq t$ corresponds to two distinct paths from $v_\gamma$ to $v_\delta$, where $pos(v_\gamma) = c$, $pos(v_\delta) = d$, and no nodes $v$ on the two paths satisfy $c < pos(v) < d$. This implies that the factor $f_i$ has two periods $p, q$ such that $p$ is its shortest period, $\gcd(p,q) \neq p$, and $2p, 2q \leq |f_i|$. Since $p < q$, we have $p + q - \gcd(p,q) \leq 2q \leq |f_i|$. By Lemma 1, $\gcd(p,q)$ is also a period of $f_i$. Since $p \neq \gcd(p,q)$ and $p < q$, we have $\gcd(p,q) < p$. However this contradicts that $p$ is the shortest period of $f_i$. Thus only one path can correspond to any repetition $f_i$ in $F$, and the path $\pi$ which corresponds to the repetition factorization $F$ is unique.

□

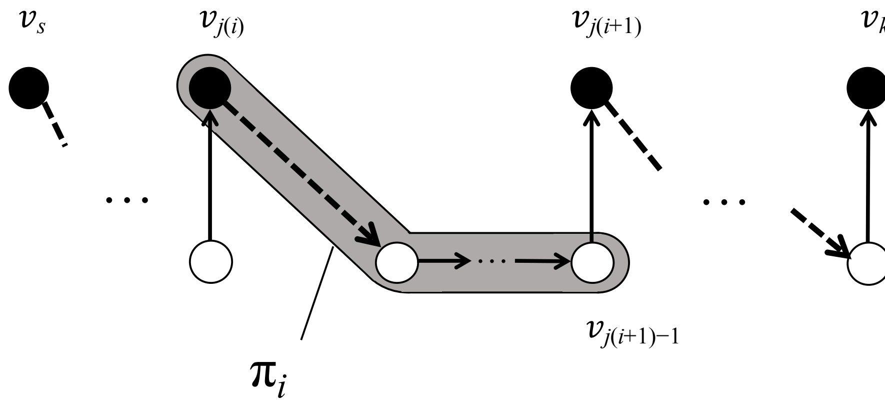



**Figure 2.** We can decompose an $s$-$t$ path so that each factor corresponds to a sub-path from a node in $V'$ to another node in $V'$.

We can regard each edge in $E'''$ as a boundary of a repetition factorization. Figure 3 and Figure 4 show examples of a correspondence between the paths and factorizations for the largest and smallest problem, respectively.

For any $1 \leq t \leq n$, let $\pi = v_s \cdots (0,t)$ be an an $s$-$t$ path on $G$. We denote by $RF_{w[1..t]}(\pi)$ the repetition factorization of $w[1..t]$ which corresponds to the $s$-$t$ path $\pi$. One can see that a path from $v_s$ to any node $v \in V''$ corresponds to a repetition factorization of $w[1..pos(v)]$ such that the shortest period of the rightmost factor is $p_{run(v)}$.

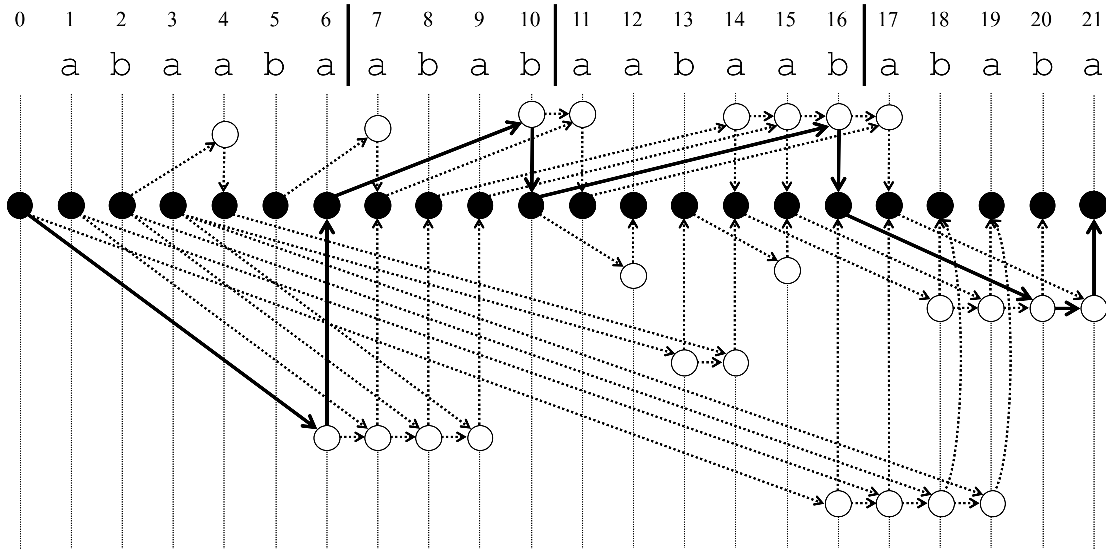



**Figure 3.** For the largest problem, one of answers is abaaba|abab|aabaab|ababa. The path consisting of bold arrows represents the $s$-$t$ path which corresponds to abaaba|abab|aabaab|ababa.

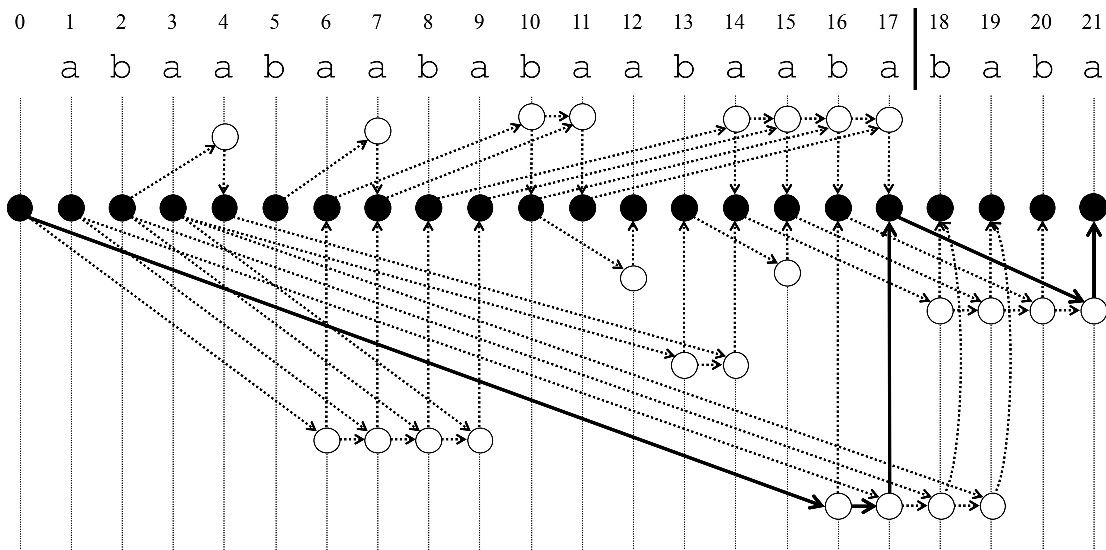



**Figure 4.** For the smallest problem, one of answers is abaabaababaabaaba|baba. The path consisting of bold arrows represents the $s$-$t$ path which corresponds to abaabaababaabaaba|baba.

### 3.3 Reduction to the longest path problem

Now we describe a reduction from the problem of computing a repetition factorization of a given string $w$ to the longest path problem on the repetition graph $G$ for $w$.

Recall that for any edge $e \in E$, we defined $weight(e) = 1$ if $e \in E'$ and $weight(e) = 0$ otherwise. Let $sum(\pi)$ be the sum of $weight(e)$ for all edges $e$ on the path $\pi$. The next lemma clearly holds.

**Lemma 10.** *For any $s$-$t$ path $\pi = v_s \cdots (0,t)$, $sum(\pi) = |RF_{w[1..t]}(\pi)|$.*

It immediately follows from Lemma 10 that if $\pi_{\max}$ is a path from $v_s$ to node $(0,t)$ of maximal total weight, then $sum(\pi_{\max})$ equals to the size of a largest repetition

factorization of $w[1..t]$. Thus, the problem of computing a largest repetition factorization of $w$ reduces to the longest path problem on the repetition graph $G$, which can be solved in $O(|V| + |E|)$ time and space.

### 3.4 Complexity

The efficiency of our algorithm depends on the size of repetition graph $G$. We show upper and lower bounds of the size of repetition graphs.

**Upper bound.** We firstly show an upper bound of the size of $G$. It is clear that $|V'| \leq n + 1$ by the definition. The number of nodes $v \in V''$ s.t. $pos(v) = i$ is equal to the number of primitively rooted squares which end at position $i$. By applying the next lemma to all positions in the input string $w$, we can get $|V''| = O(n \log n)$.

**Lemma 11 ([6]).** *For any string $x$ of length $i$, the number of primitively rooted squares that are suffixes of $x$ is $O(\log i)$.*

Overall, we get $|V| = O(n \log n)$.

Now we analyze the number of edges in the graph $G$. For any node $v \in V''$, the number of incoming edges of $v$ in $E'$ is exactly 1 and the number of outgoing edges of $v$ is at most 2 (i.e. exactly one edge in $E'''$ and at most one edge in $E''$). Thus we can see $|E| = O(|V|) = O(n \log n)$. Because of this argument, we can solve the longest path problem on $G$ in $O(n \log n)$ time after constructing $G$. In order to construct $G$, we need information of all runs. It follows from Theorem 5 that any string of length $n$ contains less than $n$ runs, and that all runs in a given string can be computed in linear time. Consequently, we obtain the following theorem.

**Theorem 12.** *Given a string $w$ of length $n$, we can compute a largest repetition factorization of $w$ in $O(n \log n)$ time and space.*

The following corollaries are immediate.

**Corollary 13.** *Given a string $w$ of length $n$, we can compute the size of a largest repetition factorization of $w[1..i]$ for every $1 \leq i \leq n$ in $O(n \log n)$ total time and space.*

**Corollary 14.** *Given a string $w$ of length $n$, we can compute the number of distinct repetition factorizations of $w$ in $O(n \log n)$ time and space.*

**Lower bound.** As was shown above, the size of the repetition graph $G$ is upper-bounded by $O(n \log n)$. This bound is indeed tight, namely, there exists a series of strings of which the repetition graphs contain $\Omega(n \log n)$ nodes and edges.

We consider the well-known *Fibonacci strings*:

**Definition 15 (Fibonacci string [16]).** *The $k$-th Fibonacci string $Fib_k$ is recursively defined as follows.*

- $Fib_1 = b$,
- $Fib_2 = a$,
- $Fib_k = Fib_{k-1} Fib_{k-2}$ *for* $k \geq 3$.

Clearly, $|Fib_k| = F_k$, where $F_k$ is the $k$-th Fibonacci number.

*Example 16.* $Fib_1 = b$, $Fib_2 = a$, $Fib_3 = ab$, $Fib_4 = aba$, $Fib_5 = abaab$, $Fib_6 = abaababa$, etc.

Fraenkel and Simpson [11] studied the number $R(k)$ of primitively rooted squares in $k$-th *Fibonacci string*. Let $\phi = \frac{1+\sqrt{5}}{2}$.

**Lemma 17 ([11]).** $R(k) = \frac{2}{5}(3-\phi)kF_k + O(F_k) = \frac{2(3-\phi)}{5\log\phi}F_k \log F_k + O(F_k)$ ($\frac{2(3-\phi)}{5\log\phi} \approx 0.7962$).

The next lemma immediately follows from Lemma 17.

**Lemma 18.** *For any Fibonacci string of length $n$, the size of $G$ is $\Theta(n \log n)$.*

# 4 $O(n \log n)$-time and $O(n)$-space solution

In this section, we propose an $O(n \log n)$-time and $O(n)$-space solution to Problem 7. This space-efficient algorithm follows the idea of the algorithm in the previous section, but does not explicitly construct the repetition graph $G$ which requires $O(n \log n)$ space. Instead, the space-efficient algorithm proposed in this section simulates the traversal on $G$ using only $O(n)$ space, by dynamic programming. In this section, we again focus on the largest repetition factorization problem, but the smallest version can also be solved analogously.

## 4.1 Simulating the previous algorithm

Each node $v$ of $G$ is assigned to a value $value(v)$, defined as follows: $value(v)$ is the size of a largest repetition factorization of $w[1..pos(v)]$ if $v \in V'$, $value(v)$ is the size of a largest repetition factorization of $w[1..pos(v)]$ s.t. the shortest period of the rightmost factor is $p_{run(v)}$ if $v \in V''$. It is easy to see that $value(v)$ is equal to the value for $v$ which is computed by the longest path problem on $G$. Thus, it suffices to compute $value(v)$ for all nodes $v \in V$. This can be done using the following formula. For any $v \in V'$, let $U_v$ be the set of nodes $u$ such that $(u, v) \in E$, and for any $v \in V''$, let $u_1$ and $u_2$ be the unique two nodes such that $(u_1, v) \in E$ and $u_1 \in V'$, and $(u_2, v) \in E$ and $u_2 \in V''$, respectively.

$$value(v) = \begin{cases} \max\{value(u) \mid u \in U_v\} & \text{if } v \in V', \quad (1) \\ \max\{value(u_1) + 1, value(u_2)\} & \text{if } v \in V''. \quad (2) \end{cases}$$

Let $RFA_w$ be an array of length $n$ s.t. $RFA_w[i]$ stores the size of a largest repetition factorization of $w[1..i]$, namely, $RFA_w[i] = value((0, i))$. Our new algorithm computes $RFA_w$ from left to right. We suppose that $RFA_w[1..i-1]$ is already computed. Below, we show how to compute $RFA_w[i]$.

Let $subRuns_w[i]$ be a set of indices $j$ s.t. $beg_j + 2p_j - 1 < i \leq end_j$ (i.e. a set of indices of runs which correspond to a node $v \in V''$ s.t. $pos(v) = i$). In order to compute $RFA_w[i]$, we need $value(v)$ s.t. $run(v) \in subRuns_w[i]$ due to Equation (1). We also suppose that for each $j' \in subRuns_w[i-1]$, the run $r_{j'}$ maintains $value(v)$ s.t. $run(v) = j'$ and $pos(v) = i-1$. For any $j \in subRuns_w[i]$, we can compute $value((j, i))$ by Equation (2) (since we know $value((j, i-1))$ and $RFA_w[i-2p_j]$). Thus we can compute $value((j, i))$ in $O(1)$ time for each $j \in subRuns_w[i]$. If we have $subRuns_w[i]$, then we can compute $RFA_w[i]$ in $O(\log n)$ time, since $|subRuns_w[i]| = O(\log n)$.

Finally, we show how to compute $subRuns_w[i]$. Let $I_b(i) = \{j \mid beg_j + 2p_j - 1 = i (1 \le j \le |Runs(w)|)\}$, and let $I_e(i) = \{j \mid end_j = i (1 \le j \le |Runs(w)|)\}$. We can compute $I_b(i)$ and $I_e(i)$ for all $i$ in $O(n)$ time and these sets takes $O(n)$ space. We assume that $subRuns_w[i-1]$ has already computed. It is easy to see that we can compute $subRuns_w[i]$ by removing $j$ s.t. $j \in I_e(i-1)$ from $subRuns_w[i-1]$ and by adding $j$ s.t. $j \in I_b(i)$ to $subRuns_w[i-1]$. From these operations, we can compute $subRuns_w[i]$ in $O(\log n)$ time if we have $I_b(i)$ and $I_e(i-1)$.

### 4.2 Complexity

First, we compute $Runs(w)$ and construct $I_b(i)$ and $I_e(i)$ for all $i$ in $O(n)$ time and these sets takes $O(n)$ space. This requires $O(n \log n)$ time and $O(n)$ space by Theorem 5. As was explained, for each position $i$ in $w$, $RFA_w[i]$ can be computed in $O(\log n)$ time, and $RFA_w$ requires a total of $O(n)$ space. We have proved the next theorem and corollaries.

**Theorem 19.** *Given a string $w$ of length $n$, we can compute a largest repetition factorization of $w$ in $O(n \log n)$ time and $O(n)$ space.*

**Corollary 20.** *Given a string $w$ of length $n$, we can compute the size of a largest repetition factorization of $w[1..i]$ for every $1 \le i \le n$ in $O(n \log n)$ total time and $O(n)$ total space.*

**Corollary 21.** *Given a string $w$ of length $n$, we can compute the number of distinct repetition factorizations of $w$ in $O(n \log n)$ time and $O(n)$ space.*

## 5 Conclusions and open question

We showed how to compute a smallest/largest repetition factorization of a given string $w$ in $O(n \log n)$ time, where $n$ is the length of $w$. The key idea is the reduction of the problems to the shortest/longest path problems on the repetition graph $G$, which is defined by the runs occurring in $w$. We first developed an algorithm which uses $O(n \log n)$ space, and showed that this space requirement is unavoidable if we explicitly construct $G$. We then showed how to simulate the first algorithm only with $O(n)$ space, by a dynamic programming approach.

Since there exists a string of length $n$ for which the repetition graph $G$ occupies $\Theta(n \log n)$ space, further speed-up seems difficult to achieve as long as we use the repetition graph $G$ implicitly or explicitly. Thus, an intriguing open question is whether there exists an efficient algorithm which computes repetition factorizations of smallest/largest size without relying on the repetition graph $G$.

## References


1. G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto: *Closed factorization*, in Proc. PSC 2014, 2014, pp. 162–168.
2. H. Bannai, T. Gagie, S. Inenaga, J. Kärkkäinen, D. Kempa, M. Piatkowski, S. J. Puglisi, and S. Sugimoto: *Diverse palindromic factorization is np-complete*, in Proc. DLT 2015, 2015, pp. 85–96.
3. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. iv. the quotient groups of the lower central series*. Annals of Mathematics, 68(1) 1958, pp. 81–95.

4. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications.* Inf. Process. Lett., 106(2) 2008, pp. 75–80.
5. M. Crochemore, L. Ilie, and W. F. Smyth: *A simple algorithm for computing the Lempel Ziv factorization*, in Proc. DCC 2008, 2008, pp. 482–488.
6. M. Crochemore and W. Rytter: *Sqares, cubes, and time-space efficient string searching.* Algorithmica, 13(5) 1995, pp. 405–425.
7. M. Dumitran, F. Manea, and D. Nowotka: *On prefix/suffix-square free words*, in Proc. SPIRE, 2015, pp. 54–66.
8. J. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
9. G. Fici, T. Gagie, J. Kärkkäinen, and D. Kempa: *A subquadratic algorithm for minimum palindromic factorization.* J. Discrete Algorithms, 28 2014, pp. 41–48.
10. N. J. Fine and H. S. Wilf: *Uniqueness theorems for periodic functions.* Proceedings of American Mathematical Society, 16(1) 1965, pp. 109–114.
11. A. S. Fraenkel and J. Simpson: *The exact number of squares in fibonacci words.* Theor. Comput. Sci., 218(1) 1999, pp. 95–106.
12. T. I, S. Sugimoto, S. Inenaga, H. Bannai, and M. Takeda: *Computing palindromic factorizations and palindromic covers on-line*, in Proc. CPM 2014, 2014, pp. 150–161.
13. R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov: *Searching of gapped repeats and subrepetitions in a word*, in Proc. CPM 2014, 2014, pp. 212–221.
14. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. FOCS 1999, 1999, pp. 596–604.
15. M. Kufleitner: *On bijective variants of the Burrows-Wheeler transform*, in Proc. PSC 2009, 2009, pp. 65–79.
16. M. Lothaire: *Combinatorics on Words*, Addison-Wesley, 1983.
17. Y. Matsuoka, S. Inenaga, H. Bannai, M. Takeda, and F. Manea: *Factorizing a string into squares in linear time*, in 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel, 2016, pp. 27:1–27:12.
18. Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Constructing LZ78 tries and position heaps in linear time for large alphabets.* Inf. Process. Lett., 115(9) 2015, pp. 655–659.
19. J. Storer and T. Szymanski: *Data compression via textual substitution.* J. ACM, 29(4) 1982, pp. 928–951.
20. Y. Tanimura, Y. Fujishige, T. I, S. Inenaga, H. Bannai, and M. Takeda: *A faster algorithm for computing maximal $\alpha$-gapped repeats in a string*, in Proc. SPIRE 2015, 2015, pp. 124–136.
21. T. A. Welch: *A technique for high performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
22. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.
23. J. Ziv and A. Lempel: *Compression of individual sequences via variable-length coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.

# Computing All Approximate Enhanced Covers with the Hamming Distance

Ondřej Guth

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
ondrej.guth@fit.cvut.cz

**Abstract.** A border $\boldsymbol{p}$ of a string $\boldsymbol{x}$ is an enhanced cover of $\boldsymbol{x}$ if the number of positions of $\boldsymbol{x}$ that lie within some occurrence of $\boldsymbol{p}$ is the maximum among all borders of $\boldsymbol{x}$. In this paper, more general notion based on the enhanced cover is introduced: a $k$-approximate enhanced cover, where fixed maximum number of errors $k$ in the Hamming distance is considered. The $k$-approximate enhanced cover of $\boldsymbol{x}$ is its border and its $k$-approximate occurrences are also considered in the covered number of positions of $\boldsymbol{x}$. An $\mathcal{O}(n^2)$-time and a $\mathcal{O}(n)$-space algorithm that computes all $k$-approximate enhanced covers of a string of length $n$ is presented.

**Keywords:** string regularity, approximate cover, enhanced cover, quasiperiodicity, suffix automaton, Hamming distance, border

## 1 Introduction

Searching repetitive structures of strings, so-called regularities of strings, has been intensively studied for many years in many fields of computer science, e.g., combinatorics on strings, pattern matching, data compression and molecular biology, and many related notions have been introduced: periods, squares, covers, seeds, etc. [10] Those long-time known repetitive structures provide compact description of a string. However, they are quite restrictive and it is rare that an arbitrary string has a non-trivial regularity of that kind (e.g., not every string has a cover shorter than itself). Therefore, there have been attempts to introduce more relaxed repetitive structures, e.g., their approximate versions. Quite recently, a term of *enhanced cover* [3] has been introduced; every string having a (non-empty) border has also an enhanced cover.

In order to provide even more general notion, the enhanced cover is extended to its approximate version in this paper, see Fig. 1. The problem of computing all $k$-approximate enhanced covers of a string with fixed maximum Hamming distance is solved using finite automata in a way which is easy to implement and understand and also consistent with finite automata based algorithms for similar problems. There is no other known solution of this problem.

This paper is organised as follows. Section 1.1 contains definitions of terms used through the text and also definition of the problem solved in this paper; in the section, previous and related work is also summarized. In Section 2, the algorithm solving the stated problem is presented; it starts with a description of a basic idea found in another paper, the algorithm is then described in words and also its pseudocode is shown; the time and space complexity is then stated and proved. In Section 3, a behaviour of an implementation of the presented algorithm is shown, depending on various input parameters.

Ondřej Guth: Computing All Approximate Enhanced Covers with the Hamming Distance, pp. 146–157.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

border a: abacaccababa
border aba: abacaccababa
cover aba (not possible): abacaccababa
1-approximate cover aba (not possible): abacaccababa
2-approximate cover aba: abacaccababa
2-approximate cover ababa: abacaccababa
enhanced cover aba (8 covered positions): abacaccababa
1-approximate enhanced cover (10 covered positions): abacaccababa

**Figure 1.** Regularities of the string $\boldsymbol{x} =$ abacaccababa

### 1.1 Preliminaries

An *alphabet* is a finite set of symbols, denoted by $A$. A *string* $\boldsymbol{x}$ over alphabet $A$ is a finite sequence of symbols of $A$, denoted by $\boldsymbol{x} \in A^*$. An empty string is denoted by $\varepsilon$. An $i$-th symbol of a string $\boldsymbol{x}$ is denoted by $\boldsymbol{x}[i]$, i.e., the first symbol of $\boldsymbol{x}$ is denoted by $\boldsymbol{x}[1]$. A substring of $\boldsymbol{x}$ starting at an $i$-th and ending at an $j$-th symbol is denoted by $\boldsymbol{x}[i..j]$, i.e., $\boldsymbol{x} = \boldsymbol{x}[1..|\boldsymbol{x}|]$. Assuming strings $\boldsymbol{p}, \boldsymbol{s}, \boldsymbol{u}, \boldsymbol{x} \in A^*$, where $\boldsymbol{x} = \boldsymbol{pus}$, the string $\boldsymbol{p}$ is a *prefix* of $\boldsymbol{x}$, the string $\boldsymbol{s}$ is a *suffix* of $\boldsymbol{x}$, and the string $\boldsymbol{u}$ is a *factor* (also known as a substring) of $\boldsymbol{x}$. An editing operation *replace* in a string $\boldsymbol{x} \in A^*$ is replacing a symbol $\boldsymbol{x}[i]$ with another symbol of $A$. Assuming strings $\boldsymbol{x}, \boldsymbol{y} \in A^*$ such that $|\boldsymbol{x}| = |\boldsymbol{y}|$, the *Hamming distance* of the strings $\boldsymbol{x}, \boldsymbol{y}$, denoted by $\mathsf{H}(\boldsymbol{x}, \boldsymbol{y})$, is the minimum number of operations replace necessary to convert $\boldsymbol{x}$ to $\boldsymbol{y}$. Assuming strings $\boldsymbol{p}, \boldsymbol{s}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in A^*$ and an integer $k \geq 0$, the string $\boldsymbol{v}$ is a *$k$-approximate factor* of the string $\boldsymbol{w}$ if $\boldsymbol{w}$ may be written as $\boldsymbol{pus}$ and $\mathsf{H}(\boldsymbol{u}, \boldsymbol{v}) \leq k$. The string $\boldsymbol{u}$ *has an occurrence* in the string $\boldsymbol{w}$ if $\boldsymbol{u}$ is a factor of $\boldsymbol{w}$. A factor $\boldsymbol{u}$ of $\boldsymbol{w}$ *occurs at position* $i$ (that is also called an *end position*) in the string $\boldsymbol{w}$ if for all $j \in \{1, \ldots, |\boldsymbol{u}|\}$ it holds that $\boldsymbol{u}[j] = \boldsymbol{w}[i - |\boldsymbol{u}| + j]$. A position $l$ of a string $\boldsymbol{w}$ *lies within some occurrence* of $\boldsymbol{u}$ in $\boldsymbol{w}$ if $\boldsymbol{u}$ occurs at a position $i$ in $\boldsymbol{w}$ and $i - |\boldsymbol{u}| < l \leq i$. A $k$-approximate factor $\boldsymbol{v}$ of $\boldsymbol{w}$ *$k$-approximately occurs at position* $i$ (that is also called a *$k$-approximate end position*) if there exists a factor $\boldsymbol{u}$ of $\boldsymbol{w}$ that occurs at the position $i$ in $\boldsymbol{w}$ and $\mathsf{H}(\boldsymbol{u}, \boldsymbol{v}) \leq k$. A position $l$ of a string $\boldsymbol{w}$ *lies within some $k$-approximate occurrence* of $\boldsymbol{v}$ in $\boldsymbol{w}$ if $\boldsymbol{v}$ $k$-approximately occurs at a position $i$ in $\boldsymbol{w}$ and $i - |\boldsymbol{v}| < l \leq i$.

A *border* of a string $\boldsymbol{x}$ is simultaneously a prefix and a suffix of $\boldsymbol{x}$. A string $\boldsymbol{w}$ is a *cover* of $\boldsymbol{x}$ if every position of $\boldsymbol{x}$ lies within some occurrence of $\boldsymbol{w}$ in $\boldsymbol{x}$. A $\boldsymbol{w}$ is a *$k$-approximate cover* of $\boldsymbol{x}$ if $\boldsymbol{w}$ is a factor of $\boldsymbol{x}$ and every position of $\boldsymbol{x}$ lies within some $k$-approximate occurrence of $\boldsymbol{w}$ in $\boldsymbol{x}$. A border $\boldsymbol{u}$ of a string $\boldsymbol{y}$ is an *enhanced cover* of $\boldsymbol{y}$ if the number of positions of $\boldsymbol{y}$ which lie within some occurrence of $\boldsymbol{u}$ in $\boldsymbol{y}$ is the maximum among all borders of $\boldsymbol{y}$ [3].

A *deterministic finite automaton* $\mathcal{M}$ is a quintuple $(Q, A, \delta, q_0, F)$ where

- $Q$ is a nonempty finite set of states,
- $A$ is a nonempty finite input alphabet,
- $\delta : Q \times A \mapsto Q$ is a transition function (partially defined, i.e., for some pair $(q, a)$, where $q \in Q, a \in A$, is $\delta(q, a)$ undefined),
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

An *extended transition function* of a deterministic automaton $\mathcal{M} = (Q, A, \delta, q_0, F)$ is denoted by $\delta^*$ and it is defined for $q \in Q, a \in A, \boldsymbol{u} \in A^*$ inductively: $\delta^*(q, \varepsilon) = q$, $\delta^*(q, \boldsymbol{u}a) = \delta(\delta^*(q, \boldsymbol{u}), a)$. String $\boldsymbol{w}$ is *accepted* by $\mathcal{M}$ if and only if $\delta^*(q_0, \boldsymbol{w}) \in F$.

An automaton $\mathcal{M}$ *accepts a set of strings* $B$ if and only if for all $\boldsymbol{u} \in B$ holds that $\boldsymbol{u}$ is accepted by $\mathcal{M}$. A *nondeterministic finite automaton* $\mathcal{M}_{\mathrm{N}}$ is a quintuple $(Q, A, \delta, q_0, F)$ where

- $Q$ is a nonempty finite set of states,
- $A$ is a nonempty finite input alphabet,
- $\delta : Q \times A \mapsto \mathcal{P}(Q)$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

An *extended transition function* of a nondeterministic finite automaton $\mathcal{M}_{\mathrm{N}}$ is denoted by $\delta^*$ and it is defined for $q_1, q_2 \in Q, a \in A, \boldsymbol{u} \in A^*$ inductively: $\delta^*(q_1, \varepsilon) = \{q_1\}$, $\delta^*(q_1, \boldsymbol{u}a) = \bigcup_{q_2 \in \delta^*(q_1, \boldsymbol{u})} \delta(q_2, a)$. A *finite automaton* (also known as a finite state machine) is either a deterministic or a nondeterministic finite automaton. A *suffix automaton* for a string $\boldsymbol{u}$ is a finite automaton that accepts a set of all suffixes of $\boldsymbol{u}$. Let us have a set $S$ of $k$-approximate suffixes of a string $\boldsymbol{u}$ defined as: $\boldsymbol{v} \in S$ if and only if for all suffixes $\boldsymbol{s}$ of $\boldsymbol{u}$, $\mathsf{H}(\boldsymbol{s}, \boldsymbol{v}) \leq k$; a *$k$-approximate suffix automaton* for the string $\boldsymbol{u}$ is a finite automaton that accepts a set of $k$-approximate suffixes $S$ of $\boldsymbol{u}$. A *d-subset* of a state of a deterministic finite automaton is an ordered set of *elements*. Each element $e$ is represented by two integers: $\mathsf{depth}(e)$ and $\mathsf{level}(e)$, where $\mathsf{depth}(e)$ corresponds to an end position and $\mathsf{level}(e)$ represents the Hamming distance of some factor of $\boldsymbol{u}$.

**Definition 1 ($k$-approximate enhanced cover).** *A string $\boldsymbol{w}$ is a $k$-approximate enhanced cover of a string $\boldsymbol{x}$ if $\boldsymbol{w}$ is a border of $\boldsymbol{x}$ and the number of positions of $\boldsymbol{x}$ which lie within some $k$-approximate occurrence of $\boldsymbol{w}$ in $\boldsymbol{x}$ is the maximum among all borders of $\boldsymbol{x}$.*

See an example of a $k$-approximate enhanced cover in Fig. 1.

*Problem definition* Given a string $\boldsymbol{w}$ and an integer $k$, the problem of *computing all $k$-approximate enhanced covers of $\boldsymbol{w}$* is to find all borders of $\boldsymbol{w}$ that satisfy Definition 1.

**Related Work.** The idea of a quasipediodic string (i.e., a string having a cover) was introduced by Apostolico and Ehrenfeucht [1], Moore and Smyth gave a linear-time algorithm for computing all covers of a given string [6,7]. An algorithm for computing all covers in generalized strings based on a suffix automaton was introduced by Voráček and Melichar [11].

Computing approximate covers was introduced by Sim et al. [9]. Christodoulakis et al. [2] implemented the algorithm based on dynamic programming and showed its practical time complexity for Hamming, edit and weighted edit distance. Guth, Melichar, and Balík [4] gave an algorithm for computing all approximate covers with the Hamming distance based on a suffix automaton.

In 2013, Flouri et al. [3] introduced a notion of the *enhanced cover* and gave a linear time algorithm for computing the minimum enhanced cover of a given string.

## 2 Problem Solution

### 2.1 Basic Idea

The presented solution of the problem of computing all $k$-approximate enhanced covers of a given string is based on the algorithm for computing all $k$-approximate

covers [4]. The referenced algorithm works in two phases: find candidate factors and compute the smallest Hamming distance for each candidate to cover the given string. To find the candidate factors, a subset construction [8][5, Alg. 1.40] of a deterministic $k$-approximate suffix automaton for the Hamming distance is used. This way, positions of each $k$-approximate occurrence of each factor of the given string is obtained. In the second phase, each factor is checked, whether it $k$-approximately covers the given string. To do that, subsequent positions (obtained by the subset construction) are compared with the factor length – there must be no gap between subsequent $k$-approximate occurrences of the factor. In order to reduce space complexity, only part of the deterministic automaton is stored in a memory – a depth-first search is done and all unnecessary states are removed.

The above mentioned algorithm [4] is used to solve the problem of computing all $k$-approximate enhanced covers after some modifications.

The idea found in [5, Section 4] and used in [4] is to use a nondeterministic $k$-approximate suffix automaton $\mathcal{M}_{\mathrm{N}}$ for a string $\boldsymbol{x}$ as an indexing structure. This automaton accepts all $k$-approximate suffixes of $\boldsymbol{x}$. Moreover, every string $\boldsymbol{u}$ "read" by $\mathcal{M}_{\mathrm{N}}$ reaches a set $B$ of states, i.e. $\delta^*(q_0, \boldsymbol{u}) = B$. With the proper labelling, depth $i$ of each such state $q \in B$ is equal to a $k$-approximate end position of $\boldsymbol{u}$ in $\boldsymbol{x}$, and level $j$ of $q$ is the minimum Hamming distance such that $i$ is a $j$-approximate end position of $\boldsymbol{u}$ in $\boldsymbol{x}$ and there exists no $l < j$ such that $i$ is an $l$-approximate end position of $\boldsymbol{u}$ in $\boldsymbol{x}$. Therefore, in addition to accept all $k$-approximate suffixes of $\boldsymbol{x}$, $\mathcal{M}_{\mathrm{N}}$ is able to identify all $k$-approximate end positions of all $k$-approximate factors of $\boldsymbol{x}$. See an example of a nondeterministic $k$-approximate suffix automaton in Fig. 2.

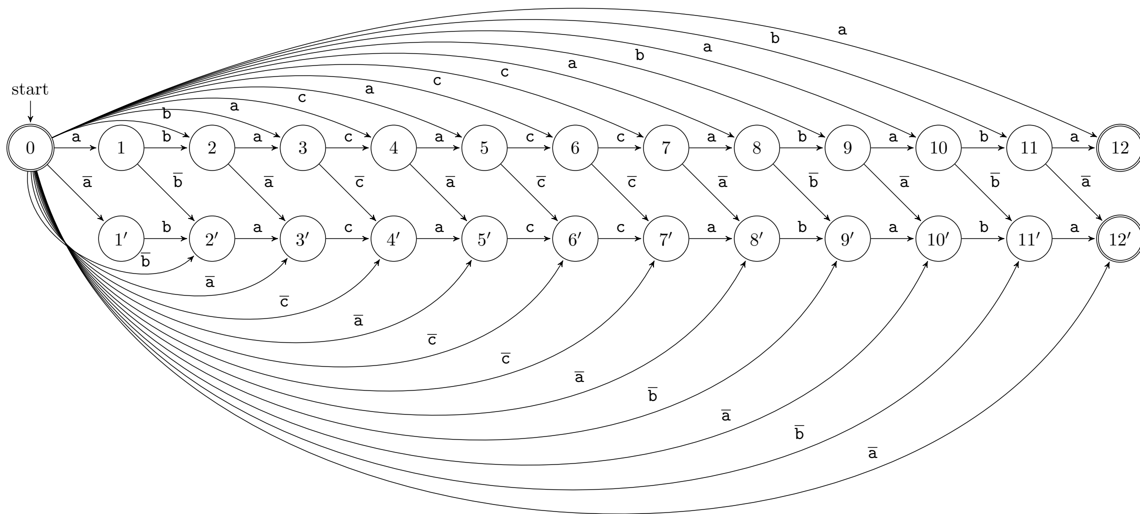



**Figure 2.** Example of a nondeterministic $k$-approximate suffix automaton for the string `abacaccababa` $\in A^*$ ($\overline{\mathtt{a}}$ denotes supplement, i.e. $\overline{\mathtt{a}} = A \setminus \{\mathtt{a}\}$; a state depth is denoted by an integer, a state level is denoted by the number of primes, a final state is denoted by a double circle)

To obtain a $k$-approximate deterministic suffix automaton $\mathcal{M}$, the subset construction [5,8] may be used. Instead of the subset construction, similar algorithm

that represents states of $\mathcal{M}$ as subsets of $\mathcal{M}_{\mathrm{N}}$ with preserved depths and levels, is used in [4]. An advantage of the deterministic $k$-approximate suffix automaton for $\boldsymbol{x}$ over the nondeterministic one is that processing a string $\boldsymbol{u}$ using $\mathcal{M}$ takes linear time in the length of $\boldsymbol{u}$, regardless of the length of $\boldsymbol{x}$.

Note that in the algorithm presented in this paper, the nondeterministic automaton is not constructed, it is used just to describe the concept. Instead, states of the deterministic automaton are constructed directly, utilising the knowledge of the regular structure of $\mathcal{M}_{\mathrm{N}}$.

### 2.2 The Algorithm

From the definition of a $k$-approximate enhanced cover for a given string $\boldsymbol{x}$ follows that every $k$-approximate enhanced cover of $\boldsymbol{x}$ is a border (exact) of $\boldsymbol{x}$. Every border of $\boldsymbol{x}$ is accepted by a $k$-approximate suffix automaton for $\boldsymbol{x}$ and even by its part, a *backbone*.

**Definition 2 (Backbone).** *[5, Def. 3.12 and Sec. 3.4.1] Assume a $k$-approximate deterministic suffix automaton $\mathcal{M} = (Q, A, \delta, q_0, F)$ for a string $\boldsymbol{x}$. A* backbone *of $\mathcal{M}$ is a deterministic automaton $\mathcal{M}_B = (Q_B, A, \delta_B, q_0, F_B)$ such that for all $0 < i \leq |\boldsymbol{x}|$ holds $Q_B = \{q_i : q_i \in Q\}$, $\delta_B(q_{i-1}, \boldsymbol{x}[i]) = q_i$, and $F_B = \{q : q \in Q_B \cap F\}$.*

In other words, the backbone is the part of $\mathcal{M}$ that enables "reading" of $\boldsymbol{x}$ (and of all its prefixes) exactly. See an example of a backbone in Fig. 3.

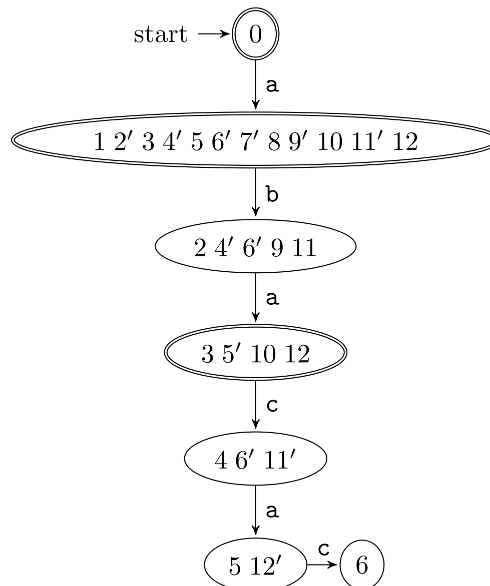



**Figure 3.** An example of a backbone of a deterministic $k$-approximate suffix automaton for the string `abacaccababa` – the part useful for computing borders (the d-subset element depth is denoted by an integer, the level is denoted by number of primes, a final state is denoted by a double circle)

In order to find $k$-approximate enhanced covers of $\boldsymbol{x}$ among its borders, the number of symbols of $\boldsymbol{x}$ that lie within some $k$-approximate occurrence of the border must be computed. This may be obtained from $k$-approximate occurrences of the border by summing the letters that lie within each occurrence, counting each letter only once. Considering each two subsequent $k$-approximate positions $i, j; i < j$ of a border $\boldsymbol{p}$ of $\boldsymbol{x}$, there are three cases:

```
    Input  : A state q of a deterministic k-approximate suffix automaton for x
    Output: The number of letters of x covered by a border corresponding to q
 1  begin
 2  |   r ← 0;
 3  |   e_f is the first d-subset element of q (one with the minimum depth);
 4  |   m ← depth(e_f);
 5  |   E is an array of d-subset elements of q having the same order as they are appended;
 6  |   for i ∈ 2..|E| do
 7  |   |   if depth(E[i]) − depth(E[i − 1]) < m then
 8  |   |   |   r ← r + depth(E[i]) − depth(E[i − 1]);
 9  |   |   else
10  |   |   |   r ← r + m;
11  |   |   end
12  |   end
13  |   return r;
14  end
```

**Function** `distEnhCov`

**overlap** $(j - i < |\boldsymbol{p}|)$ add $j - i$ to the number of letters covered by $\boldsymbol{p}$
**square** $(j - i = |\boldsymbol{p}|)$ add $|\boldsymbol{p}|$ to the number of letters covered by $\boldsymbol{p}$
**gap** $(j - i > |\boldsymbol{p}|)$ add $|\boldsymbol{p}|$ to the number of letters covered by $\boldsymbol{p}$

The pseudocode of this computation is listed as Function `distEnhCov`. All the $k$-approximate occurrences of each border of $\boldsymbol{x}$ are obtained from d-subsets of the backbone of the deterministic $k$-approximate suffix automaton for $\boldsymbol{x}$.

As in the algorithm for computing all $k$-approximate covers [4], the number of covered symbols of $\boldsymbol{x}$ for each of its borders is computed just after the state of the backbone is constructed. This state may be removed just after the next state is constructed, therefore at most two states are needed to be stored at a time. Unlike in the algorithm in [4], all the borders with the maximum number of covered symbols must be stored along with their number of covered symbols, because it is unknown what the number is, before the algorithm finishes.

In order to further reduce space complexity, the borders with the maximum number of covered symbols are not actually stored directly. Because every $k$-approximate enhanced cover is a prefix, the length is enough to specify it and therefore only the prefix length is stored and reported (variable $p$ in Algorithm 1).

*Example 3 (Computing all $k$-approximate enhanced covers).* Let us have a string $\boldsymbol{x}$ = `abacacaccababa` and maximum Hamming distance $k = 1$. The set of all 1-approximate enhanced covers of $\boldsymbol{x}$ is computed using Alg. 1. A d-subset of the first state $q_1$ of the backbone (see Fig. 3) is 1 2′ 3 4′ 5 6′ 7′ 8 9′ 10 11′ 12. Because the related prefix length is $p = 1$ and $k = 1$, no meaningful result may be obtained for this state. A d-subset of the second constructed state is 2 4′ 6′ 9 11. After its construction, $q_1$ and its d-subset are removed from a memory. Because depth of the last d-subset element is 11, it is not a final state (and does not represent a border of $\boldsymbol{x}$). A d-subset of the next constructed state is 3 5′ 10 12. Again, the previous state is now removed. Because the depth of the last element is 12 (equal to the length of $\boldsymbol{x}$) and its level is 0, the related prefix `aba` is a border of $\boldsymbol{x}$. The number of positions of $\boldsymbol{x}$ covered by 1-approximate occurrences of `aba` in $\boldsymbol{x}$ is now computed. The end positions (read from the d-subset) are 3, 5, 10, 12 and therefore the number of covered positions is 10. This is the maximum, the variable $h$ is updated and `aba` is added to the set $C$. The subsequent backbone state is not final and the next state 5 12′ is not final as well

```
    Input  : A string x, the maximum Hamming distance k
    Output: A set C of k-approximate enhanced covers of x (border lengths)
 1  begin
 2      C ← ∅;
 3      h ← 0;
 4      q_1 is a state;
 5      for i ∈ 1..|x|;                  // construct a d-subset of the first state
 6      do
 7          e is a d-subset element such that depth(e) ← i;
 8          if x[1] = x[i] then
 9              level(e) ← 0;
10              append e to q_1
11          else if k > 0 then
12              level(e) ← 1;
13              append e to q_1
14          end
15      end
16      p ← 1;                                                  // a prefix length
17      q_p ← q_1;
18      for i ∈ 2..|x| do
19          p ← p + 1;
20          q_n is a state;
21          for e_p ∈ q_p;                 // construct a d-subset of the next state
22          do
23              if depth(e_p) < |x| then
24                  e_n is a d-subset element such that depth(e_n) ← depth(e_p) + 1;
25                  if x[i] = x[depth(e_n)] then
26                      level(e_n) ← level(e_p);
27                      append e_n to q_n;
28                  else if level(e_p) < k then
29                      level(e_n) ← level(e_p) + 1;
30                      append e_n to q_n;
31                  end
32              end
33          end
34          destroy q_p;
35          if number of d-subset elements of q_n is less than 2 then
36              stop ;                                // all borders of x are examined
37          end
38          e_l is the last d-subset element of q_n;
39          if depth(e_l) = |x| and level(e_l) = 0 and |p| > k;          // q_n is final
40          then
41              h_n ← distEnhCov (q_n);
42              if h_n > h then
43                  h ← h_n;
44                  C ← ∅;
45              end
46              if h_n = h then
47                  append p to C;
48              end
49          end
50      end
51  end
```

**Algorithm 1:** Computing all $k$-approximate enhanced covers

(although the last element depth is 12, its level is 1, i.e. it represents an approximate occurrence and therefore not an exact border of $\boldsymbol{x}$). The subsequent state d-subset contains only one element and therefore cannot represent a border (it represents a string that occurs only once in $\boldsymbol{x}$). It is not needed to construct any further state, as the construction starting at line 21 cannot create a state with multiple d-subset elements.

### 2.3 Time and Space Complexity

**Theorem 4 (Space complexity).** *Given a string $\boldsymbol{x}$, $|\boldsymbol{x}| = n$, and an integer $k$, Algorithm 1 needs at most $4n$ space.*

*Proof.* The following is needed to be stored in a memory: the input string $\boldsymbol{x}$, the set of results $C$, a length $p$ of actual prefix, states and d-subsets. A state is represented by its d-subset. Every d-subset is an array of elements and its size is always known. Each d-subset element is represented by two integers, every integer used in this algorithm is between 0 and $\max(n, k)$.

The for loop starting at line 6 is iterated $n$ times. Within each iteration, at most one d-subset element is added, therefore $q_1$ needs at most $2n + 1$ space.

During construction of a next state $q_n$ (starting at line 21), for each d-subset element of $q_p$ (there are at most $n$), a new d-subset element is constructed. The new element is not stored in two cases only: exceeding the maximum level $k$, or the depth of the element over the length of $\boldsymbol{x}$ (checked at line 23). Therefore the maximum number of d-subset elements of a state are: $n$ for a first state, $n - 1$ for a second state, etc. Due to the line 34, at most two states are in a memory at a time, therefore d-subset elements need at most $4n$ space. For some states, the integer $p$ is added to $C$, so total maximum size of $C$ is $n$. □

**Theorem 5 (Time complexity).** *Given a string $\boldsymbol{x}$, $|\boldsymbol{x}| = n$, and an integer $k$, Alg. 1 needs at most $\mathcal{O}(n^2)$ time.*

*Proof.* The for loop starting at the line 6 needs $\mathcal{O}(n)$ time. The for loop starting at the line 21 needs $\mathcal{O}(|q_p|)$ time (see the proof of Theorem 4). All the statements whithin this loop are evaluated in constant time (a d-subset is an array, each element is an object with two associated integers). Therefore the evaluation of the for loop (the line 21) takes $\sum_{i=2}^{n} n - i + 1 = \frac{n^2 - n}{2}$. The line 34 takes the same time as the above for loop. Computing the Function `distEnhCov` takes at most $5(n - 2)$ for the second state (recall the d-subset size in the proof of Theorem 4), so it takes $\frac{5n^2 - 15n}{2}$ for the whole input. □

*Example 6.* For the input string $\boldsymbol{x} = \texttt{aa} \cdots \texttt{a}$, quadratic time regarding $|\boldsymbol{x}|$ is needed for Alg. 1.

## 3 Experimental Results

The algorithm has been implemented using the C++ programming language. It has been compiled using the gcc 5.3.0 with the O3 optimisation level, and run on the i5-2520M (4-core) machine under the Hardened Gentoo Linux 4.3.5 with disabled swap. As input data, Saccharomyces cerevisiae S288c chromosome IV[1] was used. For various

[1] The sequence was downloaded from `http://www.ncbi.nlm.nih.gov/nuccore/NC_001136.10`.

input lengths, the input string consists the of first $n$ characters of the chromosome. For each input length $n$ and the maximum Hamming distance $k$, the following values were measured using the GNU time utility:

- elapsed time as total number of CPU-seconds that the implemented program spent in user mode,
- memory consumption as maximum resident set size of the implemented program.

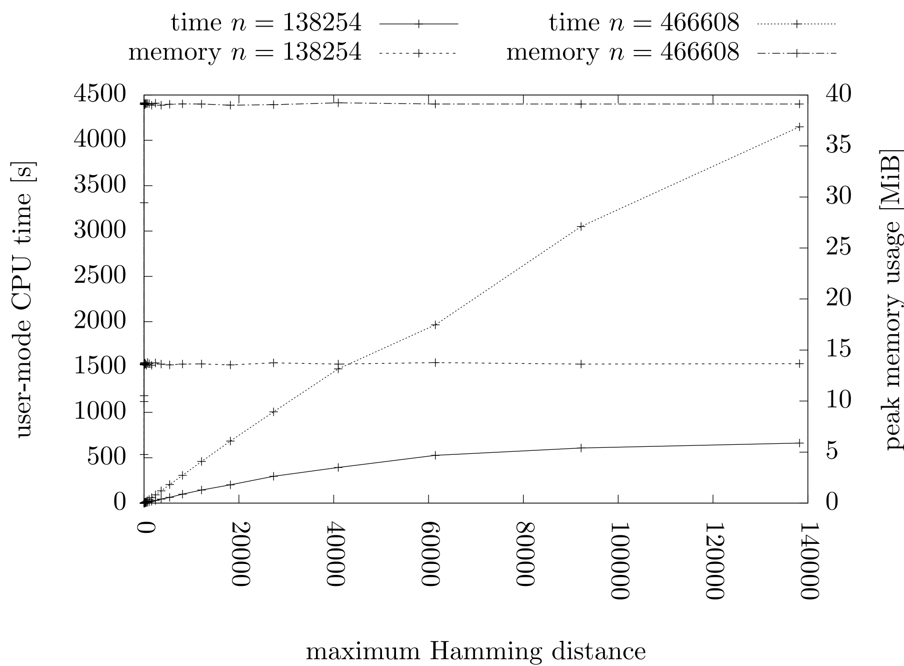



**Figure 4.** Time and memory consumption, when processing the chromosome, depending on the maximum distance $k$ when the input length $n$ is fixed

Both the time and memory consumption is shown in Figs. 4 and 5. Values of the time consumption are always shown in seconds at the left border of each figure, values of memory consumption are shown in megabytes at the right border of each figure. It is distinguished by a line type whether the time or the memory consumption is being plotted. Because the consumption depends on both the input length and the maximum allowed distance, the distance is fixed in the plots shown in Fig. 5 and the input length is fixed in the plot shown in Fig. 4. The value of the fixed length, or the distance, respectively, is shown in a key of each figure, and distinguished by a line type.

In the plots shown in Fig. 5, the time and memory consumption is shown depending on varying input string length when the maximum allowed Hamming distance is fixed to a few arbitrary values.

Note that although the time and space complexity (Theorems 4 and 5) do not depend on $k$, the real consumption is varying for different $k$. The reason is that the complexities in the above theorems are maximal, however, for the data used in the experiment, $k$ is limiting the number of d-subset elements. The limiting effect of $k$ is better shown in Fig. 4. In the plots shown in this figure, the consumption is shown depending on varying maximum allowed Hamming distance while the input string length is fixed to a few arbitrary values.

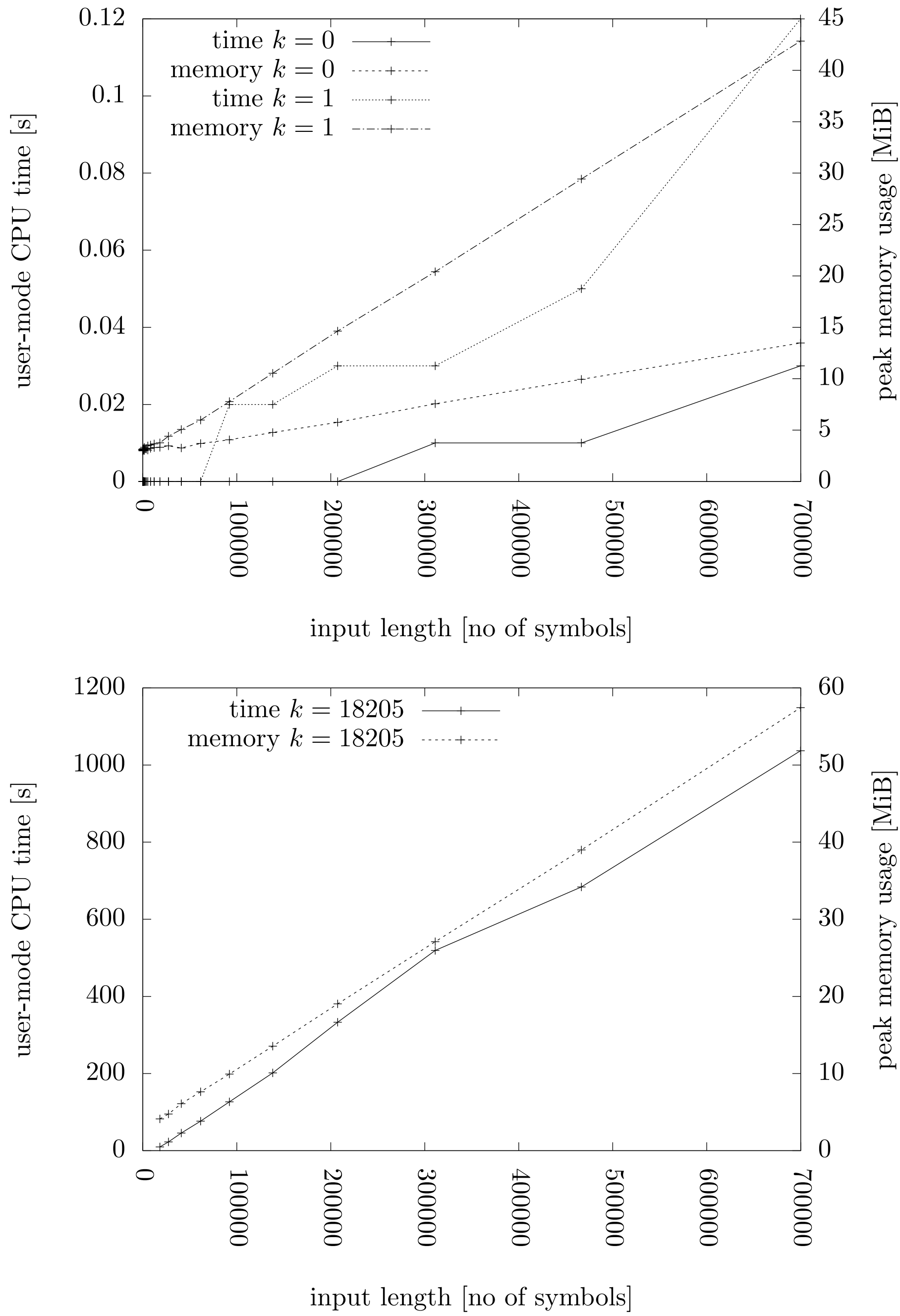



**Figure 5.** Time and memory consumption, when processing the chromosome, depending the on string size when the maximum Hamming distance $k$ is fixed

The plots in Fig. 5 seem to show linear growth of the time consumption depending on the input string length. This is due to the input data (small number of repeating factors). With a regular input string, e.g. `ab` repeating many times, time needed for processing the string is apparently quadratic (see Fig. 6) with respect to the input string length (according to Theorem 5).

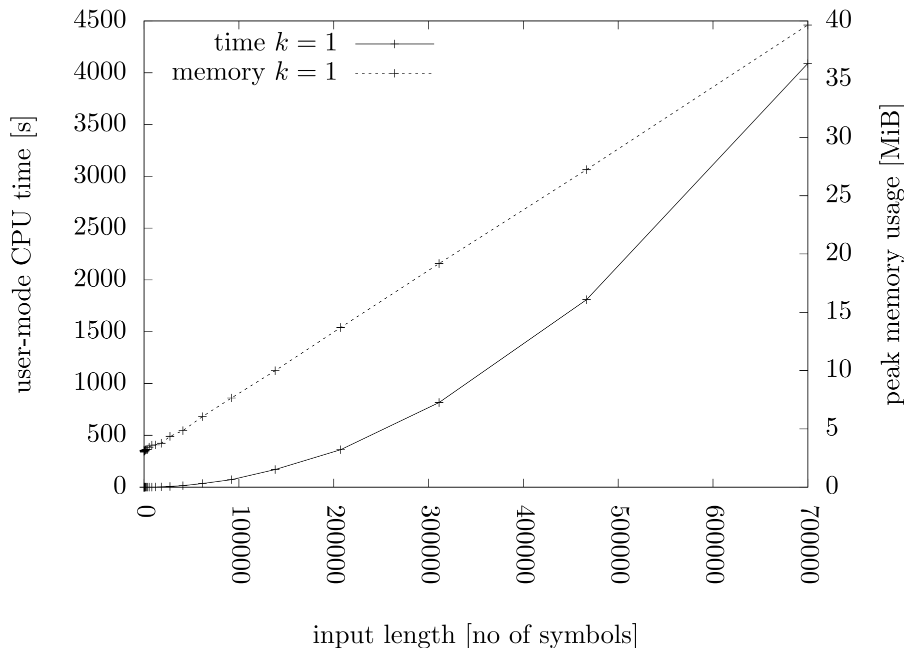



**Figure 6.** The time and memory consumption for input string $(\texttt{ab})^{4026308608}$ depending on the input string length when the maximum Hamming distance $k = 1$ is fixed

## 4 Conclusions

In this paper, new problem related to string covering has been stated and an algorithm solving the problem has been presented.

As future work, problem solutions with less than quadratic time complexity may be explored. It is probably achievable using a different data structure and technique than indexing with the subset construction of a suffix automaton. Also, the problem statement may be extended to find approximate borders that cover the maximum number of positions of a given string among all approximate borders. Also the notion of an enhanced left-cover array, introduced in [3], may be extended to accommodate the Hamming distance.

## References

1. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings*. Theoretical Computer Science, 119(2) 1993, pp. 247–265.
2. M. Christodoulakis, C. S. Iliopoulos, K. S. Park, and J. S. Sim: *Implementing approximate regularities*. Mathematical and Computer Modelling, 42 October 2005, pp. 855–866.
3. T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. Smyth, and W. Tyczyński: *Enhanced string covering*. Theoretical Computer Science, 506 2013, pp. 102–114.
4. O. Guth, B. Melichar, and M. Balik: *Searching all approximate covers and their distance using finite automata*. Information technologies–applications and theory, 2008, pp. 21–26.
5. B. Melichar, J. Holub, and T. Polcar: *Text searching algorithms, Volume I*, November 2005, Available from: `http://stringology.org/athens`.
6. D. Moore and W. F. Smyth: *An optimal algorithm to compute all the covers of a string*. Information Processing Letters, 50 1994, pp. 101–103.
7. D. Moore and W. F. Smyth: *A correction to "An optimal algorithm to compute all the covers of a string"*. Information Processing Letters, 54(2) 1995, pp. 101–103.
8. M. O. Rabin and D. Scott: *Finite automata and their decision problems*. IBM journal of research and development, 3(2) 1959, pp. 114–125.
9. J. S. Sim, K. S. Park, S. R. Kim, and J. S. Lee: *Finding approximate covers of strings*. Journal of Korea Information Science Society, 29 2002, pp. 16–21.
10. W. Smyth: *Computing regularities in strings: a survey*. European Journal of Combinatorics, 34(1) 2013, pp. 3–14.
11. M. Voráček and B. Melichar: *Searching for regularities in strings using finite automata*, in Proceedings of Workshop 2005, vol. A, Czech Technical University in Prague, 2005, pp. 264–265.

# Dynamic Index and LZ Factorization in Compressed Space

Takaaki Nishimoto[1], Tomohiro I[2], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Japan
{takaaki.nishimoto, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp
[2] Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp

**Abstract.** In this paper, we propose a new *dynamic compressed index* of $O(w)$ space for a dynamic text $T$, where $w = O(\min(z \log N \log^* M, N))$ is the size of the signature encoding of $T$, $z$ is the size of the Lempel-Ziv77 (LZ77) factorization of $T$, $N$ is the length of $T$, and $M \geq 4N$ is an integer that can be handled in constant time under word RAM model. Our index supports searching for a pattern $P$ in $T$ in $O(|P| f_{\mathcal{A}} + \log w \log |P| \log^* M (\log N + \log |P| \log^* M) + occ \log N)$ time and insertion/deletion of a substring of length $y$ in $O((y + \log N \log^* M) \log w \log N \log^* M)$ time, where $f_{\mathcal{A}} = O(\min\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\})$. Also, we propose a new space-efficient LZ77 factorization algorithm for a given text of length $N$, which runs in $O(N f_{\mathcal{A}} + z \log w \log^3 N (\log^* N)^2)$ time with $O(w)$ working space.

## 1 Introduction

### 1.1 Dynamic compressed index

Given a text $T$, the string indexing problem is to construct a data structure, called an index, so that querying occurrences of a given pattern in $T$ can be answered efficiently. As the size of data is growing rapidly in the last decade, many recent studies have focused on indexes working in compressed text space (see e.g. [11,12,7,6]). However most of them are static, i.e., they have to be reconstructed from scratch when the text is modified, which makes difficult to apply them to a dynamic text. Hence, in this paper, we consider the *dynamic compressed text indexing problem* of maintaining a compressed index for a text string that can be modified. Although there exists several dynamic *non-compressed* text indexes (see e.g. [24,3,9] for recent work), there has been little work for the compressed variants. Hon et al. [15] proposed the first dynamic compressed index of $O(\frac{1}{\epsilon}(NH_0 + N))$ bits of space which supports searching of $P$ in $O(|P| \log^2 N(\log^\epsilon N + \log |\Sigma|) + occ \log^{1+\epsilon} N)$ time and insertion/deletion of a substring of length $y$ in $O((y + \sqrt{N}) \log^{2+\epsilon} N)$ amortized time, where $0 < \epsilon \leq 1$ and $H_0 \leq \log |\Sigma|$ denotes the zeroth order empirical entropy of the text of length $N$ [15]. Salson et al. [26] also proposed a dynamic compressed index, called *dynamic FM-Index*. Although their approach works well in practice, updates require $O(N \log N)$ time in the worst case. To our knowledge, these are the only existing dynamic compressed indexes to date.

In this paper, we propose a new dynamic compressed index, as follows:

**Theorem 1.** *Let $M$ be the maximum length of the dynamic text to index, $N$ the length of the current text $T$, $w = O(\min(z \log N \log^* M, N))$ the size of the signature encoding of $T$, and $z$ the number of factors in the Lempel-Ziv 77 factorization of $T$ without self-references. Then, there exists a dynamic index of $O(w)$ space*

Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Dynamic Index and LZ Factorization in Compressed Space, pp. 158–171.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8 © Czech Technical University in Prague, Czech Republic

*which supports searching of a pattern* $P$ *in* $O(|P|f_{\mathcal{A}} + \log w \log |P| \log^* M(\log N + \log |P| \log^* M) + occ \log N)$ *time, where* $f_{\mathcal{A}} = O(\min\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\})$*, and insertion/deletion of a (sub)string* $Y$ *into/from an arbitrary position of* $T$ *in amortized* $O((|Y| + \log N \log^* M) \log w \log N \log^* M)$ *time. Moreover, if* $Y$ *is given as a substring of* $T$*, we can support insertion in amortized* $O(\log w(\log N \log^* M)^2)$ *time.*

Since $z \geq \log N$, $\log w = \max\{\log z, \log(\log^* M)\}$. Hence, our index is able to find pattern occurrences faster than the index of Hon et al. when the $|P|$ term is dominating in the pattern search times. Also, our index allows faster substring insertion/deletion on the text when the $\sqrt{N}$ term is dominating.

**Related work.** To achieve the above result, technically speaking, we use the *signature encoding* $\mathcal{G}$ of $T$, which is based on the *locally consistent parsing* technique. The signature encoding was proposed by Mehlhorn et al. for equality testing on a dynamic set of strings [17]. Since then, the signature encoding and the related ideas have been used in many applications. In particular, Alstrup et al.'s proposed dynamic index (not compressed) which is based on the signature encoding of strings, while improving the update time of signature encodings [3] and the locally consistent parsing algorithm (details can be found in the technical report [2]).

Our data structure uses Alstrup et al.'s fast string concatenation/split algorithms (update algorithm) and linear-time computation of locally consistent parsing, but has little else in common than those. Especially, Alstrup et al.'s dynamic pattern matching algorithm [3,2] requires to maintain specific locations called *anchors* over the parse trees of the signature encodings, but our index does not use anchors. Our index has close relationship to the ESP-indices [27,28], but there are two significant differences between ours and ESP-indices: The first difference is that the ESP-index [27] is static and its online variant [28] allows only for appending new characters to the end of the text, while our index is fully dynamic allowing for insertion and deletion of arbitrary substrings at arbitrary positions. The second difference is that the pattern search time of the ESP-index is proportional to the number $occ_c$ of occurrences of the so-called "core" of a query pattern $P$, which corresponds to a maximal subtree of the ESP derivation tree of a query pattern $P$. If $occ$ is the number of occurrences of $P$ in the text, then it always holds that $occ_c \geq occ$, and in general $occ_c$ cannot be upper bounded by any function of $occ$. In contrast, as can be seen in Theorem 1, the pattern search time of our index is proportional to the number $occ$ of occurrences of a query pattern $P$. This became possible due to our discovery of a new property of the signature encoding [2] (stated in Lemma 16).

As another application of signature encodings, Nishimoto et al. showed that signature encodings for a dynamic string $T$ can support Longest Common Extension (LCE) queries on $T$ efficiently in compressed space [20] (Lemma 10). They also showed signature encodings can be updated in compressed space (Lemma 12). Our algorithm uses properties of signature encodings shown in [20], more precisely, Lemmas 5-10 and 12, but Lemma 16 is a new property of signature encodings not described in [20].

In relation to our problem, there exists the library management problem of maintaining a text collection (a set of text strings) allowing for insertion/deletion of texts (see [18] for recent work). While in our problem a single text is edited by insertion/deletion of substrings, in the library management problem a text can be inserted to or deleted from the collection. Hence, algorithms for the library management problem cannot be directly applied to our problem.

### 1.2 Computing LZ77 factorization in compressed space.

As an application of our dynamic compressed index, we present a new LZ77 factorization algorithm working in compressed space.

The Lempel-Ziv77 (LZ77) factorization is defined as follows.

**Definition 2 (Lempel-Ziv77 factorization [29]).** *The Lempel-Ziv77 (LZ77) factorization of a string $s$ without self-references is a sequence $f_1, \ldots, f_z$ of non-empty substrings of $s$ such that $s = f_1 \cdots f_z$, $f_1 = s[1]$, and for $1 < i \leq z$, if the character $s[|f_1..f_{i-1}| + 1]$ does not occur in $s[|f_1..f_{i-1}|]$, then $f_i = s[|f_1..f_{i-1}| + 1]$, otherwise $f_i$ is the longest prefix of $f_i \cdots f_z$ which occurs in $f_1 \cdots f_{i-1}$. The size of the LZ77 factorization $f_1, \ldots, f_z$ of string $s$ is the number $z$ of factors in the factorization.*

Although the primary use of LZ77 factorization is data compression, it has been shown that it is a powerful tool for many string processing problems [13,12]. Hence the importance of algorithms to compute LZ77 factorization is growing. Particularly, in order to apply algorithms to large scale data, reducing the working space is an important matter. In this paper, we focus on LZ77 factorization algorithms working in *compressed space*.

The following is our main result.

**Theorem 3.** *Given the signature encoding $\mathcal{G}$ of size $w$ for a string $T$ of length $N$, we can compute the LZ77 factorization of $T$ in $O(z \log w \log^3 N (\log^* M)^2)$ time and $O(w)$ working space where $z$ is the size of the LZ77 factorization of $T$.*

In [20], it was shown that the signature encoding $\mathcal{G}$ can be constructed efficiently from various types of inputs, in particular, in $O(N f_{\mathcal{A}})$ time and $O(w)$ working space from uncompressed string $T$. Therefore we can compute LZ77 factorization of a given $T$ of length $N$ in $O(N f_{\mathcal{A}} + z \log w \log^3 N (\log^* M)^2)$ time and $O(w)$ working space.

**Related work.** Goto et al. [14] showed how, given the grammar-like representation for string $T$ generated by the LCA algorithm [25], to compute the LZ77 factorization of $T$ in $O(z \log^2 m \log^3 N + m \log m \log^3 N)$ time and $O(m \log^2 m)$ space, where $m$ is the size of the given representation. Sakamoto et al. [25] claimed that $m = O(z \log N \log^* N)$, however, it seems that in this bound they do not consider the production rules to represent maximal runs of non-terminals in the derivation tree. The bound we were able to obtain with the best of our knowledge and understanding is $m = O(z \log^2 N \log^* N)$, and hence our algorithm seems to use less space than the algorithm of Goto et al. [14]. Recently, Fischer et al. [10] showed a Monte-Carlo randomized algorithms to compute an approximation of the LZ77 factorization with at most $2z$ factors in $O(N \log N)$ time, and another approximation with at most $(i+\epsilon)z$ factors in $O(N \log^2 N)$ time for any constant $\epsilon > 0$, using $O(z)$ space each.

Another line of research is LZ77 factorization working in compressed space in terms of Burrows-Wheeler transform (BWT) based methods. Policriti and Prezza recently proposed algorithms running in $NH_0 + o(N \log |\Sigma|) + O(|\Sigma| \log N)$ bits of space and $O(N \log N)$ time [21], or $O(R \log N)$ bits of space and $O(N \log R)$ time [22], where $R$ is the number of runs in the BWT of the reversed string of $T$. Because their and our algorithms are established on different measures of compression, they cannot be easily compared. For example, our algorithm is more space efficient than the algorithm in [22] when $w = o(R)$, but it is not clear when it happens.

Examples and figures omitted due to lack of space are in a full version of this paper [19].

# 2 Preliminaries

## 2.1 Strings

Let $\Sigma$ be an ordered alphabet. An element of $\Sigma^*$ is called a string. For string $w = xyz$, $x$, $y$ and $z$ are called a prefix, substring, and suffix of $w$, respectively. The length of string $w$ is denoted by $|w|$. The empty string $\varepsilon$ is a string of length 0. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For any $1 \leq i \leq |w|$, $w[i]$ denotes the $i$-th character of $w$. For any $1 \leq i \leq j \leq |w|$, $w[i..j]$ denotes the substring of $w$ that begins at position $i$ and ends at position $j$. Let $w[i..] = w[i..|w|]$ and $w[..i] = w[1..i]$ for any $1 \leq i \leq |w|$. For any string $w$, let $w^R$ denote the reversed string of $w$, that is, $w^R = w[|w|] \cdots w[2]w[1]$. For any strings $w$ and $u$, let $\mathsf{LCP}(w, u)$ (resp. $\mathsf{LCS}(w, u)$) denote the length of the longest common prefix (resp. suffix) of $w$ and $u$. Given two strings $s_1, s_2$ and two integers $i, j$, let $\mathsf{LCE}(s_1, s_2, i, j)$ denote a query which returns $\mathsf{LCP}(s_1[i..|s_1|], s_2[j..|s_2|])$. For any strings $p$ and $s$, let $Occ(p, s)$ denote all occurrence positions of $p$ in $s$, namely, $Occ(p, s) = \{i \mid p = s[i..i + |p| - 1], 1 \leq i \leq |s| - |p| + 1\}$. Our model of computation is the unit-cost word RAM with machine word size of $\Omega(\log_2 M)$ bits, and space complexities will be evaluated by the number of machine words. Bit-oriented evaluation of space complexities can be obtained with a $\log_2 M$ multiplicative factor.

## 2.2 Context free grammars as compressed representation of strings

**Straight-line programs.** A *straight-line program* (*SLP*) is a context free grammar in the Chomsky normal form that generates a single string. Formally, an SLP that generates $T$ is a quadruple $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$, such that $\Sigma$ is an ordered alphabet of terminal characters; $\mathcal{V} = \{X_1, \ldots, X_n\}$ is a set of positive integers, called *variables*; $\mathcal{D} = \{X_i \to expr_i\}_{i=1}^{n}$ is a set of *deterministic productions* (or *assignments*) with each $expr_i$ being either of form $X_\ell X_r$ $(1 \leq \ell, r < i)$, or a single character $a \in \Sigma$; and $S := X_n \in \mathcal{V}$ is the start symbol which derives the string $T$. We also assume that the grammar neither contains *redundant* variables (i.e., there is at most one assignment whose righthand side is *expr*) nor *useless* variables (i.e., every variable appears at least once in the derivation tree of $\mathcal{G}$). The *size* of the SLP $\mathcal{G}$ is the number $n$ of productions in $\mathcal{D}$. In the extreme cases the length $N$ of the string $T$ can be as large as $2^{n-1}$, however, it is always the case that $n \geq \log_2 N$.

Let $val : \mathcal{V} \to \Sigma^+$ be the function which returns the string derived by an input variable. If $s = val(X)$ for $X \in \mathcal{V}$, then we say that the variable $X$ *represents* string $s$. For any variable sequence $y \in \mathcal{V}^+$, let $val^+(y) = val(y[1]) \cdots val(y[|y|])$. For any variable $X_i$ with $X_i \to X_\ell X_r \in \mathcal{D}$, let $X_i.\text{left} = val(X_\ell)$ and $X_i.\text{right} = val(X_r)$, which are called the *left string* and the *right string* of $X_i$, respectively. For two variables $X_i, X_j \in \mathcal{V}$, we say that $X_i$ occurs at position $c$ in $X_j$ if there is a node labeled with $X_i$ in the derivation tree of $X_j$ and the leftmost leaf of the subtree rooted at that node labeled with $X_i$ is the $c$-th leaf in the derivation tree of $X_j$. We define the function $vOcc(X_i, X_j)$ which returns all positions of $X_i$ in the derivation tree of $X_j$.

**Run-length straight-line programs.** We define *run-length SLPs*, (*RLSLPs*) as an extension to SLPs, which allow *run-length encodings* in the righthand sides of productions, i.e., $\mathcal{D}$ might contain a production $X \to \hat{X}^k \in \mathcal{V} \times \mathcal{N}$. The *size* of the RLSLP is still the number of productions in $\mathcal{D}$ as each production can be encoded in constant space. Let $Assgn_\mathcal{G}$ be the function such that $Assgn_\mathcal{G}(X_i) = expr_i$ iff $X_i \to expr_i \in \mathcal{D}$. Also, let $Assgn_\mathcal{G}^{-1}$ denote the reverse function of $Assgn_\mathcal{G}$. When clear from the context, we write $Assgn_\mathcal{G}$ and $Assgn_\mathcal{G}^{-1}$ as $Assgn$ and $Assgn^{-1}$, respectively.

We define the left and right strings for any variable $X_i \to X_\ell X_r \in \mathcal{D}$ in a similar way to SLPs. Furthermore, for any $X \to \hat{X}^k \in \mathcal{D}$, let $X$.left $= val(\hat{X})$ and $X$.right $= val(\hat{X})^{k-1}$.

**Representation of RLSLPs.** For an RLSLP $\mathcal{G}$ of size $w$, we can consider a DAG of size $w$ as a compact representation of the derivation trees of variables in $\mathcal{G}$. Each node represents a variable $X$ in $\mathcal{V}$ and stores $|val(X)|$ and out-going edges represent the assignments in $\mathcal{D}$: For an assignment $X_i \to X_\ell X_r \in \mathcal{D}$, there exist two out-going edges from $X_i$ to its ordered children $X_\ell$ and $X_r$; and for $X \to \hat{X}^k \in \mathcal{D}$, there is a single edge from $X$ to $\hat{X}$ with the multiplicative factor $k$. For $X \in \mathcal{V}$, let $parents(X)$ be the set of variables which have out-going edge to $X$ in the DAG of $\mathcal{G}$. To compute $parents(X)$ for $X \in \mathcal{V}$ in linear time, we let $X$ have a doubly-linked list of length $|parents(X)|$ to represent $parents(X)$: Each element is a pointer to a node for $X' \in parents(X)$ (the order of elements is arbitrary). Conversely, we let every parent $X'$ of $X$ have the pointer to the corresponding element in the list.

## 3 Signature encoding

Here, we recall the *signature encoding* first proposed by Mehlhorn et al. [17]. Its core technique is *locally consistent parsing* defined as follows:

**Lemma 4 (Locally consistent parsing [17,2]).** *Let $W$ be a positive integer. There exists a function $f : [0..W]^{\log^* W+11} \to \{0,1\}$ such that, for any $p \in [1..W]^n$ with $n \geq 2$ and $p[i] \neq p[i+1]$ for any $1 \leq i < n$, the bit sequence $d$ defined by $d[i] = f(\tilde{p}[i-\Delta_L], \dots, \tilde{p}[i+\Delta_R])$ for $1 \leq i \leq n$, satisfies: $d[1] = 1$; $d[n] = 0$; $d[i]+d[i+1] \leq 1$ for $1 \leq i < n$; and $d[i] + d[i+1] + d[i+2] + d[i+3] \geq 1$ for any $1 \leq i < n-3$; where $\Delta_L = \log^* W + 6$, $\Delta_R = 4$, and $\tilde{p}[j] = p[j]$ for all $1 \leq j \leq n$, $\tilde{p}[j] = 0$ otherwise. Furthermore, we can compute $d$ in $O(n)$ time using a precomputed table of size $o(\log W)$, which can be computed in $o(\log W)$ time.*

For the bit sequence $d$ of Lemma 4, we define the function $Eblock_d(p)$ that decomposes an integer sequence $p$ according to $d$: $Eblock_d(p)$ decomposes $p$ into a sequence $q_1, \dots, q_j$ of substrings called *blocks* of $p$, such that $p = q_1 \cdots q_j$ and $q_i$ is in the decomposition iff $d[|q_1 \cdots q_{i-1}| + 1] = 1$ for any $1 \leq i \leq j$. Note that each block is of length from two to four by the property of $d$, i.e., $2 \leq |q_i| \leq 4$ for any $1 \leq i \leq j$. Let $|Eblock_d(p)| = j$ and let $Eblock_d(s)[i] = q_i$. We omit $d$ and write $Eblock(p)$ when it is clear from the context, and we use implicitly the bit sequence created by Lemma 4 as $d$.

We complementarily use run-length encoding to get a sequence to which $Eblock$ can be applied. Formally, for a string $s$, let $Epow(s)$ be the function which groups each maximal run of same characters $a$ as $a^k$, where $k$ is the length of the run. $Epow(s)$ can be computed in $O(|s|)$ time. Let $|Epow(s)|$ denote the number of maximal runs of same characters in $s$ and let $Epow(s)[i]$ denote $i$-th maximal run in $s$.

The signature encoding is the RLSLP $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$, where the assignments in $\mathcal{D}$ are determined by recursively applying $Eblock$ and $Epow$ to $T$ until a single integer $S$ is obtained. We call each variable of the signature encoding a *signature*, and use $e$ (for example, $e_i \to e_\ell e_r \in \mathcal{D}$) instead of $X$ to distinguish from general RLSLPs.

For a formal description, let $E := \Sigma \cup \mathcal{V}^2 \cup \mathcal{V}^3 \cup \mathcal{V}^4 \cup (\mathcal{V} \times \mathcal{N})$ and let $Sig : E \to \mathcal{V}$ be the function such that: $Sig(x) = e$ if $(e \to x) \in \mathcal{D}$; $Sig(x) = Sig(Sig(x[1..|x| - 1])x[|x|])$ if $x \in \mathcal{V}^3 \cup \mathcal{V}^4$; or otherwise undefined. Namely, the function $Sig$ returns,

if any, the lefthand side of the corresponding production of $x$ by recursively applying the $Assgn^{-1}$ function from left to right. For any $p \in E^*$, let $Sig^+(p) = Sig(p[1]) \cdots Sig(p[|p|])$.

The signature encoding of string $T$ is defined by the following *Shrink* and *Pow* functions: $Shrink_t^T = Sig^+(T)$ for $t = 0$, and $Shrink_t^T = Sig^+(Eblock(Pow_{t-1}^T))$ for $0 < t \leq h$; and $Pow_t^T = Sig^+(Epow(Shrink_t^T))$ for $0 \leq t \leq h$; where $h$ is the minimum integer satisfying $|Pow_h^T| = 1$. Then, the start symbol of the signature encoding is $S = Pow_h^T$. We say that a node is in *level* $t$ in the derivation tree of $S$ if the node is produced by $Shrink_t^T$ or $Pow_t^T$. The height of the derivation tree of the signature encoding of $T$ is $O(h) = O(\log |T|)$. For any $T \in \Sigma^+$, let $id(T) = Pow_h^T = S$, i.e., the integer $S$ is the signature of $T$. We let $N \leq M/4$. More specifically, $M = 4N$ if $T$ is static, and $M/4$ is the upper bound of the length of $T$ if we consider updating $T$ dynamically. Since all signatures are in $[1..M-1]$, we set $W = M$ in Lemma 4 used by the signature encoding. In this paper, we implement signature encodings by the DAG of RLSLP introduced in Section 2.

### 3.1 Commmon sequences

Here, we recall the most important property of the signature encoding, which ensures the existence of common signatures to all occurrences of same substrings by the following lemma.

**Lemma 5 (common sequences [23,20]).** *Let $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{D}, S)$ be a signature encoding for a string $T$. Every substring $P$ in $T$ is represented by a signature sequence $Uniq(P)$ in $\mathcal{G}$ for a string $P$, where $|Epow(Uniq(P))| = O(\log |P| \log^* M)$.*

$Uniq(P)$, which we call the *common sequence* of $P$, is defined by the following.

**Definition 6.** *For a string $P$, let*

$$XShrink_t^P = \begin{cases} Sig^+(P) & \text{for } t = 0, \\ Sig^+(Eblock_d(XPow_{t-1}^P)[|L_t^P|..|XPow_{t-1}^P| - |R_t^P|]) & \text{for } 0 < t \leq h^P, \end{cases}$$

$$XPow_t^P = Sig^+(Epow(XShrink_t^P[|\hat{L}_t^P| + 1..|XShrink_t^P| - |\hat{R}_t^P|])|) \quad \text{for } 0 \leq t < h^P,$$

- *$L_t^P$ is the shortest prefix of $XPow_{t-1}^P$ of length at least $\Delta_L$ such that $d[|L_t^P|+1] = 1$,*
- *$R_t^P$ is the shortest suffix of $XPow_{t-1}^P$ of length at least $\Delta_R + 1$ such that $d[|d| - |R_t^P| + 1] = 1$,*
- *$\hat{L}_t^P$ is the longest prefix of $XShrink_t^P$ such that $|Epow(\hat{L}_t^P)| = 1$,*
- *$\hat{R}_t^P$ is the longest suffix of $XShrink_t^P$ such that $|Epow(\hat{R}_t^P)| = 1$, and*
- *$h^P$ is the minimum integer such that $|Epow(XShrink_{h^P}^P)| \leq \Delta_L + \Delta_R + 9$.*

Note that $\Delta_L \leq |L_t^P| \leq \Delta_L + 3$ and $\Delta_R + 1 \leq |R_t^P| \leq \Delta_R + 4$ hold by the definition. Hence $|XShrink_{t+1}^P| > 0$ holds if $|Epow(XShrink_t^P)| > \Delta_L + \Delta_R + 9$. Then,

$$Uniq(P) = \hat{L}_0^P L_0^P \cdots \hat{L}_{h^P-1}^P L_{h^P-1}^P XShrink_{h^P}^P R_{h^P-1}^P \hat{R}_{h^P-1}^P \cdots R_0^P \hat{R}_0^P.$$

We give an intuitive description of Lemma 5. Recall that the locally consistent parsing of Lemma 4. Each $i$-th bit of bit sequence $d$ of Lemma 4 for a given string $s$ is determined by $s[i - \Delta_L..i + \Delta_R]$. Hence, for two positions $i, j$ such that $P = s[i..i+k-1] = s[j..j+k-1]$ for some $k$, $d[i+\Delta_L..i+k-1-\Delta_R] = d[j+\Delta_L..j+k-1-\Delta_R]$

holds, namely, "internal" bit sequences of the same substring of $s$ are equal. Since each level of the signature encoding uses the bit sequence, all occurrences of same substrings in a string share same internal signature sequences, and this goes up level by level. $XShrink_t^P$ and $XPow_t^P$ represent signature sequences which are obtained from only internal signature sequences of $XPow_{t-1}^T$ and $XShrink_t^T$, respectively. This means that $XShrink_t^P$ and $XPow_t^P$ are always created over $P$. From such common signatures we take as short signature sequence as possible for $Uniq(P)$: Since $val^+(Pow_{t-1}^P) = val^+(L_{t-1}^P XShrink_t^P R_{t-1}^P)$ and $val^+(Shrink_t^P) = val^+(\hat{L}_t^P XPow_t^P \hat{R}_t^P)$ hold, $|Epow(Uniq(P))| = O(\log|P|\log^* M)$ and $val^+(Uniq(P)) = P$ hold. Hence Lemma 5 holds [1].

From the common sequences we can derive many useful properties of signature encodings like listed below (see the references for proofs).

The number of ancestors of nodes corresponding to $Uniq(P)$ is upper bounded by:

**Lemma 7 ([20]).** *Let $\mathcal{G}$ be a signature encoding for a string $T$, $P$ be a string, and let $\mathcal{T}$ be the derivation tree of a signature $e \in \mathcal{V}$. Consider an occurrence of $P$ in $s$, and the induced subtree $X$ of $\mathcal{T}$ whose root is the root of $\mathcal{T}$ and whose leaves are the parents of the nodes representing $Uniq(P)$, where $s = val(e)$. Then $X$ contains $O(\log^* M)$ nodes for every level and $O(\log|s| + \log|P|\log^* M)$ nodes in total.*

We can efficiently compute $Uniq(P)$ for a substring $P$ of $T$.

**Lemma 8 ([20]).** *Using a signature encoding $\mathcal{G}$ of size $w$, given a signature $e \in \mathcal{V}$ (and its corresponding node in the DAG) and two integers $j$ and $y$, we can compute $Epow(Uniq(s[j..j+y-1]))$ in $O(\log|s| + \log y \log^* M)$ time, where $s = val(e)$.*

The next lemma shows that $\mathcal{G}$ requires only *compressed space*:

**Lemma 9 ([23,20]).** *The size $w$ of the signature encoding of $T$ of length $N$ is $O(\min(z \log N \log^* M, N))$, where $z$ is the number of factors in the LZ77 factorization without self-reference of $T$.*

The next lemma shows that the signature encoding supports (both forward and backward) LCE queries on a given arbitrary pair of signatures.

**Lemma 10 ([20]).** *Using a signature encoding $\mathcal{G}$ for a string $T$, we can support queries $\mathsf{LCE}(s_1, s_2, i, j)$ and $\mathsf{LCE}(s_1^R, s_2^R, i, j)$ in $O(\log|s_1| + \log|s_2| + \log\ell\log^* M)$ time for given two signatures $e_1, e_2 \in \mathcal{V}$ and two integers $1 \le i \le |s_1|$, $1 \le j \le |s_2|$, where $s_1 = val(e_1)$, $s_2 = val(e_2)$ and $\ell$ is the answer to the $\mathsf{LCE}$ query.*

### 3.2 Dynamic signature encoding

We consider a *dynamic signature encoding* $\mathcal{G}$ of $T$, which allows for efficient updates of $\mathcal{G}$ in compressed space according to the following operations: $INSERT(Y, i)$ inserts a string $Y$ into $T$ at position $i$, i.e., $T \leftarrow T[..i-1]YT[i..]$; $INSERT'(j, y, i)$ inserts $T[j..j+y-1]$ into $T$ at position $i$, i.e., $T \leftarrow T[..i-1]T[j..j+y-1]T[i..]$; and $DELETE(j, y)$ deletes a substring of length $y$ starting at $j$, i.e., $T \leftarrow T[..j-1]T[j+y..]$.

During updates we recompute $Shrink_t^T$ and $Pow_t^T$ for some part of new $T$ (note that the most part is unchanged thanks to the virtue of signature encodings, Lemma 7).

[1] The common sequences are conceptually equivalent to the *cores* [16] which are defined for the *edit sensitive parsing* of a text, a kind of locally consistent parsing of the text.

When we need a signature for *expr*, we look up the signature assigned to *expr* (i.e., compute $Assign^{-1}(expr)$) and use it if such exists. If $Assign^{-1}(expr)$ is undefined we create a new signature $e_{new}$, which is an integer that is currently not used as signatures, and add $e_{new} \rightarrow expr$ to $\mathcal{D}$. Also, updates may produce a useless signature whose parents in the DAG are all removed. We remove such useless signatures from $\mathcal{G}$ during updates.

We can upper bound the number of signatures added to or removed from $\mathcal{G}$ after a single update operation by the following lemma. [2]

**Lemma 11.** *After* $INSERT(Y, i)$ *or* $DELETE(j, y)$ *operation,* $O(y + \log N \log^* M)$ *signatures are added to or removed from* $\mathcal{G}$*, where* $|Y| = y$*. After* $INSERT'(j, y, i)$ *operation,* $O(\log N \log^* M)$ *signatures are added to or removed from* $\mathcal{G}$*.*

*Proof.* Consider $INSERT'(j, y, i)$ operation. Let $T' = T[..i-1]T[j..j+y-1]T[i..]$ be the new text. Note that by Lemma 5 the signature encoding of $T'$ is created over $Uniq(T[..i-1])\,Uniq(T[j..j+y-1])\,Uniq(T[i..])$, and hence, $O(\log N \log^* M)$ signatures can be added by Lemma 7. Also, $O(\log N \log^* M)$ signatures, which were created over $Uniq(T[..i-1])\,Uniq(T[i..])$, may be removed.

For $INSERT(Y, i)$ operation, we additionally think about the possibility that $O(y)$ signatures are added to create $Uniq(Y)$. Similarly, for $DELETE(j, y)$ operation, $O(y)$ signatures, which are used in and under $Uniq(T[j..j+y-1])$, can be removed. □

In [20], it was shown how to augment the DAG representation of $\mathcal{G}$ to add/remove an assignment to/from $\mathcal{G}$ in $O(f_{\mathcal{A}})$ time, where $f_{\mathcal{A}} = O\left(\min\left\{\frac{\log\log M \log\log w}{\log\log\log M}, \sqrt{\frac{\log w}{\log\log w}}\right\}\right)$ is the time complexity of Beame and Fich's data structure [4] to support predecessor/successor queries on a set of $w$ integers from an $M$-element universe.[3] Note that there is a small difference in our DAG representation from the one in [20]; our DAG has a doubly-linked list representing the parents of a node. We can check if a signature is useless or not by checking if the list is empty or not, and the lists can be maintained in constant time after adding/removing an assignment. Hence, the next lemma still holds for our DAG representation.

**Lemma 12 (Dynamic signature encoding [20]).** *After processing* $\mathcal{G}$ *in* $O(wf_{\mathcal{A}})$ *time, we can insert/delete any (sub)string* $Y$ *of length* $y$ *into/from an arbitrary position of* $T$ *in* $O((y + \log N \log^* M)f_{\mathcal{A}})$ *time. Moreover, if* $Y$ *is given as a substring of* $T$*, we can support insertion in* $O(f_{\mathcal{A}} \log N \log^* M)$ *time.*

## 4 Dynamic Compressed Index

In this section, we present our dynamic compressed index based on signature encoding. As already mentioned in the introduction, our strategy for pattern matching is different from that of Alstrup et al. [2]. It is rather similar to the one taken in the static index for SLPs of Claude and Navarro [6]. Besides applying their idea to RLSLPs, we show how to speed up pattern matching by utilizing the properties of signature encodings.

**Index for SLPs.** Here we review how the index in [6] for SLP $\mathcal{S}$ generating a string $T$ computes $Occ(P, T)$ for a given string $P$. The key observation is that, any occurrence

[2] The property is used in [20], but there is no corresponding lemma to state it clearly.

[3] The data structure is, for example, used to compute $Assgn^{-1}(\cdot)$. Alstrup et al. [2] used hashing for this purpose. However, since we are interested in the worst case time complexities, we use the data structure [4] in place of hashing.

of $P$ in $T$ can be uniquely associated with the lowest node that covers the occurrence of $P$ in the derivation tree. As the derivation tree is binary, if $|P| > 1$, then the node is labeled with some variable $X \in \mathcal{V}$ such that $P_1$ is a suffix of $X$.left and $P_2$ is a prefix of $X$.right, where $P = P_1P_2$ with $1 \leq |P_1| < |P|$. Here we call the pair $(X, |X.\text{left}| - |P_1| + 1)$ a *primary occurrence* of $P$, and let $pOcc_{\mathcal{S}}(P, j)$ denote the set of such primary occurrences with $|P_1| = j$. The set of all primary occurrences is denoted by $pOcc_{\mathcal{S}}(P) = \bigcup_{1\leq j<|P|} pOcc_{\mathcal{S}}(P, j)$. Then, we can compute $Occ(P, T)$ by first computing primary occurrences and enumerating the occurrences of $X$ in the derivation tree.

The set $Occ(P, T)$ of occurrences of $P$ in $T$ is represented by $pOcc_{\mathcal{S}}(P)$ as follows: $Occ(P, T) = \{j + k - 1 \mid (X, j) \in pOcc_{\mathcal{S}}(P), k \in vOcc(X, S)\}$ if $|P| > 1$; $Occ(P, T) = vOcc(X, S)((X \to P) \in \mathcal{D})$ if $|P| = 1$.

Hence the task is to compute $pOcc_{\mathcal{S}}(P)$ and $vOcc(X, S)$ efficiently. Note that $vOcc(X, S)$ can be computed in $O(|vOcc(X, S)|h)$ time by traversing the DAG in a reversed direction from $X$ to the source, where $h$ is the height of the derivation tree of $S$. Hence, in what follows, we explain how to compute $pOcc_{\mathcal{S}}(P)$ for a string $P$ with $|P| > 1$. We consider the following problem:

*Problem 13 (Two-Dimensional Orthogonal Range Reporting Problem).* Let $\mathcal{X}$ and $\mathcal{Y}$ denote subsets of two ordered sets, and let $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{Y}$ be a set of points on the two-dimensional plane, where $|\mathcal{X}|, |\mathcal{Y}| \in O(|\mathcal{R}|)$. A data structure for this problem supports a query $report_{\mathcal{R}}(x_1, x_2, y_1, y_2)$; given a rectangle $(x_1, x_2, y_1, y_2)$ with $x_1, x_2 \in \mathcal{X}$ and $y_1, y_2 \in \mathcal{Y}$, returns $\{(x, y) \in \mathcal{R} \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$.

Data structures for Problem 13 are widely studied in computational geometry. There is even a dynamic variant, which we finally use for our dynamic index. Until then, we just use any data structure that occupies $O(|\mathcal{R}|)$ space and supports queries in $O(\hat{q}_{|\mathcal{R}|} + q_{|\mathcal{R}|}qocc)$ time with $\hat{q}_{|\mathcal{R}|} = O(\log |\mathcal{R}|)$, where $qocc$ is the number of points to report.

Now, given an SLP $\mathcal{S}$, we consider a two-dimensional plane defined by $\mathcal{X} = \{X.\text{left}^R \mid X \in \mathcal{V}\}$ and $\mathcal{Y} = \{X.\text{right} \mid X \in \mathcal{V}\}$, where elements in $\mathcal{X}$ and $\mathcal{Y}$ are sorted by lexicographic order. Then consider a set of points $\mathcal{R} = \{(X.\text{left}^R, X.\text{right}) \mid X \in \mathcal{V}\}$. For a string $P$ and an integer $1 \leq j < |P|$, let $y_1^{(P,j)}$ (resp. $y_2^{(P,j)}$) denote the lexicographically smallest (resp. largest) element in $\mathcal{Y}$ that has $P[j + 1..]$ as a prefix. If there is no such element, it just returns NIL and we can immediately know that $pOcc_{\mathcal{S}}(P, j) = \emptyset$. We define $x_1^{(P,j)}$ and $x_2^{(P,j)}$ in a similar way over $\mathcal{X}$. Then, $pOcc_{\mathcal{S}}(P, j)$ can be computed by a query $report_{\mathcal{R}}(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$.

Using this idea, we can get the next result:

**Lemma 14.** *For an SLP $\mathcal{S}$ of size $n$, there exists a data structure of size $O(n)$ that computes, given a string $P$, $pOcc_{\mathcal{S}}(P)$ in $O(|P|(h + |P|) \log n + q_n|pOcc_{\mathcal{S}}(P)|)$ time.*

*Proof.* For every $1 \leq j < |P|$, we compute $pOcc_{\mathcal{S}}(P, j)$ by $report_{\mathcal{R}}(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$. We can compute $y_1^{(P,j)}$ and $y_2^{(P,j)}$ in $O((h + |P|) \log n)$ time by binary search on $\mathcal{Y}$, where each comparison takes $O(h + |P|)$ time for expanding the first $O(|P|)$ characters of variables subjected to comparison. In a similar way, $x_1^{(P,j)}$ and $x_2^{(P,j)}$ can be computed in $O((h+|P|) \log n)$ time. Thus, the total time complexity is $O(|P|((h+|P|) \log n + \hat{q}_n) + q_n|pOcc_{\mathcal{S}}(P)|) = O(|P|(h + |P|) \log n + q_n|pOcc_{\mathcal{S}}(P)|)$. □

**Index for RLSLPs.** We extend the idea for the SLP index described above to RLSLPs. The difference from SLPs is that we have to deal with occurrences of $P$

that are covered by a node labeled with $X \rightarrow \hat{X}^k$ but not covered by any single child of the node in the derivation tree. In such a case, there must exist $P = P_1 P_2$ with $1 \leq |P_1| < |P|$ such that $P_1$ is a suffix of $X$.left $= val^+(\hat{X})$ and $P_2$ is a prefix of $X$.right $= val^+(\hat{X}^{k-1})$. Let $j = |val(\hat{X})| - |P_1| + 1$ be a position in $val^+(\hat{X}^d)$ where $P$ occurs, then $P$ also occurs at $j + c|val(\hat{X})|$ in $val^+(\hat{X}^k)$ for every positive integer $c$ with $j + c|val(\hat{X})| + |P| - 1 \leq |val^+(\hat{X}^k)|$. Using this observation, the index for SLPs can be modified for RLSLPs to achieve the same bounds as in Lemma 14.

**Index for signature encodings.** Since signature encodings are RLSLPs, we can compute $Occ(P, T)$ by querying $report_{\mathcal{R}}(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$ for "every" $1 \leq j < |P|$. However, the properties of signature encodings allow us to speed up pattern matching as summarized in the following two ideas: (1) We can efficiently compute $x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ using LCE queries in compressed space (Lemma 15). (2) We can reduce the number of $report_{\mathcal{R}}$ queries from $O(|P|)$ to $O(\log |P| \log^* M)$ by using the property of the common sequence of $P$ (Lemma 16).

**Lemma 15.** *Assume that we have the signature encoding $\mathcal{G}$ of size $w$ for a string $T$ of length $N$, $\mathcal{X}$ and $\mathcal{Y}$ of $\mathcal{G}$. Given a signature $id(P) \in \mathcal{V}$ for a string $P$ and an integer $j$, we can compute $x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ in $O(\log w(\log N + \log |P| \log^* M))$ time.*

*Proof.* By Lemma 10 we can compute $x_1^{(P,j)}$ and $x_2^{(P,j)}$ on $\mathcal{X}$ by binary search in $O(\log w(\log N + \log |P| \log^* M))$ time. Similarly, we can compute $y_1^{(P,j)}$ and $y_2^{(P,j)}$ in the same time. □

**Lemma 16.** *Let $P$ be a string with $|P| > 1$. If $|Pow_0^P| = 1$, then $pOcc_{\mathcal{G}}(P) = pOcc_{\mathcal{G}}(P, 1)$. If $|Pow_0^P| > 1$, then $pOcc_{\mathcal{G}}(P) = \bigcup_{j \in \mathcal{P}} pOcc_{\mathcal{G}}(P, j)$, where $\mathcal{P} = \{|val^+(u[1..i])| \mid 1 \leq i < |u|, u[i] \neq u[i+1]\}$ with $u = Uniq(P)$.*

*Proof.* If $|Pow_0^P| = 1$, then $P = a^{|P|}$ for some character $a \in \Sigma$. In this case, $P$ must be contained in a node labeled with a signature $e \rightarrow \hat{e}^d$ such that $\hat{e} \rightarrow a$ and $d \geq |P|$. Hence, all primary occurrences of $P$ can be found by $pOcc_{\mathcal{G}}(P, 1)$.

If $|Pow_0^P| > 1$, we consider the common sequence $u$ of $P$. Recall that substring $P$ occurring at $j$ in $val(e)$ is represented by $u$ for any $(e, j) \in pOcc(P)$ by Lemma 5 Hence at least $pOcc_{\mathcal{G}}(P) = \bigcup_{i \in \mathcal{P}'} pOcc_{\mathcal{G}}(P, i)$ holds, where $\mathcal{P}' = \{|val^+(u[1])|, \ldots, |val^+( u[..|u| - 1])|\}$. Moreover, we show that $pOcc_{\mathcal{G}}(P, i) = \emptyset$ for any $i \in \mathcal{P}'$ with $u[i] = u[i+1]$. Note that $u[i]$ and $u[i+1]$ are encoded into the same signature in the derivation tree of $e$, and that the parent of two nodes corresponding to $u[i]$ and $u[i+1]$ has a signature $e'$ in the form $e' \rightarrow u[i]^d$. Now assume for the sake of contradiction that $e = e'$. By the definition of the primary occurrences, $i = 1$ must hold, and hence, $Shrink_0^P[1] = u[1] \in \Sigma$. This means that $P = u[1]^{|P|}$, which contradicts $|Pow_0^P| > 1$. Therefore the statement holds. □

Using Lemmas 5, 15 and 16, we get a static index for signature encodings:

**Lemma 17.** *For a signature encoding $\mathcal{G}$ of size $w$ which generates a text $T$ of length $N$, there exists a data structure of size $O(w)$ that computes, given a string $P$, $pOcc_{\mathcal{G}}(P)$ in $O(|P| f_{\mathcal{A}} + \log w \log |P| \log^* M(\log N + \log |P| \log^* M) + q_w |pOcc_{\mathcal{S}}(P)|)$ time.*

*Proof.* We focus on the case $|Pow_0^P| > 1$ as the other case is easier to be solved. We first compute the common sequence of $P$ in $O(|P| f_{\mathcal{A}})$ time. Taking $\mathcal{P}$ in Lemma 16,

we recall that $|\mathcal{P}| = O(\log|P|\log^* M)$ by Lemma 5. Then, in light of Lemma 16, $pOcc_{\mathcal{G}}(P)$ can be obtained by $|\mathcal{P}| = O(\log|P|\log^* M)$ range reporting queries. For each query, we spend $O(\log w(\log N + \log|P|\log^* M))$ time to compute $x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ by Lemma 15. Hence, the total time complexity is

$$O(|P|f_{\mathcal{A}} + \log|P|\log^* M(\log w(\log N + \log|P|\log^* M) + \hat{q}_w) + q_w|pOcc_{\mathcal{S}}(P)|)$$
$$= O(|P|f_{\mathcal{A}} + \log w \log|P|\log^* M(\log N + \log|P|\log^* M) + q_w|pOcc_{\mathcal{S}}(P)|).$$

□

In order to dynamize our index of Lemma 17, we consider a data structure for "dynamic" two-dimensional orthogonal range reporting that can support the following update operations:

- $insert_{\mathcal{R}}(p, x_{pred}, y_{pred})$: given a point $p = (x, y)$, $x_{pred} = \max\{x' \in \mathcal{X} \mid x' \leq x\}$ and $y_{pred} = \max\{y' \in \mathcal{Y} \mid y' \leq y\}$, insert $p$ to $\mathcal{R}$ and update $\mathcal{X}$ and $\mathcal{Y}$ accordingly.
- $delete_{\mathcal{R}}(p)$: given a point $p = (x, y) \in \mathcal{R}$, delete $p$ from $\mathcal{R}$ and update $\mathcal{X}$ and $\mathcal{Y}$ accordingly.

We use the following data structure for the dynamic two-dimensional orthogonal range reporting.

**Lemma 18 ([5]).** *There exists a data structure that supports* $report_{\mathcal{R}}(x_1, x_2, y_1, y_2)$ *in* $O(\log|\mathcal{R}| + occ(\log|\mathcal{R}|/\log\log|\mathcal{R}|))$ *time, and* $insert_{\mathcal{R}}(p, i, j)$, $delete_{\mathcal{R}}(p)$ *in amortized* $O(\log|\mathcal{R}|)$ *time, where occ is the number of the elements to output. This structure uses* $O(|\mathcal{R}|)$ *space.* [4]

*Proof (Proof of Theorem 1).* Our index consists of a dynamic signature encoding $\mathcal{G}$ and a dynamic range reporting data structure of Lemma 18 whose $\mathcal{R}$ is maintained as they are defined in the static version. We maintain $\mathcal{X}$ and $\mathcal{Y}$ in two ways; self-balancing binary search trees for binary search, and Dietz and Sleator's data structures for order maintenance. Then, primary occurrences of $P$ can be computed as described in Lemma 17. Adding the $O(occ \log N)$ term for computing all pattern occurrences from primary occurrences, we get the time complexity for pattern matching in the statement.

Concerning the update of our index, we described how to update $\mathcal{G}$ after $INSERT$, $INSERT'$ and $DELETE$ in Lemma 12. What remains is to show how to update the dynamic range reporting data structure when a signature is added to or deleted from $\mathcal{V}$. When a signature $e$ is deleted from $\mathcal{V}$, we first locate $e.\text{left}^R$ on $\mathcal{X}$ and $e.\text{right}$ on $\mathcal{Y}$, and then execute $delete_{\mathcal{R}}(e.\text{left}^R, e.\text{right})$. When a signature $e$ is added to $\mathcal{V}$, we first locate $x_{pred} = \max\{x' \in \mathcal{X} \mid x' \leq e.\text{left}^R\}$ on $\mathcal{X}$ and $y_{pred} = \max\{y' \in \mathcal{Y} \mid y' \leq e.\text{right}\}$ on $\mathcal{Y}$, and then execute $insert_{\mathcal{R}}((e.\text{left}^R, e.\text{right}), x_{pred}, y_{pred})$. The locating can be done by binary search on $\mathcal{X}$ and $\mathcal{Y}$ in $O(\log w \log N \log^* M)$ time as Lemma 15.

Since the number of signatures added to or removed from $\mathcal{G}$ during a single update operation is upper bounded by Lemma 11, we can get the desired time bounds of Theorem 1. □

[4] The original problem considers a real plane in the paper [5], however, his solution only need to compare any two elements in $\mathcal{R}$ in constant time. Hence his solution can apply to our range reporting problem by maintains $\mathcal{X}$ and $\mathcal{Y}$ using the data structure of order maintenance problem proposed by Dietz and Sleator [8], which enables us to compare any two elements in a list $L$ and insert/delete an element to/from $L$ in constant time.

## 5 LZ77 factorization in compressed space

In this section, we show Theorem 3. Note that since each $f_i$ can be represented by the pair $(x_i, |f_i|)$, we compute incrementally $(x_i, |f_i|)$ in our algorithm, where $x_i$ is an occurrence position of $f_i$ in $f_1 \cdots f_{i-1}$.

For integers $j, k$ with $1 \leq j \leq j+k-1 \leq N$, let $\mathit{Fst}(j,k)$ be the function which returns the minimum integer $i$ such that $i < j$ and $T[i..i+k-1] = T[j..j+k-1]$, if it exists. Our algorithm is based on the following fact:

**Fact 1** *Let $f_1, \ldots, f_z$ be the LZ77-factorization of a string $T$. Given $f_1, \ldots, f_{i-1}$, we can compute $f_i$ with $O(\log |f_i|)$ calls of $\mathit{Fst}(j,k)$ (by doubling the value of $k$, followed by a binary search), where $j = |f_1 \cdots f_{i-1}| + 1$.*

We explain how to support queries $\mathit{Fst}(j,k)$ using the signature encoding. We define $e.\mathrm{min} = \min \mathit{vOcc}(e, S) + |e.\mathrm{left}|$ for a signature $e \in \mathcal{V}$ with $e \rightarrow e_\ell e_r$ or $e \rightarrow \hat{e}^k$. We also define $\mathit{FstOcc}(P, i)$ for a string $P$ and an integer $i$ as follows:

$$\mathit{FstOcc}(P,i) = \min\{e.\mathrm{min} \mid (e,i) \in \mathit{pOcc}_{\mathcal{G}}(P,i)\}$$

Then $\mathit{Fst}(j,k)$ can be represented by $\mathit{FstOcc}(P,i)$ as follows:

$$\begin{aligned} \mathit{Fst}(j,k) &= \min\{\mathit{FstOcc}(T[j..j+k-1], i) - i \mid i \in \{1, \ldots, k-1\} \\ &= \min\{\mathit{FstOcc}(T[j..j+k-1], i) - i \mid i \in \mathcal{P}\}, \end{aligned}$$

where $\mathcal{P}$ is the set of integers in Lemma 16 with $P = T[j..j+k-1]$.

Recall that in Section 4 we considered the two-dimensional orthogonal range reporting problem to enumerate $\mathit{pOcc}_{\mathcal{G}}(P,i)$. Note that $\mathit{FstOcc}(P,i)$ can be obtained by taking $(e,i) \in \mathit{pOcc}_{\mathcal{G}}(P,i)$ with $e.\mathrm{min}$ minimum. In order to compute $\mathit{FstOcc}(P,i)$ efficiently instead of enumerating all elements in $\mathit{pOcc}_{\mathcal{G}}(P,i)$, we give every point corresponding to $e$ the weight $e.\mathrm{min}$ and use the next data structure to compute a point with the minimum weight in a given rectangle.

**Lemma 19 ([1]).** *Consider $n$ weighted points on a two-dimensional plane. There exists a data structure which supports the query to return a point with the minimum weight in a given rectangle in $O(\log^2 n)$ time, occupies $O(n)$ space, and requires $O(n \log n)$ time to construct.*

Using Lemma 19, we get the following lemma.

**Lemma 20.** *Given a signature encoding $\mathcal{G}$ of size $w$ which generates $T$, we can construct a data structure of $O(w)$ space in $O(w \log w \log N \log^* M)$ time to support queries $\mathit{Fst}(j,k)$ in $O(\log w \log k \log^* M(\log N + \log k \log^* M))$ time.*

*Proof.* For construction, we first compute $e.\mathrm{min}$ in $O(w)$ time using the DAG of $\mathcal{G}$. Next, we prepare the plane defined by the two ordered sets $\mathcal{X}$ and $\mathcal{Y}$ in Section 4. This can be done in $O(w \log w \log N \log^* M)$ time by sorting elements in $\mathcal{X}$ (and $\mathcal{Y}$) by LCE algorithm (Lemma 10) and a standard comparison-based sorting. Finally we build the data structure of Lemma 19 in $O(w \log w)$ time.

To support a query $\mathit{Fst}(j,k)$, we first compute $\mathit{Epow}(\mathit{Uniq}(P))$ with $P = T[j..j+k-1]$ in $O(\log N + \log k \log^* M)$ time by Lemma 8, and then get $\mathcal{P}$ in Lemma 16. Since $|\mathcal{P}| = O(\log k \log^* M)$ by Lemma 5, $\mathit{Fst}(j,k) = \min\{\mathit{FstOcc}(P,i) - i \mid i \in \mathcal{P}\}$ can be computed by answering $\mathit{FstOcc}$ $O(\log k \log^* M)$ times. For each computation of $\mathit{FstOcc}(P,i)$, we spend $O(\log w(\log N + \log k \log^* M))$ time to compute

$x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}$ and $y_2^{(P,j)}$ by Lemma 15, and $O(\log^2 w)$ time to compute a point with the minimum weight in the rectangle $(x_1^{(P,j)}, x_2^{(P,j)}, y_1^{(P,j)}, y_2^{(P,j)})$. Hence it takes $O(\log k \log^* M(\log w(\log N + \log k \log^* M) + \log^2 w)) = O(\log w \log k \log^* M(\log N + \log k \log^* M))$ time in total. □

We are ready to prove Theorem 3 holds.

*Proof (Proof of Theorem 3).* We compute the $z$ factors of the LZ77-factorization of $T$ incrementally by using Fact 1 and Lemma 20 in $O(z \log w \log^3 N(\log^* M)^2)$ time. Therefore the statement holds. □

We remark that we can similarly compute the Lempel-Ziv77 factorization *with* self-reference of a text (defined below) in the same time and same working space.

**Definition 21 (Lempel-Ziv77 factorization with self-reference [29]).** *The Lempel-Ziv77 (LZ77) factorization of a string $s$ with self-references is a sequence $f_1, \ldots, f_k$ of non-empty substrings of $s$ such that $s = f_1 \cdots f_k$, $f_1 = s[1]$, and for $1 < i \leq k$, if the character $s[|f_1..f_{i-1}|+1]$ does not occur in $s[|f_1..f_{i-1}|]$, then $f_i = s[|f_1..f_{i-1}|+1]$, otherwise $f_i$ is the longest prefix of $f_i \cdots f_k$ which occurs at some position $p$, where $1 \leq p \leq |f_1 \cdots f_{i-1}|$.*

**Acknowledgments.** We would like to thank Paweł Gawrychowski for drawing our attention to the work by Alstrup et al. [2,3] and for fruitful discussions.

## References

1. P. K. Agarwal, L. Arge, S. Govindarajan, J. Yang, and K. Yi: *Efficient external memory structures for range-aggregate queries*. Comput. Geom., 46(3) 2013, pp. 358–370.
2. S. Alstrup, G. S. Brodal, and T. Rauhe: *Dynamic pattern matching*, tech. rep., Department of Computer Science, University of Copenhagen, 1998.
3. S. Alstrup, G. S. Brodal, and T. Rauhe: *Pattern matching in dynamic texts*, in Proc. SODA 2000, 2000, pp. 819–828.
4. P. Beame and F. E. Fich: *Optimal bounds for the predecessor problem and related problems*. J. Comput. Syst. Sci., 65(1) 2002, pp. 38–72.
5. G. E. Blelloch: *Space-efficient dynamic orthogonal point location, segment intersection, and range reporting*, in SODA, S.-H. Teng, ed., SIAM, 2008, pp. 894–903.
6. F. Claude and G. Navarro: *Self-indexed grammar-based compression*. Fundamenta Informaticae, 111(3) 2011, pp. 313–337.
7. F. Claude and G. Navarro: *Improved grammar-based compressed indexes*, in SPIRE'12, 2012, pp. 180–192.
8. P. F. Dietz and D. D. Sleator: *Two algorithms for maintaining order in a list*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA, A. V. Aho, ed., ACM, 1987, pp. 365–372.
9. A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S. Woo: *Position heaps: A simple and dynamic text indexing data structure*. J. Discrete Algorithms, 9(1) 2011, pp. 100–121.
10. J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka: *Approximating LZ77 via small-space multiple-pattern matching*, in ESA 2015, 2015, pp. 533–544.
11. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi: *A faster grammar-based self-index*, in LATA'12, 2012, pp. 240–251.
12. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi: *LZ77-based self-indexing with faster pattern matching*, in Proc. LATIN 2014, 2014, pp. 731–742.
13. T. Gagie, P. Gawrychowski, and S. J. Puglisi: *Approximate pattern matching in lz77-compressed texts*. J. Discrete Algorithms, 32 2015, pp. 64–68.

14. K. Goto, S. Maruyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda: *Restructuring compressed texts without explicit decompression.* CoRR, abs/1107.2729 2011.
15. W. Hon, T. W. Lam, K. Sadakane, W. Sung, and S. Yiu: *Compressed index for dynamic text*, in DCC 2004, 2004, pp. 102–111.
16. S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto: *ESP-index: A compressed index based on edit-sensitive parsing.* J. Discrete Algorithms, 18 2013, pp. 100–112.
17. K. Mehlhorn, R. Sundar, and C. Uhrig: *Maintaining dynamic sequences under equality tests in polylogarithmic time.* Algorithmica, 17(2) 1997, pp. 183–198.
18. J. I. Munro, Y. Nekrich, and J. S. Vitter: *Dynamic data structures for document collections and graphs.* CoRR, abs/1503.05977 2015.
19. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Dynamic index and LZ factorization in compressed space.* CoRR, abs/1605.09558 2016.
20. T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Fully dynamic data structure for LCE queries in compressed space.* CoRR, abs/1605.01488 2016.
21. A. Policriti and N. Prezza: *Fast online lempel-ziv factorization in compressed space*, in String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings, C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, eds., vol. 9309 of Lecture Notes in Computer Science, Springer, 2015, pp. 13–20.
22. A. Policriti and N. Prezza: *Computing LZ77 in run-compressed space*, in 2016 Data Compression Conference (DCC 2016), 2016, pp. 23–32, to appear.
23. S. C. Sahinalp and U. Vishkin: *Data compression using locally consistent parsing.* TechnicM report, University of Maryland Department of Computer Science, 1995.
24. S. C. Sahinalp and U. Vishkin: *Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract)*, in FOCS, IEEE Computer Society, 1996, pp. 320–328.
25. H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono: *A space-saving approximation algorithm for grammar-based compression.* IEICE Transactions, 92-D(2) 2009, pp. 158–165.
26. M. Salson, T. Lecroq, M. Léonard, and L. Mouchard: *Dynamic extended suffix arrays.* J. Discrete Algorithms, 8(2) 2010, pp. 241–257.
27. Y. Takabatake, Y. Tabei, and H. Sakamoto: *Improved esp-index: A practical self-index for highly repetitive texts*, in Proc. SEA 2014, 2014, pp. 338–350.
28. Y. Takabatake, Y. Tabei, and H. Sakamoto: *Online self-indexed grammar compression*, in SPIRE 2015, 2015, pp. 258–269.
29. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.

# Algorithms to Compute the Lyndon Array⋆

Frantisek Franek[1], A. S. M. Sohidull Islam[2], M. Sohel Rahman[3], and William F. Smyth[1,3,4]

[1] Algorithms Research Group
Department of Computing & Software
McMaster University, Hamilton, Canada
{franek/smyth}@mcmaster.ca
[2] School of Computational Science & Engineering
McMaster University, Hamilton, Canada
sohansayed@gmail.com
[3] Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology
msrahman@cse.buet.ac.bd
[4] School of Engineering & Information Technology
Murdoch University, Perth, Australia

**Abstract.** In the Lyndon array $\boldsymbol{\lambda} = \boldsymbol{\lambda_x}[1..n]$ of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\boldsymbol{\lambda}[i]$ is the length of the longest Lyndon word starting at position $i$ of $\boldsymbol{x}$. The computation of $\boldsymbol{\lambda}$ has recently become of great interest, since it was shown (Bannai *et al.*, **The "Runs" Theorem**) that the runs in $\boldsymbol{x}$ are computable in linear time from $\boldsymbol{\lambda_x}$. Here we describe two algorithms for computing $\boldsymbol{\lambda_x}$ based on previous results known in different context, but for which no explicit exposition in this context had been given. These two algorithms execute in $\mathcal{O}(n^2)$ time in the worst case. The third algorithm presented that executes in $\Theta(n)$ time had been suggested and discussed previously, and we provide a more substantial discussion and prove of correctness for one of its steps. This algorithm achieves its linearity at the expense of prior computation of both the suffix array and the inverse suffix array of $\boldsymbol{x}$. We then go on to sketch a new algorithm and its two variants that avoids prior computation of global data structures and indicate that in worst-case these algorithms perform in $\mathcal{O}(n \log n)$ time.
**Keywords:** string, Lyndon word, Lyndon array, Lyndon factorization

## 1 Introduction

If $\boldsymbol{x} = \boldsymbol{uv}$ for some $\boldsymbol{u}$ and nonempty $\boldsymbol{v}$, then $\boldsymbol{vu}$ is said to be the $|\boldsymbol{u}|^{\text{th}}$ ***rotation*** of $\boldsymbol{x}$, written $\boldsymbol{vu} = R_{|\boldsymbol{u}|}(x)$. If there exists a string $\boldsymbol{u}$ and an integer $e > 1$ such that $\boldsymbol{x} = \boldsymbol{u}^e$, then $\boldsymbol{x}$ is said to be a ***repetition***; otherwise $\boldsymbol{x}$ is ***primitive***. A primitive string $\boldsymbol{x}$ that is lexicographically strictly least among all its rotations $R_k(\boldsymbol{x}), k = 0, 1, \ldots, |\boldsymbol{x}|-1$, is said to be a ***Lyndon word***.

The ***Lyndon array*** $\boldsymbol{\lambda} = \boldsymbol{\lambda_x}[1..n]$ of a given nonempty string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ gives at each position $i$ the length of the longest Lyndon word starting at $i$. Note that equivalently we could store in the $i$th position of the Lyndon array the end position of the longest Lyndon word starting in $i$. We will use the notation $\mathcal{L}[i]$ to indicate the end position for the longest Lyndon word starting at $i$.

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a & a & b \\ \boldsymbol{\lambda} = & 2 & 1 & 5 & 2 & 1 & 2 & 1 & 3 & 2 & 1 \\ \mathcal{L} = & 2 & 2 & 7 & 5 & 5 & 7 & 7 & 10 & 10 & 10 \end{array} \tag{1}$$

⋆ This work was supported in part by the Natural Sciences & Engineering Research Council of Canada. The authors wish to thank Maxime Crochemore and Hideo Bannai for helpful discussions.

Since being Lyndon really depends on the order of the underlying alphabet of the string, the Lyndon array of a string will change when we change the order of the alphabet. The Lyndon array has recently become of interest since Bannai *et al.* [2] showed that the two Lyndon arrays, one with respect to a given order of the alphabet and the other with respect to the inverse of that order, can be used to compute all the maximal periodicities ("runs") in a string in linear time.

In this paper we describe four algorithms to compute $\boldsymbol{\lambda_x}$. Section 2 makes various observations that apply generally to the Lyndon array and its computation. In Section 3 we describe two algorithms that are based on previous results known in a different context, and we present them here explicitly in the context of computing Lyndon arrays. These two algorithms perform in $\mathcal{O}(n^2)$ time in the worst case, where $n$ is the length of the input string. Despite the high worst case complexity, in practice these algorithms perform very well as they are simple and straightforward to implement and do not require any complicated data structures; they could be characterized almost as in-place. The third algorithm discussed in this section had been described previously and we provide a more substantial discussion and prove correctness of one of its steps that we could not find anywhere in the literature. This algorithm is simple and worst-case linear-time, but requires suffix array construction and so is a little slower. Section 4 describes two variants of an algorithm we designed that uses only elementary data structures (no suffix arrays). One variant is $\mathcal{O}(n^2)$ in the worst case, the other indicates $\mathcal{O}(n \log n)$ time, but with no clear advantage in processing time. Section 5 describes the results of preliminary experiments on the algorithms; Section 6 outlines future work.

## 2 Preliminaries

Here we make various observations that apply to the algorithms described below.

**Observation 1** *Let $\boldsymbol{x} = \boldsymbol{w_1 w_2 \cdots w_k}$ be the Lyndon decompostion [5,9] of $\boldsymbol{x}$, with Lyndon words $\boldsymbol{w_1} \geq \boldsymbol{w_2} \geq \cdots \geq \boldsymbol{w_k}$. Then every Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]]$ of length $\boldsymbol{\lambda}[i]$ is a substring of some $\boldsymbol{w_h}$, $h \in 1..k$.*

*Proof.* For some $h \in 1..k-1$, consider $\boldsymbol{w_h}$ with a nonempty proper suffix $\boldsymbol{v_h}$, and for some $t \in 1..k-h$, consider $\boldsymbol{w_{h+t}}$ with a nonempty prefix $\boldsymbol{u_{h+t}}$. Since $\boldsymbol{w_h}$ is a Lyndon word, $\boldsymbol{w_h} < \boldsymbol{v_h}$, and by lexorder, $\boldsymbol{u_{h+t}} \leq \boldsymbol{w_{h+t}}$. Thus $\boldsymbol{v_h} > \boldsymbol{w_h} \geq \boldsymbol{w_{h+t}} \geq \boldsymbol{u_{h+t}}$, and so $\boldsymbol{v_h w_{h+1} \cdots w_{h+t-1} u_{h+t}}$ cannot be a Lyndon word for any choice of $h$ or $t$. □

Therefore to compute $\mathcal{L}_{\boldsymbol{x}}$ it suffices to consider separately each distinct element $\boldsymbol{w_h}$ in the Lyndon decomposition of $\boldsymbol{x}$. Hence, without loss of generality we suppose that $\boldsymbol{x}$ is a Lyndon word and write it in the form $\boldsymbol{x_1 x_2 \cdots x_m}$, where for each $r \in 1..m$, $|\boldsymbol{x_r}| = \ell_r$ and

$$\boldsymbol{x_r}[1] \leq \boldsymbol{x_r}[2] \leq \cdots \leq \boldsymbol{x_r}[\ell_r], \tag{2}$$

while for $1 \leq r < m$,

$$\boldsymbol{x_r}[\ell_r] > \boldsymbol{x_{r+1}}[1]. \tag{3}$$

We call $\boldsymbol{x_r}$ a ***range*** in $\boldsymbol{x}$ and the boundary between $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$ a ***drop***. We identify a position $j$ in range $\boldsymbol{x_r}$, $1 \leq j \leq \ell_r$, with its equivalent position $i$ in $\boldsymbol{x}$ by writing $i = S_{r,j} = \sum_{r'=1}^{r-1} \ell_{r'} + j$.

**Observation 2** *Let $i = S_{r,j}$ be a position in $\boldsymbol{x}$ that corresponds to position $j$ in range $\boldsymbol{x_r}$.*

(a) *If $\boldsymbol{x_r}[j] = \boldsymbol{x_r}[\ell_r]$, then $\mathcal{L}[i] = i$.*
(b) *Otherwise, $\mathcal{L}[i] = i'$, where $i'$ is the final position in some range $\boldsymbol{x_{r'}}$, $r' \geq r$; that is, $i' = \sum_{s=1}^{r'} \ell_s$.*

*Proof.* (a) is an immediate consequence of (2) and (3). To prove (b), suppose that $\boldsymbol{x}[i..\mathcal{L}[i]]$ is a maximum-length Lyndon word, where $\mathcal{L}[i]$ falls within range $r'$ but $\mathcal{L}[i] < i'$. Since by (2) $\boldsymbol{x}[\mathcal{L}[i]] \leq \boldsymbol{x}[\mathcal{L}[i]+1]$, there are two consecutive Lyndon words $\boldsymbol{x}[i..\mathcal{L}[i]], \boldsymbol{x}[\mathcal{L}[i]+1]$ that by the Lyndon decomposition theorem [5] can be merged into a single Lyndon word $\boldsymbol{x}[i..\mathcal{L}[i]+1]$. Thus $\boldsymbol{x}[i..\mathcal{L}[i]]$ is not maximum-length, a contradiction. □

We see then that if $\boldsymbol{x_r}[j] < \boldsymbol{x_r}[\ell_r]$, then $\boldsymbol{x_r}[j..\ell_r]$ is a (not necessarily maximum-length) Lyndon word, and for $i = S_{r,j}$, $\mathcal{L}[i] \geq S_{r,\ell_r}$:

$$\begin{array}{rccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \boldsymbol{x} = & a & a & a & b| & a & a & b| & a & b| & a & a & b & b \\ \mathcal{L} = & 13 & 13 & 4 & 4 & 9 & 7 & 7 & 9 & 9 & 13 & 13 & 12 & 13 \end{array} \tag{4}$$

More generally, the integer interval $\langle i, \mathcal{L}[i] \rangle = i..\mathcal{L}[i]$ satisfy a "Monge" property that is exploited by Algorithm NSV* (Section 4):

**Observation 3** *Suppose positions $i, j$ in $\boldsymbol{x}[1..n]$ satisfy $1 \leq i < j \leq n$. Then either $\mathcal{L}[i] \leq j$ or $\mathcal{L}[i] \geq \mathcal{L}[j]$: the intervals $\langle i, \mathcal{L}[i] \rangle$ and $\langle j, \mathcal{L}[j] \rangle$ are not overlapping.*

*Proof.* Suppose two such intervals do overlap. Then the maximum-length Lyndon words $\boldsymbol{w_1} = \boldsymbol{x}[i..\mathcal{L}[i]]$ and $\boldsymbol{w_2} = \boldsymbol{x}[j..\mathcal{L}[j]]$ have a nonempty overlap, so that we can write $\boldsymbol{w_1} = \boldsymbol{uv}$, $\boldsymbol{w_2} = \boldsymbol{vv'}$ for some nonempty $\boldsymbol{v}$. But then, by well-known properties of Lyndon words, $\boldsymbol{w_1} < \boldsymbol{v} < \boldsymbol{w_2} < \boldsymbol{v'}$, implying that $\boldsymbol{w_1 v'}$ is a Lyndon word, contradicting the assumption that $\boldsymbol{w_1}$ is of a maximal length. □

Expressing a string in terms of its ranges has the same useful lexorder property that writing it in terms of its letters does:

**Observation 4** *Suppose strings $\boldsymbol{x}$ and $\boldsymbol{y}$ are expressed in terms of their ranges: $\boldsymbol{x} = \boldsymbol{x_1 x_2} \cdots \boldsymbol{x_m}$, $\boldsymbol{y} = \boldsymbol{y_1 y_2} \cdots \boldsymbol{y_n}$. Suppose further that for some least integer $r \in 1..\min(m, n)$, $\boldsymbol{x_r} \neq \boldsymbol{y_r}$. Then $\boldsymbol{x} < \boldsymbol{y}$ (respectively, $\boldsymbol{x} > \boldsymbol{y}$) according as $\boldsymbol{x_r} < \boldsymbol{y_r}$ (respectively, $\boldsymbol{x_r} > \boldsymbol{y_r}$).*

*Proof.* If $\boldsymbol{x_r} < \boldsymbol{y_r}$, then either

(a) $\boldsymbol{x_r}$ is a nonempty proper prefix of $\boldsymbol{y_r}$; or
(b) there is some least position $j$ such that $\boldsymbol{x_r}[j] < \boldsymbol{y_r}[j]$.

In case (a), if $r = m$, then $\boldsymbol{x}$ is actually a prefix of $\boldsymbol{y}$, so that $\boldsymbol{x} < \boldsymbol{y}$, while if $r < m$, then by (3), $\boldsymbol{x_{r+1}}[1] < \boldsymbol{y_r}[|\boldsymbol{x_r}|+1]$, and again $\boldsymbol{x} < \boldsymbol{y}$. In case (b) the result is immediate. The proof for $\boldsymbol{x_r} > \boldsymbol{y_r}$ is similar. □

## 3 Basic Algorithms

Here we outline three algorithms for which no clear exposition in the context of Lyndon arrays is available in the literature. We remark that the Lyndon array computation is equivalent to "Lyndon bracketing", for which an $\mathcal{O}(n^2)$ algorithm was described in [17].

### 3.1 Folklore — Iterated MaxLyn

This algorithm, see Figure 1, is based on Duval's linear time algorithm for Lyndon factorization, [9] – it is the application of its first step which we refer to as **MaxLyn** since it returns the size of the longest Lyndon word starting at that position. This process is iterated for all positions in the input string and this thus gives immediately $O(|x|^2)$ worst case complexity for an input string $x$. Since Duval's algorithm is in-place, this algorithm is simple and almost in-place, except the space for the Lyndon array. Below, we sketch the reasons the algorithm provides the correct answer.

For a string $\boldsymbol{x}$ of length $n$, recall that the ***prefix table*** $\pi[1..n]$ is an integer array in which for every $i \in 1\,..\,n$, $\pi[i]$ is the length of the longest substring beginning at position $i$ of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. Given a nonempty string $\boldsymbol{x}$ on alphabet $\Sigma$, let us define $\boldsymbol{x'} = \boldsymbol{x}\$$, where the sentinel $\$ < \mu$ for every letter $\mu \in \Sigma$.

**Observation 5** $\boldsymbol{x}$ *is a Lyndon word if and only if for every* $i \in 2\,..\,n$, $\boldsymbol{x'}[1+k] < \boldsymbol{x'}[i+k]$, *where* $k = \pi[i]$.

This result forms the basis of the algorithm given in Figure 1 that computes the length max $\in 1\,..\,n-j+1$ of the longest Lyndon factor at a given position $j$ in $\boldsymbol{x}[1..n]$. Its efficiency is a consequence of the instruction $i \leftarrow i+k+1$ that skips over positions in the range $i+1\,..\,i+k-1$, effectively assuming that for every position $i^*$ in that range, $i^* + \pi[i^*] \le i+k$. Lemma 11, given in Appendix 1, justifies this assumption. Simply repeating MaxLyn at every position $j$ of $\boldsymbol{x}$ gives a simple, fast $\mathcal{O}(n^2)$ time and $\mathcal{O}(1)$ additional space algorithm to compute $\boldsymbol{\lambda_x}$.

```
procedure MaxLyn(x[1..n], j, Σ, ≺) : integer
i ← j + 1; max ← 1
while i ≤ n do
  k ← 0
  while x′[j + k] = x′[i + k] do
    k ← k + 1
  if x′[j + k] ≺ x′[i + k] then
    i ← i + k + 1; max ← i − 1
  else
    return max
```

**Figure 1.** Algorithm MaxLyn

Recent work on the prefix table [4,6] has confirmed its importance as a data structure for string algorithms. In this context it is interesting to find that Lyndon words $\boldsymbol{x}$ can be characterized in terms of $\pi_{\boldsymbol{x}}$:

**Observation 6** *Suppose* $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ *is a string on alphabet* $\Sigma$ *such that* $\boldsymbol{x}[1]$ *is the least letter in* $\boldsymbol{x}$. *Then* $\boldsymbol{x}$ *is a Lyndon word over* $\Sigma$ *if and only if for every* $i \in 2\,..\,n$,

(*a*) $i + \pi_{\boldsymbol{x}}[i] < n+1$; *and*

(*b*) *for every* $j \in i+1\,..\,i+\pi_{\boldsymbol{x}}[i]-1$, $j+\pi_{\boldsymbol{x}}[j] \le i+\pi_{\boldsymbol{x}}[i]$.

In Appendix 1, the reader can find an additional result that justifies the strategy employed by **MaxLyn** (Figure 1).

### 3.2 Recursive Duval Factorization: Algorithm RDuval

Rather than independently computing the maximum-length Lyndon factor at each position $i$, as MaxLyn does, Algorithm RDuval recursively computes the Lyndon decomposition, [9], into maximum factors, at each step taking advantage of the fact that $\mathcal{L}[i]$ is known for the first position $i$ in each factor, then recomputing with the first letters removed. This again gives immediate worst case complexity of $O(n^2)$. We consider it only because it allows for a more refined discussion of the complexity in special cases for strings over binary alphabets giving an average case complexity of $\mathcal{O}(n \log n)$, see below.

By Observation 1, whenever $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a Lyndon word, we know that $\mathcal{L}[1] = n$. Thus computing the Lyndon decomposition $\boldsymbol{x} = \boldsymbol{w_1 w_2} \cdots \boldsymbol{w_k}$, $\boldsymbol{w_1} \ge \boldsymbol{w_2} \ge \cdots \ge \boldsymbol{w_k}$, allows us to assign $\boldsymbol{\lambda}[i_j] = |\boldsymbol{w_j}|$, where $i_j$ is the first position of $\boldsymbol{w_j}$, $j = 1, 2, \ldots, k$.

Algorithm RDuval applies this strategy recursively, by assigning $\boldsymbol{\lambda}[i_j] \leftarrow |\boldsymbol{w_j}|$, then removing the first letter $i_j$ from each $\boldsymbol{w_j}$ to form $\boldsymbol{w'_j}$, to which the Lyndon decomposition is applied in the next recursive step. This process continues until each Lyndon word is reduced to a single letter.

The asymptotic time required for RDuval is bounded above by $n$ times the maximum depth of the recursion, thus $O(n^2)$ in the worst case — consider, for example, the string $\boldsymbol{x} = a^{n-1}b$. However, to estimate expected behaviour, we can make use of a result of Bassino *et al.* [3]. Given a Lyndon word $\boldsymbol{w}$, they call $\boldsymbol{w} = \boldsymbol{uv}$ the ***standard factorization*** of $\boldsymbol{w}$ if $\boldsymbol{u}$ and $\boldsymbol{v}$ are both Lyndon words and $\boldsymbol{v}$ is of maximum size. They then show that if $\boldsymbol{w}$ is a binary string ($\Sigma = \{a, b\}$), the average length of $\boldsymbol{v}$ is asymptotically $3|\boldsymbol{w}|/4$. Thus each recursive application of RDuval yields a left Lyndon factor of expected length $|\boldsymbol{w}|/4$ and a remainder of length $3|\boldsymbol{w}|/4$ to be factored further. It follows that the expected number of recursive calls of RDuval is $\mathcal{O}(\log_{4/3} n)$. Hence

**Lemma 7** *On binary strings RDuval executes in* $O(n \log_{4/3} n)$ *time on average.*

**Example 8** *For*

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \boldsymbol{x} = & a & a & b & a & a & b & b & a & b & b & a & b \\ \boldsymbol{\lambda} = & 12 & 2 & 1 & 9 & 3 & 1 & 1 & 3 & 1 & 1 & 2 & 1 \end{array}$$

*the factors considered are first 1–12, then*

- *2–3 and 4–12 in the first level of recursion;*
- *3, 5–7, 8–10 and 11–12 in the second level;*
- $6, 7, 9, 10, 12$ *in the third level.*

*Positions are assigned as follows:* $\boldsymbol{\lambda}[1] \leftarrow 12; \boldsymbol{\lambda}[2] \leftarrow 2, \boldsymbol{\lambda}[4] \leftarrow 9; \boldsymbol{\lambda}[3] \leftarrow 1, \boldsymbol{\lambda}[5] \leftarrow 3, \boldsymbol{\lambda}[8] \leftarrow 3, \boldsymbol{\lambda}[11] \leftarrow 2; \boldsymbol{\lambda}[6] \leftarrow 1, \boldsymbol{\lambda}[7] \leftarrow 1, \boldsymbol{\lambda}[9] \leftarrow 1, \boldsymbol{\lambda}[10] \leftarrow 1, \boldsymbol{\lambda}[12] \leftarrow 1.$

### 3.3 NSV Applied to the Inverse Suffix Array

The idea of the "next smaller value" (NSV) array for a given array $\boldsymbol{x}$ had been proposed in various forms and under various names [1,10,11,15].

**Definition 9 (Next Smaller Value)** *Given an array* $\boldsymbol{x}[1..n]$ *of ordered values,* $NSV = NSV_{\boldsymbol{x}}[1..n]$ *is the* ***next smaller value array*** *of* $\boldsymbol{x}$ *if and only if for every* $i \in 1..n$*,* $NSV[i] = j$*, where*

$(a)$ *for every* $h \in 1..j{-}1$*,* $\boldsymbol{x}[i] \le \boldsymbol{x}[i{+}h]$*; and*
$(b)$ *either* $i{+}j = n{+}1$ *or* $\boldsymbol{x}[i] > \boldsymbol{x}[i{+}j]$*.*

**Example 10**

$$\begin{array}{rl} & \scriptstyle 1\;2\;3\;\;4\;\;5\;6\;7\;8\;9\;10 \\ \boldsymbol{x} = & 3\;8\;7\;10\;2\;1\;4\;9\;6\;5 \\ NSV_{\boldsymbol{x}} = & 4\;1\;2\;\;1\;\;1\;5\;4\;1\;1\;\;1 \end{array}$$

As shown in various contexts in [11], NSV$_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ time using a stack. Our main observation here, also mentioned in [12], is that $\boldsymbol{\lambda_x}$ can be computed merely by applying NSV to the inverse suffix array ISA$_{\boldsymbol{x}}$. Proof of this claim can be found in Appendix 2; here we present the very simple $\Theta(n)$-time, $\Theta(n)$-space algorithm for this calculation:

```
procedure NSVISA(x[1..n]) : λx[1..n]
Compute SAx    (see [14,16])
Compute ISAx from SAx in place    (see [16])
λx ← NSV(ISAx) (in place)
```

**Figure 2.** Apply NSV to ISA$_{\boldsymbol{x}}$

# 4 Elementary Computation of $\boldsymbol{\lambda_x}$ Using Ranges

In this section we describe an approach to the computation of $\boldsymbol{\lambda_x}$ that applies a variant of the NSV idea to the ranges of $\boldsymbol{x}$. Figure 3 gives pseudocode for Algorithm NSV$^*$ that uses the NSV stack ACTIVE to compute $\boldsymbol{\lambda}$. The processing identifies ranges in a single left-to-right scan of $\boldsymbol{x}$, making use of two range comparison routines, COMP and MATCH. COMP compares adjacent individual ranges $\boldsymbol{x_r}$ and $\boldsymbol{x_{r+1}}$, returning $\delta_1 = -1, 0, +1$ according as $\boldsymbol{x_r} < \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} = \boldsymbol{x_{r+1}}$, $\boldsymbol{x_r} > \boldsymbol{x_{r+1}}$. MATCH similarly returns $\delta_2$ for adjacent *sequences* of ranges; that is,

$$\boldsymbol{X_r} = \boldsymbol{x_r x_{r+1} \cdots x_{r+s}}, \text{ for some } s \ge 1;$$
$$\boldsymbol{X_{r+s+1}} = \boldsymbol{x_{r+s+1} x_{r+s+2} \cdots x_{r+s+t}}, \text{ for some } t \ge 1.$$

Algorithm NSV$^*$ is based on the idea encapsulated in Lemma 15 of Appendix 2, the main basis of the correctness of Algorithm NSVISA (see Figure 2). We process $\boldsymbol{x}$ from left to right, using a stack ACTIVE initialized with index 1. At each iteration, the top of the stack (say, $j$) is compared with the current index (say, $i$). In particular, we need to compare $\boldsymbol{s_x}(i)$ with $\boldsymbol{s_x}(j)$, where $\boldsymbol{s_x}(i) \equiv \boldsymbol{x}[i..n]$. As long as $\boldsymbol{s_x}(i) \succeq \boldsymbol{s_x}(j)$, NSV$^*$ pushes the current index and continues to the next. When $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(j)$, it pops the stack and puts appropriate values in the corresponding indices of $\boldsymbol{\lambda_x}$.

```
procedure NSV* (x, λ)
nextequal ← 0^n;  period ← 0^n
push(ACTIVE) ← 1
▷ x[n+1] = $, a letter smaller than any in Σ.
for i ← 2 to n+1 do
  prev ← 0; j ← peek(ACTIVE)
▷ COMP compares suffixes specified by i, j of two ranges.
  δ1 ← COMP(x[j], x[i]);  δ2 ← 1
    while (δ1 ≥ 0 and δ2 > 0) do
      if δ1 = 0 then δ2 ← MATCH(x[j], x[i])
      if δ2 > 0 then
        if prev = 0 or nextequal[j] ≠ prev then λ[j] ← i−j
        else
          λ[j] ← offset ← prev−j
          if period[prev] = 0 then
            if λ[prev] > offset then
              λ[j] ← λ[j]+λ[prev]
          else
            if nextequal[j] = prev and offset ≠ λ[prev] then
              λ[j] ← λ[j]+period[prev]
          if λ[prev] = offset then
        ▷ Current position is a part of periodic substring
            if period[prev] = 0 then
              period[j] ← period[prev] + 2 × offset
            else
              period[j] ← period[prev]+offset
        pop(ACTIVE)
        prev ← j;  j ← peek(ACTIVE)
        ▷ Empty stack implies termination.
        if j = 0 then EXIT
        δ1 ← COMP(x[j], x[i])
▷ Finished processing i — it goes to stack.
  if δ2 = 0 then nextequal[j] ← i
  push(ACTIVE) ← i
```

**Figure 3.** Computing $\boldsymbol{\lambda}_x$ using modified NSV

As noted above, especially Observations 1–3, ranges are employed to expedite these suffix comparisons.

Two auxiliary arrays, **nextequal** and **period**, are required to handle situations in which MATCH finds that a suffix of a previous range at position $j$ equals the current range at position $i$. Thus, when $\delta_2 = 0$, the algorithm assigns nextequal$[j] \leftarrow i$ before $i$ is pushed onto ACTIVE. Then when a later MATCH yields $\delta_2 = 0$, the value of **period** — that is, the extent of the following periodicity — may need to be set or adjusted, as shown in the following example:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x}$ = | $a$ | $a$ | $a$ | $b$ | $a$ | $a$ | $b$ | $a$ | $a$ | $b$ | $a$ | $a$ | $b$ | $a$ | $b$ |
| nextequal = | 0 | 5 | 0 | 0 | 8 | 0 | 0 | 11 | 0 | 0 | 0 | 14 | 0 | 0 | 0 |
| period = | 0 | 12 | 0 | 0 | 9 | 0 | 0 | 6 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |

A straightforward implementation of COMP and MATCH could require a number of letter comparisons equal to the length of the shorter of the two sequences of ranges being matched. However, by performing $\Theta(n)$-time preprocessing, we can compare two ranges in $\mathcal{O}(\sigma)$ time, where $\sigma = |\Sigma|$ is the alphabet size. Given $\Sigma = \{\mu_1, \mu_2, \ldots, \mu_\sigma\}$,

we define Parikh vectors $P_r[1..\sigma]$, where $P_r[j]$ is the number of occurrences of $\mu_j$ in range $\boldsymbol{x_r}$. Since ranges are monotone nondecreasing in the letters of the alphabet, it is easy to compute all the $P_r, r = 1, 2, \ldots, m$, in linear time in a single scan of $\boldsymbol{x}$. Similarly, during the processing of each range $\boldsymbol{x_r}$, any value $P_{r,j}$, the Parikh vector of the suffix $\boldsymbol{x_r}[j..\ell_r]$, can be computed in constant time for each position considered. Thus we can determine the lexicographical order of any two ranges (or part ranges) $\boldsymbol{x_r}$ and $\boldsymbol{x_{r'}}$ in $\mathcal{O}(\sigma)$ time rather than time $\mathcal{O}(\max(\ell_r, \ell_{r'}))$. The variant of NSV$^*$ that uses Parikh vectors is called PNSV$^*$; otherwise NPNSV$^*$ for Not Parikh.

In Appendix 3 we describe briefly another approach to this suffix comparison problem, which we believe achieves run time $\mathcal{O}(n \log n)$ by maintaining a simple data structure requiring $\mathcal{O}(n \log n)$ space.

Now consider the worst case behaviour of Algorithm NSV$^*$. Given the initial string $\boldsymbol{x_0} = a^h b a^h c_0,\ h \geq 1,\ c_0 > b > a$, let $\boldsymbol{x_k^{(h)}} = \boldsymbol{x_k} = \boldsymbol{x_{k-1}} \boldsymbol{x^*_{k-1}},\ k = 1, 2, \ldots$, with $\boldsymbol{x^*_{k-1}}$ identical to $\boldsymbol{x_{k-1}}$ except in the last position, where the letter $c_k > c_{k-1}$ replaces $c_{k-1}$. Then $\boldsymbol{x_k}$ has length $n = (h+1)m$, where $m = 2^{k+1}$ is the number of ranges in $\boldsymbol{x_k}$. We believe and are working towards a proof that $\boldsymbol{x_k}$ is a worst-case input for Algorithm NSV$^*$, which requires $\mathcal{O}(n \log n)$ range matches in such cases. Since PNSV$^*$ compares two ranges in $\mathcal{O}(\sigma)$ time, it therefore would require $\mathcal{O}(\sigma n \log n)$ time in the worst case, thus $\mathcal{O}(n \log n)$ for constant $\sigma$.

## 5 Preliminary Experimental Results

We have done some preliminary tests on the implementations of the two variants of NSV$^*$, with and without employing Parikh vectors. The equipment used was an Intel(R) Core i3 at 1.8GHz and 4GB main memory under a 64-bit Windows 7 operating system. For each length $10000, 20000, \ldots, 100000$ we generated 500 random strings for alphabets of sizes $\sigma = 2, 4$ and 8. The results indicate, that at least for random strings, the processing time seems linear. The processing time for "with Parikh vectors" is greater because of the initial pre-processing. The data and the corresponding graphs are in Figures 4..11 below.

Processing time versus length of the string for the program without Parikh vectors

| length of the input strings | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 | 100000 |
|---|---|---|---|---|---|---|---|---|---|---|
| alphabet size 2 | 0.001122 | 0.002265 | 0.003335 | 0.004444 | 0.005552 | 0.006682 | 0.007813 | 0.008936 | 0.010117 | 0.011167 |
| alphabet size 4 | 0.001103 | 0.002233 | 0.003345 | 0.004464 | 0.0056 | 0.006677 | 0.007792 | 0.008903 | 0.010036 | 0.011157 |
| alphabet size 8 | 0.001067 | 0.002112 | 0.003181 | 0.004255 | 0.005337 | 0.006372 | 0.007418 | 0.008522 | 0.009646 | 0.010653 |

**Figure 4.** Processing times in seconds for the implementation without Parikh vectors

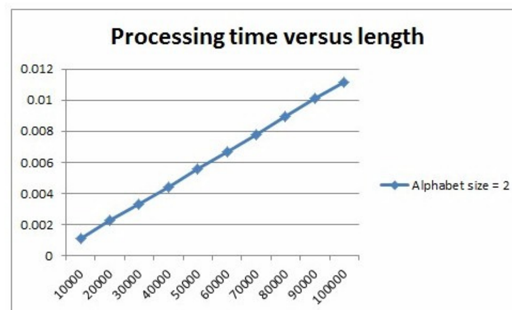



**Figure 5.** Processing times for random strings over the binary alphabet; without Parikh vectors

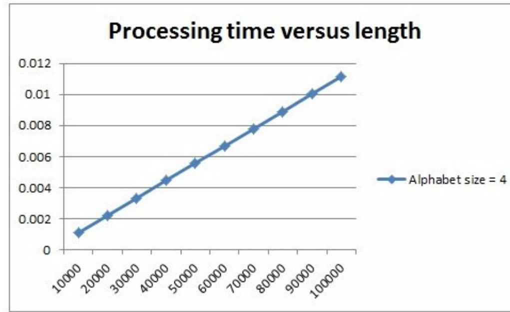



**Figure 6.** Processing times for random strings over the alphabet of size 4; without Parikh vectors

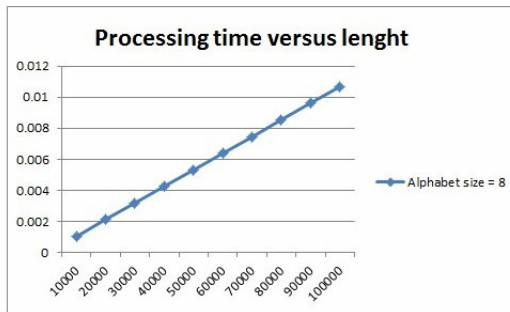



**Figure 7.** Processing times for random strings over the alphabet of size 8; without Parikh vectors

Processing time versus length of the string for the program with Parikh vectors

| length of the input strings | 10000 | 20000 | 30000 | 40000 | 50000 | 60000 | 70000 | 80000 | 90000 | 100000 |
|---|---|---|---|---|---|---|---|---|---|---|
| alphabet size 2 | 0.001368 | 0.002724 | 0.004072 | 0.005413 | 0.006782 | 0.008304 | 0.009465 | 0.010868 | 0.012201 | 0.013669 |
| alphabet size 4 | 0.001898 | 0.003782 | 0.005694 | 0.007516 | 0.009498 | 0.011421 | 0.01319 | 0.015226 | 0.017124 | 0.019068 |
| alphabet size 8 | 0.002385 | 0.004721 | 0.007053 | 0.009409 | 0.011874 | 0.014219 | 0.016546 | 0.018967 | 0.021337 | 0.024008 |

**Figure 8.** Processing times in seconds for the implementation with Parikh vectors

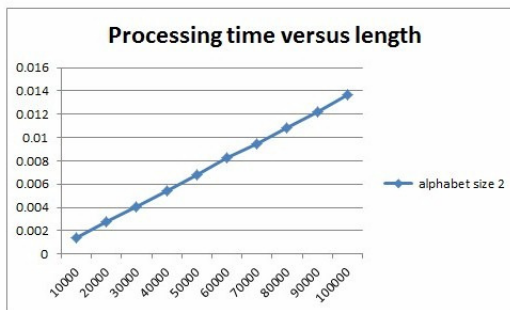



**Figure 9.** Processing times for random strings over the binary alphabet; with Parikh vectors

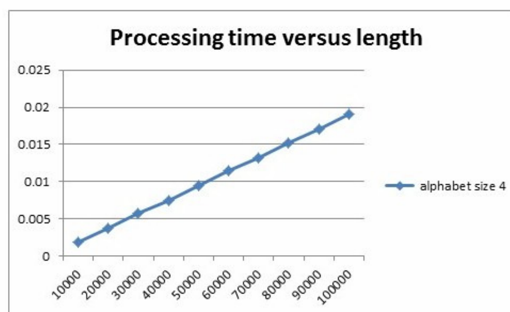



**Figure 10.** Processing times for random strings over the alphabet of size 4; with Parikh vectors

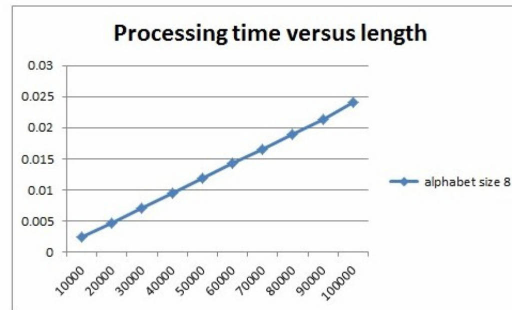



**Figure 11.** Processing times for random strings over the alphabet of size 8; with Parikh vectors

## 6 Future Work

There is reason to believe [13] that the Lyndon array computation is less hard than suffix array construction. Thus the authors conjecture that there is a linear-time elementary algorithm (no suffix arrays) to compute the Lyndon array.

## References

1. S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe: *Nearest common ancestors: a survey and new distributed algorithm*, in Proc. 1th Annual ACM Symp. on Parallel Algorithms & Architectures, 2002, pp. 258–264.
2. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The “runs” theorem*, 2014, *arXiv:1406.0263v6*.
3. F. Bassino, J. Clément, and C. Nicaud: *The standard factorization of Lyndon words: an average point of view.* Discrete Mathematics, 290(1) 2005, pp. 1–25.
4. W. Bland, G. Kucherov, and W. F. Smyth: *Prefix table construction and conversion.* Proc. 24th Internat. Workshop on Combinatorial Algs. (IWOCA), 2013, pp. 41–53.
5. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. iv. the quotient groups of the lower central series.* Annals of Mathematics, 68(1) 1958, pp. 81–95.
6. M. Christodoulakis, P. J. Ryan, W. F. Smyth, and S. Wang: *Indeterminate strings, prefix arrays and undirected graphs.* Theoretical Comput. Sci., 600 2015, pp. 34–48.
7. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, New York, NY, USA, 2007.
8. M. Crochemore and W. Rytter: *Jewels of stringology*, World Scientific, 2002.
9. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
10. J. Fischer, V. Mäkinen, and G. Navarro: *An(other) entropy-based compressed suffix tree*, in 19th Annual Symp. on Combinatorial Pattern Matching, vol. 5029 of Lecture Notes in Computer Science, Springer, 2008, pp. 152–165.
11. K. Goto and H. Bannai: *Simpler and faster Lempel-Ziv factorization*, in Data Compression Conference, 2013, pp. 133–142.
12. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theor. Comput. Sci., 307(1) 2003, pp. 173–178.
13. D. Kosolobov: *Lempel-Ziv factorization may be harder than computing all runs*, in Proc. 32nd Symp. on Theoretical Aspects of Computer Science, 2015, *arXiv:1409.5641*.
14. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced–sorting.* Data Compression Conference, 0 2009, pp. 193–202.
15. E. Ohlebusch and S. Gog: *Lempel-Ziv factorization revisited*, in 22nd Annual Symp. on Combinatorial Pattern Matching, vol. 6661 of Lecture Notes in Computer Science, Springer, 2011, pp. 15–26.
16. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39(2) July 2007, pp. 1–31.
17. J. Sawada and F. Ruskey: *Generating Lyndon brackets: an addendum to “Fast algorithms to generate necklaces, unlabeled necklaces amd irreducible polynomials over GF(2)”.* J. Algorithms, 46 2003, pp. 21–26.

# Appendix 1

The following result justifies the strategy employed in Algorithm MaxLyn (Figure 1):

**Lemma 11** *Suppose that for some position $i$ in a Lyndon word $\boldsymbol{x}[1..n]$, $k = \pi[i] \geq 2$. Then for every $j \in i+1\,..\,i+k-1$, $\pi[j] \leq i+k-j$.*

*Proof.* The result certainly holds for $i+k = n+1$, so we consider $i+k \leq n$. Assume that for some $j \in i+1\,..\,i+k-1$, $\pi[j] > i+k-j$. It follows that

$$\boldsymbol{x}[1\,..\,i+k-j+1] = \boldsymbol{x}[j\,..\,i+k], \tag{5}$$

while $\boldsymbol{x}[j-i+1\,..\,k] = \boldsymbol{x}[j\,..\,i+k-1]$. Since $\boldsymbol{x}$ is Lyndon, therefore $\boldsymbol{x}[1+k] \prec \boldsymbol{x}[i+k]$, and so we find that

$$\boldsymbol{x}[j-i+1\,..\,1+k] \prec \boldsymbol{x}[j\,..\,i+k]. \tag{6}$$

From (5) and (6) we see that $\boldsymbol{x}[1..k+1]$ has suffix $\boldsymbol{x}[j-i+1..k+1]$ satisfying $\boldsymbol{x}[j-i+1..k+1] \prec \boldsymbol{x}[1..i+k-j+1]$, contradicting the assumption that $\boldsymbol{x}$ is Lyndon.
□

# Appendix 2

Here we prove Theorem 12 that justifies th algorithm given in Figure 2:

**Theorem 12** *For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on alphabet $\Sigma$, totally order by $\prec$, let $ISA = ISA^{\prec}_{\boldsymbol{x}}$. Then for every $i \in 1..n$, the substring $\boldsymbol{x}[i..j]$ is a longest Lyndon factor with respect to $\prec$ if and only if*

(a) *for every $h \in i{+}1..j$, $ISA[j] < ISA[h]$; and*
(b) *either $j = n$ or $ISA[j{+}1] < ISA[i]$.*

The following well-known result is needed to prove Lemma 14:

**Lemma 13 (Duval, Lemma 1.6, [9])** *Suppose $\boldsymbol{x} \in \Sigma^+$, where $\Sigma$ is an alphabet totally ordered by $\prec$. Let $\boldsymbol{x} = \boldsymbol{u}^r\boldsymbol{u_1}b$, where $\boldsymbol{u}$ is nonempty, $r \geq 1$, $\boldsymbol{u_1}$ a possibly empty proper prefix of $\boldsymbol{u}$, and the letter $b \neq \boldsymbol{u}[|\boldsymbol{u_1}|{+}1]$.*

(a) *If $b \prec \boldsymbol{u}[|\boldsymbol{u_1}|{+}1]$, then $\boldsymbol{u}$ is a longest Lyndon prefix of $\boldsymbol{xy}$ for any $\boldsymbol{y}$;*
(b) *if $b \succ \boldsymbol{u}[|\boldsymbol{u_1}|{+}1]$, then $\boldsymbol{x}$ is Lyndon with respect to $\prec$.*

For a given string $\boldsymbol{x}[1..n]$, let $\boldsymbol{s_x}(i) = \boldsymbol{x}[i..n]$ denote the suffix of $\boldsymbol{x}$ beginning at position $i$. When clear from context we write just $\boldsymbol{s}(i)$.

**Lemma 14** *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over alphabet $\Sigma$ totally ordered by $\prec$. Let $\boldsymbol{x}[i\,..\,j]$ be the longest Lyndon factor of $\boldsymbol{x}$ starting at $i$. Then $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(k)$ for every $k \in i{+}1..j$ and either $j = n$ or $\boldsymbol{s_x}(j{+}1) \prec \boldsymbol{s_x}(i)$.*

*Proof.* Because $\boldsymbol{x}[i\,..\,j]$ is Lyndon, therefore for any $i < k \le j$, $\boldsymbol{x}[i\,..\,j] \prec \boldsymbol{x}[k\,..\,j]$ and so $\boldsymbol{s}(i) \prec s(k)$. If $j = n$, we are done. So we may assume $j < n$, and we want to show that $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)$. Suppose then that $\boldsymbol{s}(j{+}1) \not\prec \boldsymbol{s}(i)$. Since $\boldsymbol{s}(i)$ and $\boldsymbol{s}(j{+}1)$ are distinct, it follows that $\boldsymbol{s}(i) \prec \boldsymbol{s}(j{+}1)$. If we let $d = lcp(\boldsymbol{s}(i), \boldsymbol{s}(j{+}1)) + 1$, two cases arise:

(a) $0 \le d \le j - i$.
Here $i \le i + d \le j$. Thus $\boldsymbol{x}[i\,..\,i{+}d{-}1] = \boldsymbol{x}[j{+}1\,..\,j{+}d]$ and $\boldsymbol{x}[i{+}d] \prec \boldsymbol{x}[j{+}1{+}d]$, and so for $j < k \le j{+}1{+}d$, $\boldsymbol{x}[i\,..\,j{+}1{+}d] \prec \boldsymbol{x}[k\,..\,j{+}1{+}d]$. Since $\boldsymbol{x}[i\,..\,j]$ is Lyndon, $\boldsymbol{x}[i\,..\,j] \prec \boldsymbol{x}[k\,..\,j]$ and so $\boldsymbol{x}[i\,..\,j{+}1{+}d] \prec \boldsymbol{x}[k\,..\,j{+}1{+}d]$ for any $i < k \le j$. Thus $\boldsymbol{x}[i\,..\,j{+}1{+}d]$ is Lyndon, contradicting the assumption that $\boldsymbol{x}[i\,..\,j]$ is the longest Lyndon factor starting at $i$.

(b) $0 < j - i \le d$.
Let $d = r(j-i) + d_1$, where $0 \le d_1 < j - i$. Then $r \ge 1$ and $\boldsymbol{x}[i\,..\,j{+}1{+}d] = \boldsymbol{u}^r \boldsymbol{u_1} b$ where $\boldsymbol{u} = \boldsymbol{x}[i\,..\,j]$,

$$\boldsymbol{u_1} = \boldsymbol{x}[j{+}r(j{-}i){+}1\,..\,j{+}r(j{-}i){+}d_1{-}1] = \boldsymbol{x}[j{+}r(j{-}i){+}1\,..\,j{+}d{-}1]$$

is a prefix of $\boldsymbol{x}[i\,..\,j]$, and $\boldsymbol{x}[i{+}d] \prec \boldsymbol{x}[j{+}1{+}d]$, so that by Lemma 13 (b), $\boldsymbol{x}[i\,..\,j{+}1{+}d]$ is Lyndon, contradicting the assumption that $\boldsymbol{x}[i\,..\,j]$ is the longest Lyndon factor starting at $i$.

Thus $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)$, as required. □

Lemma 15 describes the property of being a longest Lyndon factor of a string $\boldsymbol{x}$ in terms of relationships between corresponding suffixes.

**Lemma 15** *Consider a string $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ over an alphabet $\Sigma$ with an ordering $\prec$. A substring $\boldsymbol{x}[i\,..\,j]$ is a longest Lyndon factor of $\boldsymbol{x}$ with respect to $\prec$ if and only if $\boldsymbol{s_x}(i) \prec \boldsymbol{s_x}(k)$ for every $k \in i + 1..j$ and either $j = n$ or $\boldsymbol{s_x}(j{+}1) \prec \boldsymbol{s_x}(i)$.*

*Proof.* Let (A) denote $\{\boldsymbol{x}[i\,..\,j]$ *is a longest Lyndon factor of* $\boldsymbol{x}\}$ and let (B) denote $\{\boldsymbol{s}(i) \prec \boldsymbol{s}(k)$ *for any* $1 \le k \le j$ *and* $\boldsymbol{s}(j{+}1) \prec \boldsymbol{s}(i)\}$. Then (A) $\Rightarrow$ (B) follows from Lemma 14, so we need to prove that (B) $\Rightarrow$ (A).

Suppose then that (B) holds, and let $\boldsymbol{x}[i\,..\,k]$ be a longest Lyndon factor of $\boldsymbol{x}$ starting at position $i$. If $k < j$, then by Lemma 14, $\boldsymbol{s}(k{+}1) \prec \boldsymbol{s}(i)$, a contradiction since $k{+}1 \le j$. If $k > j$, then by Lemma 14, $\boldsymbol{s}(i) \prec \boldsymbol{s}(j{+}1)$ because $j{+}1 \le k$, which again gives us a contradiction. Thus $k = j$ and $\boldsymbol{x}[i\,..\,j]$ is a longest Lyndon factor of $\boldsymbol{x}$. □

Now we reformulate Lemma 15 in terms of the inverse suffix array ISA of $\boldsymbol{x}$ using the relationship that $\boldsymbol{s}(i) \prec s(j) \iff \mathrm{ISA}[i] < \mathrm{ISA}[j]$, thus yielding Theorem 12, as required. Hence the Lyndon array can be computed in a simple three-step algorithm, as shown in Figure 2, that executes in $\theta(n)$ time and uses only one additional array of integers.

## Appendix 3

Here we describe a simple data structure that yields an alternative approach to Algorithm NSV*, based on the comparison of longest Lyndon factors as described in Lemma 15. The ***dictionary of basic factors*** [7,8] of string $\boldsymbol{x}[1..n]$ consists of a

sequence of arrays $\mathcal{D}_t, 0 \leq t \leq \log n$. The array $\mathcal{D}_t$ records information about factors of $\boldsymbol{x}$ of length $2^t$ — that is, the basic factors. In particular, $\mathcal{D}_t[i]$ stores the rank of $\boldsymbol{x}[i..i+2^t-1]$, so that

$$\boldsymbol{x}[i..i+2^t-1] \preceq \boldsymbol{x}[i..i+2^t-1] \Leftrightarrow \mathcal{D}_t[i] \leq \mathcal{D}_t[i].$$

This dictionary requires $O(n \log n)$ space and can be constructed in $O(n \log n)$ time as follows. $\mathcal{D}_0$ contains information about consecutive symbols of $\boldsymbol{x}$ and hence can be computed in $O(n \log n)$ time by sorting all the symbols appearing in $\boldsymbol{x}$ and mapping them to numbers from 1 and onward. Once $\mathcal{D}_t$ is computed, we can easily compute $\mathcal{D}_{t+1}$ by spending $O(n)$ time on a radix sort, because $u[i..i+2^{t+1}-1]$ is in fact a concatenation of the factors $u[i..i+2^t-1]$ and $u[i+2^t..i+2^{t+1}-1]$.

Once this dictionary is computed, we can compare any two factors by comparing two appropriate overlapping basic factors (i.e., factors having length power of two), which is done by checking the corresponding $\mathcal{D}$ array from the dictionary. This will require constant time and hence each suffix-suffix comparison can be done in constant time.

# Author Index

Anisimov, Anatoly V., 71
Awid, Kamil, 22

Bannai, Hideo, 135, 158
Baruch, Gilad, 63
Barylska, Kamila, 33
Bittner, Lucie, 85
Borzì, Stefano, 99

Cleophas, Loek, 22

Di Mauro, Simone, 99
Diptarama, 7

Erofeev, Evgeny, 33

Faro, Simone, 99
Franek, Frantisek, 172

Ghuman, Sukhpal Singh, 114
Guth, Ondřej, 146

I, Tomohiro, 158
Inenaga, Shunsuke, 135, 158
Inoue, Hiroe, 135
Islam, A. S. M. Sohidull, 172

Klein, Shmuel T., 1, 63

Lecroq, Thierry, 99
Limasset, Antoine, 85

Maggio, Alessandro, 99
Marchet, Camille, 85
Matsuoka, Yoshiaki, 135
Mhaskar, Neerja, 125
Mikulski, Łukasz, 33
Msiska, Mwawi, 48

Nakashima, Yuto, 135
Nishimoto, Takaaki, 158

Peterlongo, Pierre, 85
Piątkowski, Marcin, 33

Rahman, M. Sohel, 172

Shapira, Dana, 63
Shinohara, Ayumi, 7
Smyth, William F., 172
Soltys, Michael, 125

Takeda, Masayuki, 135, 158
Tarhio, Jorma, 114

Watson, Bruce W., 22

Yoshinaka, Ryo, 7

Zavadskyi, Igor O., 71
van Zijl, Lynette, 48

**Proceedings of the Prague Stringology Conference 2016**
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.

ISBN 978-80-01-05996-8

URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811

Printed by Česká technika – Nakladatelství ČVUT
Zikova 4, Praha 6, 166 36, Czech Republic

© Czech Technical University in Prague, Czech Republic, 2016