Proceedings of the Prague Stringology Conference 2010

Edited by Jan Holub and Jan Žďárek

August 2010

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

Proceedings of the Prague Stringology Conference 2010
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
Kolejní 550/2, Praha 6, 160 00, Czech Republic.
URL: http://www.stringology.org/
E-mail: psc@stringology.org Phone: +420-2-2435-9811
Printed by Česká technika – Nakladatelství ČVUT, Thákurova 550/1, Praha 6, 160 41, Czech Republic

ISBN 978-80-01-04597-8

Conference Organisation

Program Committee

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

Organizing Committee

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

External Referees

Derrick Kourie	Hideo Bannai	Loek Cleophas
Elise Prieur	Jan Janoušek	Solon Pissis
Ernest Ketcha Ngassam	Kaichun Chang	Tinus Strauss
German Tischler	Kathleen Steinhöfel	

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2010 (PSC'10) which was organized by the old team in a new environment: The members of the Prague Stringology Club participated in founding the new Faculty of Information Technology of the Czech Technical University in Prague. The organizing team resides at the Department of Theoretical Computer Science of the new faculty. The conference was held on August 30–September 1 and it focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the program committee. Fifteen were selected as regular papers and one as a poster for a presentation at the conference, based on originality and quality. This volume contains not only these selected papers but also an abstract of one invited talk devoted to the reactive automata.

The Prague Stringology Conference has a long tradition. PSC'10 is the fifteenth event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2009 preceded this conference. The proceedings of these workshops and conferences had been published by the Czech Technical University in Prague and are available on WWW pages of the Prague Stringology Club. Selected contributions were published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science. The series of stringology conferences was interrupted in 2007 when the members of the Prague Stringology Club were honoured to organize Conference on Implementation and Application of Automata 2007 (CIAA 2007).

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

I would like to thank all those who had submitted papers for PSC'10 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'10. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2010

Jan Holub

Table of Contents

Invited Talk

Contributed Talks

Poster

Reactive Links to Save Automata States

Maxime Crochemore and Dov M. Gabbay

King's College London

Abstract. The goal of the reactive automata model is to reduce the space required for the implementation of automata. A reactive automaton has extra links whose role is to change the behaviour of the whole automaton. These links do not increase their expressiveness. Typical examples of regular expressions associated with deterministic automata of exponential size according to the length of the expression show that reactive links provide an alternative representation of total linear size.
Keywords: automaton, language representation, reactivity.

1 Introduction and background

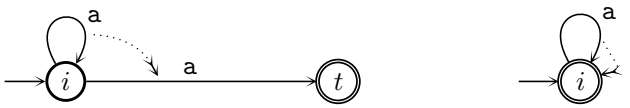
This note introduces the notion of reaction for automata and shows that using reactive links can reduce dramatically the number of states of an automaton. Within this framework some examples of state reduction are striking.

The basic use of automata is for testing if a word belongs to a regular language (membership testing). It can be done either on a regular describing the language or with an automaton, which is equivalent due to Kleene's Theorem (see for example [5]). When the automaton is deterministic the acceptance of a word of length m can be tested in linear time using $O(kn)$ space, where k is the size of the alphabet and n the number of states of the automaton. If the automaton is non deterministic the alternative is to simulate an equivalent deterministic automaton or to transform it into a deterministic automaton. The first option leads to $O(mn)$ membership time with space proportional to that of the non-deterministic automaton. The second option yields linear membership but at the cost of the determinisation which can be time and space exponential in the number of states. For related algorithms, see [1] or [6] and references therein. Both solutions are indeed implemented in the many variants of `grep` software aimed at locating regular motifs in texts. See also [7] for extra implementations and applications.

The notion of reactive automata was introduced in [3] and [4], as part of a general reactive methodology. The basic idea is that a reactive system is a system that dynamically changes during its execution as a reaction to the manner it is being utilised. A reactive system has to be distinguished from a time-dependent system as it is not dependent on an objective clock.

Reactive automaton Figure 1 displays an example of a reactive automaton. Consider the automaton (ii), which has one state i. The transition is indicated as above by the single-head arrow and reactivity by the double-head arrow. For simplicity we assume we have only one letter `a`. Assume i is the initial state as well as the terminal state. Upon receiving a letter `a` the machine stays at i, and can accept or continue. The status of i as a terminal state is then cancelled by the reactive arrow. Upon receiving a second letter `a` the machine cannot accept anymore the word `aa` as i is no longer a terminal state. Instead if the machine receives a third letter `a` state i is reactivated as a terminal state and the machine can accept the word $\mathtt{a}^3$. Obviously

Maxime Crochemore, Dov M. Gabbay: Reactive Links to Save Automata States, pp. 1–8 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8



(i) (ii)

Figure 1. Two reactive automata accepting the language $\mathtt{a}^{2n+1}$. All arcs are initially active. (i) The automaton uses a reactive edge-to-edge link to cancel or activate the arc from the initial state i to the terminal state t. (ii) The automaton uses a reactive edge-to-state link to flip the status of state i alternately as terminal and non terminal.

this reactive automaton accepts words of the form $\mathtt{a}^{2n+1}$, $n \geq 0$, only. To implement such an acceptor without reactivity we would need more states (at least two for this example).

In the next section we define the notion of a reactive automaton and show in Section 3 that the expressive power of automata is unchanged by adding reactive links. In Section 4 we state the reduction power of reactive links. Examples of reactive automata given in Section 5 have only a linear number of reactive links while they are logarithmically smaller than the minimal automata associated with their accepted language. Some remarks and open questions are stated in the conclusion.

2 Reactive automata

We formally define reactive automata starting with the notion of reactive transformation assuming the reader has some knowledge of automata definition.

Definition 1 (Switch Reactive Transformation). *Let $R \subseteq S \times \Sigma \times S$ be the transition relation of an (ordinary) automaton $\mathcal{A}$. Let $\mathbf{T}^+$, $\mathbf{T}^-$ be two subsets of $(S \times \Sigma \times S) \times (S \times \Sigma \times S)$; they are composed of pairs of the form $((p, \sigma, q), (r, \tau, s))$ where $\sigma, \tau \in \Sigma$, $p, q, r, s \in S$, and $(p, \sigma, q) \in R$.*

We define a transformation $(p, \sigma, q) \longrightarrow R^{(p,\sigma,q)}$ for $(p, \sigma, q) \in R$ using the sets $\mathbf{T}^+$ and $\mathbf{T}^-$ as follows:

$$R^{(p,\sigma,q)} = (R \setminus \{(r, \tau, s) \mid (r, \tau, s) \in R \text{ and } ((p, \sigma, q), (r, \tau, s)) \in \mathbf{T}^-\}) \cup \{(r, \tau, s) \mid (r, \tau, s) \in R \text{ and } ((p, \sigma, q), (r, \tau, s)) \in \mathbf{T}^+\}$$

Definition 2 (Switch Reactive Automaton).

1. *A reactive automaton is an ordinary non-deterministic automaton with a switch reactive transformation, i.e. a triple $\mathcal{R} = (\mathcal{A}, \mathbf{T}^+, \mathbf{T}^-)$ which defines the switch reactive transformation above.*
2. *Let $\sigma_1\sigma_2 \cdots \sigma_n$ be a word on the alphabet Σ. We define the notion of a (non-deterministic) run of $\mathcal{R}$ over $\sigma_1\sigma_2 \cdots \sigma_n$. The run is a sequence of pairs (p_k, R_k), $k = 0, \ldots, n$, defined as follows:*
   Step 0. *We start with the pair $(p_0, R_0) = (i, R)$ from the automaton $\mathcal{A} = (S, i, \Sigma, F, R)$.*
   Step $k > 0$. *Assume pairs (p_0, R_0), (p_1, R_1), $\ldots$, (p_{k-1}, R_{k-1}) have been defined as the result of a run over $\sigma_1\sigma_2 \cdots \sigma_{k-1}$. Then, state p_k is such that $(p_{k-1}, \sigma_k, p_k) \in R_{k-1}$ and $R_k = R_{k-1}^{(p_{k-1},\sigma_k,p_k)}$.*

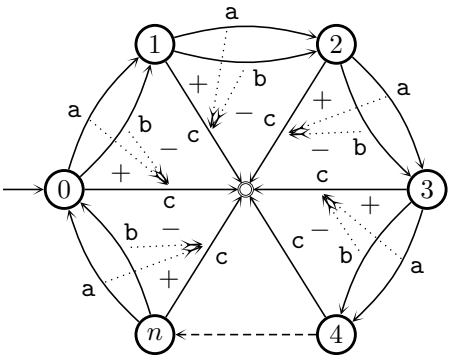


Figure 2. Reactive automaton on the alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ accepting the language $A^*\mathtt{a}A^n\mathtt{c}$. The automaton is deterministic. Any a-transition activates a c-arc from its origin state to the centre, which is the only terminal state. Any b-transition switches off such an arc. If a run over a word stops in the terminal state (the last step is a c-transition), the $(n+1)$th letter before the end of the word must be a, so the word belongs to the language, and conversely.

3. *We say that the reactive automaton $\mathcal{R}$ accepts the word $\sigma_1\sigma_2\cdots\sigma_n$ if there is a run of the automaton over this word that ends with $p_n \in F$.*

Figure 2 shows a reactive automaton that changes its arcs going to the unique terminal state. The automaton is deterministic and has linear size, $O(n)$. The automaton accepts the language $A^*\mathtt{a}A^n\mathtt{c}$ over the alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. It is known that the minimal deterministic (ordinary) automaton accepting the same language has $2^{n+1} + 1$ states (see [5]).

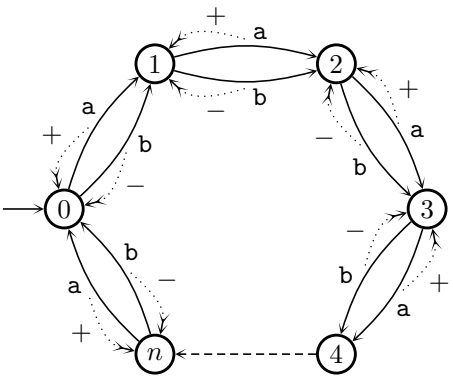


Figure 3. Reactive deterministic automaton on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ accepting the language $A^*\mathtt{a}A^n$ similar to the language of the automaton of Figure 2. Initially, the automaton has no terminal state and it uses edge-to-state reaction. Any a-transition makes its starting state a terminal state. Any b-transition transforms its starting state into a non-terminal state. If the run over a word stops in a terminal state, since the state has been set as a terminal state by an a-transition on the nth letter before, the word belongs to the language, and conversely.

The definitions extend to accommodate edge-to-state reaction. For example, Figure 3 shows another reactive automaton which changes its terminal states only and

accepts a language similar to that of Figure 2. The definition of a reactive automaton changing its terminal states is a simple adaptation of the above definition. Changing terminal states can be simulated in the model of switch reactive automata as follows without altering the possible determinism: all potential terminal states are made non terminal and linked by an ε-arc to a unique terminal node; reactive links to states are redirected to the new ε-arcs.

3 Reactivity and non-reactivity

We focus on switch reactive automata and show that their expressive power is identical to the one of ordinary automata. The proof can be adapted to automata with the various types of reactive links described in previous sections.

Theorem 3. *Any switch reactive deterministic or non-deterministic automaton is equivalent to an (ordinary) deterministic or non-deterministic automaton, respectively.*

Proof. Let $\mathcal{R} = (\mathcal{A}, \mathbf{T}^+, \mathbf{T}^-)$. We define the automaton $\mathcal{B} = (\hat{S}, (i, R), \Sigma, F, \hat{R})$ whose set of states $\hat{S}$ is composed of pairs of the form (x, R') where $x \in S$ and R' is related to R via the switch reactive transformation using $\mathbf{T}^+$ and $\mathbf{T}^-$. The transition relation $\hat{R}$ of $\mathcal{B}$ is defined using $\mathbf{T}^+$ and $\mathbf{T}^-$ as follows: $((x_1, R_1), \sigma, (x_2, R_2)) \in \hat{R}$ iff $(x_1, \sigma, x_2) \in R_1$ and $R_2 = R_1^{(x_1,\sigma,x_2)}$. Then $\mathcal{B} = (\hat{S}, i, \Sigma, F, \hat{R})$.

It is straightforward to see that a run of $\mathcal{R}$ on $\sigma_1\sigma_2\cdots\sigma_n$ corresponds to a run of $\mathcal{B}$ on the same word and vice versa. □

Remark We saw that a switch reactive automaton actually starts as an ordinary automaton $\mathcal{A} = (S, i, \Sigma, F, R)$ and then changes into different automata while receiving the input. The proof extends to Reactive Automata with state-to-state or edge-to-state reactive links. Therefore, changes can either affect the transition relation or the terminal states as shown on the previous examples.

4 Saving states of automata using reactive links

Reactive links can be used to reduce the number of states of (ordinary) automata. We state the fundamental results for deterministic and non-deterministic automata.

Theorem 4. *Any automaton $\mathcal{A}$ with kn states admits an equivalent reactive automaton $\mathcal{R}(\mathcal{A})$ with $k + n$ states. If $\mathcal{A}$ is deterministic, so is $\mathcal{R}(\mathcal{A})$.*

The construction in the proof of the above theorem cannot be iterated to reduce further the number of states.

Using higher levels of reactive links (e.g. reactive links from arc to reactive links) the next statement holds. As above the construction in the proof cannot be iterated.

Theorem 5. *If $\mathcal{A}$ is deterministic automaton with k^n states, it has an equivalent reactive automaton $\mathcal{R}(\mathcal{A})$ with $k \cdot n$ states.*

5 Examples in size reduction

The use of reactive links in automata can dramatically reduce the size of a deterministic automaton accepting a given regular language. The two previous sections show that this is possible for deterministic as well as non-deterministic automata but with a large number of reactive arcs. Instead, in the next examples, reactivity keeps the total size of automata as small as the size of their non-deterministic equivalent but without loosing determinism. In these examples determinism without reactivity leads to an exponential blow up of the number of states and then of the total size of automata.

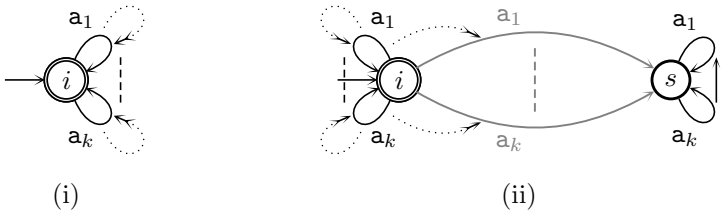


Figure 4. Two deterministic reactive automata accepting the set of strings in which each letter of the alphabet $\{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k\}$ appears at most once. All loop arcs are initially active. Loops on state i are made inactive after their first use. (i) Incomplete version. (ii) Complete version: only arcs from i to s are initially inactive and become active after the first use of their corresponding loop on state i.

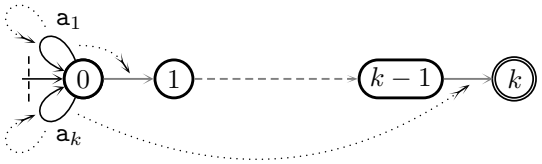


Figure 5. A deterministic reactive automaton accepting the set of strings that are permutations of the letters $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k$. All loops on the initial state are initially active and other ε-arcs are inactive. One reactive link for letter $\mathtt{a}_i$ cancels its respective loop while the second activates its associated ε-arc.

The first example corresponds to the finite language of words in which each letter of the alphabet appears at most once (see Figure 4). Its principle is that loops on the initial state are cancelled by a reactive link immediately after being used. States of a deterministic automaton for the language have to store the set of letters already treated and therefore the minimal automaton has at least an exponential number of states. Instead the total size of the reactive automaton accepting the language is $O(k)$ on a k-letter alphabet.

The automaton of Figure 5 accepts all the $k!$ permutations of letters. To do that we add a path of length k from the initial state to the unique terminal state. Compared with the automaton of the previous example, the aim is to count the number of letters treated on the initial state. Each loop on the initial state has an additional reactive link that activates its associated ε-arc on the path. So, the terminal state can

be reached only if all ε-arc are activated. We view the automaton as deterministic because its light non-determinism due to ε-arcs can be remove by considering a special symbol marking the end of words. The size of the reactive automaton is $O(k)$, which contrasts with the $O(2^k)$ size of the minimal automaton accepting the language.

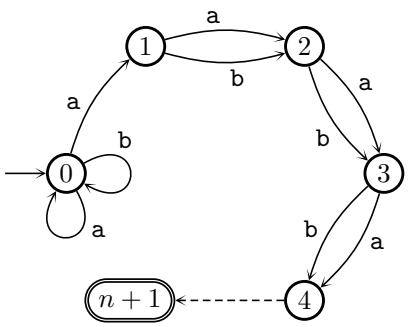


Figure 6. Non-deterministic automaton on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ accepting the language $A^*\mathtt{a}A^n$. Its equivalent reactive automata of Figures 2 and 3 have sizes of the same order.

The language $A^*\mathtt{a}A^n$ is accepted by the non-deterministic automaton of Figure 6 that has $n+2$ states and $O(n)$ total size. The non-determinism appears on the initial state only. It is known that the minimal deterministic automaton accepting the same language has 2^{n+1} states (see [5]) and then also $O(2^n)$ total size, while the equivalent reactive automaton of Figure 2 has only also $n+2$ states. It is noticeable that it has $O(n)$ total size despite the addition of reactive links.

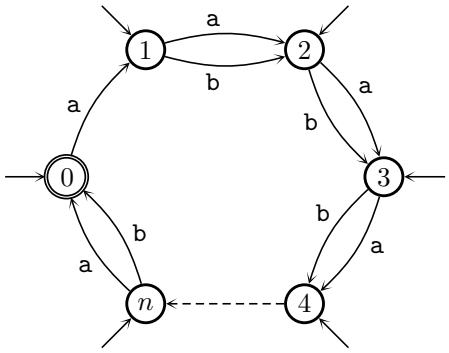


Figure 7. Non-deterministic automaton on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ accepting the language $A^{\leq n}(\mathtt{a}A^n)^*$. All states are initial states.

The last remarkable example concerns the language $A^{\leq n}(\mathtt{a}A^n)^*$ on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Figure 7 displays a non-deterministic automaton accepting it. It is non-deterministic because all its $n+1$ states are initial states. Figure 8 shows an equivalent reactive automaton, which, as above, may be considered as deterministic since ε-arcs are useful only at the end of the input word. Reactive links are from $\mathtt{b}$-transitions and cancel their associated ε-arc to the terminal state. The number of states is $n+2$ and the total size is $O(n)$. This is to be compared with the result of Béal et al. [2], which shows that the minimal deterministic automaton for this language has an exponential number of states.

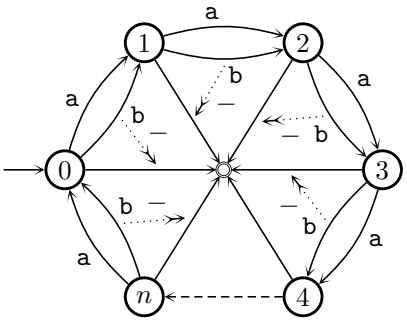


Figure 8. Reactive deterministic automaton on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ accepting the same language $A^{\leq n}(\mathtt{a}A^n)^*$ as the automaton of Figure 7. It has only one terminal state in the center. During a run, at least one ε-arc remains if some positions of letter a in the input word form a non extendible arithmetic progression of period n. The automaton has only one more state and twice as many arcs as the automaton of Figure 7.

6 Conclusion

The strength of reactive links in automata comes essentially from the reduction of the size of their implementation. Designing a non-deterministic automaton to solve a Pattern Matching question leads to slow algorithms requiring extra work to be implemented. Instead, the use of reactive links can turn a non-deterministic automaton into a deterministic Pattern Matching machine. This has two simultaneous advantages: little effort at implementation and efficient running time since the basic operation required when parsing a stream of data with an automaton is table lookup to change state, which avoids any other more time demanding low or high level instruction.

Open question In some sense, reactivity competes with non-determinism to get small automata accepting a given language, although they are not antagonist concepts. Because despite results of Section 4 showing that reactivity reduces significantly the number of states of automata, the solution requires in general a large number of extra links. But the significant examples of Section 5 raise our hope that this number can indeed be fairly small.

This note leaves open the question of whether, given a language described by a regular expression of size r, there is a reactive deterministic automaton of size $O(r)$ accepting it.

References

1. A. V. Aho: *Algorithms for finding patterns in strings*, in Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), MIT Press, 1990, pp. 255–300.
2. M.-P. Béal, M. Crochemore, F. Mignosi, A. Restivo, and M. Sciortino: *Computing forbidden words of regular languages.* Fundamenta Informaticae, 56(1,2) 2003, pp. 121–135.
3. D. M. Gabbay: *Reactive Kripke semantics and arc accessibility*, in Combinatorial Logic, W. Carnielli, F. M. Dionesio, and P. Mateus, eds., Centre of Logic and Computation, University of Lisbon, 2004, pp. 7–20.
4. D. M. Gabbay: *Reactive Kripke semantics and arc accessibility*, in Pillars of Computer Science: Essays dedicated to Boris (Boaz) Trakhtenbrot on the occasion of his 85th birthday, A. Avron, N. Dershowitz, and A. Rabinovitch, eds., vol. 4800 of LNCS, Springer-Verlag, Berlin, 2008, pp. 292–341.

5. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, third edition ed., 2006.
6. M. Lothaire, ed., *Applied Combinatorics on Words*, Cambridge University Press, 2005.
7. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings—Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.

Improving Automata Efficiency by Stretching and Jamming

Noud de Beijer[1], Loek Cleophas[1,2], Derrick G. Kourie[1], and Bruce W. Watson[1⋆]

[1] FASTAR Research Group, Department of Computer Science, University of Pretoria, 0002 Pretoria, Republic of South Africa, http://www.fastar.org

[2] Software Engineering & Technology Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, http://www.win.tue.nl/set

nouddebeijer@gmail.com, loek@loekcleophas.com, dkourie@cs.up.ac.za, bruce@fastar.org

Abstract. In recent years, the range of alphabet sizes typically used in applications of finite automata has grown considerably, now ranging from DNA alphabets—whose symbols are representable using 2 bits—to Unicode alphabets—whose symbol representation may take up to 32 bits. As automata traditionally use symbol encodings taking 8 bits, the different alphabet and symbol sizes bring up the question whether they may be exploited to either decrease memory use for the automata's transition tables or to decrease string processing time. In [3], *stretching* and *jamming* were introduced as transformations on finite automata. Given a finite automaton, we can stretch it by splitting each single transition into two or more sequential transitions, thereby introducing additional intermediate states. Jamming is the inverse transformation, in which two or more successive transitions are joined into a single transition, thereby removing redundant intermediate states. In this paper, we only consider a restricted form of stretching and jamming, in which a fixed factor is used to stretch (jam) transitions (transition paths) in a given automaton, and in which transition symbols are assumed to be encoded as bit strings. We consider improved versions of the algorithms that were presented in [3] for this particular form of stretching and jamming. The algorithms were implemented in C++ and used to benchmark the transformations. The results of this benchmarking indicate that, under certain conditions, stretching may be beneficial to memory use to the detriment of processing time, while jamming may be beneficial to processing time to the detriment of memory use. The latter seems potentially useful in the case of DNA processing, while the former may be for Unicode processing.

Keywords: finite automata; transformation; split transition; join transition; transition table size; string processing time

1 Introduction

In recent years, the range of alphabet sizes typically used in applications of finite automata has grown considerably. The alphabet typically used to be restricted to (some subset of) the 256 symbol ASCII set or a similarly sized alphabet. Nowadays the prevalence of data processing in bioinformatics and the frequent use of internationalisation in text processing have lead to the frequent use of much smaller alphabets for DNA, RNA and proteins on the one hand and of much larger alphabets for various encodings of Unicode on the other hand. Such alphabets require quite different numbers of bits to encode, ranging from 2 bits for DNA to up to 32 bits for certain encodings of Unicode. As automata representations typically used symbols of 8

⋆ This author is now also with the Department of Information Science, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa, http://www.informatics.sun.ac.za

Noud de Beijer, Loek Cleophas, Derrick G. Kourie, Bruce W. Watson: Improving Automata Efficiency by Stretching and Jamming, pp. 9–24 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

bits, this begs the question whether it makes sense to consider multiple automata transitions on symbols from e.g. a 2 bit alphabet as a single transition for a new, larger 8 bit alphabet, or a single transition on a symbol from e.g. a 32 bit alphabet as multiple transitions from a smaller 8 bit alphabet. The aim of such transformations would be to either decrease memory use for the transition tables or to decrease string processing time.

In [3], stretching and jamming were introduced as transformations on finite automata. Given a finite automaton, we can stretch it by splitting each single transition into two or more sequential transitions, thereby introducing additional intermediate states while decreasing the size of the alphabet. Jamming is the inverse transformation, in which two or more successive transitions are joined into a single transition, thereby removing redundant intermediate states yet increasing the size of the alphabet. Here, we mainly consider a restricted form of stretching and jamming, in which a fixed factor is used to stretch (jam) every transition in a given automaton, and in which transition symbols are assumed to be encoded as bit strings. We consider slightly improved versions of the algorithms presented for this type of stretching and jamming in [3]. The improved version of the stretching algorithm does not introduce (additional) nondeterminism during stretching. For jamming, we consider situations in which the original algorithm is not applicable and discuss a number of solutions for this situation. The algorithms were implemented in C++ and used to benchmark the transformations. Our benchmarking results indicate that, under certain conditions, stretching may be beneficial for memory use to the detriment of processing time, while jamming may be beneficial for processing time to the detriment of memory use.

As far as we are aware, with the exception of the earlier paper by de Beijer et al [3], no work on this topic has been published so far, although some work on the upper bound on the number of states [5] has been done.

2 Preliminaries

In this section we present the basic notions used in this paper. Most of the notation used is standard (see for example [4]) but some new notation is introduced.

A deterministic finite automaton or *DFA*, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is the *alphabet*, $\delta : Q \times \Sigma \nrightarrow Q$ is the (partial) *transition function*, q_0 is the *initial state* and F is a subset of Q whose elements are *final states*. $|Q|$ is the number of states and $|\Sigma|$ is the number of elements in the alphabet, or *alphabet size*.

The *n-closure* of an alphabet is the set of all symbols that consist of concatenating n symbols from Σ. Σ^+ is the *plus-closure* of the alphabet, the set of symbols obtained by concatenating one or more symbols from Σ. We use the special alphabet $\mathbb{B} = \{0, 1\}$, the single bit alphabet. The n-closure of this alphabet allows us to define an alphabet of sequences of n bits: $\mathbb{B}^n$.

$|Q||\Sigma|$ is the theoretical *transition table size*. Note that since cells represent states, the minimum cell size is determined by the minimum space requirements to represent a state, which is in turn determined by the total number of states. Although stretching and jamming will change the number of states in an automaton we will assume that the transition table cell size does not change in either transformation. We expect that in most cases the practical effects of this assumption are unlikely to be significant. Preliminary benchmarking results indicate that our theoretical transition table size is indeed a good estimate for the real transition table size.

A transition in a DFA M from p to q with label a will be denoted by (p, a, q) where $(p, a, q) \in Q \times \Sigma \times Q$ and $q = \delta(p, a)$. We will also use the notation $((p, a), q) \in \delta$.

A *path* of length k in a DFA M is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} = \delta(r_i, a_i)$ for $0 \leq i < k$. The *string*, or *word* $a_0 a_1 \cdots a_{k-1} \in \Sigma^k$ is the *label* of the path.

The *extended transition function* of a DFA M, $\hat{\delta} : Q \times \Sigma^+ \nrightarrow Q$, is defined so that $\hat{\delta}(r_i, w) = r_j$ iff there is a path from r_i to r_j, labeled w.

A nondeterministic finite automaton, *NFA*, is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, defined in the same way as a DFA, with the following exception: $\delta : Q \times \Sigma \nrightarrow \mathcal{P}(Q)$ is the transition function. Note that $\mathcal{P}(Q)$ is the powerset of Q. For present purposes, ϵ-transitions can be ignored without loss of generality.

A transition in an NFA M from p to q with label a will also be denoted by (p, a, q) where $(p, a, q) \in Q \times \Sigma \times Q$ and $q \in \delta(p, a)$.

A path of length k in an NFA M is a sequence $\langle (r_0, a_0, r_1), \ldots, (r_{k-1}, a_{k-1}, r_k) \rangle$, where $(r_i, a_i, r_{i+1}) \in Q \times \Sigma \times Q$ and $r_{i+1} \in \delta(r_i, a_i)$ for $0 \leq i < k$.

In an NFA M, the extended transition function, $\hat{\delta} : Q \times \Sigma^+ \nrightarrow \mathcal{P}(Q)$, is also defined so that $r_j \in \hat{\delta}(r_i, w)$ iff there is a path from r_i to r_j, labeled w.

To stretch a transition we need to split up one symbol into 2 or more sub-symbols. Therefore, we conceive of alphabet elements as strings of *subelements* (typically bits). If alphabet element $a \in \Sigma$ has length $|a|$ then we number the subelements $a.0, \ldots, a.(|a| - 1)$. Thus, if $a = 0111$ then $a.0 = 0$, $a.1 = 1$, $a.2 = 1$ and $a.3 = 1$.

We use square brackets to denote a *substring* of an alphabet element. For $a \in \Sigma$, f a *factor* or *divisor* of $|a|$ (not to be confused with a factor or substring of a word) and $0 \leq i < f$, we use $a[f, i]$ to denote $a.(i\frac{|a|}{f}) \cdots a.((i + 1)\frac{|a|}{f} - 1))$. The length of the substring depends on the factor f used in stretching; every substring of symbol a has exactly length $|a|/f$. If there is no confusion, we sometimes leave out the factor f and use just $a[i]$ instead of $a[f, i]$. So for example, if FA_0 is stretched by a factor 2 and $a = 0111$ (so $|a| = 4$), then $a[0] = 01$ and $a[1] = 11$. If FA_0 is stretched by a factor 4 then $a[0] = 0$, $a[1] = 1$, $a[2] = 1$ and $a[3] = 1$.

By definition, a *word* is a string of symbols over an alphabet. We also number the individual symbols of a word. For the word $w \in \Sigma^k$, we number the individual symbols $w.0 \cdots w.(k - 1)$. Note that we use the same notation for the subelements of a single symbol as for those of a word. Which of these is meant will always be clear from the context.

3 Stretching and jamming

Stretching and jamming were first defined in [3] and [2]. There, successively more restrictive definitions of stretching and jamming were provided, starting from general definitions, via stretching or jamming by a fixed factor per automaton, to stretching and jamming at the bit level (i.e. relying on the fact that characters are typically encoded as or represented by bit strings). Here, we present the transformations succinctly yet informally, focusing on the last, most restricted and practical kind of stretching and jamming.

One way in which we can stretch an automaton is by splitting each transition into k sequential transitions. This stretching operation on a single transition is pictured in Figure 1.

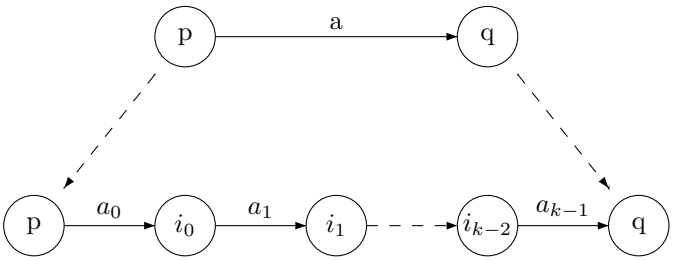


Figure 1. Stretching transition (p, a, q) into k sequential transitions.

In this example we see that transition (p, a, q) is stretched into k sequential transitions and $k - 1$ new states are introduced. In this sequence of transitions we call p and q the *original states*, and $i_0, \ldots, i_{k-2}$ the *additional intermediate states*.

Jamming is the inverse transformation, in which k sequential transitions are joined into a single transition. In Figure 1 this can be seen as performing a transformation in the opposite direction to stretching. This means that the intermediate states are removed. In the case of jamming we call these states *redundant intermediate states*.

If NFA FA_0 can be stretched into NFA FA_1, we call FA_1 *a stretch* of FA_0. The set of states of FA_1 consists of a subset S_1 of original states and a subset I of newly introduced additional intermediate states. Stretching requires

- An injection τ from the alphabet of FA_0, Σ_0, to Σ_1^+, the plus-closure of the alphabet of FA_1.
- A bijection φ between the original states of FA_0 and the original states of FA_1. This bijection connects the start states of FA_0 and FA_1 and defines a one-to-one relationship between the final states of both automata. It also ensures that for every transition from state p to q with label a in FA_0 there exists a path from $\varphi(p)$ to $\varphi(q)$ with label $\tau(a)$ in FA_1, which travels from $\varphi(p)$ in S_1 via a number of intermediate states to $\varphi(q)$ in S_1. The inverse of this property is also true.

We define jamming as the inverse transformation of stretching.

If NFA FA_0 is jammed into NFA FA_1 (FA_1 is *a jam* of FA_0) then FA_0 is a stretch of FA_1. The set of states of FA_0 consists of a subset S_0 of original states and a subset R of redundant intermediate states. These redundant intermediate states are removed by the jamming transformation.

The preceding requirements are very general, in that they allow every single transition to be stretched into a different number of sequential transitions—i.e. k can have a different value for every single transition of the original automaton. We therefore restrict stretching (and hence jamming) to stretching (jamming) by a fixed factor f:

If NFA FA_0 is stretched by a factor f into NFA FA_1, we call FA_1 *an f-stretch* of FA_0. This means that the relation τ specialises to a one-to-one relationship between the alphabet of FA_0 and the f-closure of the alphabet of FA_1. Furthermore, for each transition in FA_0 there are exactly f sequential transitions in FA_1.

Jamming by a factor f is defined analogously to stretching by a factor f. Note that, by definition, jamming by a factor f is not always possible. NFA FA_0 can only be jammed if there exists an NFA FA_1 which can be stretched into FA_0. In such a case, we call FA_0 an *f-jam* of FA_1. In Section 3.2 we consider this problem as well as some possible solutions to it.

As indicated in the introduction, alphabet symbols are typically represented as or encoded by a sequence of bits. We therefore only consider automata in which each element of the alphabet is a bit string. An *n-bit automaton* is an automaton whose alphabet consists of all the 2^n bit strings of length n.

Property 1. Let f be a factor of n. Then we can f-stretch the n-bit DFA FA_0 into NFA FA_1 in the following way:

- FA_1 is an $\frac{n}{f}$-bit NFA.
- There is a bijection between Σ_0 and Σ_1^f i.e. for every bit string of length n in Σ_0 there is a sequence of f bit strings of length $\frac{n}{f}$ in Σ_1^f and vice versa.
- For every transition in FA_0 there are f sequential transitions in FA_1, obeying the above bijection between Σ_0 and Σ_1^f for the labels of the transitions.

Of course, this specialisation of stretching is only allowed if n is divisible by f. In that case we call the DFA *f-stretchable*. Again, jamming is the inverse transformation: if an n-bit NFA is *f-jammable*, the resulting automaton is an *nf-bit* DFA.

Example 2. To illustrate the stretching of n-bit automata we give an example. The 2-bit DFA FA_0 of Figure 2 can be stretched by a factor 2 into the 1-bit NFA FA_1 of Figure 4. (We leave out the explicit definition of τ and ψ, both of which are implicitly defined by the figures.) Also, the 1-bit NFA FA_1 can be jammed into the 2-bit DFA FA_0.

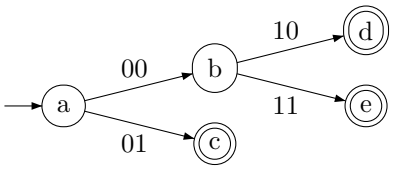


Figure 2. DFA FA_0

	00	01	10	11
a	b	c	-	-
b	-	-	d	e
c	-	-	-	-
d	-	-	-	-
e	-	-	-	-

Figure 3. Transition table of DFA FA_0

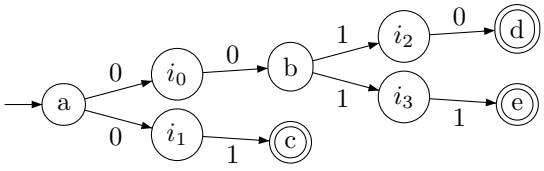


Figure 4. NFA FA_1, a 2-stretch of DFA FA_0

	0	1
a	$\{i_0,i_1\}$	-
i_0	{b}	-
i_1	-	{c}
b	-	$\{i_2,i_3\}$
i_2	{d}	-
i_3	-	{e}
c	-	-
d	-	-
e	-	-

Figure 5. Transition table of NFA FA_1

Note that in the previous example, we stretched a transition by using the most significant bit (MSB) first. For example, we stretched transition $(a, 01, c)$ into $(a, 0, i_1)$ and $(i_1, 1, c)$, taking the 0 first and then the 1. In practice, we will often use the least significant bit (LSB) first. This is due to the little-endianness of processor architectures such as the Intel x86 family. As this choice only has a minor influence on the stretching and jamming algorithms, it will not be considered further in this paper.

3.1 Stretching and nondeterminism

We can stretch NFAs as well as DFAs. In general, NFAs can be stretched and the result of such transformations will also be NFAs. Because DFAs are a subset of NFAs, the stretching of DFAs is automatically defined.

If we stretch a DFA, in some cases the resulting automaton may have more than one transition with the same label from a given state and therefore the result of stretching a DFA might be an NFA. This is apparent in the example, in which our choice of $\tau(00) = 0 \cdot 0$[1] and $\tau(01) = 0 \cdot 1$ causes two outgoing transitions with label 0 from state a to appear in FA_1 resulting from stretching FA_0. Therefore FA_1 is an NFA. Because of symmetry, if we jam certain NFAs the result will be a DFA.

We can easily prevent the introduction of such (additional) nondeterminism by checking for already created transitions during the stretching process, and following such transitions whenever possible. The stretching algorithm in Section 4 uses this approach.

3.2 Jamming and path lengths

Because of the symmetry in our definitions of stretching and jamming, jamming is only defined on the subset of NFAs that are stretched NFAs. This means that some NFAs are not jammable. For example, the NFA in Figure 6 clearly is not a stretched NFA and can therefore not be jammed according to the earlier definition.

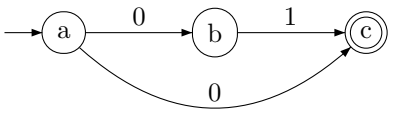


Figure 6. NFA $\mathrm{FA}_0 = (\{a, b, c\}, \mathbb{B}, \delta_0, a, \{c\})$

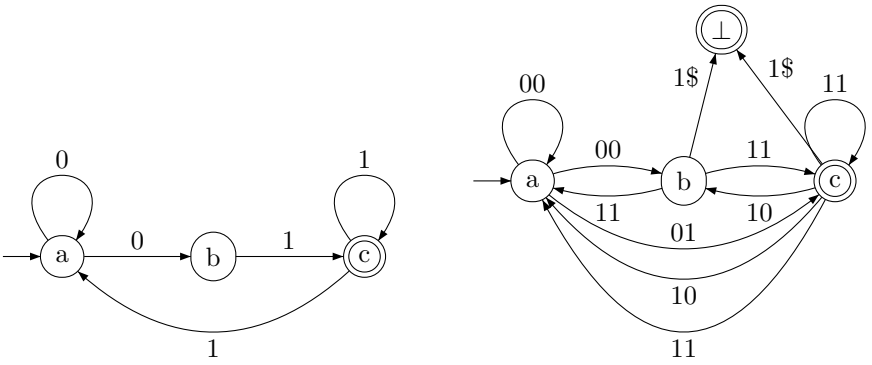
Let us now consider how to jam *any* NFA by a factor f. If we want to jam NFA FA_0 by a factor f into NFA FA_1, we essentially want to make f transitions in one step, instead of one transition at a time. We can achieve this if we find all paths of length f from the start state q_0 to states $t_0, \ldots, t_i$. Then we can add all transitions (q_0, w_j, t_j) to FA_1, where w_j is the label of the path from q_0 to t_j, $(0 \leq j \leq i)$. Next, we repeat the same process for states $t_0, \ldots, t_i$, and so on until no more new states are found.

Unfortunately, a transition added to the jammed NFA, FA_1, cannot always be guaranteed to represent a path whose length is f. For example, if there is an (extendable or non-extendable) path of length m $(m < f)$, in FA_0 from state s to a final state t with label w, the path with label w to this final state *must* be added in order to accept the word with label w in the jammed NFA. In the example in Figure 6, this occurs if we want to jam the automaton by say $f = 2$, as there is a path of length 1 from state a to final state c. We can solve this problem by creating a special final state $\perp$ in FA_1 with no outgoing transitions. Then we can add transition $(s, w\$^{f-m}, \perp)$ to FA_1, where $\$^{f-m}$ is used as *padding* to make the label $w\$^{f-m}$ exactly size f.

To illustrate this new approach to jamming an NFA, we give an example.

[1] Note that argument 00 represents a *single* symbol from the original automaton's alphabet.

Example 3. NFA FA_0 of Figure 7(a) can be jammed by a factor 2 into NFA FA_1 of Figure 7. We do this by finding all paths of length 2, as described before. For example, if we look at all such paths from state a in FA_0 we find a path with label 00 to state a itself, one with label 00 to state b, and one with label 01 to state c. If we look at the paths of length 2 from state b in FA_0 we find that on the path with label 11 ending in state c, c also occurs as a final state *inside the path*. Therefore, we have to add a transition to FA_1 from state b to state $\perp$ with label 1\$.



(a) NFA $FA_0 = (\{a, b, c\}, \mathbb{B}, \delta_0, a, \{c\})$ (b) NFA $FA_1 = (\{a, b, c, \perp\}, \mathbb{B}^2 \cup \{0\$, 1\$\}, \delta_1, a, \{c, \perp\})$

Figure 7. NFA FA_0 and NFA FA_1, a 2-jam of FA_0

The main disadvantage of the above solution is that substrings of input strings are not always recognised. For example, if the bit string 0111 is used as input for FA_0 of the previous example, both 0111 and its proper substrings 01 and 011 are accepted. However, if the same bit string is used as input for FA_1 only the bit string itself and the single proper substring 01 are accepted. The substring 011 is not accepted in this case because 2-bit value 01 leads to state c and subsequently 11 leads back to state a again—since we jammed by a factor of 2, no transition on the single original symbol 1 from state c exists and the match for 011 is not detected. This would make jamming unsuitable for cases where substrings must always be recognised e.g. for pattern matching. Various solutions to this secondary problem exist:

- One option is to not only process each jammed symbol $s = s_0 \dots s_{f-1}$, but also the $f - 1$ *padded prefix symbols* $s_0 \dots s_{f-1-m}\$^m$ for all m $(1 \leq m < f)$—i.e. the $f - 1$ padded prefixes of s's encoding. Clearly, this would completely negate any savings in string processing time one would hope to achieve in the first place by using jamming.
- A second option would be to mark states that have outgoing transitions on not only a jammed symbol s, but also on one or more of its $f-1$ padded prefix symbols. Upon reaching such a marked state, before continuing with regular processing of the next padded symbol from the input, say s, the state's outgoing transitions on padded prefix symbols should be processed to see whether they lead to accepting states.
- A third option is a variant of the second one, in which, instead of marking the state having the outgoing transition on jammed symbol s, the state to which this transition leads is marked. This state could then simply be annotated with exactly those of the padded prefix symbols derived from s that should be accepted.

Although our current implementation and benchmarking results do not take the above problems into account, it seems relatively straightforward to implement the solution with the third option indicated above. We plan to do so as part of more extensive benchmarking.

4 Algorithms

4.1 Stretch Algorithm

In this section we present an algorithm to stretch an n-bit DFA by a factor f. This algorithm can easily be generalised to an algorithm that can stretch NFAs.

As indicated before, to stretch an automaton, we stretch each single transition into f sequential transitions. Therefore, our algorithm must find each transition in the automaton. This is done using a variant of the well known *Breadth-First Search* (BFS) [1, Section 22.2]. Algorithm 5 starts in the start state and finds all outgoing transitions. All found transitions are stretched and added to the stretched DFA. This process is repeated for all states and transitions that are found by the algorithm. In the algorithm, set I represents the intermediate states added, queue Q is used to enqueue the 'grey' states as per the BFS algorithm, and set V represents the 'non-white' states as per that algorithm.

As indicated before, naive stretching of an automaton may introduce (additional) nondeterminism. In our algorithm, we prevent this from happening. If we have to add a transition with label a from a state p but such a transition exists already, then we do not add it but instead we take the existing transition. We continue with this until we have to add a transition that does not exist already. The innermost do-loop in Algorithm 5 takes care of this process.

Property 4. Let $\mathrm{FA}_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an n-bit DFA. Algorithm 5 will Stretch FA_0 by a factor f into $\frac{n}{f} - bit$ DFA $\mathrm{FA}_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

Algorithm 5 (**STRETCH**($\mathrm{FA}_0, \mathrm{FA}_1, f$))

Pre : $n \in \mathbb{Z}^+ \wedge f \in \mathbb{Z}^+ \backslash \{1\} \wedge n$ **mod** $f = 0$
Post : FA_1 is an f-stretch of FA_0

$$
\begin{array}{l}
[\![\ Q, V := \emptyset, \emptyset \\
\ ; S_1 := S_0 \\
\ ; q_1 := q_0 \\
\ ; F_1 := F_0 \\
\ ; \Sigma_1 := \mathbb{B}^{\frac{n}{f}} \\
\ ; \delta_1 := \emptyset \\
\ ; enqueue(q_0, Q) \\
\ ; \textbf{do}\ Q \neq \emptyset \rightarrow \\
\qquad p := dequeue(Q) \\
\qquad ; \textbf{for all}\ q, a : q = \delta_0(p, a) \rightarrow \\
\qquad\quad s_0^{pq} := p \\
\qquad\quad ; i := 0 \\
\qquad\quad ; \textbf{do}\ \delta_1(s_i^{pq}, a[i]) \neq \emptyset \rightarrow s_{i+1}^{pq} := \delta_1(s_i^{pq}, a[i]) \\
\qquad\qquad\qquad\qquad\qquad\qquad ; i := i + 1
\end{array}
$$

```
                od
               ;let s_{i+1}^{pq}, ..., s_{f-1}^{pq} be new states
               ;I := I ∪ {s_{i+1}^{pq}, ..., s_{f-1}^{pq}}
               ;s_f^{pq} := q
               ;for j := i to f − 1 → δ1 := δ1 ∪ {((s_j^{pq}, a[j]), s_{j+1}^{pq})} rof
               ;as q ∉ V → V := V ∪ {q}; enqueue(q, Q) sa
            rof
  od
]|
```

Proof. We need to prove that algorithm 5 will correctly stretch FA_0.

- We have to prove there exists a bijection $\tau : \Sigma_0 \leftrightarrow \Sigma_1^f$. Because $\Sigma_1^f = (\mathbb{B}^{\frac{n}{f}})^f = \mathbb{B}^n = \Sigma_0$, there is a trivial bijection between the two sets. Intuitively, we can see that every bit string of length n can be constructed from a unique sequence of f bit strings of length $\frac{n}{f}$.
- After the algorithm is executed, $S_1 = S_0$, $q_1 = q_0$ and $F_1 = F_0$. Therefore there is an obvious bijection $\varphi : S_0 \leftrightarrow S_1$. Because the algorithm is based on the BFS algorithm, all states p that are reachable from the start state are found. For each such state p, all transitions (p, a, q) are found, for all symbols a and states q. According to our definition of stretching, if we find transition (p, a, q) we have to add one or more transitions such that $\hat{\delta}_1(p, w) = q$, with $\tau(a) = w$. We do this by ensuring that, after stretching the transition, a path from state p to state q with label $a[0]a[1]\cdots a[f-1]$ exists, where $a[0]a[1]\cdots a[f-1] = w$. In this path, state p and q are in set S_1 and the rest of the states are in the set of intermediate states I. DFA FA_1 might already have a path from state p with label $a[0], a[1]\cdots a[i]$, $(0 \leq i < f-1)$. If this is the case, this path is followed and only the remaining transitions are added to FA_1. □

During a stretch operation a number of intermediate states is inserted for every transition in the original automaton. Note that the number of states inserted by Algorithm 5 is not necessarily minimal, as exemplified in Figure 8.

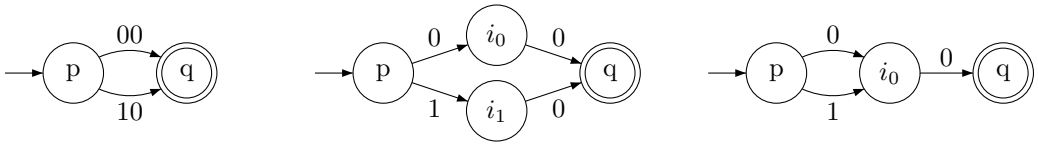


Figure 8. Minimisation after a stretching operation

This problem is a standard minimisation problem for DFAs. Therefore, after stretching a DFA, the resulting DFA can be minimised using one of the many DFA minimisation algorithms. It is also possible to minimise incrementally, after all outgoing transitions from a certain state have been stretched.

4.2 Jam Algorithm

If we want to jam NFA FA_0 by a factor f into NFA FA_1, we essentially want to make f transitions in one step, instead of one transition at a time. We can achieve this if we find all paths of length f from the start state q_0 to states $t_0, \ldots, t_i$. Then we can add all transitions (q_0, w_j, t_j) to FA_1, where w_j is the label of the path from q_0 to t_j, $(0 \leq j \leq i)$. Next, we repeat the same process for states $t_0, \ldots, t_i$, and so on until no more new states are found. To find all paths of length f from a give state p we can use a variant of the *Depth-First Search* (DFS) algorithm [1].

In the original DFS algorithm a set V is used for states that have already been found. If the search encounters a state that has already been found it does not explore the transitions out of that state. Because we need to find *all paths*, we also need to explore transitions out of states that have already been found. Therefore, instead of using a set V of states found, we use a variable c to indicate the current depth. If the search reaches depth f we stop searching.

Our algorithm uses a recursive procedure to search all paths of length f. The abstract algorithm can be described concisely and can be implemented easily. It is possible to do the search with a non-recursive procedure, using a stack to store all states found and the paths to these states.

Algorithm 6 (**JAM(**FA_0**,** FA_1**,** f**)**)

Pre : $n \in \mathbb{Z}^+ \wedge f \in \mathbb{Z}^+ \backslash \{1\}$
Post : FA_1 is an f-jam of FA_0
$[\![$ $Q := \emptyset$
$; S_1 := \{\perp\}$
$; q_1 := q_0$
$; F_1 := F_0 \cup \{\perp\}$
$; \Sigma_1 := \{x\$^{nf-ni} \colon x \in \mathbb{B}^{ni}, 1 \leq i \leq f\}$
$; \delta_1 := \emptyset$
$; enqueue(q_0, Q)$
$; S_1 := S_1 \cup \{q_0\}$
; **do** $Q \neq \emptyset \rightarrow q := dequeue(Q)$
$;$JAM-PATH$(\mathrm{FA}_0, \mathrm{FA}_1, Q, q_0, q_0, f, 0, \epsilon)$
od
$]\!]$

Algorithm 7 (**JAM-PATH(**FA_0**,** FA_1**,** Q**,** r**,** p**,** d**,** c**,** l**)**)

Pre : $r, p \in S_0 \wedge d \in \mathbb{Z}^+ \wedge c \in \mathbb{N} \wedge c \leq d \wedge l \in \Sigma_0^c$
Post : All paths in FA_0 of length d, and starting in state r,
are jammed and added to FA_1
$[\![$ **if** $c < d \rightarrow$ **as** $p \in F_0 \rightarrow \delta_1 := \delta_1 \cup \{((r, l\$^{d-c}), \perp)\}$ **sa**
$;$**for all** $q, a : q \in \delta_0(p, a) \rightarrow$
JAM-PATH$(\mathrm{FA}_0, \mathrm{FA}_1, Q, r, q, d, c+1, la)$
rof

$$
\begin{array}{l}
[\!] \quad c = d \rightarrow \textbf{as}\ \ p \notin S_1 \rightarrow S_1 := S_1 \cup \{p\};\ \ \mathit{enqueue}(p, Q)\ \ \textbf{sa} \\
\qquad\qquad ; \delta_1 := \delta_1 \cup \{((r, l), p)\} \\
\textbf{fi} \\
[\!]
\end{array}
$$

Property 8. Let $\mathrm{FA}_0 = (S_0, \Sigma_0, \delta_0, q_0, F_0)$ be an n-bit NFA. Algorithms 6 and 7 will jam FA_0 by a factor f into nf-bit NFA $\mathrm{FA}_1 = (S_1, \Sigma_1, \delta_1, q_1, F_1)$.

Proof. The algorithm is based on the DFS algorithm, which is used to find all paths of length f from the start state q_0. All end states of the path, or in other words, all states that are at distance f from q_0, are added to the queue and for these states the same process is repeated recursively.

The label of each path is recorded in variable l, and every time a new transition (p, a, q) is found, the algorithm recursively calls itself with the new label la and new state q. This way, the algorithm recursively descends into the NFA.

Variable c indicates the current depth of the path, so if depth $d = f$ is reached, a path of length f from state r to state p with label l has been found. This means a new transition (r, l, p) can be added to the jammed NFA. If a final state is found on a path that does not have length $d = f$, a transition $(r, l, \perp)$ is added to FA_1, where r is the first state of the path, l the label, and $\perp$ is the special final state introduced.

Therefore, if a string is processed by the jammed NFA, it can travel any path to a final state as it would have done in the original NFA. This means the jammed NFA will accept the same set of strings as the original NFA.

We conclude this proof by proving the correctness of the construction of the alphabet. The original NFA FA_0 has an n-bit alphabet. The algorithm finds all paths of length f, but on each path a final state might be found. Therefore, paths of length $1, 2, \ldots, f$ can be found. Because each symbol in the original NFA is n bits long, a symbol in the new alphabet can have length n, $2n$,$\ldots$, fn bits. The alphabet symbols of the jammed NFA must have bit-length nf, therefore padding with \$ symbols is added to the alphabet symbols if they do not have length nf. □

5 Implementation

The (improved) algorithms for stretching and jamming as presented in the preceding section were implemented in C++. This language was chosen for its flexibility and efficiency, in particular in combination with the Standard Template Library (STL).

Due to the similarities between DFAs and NFAs—both conceptual and hence in implementation as well—the implementation uses a single abstract base class FA, which in turn uses a class TransitionTable. Both are parameterised by the type of the cells of the transition table—i.e. State for DFAs and set<State> for NFAs. Parameterised class TransitionTable inherits from class Matrix, which in turn is further parameterised by the type of rows and columns to be used. Clearly, for a matrix used as a transition table, these correspond to states and alphabet symbols. In turn, such a matrix is naturally composed of vectors, hence the use of a parameterised type Vec.

Class FA, its derived classes and class TransitionTable have a method NextState which uses the current state (which TransitionTable keeps track of) and the next input symbol to advance to the next state(s). They also have methods to add transitions, resize the transition table, determine whether an automaton was created by stretching or jamming, and to return the size of the transition table or its density.

Classes DFA and NFA have a method to stretch by a factor of f, passed as a parameter together with a boolean indicating whether LSB and MSB should be reversed compared to the default (which on the Intel x86 architecture is LSB before MSB). Both methods return a new DFA/NFA object containing the stretched version of the original DFA/NFA. Class DFA also has a method to jam by a factor of f, with similar parameters. For NFAs, jamming has not yet been implemented.

Due to a technicality, jammed DFAs use additional data structures to represent transitions on any of the newly added padded symbols. Jammed DFAs are therefore represented by instances of class JammedDFA instead of those of class DFA. As jamming for NFAs is currently not implemented, no class JammedNFA is needed.

6 Benchmarking Results

We have performed preliminary benchmarking experiments using the implementations from the preceding section. Due to lack of space, we only report some of the results here. More details on the experiments, the environment and the results can be found in [2, Chapter 7]. The experiments were focused on stretching and jamming of DFAs. The experiments were used to compare DFAs' transition table memory use and their string processing time before and after stretching and jamming. To do so, random DFAs were generated using a (pseudo-)random number generator, and the computed memory use of their transition tables before and after stretching or jamming were compared. Random paths from these DFAs were generated to measure the DFAs' string processing time, both before and after stretching or jamming. The experiments were performed on an x86 family system running a version of Linux in single user mode, on which the implementations had been compiled using GCC 3.3.2. String processing time was measured using the CPU's time stamp counter. Memory use was computed based on the state set size, alphabet size, and resulting transition table cell size, as the STL vectors used for the transition table implementations do not guarantee the vector sizes not to be larger than needed to store just its current elements. (Note that memory use is still dependent on the number of intermediate states removed by jamming or inserted by stretching and therefore varies depending on the original DFA's structure.)

The benchmarking was performed with a range of parameters, and for each choice of parameter values used, 100 runs of the benchmark were performed and the mean and variance of memory use and processing time were collected. For both stretching and jamming, factors of 2, 4, and 8 were used, and alphabet sizes 2^1, 2^2, 2^4 and 2^8 were used. The generated DFAs had 10, 100 or 1000 states, with stretching (jamming) obviously increasing (decreasing) these numbers. Transition densities considered ranged from 1% - 100%. As input strings, strings of 8, 16, and 32 were considered. As we will see, not all combinations of these parameter values were used (some do not even make sense for all experiments).

Our theoretical analyses of the effects of the two transformations were reported in [3, Section 4] and [2, Chapter 3]. They show that stretching is expected to reduce memory use for low transition table densities: as few intermediate states will be introduced at such densities, this will be more than offset by the decrease in alphabet size, and hence the theoretical transition table size $|Q||\Sigma|$ will decrease. Clearly, jamming is primarily meant to reduce string processing time and expected to do so, while stretching is expected to increase it. In the remainder of this section we will see

how the benchmarking confirms these expectations and briefly discuss under which conditions stretching and jamming seem practically useful.

6.1 Stretching

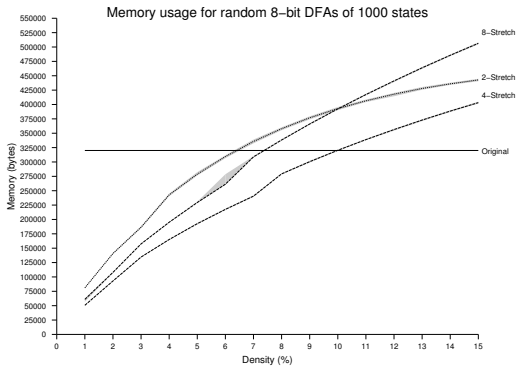


Figure 9. Memory usage results for stretching 8-bit DFAs

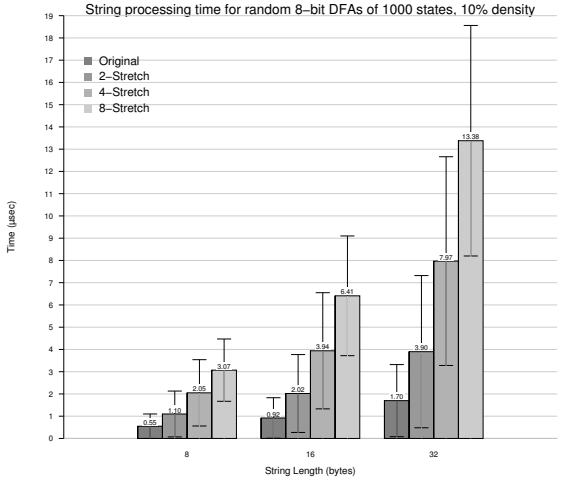


Figure 10. String processing time results for stretching 8-bit DFAs

Figures 9 and 10 show some of the results of the benchmarks for stretching. Figure 10 shows the influence of stretching on string processing time for a typical case, i.e. for 8-bit DFAs with 1000 states. Results for fewer states and higher densities are similar, and as expected density does not influence string processing time.

Figure 9 shows the impact on memory use, again for 8-bit DFAs with 1000 states. For 10 or 100 states, the memory use graphs are roughly the same, albeit with break even points in the $4-6\%$ and $5-9\%$ range, respectively. (Note the seemingly counter-intuitive high memory use of 8-stretches compared to 4-stretches. We suspect this is caused by memory allocation issues related to alignment and packing of objects, as well as the memory consumption reporting mechanisms used.) For 4-bit DFAs, the break even points are somewhat higher, ranging from $12-19\%$ depending on the number of states and stretch factor used. We therefore expect that benchmarking DFAs using larger symbols such as typically needed for applications of Unicode will

show that if such automata are of low density, memory use can be reduced manifold by stretching, even if just stretching by a factor of 2. Since applications of Unicode in automata tend to lead to large memory use because of the large number of symbols, this might be a worthwhile avenue to explore further.

6.2 Jamming

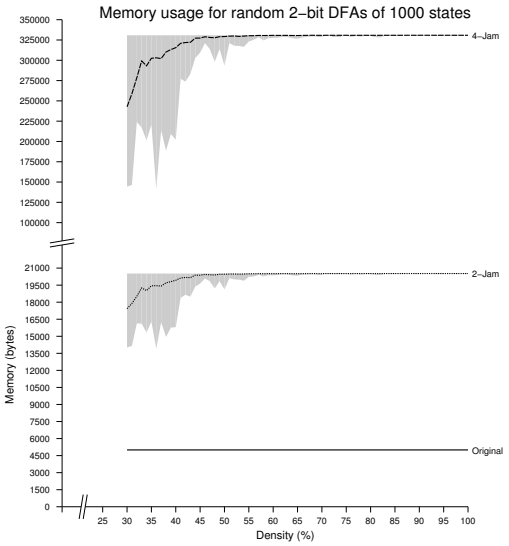


Figure 11. Memory usage results for jamming 2-bit DFAs

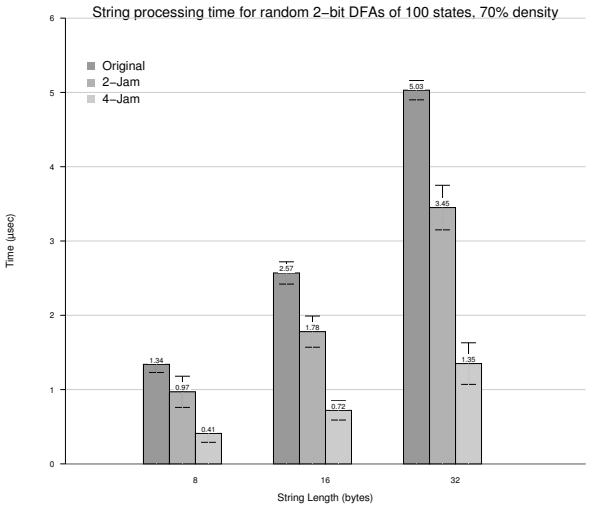


Figure 12. String processing time results for jamming 2-bit DFAs

Figures 11 and 12 show some of the results for jamming. Figure 11 shows the impact of jamming on memory use, for the case of 2-bit DFAs of 1000 states. Results for fewer states and smaller alphabets are again roughly similar. What stands out immediately is the horizontal upper bound in the range of memory usage for all three

factors of jamming. This is easily explained though, as even though jamming has the potential to reduce the number of states by removing redundant intermediate states, in the worst case no such states exist, while the alphabet size is always increased by jamming. The *average* memory use approaches the upper bound as density increases, as the likelihood of states being redundant decreases with an increase in transition density. (Note that the fact that transition density does not start at 0% but starts above 25% is due to our requirement that the DFAs generated be connected.)

Figure 12 clearly shows how jamming improves string processing time by $40-45\%$ when 2-jamming and reduces it by a further $2.3-2.6$ times when comparing this to 4-jamming—i.e. by $3.2-3.7$ times in total when going from the original 2-bit symbols to 8-bit symbols. As the DNA alphabet corresponds to a 2-bit alphabet, the results show the potential of using jamming for such an alphabet, provided memory use is not an issue at all or transition density is not too high.

7 Conclusions and Future Work

We have presented stretching and jamming and given improved versions of the algorithms from [3], which prevent nondeterminism from being introduced during stretching. Furthermore, we have sketched solutions to prevent jammed automata from no longer detecting all matches the underlying original automata would detect. The preliminary benchmarking studies reported here and in [2], for the case of DFAs, confirm our theoretical analysis from [3]: while jamming increases memory use, particularly for DFAs with a high transition table density, it reduces string processing time drastically. Conversely, stretching increases string processing time considerably, but reduces memory use for DFAs with a low transition table density.

Our benchmarking results can be extended in multiple directions. Although the benchmarking we performed already covers DNA and protein alphabets (which need between 2 and 5 bits to represent a symbol), it does not cover Unicode alphabets (needing up to 32 bits to represent a symbol). Extending the experiments to such alphabets could therefore be considered. In particular, with an eye on when and how to apply the transformations in practice, experiments with DFAs from practical applications in DNA and Unicode text processing should be performed. In particular, it should be investigated whether the increased memory use when jamming DNA automata is acceptable when dealing with the amount of data resulting from current high throughput DNA sequencing technologies. Furthermore, the experiments could be extended to cover NFAs in addition to DFAs, using the solutions we sketched for the match detection problem.

There are a number of additional problems that can be investigated further. We only considered stretching or jamming the complete transition table. Transforming only a small part of the transition table, in other words local stretching and jamming, is an interesting problem for further research. So is the use of stretching and jamming when sparse matrices are used to represent low density transition tables.

References

1. T. Cormen, C. Leiserson, and R. Rivest: *Introduction to algorithms*, McGraw-Hill, 2001.
2. A. de Beijer: *Stretching and jamming of automata*, Master's thesis, Eindhoven University of Technology, 2004.

3. N. De Beijer, B. W. Watson, and D. G. Kourie: *Stretching and jamming of automata*, in SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, South African Institute for Computer Scientists and Information Technologists, 2003, pp. 198–207.
4. J. Hopcroft, R. Motwani, and J. Ullman: *Introduction to automata theory, languages, and computation*, Addison-Wesley, 2001.
5. B. Melichar and J. Skryja: *On the size of deterministic finite automata.*, in Proc. of the Sixth International Conference on Implementation and Application of Automata (CIAA 2001), B. Watson and D. Wood, eds., vol. 2494 of LNCS, Pretoria, South Africa, July 2001, pp. 202–213.

Simple Tree Pattern Matching for Trees in the Prefix Bar Notation

Jan Lahoda[1] and Jan Žďárek[2,⋆]

[1] Sun Microsystems Czech,
V Parku 2308/8, 148 00 Praha 4, Czech Republic,
Jan.Lahoda@sun.com
[2] Department of Theoretical Computer Science,
Faculty of Information Technology,
Czech Technical University in Prague,
Kolejní 550/2, 160 00 Prague 6, Czech Republic
Jan.Zdarek@fit.cvut.cz

Abstract. A new pushdown automata based algorithm for searching all occurrences of a tree pattern in a subject tree is presented. The algorithm allows pattern matching with don't care symbols and multiple patterns. A simulation algorithm is also proposed, and practical experimental results are presented.

1 Introduction

Tree pattern matching has numerous applications in computing, for example in program optimization, code generation and refactoring. It has been researched thorougly for several decades, see Janoušek and Melichar [9]. Recently, a new stream of research has been started by Janoušek and Melichar [9]. They consider trees in the postfix notation as strings and present a transformation from any given bottom-up finite tree automaton recognizing a regular tree language to a deterministic pushdown automaton accepting the same tree language in postfix notation. Based on this fundamental result, Melichar *et al.* started to extend principles of text pattern matching using finite automata into the tree pattern matching domain. They use pushdown automata for matching in trees, where trees are represented by their prefix or postfix notation [8,4,5]. These automata are either constructed directly as deterministic pushdown automata, or they are nondeterministic input-driven pushdown automata. The nondeterminism can be removed in the latter case, as it is known that any input-driven pushdown automaton can be determinised [13]. The prefix bar notation is the prefix notation of a rooted ordered labeled directed tree where only closing bracket of a bracket pair is used. The prefix bar notation was introduced by Stoklasa, Janoušek and Melichar in [10,11]. A detailed overview of the tree matching algorithms based on pushdown automata is due to Janoušek [7].

In this paper we propose a new algorithm for tree pattern matching. The algorithm allows to perform tree pattern matching with don't cares, including multiple tree patterns, by means of pushdown automata. The pushdown automata constructed by our algorithm are visibly pushdown [1] and so can be determinised. As the determinised versions of the visibly pushdown automata can be quite big (see Section 3), a simulation algorithm is also proposed for the constructed automata. The simulation algorithm was evaluated experimentally and the results are presented in the final part of the paper.

⋆ Partially supported by GAČR project No. 201/09/0807 and MŠMT project No. MSM 6840770014.

Jan Lahoda, Jan Ždárek: Simple Tree Pattern Matching for Trees in the Prefix Bar Notation, pp. 25–24 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

The motivation to invent the algorithm described herein was a tool that allows to quickly search vast amounts of source code and find given patterns in the code. The tool is given one or more AST snippets ("patterns"), including don't cares, and then it processes a huge number of ASTs and searches for occurrences of the pattern(s) in these ASTs. To fulfill this task, the tool preprocesses the pattern once, and then analyses the ASTs, processing them on the fly.

1.1 Definitions

Let A be a finite alphabet and its elements be called symbols. A set of strings over A is denoted by A^*. A language L is any subset of A^*, $L \subseteq A^*$. The empty string is denoted by ε. The "don't care" symbol is a special universal symbol that matches any other symbol including itself [3].

A finite automaton (FA) is a quintuple (Q, A, δ, I, F). Q is a finite set of states, A is a finite input alphabet, $F \subseteq Q$ is a set of final states. If an FA is nondeterministic (NFA), then δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$ and $I \subseteq Q$ is a set of initial states. A deterministic FA (DFA) is (Q, A, δ, q_0, F), where δ is a (partial) function $Q \times A \mapsto Q$; $q_0 \in Q$ is the only initial state.

The following definitions introduce pushdown automata and related notions. A (nondeterministic) pushdown automaton (PDA), is a septuple $(Q, A, G, \delta, q_0, Z_0, F)$, where Q is a finite set of states, A is a finite input alphabet, G is a finite pushdown store alphabet, δ is a mapping $Q \times (A \cup \{\varepsilon\}) \times G \mapsto \mathcal{P}(Q \times G^*)$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown store symbol, $F \subseteq Q$ is a set of final states. A pushdown store operation of PDA M, $M = (Q, A, G, \delta, q_0, Z_0, F)$, is a relation $(A \cup \{\varepsilon\}) \times G \mapsto G^*$. A pushdown store operation produces new contents on the top of the pushdown store by taking one input symbol or the empty string from the input and the current contents on the top of the pushdown store. The pushdown store grows to the right if written as a string x, $x \in G^*$. A transition of PDA M is the relation $\vdash_M \subseteq (Q \times A^* \times G) \times (Q \times A^* \times G^*)$. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively.

A PDA is a deterministic PDA if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in A \cup \{\varepsilon\}$, $\gamma \in G$.
2. For all $\alpha \in G$, $q \in Q$, if $\delta(q, \varepsilon, \alpha) \neq \emptyset$, then $\delta(q, a, \alpha) = \emptyset$ for all $a \in A$.

A language L accepted by PDA M is a set of words over finite alphabet A. It is defined in two distinct ways:

1. *Accepting by final state*:
$$L(M) = \{x : \ \delta(q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma), x \in A^*, \gamma \in G^*, q \in F\}.$$
2. *Accepting by empty pushdown store*:
$$L_\varepsilon(M) = \{x : \ (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon), x \in A^*, q \in Q\}.$$

If the PDA accepts the language by empty pushdown store then the set F of its final states is defined to be the empty set.

Alur and Madhusudan [1] introduced a special type of pushdown automata and languages they accept.

A *visible alphabet* $\tilde{A}$ is a triple $\tilde{A} = (A_c, A_r, A_{int})$. $\tilde{A}$ comprises three categories of symbols, three disjoint finite alphabets: A_c is a finite set of calls (pushdown store

grows), A_r is a finite set of returns (pushdown store shrinks), A_{int} is a finite set of internal actions that do not use the pushdown store.

A *visibly pushdown automaton* (VPA) is a septuple $(Q, \tilde{A}, G, \delta, Q_{in}, Z_0, F)$, where Q is a finite set of states, $\tilde{A}$ is a finite visible alphabet, G is a finite pushdown store alphabet, δ is a mapping: $(Q \times A_c \times \varepsilon) \mapsto (Q \times (G \setminus \{Z_0\})) \cup$ $(Q \times A_r \times G) \mapsto (Q \times \varepsilon) \cup$ $(Q \times A_{int} \times \varepsilon) \mapsto Q \times \varepsilon)$, $Q_{in} \subseteq Q$ is a set of initial states, $Z_0 \in G$ is the initial pushdown store symbol, $F \subseteq Q$ is a set of final states.

All notions related to extended pushdown automata hold for visibly pushdown automata as well. Specifically, the language accepted by visibly pushdown automaton: A *language $L(M)$ accepted by visibly pushdown automaton M* is the set of words accepted by M. A *visibly pushdown language* is a set of words over some finite alphabet A, $L \subseteq A^*$ with respect to $\tilde{A}$ ($\tilde{A}$-VPL) if there exists a visibly pushdown automaton M over $\tilde{A}$ such that $L(M) = L$. Visibly pushdown automata can be determinised.

Let V be a set of nodes, E be a set of edges. A rooted ordered labeled directed tree T, $T = (V, E)$, is a rooted directed tree where every node $v \in V$ is labeled by symbol $a \in A$ and its out-degree is given by the arity of the symbols of A. Nodes labeled by nullary symbols (constants) are called leaves. All trees used in this paper are rooted ordered labeled directed trees.

Let us define the prefix bar notation [11, analogous to Def. 2].

Definition 1. *The prefix bar notation of tree P, with root r and its children $c_1, \ldots, c_n$, denoted by $d(P)$ is defined recursively as follows: $d(P) = rd(c_1) \cdots d(c_n) \uparrow$.*

Note also that r and $\uparrow$ in Definition 1 are symbols of alphabets A_c and A_r, respectively, of a particular visibly pushdown automaton we simulate.

2 Main Idea

In this section, we will describe new algorithms for exact tree pattern matching and for tree pattern matching with don't cares. The algorithms use Euler-like notation to serialize the trees (Žďárek [12, Alg. 3.8]) and finite automata or pushdown automata to perform the matching. The algorithms described herein in fact extend the algorithm described by Flouri, Janoušek and Melichar in [5] where they consider deterministic PDA constructions for subtree pattern matching.

2.1 Exact Pattern Matching

In this section, we will present a simple algorithm for exact tree pattern matching based on finite automata. As noted by Stoklasa, Janoušek and Melichar [11, Theorem 1], the prefix bar notation of a tree contains prefix bar notations of all subtrees of the tree as substrings. The exact pattern matching can therefore be performed easily as follows. First, a finite automaton for exact string pattern matching is constructed for $d(P)$. A string matching algorithm is then used to locate the occurrences of $d(P)$ in $d(T)$, which correspond to occurrences of P in T.

Theorem 2. *Given tree pattern P, containing m total nodes, and subject tree T, containing n total nodes, the aforementioned algorithm for tree pattern matching runs in $\mathcal{O}(n + m)$ time.*

Proof. The deterministic finite automaton constructed for the prefix bar notation of P will have $2m + 1$ states, and can be constructed in $\mathcal{O}(m)$ time (Crochemore [2], Holub [6]). Pattern matching over the prefix bar notation of T using this automaton then takes $\mathcal{O}(n)$ time.

2.2 Pattern Matching with Don't Cares

In this section, we will show how to extend the algorithm described in the previous section to handle don't care tree nodes.

Definition 3. *A leaf node of tree pattern P marked with don't care symbol $+$ matches any single complete subtree in the subject tree T.*

Definition 4. *The prefix bar notation of tree P with don't care symbols, root r and children $c_1, \ldots, c_n$, denoted by $d(P)$ is defined recursively as follows:*

$$d(P) = \begin{cases} rd(c_1) \cdots d(c_n) \uparrow & \text{iff } r \neq + \\ + & \text{iff } r = + \end{cases}$$

The finite automata are not sufficient to model tree pattern matching with don't care symbols. Pushdown automata (PDA) will be used for this task.

Algorithm 1 shows the construction of the pushdown automaton for tree pattern matching with don't cares. The tree pattern matching is then performed over the prefix bar notation of the subject tree.

The PDA constructed by Algorithm 1 is structurally similar to finite automaton for exact tree pattern matching described in the previous section. The pushdown operations for "down" and "up" symbols are as follows. For "down" symbols, pushdown symbol e is pushed to the store, for "up" symbols the same symbol is popped from the store. The reason for pushing and popping this symbol is to ensure that during matching the pushdown store contains as many symbols as is the current depth in the subject tree. This limits the actual pushdown-store non-determinism of the automaton.

The don't cares are translated into the PDA as shown in Figure 1. In the transition from the state s to (inner) state i, the "target" state is remembered in the pushdown store. The loop transitions on the state i ensure that whole subtree will be skipped.

In Algorithm 1, lines 4– 11 construct the states for matching the don't care symbol (as shown in Figure 1). Lines 13–19 construct the basic structure of the automaton. Lines 22–25 construct the loop in the initial state.

For patterns without don't care symbols, the Algorithm 1 constructs automata that are functionally equivalent to finite automata for exact tree pattern matching of trees in prefix bar notation described in Section 2.1. The states and transitions that are created for don't care symbols assure that the automaton will skip symbols that correspond to a complete subtree in the prefix bar notation. The first such symbol pushes a marker symbol at the top of the of the pushdown store. When the closing $\uparrow$ symbol of the complete subtree is processed, this marker is found at the top of the pushdown store, and the matching continues with the following symbol of the pattern's prefix bar notation. The algorithm therefore performs tree pattern matching with don't care symbols for tree in the prefix bar notation.

The PDA constructed by Algorithm 1 is a non-deterministic one. Note, however, that the non-determinism is caused solely by the loop in the initial state. Without it, the automaton would be deterministic and would implement a tree-top search. The PDA is also a visibly pushdown automaton ([1]) and so can always be determinised.

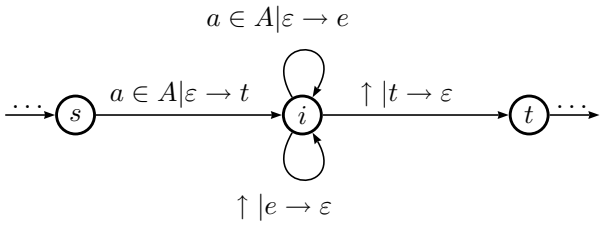


Figure 1. PDA states representing don't care symbol

Algorithm 1 Construction of PDA from tree pattern for tree pattern matching

Input: Pattern tree P in the prefix bar notation
Output: PDA $M = (Q, A, G, \delta, q_0, e, F)$

create state q_0, $q = q_0$, $Q = \{q_0\}$, $G = \{e\}$
for all $a \in d(P)$ **do**
  if a is $+$ **then**
    create new states q_1, q_2
    for all $b \in A$ **do**
      $\delta(q, b, \varepsilon) = \{(q_1, q_2)\}$
      $\delta(q_1, b, \varepsilon) = \{(q_1, e)\}$
    end for
    $\delta(q_1, \uparrow, e) = \{(q_1, \varepsilon)\}$
    $\delta(q_1, \uparrow, q_2) = \{(q_2, \varepsilon)\}$
    $q = q_2$, $G = G \cup \{q_2\}$
  else
    create new state q'
    if a is $\uparrow$ **then**
      $\delta(q, a, e) = \{(q', \varepsilon)\}$
    else
      $\delta(q, a, \varepsilon) = \{(q', e)\}$
    end if
    $q = q'$
  end if
end for
for all $b \in A$ **do**
  $\delta(q_0, b, \varepsilon) = \delta(q_0, b, \varepsilon) \cup \{(q_0, e)\}$
end for
$\delta(q_0, \uparrow, e) = \{(q_0, \varepsilon)\}$
$F = \{q\}$

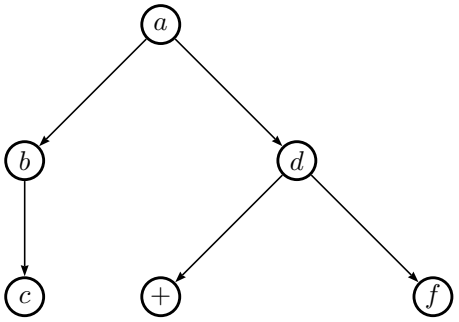


Figure 2. Example tree pattern with don't cares - its prefix bar notation is $abc \uparrow\uparrow d + f \uparrow\uparrow\uparrow$

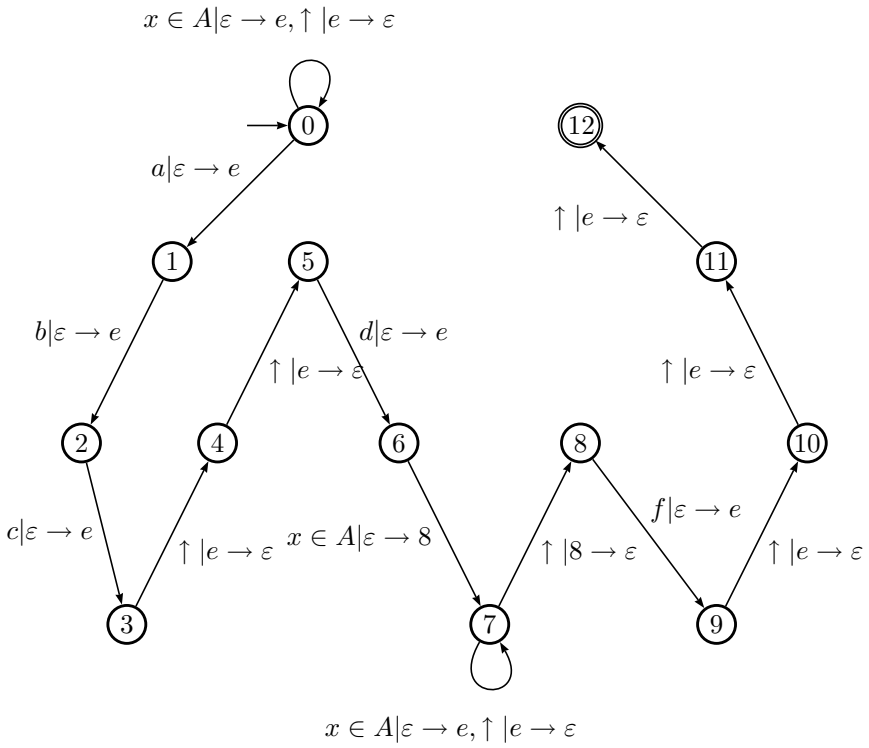


Figure 3. Pattern matching PDA for tree pattern shown in Figure 2

2.3 Multiple Pattern Matching

The algorithms described in the previous sections can be straightforwardly extended to perform the tree pattern matching with don't cares for multiple patterns. For each pattern, the PDA is constructed using Algorithm 1. A new PDA is then constructed from these sub-PDAs by uniting the initial states.

2.4 Simulation

In the previous sections, we have shown a way to construct a non-deterministic pushdown automaton for tree pattern matching with don't cares. The constructed automaton is also visibly pushdown, and so can be determinised. However, as the automaton can become quite big during the determinization process, we will show how to efficiently simulate the non-deterministic automaton. In this section, we will show that the pushdown store non-determinism is limited and base the simulation on this fact.

Lemma 5 (Absence of variable-length branches). *Let T be a tree and $d(T)$ its prefix bar notation. Let $M = (Q, A, G, \delta, q_0, e, F)$ be a PDA constructed by Algorithm 1 for a tree pattern P. Then for each prefix p of $d(T)$ exists an integer l such that for each transition sequence $(q_0, p, e) \vdash^* (q', \varepsilon, s)$, $l = |s|$.*

Proof. In the PDA created by Algorithm 1, all the transitions for $\uparrow$ pop a symbol from the pushdown store and all transitions for the other symbols push a symbol to the pushdown. Consequently, the depth of the pushdown store depends only on the number of $\uparrow$ and non-$\uparrow$ symbols in the currently processed prefix, not on the sequence of transitions.

Note 6 (No interference on the pushdown store). Let $M = (Q, A, G, \delta, q_0, e, F)$ be a PDA constructed by Algorithm 1 for a tree pattern P. Then there are no two states $q_1, q_2 \in Q$, $q_1 \neq q_2$, $a \in A$ and $s, u, w \in G$, $s \neq e$ such that $\delta(q_1, a, w) = (q_1', ws)$, $\delta(q_2, a, u) = (q_2', us)$.

The meaning of Lemma 5 is that there are no variable-length branches of the pushdown store while simulating this automaton. Note 6 points out that no two distinct transitions store the same symbol to the pushdown store. These two observations together assure that the pushdown store of PDA constructed by Algorithm 1 can be simulated using bit parallelism. The simulation algorithm based on bit-parallelism is described below.

The simulation is shown in Algorithm 2. The simulation is based on the simulation of non-deterministic finite automata using bit-parallelism. Variable W consisting of $|Q|$ bits contains the bit mask of active states (one bit of W is assigned to each state, active and inactive states have bit value 1 and 0, respectively). Each entry of pushdown store S, consisting of $|G|$ bits, contains the bit mask of the current pushdown symbols (one bit is assigned to each pushdown symbol except e, if the symbol is in the pushdown store at the given level, value 1 is used, 0 otherwise). Symbol e is always on the pushdown store and does not need to be encoded.

Theorem 7. *Algorithm 2 runs in $\mathcal{O}(nm^2)$ worst case time, where n is the number of nodes of the subject tree and m is the number of nodes of the pattern tree.*

Algorithm 2 Simulation of PDA for tree pattern matching

Input: Subject tree T in the prefix bar notation, PDA $M = (Q, A, G, \delta, q_0, e, F)$
Output: Root nodes of the occurrences of tree pattern P in subject tree T

```
 1: W = {q_0}, S = ∅
 2: for all a ∈ d(T) do
 3:    if a is ↑ then
 4:       W' = ∅
 5:       for all q ∈ W do
 6:          (q', ε) = δ(q, ↑, e)                    /*at most one such entry*/
 7:          W' = W' ∪ {q'}
 8:       end for
 9:       W' = W' ∪ pop element from S
10:       W = W'
11:    else
12:       W' = ∅, S' = ∅
13:       for all q ∈ W do
14:          for all (q', s') in δ(q, a, ε) do
15:             if s' = e then
16:                W' = W' ∪ {q'}
17:             else
18:                S' = S' ∪ {s'}
19:             end if
20:          end for
21:       end for
22:       W = W', push S' to S
23:    end if
24:    if W ∩ F ≠ ∅ then
25:       found occurrences of P in T
26:    end if
27: end for
```

Proof. The main loop starting at line 2 iterates over $2n$ elements of the prefix bar notation of the subject tree T. Both branches of the *if* statement on line 3 iterate over the currently active states, perform transitions in the PDA and merge the results using union. There may be up to $\mathcal{O}(m)$ active states and each union takes up to $\mathcal{O}(m)$ time. As $\delta(q, a, \varepsilon)$ may contain at most two elements (for $q = q_0$), the loop on line 14 is performed at most twice. Therefore, each pass through the if statement on line 3 takes $\mathcal{O}(m^2)$ time. The intersection on line 24 takes $\mathcal{O}(|F|)$ time ($\mathcal{O}(m)$ in the worst case). Therefore, the total worst case time complexity of the algorithm is $\mathcal{O}(nm^2)$.

An example of the simulation is given in Figure 5. The subject tree of the tree pattern matching is depicted in Figure 4, the tree pattern is shown in Figure 2 and the corresponding PDA for tree pattern matching is depicted in Figure 3. The figure presents set of the active states W and pushdown store S, used to identify the end of a subtree matched by a don't care symbol in the pattern. Their values are displayed before and after the reading of each input symbol.

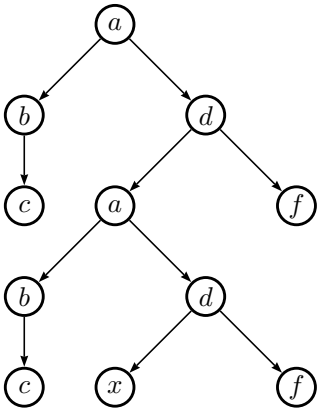


Figure 4. Example subject tree for example shown in Figure 5

text		*a*	*b*	*c*	↑	↑	*d*	*a*	*b*	*c*	↑	↑	*d*	*x*	↑	*f*	↑	↑	↑	*f*	↑	↑	↑	
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	2	3	4	5	6													8	9	10	11	**12**
								1	2	3	4	5	6		8	9	10	11	**12**				
S								{8}				{8}		{8}				{8}						
									{8}		{8}		{8}		{8}		{8}							
										{8}				{8}		{8}								

Figure 5. Example of simulation using PDA depicted in Figure 3 in the subject tree depicted in Figure 4; W is the set of active states, S is the pushdown store; $F = \{12\}$

3 Experimental Results

To evaluate the practical properties of the proposed algorithm, we have implemented the algorithm and performed several experiments.

First, let us discuss the number of states and pushdown symbols in non-deterministic and deterministic pushdown automata for tree pattern matching. We have implemented an incremental version of determinization algorithm described by Alur and Madhusudan [1]. Consider for example tree pattern depicted in Figure 6. The corresponding non-deterministic pushdown automaton has 19 states and 6 pushdown symbols. When this automaton is determinised, the resulting deterministic pushdown automaton has 20087 accessible states and 154 used pushdown symbols. Although this is significantly less than the upper bound of number of states and number of pushdown symbols (which is in this case $2^{19 \cdot 6}$ states and and $2^{6 \cdot 6}$ pushdown symbols), the absolute number of states is still significant and it would be impractical to keep such big automata.

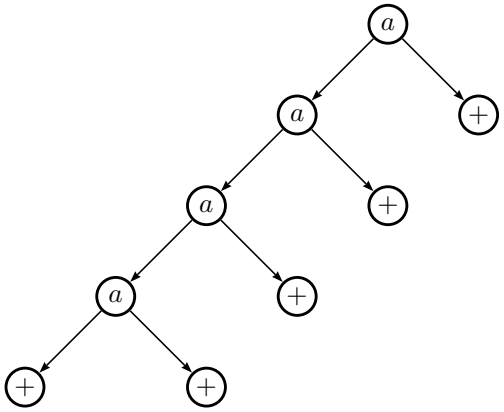


Figure 6. Example tree for which corresponding tree pattern matching PDA is determinised

Furthermore, we tested the impact of the use of bit parallelism for simulating the non-deterministic PDA for tree pattern matching, 46433 Java source files were searched for selected patterns. The tested source files were a complete NetBeans IDE main code base, available at `http://hg.netbeans.org/main-golden`, the changeset on which the experiments were performed was `96f614d8662d`. The tested patterns are shown in Table 1. Let us note that the patterns are expanded before the PDA is constructed using the algorithms denoted above. This expansion produces a set of patterns for each input pattern for the user's convenience. For example the pattern

`org.openide.util.RequestProcessor.getDefault()`

is augmented with patterns like

`RequestProcessor.getDefault()` and `getDefault()`.

The experimental results are summarized in Table 2. The results suggest that the number of states active at any given time during the matching is very low in practice. The simulation algorithm is therefore practically viable.

1. method invocation:

```
org.openide.util.RequestProcessor.getDefault()
```

2. double checked locking:

```
if ($var == null) {
     synchronized($lock) {
         if ($var == null) $statements;
     }
}
```

3. 151 standard NetBeans IDE patterns

Table 1. Tested patterns

pattern name	total states	active states		running time [s]
		average	maximum	
method invocation	20	1.54	3	163
double checked locking	449	1.52	12	121
all	3853	6.67	70	254

Table 2. Summary of experimental results. For tested patterns see Table 1

4 Conclusion

In this paper we have presented a new algorithm for tree pattern matching. The algorithm is based on pushdown automata and supports both don't care symbols and multiple patterns. An algorithm for efficient simulation of the automaton is given.

We see several possible directions for future research. One possible direction is to investigate possibility of don't cares which would match any number of complete subtrees (i.e. don't cares with a variable arity). It would also be possible to investigate the behaviour of the determinisation algorithm with regard to the tree pattern matching PDA (not only the one described in this paper), and if it is possible to adjust the PDAs in such a way that the determinisation algorithm would provide smaller results. Finally, the don't cares may not be independent: e.g. it is possible to say that two subtrees that are covered by two don't cares must in fact be equivalent. Would it be possible to extend the tree pattern matching algorithm to understand such constraint?

4.1 Acknowledgements

The authors wish to thank the anonymous referees for their detailed reviews and helpful comments.

References

1. R. Alur and P. Madhusudan: *Visibly pushdown languages*, in Proceedings of the thirty-sixth Annual ACM Symposium on Theory of Computing, New York, NY, 2004, ACM Press, pp. 202–211.
2. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific Publishing, Hong-Kong, 2002, 310 pages.
3. M. J. Fischer and M. S. Paterson: *String matching and other products*, in Complexity of Computation, R. M. Karp, ed., vol. 7, SIAM-AMS Proceedings, 1974, pp. 113–125.
4. T. Flouri, J. Janoušek, and B. Melichar: *Tree pattern matching by deterministic pushdown automata*, in Proceedings of the International Multiconference on Computer Science and Information Technology, Workshop on Advances in Programming Languages, M. Ganzha and M. Paprzycki, eds., vol. 4, IEEE Computer Society Press, 2009, pp. 659–666.
5. T. Flouri, J. Janoušek, and B. Melichar: *Subtree matching by pushdown automata*. Computer Science and Information Systems, 7(2) Apr. 2010.
6. J. Holub: *Simulation of Nondeterministic Finite Automata in Pattern Matching*, PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, 2000.
7. J. Janoušek: *Arbology: Algorithms on trees and pushdown automata*, habilitation thesis, Brno University of Technology, 2010, submitted.
8. J. Janoušek: *String suffix automata and subtree pushdown automata*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, 2009, pp. 160–172.
9. J. Janoušek and B. Melichar: *On regular tree languages and deterministic pushdown automata*. Acta Inform., 46(7) Nov. 2009, pp. 533–547.
10. J. Stoklasa, J. Janoušek, and B. Melichar: *Subtree pushdown automata for trees in bar notation*, 2010, London Stringology Days 2010, London.
11. J. Stoklasa, J. Janoušek, and B. Melichar: *Subtree pushdown automata for trees in bar notation*, 2010, submitted to J. Discret. Algorithms.
12. J. Žďárek: *Two-dimensional Pattern Matching Using Automata Approach*, PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, 2010, submitted, `http://www.stringology.org/papers/Zdarek-PhD_thesis-2010.pdf`.
13. K. Wagner and G. Wechsung: *Computational Complexity*, Springer-Verlag, Berlin, 2001.

Approximate String Matching Allowing for Inversions and Translocations

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | faro | giaquinta}@dmi.unict.it

Abstract. The approximate string matching problem consists in finding all locations at which a pattern P of length m matches a substring of a text T of length n, after a given finite number of edit operations.
In this paper we investigate such problem when the string distance involves translocations of equal length adjacent factors and inversions of factors. In particular, we devise a $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space algorithm, where α and β are respectively the maximum length of the factors involved in any translocation and inversion. Our algorithm is based on the dynamic-programming approach and makes use of the Directed Acyclic Word Graph of the pattern. Moreover we show that under the assumptions of equiprobability and independence of characters our algorithm has a $\mathcal{O}(n \log_\sigma m)$ average time complexity. Finally, we briefly sketch in an appendix an efficient implementation, based on bit-parallelism.

1 Introduction

Retrieving information and teasing out the meaning of biological sequences are central problems in modern biology. Generally, basic biological information is stored in strings of nucleic acids (DNA, RNA) or amino acids (proteins). Aligning sequences helps in revealing their shared characteristics, while matching sequences can infer useful information from them.

With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval. In recent years, much work has been devoted to the development of efficient methods for aligning strings and, despite sequence alignment seems to be a well-understood problem (especially in the edit-distance model), the same can not be said for the approximate string matching problem on biological sequences.

Approximate string matching is a fundamental problem in text processing and consists in finding approximate matches of a pattern in a string. The closeness of a match is measured in terms of the sum of the costs of the edit operations necessary to convert the string into an exact match.

Most biological string matching methods are based on the *edit distance* [7] (also called the *Levenshtein distance*) or on the *Damerau edit distance* [6]. The edit operations in the former edit distance are *insertion*, *deletion*, and *substitution* of characters, while the latter one allows *swaps* of characters, i.e., traspositions of two adjacent characters (for an in-depth survey on approximate string matching, see [8]). These distances assume that changes between strings occur locally, i.e., only a small portion of the string is involved in the mutation event. In contrast, evidence shows that large scale changes are possible. For example, large pieces of DNA can be moved from

Domenico Cantone, Simone Faro, Emanuele Giaquinta: Approximate String Matching Allowing for Inversions and Translocations, pp. 37–51 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

one location to another (*translocations*), or replaced by their reversed complements (*inversions*).

In this paper we investigate the approximate string matching problem under a string distance whose edit operations are translocations of equal length adjacent factors and inversions of factors. In particular, we present a $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space algorithm, where α and β are the maximum length of the factors involved in a translocation and in an inversion, respectively. Our algorithm is based on a dynamic-programming approach and makes use of the Directed Acyclic Word Graph of the pattern. The DAWG data structure has already been used in algorithms for the approximate string matching problem [11,10], to keep track of the substrings of the pattern that match the text at every location. We show that under the assumption of equiprobability and independence of characters in the alphabet, on the average our algorithm has a $\mathcal{O}(n \log_\sigma m)$-time complexity. Finally, we present also an efficient implementation of our algorithm, based on bit-parallelism, which has $\mathcal{O}(n \max(\alpha, \beta))$-time and $\mathcal{O}(\sigma + m)$-space complexity, when the pattern length is comparable with the size of the computer word. To our knowledge there is no report in the literature of a similar formalization of the above problem.

The rest of the paper is organized as follows. In Section 2 we introduce some preliminary notions and definitions. Subsequently, in Section 3 we present a new automaton-based algorithm for the approximate string matching problem with translocations and inversions. Section 4 is devoted to the analysis of our algorithm both in the worst and in the average-case. In Section 5 we present experimental results which allow us to evaluate the practical performance of our newly proposed algorithm and its bit-parallel variant, briefly described in Appendix A. Finally, we draw our conclusions in Section 6.

2 Basic notions and definitions

Let P be a string of length $m \geq 0$, over an alphabet Σ. We represent it as a finite array $P[0\,..\,m-1]$ of characters of Σ and write $|P| = m$. In particular, for $m = 0$ we obtain the empty string ε. We denote by $P[i]$ the $(i+1)$-st character of P, for $0 \leq i < m$. Likewise, the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P is indicated with $P[i\,..\,j]$, for $0 \leq i \leq j < m$. The set of substrings (also called *factors*) of P is denoted by $\mathit{Fact}(P)$. Given another string P', we say that P' is a suffix of P (in symbols, $P' \sqsupseteq P$) if $P' = P[i\,..\,m-1]$, for some $0 \leq i < m$, and indicate with $\mathit{Suff}(P)$ the set of the suffixes of P. Similarly, we say that P' is a prefix of P if $P' = P[0\,..\,i]$, for some $0 \leq i < m$. We also put $P_i =_{\text{Def}} P[0\,..\,i]$, for $0 \leq i < m$, and make the convention that P_{-1} denotes the empty string ε. In addition, we write PP' to denote the concatenation of P and P', and P^{r} for the reverse of the string P, i.e., $P^{\mathrm{r}} =_{\text{Def}} P[m-1]P[m-2]\ldots P[0]$.

A *distance* $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ is a function which associates to any pair of strings X and Y the minimal cost of any finite sequence of edit operations which transforms X into Y, if such a sequence exists, ∞ otherwise. Edit operations have the form $Z \to_t W$, with $Z, W \in \Sigma^*$ and t a nonnegative real number which represents the cost. If, for every operation $Z \to_t W$, there is also the symmetric operation $W \to_t Z$ (with the same cost), then the distance d is symmetric, i.e., $d(X, Y) = d(Y, X)$, for all $X, Y \in \Sigma^*$.

For $X \in \mathit{Fact}(P)$, we denote with $\mathit{end\text{-}pos}(X)$ the set of all positions in P where an occurrence of X ends; formally,

$$\mathit{end\text{-}pos}(X) =_{\text{Def}} \{i \mid |X| - 1 \leq i < m \text{ and } X \sqsupseteq P_i\}\,.$$

For any given pattern P, we define an equivalence relation $\mathcal{R}_P$ by putting

$$X \,\mathcal{R}_P\, Y \iff_{\text{Def}} \mathit{end\text{-}pos}(X) = \mathit{end\text{-}pos}(Y)\,,$$

for all $X, Y \in \Sigma^*$, and denote with $\mathcal{R}_P(X)$ the equivalence class over Σ^* of the string X.

The Directed Acyclic Word Graph [3,4,5] of a pattern P (DAWG, for short) is the deterministic automaton $\mathcal{A}(P) = (Q, \Sigma, \delta, \mathit{root}, F)$ whose language is $\mathit{Fact}(P)$, where

- $Q = \{\mathcal{R}_P(X) : X \in \mathit{Fact}(P)\}$ is the set of states,
- Σ is the alphabet of the characters in P,
- $\delta : Q \times \Sigma \to Q$ is the transition function defined, for all $c \in \Sigma$ and $Yc \in \mathit{Fact}(P)$, by $\delta(\mathcal{R}_P(Y), c) =_{\text{Def}} \mathcal{R}_P(Yc)$,
- $\mathit{root} = \mathcal{R}_P(\varepsilon)$ is the initial state,
- $F = Q$ is the set of final states.

For each equivalence class q of $\mathcal{R}_P$, let $\mathit{val}(q)$ be the longest string X in the equivalence class q and put $\mathit{length}(q) =_{\text{Def}} \mathit{length}(\mathit{val}(q))$. In addition, we define a failure function, $s\ell : \mathit{Fact}(P) \setminus \{\varepsilon\} \to \mathit{Fact}(P)$, called *suffix link*, by putting, for any $X \in \mathit{Fact}(P) \setminus \{\varepsilon\}$,

$$s\ell(X) =_{\text{Def}} \text{longest } Y \in \mathit{Suff}(X) \text{ such that } Y \not\mathcal{R}_P X\,.$$

The function $s\ell$ has the following property:

$$X \,\mathcal{R}_P\, Y \Longrightarrow s\ell(X) = s\ell(Y)\,.$$

We extend the functions $s\ell$ and *end-pos* to Q by putting, for each $q \in Q$,

$$\begin{aligned} s\ell(q) &=_{\text{Def}} \mathcal{R}_P(s\ell(\mathit{val}(q))) \\ \mathit{end\text{-}pos}(q) &=_{\text{Def}} \mathit{end\text{-}pos}(\mathit{val}(q))\,. \end{aligned}$$

Definition 1. *Given two strings X and Y, the* mutation distance $md(X, Y)$ *is based on the following edit operations:*

(1) ***Translocation****: a factor of the form ZW is transformed into WZ, provided that $|Z| = |W| > 0$.*
(2) ***Inversion****: a factor Z is tranformed into Z^{r}.*

Both operations are assigned unit cost. □

Observe that, by definition, the maximum length of the factors involved in a translocation is $\lfloor |X|/2 \rfloor$, whereas the length of the factors involved in an inversion can be up to $|X|$. Note, moreover, that there are strings X, Y such that X can not be converted into Y by any sequence of translocations and inversions, in which case $md(X, Y) = \infty$. When $md(X, Y) < \infty$, we say that X and Y have an md-match. Additionally, if X has an md-match with a suffix of Y, we write $X \sqsupseteq_{md} Y$.

3 An automaton-based approach for the pattern matching problem with translocations and inversions

We present an efficient algorithm, called M-Sampling, which finds the md-matches of a given pattern P (of length m) in a text T (of length n). Our algorithm, based on the dynamic programming approach, has a $\mathcal{O}(nm\max(\alpha,\beta))$-time and $\mathcal{O}(m^2)$-space complexity, where $\alpha \leq \lfloor m/2 \rfloor$ is a bound on the length of the factors involved in any translocation and $\beta \leq m$ is a bound on the length of the factors involved in any inversion.

Given P, T, m, n, α, and β as above, the M-Sampling algorithm iteratively computes for $j = m-1, m, \ldots, n-1$ all the prefixes of P which have an md-match with a suffix of T_j, by exploiting information gathered at previous iterations. For this purpose, a set $\mathcal{S}_j$ is maintained, defined by

$$\mathcal{S}_j =_{\text{Def}} \{0 \leq i \leq m-1 \mid P_i \sqsupseteq_{md} T_j\}\,.$$

Thus, the pattern P has an md-match ending at position j of the text T if and only if $(m-1) \in \mathcal{S}_j$.

Since the allowed edit operations involve substrings of the pattern P, it is useful to introduce the set $\mathcal{F}_j^k$ of all the positions in P where an occurrence of the suffix of T_j of length k ends. More precisely, for $1 \leq k \leq \alpha$ and $k-1 \leq j < n$, we put

$$\mathcal{F}_j^k =_{\text{Def}} \{k-1 \leq i \leq m-1 \mid T[j-k+1\,..\,j] \sqsupseteq P_i\}\,.$$

Observe that $\mathcal{F}_j^k \subseteq \mathcal{F}_j^h$, for $1 \leq h \leq k \leq m$.

Similarly, to handle inversions, it is convenient to define the set $\mathcal{I}_j^k$ of the positions in P where an occurrence of the reverse of the suffix of T_j of length k ends. More precisely, for $1 \leq k \leq \beta$ and $k-1 \leq j < n$, we put

$$\mathcal{I}_j^k =_{\text{Def}} \{k-1 \leq i \leq m-1 \mid (T[j-k+1\,..\,j])^{\mathsf{r}} \sqsupseteq P_i\}\,.$$

The sets $\mathcal{S}_j$ can then be computed based on the following elementary recursion.

Lemma 2. *Let T and P be a text of length n and a pattern of length m, respectively. Then $i \in \mathcal{S}_j$, for $0 \leq i < m$ and $i \leq j < n$, if and only if one of the following three facts holds*

(a) $P[i] = T[j]$ and $(i-1) \in \mathcal{S}_{j-1} \cup \{-1\}$ (standard match);
(b) $(i-k) \in \mathcal{F}_j^k$, $i \in \mathcal{F}_{j-k}^k$, and $(i-2k) \in \mathcal{S}_{j-2k} \cup \{-1\}$, for some $1 \leq k \leq \lfloor \frac{i+1}{2} \rfloor$ (translocation);
(c) $i \in \mathcal{I}_j^k$ and $(i-k) \in \mathcal{S}_{j-k} \cup \{-1\}$, for some $1 \leq k \leq i+1$ (inversion). □

Conditions (b) and (c) refer to a translocation of adjacent factors of length k and an inversion of a factor of length k, respectively.

Likewise, the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ can be computed according to the following lemma:

Lemma 3. *Let T and P be a text of length n and a pattern of length m, respectively. Then $i \in \mathcal{F}_j^k$, for $1 \leq k \leq \alpha$, $k-1 \leq i < m$, and $k-1 \leq j < n$, if and only if the following condition holds*

$$(k = 1 \text{ or } (i-1) \in \mathcal{F}_{j-1}^{k-1}) \text{ and } P[i] = T[j]\,.$$

Similarly, $i \in \mathcal{I}_j^k$, for $1 \leq k \leq \beta$, $k-1 \leq i < m$, and $k-1 \leq j < n$, if and only if the following condition holds

$$(k = 1 \text{ or } i \in \mathcal{I}_{j-1}^{k-1}) \text{ and } P[i-k+1] = T[j]\,.$$ □

Based on Lemmas 2 and 3, a general dynamic programming algorithm can be readily constructed, characterized by an overall $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space complexity. However, the overhead due to the computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ turns out to be quite large. By suitably preprocessing the pattern with the DAWG data structure, as will be described in the next section, the M-Sampling algorithm succeeds in reducing drastically such overhead (see Fig. 2). The code of the algorithm M-Sampling is shown in Fig. 1 (on the left).

3.1 Efficient computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$

An efficient method for computing the sets $\mathcal{F}_j^k$ described above, for $1 \leq k \leq \alpha$ and $k-1 \leq j < n$, makes use of the DAWG of the pattern P and the function *end-pos*. Later we will also show how to compute efficiently the sets $\mathcal{I}_j^k$.

Let $\mathcal{A}(P) = (Q, \Sigma, \delta, root, F)$ be the DAWG of P. For each position j in T, let P' be the longest factor of P, of length at most α, which is a suffix of T_j, let q_j be the state of $\mathcal{A}(P)$ such that $\mathcal{R}_{\mathcal{P}}(P') = q_j$, and let l_j be the length of P'. We call the pair (q_j, l_j) a T-configuration of $\mathcal{A}(P)$. The idea is then to compute the T-configuration (q_j, l_j) of $\mathcal{A}(P)$, for each position j of the text, while scanning the text. The set $\mathcal{F}_j^k$ computed at previous iterations are not maintained explicitly; rather, only T-configurations are maintained. These are then used to compute efficiently the set $\mathcal{F}_j^k$ only when needed.

The longest factor of P ending at position j of T is computed in the same way as in the Forward-Dawg-Matching algorithm for the exact pattern matching problem (cf. [5]). Since we are interested in factors of length at most α, we maintain the invariant that the current state of the automaton never corresponds to factors longer than α (we discovered that much the same idea was used in [10]).

Let (q_{j-1}, l_{j-1}) be the T-configuration of $\mathcal{A}(P)$ at step $(j-1)$. Two cases must be distinguished.

Case $l_{j-1} < \alpha$**:** The new T-configuration (q_j, l_j) is set to $(\delta(q, T[j]), length(q) + 1)$, where q is the first node in the suffix path $(q_{j-1}, s\ell(q_{j-1}), s\ell^{(2)}(q_{j-1}), \ldots)$ of q_{j-1}, including q_{j-1}, having a transition on $T[j]$, if such a node exists; otherwise (q_j, l_j) is set to $(root, 0)$.[1]

Case $l_{j-1} = \alpha$**:** We first compute the T-configuration corresponding to the factor $T[j-\alpha+1\,..\,j-1]$ of P of length $(\alpha-1)$ ending at position $j-1$ in T, namely the T-configuration (q'_{j-1}, l'_{j-1}), where

$$(q'_{j-1}, l'_{j-1}) =_{\text{Def}} \begin{cases} (s\ell(q_{j-1}), l_{j-1} - 1) & \text{if } length(s\ell(q_{j-1})) = l_{j-1} - 1 \\ (q_{j-1}, l_{j-1} - 1) & \text{otherwise.} \end{cases}$$

Then we compute the new T-configuration (q_j, l_j) starting from (q'_{j-1}, l'_{j-1}) as in the previous case, observing that $l'_{j-1} = \alpha - 1$. The algorithm to update the T-configuration of the DAWG $\mathcal{A}(P)$ is given in Fig. 1 (on the right), where $s\ell^*$ denotes the improved suffix link [5].

Before explaining how to compute the sets $\mathcal{F}_j^k$, it is convenient to introduce a partial function, $\phi : Q \times \mathbb{N} \to Q$, which given a node $q \in Q$ and a length $k \leq length(q)$ computes the node $\phi(q, k)$ whose corresponding set of factors contains the suffix of

[1] We recall that $s\ell^{(0)}(q) =_{\text{Def}} q$ and, recursively, $s\ell^{(h+1)}(q) =_{\text{Def}} s\ell(s\ell^{(h)}(q))$, for $h \geq 0$, provided that $s\ell^{(h)}(q) \neq root$.

$val(q)$ of length k. This is the same as saying, more formally, that $\phi(q,k)$ is the node $s\ell^{(i)}(q)$ such that

$$length(s\ell^{(i+1)}(q)) < k \leq length(s\ell^{(i)}(q))\,,$$

for each $q \in Q$ and each integer $k \leq length(q)$. Roughly speaking, $\phi(q,k)$ is the first node p in the suffix path of q such that $length(s\ell(p)) < k$.

In the preprocessing phase, the DAWG $\mathcal{A}(P) = (Q, \Sigma, \delta, root, F)$ together with the associated *end-pos* function is computed. Since for a pattern P of length m we have that $|Q| \leq 2m+1$ and $|end\text{-}pos(q)| \leq m$, for each $q \in Q$, we need only $\mathcal{O}(m^2)$ extra space (see [3,4]).

To compute the set $\mathcal{F}_j^k$, for $1 \leq k \leq l_j$, one can take advantage of the following relation

$$\mathcal{F}_j^k = end\text{-}pos(\phi(q_j,k))\,. \tag{1}$$

Notice that, in particular, we have $\mathcal{F}_j^{l_j} = end\text{-}pos(q_j)$.

The time complexity of the computation of $\phi(q,k)$ can be bounded by the length of the suffix path of node q. Specifically, since the sequence

$$(length(s\ell^{(0)}(q)), length(s\ell^{(1)}(q)), \ldots, 0)$$

of the lengths of the nodes in the suffix path from q is strictly deacreasing, we can do at most $length(q)$ iterations over the suffix link, obtaining a $\mathcal{O}(m)$-time complexity.

According to Lemma 2, a translocation of length $2k$ at position j of the text T is possible only if factors of P of length at least k have been recognized at both positions j and $j-k$, namely if $l_j \geq k$ and $l_{j-k} \geq k$.

Let $\langle k_1, k_2, \ldots, k_r \rangle$ be the increasing sequence of all values k such that $1 \leq k \leq \min(l_j, l_{j-k})$. For each $1 \leq i \leq r$, condition (b) of Lemma 2 requires member queries on the sets $\mathcal{F}_j^{k_i}$ and $\mathcal{F}_{j-k_i}^{k_i}$.

We notice that, if we proceed for decreasing values of k, the sets $\mathcal{F}_j^k$, for $1 \leq k \leq l_j$, can be computed in constant time. Specifically, the set $\mathcal{F}_j^k$ can be computed in constant time from $\mathcal{F}_j^{k+1}$, for $k = 1, \ldots, l_j - 1$, with at most one iteration over the suffix link of the state $\phi(q_j, k+1)$.

The computation of $\mathcal{F}_{j-k_r}^{k_r}$ has a $\mathcal{O}(\alpha)$-time complexity, since $length(q_{j-k_r}) \leq \alpha$. To compute $\mathcal{F}_{j-k_i}^{k_i}$, for $i = r-1, r-2, \ldots, 1$, we distinguish the following two cases:

Case $k_{i+1} = k_i + 1$**:** Let $q' = \phi(q_{j-k_{i+1}}, k_{i+1})$. Given the node q' computed in the previous iteration, the node $\phi(q_{j-k_i}, k_i)$ can be computed in two steps: first, we look up the node corresponding to the suffix of length $k_{i+1} - 2$ of the factor represented by q', with at most two iterations of the suffix link of q'; then, we perform a transition on $T[j-k_i]$ on the node so found. Formally:

$$\phi(q_{j-k_i}, k_i) = \delta(\phi(q', k_{i+1} - 2), T[j-k_i])\,.$$

Case $k_{i+1} > k_i + 1$**:** Observe that $l_{j-s} \leq s-1$ must hold, for each $s = k_{i+1}-1, \ldots, k_i+1$. In particular, we have $l_{j-(k_i+1)} \leq k_i$ which implies that $l_{j-k_i} \leq k_i + 1$ since $l_j \leq l_{j-1} + 1$ always holds. Hence, the computation of $\phi(q_{j-k_i}, k_i)$ requires at most one iteration of the suffix link of q_{j-k_i}.

```
M-Sampling (P, T, α, β, A, A')
   /* A is the DAWG of P and A' is the DAWG of P^r */
1.  m ← |P|,   n ← |T|
2.  (q_0, l_0) ← Dawg-Delta(root_A, 0, α, T[0], A)
3.  (q'_0, l'_0) ← Dawg-Delta(root_A', 0, β, T[0], A')
4.  S_0 ← ∅
5.  if P[0] = T[0] then S_0 ← {0}
6.  for j ← 1 to n − 1 do
7.    (q_j, l_j) ← Dawg-Delta(q_{j−1}, l_{j−1}, α, T[j], A)
8.    (q^r_j, l^r) ← Dawg-Delta(q^r_j, l^r, β, T[j], A')
9.    S_j ← ∅
      /* Standard matches */
10.   if P[0] = T[j] then S_j ← {0}
11.   for i ∈ S_{j−1} do
12.     if i < m − 1 and P[i + 1] = T[j] then
13.       S_j ← S_j ∪ {i + 1}
      /* Inversions */
14.   p ← q^r_j
15.   for k ← l^r_j downto 2 do
16.     for i ∈ S_{j−k} ∪ {−1} do
17.       if (m − 2 − i) ∈ end-pos^r(p) then
18.         S_j ← S_j ∪ {i + k}
19.     if k = length(sℓ_{A'}(p)) + 1 then
20.       p ← sℓ_{A'}(p)
      /* Translocations */
21.   last ← 0
22.   p ← q_j
23.   for k ← l_j downto 1 do
24.     if k ≤ j and k ≤ l_{j−k} then
25.       if last = k + 1 then
26.         while p' ≠ root_A
              and k − 1 ≤ length(sℓ_A(p')) do
27.           p' ← sℓ_A(p')
28.         p' ← δ_A(p', T[j − k])
29.       else
30.         p' ← q_{j−k}
31.         while k ≤ length(sℓ_A(p')) do
32.           p' ← sℓ_A(p')
33.       last ← k
34.       for i ∈ S_{j−2k} ∪ {−1} do
35.         if (i + k) ∈ end-pos(p)
              and (i + 2k) ∈ end-pos(p') then
36.           S_j ← S_j ∪ {i + 2k}
37.     if k = length(sℓ_A(p)) + 1
38.       then p ← sℓ_A(p)
39.   if (m − 1) ∈ S_j then
40.     Output(j)
```

```
DAWG-DELTA(q, l, k, c, B)
1.  if l = k then
2.    l ← l − 1
3.    if length(sℓ_B(q)) = l
4.      then q ← sℓ_B(q)
5.  if δ_B(q, c) = NIL then
6.    do
7.      q ← sℓ*_B(q)
8.    while q ≠ NIL and δ_B(q, c) = NIL
9.    if q = NIL then
10.     l ← 0, q ← root_B
11.   else l ← length(q) + 1
12.        q ← δ_B(q, c)
13. else l ← l + 1
14.      q ← δ_B(q, c)
15. return (q, l)
```

Figure 1. On the left: the M-Sampling algorithm for solving the pattern matching problem with translocations and inversions. **On the right:** the DAWG state update algorithm.

Thus, in both cases, $\mathcal{F}^{k_i}_{j-k_i}$ can be computed in constant time, for $1 \leq i < r$. Therefore, the total complexity for computing all the sets $\mathcal{F}^{k_i}_{j-k_i}$, for $i = 1, \ldots, r$, is $\mathcal{O}(\alpha)$.

Next, to compute the sets $\mathcal{I}^k_j$ we use the DAWG $\mathcal{A}(P^{\mathsf{r}})$ of P^{r}. Specifically, we compute the longest reversed factor ending at j and maintain the invariant that the current state of the automaton never corresponds to factors longer than β, using algorithm given in Fig. 1 (on the right), as for the computation of the sets $\mathcal{F}^k_j$. Let

$(q_j^{\mathsf{r}}, l_j^{\mathsf{r}})$ denote the T-configuration of $\mathcal{A}(P^{\mathsf{r}})$ after having read the character of T at position j, where l_j^{r} is the length of the longest reversed factor of P recognized. Then the sets $\mathcal{I}_j^k$ can be computed, for $2 \leq k \leq l_j^{\mathsf{r}}$, by

$$\mathcal{I}_j^k = \{i \mid (m-i+k-2) \in \mathit{end\text{-}pos}(\phi(q_j^{\mathsf{r}}, k))\}\,. \tag{2}$$

Indeed, $i \in \mathcal{I}_j^k$ iff $P[i-k+1\,..\,i] = (T[j-k+1\,..\,j])^{\mathsf{r}}$ iff

$$P^{\mathsf{r}}[(m-1)-i\,..\,(m-1)-(i-k+1)] = (P[i-k+1\,..\,i])^{\mathsf{r}} = T[j-k+1\,..\,j]\,.$$

Thus (2) follows, since the latter is equivalent to $(m-i+k-2) \in \mathit{end\text{-}pos}(\phi(q_j^{\mathsf{r}}, k))$.

For each $k = 1, \ldots, l_j^{\mathsf{r}}$, condition (c) of Lemma 2 requires member queries on the sets $\mathcal{I}_j^k$. As in the case of the sets $\mathcal{F}_j^k$, the set $\mathit{end\text{-}pos}(\phi(q_j^{\mathsf{r}}, k))$ can be computed in constant time, in decreasing order of k, by iterating the suffix link on q_j^{r}. Although $\mathcal{I}_j^k$ is not equal to $\mathit{end\text{-}pos}(\phi(q_j^{\mathsf{r}}, k))$, a member query on $\mathcal{I}_j^k$ can still be done in constant time, using (2).

4 Complexity analysis

We first analyze the worst-case time complexity of the M-Sampling algorithm and then its average-case complexity. Our analysis assumes that sets are implemented as bit vectors so that any member query on a set takes constant time. We will also evaluate the space complexity of the M-Sampling algorithm.

4.1 Worst-case analysis

First of all, observe that the main **for**-loop at line 6 is always executed n times. Moreover, observe that $|\mathcal{S}_j| \leq m$, $l_j \leq \alpha$, and $l_j^{\mathsf{r}} \leq \beta$, for all $0 \leq j < n$. For each iteration of the **for**-loop at line 23, the amortized cost of the two **while**-loops at lines 26 and 31 is $\mathcal{O}(1)$. Thus, at each iteration of the main **for**-loop, the **for**-loop at line 11 takes at most $\mathcal{O}(m)$ time while the **for**-loops at lines 15 and 23 take at most $\mathcal{O}(m\beta)$ and $\mathcal{O}(m\alpha)$ time respectively. Summing up, the algorithm has a $\mathcal{O}(nm\max(\alpha, \beta))$ worst-case time complexity, which becomes $\mathcal{O}(nm^2)$-time when $\max(\alpha, \beta) = \Theta(m)$.

4.2 Average-case analysis

Next, we evaluate the average time complexity of the algorithm M-Sampling assuming the uniform distribution and independence of characters.

Given integers $1 \leq \alpha, \beta \leq m \leq n$ and an alphabet Σ of size $\sigma \geq 4$, for $j = 0, 1, \ldots, n-1$ we consider the following nonnegative random variables over the sample space of the pairs of strings $P, T \in \Sigma^*$ of length m and n, respectively:

- $X(j) =_{\text{Def}}$ the length $l_j \leq \alpha$ of the longest factor of P which is a suffix of T_j,
- $Y(j) =_{\text{Def}}$ the length $l_j^{\mathsf{r}} \leq \beta$ of the longest factor of P^{r} which is a suffix of T_j,
- $Z(j) =_{\text{Def}} |\mathcal{S}_j|$, where we recall that $\mathcal{S}_j = \{0 \leq i \leq m-1 \mid P_i \sqsupseteq_{md} T_j\}$.

Then the run-time of a call to the M-Sampling algorithm with parameters (P, T, α, β) is proportional to

$$\sum_{j=1}^{n-1} \left(Z(j-1) + \sum_{k=2}^{Y(j)} Z(j-k) + \left(\sum_{k=1}^{X(j)} Z(j-2k) + X(j) \right) \right) , \tag{3}$$

where the external summation refers to the main **for**-loop (at line 6), and the three terms within it take care of the internal **for**-loops at lines 11, 15, and 23, in that order.

The average-case complexity of the M-Sampling algorithm is thus the expectation of (3), which, in view of the linearity of expectation, is equal to

$$\sum_{j=1}^{n-1} \left(E(Z(j-1)) + E\left(\sum_{k=2}^{Y(j)} Z(j-k) \right) + E\left(\sum_{k=1}^{X(j)} Z(j-2k) \right) + E(X(j)) \right) . \tag{4}$$

Since

$$\begin{aligned} E(X(j)) &\leq E(X(n-1)) \\ E(Y(j)) &\leq E(Y(n-1)) \\ E(Z(j)) &\leq E(Z(n-1)) , \end{aligned}$$

for $0 \leq j \leq n-1$,[2] and also

$$E(X(n-1)) = E(Y(n-1)) ,$$

by putting

$$X =_{\text{Def}} X(n-1) \qquad \text{and} \qquad Z =_{\text{Def}} Z(n-1) ,$$

expression (4) gets bounded from above by

$$\sum_{j=1}^{n-1} \left(E(Z) + E\left(\sum_{k=2}^{X} Z \right) + E\left(\sum_{k=1}^{X} Z \right) + E(X) \right) . \tag{5}$$

For $i = 0, \ldots, m-1$, let Z_i be the indicator variable

$$Z_i =_{\text{Def}} \begin{cases} 1 & \text{if } i \in \mathcal{S}_{n-1} \\ 0 & \text{otherwise} , \end{cases}$$

so that

$$Z = \sum_{i=0}^{m-1} Z_i \qquad \text{and} \qquad E(Z_i^2) = E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\} .$$

Likewise, for $k = 1, \ldots, m$, let X_k be the indicator variable

$$X_k =_{\text{Def}} \begin{cases} 1 & \text{if } X \geq k \\ 0 & \text{otherwise} , \end{cases}$$

so that

$$X = \sum_{k=1}^{m} X_k \qquad \text{and} \qquad E(X_k^2) = E(X_k) = Pr\{X \geq k\} .$$

[2] In fact, for $j = m, \ldots, n-1$ all inequalities hold as equalities.

The we have

$$\sum_{k=1}^{X} Z = XZ = \left(\sum_{k=1}^{m} X_k\right) \cdot \left(\sum_{i=0}^{m-1} Z_i\right) = \sum_{k=1}^{m}\sum_{i=0}^{m-1} X_k Z_i \,.$$

Therefore

$$E\left(\sum_{k=2}^{X} Z\right) \leq E\left(\sum_{k=1}^{X} Z\right) = \sum_{k=1}^{m}\sum_{i=0}^{m-1} E(X_k Z_i) \;,$$

yielding the following upper bound for (5):

$$\sum_{j=1}^{n-1}\left(E(Z) + 2 \cdot \sum_{k=1}^{m}\sum_{i=0}^{m-1} E(X_k Z_i) + E(X)\right) \;. \tag{6}$$

To estimate each of the terms $E(X_k Z_i)$ in (6), we use the well-known Cauchy-Schwarz inequality which in the context of expectations assumes the form

$$|E(UV)| \leq \sqrt{E(U^2)E(V^2)}\,,$$

for any two random variables U and V such that $E(U^2)$, $E(V^2)$ and $E(UV)$ are all finite.

Then, for $1 \leq k \leq m$ and $0 \leq i \leq m-1$, we have

$$E(X_k Z_i) \leq \sqrt{E(X_k^2)E(Z_i^2)} = \sqrt{E(X_k)E(Z_i)}\,. \tag{7}$$

From (7), it then follows that (6) is bounded from above by

$$\begin{aligned}
&\sum_{j=1}^{n-1}\left(E(Z) + 2 \cdot \sum_{k=1}^{m}\sum_{i=0}^{m-1} \sqrt{E(X_k)E(Z_i)} + E(X)\right) \\
&\quad = \sum_{j=1}^{n-1}\left(E(Z) + 2 \cdot \left(\sum_{k=1}^{m} \sqrt{E(X_k)}\right) \cdot \left(\sum_{i=0}^{m-1} \sqrt{E(Z_i)}\right) + E(X)\right) \;.
\end{aligned} \tag{8}$$

To better understand (8), we evaluate the expectations $E(X)$ and $E(Z)$ and the sums $\sum_{k=1}^{m} \sqrt{E(X_k)}$ and $\sum_{i=0}^{m-1} \sqrt{E(Z_i)}$. To this purpose, it will be useful to estimate also the expectations

- $E(X_k) = Pr\{X \geq k\}$, for $1 \leq k \leq m$, and
- $E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\}$, for $0 \leq i \leq m-1$.

Concerning $E(X_k) = Pr\{X \geq k\}$, we reason as follows. Since $T[n-k\,..\,n-1]$ ranges uniformly over a collection of σ^k strings and there can be at most $\min(\sigma^k, m-k+1)$ distinct factors of length k in P, the probability $Pr\{X \geq k\}$ that one of them matches $T[n-k\,..\,n-1]$ is at most $\min\left(1, \frac{m-k+1}{\sigma^k}\right)$, so that, for $k = 1, \ldots, m$, we have

$$E(X_k) \leq \min\left(1, \frac{m-k+1}{\sigma^k}\right) \;. \tag{9}$$

Then, in view of (9), we have:

$$E(X) = \sum_{i=0}^{m} i \cdot Pr\{X = i\} = \sum_{i=1}^{m} Pr\{X \geq i\} \leq \sum_{i=1}^{m} \min\left(1, \frac{m-i+1}{\sigma^i}\right) \;. \tag{10}$$

Let $\overline{k}$ be the smallest integer $1 \leq k < m$ such that $\frac{m-k+1}{\sigma^k} < 1$. Then from (10) we have

$$\begin{aligned} E(X) &\leq \sum_{i=1}^{\overline{k}-1} 1 + \sum_{i=\overline{k}}^{m} \frac{m-i+1}{\sigma^i} \leq \overline{k} - 1 + (m-\overline{k}+1) \sum_{i=\overline{k}}^{m} \frac{1}{\sigma^i} \\ &< \overline{k} - 1 + \frac{\sigma}{\sigma-1} \cdot \frac{m-\overline{k}+1}{\sigma^{\overline{k}}} < \overline{k} - 1 + \frac{\sigma}{\sigma-1} < \overline{k} + 1 \,. \end{aligned} \tag{11}$$

Since $\frac{m-(\overline{k}+1)+1}{\sigma^{\overline{k}+1}} \geq 1$, then $\sigma^{\overline{k}+1} \leq m - (\overline{k}+1) + 1 \leq m - 1$, so that

$$\overline{k} + 1 < \log_\sigma m \,. \tag{12}$$

From (11) and (12), we obtain

$$E(X) < \log_\sigma m \,. \tag{13}$$

Likewise, from (9) and (12) we have

$$\begin{aligned} \sum_{k=1}^{m} \sqrt{E(X_k)} &\leq \sum_{k=1}^{m} \sqrt{\min\left(1, \frac{m-k+1}{\sigma^k}\right)} = \sum_{k=1}^{\overline{k}-1} 1 + \sum_{k=\overline{k}}^{m} \sqrt{\frac{m-k+1}{\sigma^k}} \\ &\leq \overline{k} - 1 + \sqrt{m-\overline{k}+1} \cdot \sum_{k=\overline{k}}^{m} \frac{1}{\sqrt{\sigma^k}} < \overline{k} - 1 + \frac{\sqrt{\sigma}}{\sqrt{\sigma}-1} \cdot \sqrt{\frac{m-\overline{k}+1}{\sigma^{\overline{k}}}} \\ &< \overline{k} - 1 + \frac{\sqrt{\sigma}}{\sqrt{\sigma}-1} \leq \overline{k} + 1 < \log_\sigma m \,, \end{aligned} \tag{14}$$

where $\overline{k}$ is defined as above.

Next we estimate $E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\}$, for $0 \leq i \leq m-1$.

Let us denote by $\mu(i)$ the number of distinct strings which have an *md*-match with a given string of length i and whose characters are pairwise distinct. Then

$$Pr\{P_i \sqsupseteq_{md} T\} \leq \frac{\mu(i+1)}{\sigma^{i+1}} \,.$$

From the recursion

$$\begin{cases} \mu(0) = 1 \\ \mu(k+1) = \sum_{h=0}^{k} \mu(h) + \sum_{h=1}^{\lfloor \frac{k-1}{2} \rfloor} \mu(k-2h-1) \qquad (\text{for } k \geq 0) \,, \end{cases}$$

it is not hard to see that $\mu(i+1) \leq 3^i$, for $i = 0, 1, \ldots, m-1$, so that we have

$$E(Z_i) = Pr\{P_i \sqsupseteq_{md} T\} \leq \frac{3^i}{\sigma^{i+1}} \,. \tag{15}$$

Then, concerning $E(Z)$, from (15) we have

$$E(Z) = E\left(\sum_{i=0}^{m-1} Z_i\right) = \sum_{i=0}^{m-1} E(Z_i) \leq \sum_{i=0}^{m-1} \frac{3^i}{\sigma^{i+1}} < \frac{1}{\sigma} \cdot \frac{1}{1-\frac{3}{\sigma}} = \frac{1}{\sigma-3} \leq 1 \tag{16}$$

(we recall that we have assumed $\sigma \geq 4$).

Likewise, from (15) we have

$$\sum_{i=0}^{m-1} \sqrt{E(Z_i)} \leq \sum_{i=0}^{m-1} \sqrt{\frac{3^i}{\sigma^{i+1}}} < \frac{1}{\sqrt{\sigma}} \cdot \frac{1}{1-\sqrt{\frac{3}{\sigma}}} = \frac{1}{\sqrt{\sigma}-\sqrt{3}} < 4\,. \tag{17}$$

From (16), (13), (14), and (17), it then follows that (8) is bounded from above by

$$(n-1) \cdot (9 \log_\sigma m + 1)\,,$$

yielding a $\mathcal{O}(n \log_\sigma m)$ average-time complexity for the M-Sampling algorithm.

4.3 Space complexity

In order to evaluate the space complexity of the M-Sampling algorithm, we observe that in the worst case, during the j-th iteration of its main **for**-loop, the sets $\mathcal{F}^k_{j-k}$ and $\mathcal{S}_{j-2k}$, for $1 \leq k \leq \alpha$, must be kept in memory to handle translocations, as well as the sets $\mathcal{S}_{j-k}$, for $2 \leq k \leq \beta$, to handle inversions. However, as explained before, we do not keep the values of $\mathcal{F}^k_{j-k}$ explicitly but rather we maintain only their corresponding T-configurations of the automaton $\mathcal{A}(P)$. Thus, we need $\mathcal{O}(\alpha)$-space for the last α configurations of the automaton and $\mathcal{O}(m \max(\alpha, \beta))$-space to keep the last $\max(2\alpha, \beta)$ values of the sets $\mathcal{S}_{j-k}$, considering the maximum cardinality of each set is m. Observe also that, although the size of the DAWG is linear in m, the *end–pos*$(\cdot)$ function can require $\mathcal{O}(m^2)$-space. Therefore, the total space complexity of the M-Sampling algorithm is $\mathcal{O}(m^2)$.

5 Experimental evaluation

In this section we present some experimental results which allow to compare in terms of running times the M-Sampling algorithm, based on the DAWG approach, against its direct dynamic programming implementation. We have also included in our comparison the variant BPM-Sampling of the M-Sampling algorithm, based on the bit-parallelism technique [2], which is briefly described in Appendix A.

We remark that sets have been implemented as bit vectors also in the first two algorithms, so that *member* and *insert* operations can be performed in constant time.

Iteration over the elements of a set represented as a bit vector can then be implemented efficiently in time proportional to its cardinality by repeatedly

(a) extracting the lowest bit set,
(b) computing its index, and
(c) masking it, until there are no more bits set.

Observe also that the index of the lowest bit set of a word x can be computed very efficiently by the operation

$$\lfloor \log_2(x \;\&\; (\sim x + 1)) \rfloor\,,$$

where & and $\sim$ stand respectively for the bitwise and and the bitwise complementation.

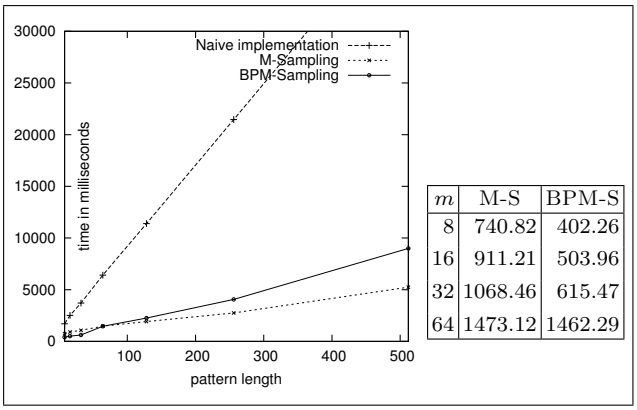
In the BPM-Sampling algorithm, bitwise operations have a $\Theta(\lceil m/w \rceil)$ complexity, since they have to update $\lceil m/w \rceil$ words. Instead, in the M-Sampling algorithm

the corresponding operations have a $\Theta(\lceil m/w \rceil + |\mathcal{S}_j|)$ complexity, because, for each word of the bit vector that encodes $\mathcal{S}_j$, it iterates over all the bits set ($|\mathcal{S}_j|$ in total). Since, on average, the sets $\mathcal{S}_j$ contain only a few elements, the average complexity of iterating over all the elements of a set is $\mathcal{O}(\lceil m/w \rceil)$.

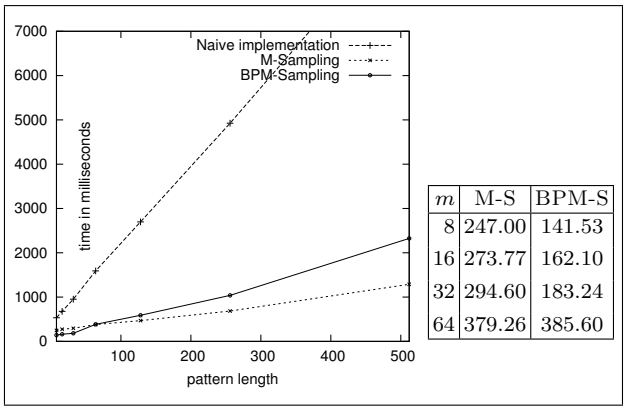
All algorithms have been implemented in the `C` programming language and have been compiled with the `GNU C Compiler`, using the optimization options `-O2 -fno-guess-branch-probability`. The tests have been performed on a 1.5 GHz PowerPC G4 with a computer word of size 32 and running times have been measured with a hardware cycle counter, available on modern CPUs.

As input files, we used a genome sequence of 4,638,690 base pairs of *Escherichia coli* [1] and a protein sequence from the *Saccharomyces cerevisiae* genome [9].

For each input file, we have generated sets of 50 patterns of fixed length m, randomly extracted from the text, for m ranging in the set {8, 16, 32, 64, 128, 256, 512}. For each set of patterns, we have calculated the mean over the running times of the 50 runs.



m	M-S	BPM-S
8	740.82	402.26
16	911.21	503.96
32	1068.46	615.47
64	1473.12	1462.29



m	M-S	BPM-S
8	247.00	141.53
16	273.77	162.10
32	294.60	183.24
64	379.26	385.60

Figure 2. Experimental results relative to a genome sequence of *Escherichia coli* with $\sigma = 4$ (on the left) and to a protein sequence of the *Saccharomyces cerevisiae* genome with $\sigma = 20$ (on the right). To ease comparison of the M-Sampling algorithm (M-S) and the BPM-Sampling algorithm (BPM-S) for small values of m, we have also tabulated their running times.

As can be seen from the plots in Figure 2, the M-Sampling algorithm is considerably faster than its naive implementation. Indeed, even if their asymptotic time complexity is the same, the hidden constant in the naive implementation, due to the explicit computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$, is quite large. In our experiment with a computer word of size 32, it turns out that the BPM-Sampling algorithm is faster than the M-Sampling algorithm only for $m \leq 32$, as can be observed by looking at the running times (in milliseconds) reported in the tables. As explained above, the complexity of the bitwise operations, on average, is the same for both algorithms. However, the M-Sampling algorithm scales better because it requires fewer bitwise operations. Finally, observe that the rate of growth of the M-Sampling and the BPM-Sampling aglorithm matches the average $\mathcal{O}(n \log_\sigma m)$-time complexity estimated in Section 4.2 under the assumptions of equiprobability and independence of characters.

6 Conclusions

In this paper we have presented an algorithm, based on the dynamic programming paradigm, to solve the pattern matching problem under a string distance which allows translocations of equal length adjacent factors and inversions of factors. Our algorithm, named M-Sampling, has a worst-case $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space complexity, where α and β are respectively the bounds on the maximum length of any factor involved in a translocation and in an inversion. Moreover, we have shown that under the assumption of equiprobability and independence of characters, the M-Sampling algorithm has a $\mathcal{O}(n \log_\sigma m)$ average-time complexity. Finally, in the appendix we have also briefly described an efficient implementation of the M-Sampling algorithm based on the bit-parallelism technique, which achieves a worst-case $\mathcal{O}(n\lceil m/w \rceil \max(\alpha, \beta))$-time and $\mathcal{O}(\sigma\lceil m/w \rceil + m\lceil m/w \rceil)$-space complexity.

We are currently investigating how to extend our approach to handle efficiently also translocations of factors which are not necessarily adjacent or of equal length and how to to compute the minimum cost, when the weights are either unitary or generic.

Acknowledgements

The authors wish to thank Paul Doukhan and Salvatore Ingrassia for helpful suggestions.

References

1. R. Arnold and T. Bell: *A corpus for the evaluation of lossless compression algorithms*, in DCC'97: Proceedings of the Conference on Data Compression, Washington, DC, USA, 1997, IEEE Computer Society, `http://corpus.canterbury.ac.nz/`.
2. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992.
3. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theor. Comput. Sci., 40(1) 1985, pp. 31–55.
4. M. Crochemore: *Transducers and repetitions.* Theor. Comput. Sci., 45(1) 1986, pp. 63–86.
5. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
6. F. J. Damerau: *A technique for computer detection and correction of spelling errors.* Commun. ACM, 7(3) 1964, pp. 171–176.
7. V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions and reversals.* Sov. Phys. Dokl., 10 1966, pp. 707–710.
8. G. Navarro: *A guided tour to approximate string matching.* ACM Comp. Surv., 33(1) 2001, pp. 31–88.
9. C. G. Nevill-Manning and I. H. Witten: *Protein is incompressible*, in DCC'99: Proceedings of the Conference on Data Compression, Washington, DC, USA, 1999, IEEE Computer Society, `http://data-compression.info/Corpora/ProteinCorpus/`.
10. E. Ukkonen: *Approximate string-matching with q-grams and maximal matches.* Theor. Comput. Sci., 92(1) 1992.
11. E. Ukkonen and D. Wood: *Approximate string matching with suffix automata.* Algorithmica, 10(5) 1993.

A A bit-parallel implementation

In this appendix we present an efficient simulation of the M-Sampling algorithm based on the bit-parallelism technique [2]. The bit-parallelism technique takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor of at most w, where w is the number of bits in the computer word. All sets are represented by vectors of m bits, where m is the length of the pattern. The i-th bit of a vector is set to 1 if the element i belongs to the corresponding set, 0 otherwise. Note that if $m \leq w$, a whole vector fits into a single computer word, whereas if $m > w$ then $\lceil m/w \rceil$ computer words are needed to represent each set.

In the following we denote with & the bitwise and, with | the bitwise or, and with $\ll$ the shift-to-left operator.

We associate to each node of the DAWG a bit vector pos. For each node q of the DAWG of P, $\mathsf{pos}(q)$ encodes the *end-pos* function, while, for each node q of the DAWG of P^{r}, $\mathsf{pos}(q)$ encodes the starting positions in P of the reversed factors represented by the node, i.e. $\{(m-1-i) \mid i \in \mathit{end\text{-}pos}(q)\}$.

The bit-vectors F_j^k and I_j^k, corresponding to $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ respectively, can be computed by the following assignments:

$$\begin{aligned} \mathsf{F}_j^k &\leftarrow \mathsf{pos}(\phi(q_j, k)) \\ \mathsf{I}_j^k &\leftarrow \mathsf{pos}(\phi(q_j^{\mathsf{r}}, k)) \ll (k-1)\,. \end{aligned}$$

Each set $\mathcal{S}_j$ is mapped into a corresponding bit-vector S_j. Finally, for each character c of the alphabet Σ, a bit mask $\mathsf{B}[c]$, representing the positions of c in P, is maintained.

The algorithm scans T from left to right and, for each position $j \geq 0$, it computes the vector S_j in terms of S_{j-1}, of S_{j-2k}, F_{j-k}^k, and F_j^k, for $1 \leq k \leq l_j$, and of S_{j-k} and I_j^k for $1 \leq k \leq l_j^{\mathsf{r}}$, with the following bitwise operations:

$$\begin{aligned} &\mathsf{S}_j \leftarrow ((\mathsf{S}_{j-1} \ \ll \ 1) \mid 1) \ \& \ B[T[j]] \\ &\mathsf{S}_j \leftarrow \mathsf{S}_j \mid ((((\mathsf{S}_{j-2k} \ll k) \mid (1 \ll (k-1))) \ \& \ \mathsf{F}_j^k) \ll k) \ \& \ \mathsf{F}_{j-k}^k \\ &\mathsf{S}_j \leftarrow \mathsf{S}_j \mid (((\mathsf{S}_{j-k} \ll k) \mid (1 \ll (k-1))) \ \& \ \mathsf{I}_j^k)\,, \end{aligned}$$

corresponding respectively to the relations:

$$\begin{aligned} &\mathcal{S}_j = \{i+1 : i \in \mathcal{S}_{j-1} \cup \{-1\} \wedge P[i] = T[j]\} \\ &\mathcal{S}_j = \mathcal{S}_j \cup \{i+2k : i \in \mathcal{S}_{j-2k} \cup \{-1\} \wedge (i+k) \in \mathcal{F}_j^k \wedge (i+2k) \in \mathcal{F}_{j-k}^k\} \\ &\mathcal{S}_j = \mathcal{S}_j \cup \{i+k : i \in \mathcal{S}_{j-k} \cup \{-1\} \wedge (i+k) \in \mathcal{I}_j^k\}\,. \end{aligned}$$

During the j-th iteration, if the m-th bit of S_j is set to 1, i.e., if $\mathsf{S}_j \ \& \ 10^{m-1} \neq 0^m$, a match at position j is reported.

The resulting algorithm has a $\mathcal{O}(n \max(\alpha, \beta)\lceil m/w \rceil)$ worst-case time complexity and a $\mathcal{O}((m+\sigma)\lceil m/w \rceil)$-space complexity, where σ is the size of the alphabet. When the length of the pattern satisfies $m \leq w$, the worst-case time and space complexity become $\mathcal{O}(n \max(\alpha, \beta))$ and $\mathcal{O}(\sigma + m)$, respectively.

(In)approximability Results for Pattern Matching Problems

Raphaël Clifford and Alexandru Popa

Department of Computer Science
University of Bristol
Merchant Venturer's Building
Woodland Road, Bristol, BS8 1UB
United Kingdom
{clifford,popa}@cs.bris.ac.uk

Abstract. We consider the approximability of three recently introduced pattern matching problems which have been shown to be **NP**-hard. Given two strings as input, the first problem is to find the longest common parameterised subsequence between two strings. The second is a maximisation variant of generalised function matching and the third is a a maximisation variant of generalised parameterised matching. We show that in all three cases there exists an $\epsilon > 0$ such that there is no polynomial time $(1-\epsilon)$-approximation algorithm, unless $\mathbf{P} = \mathbf{NP}$. We then present a polynomial time $\sqrt{1/OPT}$-approximation algorithm for a variant of generalised parameterised matching for which no previous approximation results are known.

1 Introduction

We investigate the complexity and approximability of three recently introduced classes of pattern matching problems. Given two strings, typically termed the pattern and text, the traditional pattern matching question is to determine the minimum number of operations required to transform the pattern to either the whole or some portion of the text. The challenge arises in the specific detail of the definition of an operation or more generally in how the distance between two strings is measured. Popular examples have included pattern matching under the Hamming norm [1,9] and the edit distance [10,11] where efficient polynomial time algorithms are known. The algorithms are highly dependent on the distance measure being considered. Recent results have shown that when two or more different measures of distance are combined, the resulting problem may be **NP**-hard despite the individual measures sometimes permitting efficient linear time solutions [3,6].

The first problem we consider is known as longest common parameterised subsequence (LCPS). This combined problem is introduced by Keller et. al. in [8] and introduces the property of parameterisation into the classic and extensively studied problem of determining the longest common subsequence (LCS) between two strings. The term parameterisation, as introduced by Baker [5] in the pattern matching context, refers to the relabelling of the characters of the input so as to transform the pattern into a match for the text. A particular example given for LCPS is the practical question of comparing two code fragments, an original, and a suspected copy, after the alleged copy has been edited both by inserting new code or comments and also by possibly renaming variables. LCPS was previously shown to be **NP**-hard to solve exactly. We prove a stronger bound. We show that it is also hard to approximate within a $(1-\epsilon)$ factor, for some $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. One consequence is that it is unlikely that any PTAS can be found.

Raphaël Clifford, Alexandru Popa: (In)approximability Results for Pattern Matching Problems, pp. 52–62 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

We then consider a class of problems introduced in 2004 by Amir and Nor where individual characters in the pattern are permitted to match entire substrings of the text [3]. These are called generalised matching problems and for optimisation variants of both generalised function matching (GFM) and generalised parameterised matching (GPM) **NP**-hardness results are also known [6]. In generalised matching a character of the pattern can be mapped to a substring of the text. The main difference between GFM and GPM is that in the latter case, no two characters can map to the same substring. These problems arise from a natural extension of parameterised matching and function matching, which was first considered by Amir et al. [2] and can be applied to problems where one wants to determine the structure of a text in terms of repeated patterns. For example, if $t = annabobanna$ and $p = ABA$, then the mapping $A \rightarrow anna$ and $B \rightarrow bob$ is valid both in the GFM and GPM models. However, if $t = bobbobbob$ and $p = ABA$, then $A \rightarrow bob$ and $B \rightarrow bob$ is permitted in the GFM model, but not for GPM [6].

We show that GFM and GPM are also hard to approximate within a $(1 - \epsilon)$ factor unless **NP** = **P**. Finally, we present the first polynomial time $\sqrt{1/OPT}$-approximation algorithm for GPM, in the case when the length of the text is at least twice the length of the pattern.

2 Hardness of the longest common parameterised subsequence problem

The *longest common parameterised subsequence (LCPS)* problem attempts to combine the LCS measure with parameterised matching. The definition of the LCPS problem is the following:

Problem 1. [8] The input consists of two strings of the same length $t = t_1 t_2 \dots t_n$ and $p = p_1 p_2 \dots p_n$ over an alphabet Σ. The goal is to find a permutation $\pi : \Sigma \rightarrow \Sigma$ such that the LCS between $\pi(t_1)\pi(t_2)\dots\pi(t_n)$ and $p_1 p_2 \dots p_n$ is maximized.

The problem is introduced by Keller et. al. in [8] where they show it is **NP**-hard. In this paper we prove that the problem is hard to approximate within $(1 - \epsilon)$, for some $\epsilon > 0$.

Before we present the main result, we give the definition of gap-preserving reduction between two maximisation problems. A similar definition is presented in [12] for the case when the first problem is a minimisation problem and the second is a maximisation problem.

Definition 2. *Assume Π_1 and Π_2 are maximisation problems. A gap-preserving reduction from Π_1 to Π_2 comes with four parameters (functions) f_1, α, f_2 and β. Given an instance x of Π_1 it computes in polynomial time, an instance y of Π_2 such that:*

$$OPT(x) \geq f_1(x) \Rightarrow OPT(y) \geq f_2(y)$$

$$OPT(x) < \alpha(|x|) f_1(x) \Rightarrow OPT(y) < \beta(|y|) f_2(y)$$

In our proofs we make use of the following remark which is also stated in [12].

Remark 3. [12] A gap-preserving reduction from Π_1 to Π_2 with the above parameters implies that if problem Π_1 cannot have a polynomial-time α-approximation, then problem Π_2 cannot have a polynomial-time β-approximation.

To prove the inapproximability result for the LCPS problem we use a gap-preserving reduction from MAX-3SAT(13) to LCPS. MAX-3SAT(13) is a variant of MAX-SAT problem in which each clause has exactly three literals (where a literal is either a variable or its negation) and each variable appears in at most 13 clauses. In [4] Arora proves that MAX-3SAT(13) cannot be approximated within a factor of $1 - \delta/19$, if MAX3-SAT cannot be approximated within a factor of $1 - \delta$ [4]. In [7] Håstad proves that MAX3-SAT cannot be approximated within a factor better than 7/8 and thus, MAX-3SAT(13) cannot be approximated within a factor of 151/152, unless $\mathbf{P} = \mathbf{NP}$.

Consider a MAX-3SAT(13) instance ϕ with n variables and m clauses. The alphabet Σ of the LCPS instance consists of a special character \$ which has the role of a delimiter between different blocks of text (the usefulness of this character becomes clearer when we present the reduction) and two characters corresponding to each variable x of ϕ, x_T and x_F. We denote by $\$^{15}$ the string formed by concatenating 15 \$ symbols.

The reduction is the following. For each variable x we add to the text t the gadget $x_T x_F x_T x_F \$^{15}$ and to the pattern p the gadget $x_T x_F x_F x_T \$^{15}$. Then, for each clause $x_1 \vee x_2 \vee x_3$ we add to the pattern the gadget $x_{3T} x_{2T} x_{1T} \$^{15}$ (notice that the variables are placed in reverse order), and to t, the gadget $x_{1V} x_{2V} x_{3V} \$^{15}$, where x_{iV}, for every $i = \{1, 2, 3\}$ is:

$$x_{iV} = \begin{cases} x_{iT} \text{ if } x_i \text{ is a non-negated variable} \\ x_{iF} \text{ if } x_i \text{ is a negated variable} \end{cases}$$

Example 4. Consider the formula:

$$\phi = (x \vee y \vee z) \wedge (x \vee \bar{y} \vee z)$$

The corresponding instance of the LCPS problem is:

$$t = x_T x_F x_T x_F \$^{15} y_T y_F y_T y_F \$^{15} z_T z_F z_T z_F \$^{15} x_T y_T z_T \$^{15} x_T y_F z_T \$^{15}$$
$$p = x_T x_F x_F x_T \$^{15} y_T y_F y_F y_T \$^{15} z_T z_F z_F z_T \$^{15} z_T y_T x_T \$^{15} z_T y_T x_T \$^{15}$$

Theorem 5 summarises the main inapproximability result for LCPS.

Theorem 5. *There exists an $\epsilon > 0$ such that the LCPS problem does not have a $(1 - \epsilon)$-approximation algorithm, unless $\mathbf{P} = \mathbf{NP}$.*

Proof. We have to prove that this is a gap-preserving reduction from MAX-3SAT(13). Specifically, we prove that there exists a constant $\sigma > 0$ such that:

$$OPT(\phi) = m \Rightarrow OPT(t, p) = f(|t|, |p|) \quad (1)$$
$$OPT(\phi) < (1 - \epsilon)m \Rightarrow OPT(t, p) < (1 - \sigma) f(|t|, |p|) \quad (2)$$

where $OPT(\phi)$ is the optimal value of MAX-3SAT(13) problem on the instance ϕ, $OPT(t, p)$ is the maximum LCPS on the text t, and pattern p constructed using the above reduction, and $f(|t|, |p|) = 3n + 15(n + m) + m$. These implications, together with Remark 3 prove the theorem.

Now, we explain why the two implications are true. The reduction forces the \$ symbol to be matched with itself under the permutation of the alphabet. Suppose that the \$ symbol is matched with another character, say x. In this case the LCPS can be at most $4n + 3m$ (this is the number of the characters in the text and in the

pattern which are different from \$), assuming that we can ideally match everything else. However a longer LCPS can be achieved by simply matching the \$ symbols alone.

The mapping \$ to \$ imposes a lot of structure on the other possible mappings: each pair (x_T, x_F) is either matched to (x_T, x_F), meaning that the variable x is assigned to *True* in the formula ϕ, or is matched to (x_F, x_T), meaning that the variable x is assigned to *False*. Since each variable appears in at most 13 clauses it is not optimal to match x_T or x_F with a character corresponding to any other variable y. This matching stops a block of 15 \$ symbols from the text to match a block of 15 \$ symbols from the pattern, but it can add no more than 13 to the LCPS length.

Thus, each variable gadget adds 3 to the final LCPS. Then, each block of dollars adds 15 to the final LCPS, independent of the other matchings. Finally, a gadget corresponding to a satisfied clause contributes exactly one to the LCPS. It cannot contribute more than one, if more than one literal is true in that clause, since we have placed the literals in reverse order in the pattern.

Therefore, if all the m clauses of ϕ can be satisfied, then the LCPS length has to be equal to $3n + 15(m + n) + m$. On the other hand, if at most $1 - \epsilon$ clauses of ϕ can be satisfied, then LCPS is at most $3n + 15(m + n) + (1 - \epsilon)m$, since the best strategy is to match a character according to the assignment of the corresponding variable: if a variable x is set to true in the assignment that maximises the number of satisfied clauses, then $\pi(x_T) = x_T$; otherwise $\pi(x_T) = x_F$.

To complete the proof of the second implication, we now explicitly calculate the value of σ. Since the optimal value of the LCPS is less than $3n+15(m+n)+(1-\epsilon)m$, we want the latter to be equal to $(1-\sigma)(3n+15(m+n)+m)$. By solving this equation we find the value of σ. Formally:

$$3n + 15(m + n) + (1 - \epsilon)m = (1 - \sigma)(3n + 15(m + n) + m)$$

Therefore,

$$\sigma = 1 - \frac{3n + 15(m + n) + (1 - \epsilon)m}{3n + 15(m + n) + m} = \frac{\epsilon m}{3n + 15(m + n) + m}$$

Since each variable appears in at most 13 clauses we can set $m = 13n$ and we get $\sigma = 13\epsilon/76$.

Therefore, the LCPS problem cannot be approximated within a factor of 0.99887, unless $\mathbf{P} = \mathbf{NP}$. □

3 Hardness of Generalised Function Matching

The two problems we consider in this section are termed generalised function matching (GFM) and generalised parameterised matching (GPM). Their formal definitions follow:

Problem 6. [6](GFM) Given a pattern p over an alphabet Σ_p and a text t over an alphabet Σ_t, determine if there exists a mapping f from Σ_p to Σ_t^+ such that $t = f(p_1)f(p_2)\ldots f(p_m)$.

Problem 7. [6](GPM) The problem of generalised parameterised matching (GPM) is defined in an analogous way to GFM except that f is now required to be an injection.

Recently, Clifford et. al. [6] prove that the two problems are **NP**-Complete under a wide range of conditions. They also define an optimisation version of the GFM problem, which is connected to the classical Hamming distance.

The Hamming similarity between two strings of the same length is the number of positions in which the two strings are equal. For input text t and pattern p, we are interested in the maximum Hamming similarity between p and any string p' of the same length which has a GFM match with t. As the original GFM problem is **NP**-Complete, this optimisation problem is **NP**-Hard. In the rest of the paper we refer to the "Hamming similarity", simply as "similarity".

To simplify the description of our approximation algorithms, we introduce the idea of a wildcard symbol which we define here.

Definition 8. *A* wildcard *is a special character which can be mapped to any substring of the text.*

We can replace the wildcard characters with regular characters as follows. In the GFM, just replace the wildcards with distinct characters. In the GPM, we partition the pattern alphabet into groups by which substring of the text they are mapped to. Each partition has only one non-wildcard character in it. Replace all the wildcard characters in a partition by the single non-wildcard character that are in the same partition.

We now show that there exists an $\epsilon > 0$ such that the problem of generalised function matching does not have a $(1-\epsilon)$-approximation algorithm, unless $\mathbf{P} = \mathbf{NP}$, via a gap preserving reduction from MAX3-SAT(13).

Consider a MAX-3SAT(13) instance ϕ with n variables and m clauses. The construction starts by placing $m+n$ \$ symbols to the beginning of both the pattern and the text. The text alphabet Σ_t has only two symbols, \$ and 0 with \$ serving as a delimiter in both the pattern and text. The pattern alphabet Σ_p includes the delimiter, a pair a_i and A_i for each variable and a distinct symbol c_i for each clause. The A_i's represent the negation of the variables a_i. The constructed pattern and text contain an equal number of \$ characters which forces \$ to map to \$ under any valid function. Also, as we show later, replacing some of the \$ symbols with wildcards cannot yield to an optimal strategy. We prove that the minimal Hamming distance between the pattern and the text is achieved when exactly one variable from each unsatisfied clause is replaced by a wildcard, where a wildcard, as we mention before, is a special character which is allowed to match any non-empty substring of the text.

For each variable a_i, we add to the text the string $\$^{13}000\$^{13}$ 13 times and to the pattern $\$^{13}a_iA_i\$^{13}$ 13 times. In this way a variable can be mapped to 00 or 0. Replacing all the 13 variables with wildcards is not optimal since a variable appears in at most 13 clauses and, therefore, at most 13 clauses can be satisfied. To fix notation we say that 0 represents True and 00 represents False. For each clause, we add to the text the string $\$^{13}000000\$^{13}$ (6 zeros) and to the pattern the string $\$^{13}xyzc_i\$^{13}$ where x, y, z are the variables from the clause (or their negations, as appropriate) and c_i is a different symbol for each clause.

Example 9. Consider the following instance of MAX3-SAT(13),

$$(x \vee y \vee z) \wedge (\bar{x} \vee y \vee \bar{z}) \ .$$

The GFM input is the following:

$$t = \$\$\$\$\$ \overbrace{\$^{13}000\$^{13} \ldots \$^{13}000\$^{13}}^{13} \overbrace{\$^{13}000\$^{13} \ldots \$^{13}000\$^{13}}^{13}$$

$$\overbrace{\$^{13}000\$^{13}\ldots\$^{13}000\$^{13}}^{13}\$^{13}000000\$^{13}\$^{13}000000\$^{13}$$

$$p = \$\$\$\$\$\overbrace{\$^{13}xX\$^{13}\ldots\$^{13}xX\$^{13}}^{13}\overbrace{\$^{13}yY\$^{13}\ldots\$^{13}yY\$^{13}}^{13}$$
$$\overbrace{\$^{13}zZ\$^{13}\ldots\$^{13}zZ\$^{13}}^{13}\$^{13}xyzc_1\$^{13}\$^{13}XyZc_2\$^{13}$$

The inapproximability result for the GFM problem is stated in Theorem 10.

Theorem 10. *There exists an $\epsilon > 0$ such that the GFM problem does not have a $(1-\epsilon)$-approximation algorithm, unless* $\mathbf{P} = \mathbf{NP}$.

Proof. We have to prove that this is a gap-preserving reduction from MAX-3SAT(13). Specifically, we prove that there exists a constant $\sigma > 0$ such that conditions (1) and (2) from the proof of Theorem 5 are satisfied, where $OPT(\phi)$ is the optimal value of the MAX-3SAT(13) problem on the instance ϕ, $OPT(t,p)$ is the maximum similarity on the text t, and the pattern p constructed using the above reduction, and $f(|t|,|p|) = (n+m) + 26 \cdot 13n + 26m + 2 \cdot 13n + 4m$. These implications, together with Remark 3 prove the theorem.

Now, we explain why the two implications are true. If ϕ is satisfiable, then there must be a GFM solution for p and t. This follows as we are guaranteed that not all the symbols from a clause can be mapped to 00 and therefore c_i can be matched to a nonempty substring in each clause gadget.

Now, suppose that at most $(1-\epsilon)m$ clauses of ϕ can be satisfied. Then, we must have at least ϵm wildcards in order to have a valid matching, one wildcard for each unsatisfied clause. This wildcard replaces one of the variables in that clause and matches only one 0 (notice that since this is an unsatisfied clause all the variables are matched to 00 and the dummy variable c_i does not have any 0's to be matched to) and gives the possibility of the last variable to be matched to a 0.

If one replaces all the dollar signs from the beginning with a wildcard the number of wildcards used is $n+m$, which is not optimal. If you replace with wildcards a block of 13 \$'s which separates the variable gadgets, then you can satisfy at most 13 new clauses, at a price of 13 wildcard symbols, which is not an improvement to the strategy presented in the previous paragraph. If you replace with wildcards a block of 13 \$'s which separates the clause gadgets, then you can satisfy at most 2 new clauses. The last option is to try to allow a variable to have an inconsistent assignment in order to satisfy more clauses, but to do this you need to place at least 13 wildcards and since a variable appears in at most 13 clauses, then this is not optimal.

Therefore, if at most $(1-\epsilon)m$ clauses of ϕ can be satisfied, then the maximum similarity has to be less or equal to $(n+m) + 26 \cdot 13n + 26m + 2 \cdot 13n + 4(1-\epsilon)m$. We want:

$$(n+m) + 26 \cdot 13n + 26m + 2 \cdot 13n + 4(1-\epsilon)m =$$
$$(1-\sigma)((n+m) + 26 \cdot 13n + 26m + 2 \cdot 13n + 4m)$$

As before, we can set $m = 13n$ and, therefore:

$$(1-\sigma) = \frac{716 + 52(1-\epsilon)}{768}$$

Therefore, the GFM problem cannot be approximated within a factor of 0.99955, unless $\mathbf{P} = \mathbf{NP}$. □

4 Hardness of Generalised Parameterised Matching

In this section we present an inapproximability result for a maximum similarity variant of GPM. In this variant, for input text t and pattern p, we are interested in the maximum similarity between p and any string p' of the same length which has a GPM match with t.

We now show that there exists an $\epsilon > 0$ such that the problem of generalised parameterised matching does not have a $(1-\epsilon)$-approximation algorithm, unless $\mathbf{P} = \mathbf{NP}$, via another gap preserving reduction from MAX3-SAT(13).

Consider a MAX-3SAT(13) instance ϕ with n variables and m clauses. Fix an ordering of the variables and for a variable x define $L(x)$ to be the position of this variable according to this ordering. Fix also an ordering of the clauses and for a clause c we define $L(c)$ to be the position of the clause according to this ordering (we overload the notation from the variables and it should be clear from the context to which one we are referring to).

The text alphabet Σ_t has $n+m+1$ symbols, \$ and 1, 2, ..., $n+m$, with \$ serving as a delimiter in both the pattern and text. The pattern alphabet Σ_p includes the delimiter \$, two characters x, X for each variable x and two characters for each clause c, $w_{L(c)}$ and $w'_{L(c)}$. X represents the negation of the variable x. The constructed pattern and text contain an equal number of \$ characters which forces \$ to map to \$ under any valid injective function. Also, as we show later, replacing some of the \$ symbols with wildcards cannot yield to an optimal strategy.

The construction starts by placing $m+n$ \$ symbols to the beginning of both the pattern and the text. For a variable x, we add to the text the string $\$^{13}L(x)L(x)L(x)\$^{13}$ 13 times and to the pattern $\$^{13}xX\$^{13}$ 13 times. In this way the variable x can be mapped either to $L(x)$ or $L(x)L(x)$. Replacing all the 13 variables with wildcards is not optimal since a variable appears in at most 13 clauses and, therefore, at most 13 clauses can be satisfied. To fix notation we say that for a variable x, $L(x)$ represents True and $L(x)L(x)$ represents False.

For the clauses we have the following construction. Consider that a clause c has literals x, y, z. Then we add to the text the string $\$^{13}L(x)L(x)n+L(c)L(y)L(y)n+L(c)L(z)L(z)n+L(c)\$^{13}$ and to the pattern the string $\$^{13}xw_{L(c)}yw_{L(c)}zw_{L(c)}\$^{13}$ where x, y, z are the variables from the clause, or their negations, as appropriate.

In the end we add for each clause the string $\$^{13}n+L(c)\$^{13}$ 13 times to the text and the string $\$^{13}w'_{L(c)}\$^{13}$ 13 times to the pattern. In this way the character $w'_{L(c)}$ is forced to match with $n+L(c)$ and no other characters can match $L(c)$. Also, if we want to make other characters to match $L(c)$ by replacing $w'_{L(c)}$ with wildcards, then the cost is too high (at least 13).

Example 11. Consider the following instance of MAX3-SAT(13),

$$(x \vee y \vee z) \wedge (\bar{x} \vee y \vee \bar{z}) \ .$$

The GPM input is the following:

$$t = \$\$\$\$\overbrace{\$^{13}111\$^{13}\ldots\$^{13}111\$^{13}}^{13}\overbrace{\$^{13}222\$^{13}\ldots\$^{13}222\$^{13}}^{13}$$

$$\overbrace{\$^{13}333\$^{13}\ldots\$^{13}333\$^{13}}^{13}\$^{13}114224334\ \$^{13}115225335\$^{13}\overbrace{\$^{13}4\$^{13}}^{13}\overbrace{\$^{13}5\$^{13}}^{13}$$

$$p = \$\$\$\$\$\overbrace{\$^{13}xX\$^{13}\ldots\$^{13}xX\$^{13}}^{13}\overbrace{\$^{13}yY\$^{13}\ldots\$^{13}yY\$^{13}}^{13}$$

$$\overbrace{\$^{13}zZ\$^{13}\ldots\$^{13}zZ\$^{13}}^{13}\$^{13}xw_1yw_1zw_1\$^{13}\$^{13}Xw_2yw_2Zw_2\$^{13}$$

$$\overbrace{\$^{13}w_1'\$^{13}}^{13}\overbrace{\$^{13}w_2'\$^{13}}^{13}$$

The inapproximability result for the GPM problem is stated in Theorem 12.

Theorem 12. *There exists an $\epsilon > 0$ such that the GPM problem does not have a $(1-\epsilon)$-approximation algorithm, unless* $\mathbf{P} = \mathbf{NP}$.

Proof. We have to prove that this is a gap-preserving reduction from MAX-3SAT(13). Specifically, we prove that there exists a constant $\sigma > 0$ such that conditions (1) and (2) are satisfied, where $OPT(\phi)$ is the optimal value of the MAX-3SAT(13) problem on the instance ϕ, $OPT(t,p)$ is the maximum similarity on the text t, and the pattern p constructed using the above reduction, and $f(|t|,|p|) = (n+m) + 26 \cdot 13n + 26m + 26 \cdot 13m + 2 \cdot 13n + 13m + 4m$. These implications, together with Remark 3 prove the theorem.

Now, we explain why the two implications are true. We first prove that if ϕ is satisfiable, then there must be a GPM solution for p and t where exactly $f(|t|,|p|)$ characters are not replaced with wildcards. We describe such a matching. The \$ symbol from the pattern matches the \$ symbol from the text and $w'_{L(c)}$ matches $L(c)$ for any clause c. If a variable x is assigned to True, then the character x matches $L(x)$ and X matches $L(x)L(x)$. Otherwise, if a variable x is assigned to False, then the character x matches $L(x)L(x)$ and X matches $L(x)$. Then, in every satisfied clause c, the character $w_{L(c)}$ after the True literal (we choose one literal arbitrarily if there are more than one True) is left unchanged an the other two are replaced with an wildcard. Since all the clauses are satisfied, in a clause gadget four characters are not replaced with a wildcard.

Now, suppose that at most $(1-\epsilon)m$ clauses of ϕ can be satisfied. Then, we must have at least $3\epsilon m$ wildcards in order to have a valid matching, three wildcards for each unsatisfied clause. These wildcards replace all three $w_L(c)$ characters in an unsatisfied clause c.

If we replace all the dollar signs from the beginning with a wildcard the number of wildcards used is $n+m$, which is not optimal. If we replace with wildcards a block of 13 \$'s which separates the variable gadgets, then you can satisfy at most 13 new clauses, at a price of 13 wildcard symbols, which is not an improvement to the strategy presented in the previous paragraph. If we replace with wildcards a block of 13 \$'s which separates the clause gadgets, then you can satisfy at most 2 new clauses. If we replace the characters w' with wildcards, then we have to use 13 wildcards to satisfy one clause and again the cost is too high. The last option is to try to allow a variable to have an inconsistent assignment in order to satisfy more clauses, but to do this we need to place at least 13 wildcards. Since a variable appears in at most 13 clauses, this, again, is not optimal.

Therefore, if at most $(1-\epsilon)m$ clauses of ϕ can be satisfied, then the maximum similarity has to be less or equal to $(n+m)+26\cdot 13n+26m+26\cdot 13m+2\cdot 13n+13m+(1-\epsilon)4m+3\epsilon m$. We want:

$$(n+m)+26\cdot 13n+26m+26\cdot 13m+2\cdot 13n+13m+(1-\epsilon)4m+3\epsilon m = (1-\sigma)((n+m)+26\cdot 13n+26m+26\cdot 13m+2\cdot 13n+13m+4m)$$

As before, we can set $m=13n$ and, therefore:

$$(1-\sigma)=\frac{5331-13\epsilon}{5331}$$

Therefore, the GPM problem cannot be approximated within a factor of 0.999983, unless $\mathbf{P}=\mathbf{NP}$. □

5 A $\sqrt{1/OPT}$-approximation algorithm for Generalised Parameterised Matching

In this section we present a $\sqrt{1/OPT}$-approximation algorithm for the maximum similarity for GPM in a special case where the text is at least twice as long as the pattern, where OPT is the maximum similarity between the pattern and any string p' of the same length which has a GPM match with the text.

In [6] Clifford et. al. present a $\sqrt{k/OPT}$-approximation algorithm for the maximum similarity for GFM, for any fixed k. Unlike GFM, for the maximum similarity for GPM, no approximation algorithms are known.

Informally, the $\sqrt{1/OPT}$-approximation algorithm works as follows. We define the length of the text t to be n and the length of the pattern to be m. We divide the input instances in two cases, which are treated separately: if the pattern alphabet $|\Sigma_p| \geq \sqrt{m}$ or if it is less than $\sqrt{m}$. The entire procedure is described by Algorithm 1.

Algorithm 1 A $\sqrt{1/OPT}$ approximation for maximum GPM similarity

Input: A pattern $p=p_1p_2\ldots p_m$ over the alphabet Σ_p and a text $t=t_1t_2\ldots t_n$ over the alphabet Σ_t.

1. If $|\Sigma_p| \geq \sqrt{m}$, then choose a set S of $\sqrt{m}$ distinct characters from the pattern p and find a generalised matching using Algorithm 2 and *output* it.
2. If $|\Sigma_p| < \sqrt{m}$, then:
   (a) iterate over each character $c \in \Sigma_p$ and each substring s of t;
      i. create a new pattern $p^{c,s}$ from p by replacing every character different from c with a wildcard;
      ii. compute the maximum similarity between $p^{c,s}$ and t using Algorithm 3.
   (b) *output* the pattern $p^{c,s}$ which has the highest maximum GPM similarity with t.

In the first case, select a set S of size $\sqrt{m}$ distinct characters from p, which also contains p_m, the last character of the pattern, and process p from left to right. We construct a new pattern that has similarity at least $\sqrt{m}$ to p. If p_i is not in S, then just change it to the character in t to which it is currently aligned. If it is in S, then leave it unchanged but skip $2,3,4,\ldots$ places in the text. The mapping is from characters to themselves for those positions that are not in S and from characters to substrings of length $2,3,4,\ldots$. for those in S. Algorithm 2 presents this process.

Algorithm 2 Computes a generalised parameterised matching of a pattern p, which has at least $\sqrt{m}$ distinct characters, with the text t

Input: A pattern $p = p_1 p_2 \dots p_m$ over the alphabet Σ_p, a set S of exactly $\sqrt{m}$ distinct characters from p and a text $t = t_1 t_2 \dots t_n$ over the alphabet Σ_t.

1. $j := 1$; $k := 2$
2. for $i = 1$ to m do:
   (a) If $i = m$ match p_m with $t_j t_{j+1} \dots t_n$. *return*;
   (b) If $p_i \notin S$, then change it to character t_j and match it with t_j. Set $j := j + 1$;
   (c) If $p_i \in S$, then match it with the string $t_j t_{j+1} \dots t_{j+k-1}$. Set $j := j + k$; $k := k + 1$

Output: The matching between p and t.

In the latter case we fix one character in turn and we replace everything else by a wildcard. Then, we solve the problem independently for each substring s of the text t and then we choose the substring for which the similarity is the highest. The similarity is computed using dynamic programming (Algorithm 3). We define the function $f(i, j)$ to be the best solution to the problem for $t_1 t_2 \dots t_j$ and $p_1 p_2 \dots p_i$. When the algorithm finishes, the maximum similarity is stored in $f(m, n)$.

We finally output the pattern $p_{c,s}$ with the maximum similarity over all characters and all substrings.

Algorithm 3 Computes maximum GPM similarity of a pattern $p_{c,s}$ with the text t

Input: A pattern $p^{c,s} = p_1^{c,s} p_2^{c,s} \dots p_m^{c,s}$ over the alphabet Σ_p with exactly one non-wildcard character and a text $t = t_1 t_2 \dots t_n$ over the alphabet Σ_t.

$$f(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } i > j \\ \max\{f(i-1,j-1), f(i-1,j-2), \dots, f(i-1,i-1)\}, & \text{if } p_i^{c,s} \neq c \\ \max_k\{f(i-1,j-k) + I(t_{j-k+1} \dots t_j = s)\}, & \text{if } p_i^{c,s} = c \end{cases}$$

Output: $f(m, n)$ - the maximum GPM similarity between $p_{c,s}$ and t.

In order to prove the correctness of Algorithm 1 we need the following lemmas.

Lemma 13. *If $|t| \geq 2|p|$ Algorithm 2 computes a GPM match between p and t.*

Proof. The number of characters in S is $\sqrt{m}$. Each character which is not in S matches only one character of the text and therefore the number of characters from the text used is $m - \sqrt{m}$. Characters from S use $2 + 3 + \dots + \sqrt{m} + 1$ characters of the text. Therefore, the total number of characters of the text used in the matching is $(\sqrt{m}+1)(\sqrt{m}+2)/2-1+m-\sqrt{m}$. Since $n \geq 2m$, $(\sqrt{m}+1)(\sqrt{m}+2)/2-1+m-\sqrt{m} \leq n$. The matching is valid (i.e. all the characters are mapped injectively) since the strings that are mapped to characters from S have different length. □

Lemma 14. *Algorithm 3 computes the GPM similarity between $p_c^{c,s}$ and t.*

Proof. We must first show that the dynamic programming procedure computes the right function and then that it runs in polynomial time. We can see immediately that $f(0, i) = 0\ \forall i$ because in this case the pattern is empty. Also, $f(i, j) = 0\ \forall i > j$ because every character of the pattern must map at least one character from the text, even if it is replaced by a wildcard. The computation of $f(i, j)$ has two cases.

- $p_i^{c,s} \neq c$. In this case we cannot increase the number of characters in our set that can be mapped. However we know that $p_i^{c,s}$ is set to a wildcard and therefore we find the maximum of the previous results for different length substrings that the wildcard maps to.

– $p_i^{c,s} = c$. We can either map $p_i^{c,s}$ to s and increase the number of mapped characters by one, which can only happen if $t_{j-|s_z|+1} \dots t_j = s$ or we do the same as in the previous case. □

We are now prepared to prove the main result of this section.

Theorem 15. *Algorithm 1 is a $\sqrt{1/OPT}$-approximation algorithm if the length of t is at least twice the length of p.*

Proof. Let M be the maximum GPM-similarity over all $p^{c,s}$. We know that $OPT \leq M \cdot |\Sigma_p|$ since M is the maximum over all characters of p. Therefore, either M or $|\Sigma_p|$ have to be greater than or equal to $\sqrt{OPT}$. The total running time of the algorithm is polynomial in n and m. □

Acknowledgments. The second author is funded by an EPSRC PhD studentship.

References

1. K. R. Abrahamson: *Generalized string matching.* SIAM journal on Computing, 16(6) 1987, pp. 1039–1051.
2. A. Amir, A. Aumann, R. Cole, M. Lewenstein, and E. Porat: *Function matching: Algorithms, applications, and a lower bound*, in Proceedings of the 30*th* International Colloquium on Automata, Languages and Programming (ICALP), 2003, pp. 929–942.
3. A. Amir and I. Nor: *Generalized function matching*, in Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC), 2004, pp. 41–52.
4. S. Arora: *Probabilistic checking of proofs and the hardness of approximation problems*, PhD thesis, UC Berkeley, 1994.
5. B. S. Baker: *A theory of parameterized pattern matching: algorithms and applications*, in Proceedings of the 25*th* Annual ACM Symposium on the Theory of Computing (STOC), 1993, pp. 71–80.
6. R. Clifford, A. W. Harrow, A. Popa, and B. Sach: *Generalised matching*, in Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE), 2009, pp. 295–301.
7. J. Håstad: *Some optimal inapproximability results.* J. ACM, 48(4) 2001, pp. 798–859.
8. O. Keller, T. Kopelowitz, and M. Lewenstein: *On the longest common parameterized subsequence.* Theoretical Computer Science, 410(51) 2009, pp. 5347–5353.
9. S. R. Kosaraju: *Efficient string matching.* Manuscript, 1987.
10. V. Levenshtein: *Binary codes capable of correcting spurious insertions and deletions of ones.* Problems of Information Transmission, 1 1965, pp. 8–17.
11. V. Levenshtein: *Binary codes capable of correcting insertions and reversals.* Soviet Physics Doklady, 10(8) 1966, pp. 707–710.
12. V. V. Vazirani: *Approximation Algorithms*, Springer, 2004.

A Space-Efficient Implementation of the Good-Suffix Heuristic

Domenico Cantone, Salvatore Cristofaro, and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | cristofaro | faro}@dmi.unict.it

Abstract. We present an efficient variation of the *good-suffix* heuristic, firstly introduced in the well-known Boyer-Moore algorithm for the exact string matching problem. Our proposed variant uses only constant space, retaining much the same time efficiency of the original rule, as shown by extensive experimentation.

Keywords: string matching, experimental algorithms, text-processing, good-suffix rule, costant-space algorithms

1 Introduction

Given a text T and a pattern P (of length m) over some alphabet Σ, the *string matching problem* consists in finding *all* occurrences of the pattern P in the text T. It is a very extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

The most practical string matching algorithms show a sublinear behavior in practice, at the price of using extra memory of non-constant size to maintain auxiliary information. For instance, the Boyer-Moore algorithm [2] requires additional $\mathcal{O}(m+|\Sigma|)$-memory to compute two tables of shifts, which implement the well-known good-suffix and bad-character heuristics. Other efficient variants of the Boyer-Moore algorithm use additional $\mathcal{O}(m)$-space [7], or $\mathcal{O}(|\Sigma|)$-space [14,18], whereas, interestingly enough, two of the fastest algorithms require respectively $\mathcal{O}(|\Sigma|^2)$-space [1] and $\mathcal{O}(m \cdot |\Sigma|)$-space [5].

The first non-trivial constant-space string matching algorithm is due to Galil and Seiferas [10]. Their algorithm, though linear in the worst-case, was too complicated to be of any practical interest. Slightly more efficient constant-space algorithms have been subsequently reported in the literature (see [3,8,9,11,12]), and more recently two new constant-space algorithms have been presented, which have a sublinear average behavior though they are quadratic in the worst-case [6]. It is to be pointed out, though, that no constant-space algorithm which is competitive with the most efficient variants of the Boyer-Moore algorithm is known as yet.

Starting from the observation that most of the accesses to the good-suffix table are limited to very few locations, in this paper we propose a truncated good-suffix heuristic which require only constant-space and show by extensive experimentation that the Boyer-Moore algorithm and two of its more effective variants maintain much the same running times, when the truncated variant is used in place of the classical one.

Domenico Cantone, Salvatore Cristofaro, Simone Faro: A Space-Efficient Implementation of the Good-Suffix Heuristic, pp. 63–75 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

The paper is organized as follows. In Section 2 we give some preliminary notions. Then, in Section 3 we describe the preprocessing techniques introduced in the Boyer-Moore algorithm together with some efficient variants which make use of the same shift heuristics. In Section 4 we estimate the probability that a given entry of a good-suffix table is accessed and, based on such analysis, we come up with the proposal to memorize only a constant number of entries. We also show how such entries can be computed in constant-space. Subsequently, in Section 5 we present the experimental data obtained by running under various conditions the algorithms reviewed, and their modified versions. Such results confirm experimentally that by truncating the good-suffix table much the same running times are maintained. Finally, our conclusions are given in Section 6.

2 Preliminaries

Before entering into details, we need a bit of notations and terminology. A string P is represented as a finite array $P[0\,..\,m-1]$, with $m \geq 0$. In such a case we say that P has length m and write $\text{length}(P) = m$. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < \text{length}(P)$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, where $0 \leq i \leq j < \text{length}(P)$. Moreover, for any $i, j \in \mathbb{Z}$, we put $P[i\,..\,j] = \varepsilon$ if $i > j$, and $P[i\,..\,j] = P[\max(i,0)\,..\,\min(j, \text{length}(P)-1)]$ otherwise.

For any two strings P and P', we write $P' \sqsupset P$ to indicate that P' is a suffix of P, i.e., $P' = P[i\,..\,\text{length}(P)-1]$, for some $0 \leq i < \text{length}(P)$. Similarly, we write $P' \sqsubset P$ to indicate that P' is a prefix of P, i.e., $P' = P[0\,..\,i-1]$, for some $0 \leq i \leq \text{length}(P)$. In addition, we denote by P^R the reverse of the string P.

Let T be a text of length n and let P be a pattern of length m. If the character $P[0]$ is aligned with the character $T[s]$ of the text, so that $P[i]$ is aligned with $T[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern P has *shift* s in T. In this case the substring $T[s\,..\,s+m-1]$ is called the *current window* of the text. If $T[s\,..\,s+m-1] = P$, we say that the shift s is *valid*.

Most string matching algorithms have the following general structure:

```
Generic_String_Matcher(T, P)
1.  Precompute_Globals(P)
2.  n := length(T)
3.  m := length(P)
4.  s := 0
5.  while s ≤ n − m do
6.      j := Check_Shift(s, P, T)
7.      s := s + Shift_Increment(s, P, T, j)
```

where

- the procedure *Precompute_Globals*(P) computes useful mappings, in the form of tables, which later may be accessed by the function *Shift_Increment*(s, P, T);
- the function *Check_Shift*(s, P, T) checks whether s is a valid shift and returns the position j of the last matched character in the pattern;
- the function *Shift_Increment*(s, P, T, j) computes a *positive* shift increment according to the information tabulated by procedure *Precompute_Globals*(P) and to the position j of the last matched character in the pattern.

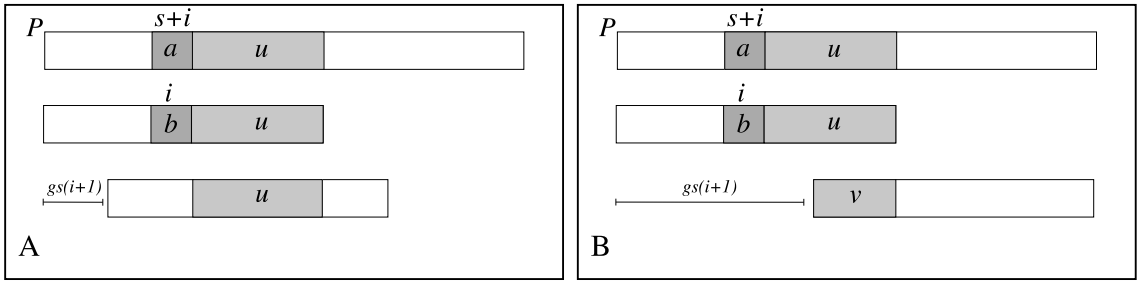


Figure 1. The good-suffix heuristic. Assuming that the suffix $u = P[i+1\,..\,m-1]$ of the pattern P has a match on the text T at shift s and that $P[i] \neq T[s+i]$, then the good-suffix heuristic attempts to align the substring $T[s+i+1\,..\,s+m-1] = P[i+1\,..\,m-1]$ with its rightmost occurrence in P preceded by a character different from $P[i]$ (see **(A)**). If this is not possible, the good-suffix heuristic suggests a shift increment corresponding to the match between the longest suffix of u with a prefix, v, of P (see **(B)**).

Observe that for the correctness of procedure *Generic_String_Matcher*, it is plainly necessary that the shift increment Δs computed by *Shift_Increment*(s, P, T) be *safe*, namely no valid shift may belong to the interval $\{s+1, \ldots, s+\Delta s-1\}$.

In the case of the naive string matching algorithm, for instance, the procedure *Precompute_Globals* is just dropped, procedure *Check_Shift*(s, P, T) checks whether the current shift is valid by scanning the pattern from left to right, and the function *Shift_Increment*(s, P, T, j) always returns a unitary shift increment.

3 The good-suffix heuristic for preprocessing

Information gathered during the execution of the *Shift_Increment*(s, P, T, j) function, in combination with the knowledge of P, as suitably extracted by procedure *Precompute_Globals*(P), can yield shift increments larger than 1 and ultimately lead to more efficient algorithms. In this section we focus our attention to the use of the good-suffix heuristic for preprocessing the pattern, introduced by Boyer and Moore in their celebrated algorithm [2].

The Boyer-Moore algorithm is the progenitor of several algorithmic variants which aim at computing close to optimal shift increments very efficiently. Specifically, the Boyer-Moore algorithm checks whether s is a valid shift, by scanning the pattern P from right to left and, at the end of the matching phase, it calls procedure *Boyer-Moore_Shift_Increment*(s, P, T, j) to compute the shift increment, where j is the position of last matched character in the pattern. Such procedure computes the shift increment as the maximum value suggested by the *good-suffix heuristic* and the *bad-character heuristic* below, using the functions gs_P and bc_P respectively, provided that both of them are applicable.

Boyer-Moore_Shift_Increment(s, P, T, j)
1. **if** $j > 0$ **then**
2. **return** $\max(gs_P(j), j - bc_P(T[s+j-1]) - 1)$
3. **return** $gs_P(0)$

Let us briefly review the shifting strategy of the good-suffix and the bad-character heuristics.

If the last matching character occurs at position j of the pattern P, the good-suffix heuristic suggests to align the substring $T[s+j\,..\,s+m-1] = P[j\,..\,m-1]$ with its rightmost occurrence in P (preceded by a character different from $P[j-1]$, provided that $j > 0$); this case is illustrated in Fig. 1A. If such an occurrence does not exist, the good-suffix heuristic suggests a shift increment which allows to match the longest suffix of $T[s+j\,..\,s+m-1]$ with a prefix of P; see Fig. 1B.

More formally, if the last matching character occurs at position j of the pattern P, the good-suffix heuristic states that the shift can be safely incremented by $gs_P(j)$ positions, where

$$gs_P(i) =_{\text{Def}} \min\{0 < k \leq m \mid P[i-k\,..\,m-k-1] \sqsupset P \\ \text{and } (k \leq i-1 \rightarrow P[i-1] \neq P[i-1-k])\} \ ,$$

for $i = 0, 1, \ldots, m$.

The bad-character heuristic states that if $c = T[s+j-1] \neq P[j-1]$ is the first mismatching character, while scanning P and T from right to left with shift s, then P can be safely shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s+j-1)$ in T. In the case in which c does not occur in P, then P can safely be shifted just past position $(s+j-1)$ in T. More formally, the shift increment suggested by the bad-character heuristic is given by the expression $(j - bc_P(T[s+j-1]) - 1)$, where

$$bc_P(c) =_{\text{Def}} \max(\{0 \leq k < m \mid P[k] = c\} \cup \{-1\}) \ ,$$

for $c \in \Sigma$, and where we recall that Σ is the alphabet of the pattern P and text T. Notice that in some situations the shift increment proposed by the bad-character heuristic may be negative.

It turns out that the functions gs_P and bc_P can be computed during the preprocessing phase in time $\mathcal{O}(m)$ and $\mathcal{O}(m + |\Sigma|)$ and space $\mathcal{O}(m)$ and $\mathcal{O}(|\Sigma|)$, respectively, and that the overall worst-case running time of the Boyer-Moore algorithm, as described above, is linear (cf. [13]).

Due to the simplicity and ease of implementation of the bad-character heuristic, some variants of the Boyer-Moore algorithm have focused just around it and dropped the good-suffix heuristic. This is the case, for instance, of the Horspool algorithm [14], which computes shift advancements by aligning the rightmost character $T[s+m-1]$ with its rightmost occurrence on $P[0\,..\,m-2]$, if present; otherwise it shifts the pattern just past the current window.

Similarly the Quick-Search algorithm [18] uses a modification of the original heuristics of the Boyer-Moore algorithm, much along the same lines of the Horspool algorithm. Specifically, it is based on the observation that the character $T[s+m]$ is always involved in testing for the next alignment, so that one can apply the bad-character heuristic to $T[s+m]$, rather than to the mismatching character, obtaining larger shift advancements.

A further example is given by the Berry-Ravindran algorithm [1], which extends the Quick-Search algorithm by using in the bad-character heuristic also the character $T[s+m+1]$ in addition to $T[s+m]$. In this case, the table used by the bad-character heuristic requires $\mathcal{O}(|\Sigma|^2)$-space and $\mathcal{O}(m + |\Sigma|^2)$-time complexity.

Experimental results show that the Berry-Ravindran algorithm is fast in practice and performs a low number of text/pattern character comparisons and that the Quick-Search algorithm is very fast especially for short patterns (cf. [16]).

The role of the good-suffix heuristic in practical string matching algorithms has recently been reappraised, also in consideration of the fact that often it is as effective as the bad-character heuristic, especially in the case of non-periodic patterns.

This is the case of the Fast-Search algorithm [4], a very simple, yet efficient, variant of the Boyer-Moore algorithm. The Fast-Search algorithm computes its shift increments by applying the bad-character heuristic if and only if a mismatch occurs during the first character comparison, namely, while comparing characters $P[m-1]$ and $T[s+m-1]$, where s is the current shift. In all other cases it uses the good-suffix heuristic. This translates in the following pseudo-code:

Fast-Search_Shift_Increment(s, P, T, j)
1. $m := \text{length}(P)$
2. **if** $j = m - 1$ **then**
3. **return** $bc_P(T[s+m-1])$
4. **else**
5. **return** $gs_P(j)$

A more effective implementation of the Fast-Search algorithm is obtained by iterating the bad-character heuristic until the last character $P[m-1]$ of the pattern is matched correctly against the text, at which point it is known that $T[s+m-1] = P[m-1]$, so that the subsequent matching phase can start with the $(m-2)$-*nd* character of the pattern. At the end of the matching phase the good-suffix heuristic is applied to compute the shift increment.

Another example is the Forward-Fast-Search algorithm [5], which maintains the same structure of the Fast-Search algorithm, but is based upon a modified version of the good-suffix heuristic, called *forward good-suffix* heuristic, which uses a look-ahead character to determine larger shift advancements. More precisely, if the last matching character occurs at position $j \leq m-1$ of the pattern P, the forward good-suffix heuristic suggests to align the substring $T[s+j \,..\, s+m-1]$ with its rightmost occurrence in P preceded by a character different from $P[j-1]$. If such an occurrence does not exist, the forward good-suffix heuristic proposes a shift increment which allows to match the longest suffix of $T[s+j \,..\, s+m-1]$ with a prefix of P. This corresponds to advance the shift s by $\overrightarrow{gs}_P(j, T[s+m])$ positions, where

$$\overrightarrow{gs}_P(i,c) =_{\text{Def}} \min(\{0 < k \leq m \mid P[i-k \,..\, m-k-1] \sqsupset P \text{ and } (k \leq i-1 \rightarrow P[i-1] \neq P[i-1-k]) \text{ and } P[m-k] = c\} \cup \{m+1\}),$$

for $i = 0, 1, \ldots, m$ and $c \in \Sigma$.

The forward good-suffix heuristic requires a table of size $m \cdot |\Sigma|$ which can be constructed in time $\mathcal{O}(m \cdot \max(m, |\Sigma|))$.

Experimental results show that both the Fast-Search and the Forward-Fast-Search algorithms, though not linear, achieve very good results especially in the case of very short patterns or small alphabets.

4 Truncating the Good-Suffix Tables

Let us assume that we run the Boyer-Moore algorithm on a pattern P and a text T. Then, at the end of each matching phase, the Boyer-Moore algorithm accesses the entry at position $j > 0$ in the good-suffix table if and only if the last matched

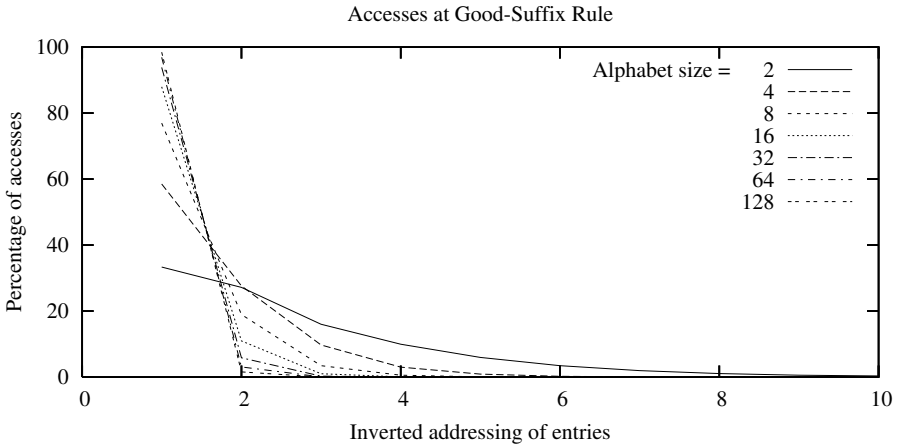


Figure 2. The percentage of accesses for each entry of the good-suffix heuristic, for different sizes of the alphabet. The values have been computed by running the Fast-Search algorithm with a set of 200 pattern, of length 40, and a 20Mb text buffer as input. The values in each curve f are relative to the inverted addressing of the entries, i.e. $f(j)$ is the percentage of accesses to the entry at position $m - j$.

character in the pattern occurs at position j of the pattern, i.e. if $P[j\,..\,m-1] = T[s+j\,..\,s+m-1]$ and $P[j-1] \neq T[s+j-1]$, where s is the current shift. Likewise, the Boyer-Moore algorithm accesses the entry at position $j = 0$ if and only if $P[0\,..\,m-1] = T[s\,..\,s+m-1]$, i.e. if and only if s is a valid shift.

Therefore, it is intuitively expected that the probability to access an entry at position j of the good-suffix table becomes higher as the value of j increases. In other words, it is expected that entries on the right-hand side of the good-suffix table have (much) higher probability to be accessed than entries on the left end side.

The above considerations, which will be formalized below under suitable simplifying hypotheses, suggest that the initial segment of the good-suffix tables can be dropped, without affecting very much the performance of the algorithm. In fact, we will see that in most cases, it is enough to maintain just a few entries of the good-suffix tables.

For the sake of simplicity, in the following analysis we will assume that the text T and pattern P are strings over a common alphabet Σ of size σ, randomly selected relatively to a uniform distribution.

Thus, for a shift $0 \leq s \leq n - m$ in T and a position $0 \leq j < m$ in P, the probability that $P[j] = T[s+j]$ is $1/\sigma$, whereas the probability that $P[j] \neq T[s+j]$ is $(\sigma - 1)/\sigma$.

Therefore, the probability p_j that j is the position of the last matched character in the pattern P, relatively to a shift s of the text, is given by

$$p_j = \begin{cases} \dfrac{\sigma - 1}{\sigma^{m-j+1}} & \text{if } 0 < j \leq m \\ \dfrac{1}{\sigma^m} & \text{if } j = 0\,. \end{cases}$$

Plainly, p_j is also the probability that location j of the good-suffix table is accessed.

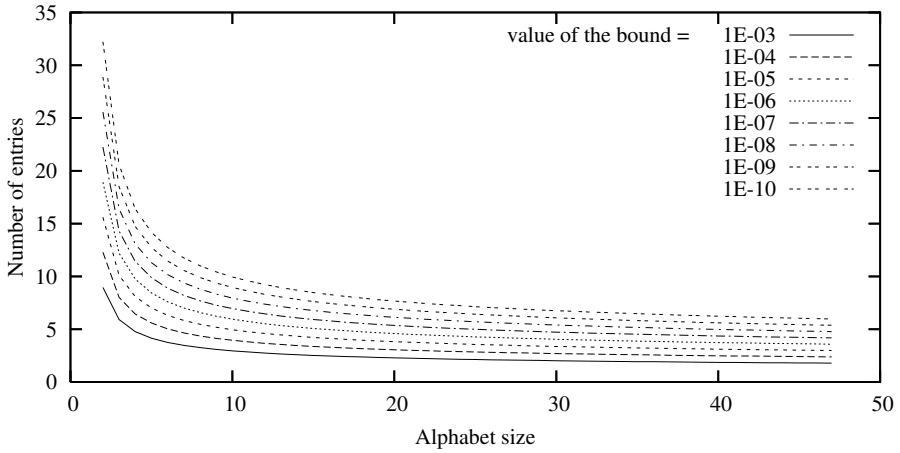


Figure 3. The function $\log_\sigma \frac{\sigma-1}{\beta} - 1$, for different values of the bound β. Note that if the bound β is greater or equal to 10^{-4}, then the number of entries accessed with probability greater than β is always no greater than 12 and in most cases no greater than 3.

As experimental evidence of the above analysis, we report in Fig. 2 the plots of the accesses to each entry of the good-suffix table, for different sizes of the alphabet, when running the Fast-Search algorithm with a set of 200 patterns of length 40 and a 20Mb text buffer as input. More precisely, for each function f in Fig. 2, $f(j)$ is the percentage of accesses to the entry at position $m-j$ in the good-suffix table. We can observe that, in general, only a very small number of entries is really used during a computation and, in particular, when the alphabet size is greater than or equal to 16 about 98% of the accesses are limited to the last three entries of the table.

We can readily evaluate the number $K_{\sigma,\beta}$ of entries of the good-suffix table which are accessed with probability greater than a fixed threshold $0 < \beta < 1$, for an alphabet of size σ. To begin with, notice that if $p_j > \beta$, then $\frac{\sigma-1}{\sigma^{m-j+1}} > \beta$, so that

$$j > m + 1 - \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil . \quad \text{and} \quad m_{\sigma,\beta} \leq \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1 .$$

Observe that for $\bar{\beta} = 10^{-4}$, we have $K_{\sigma,\bar{\beta}} \leq 12$. Additionally, we have $K_{\sigma,\bar{\beta}} \leq 3$, for $14 \leq \sigma \leq 39$, and $K_{\sigma,\bar{\beta}} \leq 2$, for $\sigma \geq 40$. In other words, for alphabets of at least 14 characters, at most the last three entries of the good-suffix table are accessed with probability at least 10^{-4} (under the assumption of uniform distribution). Fig. 3 shows the shape of the function $\log_\sigma \frac{\sigma-1}{\beta} - 1$ for the following values of the bound $\beta = 10^{-3}, 10^{-4}, \ldots, 10^{-10}$. Note that if the bound β is greater or equal to 10^{-4}, then the number of entries accessed with probability greater than β is always no greater than 12 and in most cases no greater than 3.

4.1 The Bounded-Good-Suffix Heuristic

The above considerations justify the following *bounded good-suffix* heuristic. Let $\beta > 0$ be a fixed bound[1] and let $K = \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1$, where, as usual, σ denotes the size of the alphabet. Then the bounded good-suffix heuristic works as follows.

[1] A good practical choice is $\beta = 10^{-4}$, as shown in Section 5.

During a matching phase, if the first mismatch occurs at position i of the pattern P and $i \geq m - K$, the bounded good-suffix heuristic suggests that the pattern is shifted $gs_P(i+1)$ positions to the right. Otherwise, if the first mismatch occurs at position i of the pattern P, with $i < m - K$, or if the pattern P matches the current window in the text, then the bounded good-suffix heuristic suggests that the pattern is shifted one position to the right.

More formally, if the first mismatch occurs at position i of the pattern P, the bounded good-suffix heuristic suggests that the shift s can be safely advanced $\beta gs_P(i-m+K)$ positions to the right, where, for $j = K - m - 1, \ldots, K - 1$, we have

$$\beta gs_P(j) =_{\text{Def}} \begin{cases} gs_P(j+m-K+1) & \text{if } j \geq 0 \\ 1 & \text{otherwise}. \end{cases}$$

Likewise, the *bounded forward good-suffix heuristic* suggests that when the first mismatch occurs at position i of the pattern P, then the shift s is advanced by $\overrightarrow{\beta gs}_P(i-m+k, T[s+m])$ positions to the right, where, for $j = K-m-1, \ldots, K-1$ and $c \in \Sigma$, we have

$$\overrightarrow{\beta gs}_P(j, c) =_{\text{Def}} \begin{cases} \overrightarrow{gs}_P(j+m-K+1, c) & \text{if } j \geq 0 \\ 1 & \text{otherwise}. \end{cases}$$

By way of example, when the bounded good-suffix heuristic is adopted in place of the good-suffix heuristic, the *Shift_Increment* procedure of the Boyer-Moore algorithm becomes:

βBoyer-Moore_Shift_Increment$(s, P, T, j, \sigma, \beta)$
1. $m := \text{length}(P)$
2. $K := \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1$
3. **if** $j \geq m - K - 1$ **then**
4. **if** $j > 0$ **then**
5. **return** $\max(\beta gs_P(j-m+K-1), j - bc_P(T[s+j-1]))$
6. **else return** $\beta gs_P(0)$
7. **else if** $j > 0$ **then**
8. **return** $\max(1, j - bc_P(T[s+j-1]))$
9. **else return** 1

Next we discuss how the bounded good-suffix function βgs_P can be constructed (analogous remarks apply for the bounded forward good-suffix function). A first very natural way to compute the function βgs_P consists in computing a slightly modified version of the standard good-suffix function gs_P, and then keeping only the last K entries of the function. However such procedure, based on the one firstly given in [2] and later corrected in [17], has $\mathcal{O}(m)$-time and space complexity.

An alternative way to compute the bounded good-suffix function using only constant space, but still in $\mathcal{O}(m)$ worst-case time, is given by procedure *Precompute_βgs*, whose pseudo-code is presented below:

Precompute_βgs(P, σ, β)
1. $m := \text{length}(P)$
2. $K := \left\lceil \log_\sigma \frac{\sigma-1}{\beta} \right\rceil - 1$
3. **for** $\ell := 0$ **to** $K-1$ **do**
4. $\quad j := m-2$
5. $\quad$ **repeat**
6. $\quad\quad q := j - occur(P^R_{m-\ell..m-1}, P^R_{0..j})$
7. $\quad\quad j := q-1$
8. $\quad$ **until** $q < \ell$ **or** $P[m-\ell-1] \neq P[q-\ell]$
9. $\quad \beta gs_P(K-\ell-1) := m-q-1$
10. **return** βgs_P

First of all, we give the specification of the function *occur*, which is called by procedure *Precompute_βgs*. Given two strings X and Y, $occur(X, Y)$ computes the leftmost occurrence of X in Y, i.e.,

$$occur(X, Y) =_{\text{Def}} \min\{p \geq 0 \mid Y[\,p\,..\,p + |X| - 1] = X\} \cup \{|Y|\}\,.$$

Observe that the function $occur(X, Y)$ can be computed by means of a linear-time string matching algorithm such as the Knuth-Morris-Pratt algorithm [15], thus requiring $\mathcal{O}(|X| + |Y|)$-time and $\mathcal{O}(|X|)$ additional space.

We are now ready to explain how the procedure *Precompute_βgs* works.

For $\ell = 0, 1, \ldots, K-1$, the ℓ-th iteration of the **for**-loop in line 3 finds the rightmost occurrence, $P[q-\ell+1\,..\,q]$, in P of its suffix of length ℓ preceded by a character different from $P[m-\ell-1]$. If such an occurrence does not exist, the ℓ-th iteration finds the rightmost position $q < \ell$ in the pattern such that $P[0\,..\,q] = P[m-q-1\,..\,m-1]$. More precisely, the search is performed within the **repeat**-loop in line 5, by means of repeated calls of type $occur((P[m-\ell\,..\,m-1])^R, (P[0\,..\,j])^R)$, each of which looks for the leftmost occurrence of the reverse of $P[m-\ell\,..\,m-1]$ in the reverse of $P[0\,..\,j]$. When such an occurrence is found at position q, so that $P[q-\ell+1\,..\,q]$ is a suffix of P, it is checked whether $q < \ell$ holds or whether the character $P[m-\ell-1]$ is different from $P[q-\ell]$. If any of such conditions is true, the **repeat**-loop stops, whereas if both conditions are false, another iteration is performed with $j = q-1$.

The value q, discovered during the ℓ-th iteration of the **for**-loop in line 3, is then used in line 9 to set the $(K-\ell-1)$-th entry of the βgs_P function to $m-q-1$.

Concerning the time and space analysis of the procedure *Precompute_βgs*, notice that each iteration of the **for**-loop, for $\ell = 0, 1, \ldots, K-1$, takes $\mathcal{O}(K+m)$-time, using only $\mathcal{O}(K)$-space. Indeed, each call $occur(P^R_{m-\ell..m-1}, P^R_{0..j})$ in the **repeat**-loop takes time proportional to $j-r$, where $r = occur(P^R_{m-\ell..m-1}, P^R_{0..j})$, and uses $\mathcal{O}(\ell)$ (reusable) space. Additionally, after each such call, the value of j is decreased by $r+1$. Hence, the overall running time of all calls to the function *occur* made in the **repeat**-loop is bounded by $\mathcal{O}(K+m)$, for each iteration of the **for**-loop.

Since the number of iterations in the **for**-loop is K, the overall running time of the procedure *Precompute_βgs* is $\mathcal{O}(K^2 + Km)$.

Notice that if we fix the value of $\beta = 10^{-4}$, then we have $K \leq 12$, as observed just before Section 4.1. Therefore, in such a case, the time and space complexity of the procedure *Precompute_βgs* are $\mathcal{O}(1)$ and $\mathcal{O}(m)$, respectively.[2]

[2] As will be shown in Section 5, the choice $\beta = 10^{-4}$ has very good practical results.

5 Experimental Results

To evaluate experimentally the impact of the bounded good-suffix heuristic, we have chosen to test it with the Boyer-Moore algorithm (in short, BM) and with two of its fastest variants in practice, namely the Fast-Search (FS) and the Forward-Fast-Search (FFS) algorithms. Their modified versions, obtained by using the bounded good-suffix heuristic in place of the good-suffix heuristic (in the case of the Boyer-Moore and the Fast-Search algorithms) and the bounded forward good-suffix heuristic in place of the forward good-suffix heuristic (in the case of the Forward-Fast-Search algorithm), are respectively denoted in short by βBM, βFS, and βFFS.

All algorithms have been implemented in the C programming language and were used to search for the same strings in large fixed text buffers on a PC with AMD Athlon processor of 1.19GHz. In particular, all algorithms have been tested on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, with patterns of length $m = 2, 4, 8, 10, 20, 40, 80$ and 160, and on two real data problems.

Each Randσ problem consists in searching a set of 200 random patterns of a given length in a 20Mb random text over a common alphabet of size σ.

The tests on the real data problems have been performed on a 180Kb natural language text file, containing the "Hamlet" by William Shakespeare (NL), and on a 2.4Mb file containing a protein sequence from the human genome. In both cases, the patterns to be searched for have been constructed by selecting 200 random substrings of length m from the files, for each $m = 2, 4, 8, 10, 20, 40, 80$ and 160.

For the implementation of the bounded versions of the (forward) good-suffix heuristic we have used the bound $\beta = 10^{-4}$.

With the exception of the last two tables in which running times are expressed in thousandths of seconds, all other running times in the remaining tables are expressed in hundredths of seconds.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
BM	46.82	39.77	31.51	25.89	21.23	19.93	19.56	17.56	15.64
βBM	47.34	40.33	31.94	26.10	21.64	20.24	19.82	17.95	15.93
FS	35.62	31.28	25.80	21.93	19.28	18.33	17.91	16.48	14.96
βFS	36.32	32.34	26.36	22.46	19.37	18.35	17.94	16.71	14.95
FFS	31.02	28.39	23.46	19.76	17.83	16.97	16.64	15.03	13.78
βFFS	37.27	32.44	25.83	21.71	19.30	18.53	18.18	16.33	14.95

Running times in hundredths of seconds for a Rand2 problem

$\sigma = 4$	2	4	6	8	10	20	40	80	160
BM	38.84	28.41	23.23	21.04	20.15	19.30	18.95	17.70	16.42
βBM	39.09	28.58	23.35	21.29	20.30	19.54	19.10	17.87	16.52
FS	26.08	21.15	18.95	18.14	17.64	17.07	16.70	15.91	14.84
βFS	26.56	21.49	19.17	18.30	17.65	17.12	16.72	16.02	14.93
FFS	25.14	20.58	18.58	17.34	16.52	16.11	15.90	14.32	13.29
βFFS	26.61	21.18	18.68	17.66	16.70	16.30	16.02	14.55	13.35

Running times in hundredths of seconds for a Rand4 problem

$\sigma = 8$	2	4	6	8	10	20	40	80	160
BM	33.24	23.16	18.97	17.86	17.36	17.12	17.00	16.42	15.83
βBM	33.01	23.09	19.14	17.93	17.36	17.14	17.09	16.47	15.91
FS	21.02	18.26	16.43	16.04	15.93	15.81	15.82	15.29	14.88
βFS	21.32	18.34	16.46	16.10	15.96	15.88	15.75	15.39	14.91
FFS	20.84	18.23	16.39	16.05	15.78	15.64	15.26	13.96	13.18
βFFS	21.12	18.36	16.43	16.05	15.82	15.67	15.31	14.00	13.02

Running times in hundredths of seconds for a Rand8 problem

$\sigma = 16$	2	4	6	8	10	20	40	80	160
BM	31.09	21.37	18.18	16.39	16.04	15.92	15.86	15.64	15.39
βBM	30.45	21.31	18.41	16.42	16.04	15.85	15.84	15.60	15.32
FS	19.14	16.77	15.94	15.66	15.40	15.32	15.28	15.12	14.91
βFS	19.29	16.89	16.05	15.61	15.45	15.39	15.25	15.09	14.92
FFS	19.19	16.84	15.90	15.65	15.44	15.35	15.11	13.98	13.26
βFFS	19.22	16.88	16.00	15.60	15.36	15.29	15.00	13.84	13.09

Running times in hundredths of seconds for a Rand16 problem

$\sigma = 32$	2	4	6	8	10	20	40	80	160
BM	29.96	20.38	17.49	16.03	15.78	15.47	15.25	15.15	15.02
βBM	29.44	19.97	17.63	16.12	15.79	15.47	15.19	15.11	15.03
FS	18.78	16.38	15.86	15.52	15.12	15.13	14.75	14.70	14.61
βFS	18.87	16.38	15.84	15.54	15.13	15.12	14.78	14.70	14.66
FFS	18.89	16.47	15.87	15.56	15.15	15.15	14.79	14.21	13.54
βFFS	18.84	16.32	15.89	15.52	15.12	15.10	14.65	14.05	13.35

Running times in hundredths of seconds for a Rand32 problem

$\sigma = 64$	2	4	6	8	10	20	40	80	160
BM	29.50	19.39	17.31	15.96	15.63	15.39	14.39	13.65	13.51
βBM	29.00	19.51	17.53	15.97	15.66	15.27	14.54	13.65	13.51
FS	18.60	16.24	15.75	15.61	14.96	15.06	14.22	13.63	13.32
βFS	18.71	16.35	15.81	15.51	14.94	15.10	14.20	13.63	13.47
FFS	18.63	16.30	15.78	15.50	14.98	15.16	14.29	13.51	13.44
βFFS	18.73	16.33	15.82	15.55	14.97	15.14	14.24	13.36	13.04

Running times in hundredths of seconds for a Rand64 problem

$\sigma = 128$	2	4	6	8	10	20	40	80	160
BM	29.36	19.29	17.13	15.96	15.59	15.27	14.00	12.42	11.90
βBM	28.84	19.40	17.40	15.95	15.64	15.24	13.93	12.38	11.96
FS	18.59	16.32	15.78	15.57	14.90	15.32	13.84	12.37	11.93
βFS	18.58	16.38	15.83	15.61	14.88	15.30	13.87	12.35	12.07
FFS	18.59	16.29	15.83	15.59	14.96	15.34	13.96	12.51	12.42
βFFS	18.56	16.28	15.90	15.60	14.95	15.37	13.86	12.34	11.87

Running times in hundredths of seconds for a Rand128 problem

NL	2	4	8	16	32	64	128	256	512
BM	5.56	3.35	3.46	2.75	2.65	2.70	2.30	1.45	2.30
βBM	5.57	3.11	2.56	2.25	2.41	2.60	2.35	1.30	2.05
FS	2.65	2.60	2.60	2.46	2.45	2.25	1.70	1.40	1.65
βFS	3.56	2.71	2.87	2.50	2.30	2.81	1.15	1.35	1.76
FFS	3.55	2.85	2.40	2.90	2.85	2.65	2.42	2.21	1.91
βFFS	2.45	2.75	2.30	2.46	2.41	2.55	1.40	1.25	1.26

Running times in thousandths of seconds for a natural language problem

Prot	2	4	8	16	32	64	128	256	512
BM	73.16	49.87	43.46	38.75	38.46	37.19	37.26	34.95	34.55
βBM	71.94	49.08	43.49	38.76	37.59	37.77	36.74	35.53	34.39
FS	45.81	40.01	38.06	36.85	35.97	36.34	35.66	33.77	33.34
βFS	45.38	39.65	37.91	36.99	36.41	36.10	35.24	33.79	33.86
FFS	45.26	39.71	37.61	37.50	37.43	36.45	36.21	33.90	36.40
βFFS	45.82	40.01	38.01	37.20	35.80	36.55	34.95	32.45	31.65

Running-times in thousandths of seconds for a protein sequence problem

The above experimental results show that the algorithms βBM, βFS, and βFFS have much the same running times of the algorithms BM, FS, and FFS. Only when the size of the alphabet is 2 the "bounded" versions have a slightly worse performance than their counterparts, especially for short patterns. On the other hand, as the size of the alphabet and pattern increases, often the "bounded" versions moderately outperform their counterpart. In particular, this behavior is more noticeable in the case of the Forward-Fast-Search algorithm and in the cases of the real data problems. The latter remark shows that our simplifying hypotheses in the analysis put forward in Section 4 do not lead to unrealistic results.

6 Conclusions

Space and time economy are essential features of any practical algorithm. However, they are often sacrificed in favor of asymptotic efficiency. This is the case of the most practical string matching algorithms which show in practice a sublinear behavior at the price of using extra memory of non-constant size to maintain auxiliary information. The Boyer-Moore algorithm, for instance, requires additional $\mathcal{O}(m)$ and $\mathcal{O}(|\Sigma|)$-space to compute the tables relative to the good-suffix and to the bad-character heuristics, respectively.

In this paper we have presented a practical modification of the good-suffix heuristic, called bounded good-suffix heuristic, which uses only constant space and can be computed in $\mathcal{O}(m)$-time and constant space.

Through an extensive collection of experimental tests on the Boyer-Moore algorithm and two of its most efficient variants (namely the algorithms Fast-Search and Forward-Fast-Search) we have shown that the "bounded" versions are comparable with their counterparts, which are often outperformed by them.

We are currently investigating the problem of finding an effective string matching algorithm which requires only extra constant space. To this purpose, we expect that the bad-character heuristic (which needs $\mathcal{O}(|\Sigma|)$-space) needs to be dropped and substituted by a heuristic of a different kind.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Conference '99, J. Holub and M. Šimánek, eds., Czech Technical University in Prague, 1999, Collaborative Report DC–99–05.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. D. Breslauer: *Saving comparisons in the Crochemore-Perrin string matching algorithm.* Theor. Comput. Sci., 158 1996.
4. D. Cantone and S. Faro: *Fast-Search: a new variant of the Boyer-Moore string matching algorithm*, in Proceedings of Second International Workshop on Experimental and Efficient Algorithms (WEA 2003), K. Jansen, M. Margraf, M. Mastrolilli, and J. Rolim, eds., vol. 2647 of Lecture Notes in Computer Science, Springer-Verlag, 2003.
5. D. Cantone and S. Faro: *Forward-Fast-Search: another fast variant of the Boyer-Moore string matching algorithm*, in Proceedings of the Prague Stringology Conference '03, M. Šimánek, ed., Czech Technical University in Prague, 2003.
6. D. Cantone and S. Faro: *Searching for a substring with constant extra-space complexity*, in Proceedings of Third International Conference on FUN with Algorithms (FUN 2004), P. Ferragina and R. Grossi, eds., Edizioni Plus, Università di Pisa, 2004.

7. M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: *Speeding up two string matching algorithms.* Algorithmica, 12(4/5) 1994.
8. M. Crochemore, L. Gąsieniec, and W. Rytter: *Constant-space string-matching in sublinear average time.* Theor. Comput. Sci., 218(1) 1999.
9. M. Crochemore and D. Perrin: *Two-way string-matching.* Journal of the ACM, 38(3) 1991.
10. Z. Galil and J. Seiferas: *Saving space in fast string-matching.* SIAM J. Comput., 9(2) 1980.
11. L. Gąsieniec, W. Plandowski, and W. Rytter: *Constant-space string matching with smaller number of comparisons: sequential sampling*, in Proc. 6th Symp. Combinatorial Pattern Matching, Z. Galil and E. Ukkonen, eds., vol. 937 of Lecture Notes in Computer Science, Springer-Verlag, 1995.
12. L. Gąsieniec, W. Plandowski, and W. Rytter: *The zooming method: a recursive approach to time-space efficient string-matching.* Theor. Comput. Sci., 147(1–2) 1995.
13. L. J. Guibas and A. M. Odlyzko: *A new proof of the linearity of the Boyer-Moore string searching algorithm.* SIAM J. Comput., 9(4) 1980.
14. R. N. Horspool: *Practical fast searching in strings.* Softw. Pract. Exp., 10(6) 1980.
15. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(2) 1977, pp. 323–350.
16. T. Lecroq: *New experimental results on exact string-matching*, Université de Rouen, France, 2000, Rapport LIFAR 2000.03.
17. W. Rytter: *A correct preprocessing algorithm for Boyer-Moore string searching.* SIAM J. Comput., 9 1980.
18. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990.

On the Complexity of Variants of the k Best Strings Problem

Martin Berglund and Frank Drewes

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

Abstract. We investigate the problem of extracting the k best strings from a non-deterministic weighted automaton over a semiring $\mathbb{S}$. This problem, which has been considered earlier in the literature, is more difficult than extracting the k best runs, since distinct runs may not correspond to distinct strings. Unsurprisingly, the computational complexity of the problem depends on the semiring $\mathbb{S}$ used. We study three different cases, namely the tropical and complex tropical semirings, and the semiring of positive real numbers. For the first case, we establish a polynomial algorithm. For the second and third cases, NP-completeness and undecidability results are shown.

1 Introduction

Weighted finite-state automata (WFA) are a popular tool for representing weights assigned to potentially infinite languages of strings. This is useful in many areas, notable cases being natural language processing and speech recognition. These automata are constructed in a way that conveniently solves the weighted version of the membership problem, that is, the problem of computing the weight of a string. In many cases, however, the WFA represents something like a "hypothesis space", where the weights represent some kind of desirability or quality. For example a natural language translation system may be implemented as a weighted transducer, which for an input string produces a WFA as output. This WFA then assigns weights to strings according to the likelihood that the string is a good translation of the original input. We may then wish to somehow enumerate a few of the "best" runs or strings with respect to a given WFA.

In a WFA over a semiring $\mathbb{S}$, transitions are assigned a weight in $\mathbb{S}$. Essentially, the weight of a run is the product of the weights of the edges traversed, and the weight of an input string is the sum of the weights of all its runs.[1] The problem of finding the best *runs* is well-explored: it is the problem of finding the shortest paths in a directed weighted graph. Notably, there are very efficient algorithms to, in order, enumerate the shortest paths through a graph, where the edge weights are usually interpreted as lengths and are, accordingly, summed up [2]. This corresponds to the use of the tropical semiring, whose multiplication is ordinary addition and whose addition takes the minimum. Algorithms for best runs in WFA over other types of semirings have also been investigated [6].

Not quite as well investigated is the k best strings problem (k-BSP) where the aim is to find the best *strings*, i.e., those with the least weight. This is different from the problem of finding the best runs if non-deterministic WFA are considered, as the

[1] Here, products and sums are built by using the multiplication and addition, respectively, of the semiring.

Martin Berglund, Frank Drewes: On the Complexity of Variants of the k Best Strings Problem, pp. 76–88 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

weight of a string is the sum of the weights of all runs for that particular input string. Thus, while every run corresponds to a particular input string, the weight of the run does not in general coincide with the weight of the string. Furthermore, distinct runs may correspond to the same input string, whereas the k-BSP asks for the k best *unique* strings. This makes the problems differ even for the particular case of WFA over the tropical semiring, where the weight of an input string is always the lowest weight among all runs associated with the string. In the extreme case, the k best runs may all belong to the same input string.

In [7], an algorithm is presented that solves the k-BSP for WFA over the tropical semiring. This algorithm is based on a clever on-demand (or lazy) determinization, which stops when the k runs with the least weight (in the determinized part) have been found. The algorithm has been reported to be very efficient in practice. However, it appears to run in exponential time in some bad cases. Therefore, it is a natural question to ask whether there is a polynomial algorithm solving the problem.

In this paper, we give a positive answer to this question. Of course, this raises the question whether the setting can be generalized to other semirings without losing tractability. The first answers to this question will be given by considering a decision problem closely related to the k-BSP for $k = 1$, namely the *string quality threshold problem* (SQTP). Here, we are given a WFA and a threshold $t \in \mathbb{S}$, and the question is whether there exists a string whose weight is less than or equal to t (with respect to the order considered).

In summary, we establish the following three main results:

- The k-BSP for WFA over the tropical semiring is solvable in polynomial time.
- For WFA over the tropical semiring on pairs of numbers (which we call the complex tropical semiring), the SQTP is NP-complete.
- For WFA over the semiring of positive real numbers with the usual addition and multiplication, the SQTP is undecidable.

The remainder of the paper is structured as follows. In the next section, basic notions and notation are compiled. In Section 3, the problems to be investigated are defined. In Sections 4, 5, and 6, the three main results are shown. Finally, a short conclusion is given in Section 7.

2 Basic Notions and Notation

For $n \in \mathbb{N}$, we denote the set $\{1, \ldots, n\}$ by $[n]$. The set $\{x \mid x \in \mathbb{R}, x \geq 0\} \cup \{\infty\}$ is denoted by $\mathbb{R}_+^\infty$. Similarly, $\mathbb{C}_+^\infty$ denotes $\{x + yi \mid x \in \mathbb{R}_+^\infty, y \in \mathbb{R}_+^\infty\}$.

We denote a semiring as a tuple $(\mathbb{S}, \oplus, \otimes)$ where $\mathbb{S}$ is the domain, $\oplus$ the addition operator and $\otimes$ the multiplication operator. Semirings will often be equipped with a (possibly partial) order $\leq$, in which case the semiring is denoted by $(\mathbb{S}, \oplus, \otimes, \leq)$. If $\oplus$ and $\otimes$ (and $\leq$) are clear from the context, then the semiring may simply be denoted by $\mathbb{S}$.

For an alphabet Σ, Σ^* denotes the set of all strings over Σ. The empty string, i.e., the string of length 0, is denoted by ϵ. The length of a string s is denoted by $|s|$, and $s \cdot s'$ or simply ss' denotes the concatenation of s with another string s'. The notation $|S|$ is also used to denote the cardinality of a set S.

A weighted finite-state automaton (WFA) is a tuple $A = (\Sigma, Q, \mathbb{S}, \mu, \lambda, \rho)$ where Σ is a finite alphabet of input symbols, Q is a finite set of states, $\mathbb{S}$ is the semiring from which the weights are taken, $\mu \colon Q \times \Sigma \times Q \to \mathbb{S}$ is the weighted transition

function and $\lambda, \rho\colon Q \to \mathbb{S}$ are the initial and final weight vectors, respectively. The WFA A is *deterministic* if, for all $q \in Q$ and $a \in \Sigma$, there is at most one $q' \in Q$ such that $\mu(q, a, q') \neq 0$.

The transition function μ can alternatively be viewed as a set of rules, namely

$$R_\mu = \{q \xrightarrow{w,a} q' \mid (q, a, q') \in Q \times \Sigma \times Q \text{ and } \mu(q, a, q') = w \in \mathbb{S} \setminus \{0\}\}$$

(where 0 is the additive identity of $\mathbb{S}$). Thus, the case $\mu(q, a, q') = 0$ corresponds to a non-existing rule. We define the size $|A|$ of an automaton as the number of rules, i.e., $|A| = |R_\mu|$.

A WFA A computes a function $\underline{A}\colon \Sigma^* \to \mathbb{S}$, called a string series in the theory of weighted automata [1], as follows: for all strings $s = a_1 \cdots a_n$,

$$\underline{A}(s) = \sum_{p_1,\ldots,p_{n+1} \in Q} \lambda(p_1) \left(\prod_{i=1}^{n} \mu(p_i, a_i, p_{i+1}) \right) \rho(p_{n+1}).$$

Here, the sums and products are defined using the operators $\oplus$ and $\otimes$, resp., of the semiring. An alternating sequence of states and input symbols $p_1, a_1, p_2, \ldots, a_n, p_{n+1}$ is also called a run of A on s. The weight of the run is given by the product $\lambda(p_1) \left(\prod_{i=1}^{n} \mu(p_i, a_i, p_{i+1})\right) \rho(p_{n+1})$. In other words, $\underline{A}(s)$ is the sum of the weights of all runs on s.

By abuse of notation, $\underline{A}$ will simply be denoted by A from now on. We write $\mathrm{WFA}_\Sigma^{\mathbb{S}}$ to denote the set of all WFA over Σ and $\mathbb{S}$.

The reader may have noticed that we do not allow ϵ transitions in our WFA. However, this restriction is not essential in any case studied here. Its sole purpose is to simplify the presentation of the algorithm in Section 4. Let us have a look at an example semiring to wrap this section up.

Example 1 (The tropical semiring). The tropical semiring is an important case both for the following sections and in many practical applications. It is defined as $\mathsf{Trop} = (\mathbb{R}_+^\infty, \min, +, \leq)$, where min, + and $\leq$ all have their usual meanings on $\mathbb{R}_+^\infty$. Note that the product in Trop is ordinary addition. For all $A = (\Sigma, Q, \mathsf{Trop}, \mu, \lambda, \rho) \in \mathrm{WFA}_\Sigma^{\mathsf{Trop}}$ and all strings $s = a_1 \cdots a_n$ the formula above gives us

$$A(s) = \min_{p_1,\ldots,p_{n+1} \in Q} \lambda(p_1) + \left(\sum_{i=1}^{n} \mu(p_i, a_i, p_{i+1}) \right) + \rho(p_{n+1}).$$

That is, finding the weight that A assigns to the string s corresponds to finding the run on s with the minimal total weight. This makes the tropical semiring case closely related to various shortest-path problems, as we will see in coming sections.

Consider the input alphabet $\Sigma = \{a, b\}$. We give a WFA over Trop such that, for every $s \in \Sigma^*$, $A(s) = |s| - l$, where l is the length of the longest substring in s consisting only of the symbol a. For this, we let $Q = \{\text{prefix}, \text{middle}, \text{suffix}\}$, $\lambda(q) = 0$ for all $q \in Q$,

$$\begin{aligned} R_\mu = \{ &\text{prefix} \xrightarrow{1,x} \text{prefix } (x \in \Sigma), \\ &\text{prefix} \xrightarrow{1,b} \text{middle}, \\ &\text{middle} \xrightarrow{0,a} \text{middle}, \\ &\text{middle} \xrightarrow{1,b} \text{suffix}, \\ &\text{suffix} \xrightarrow{1,x} \text{suffix } (x \in \Sigma) \ \}, \end{aligned}$$

and

$$\rho(q) = \begin{cases} 0 & \text{if } q \in \{\text{middle}, \text{suffix}\} \\ \infty & \text{otherwise.} \end{cases}$$

Intuitively, the WFA guesses non-deterministically the part to be left out when counting the symbols the input string consists of.

3 Problem Definitions

As mentioned above, given a WFA $A \in \text{WFA}_\Sigma^{\mathbb{S}}$ and some $k \in \mathbb{N}$, we are interested in computing the k "best" strings in the sense that these strings are assigned the lowest weights by A. The formal definition of the problem reads as follows.

Definition 2 (k best strings problem). *Let $(\mathbb{S}, \oplus, \otimes, \leq)$ be a partially ordered semiring, and let Σ be an alphabet. An instance of the* k best strings problem *(k-BSP) over $\mathbb{S}$ is a pair $(A, k) \in \text{WFA}_\Sigma^{\mathbb{S}} \times \mathbb{N}$. A* solution *to the instance is a set S of strings in Σ^* such that $|S| = k$ and, for all strings s, s', if $A(s) < A(s')$ and $s' \in S$ then $s \in S$.*

The 1-BSP is the special case of the k-BSP where $k = 1$ is considered to be fixed.

Notice that the solution S is not necessarily unique, because each string $s' \in S$ can be replaced with any other string $s \in \Sigma^* \setminus S$ such that $A(s) = A(s')$. Thus, there may even be an infinite number of solutions. Also note that in some cases no solution may exist, because a solution cannot include elements from an infinite chain $s_0, s_1, s_2, \ldots$ such that $A(s_{i+1}) < A(s_i)$ for all $i \in \mathbb{N}$, i.e., the s_i get "better and better". This cannot happen if $<$ is well-founded, as will be the case in the next two sections.

We also consider a closely related decision problem.

Definition 3 (String quality threshold problem). *Let $(\mathbb{S}, \oplus, \otimes, \leq)$ be a partially ordered semiring, and let Σ be an alphabet. An instance of the* string quality threshold problem *(SQTP) is a pair $(A, t) \in \text{WFA}_\Sigma^{\mathbb{S}} \times \mathbb{S}$. The question to be answered is whether there exists a string $s \in \Sigma^*$ such that $A(s) \leq t$.*

As usual, we shall identify a decision problem such as the SQTP with the set of all its *yes* instances. Thus, given an instance I, we write $I \in \text{SQTP}$ to express that I is a *yes* instance of *SQTP*. Note that if the 1-BSP problem A has a solution $\{s\}$ we will have $(A, t) \in \text{SQTP}$ for all $t \geq A(s)$.

The problems are closely related in the other direction as well: as long as $\leq$ is a total order, it holds that for all $(A, t) \in \text{SQTP}$ all solutions $\{s\}$ to the 1-BSP problem $(A, 1)$ satisfy $A(s) \leq t$. That is, if we know that there exists some string with weight less than or equal to t then any algorithm solving the 1-BSP will have to find such a string.

4 A Polynomial k Best Strings Algorithm for the Tropical Case

In this section, we show that the k-BSP can be solved in polynomial time for the tropical semiring $\mathsf{Trop} = (\mathbb{R}_+^\infty, \min, +, \leq)$ (as defined in Example 1). For the rest of this section, let us consider an instance (A, k) of the k-BSP over Trop, where $A = (\Sigma, Q, \mathsf{Trop}, \mu, \lambda, \rho)$. Let us start with an important lemma.

Lemma 4 (Short minimal strings). *For any string $s \in \Sigma^*$ let $l = \left\lfloor \frac{|s|}{|Q|} \right\rfloor$. Then there exists at least l distinct strings $s_1, \ldots, s_l$ such that $|s_i| \leq |s|$ and $A(s_i) \leq A(s)$ for all $i \in [l]$.*

Proof. Let $s = a_1 \cdots a_n$. By the definition of $A(s)$, together with the fact that A is defined over Trop, we know that $A(s) = \min_{p_1,\ldots,p_{n+1} \in Q} \lambda(p_1) + (\sum_{i=1}^{n} \mu(p_i, a_i, p_{i+1})) + \rho(p_{n+1})$. Let $p_1, \ldots, p_{n+1} \in Q$ be one of the (not necessarily unique) choices of states that minimize the expression. Then, since $n + 1 > l|Q|$ there exists a state $q \in Q$ which occurs $l + 1$ or more times among $p_1, \ldots, p_{n+1}$ (by the pigeon hole principle). Let $i_1, \ldots, i_{l+1} \in [n]$ be the distinct indices such that $p_{i_j} = q$ for all j. For all $j \in [l+1]$ let

$$F(j) = \lambda(p_1) + \left(\sum_{h=1}^{i_j - 1} \mu(p_h, a_h, p_{h+1})\right) + \left(\sum_{h=i_{l+1}}^{n} \mu(p_h, a_h, p_{h+1})\right) + \rho(p_{n+1}).$$

Notice that $F(j) \leq A(s)$ for all $j \in [l+1]$, owing to the fact that all terms of the sum defining $F(j)$ are also part of the sum defining $A(s)$. Now, for each $j \in [l+1]$ construct the string $s'_j = a_1 \cdots a_{i_j - 1} a_{i_{l+1}} \cdots a_n$. Notice that these strings are pairwise distinct. For each of them it holds that $A(s'_j) \leq F(j)$, since $F(j)$ is the weight corresponding to one of the possible state choices for the minimization in the evaluation of $A(s'_j)$.

Thus, as required, we have obtained l distinct strings $s'_1, \ldots, s'_l$ such that $A(s'_j) \leq F(j) \leq A(s)$. □

As a rather direct consequence, we get the following.

Corollary 5 (Existence of a short k-BSP solution). *If (A, k) has a solution, then it has a solution S such that $|s| \leq k|Q|$ for all $s \in S$.*

To see this, simply consider a solution S that contains a string s longer than the corollary states is necessary. By Lemma 4, there exists a shorter string $s' \notin S$ of the same weight, which s can be replaced with.

Now we are ready to describe, as a first step towards solving the k-BSP, an algorithm that solves the 1-BSP. This does in the process solve the SQTP, since our semiring is totally ordered. For solving the 1-BSP, we can simply apply Dijkstra's algorithm to A.

Algorithm 6 (1-BSP and SQTP algorithm). View A as a directed labeled weighted graph by considering the states to be nodes and the rules to be weighted edges. Then simply apply Dijkstra's algorithm [8] to A in the following way:

1. For each $q \in Q$ the algorithm assigns a weight $\mathit{weight}(q)$ to q. Initially, $\mathit{weight}(q) = \lambda(q)$ and $U = Q$.
2. Take any $q \in U$ with $\mathit{weight}(q) = \min_{q' \in U} \mathit{weight}(q')$.
3. For all edges $q \xrightarrow{w,a} q'$ set $\mathit{weight}(q') = \min(\mathit{weight}(q'), \mathit{weight}(q) + w)$.
4. Let $U = U \setminus \{q\}$. If $U \neq \emptyset$ go to step 2.

Now create the directed labeled graph $G = (V, E)$ where $V = Q$ and $E = \{q \xrightarrow{a} q' \mid (q, a, q', w) \in \mu, \mathit{weight}(q') = \mathit{weight}(q) + w\}$. Then let $\hat{w} = \min_{q \in Q} \mathit{weight}(q) + \rho(q)$, let $F = \{q \in Q \mid \mathit{weight}(q) + \rho(q) = \hat{w}\}$, and let $I = \{q \in Q \mid \mathit{weight}(q) = \lambda(q)\}$. Then simply perform a breadth-first search to find the (not necessarily unique) shortest path $q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} q_{n+1}$ through G such that $q_1 \in I$ and $q_{n+1} \in F$.

Now, $s = a_1 \cdots a_n$ is a solution to the 1-BSP, and we have $A(s) = \hat{w}$. Consequently, $(A, t) \in$ SQTP if and only if $t \geq \hat{w}$. The running time of this algorithm,

being dominated by the running time of Dijkstra's algorithm, is $\mathcal{O}(|A| + |Q| \log |Q|)$, provided that some well-known optimizations are made [8].

The following lemma summarizes the properties of Algorithm 6.

Lemma 7. *Algorithm 6 solves the* 1*-BSP in time* $\mathcal{O}(|A| + |Q| \log |Q|)$*. Furthermore, if it returns* $\{s\}$*, then* $|s| \leq |s'|$ *holds for every other solution* $\{s'\}$ *to the* 1*-BSP instance.*

The fact that the lemma above ensures $|s| \leq |s'|$ will enable us to exploit Corollary 5. As building blocks for the general algorithm, let us make two more definitions.

Definition 8 (*S*-complement WFA**).** *For any finite set* $S \subset \mathcal{P}(\Sigma^*)$ *let* $A_S \in \mathrm{WFA}_\Sigma^{\mathsf{Trop}}$ *denote the automaton* $(\Sigma, Q_S, \mathsf{Trop}, \mu_S, \lambda_S, \rho_S)$*, constructed in the following way.*

- $Q_S = \{sink, r_\epsilon\} \cup \{r_s \mid s \text{ is a prefix of a string in } S\}$.
- *For all* $q \in Q_S$,

$$\lambda(q) = \begin{cases} 0 & \text{when } q = r_\epsilon \\ \infty & \text{otherwise,} \end{cases}$$

$$\rho(q) = \begin{cases} \infty & \text{if } q = r_s \text{ for some } s \in S \\ 0 & \text{otherwise.} \end{cases}$$

- *For all* $r_s \in Q_S$ *and all* $c \in \Sigma$, R_{μ_S} *contains the rules*
  - $r_s \xrightarrow{0,c} r_{sc}$ *if* $r_{sc} \in Q_S$,
  - $r_s \xrightarrow{0,c} sink$ *if* $r_{sc} \notin Q_S$, *and*
  - $sink \xrightarrow{0,c} sink$.

  The reader should easily be able to check that, for all $s \in \Sigma^*$,

$$A_S(s) = \begin{cases} \infty \text{ if } s \in S \\ 0 \quad \text{otherwise.} \end{cases}$$

Definition 9 (Product-$\mathrm{WFA}^{\mathsf{Trop}}$**).** *For all WFA* $A_1 = (\Sigma, Q_1, \mathsf{Trop}, \mu_1, \lambda_1, \rho_1)$ *and* $A_2 = (\Sigma, Q_2, \mathsf{Trop}, \mu_2, \lambda_2, \rho_2)$*, let* $A_1 \times A_2$ *denote the* product automaton*, defined by* $A_1 \times A_2 = (\Sigma, Q_1 \times Q_2, \mathsf{Trop}, \mu, \lambda, \rho)$ *where, for all* $(q_1, q_2) \in Q_1 \times Q_2$,

- $\lambda(q_1, q_2) = \lambda_1(q_1) + \lambda_2(q_2)$,
- $\rho(q_1, q_2) = \rho_1(q_1) + \rho_2(q_2)$, *and*
- $\mu((q_1, q_2), a, (q_1', q_2')) = \mu_1(q_1, a, q_1') + \mu_2(q_2, a, q_2')$ *for all* $(q_1', q_2') \in Q_1 \times Q_2$ *and all* $a \in \Sigma$.

It should be clear that, for all $s \in \Sigma^*$, we have $(A_1 \times A_2)(s) = A_1(s) + A_2(s)$. Given the 1-BSP algorithm, it is now straightforward to construct the algorithm that solves the k-BSP.

Algorithm 10 (*k*-BSP algorithm). To compute a solution to the k-BSP instance (A, k), we proceed as follows.

1. Initially, let $S = \emptyset$.
2. If $|S| = k$ halt and return S as the answer.

3. Construct the automaton $A' = A \times A_S$ where A_S is the S-complement WFA as in Definition 8 (this gives $A' = A$ for $S = \emptyset$).
4. Apply Algorithm 6 to the 1-BSP instance A', and let s be the answer the algorithm computes.
5. Let $S = S \cup \{s\}$ and go to step 2.

There is one degenerate case, where the string s computed in step 4 satisfies $A'(s) = \infty$ (that is, all strings that have a finite weight in A have already been picked). To handle that edge case simply pick the remaining $|S| - k$ strings from $\Sigma^* \setminus S$ arbitrarily and halt. In all other cases, $s \notin S$ will hold at step 3.

Next, we establish the correctness and complexity of Algorithm 10 to complete this section.

Theorem 11. *If applied to a k-BSP instance (A, k), Algorithm 10 returns a correct solution in time $\mathcal{O}((k^3|A|^2)\log(k|A|))$.*

Proof. The correctness of the algorithm is straightforward to show. Consider steps 3 and 4 of the algorithm, and suppose that $A'(s) \neq \infty$. By the properties of A' noted above, and by Lemma 7, s is a shortest string such that $A(s) = \min\{A(s') \mid s' \in \Sigma^* \setminus S\}$. By induction, this means that the set S that is eventually returned is a valid solution. Furthermore, S is a shortest solution, i.e., every other solution S' satisfies $\sum_{s \in S'} |s| \geq \sum_{s \in S} |s|$.

As for the complexity of the algorithm, consider the k^{th} iteration. Lemma 4 and the fact that S is a shortest solution yield $\max_{s \in S} |s| \leq k|Q|$. This means that $\sum_{s \in S} |s| \in \mathcal{O}(k^2|Q|)$. Constructing the automaton A_S according to Definition 8 will then give us $\mathcal{O}(k^2|Q|)$ states, and since A_S is deterministic we have $|A_S| \in \mathcal{O}(k^2|Q|)$.[2] Thus, A' consists of $\mathcal{O}(k^2|Q|^2)$ states and $\mathcal{O}(k^2|Q||A|)$ rules. By Lemma 7, this means that Algorithm 6 runs in time $\mathcal{O}(k^2|A||Q| + (k^2|Q|^2)\log(k^2|Q|^2))$. Summing up over the k iterations of the algorithm, this yields a running time of

$$\mathcal{O}(k^3|A||Q| + k^3|Q|^2 \log(k^2|Q|^2)) = \mathcal{O}(k^3|A||Q| + k^3|Q|^2 \log(k|Q|)).$$

Since $|Q| < |A|$ in all non-degenerate cases, this yields the bound stated. □

5 The Complex Tropical Case is NP-Complete

We now consider the extension of the tropical semiring to the plane, called the complex tropical semiring. It is defined as $\mathsf{Trop}^2 = (\mathbb{C}_+^\infty, \min, +, \leq)$ where min and + are component-wise minimum and addition (i.e., $\min(x + yi, x' + y'i) = \min(x, x') + \min(y, y')i$, and similarly for +). The (partial) order $\leq$ of this semiring is also defined component-wise, i.e., for all $a, b \in \mathbb{C}_+^\infty$ we have $a \leq b$ if and only if $a = \min(a, b)$. This case is an interesting extension of the normal tropical case, and would be useful in settings where one wishes to track multiple qualities of a string independently. For example in the case of natural language correction one could let the first component signify the prevalence of typing mistakes (spelling fixes), whereas the second component could signify the severity of structural mistakes (for example typical mistakes for non-native speakers, like verb-subject agreement). We prove that the SQTP is NP-complete even for deterministic WFA over Trop^2. We start by showing that the problem is in NP, followed by showing that it is NP-hard.

[2] Here, we consider $|\Sigma|$ to be a constant.

Lemma 12. *The SQTP for* WFA *over* Trop^2 *is in NP.*

Proof. Lemma 4 holds even for Trop^2, with precisely the same proof. As a direct consequence Corollary 5 also holds. From this it follows that for any WFA $A = (\Sigma, Q, \mathsf{Trop}^2, \mu, \lambda, \rho)$ and $t \in \mathbb{C}_+^\infty$ such that $(A, t) \in$ SQTP there exists some $s \in \Sigma^*$ with $|s| \leq |Q|$ such that $A(s) \leq t$.

We can then solve the SQTP instance (A, t) by non-deterministically choosing any string $s \in \Sigma^*$ with $|s| \leq |Q|$ and checking if $A(s) \leq t$. If this succeeds then $(A, t) \in$ SQTP. □

Lemma 13. *The SQTP for deterministic* WFA *over* Trop^2 *is NP-hard .*

To prove the theorem by reduction, recall the shortest weight-constrained path problem [3].

Definition 14 (Shortest weight-constrained path). *An instance of the shortest weight-constrained path problem (SWCP) is a tuple $I = (G, w, l, (u, v), (M_w, M_l))$, where G is a directed graph $G = (V, E)$, $w \colon E \to \mathbb{N}^+$ is a weight function, $l \colon E \to \mathbb{N}^+$ is a length function, $u, v \in V$ and $M_w, M_l \in \mathbb{N}^+$. The question to be answered is whether there exists a path from u to v such that the total weight of all edges on the path is less than M_w and the total length of all edges on the path is less than M_l.*

This problem is known to be NP-complete [3]. Given an instance I of the SWCP as above, we construct a weighted automaton A over Trop^2 such that $(A, t) \in$ SQTP for $t = M_w + M_l i$ if and only if $I \in$ SWCP. The construction is straightforward, as follows.

Construct the WFA $A = (\Sigma, Q, \mathsf{Trop}^2, \mu, \lambda, \rho)$ as follows. Let $Q = V$, let $\Sigma = \{\widehat{v_1 v_2} \mid (v_1, v_2) \in E\}$, for all $q \in Q$ let

$$\lambda(q) = \begin{cases} 0 + 0i & \text{if } q = u \\ \infty + \infty i & \text{otherwise,} \end{cases}$$

and

$$\rho(q) = \begin{cases} 1 + 0i & \text{if } q = v \\ 0 + 0i & \text{otherwise.} \end{cases}$$

Then we simply let R_μ consist of all rules $v_1 \xrightarrow{\hat{w}, \widehat{v_1 v_2}} v_2$, such that $(v_1, v_2) \in E$ and $\hat{w} = w(v_1, v_2) + l(v_1, v_2)i$.

Thus, the WFA interprets the input string as a sequence of edges in G. If this sequence is a path starting at u, the state corresponding to the node reached on this path will carry the weight equal to the weight-length combination up to that point; all other states will carry the weight $\infty + \infty i$. Owing to the choice of ρ, this yields the desired result, i.e., $(A, t) \in$ SQTP for $t = M_w + M_l i$ if and only if $I \in$ SWCP.

Summing up, we have proved the following theorem.

Theorem 15. *The SQTP for WFA over* Trop^2 *is NP-complete.*

Proof. Follows from Lemma 12 together with Lemma 13. □

6 The General Case is Undecidable

We finally identify a semiring for which the SQTP turns out to be undecidable: the semiring $(\mathbb{R}_+, +, *, \leq)$, where $+$ and $*$ are ordinary addition and multiplication and $\leq$ is the usual order on $\mathbb{R}_+$. This case is not in itself practically motivated, but it is very useful to clearly illustrate that there is no hope for a truly general solution to the SQTP for arbitrary semirings.

For a proof by reduction, we consider the problem whether a Turing machine accepts the empty string (or, equivalently, whether a Turing machine without input halts). We reduce this problem to the SQTP for WFA over $\mathbb{R}_+$, constructing the WFA in such a way that the string series it computes will assign the weight 1 to some string only if the Turing machine has an accepting run. Otherwise, the weights of all strings will be strictly larger than 1. Throughout this section non-determinism, in the same sense as in non-deterministic finite automata, will be a key concern, importantly we will use non-deterministic Turing machines [4] as the starting point of the reduction. That is, during the computation the Turing machine will sometimes make non-deterministic choices between different instructions to jump to. As usual, the acceptance criterion is that a computation ending in the accepting state exists. The WFA constructed by the reduction will be non-deterministic as well.

Let us start by defining the precise machine model we will use. Rather than using ordinary Turing machines, we use the well-known two-counter machines [5] as our starting point. Let us first recall the basic definition of the original two-counter machine.

Definition 16 (Two-counter machine). *A two-counter machine without input is a tuple $M = (C, P, c_0)$ consisting of a finite set C of states, a starting state $c_0 \in C$, and a program $P\colon C \to (\{\mathit{inc}_1, \mathit{inc}_2\} \times C) \cup (\{\mathit{jzdec}_1, \mathit{jzdec}_2\} \times C \times C) \cup \{\mathit{accept}\}$.*

The semantics of a two-counter machine is the usual one from [5]. The machine starts in state c_0 with the counters set to zero. In state c, the instruction $P(c)$ is executed:

1. $(\texttt{inc}_i, c')$ increments counter i and continues in state c',
2. $(\texttt{jzdec}_i, c', c'')$ continues in state c' if counter i is zero, and decrements the counter and continues in state c'' if it is not zero, and
3. `accept` halts and accepts the input.

We now adjust this into another type of Turing machine which is equivalent but more convenient for our purpose. The adjustment consists in

- adding another two counters (used for temporary "scratch" values) and allowing all counters to contain negative values, and
- breaking the `jzdec` instruction into two, a `zero` instruction which simply makes the computation immediately fail if the counter tested is non-zero, and a `jump` instruction which *non-deterministically* chooses where to jump. That is, the `jump` instruction has two targets and the machine non-deterministically picks one of them.

This adjustment does not restrict the computational power of counter machines. We provide the definition here for convenience. Four counters are not strictly needed, but are convenient to let us quickly sketch how the `jzdec` instruction can be simulated using the `zero` and `jump` instructions, demonstrating equivalence.

Definition 17 (Four-counter machine). *A non-deterministic four-counter machine is a triple $M = (C, P, c_0)$ consisting of a finite set C of states, a starting state $c_0 \in C$, and a program*

$$P\colon C \to \Big(\bigcup_{i\in[4]} \{\texttt{inc}_i, \texttt{dec}_i, \texttt{zero}_i\} \times C\Big) \cup (\{\texttt{jump}\} \times C \times C) \cup \{\texttt{accept}\}.$$

The computation starts in state c_0 with all four counters set to zero. The semantics of the instructions is given as follows, for all $i \in [4]$:

- $\texttt{inc}_i$ *increments the counter i,*
- $\texttt{dec}_i$ *decrements the counter i (which may result in negative values),*
- $\texttt{zero}_i$ *makes the computation immediately fail if the counter i is not zero,*
- `jump` *non-deterministically jumps to one of the two states given in the instruction, and*
- `accept` *halts and accepts.*

Such a machine is computationally equivalent to a two-counter machine. There is a straightforward simulation of a two-counter machine by a four-counter machine. The latter mimics the two-counter behavior in counters 1 and 2 while using counters 3 and 4 as temporary "scratch" variables. As a building block we can, using counter 4 as a temporary variable, implement the macro instruction $\texttt{copy}_{i\to 3}$ (for $i \in [2]$), which sets the value of counter 3 equal to the (non-negative) value in counter i. Assume that we have counters 3 and 4 set to zero, then transfer the value in counter i into both counter 3 and counter 4 by running the sequence $\texttt{dec}_i$, $\texttt{inc}_3$, $\texttt{inc}_4$ in a loop until counter i is zero (simply loop non-deterministically many times and execute $\texttt{zero}_i$ when the loop ends). Finish the procedure by transferring the contents of counter 4 back into counter i in the same way. Next consider the two-counter machine instruction $P(c) = (\texttt{jzdec}_1, c', c'')$, as given in to Definition 16. This can be translated into the following instructions (using pairwise distinct new states q_x):

$$\begin{aligned}
P(c) &= (\texttt{jump}, q_{zero1}, q_{nonzero1}),\\
P(q_{zero1}) &= (\texttt{zero}_1, c'),\\
P(q_{nonzero1}) &= (\texttt{dec}_1, q_{copy3}),\\
P(q_{copy3}) &= (\texttt{copy}_{1\to 3}, q_{test3}),\\
P(q_{test3}) &= (\texttt{jump}, q_{end}, q_{dec3}),\\
P(q_{dec3}) &= (\texttt{dec}_3, q_{loop}),\\
P(q_{loop}) &= (\texttt{jump}, q_{test3}, q_{test3}),\\
P(q_{end}) &= (\texttt{zero}_3, c'').
\end{aligned}$$

Notice that if this scheme of translating a two-counter machine is used, no counter in an accepting computation will ever actually become negative, since the above snippet will immediately run into an infinite loop if it ever produces a negative counter (by making the wrong non-deterministic choice in the first line). Note that this shows that the accepting run of the four-counter machine simulating a two-counter machine is uniquely determined if it exists. In fact, this holds for arbitrary starting configurations. In the following, we assume that the four-counter machines we are dealing with have this property. For our reduction, it is useful to define its unique accepting computation (if it accepts), called the *trace* of the machine.

Definition 18 (Run of a four-counter machine M). *Let $M = (C, P, c_0)$ be a four-counter machine. The* trace alphabet *of M is*

$$\begin{aligned}\Sigma(M) = \{&\mathit{jump}_{c_i}^{[c]} \mid c \in C, P(c) = (\mathit{jump}, c_1, c_2), i \in [2]\} \cup \\ &\{o_i^{[c]} \mid c \in C, o \in \{\mathit{inc}, \mathit{dec}, \mathit{zero}\}, i \in [4], P(c) = (o_i, c') \text{ for a } c' \in C\} \cup \\ &\{\mathit{accept}^{[c]} \mid c \in C, P(c) = \mathit{accept}\}.\end{aligned}$$

The trace of M, *starting in state c with counter values $\kappa_1, \ldots, \kappa_4 \in \mathbb{N}$, is denoted by $\mathit{trace}(M, c, \kappa_1, \ldots, \kappa_4)$. It is the string $r \in \Sigma(M)^*$ defined as follows:*

1. *If $P(c) = \mathit{accept}$, then $r = \mathit{accept}^{[c]}$.*
2. *If $P(c) = (\mathit{inc}_i, c')$, then the trace is $\mathit{inc}_i^{[c]} \cdot \mathit{trace}(M, c', \kappa_1', \ldots, \kappa_4')$, where $\kappa_i' = \kappa_i + 1$ and $\kappa_j' = \kappa_j$ for all $j \neq i$.*
3. *If $P(c) = (\mathit{dec}_i, c')$, then the trace is $\mathit{dec}_i^{[c]} \cdot \mathit{trace}(M, c', \kappa_1', \ldots, \kappa_4')$, where $\kappa_i' = \kappa_i - 1$ and $\kappa_j' = \kappa_j$ for all $j \neq i$.*
4. *If $P(c) = (\mathit{zero}_i, c')$ then r is undefined unless $\kappa_i = 0$, in which case $r = \mathit{zero}_i^{[c]} \cdot \mathit{trace}(M, c', \kappa_1, \ldots, \kappa_4)$.*
5. *If $P(c) = (\mathit{jump}, c', c'')$ then $r = \mathit{jump}_{c'}^{[c]} \cdot \mathit{trace}(M, c', \kappa_1, \ldots, \kappa_4)$ or $r = \mathit{jump}_{c''}^{[c]} \cdot \mathit{trace}(M, c'', \kappa_1, \ldots, \kappa_4)$, depending on which one is defined. If neither of them is defined then r is undefined.*

Notice that $\mathit{trace}(M, c_0, 0, \ldots, 0)$ is defined if and only if the machine accpets.

With this out of the way we get to the core part of this section. The following construction will take any four-counter machine M and construct a WFA $A \in \mathrm{WFA}_{\Sigma(M)}^{\mathbb{R}_+}$ such that $A(s) \leq 1$ if and only if s is a valid trace of M, that is $s = \mathit{trace}(M, c_0, 0, \ldots, 0)$.

Algorithm 19 (Four-counter WFA reduction). Let $M = (C, P, c_0)$ be a four-counter machine as above. We construct $A = (\Sigma, Q, \mathbb{R}_+, \mu, \lambda, \rho)$ such that there exists $s \in \Sigma^*$ with $A(s) \leq 1$ if and only if $\mathit{trace}(M, c_0, 0, \ldots, 0)$ is defined. This construction can be performed in the following way. Let $Q = \{p_c \mid c \in C\} \cup \bigcup_{i \in [4]} \{c_{i,\mathrm{up}}, c_{i,\mathrm{down}}\} \cup \{\mathrm{fail}, \mathrm{final}\}$. Let $\Sigma = \Sigma(M)$ be as in Definition 18. For all $q \in Q$, let

$$\lambda(q) = \begin{cases} 1 \text{ for } q \in \{p_{c_0}, \mathrm{fail}\} \cup \{c_{i,\mathrm{up}}, c_{i,\mathrm{down}} \mid i \in [4]\} \\ 0 \text{ otherwise} \end{cases}$$

and

$$\rho(q) = \begin{cases} 1 \text{ if } q \in \{\mathrm{fail}\} \cup \{p_c \mid c \in C\} \\ 0 \text{ otherwise.} \end{cases}$$

Of course, the trick lies in the way in which R_μ is constructed. Each state carries a weight holding some invariant meaning in each step over a string. We start with some rules that will keep weights constant in certain situations (i.e., the rules are loops with a weight of 1, the neutral element with respect to multiplication):

C1. $\{\mathrm{fail} \xrightarrow{1,x} \mathrm{fail} \mid x \in \Sigma \setminus \{\mathtt{zero}_i^{[c]} \mid i \in [4], c \in C\}\}$
C2. $\{c_{i,d} \xrightarrow{1,x} c_{i,d} \mid d \in \{\mathrm{up}, \mathrm{down}\}, i \in [4], x \in \Sigma \setminus \{\mathtt{inc}_i^{[c]}, \mathtt{dec}_i^{[c]} \mid c \in C\}\}$

Next, we define rules to manage the accepting state:

S1. $\{\mathrm{fail} \xrightarrow{1,\mathtt{accept}^{[c]}} \mathrm{final} \mid c \in C\}$
S2. $\{\mathrm{final} \xrightarrow{1,x} \mathrm{fail} \mid x \in \Sigma\}$

The following rules are for managing the program counter:

P1. $\{p_c \xrightarrow{1,o_i^{[c]}} p_{c'} \mid c, c' \in C, i \in [4], o \in \{\texttt{inc}, \texttt{dec}, \texttt{zero}\}, (o_i, c') = P(c)\}$

P2. $\{p_c \xrightarrow{1,\texttt{jump}_{c'}^{[c]}} p_{c'} \mid (\texttt{jump}, c_1, c_2) = P(c), c' \in \{c_1, c_2\}\}$

P3. $\{p_{c'} \xrightarrow{1,x^{[c]}} \text{fail} \mid c, c' \in C, x^{[c]} \in \Sigma, c \neq c'\}$

Some rules are needed to manage the counters:

N1. $\{c_{i,\text{up}} \xrightarrow{2,\texttt{inc}_i^{[c]}} c_{i,\text{up}} \mid i \in [4], c \in C\}$

N2. $\{c_{i,\text{down}} \xrightarrow{1/2,\texttt{inc}_i^{[c]}} c_{i,\text{down}} \mid i \in [4], c \in C\}$

N3. $\{c_{i,\text{up}} \xrightarrow{1/2,\texttt{dec}_i^{[c]}} c_{i,\text{up}} \mid i \in [4], c \in C\}$

N4. $\{c_{i,\text{down}} \xrightarrow{2,\texttt{dec}_i^{[c]}} c_{i,\text{down}} \mid i \in [4], c \in C\}$

Finally, the following rules implement the `zero` instruction:

Z1. $\{c_{i,\text{up}} \xrightarrow{1/3,\texttt{zero}_i^{[c]}} \text{fail} \mid i \in [4], c \in C\}$

Z2. $\{c_{i,\text{down}} \xrightarrow{1/3,\texttt{zero}_i^{[c]}} \text{fail} \mid i \in [4], c \in C\}$

Z3. $\{\text{fail} \xrightarrow{1/3,\texttt{zero}_i^{[c]}} \text{fail} \mid i \in [4], c \in C\}$

The idea is that the weight of the trace $\mathit{trace}(M, c_0, 0, 0, 0, 0)$, if it exists, is 1, while all other strings over Σ will get a weight strictly greater than 1. This algorithm may take some explanation to be convincing. Let us note the key invariants exhibited by the construction.

The weight carried by the states $c_{i,\text{up}}$ and $c_{i,\text{down}}$ ($i \in [4]$) will always be 2^x and 2^{-x}, respectively, for some $x \in \mathbb{Z}$. In fact, the rule schemas N1–N4 ensure that the value x corresponds exactly to the value that the counter i would have at the corresponding point in the computation of M. In this way, the counters are represented.

The states p_c represent the current state of M. At each point in time, there is exactly one such state with weight 1, all others carrying the weight 0. Rule schema P3 ensures that if operations ever occur in an order that is impossible in M, weight will be added to the state "fail". This enforces program flow.

This brings us to the important "fail" state, which starts out with the weight 1 and will, by rule schema C1, always keep its weight from the previous step for all input symbols except `zero`. The symbol `zero` is handled specially by the rule schemas Z1–Z3. Let f be the weight of "fail" when encountering the symbol $\texttt{zero}_i^{[c]}$. Since the counter states will have weights 2^x and 2^{-x} (for some $x \in \mathbb{Z}$) the state "fail" gets assigned the weight $\frac{1}{3}f + \frac{1}{3}2^x + \frac{1}{3}2^{-x}$. Notice that if $f = 1$ this sum will be equal to 1 if and only if $x = 0$, and if $f > 1$ then the sum will always be greater than 1. This means that the state "fail" will carry the weight 1 if it did so before and the counter i is zero. Otherwise, it will carry a weight strictly larger than 1. This encodes the effect of the instruction $\texttt{zero}_i$.

Notice that the state "final" gets a weight greater than or equal to 1 when $\texttt{accept}^{[c]}$ is encountered (rule schema S1). On all symbols the weight of "final" gets added to "fail" (rule schema S2), which means that if `accept` is encountered in any but the last position this forces the weight of "fail" to be greater than 1. This enforces a valid end state.

Finally, ρ sums up the weights of "fail" and all the p states. The p states get set to 0 when encountering the right $\texttt{accept}^{[c]}$, so all that remains is the "fail" state weight. As we have seen, "fail" will maintained equal to 1 if all steps follow the constraints of M, and will end up greater than 1 otherwise.

Let us wrap this section up with the theorem stating the result of the reduction.

Theorem 20. *The SQTP is undecidable for WFA over* $(\mathbb{R}_+, +, *, \leq)$.

Proof. Algorithm 19 converts a non-deterministic four-counter machine M into a WFA A over $\mathbb{R}_+$ such that $(A, 1) \in$ SQTP if and only if M accepts (which is an undecidable problem). □

7 Conclusions

We have shown that the k-BSP is efficiently computable in the tropical case by a straightforward algorithm, while even the SQTP is difficult in the complex tropical case and undecidable for the positive real numbers.

Some remaining tasks for future work include improving the polynomial bound for the tropical k-BSP algorithm, which is likely to be possible since the worst-cases of the different parts of the algorithm almost seem to be mutually exclusive. In fact, it may also be worthwhile to analyze the situations in which the algorithm in [7] may exhibit an exponential worst-case behaviour. The algorithm uses a priority queue to determine in which direction the determinization algorithm should proceed. In some bad cases, the problem seems to be that this priority queue may not contain enough information in order for the strategy to become efficient. Hence, one could try to find a refined definition of priorities that (provably) avoids the problem.

Of course, there are many other semirings left for which polynomial solutions of the k-BSP may be obtainable. Moreover, one could try to abstract from concrete semirings by studying properties that give rise to polynomial solutions.

References

1. M. Droste, W. Kuich, and H. Vogler: *Handbook of Weighted Automata*, Springer Publishing Company, Incorporated, 2009.
2. D. Eppstein: *Finding the k shortest paths.* SIAM J. Comput., 28(2) 1999, pp. 652–673.
3. M. R. Garey and D. S. Johnson: *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1990.
4. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, Pearson Education International, Upper Saddle River, N.J. 04758, 2003.
5. M. L. Minsky: *Computation: finite and infinite machines*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
6. M. Mohri: *Semiring frameworks and algorithms for shortest-distance problems.* J. Autom. Lang. Comb., 7(3) 2002, pp. 321–350.
7. M. Mohri and M. Riley: *An efficient algorithm for the n-best-strings problem*, in In Proceedings of the International Conference on Spoken Language Processing 2002, 2002.
8. M. A. Weiss: *Data structures and algorithm analysis*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1992.

Tiling Binary Matrices in Haplotyping: Complexity, Models and Algorithms

Giuseppe Lancia[1], Romeo Rizzi[1], and Russell Schwartz[2]

[1] D.I.M.I., University of Udine
Via delle Scienze 206, 33100 Udine, Italy
{lancia}{rizzi}@dimi.uniud.it
[2] Department of Biological Sciences and Lane Center for Computational Biology, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh 15213, PA, USA
russells@andrew.cmu.edu

Abstract. A tiling of a matrix is an exact cover of its elements by a set of row fragments, called tiles. A particular variant of the tiling problem has arisen in the context of computational biology for studying genetic variations between individuals, in which one wishes to find the minimum-cardinality tiling of a matrix whose rows correspond to genomic sequences of a set of individuals. In this case, the tiles define a set of *haplotype motifs* strings of consecutive variants that frequently co-occur on a single chromosome. By minimizing the number of tiles needed to explain a data set, we seek to identify frequent haplotypes that will be more amenable to statistical analysis than the raw variation data. Although the haplotype motif model was first proposed several years ago, the complexity of the associated optimization problem has never been settled. Here, we show that the minimum tiling problem is NP-hard. We also describe ILP models and Dynamic Programming procedures for its exact solution.

Keywords: haplotyping, tiling, computational biology, computational complexity

1 Introduction

The most common form of genetic variation between genomes of different people is the single nucleotide polymorphism (SNP, pronounced "snip") at which a single DNA base takes on two common variants *alleles* in a population. While in rare cases more than two of the four DNA bases (`A`, `T`, `C` and `G`) are commonly found at a single genomic site, these are generally excluded from analysis. A chromosome of any single individual can then be modeled as a binary string of SNP alleles, in which the i^{th} bit is zero if that individual has the more common (major) allele at the i^{th} SNP locus and is one if that individual has the less common (minor) allele. Nearly 18 million such SNPs are now known in the human genome [22,15,9,16] and there is great interest in them for studies of human ancestry (cf., [23]) and as markers for statistical tests of association between genotype and phenotype (cf., [10]).

SNP alleles are not independent of one another but rather tend to be strongly correlated when nearby on the genome. These correlations occur as a side-effect of the way in which we inherent DNA from our parents. Humans are *diploid* organisms, meaning that most of our DNA is organized in pairs of chromosomes of which we inherent one copy from the mother and one from the father. The copy inherited from a single parent is not identical to either of that parent's copies, though, but rather is assembled through a process called *recombination* (or *crossing-over*) that leads to a concentenation of pieces of both of the parent's chromosome copies. For instance, if a parent's haplotypes are

Giuseppe Lancia, Romeo Rizzi, Russell Schwartz: Tiling Binary Matrices in Haplotyping: Complexity, Models and Algorithms, pp. 89–102 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

```
C C G A G A A C C A T G C G
a c c g g a t g g a a t c g
```

a haplotype obtainable by cross-over could be

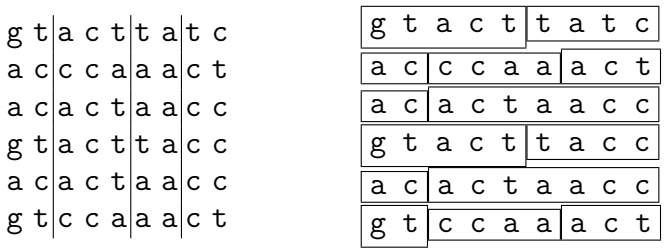
```
C C G A G a t g g a a G C G
```

As a result, when a new variation first appears in the genome through random mutation, that variation will remain strongly correlated with nearby SNPs for many generations, but will rapidly lose any detectable correlation with more physically distant SNPS. The result of this process is that when one examines patterns of variation across the human genome, one observes many strongly conserved sub-strings, called *haplotypes*, which are believed to represent sets of alleles that were found together in a subset of early human ancestors and which have largely escaped being broken up by recombination.

The haplotype structure of the genome is not merely an intellectual curiosity, but is the focus of intensive practical efforts to harness it for use in genetic association studies. Association studies, in which one seeks SNPs statistically associated with some phenotype (e.g., a disease), are hampered by the fact that the large number of SNPs means that corrections for multiple hypothesis obscure all but the strongest associations. It is hoped that testing for association with haplotypes instead of genotypes [6,21,1] will mitigate this problem by allowing one to identify any real associations in the data while performing many fewer tests. Several major studies are underway to examine haplotypes across human populations [25,26,11,12] for this purpose. These haplotypes also provide important information for applications in inferring ancestry and population substructure in human populations.

Attempts to analyze and use these data have led to several different approaches to mathematically model haplotype structure in ways that will be amenable to model inference and application in association study design and other contexts. At one extreme are "haplotype block" models [8], which assume that the genome can be decomposed into short regions (blocks) and that all haplotypes are broken at the boundaries of these blocks. The block model makes the simplifying assumption that cross-over has repeatedly occurred at the same boundaries. Block models are amenable to efficient computational inference [28] but at the expense of obscuring some information on correlations across block boundaries. While there are many roughly similar ways of optimizing for block boundaries (c.f., [20,28,27,13]), they appear to give relatively poor agreement between measures, population groups, or even sub-samples of a single population [20]. At the opposite extreme are models allowing for haplotypes to be broken arbitrarily within any given individual chromosome, effectively treating the genome as a Markov model in which each chromosome represents a unique path between a set of ancestral chromsomes [19,7]. These general models can more accurately capture true haplotype structure, but at the expense of being much more difficult to learn reliably and to apply to subsequent optimizations. A compromise between these two extremes is the *motif model* [17] (or the independently developed *dictionary model* [2]), which models each chromosome as a concentenation of a set of conserved DNA segments, called *motifs* or *tiles* but without the assumption that block model assumption that boundaries between conserved segments are shared across the population. Several studies have shown that these models are more effective than raw SNPs or block-based haplotypes for association testing [5,3] and for several associated optimization problems [18].

The following are examples of block decomposition (left) and motif decomposition (right) of six haplotypes:



A motif model, like a block model, can nonetheless be defined in many different ways depending on the criteria for which one optimizes the fit of motifs to observed haplotype data. The motif model was originally implemented by Schwartz using a heuristic method approximately optimizing for a likelihood model [17], and has since been studied using other probabilistic models [2] and minimum description length (MDL) models [14,24]. An obvious metric, with some practical motivation for the multiple hypothesis testing problem in association testing, is parsimony [17]: minimizing the total number of tiles needed to explain a data set. For instance, in the above example on the right, the given explanation consists of 8 tiles. Note that this is not the minimum tiling. In fact, a trivial tiling in which each row is a tile by itself has value 6 (if, on the other hand, we introduce restrictions on the minimum and/or maximum length allowed for a tile, the trivial tiling may not be feasible). The parsimony metric has, however, only been used heuristically [5]. There has been neither any efficient algorithm for this problem or any proof of its hardness.

In this paper we prove that this problem is APX-hard. We then proceed to describe an ILP formulation which can be used for its exact solution. The formulation has an exponential number of variables, but its LP relaxation can be solved in polynomial time by column-generation techniques. We also describe an alternative, but equivalent, polynomial-size ILP formulation based on a reduction to a multicommodity flow problem. We finally give an exact, dynamic programming, polynomial algorithm for the parsimony version of the block model problem (i.e., find a decomposition of the matrix in blocks so that the total number of tiles that the decomposition defines is minimum).

2 The problem

We are given a binary $m \times n$ matrix M. A *tile* t is specified by a first starting column $f(t)$, an ending column $e(t)$, and a string $s(t)$ of length $e(t) - f(t) + 1$. A tile t *applies to* (or is compatible with) any row M_i of M such that

$$s(t) = M[i, f(t)] \cdots M[i, e(t)]\,. \tag{1}$$

For any row M_i, let us denote by $\mathcal{T}(i)$ the set of tiles that apply to it. Furthermore, let $\mathcal{T} := \cup_i \mathcal{T}(i)$ the set of tiles which apply to some row of M. Notice that each triple r, c_f, c_e, with $1 \leq r \leq m$ and $1 \leq c_f \leq c_e \leq n$ identifies a unique tile in $\mathcal{T}$, namely the tile t for which $f(t) = c_f$, $e(t) = c_e$ and $s(t) = M[r, c_f] \cdots M[r, c_e]$. We denote this tile as $< r, c_f, c_e >$. Notice that it is possible for different triples $< r, c_f, c_e >$ to identify the same tile in $\mathcal{T}$, as long as they refer to different rows. (For example, $< 1, 1, 3 >$ and $< 4, 1, 3 >$ identify the same tile in the binary matrix M displayed in Fig. 1). Furthermore, $\mathcal{T}(r) = \{< r, c_f, c_e > \mid 1 \leq c_f \leq c_e \leq n\}$.

We say that a set $\hat{T}$ of tiles *covers* a row M_i of M if there exists a subset $T_i = \{t_1, \ldots, t_k\} \subseteq \hat{T}$, with $f(t_1) = 1$, $f(t_i) = e(t_{i-1}) + 1$ for $i = 2, \ldots, k$, and $e(t_k) = n$, such that

$$M_i = s(t_1) \cdot s(t_2) \cdots s(t_k)\,. \tag{2}$$

A *tiling* of M is a set $\hat{T}$ of tiles which covers every row of M.

In this paper we study the problem Tile where, given a binary matrix M as input, we seek for a tiling $\hat{T}$ of M with the minimum possible number of tiles. In Figure 1 we show an example of three different tilings for a same binary matrix.

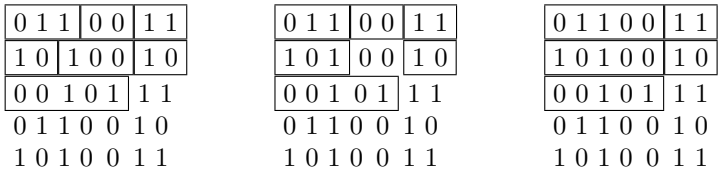


Figure 1. Three different tilings of a same binary matrix: one of size 7 (on the left), one of size 6 (in the middle), and an optimal tiling of size 5 (on the right).

Notice that, for any $m \times n$ input binary matrix M, the optimal value for problem Tile satisfies $OPT \leq \min\{m, 2n\}$, as both the whole rows of M and the single entries of M are considered as valid tiles.

3 Problem complexity

In this section we prove that the problem Tile is APX-hard.

The proof is split in two parts. First, in Subsection 3.1, we prove that Tile is APX-hard when matrices over general alphabets are considered. Next, in Subsection 3.2, we show that allowing non-binary matrices does not significantly affect the approximability of problem Tile.

We close this section by listing a few elementary facts which are useful both in establishing the reductions here proposed and also as a first aid in algorithmically managing the problem.

Fact 1. *If two rows are identical then we can remove one of them.*

Fact 2. *When the matrix is binary, then flipping the values in one column does not change the problem.*

Fact 3. *If two consecutive columns are identical (possibly after inversion of one of the two, in case of binary matrices) then we can remove one of them.*

Fact 4. *The problem can be solved in poly-time when the number of rows or the number of columns is bounded by a constant.*

3.1 APX-hardness of Tile in the general non-binary case

In this subsection, we prove that Tile is APX-hard for general non-binary matrices. This lemma is at the core of the result given in the next subsection (the APX-hardness of Tile for binary matrices) and is obtained by reducing Node-Cover

on cubic graphs to TILE. Explicit values of $\varepsilon > 0$ such that NODE-COVER on cubic graphs admits no $(1+\epsilon)$-approximation algorithm unless P=NP are given in [4].

Assume therefore to be given a cubic graph $G = (V, E)$ as instance of NODE-COVER. Let $m := |E|$ and $n := |V|$[1] Clearly, $m = \frac{3}{2}n$ since G is cubic. We assume the nodes in V to be labeled with the first naturals $0, 1, 2, \ldots, n-1$. In other words, $V = \mathbb{N}_n$. We can hence speak of the *small endnode* $s(e)$ and of the *big endnode* $b(e)$ for each edge $e \in E$. When we say that $e = uv$ is an edge in E we are implicitly assuming that $u < v$, that is, $u = s(e)$ and $v = b(e)$. We let $E = \{e_0, e_1, \ldots, e_{m-1}\}$.

We construct a matrix M with $\hat{n} := 3m$ columns and $\hat{m} := 4m + n + 2\hat{n}$ rows. The rows and columns of M are numbered starting from 0. All the entries of M are symbols from the alphabet $\Sigma := \{A, B, X, \sigma_1, \sigma_2\}$. For each $i, j \in \mathbb{N}_{\hat{n}}$ with $i \neq j$, let $M[4m+n+i, i] = M[4m+n+\hat{n}+i, j] = A$ and $M[4m+n+i, j] = M[4m+n+\hat{n}+i, i] = B$. That is, the last (second last) $\hat{n}$ rows of M are obtained from an $\hat{n} \times \hat{n}$ identity matrix by replacing each 0 with an A (respectively, with a B) and each 1 with a B (respectively, with an A). For each $i \in \mathbb{N}_n$, row i is associated to the node i of G and, for each $c \in \mathbb{N}_m$ and $t = 0, 1, 2$, the value of $M[i, 3c+t]$ is defined as follows.

$$M[i, 3c+t] = \begin{cases} \sigma_1 \text{ if } t = 0 \text{ and } i = s(e_c), \\ \sigma_1 \text{ if } t = 2 \text{ and } i = b(e_c), \\ A \text{ otherwise.} \end{cases}$$

Finally, for each $j \in \mathbb{N}_m$, rows $n+4j, n+4j+1, n+4j+2, n+4j+3$ are associated to the edge e_j of G and, for each $c \in \mathbb{N}_m$, $t = 0, 1, 2$ and $k = 0, 1, 2, 3$, the value of $M[n+4j+k, 3c+t)]$ is defined as follows.

$$M[n+4j+k, 3c+t)] = \begin{cases} X \text{ if } j = c \text{ and } t = 1, \\ \sigma_1 \text{ if } j = c \text{ and } (k,t) \in \{(0,0), (0,2), (1,0), (2,2)\}, \\ \sigma_2 \text{ if } j = c \text{ and } (k,t) \in \{(1,2), (2,0), (3,0), (3,2)\}, \\ A \text{ otherwise.} \end{cases}$$

Lemma 5. *Let X be a node cover of G. Then there exists a valid tiling T for M such that $|T| = 2\hat{n} + 4m + |X|$.*

Proof. Remember that each tile t is specified by a triple $(f(t), e(t), s(t))$ where $f(t)$ is the index of the first column, $e(t)$ is the index of the last column, and $s \in \Sigma^{e(t)-f(t)+1}$. We construct T in three phases. First, we place in T all the $\hat{n}$ tiles in the set $A_T := \{(i, i, A) : i \in \mathbb{N}_{\hat{n}}\}$ and all the $\hat{n}$ tiles in the set $B_T := \{(i, i, B) : i \in \mathbb{N}_{\hat{n}}\}$. Notice that the last $2\hat{n}$ rows of M are already covered by the tiles in $A_T \cup B_T$. Next, for each $e_j = uv \in E$, with $u < v$, we have two possible cases. If $u \in X$, then we add to T the 4 tiles $(3j, 3j+1, \sigma_1 X)$, $(3j, 3j+1, \sigma_2 X)$, $(3j+2, 3j+2, \sigma_1)$, $(3j+2, 3j+2, \sigma_2)$. Otherwise, if $u \notin X$, then we add to T the 4 tiles $(3j, 3j, \sigma_1)$, $(3j, 3j, \sigma_2)$, $(3j+1, 3j+2, X\sigma_1)$, $(3j+1, 3j+2, X\sigma_2)$. Notice here that, in either case, these 4 tiles plus the tiles in A_T suffice in covering the four rows $n+4j, n+4j+1, n+4j+2, n+4j+3$. Finally, for each $i \in X$, the whole row i of matrix M is placed as a tile in T. Notice that $|T| = 2\hat{n} + 4m + |X|$. It remains to check that, for each $i \in \mathbb{N}_n$, the i-th row of M is also covered by T. To see this, we distinguish between two cases: If $i \in X$, then row i appears in T as a tile. Otherwise, if $i \notin X$ and since X is a node cover of G,

[1] Notice that, within this section, m and n do not denote the number of rows and columns of the tiling matrix, but the number of edges and vertices of G.

then, for every edge $e_j = iu \in E$ (respectively, for every edge $e_j = ui \in E$) we have that $u \in X$ and hence the tile $(3j, 3j, \sigma_1)$ has been placed in T (respectively, the tile $(3j+2, 3j+2, \sigma_1)$ has been placed in T). Notice that row i is covered by these tiles (with $e_j \ni i$) plus the tiles in A_T. □

Lemma 6. *Let T be a feasible tiling for M. Then G admits a node cover X with $|X| = |T| - 2\hat{n} - 4m$.*

Proof. As in the proof of Lemma 5, each tile t is specified by a triple $(f(t), e(t), s(t))$, and $A_T := \{(i, i, A) : i \in \mathbb{N}_{\hat{n}}\}$, and $B_T := \{(i, i, B) : i \in \mathbb{N}_{\hat{n}}\}$. Let M_H (respectively, M_L) be the matrix comprising the first $4m + n$ (respectively, the last $2\hat{n}$) rows of matrix M. In other words, $M = \begin{pmatrix} M_H \\ M_L \end{pmatrix}$. For $i \in \{H, L\}$, let T_i be the set of tiles in T which are compatible with some row in M_i. Notice that there can be some tile $t \in T$ which belongs both to T_H and to T_L. However, for any such tile t, we have both that $s(t) \in \{A, X, \sigma_1, \sigma_2\}^*$, since $t \in T_H$, and that $s(t) \in \{A, B\}^*$, since $t \in T_L$. Indeed, all entries of M_H are in $\{A, X, \sigma_1, \sigma_2\}$ and all entries of M_L are in $\{A, B\}$. It follows that $s(t) \in \{A\}^*$ for each $t \in T_H \cap T_L$. Notice also that $A_T \cup B_T$ would be a tiling for M_L with $|A_T \cup B_T| = 2\hat{n}$. At the same time, $|T_L| \geq 2\hat{n}$ by Lemma 7 here below. These facts imply that we can always assume that $A_T \cup B_T \subseteq T$. Indeed, a tile $t \in T$ with $s(t) \in \{A\}^*$ which might possibly help in covering some rows of M_H can always be substituted with tiles in A_T.

Consider now the four rows $n+4j, n+4j+1, n+4j+2, n+4j+3$ associated with a generic edge e_j, $j \in \mathbb{N}_m$. Let $T(e_j)$ be the set of the tiles in $T \setminus A_T$ which are compatible with some of the above four rows. Notice that $|T(e_j)| \geq 4$. Notice furthermore that $T(e_p) \cap T(e_q) = \emptyset$ whenever $e_p, e_q \in E$ with $e_p \neq e_q$. Let $J_T := \{e_j : |T(e_j)| \geq 5\}$. We call a tiling T of M *standard* if $J_T = \emptyset$. We now show how to produce a standard tiling $\tilde{T}$ with $|\tilde{T}| \leq |T|$. To do so, it suffices to show how to obtain a tiling T' of M with $|T'| \leq |T|$ and such that $J_{T'}$ is strictly contained in J_T, whenever $J_T \neq \emptyset$. Indeed, where $|T(e_j)| \geq 5$, then let T' be obtained from T by removing all tiles in $T(e_j)$ and by adding the 5 tiles $t_1 = (3j, 3j+1, \sigma_1 X)$, $t_2 = (3j, 3j+1, \sigma_2 X)$, $t_3 = (3j+2, 3j+2, \sigma_1)$, $t_4 = (3j+2, 3j+2, \sigma_2)$ and $t_5 = (0, 3\hat{n} - 1, M_i)$ where $i = s(e_j)$ and M_i is the i-th row of M, i.e. the row of M associated with node i. Notice that T' is indeed a tiling of M, and indeed $|T'| \leq |T|$. Moreover, $T'(e_j) = \{t_1, t_2, t_3, t_4\}$, whence $J_{T'} \subset J_T$.

We hence assume T is standard, that is, $|T(e_j)| = 4$ for each $e_j \in E$. Since $A_T \subset T$, we can actually assume that either $T(e_j)$ comprises precisely the 4 tiles $(3j, 3j+1, \sigma_1 X)$, $(3j, 3j+1, \sigma_2 X)$, $(3j+2, 3j+2, \sigma_1)$, and $(3j+2, 3j+2, \sigma_2)$, or $T(e_j)$ comprises precisely the 4 tiles $(3j, 3j, \sigma_1)$, $(3j, 3j, \sigma_2)$, $(3j+1, 3j+2, X\sigma_1)$, and $(3j+1, 3j+2, X\sigma_2)$. For $i \in \mathbb{N}_n$, let $T(i)$ be the set of those tiles in $T \setminus A_T \setminus \cup_{e_j \in E} T(e_j)$ which are compatible with the i-th row of M. Notice that $T(i_1) \cap T(i_2) = \emptyset$ for each $i_1 \neq i_2$ with $i_1, i_2 \in \mathbb{N}_n$. Let now X be the set of those $i \in \mathbb{N}_n$ such that $T(i) \neq \emptyset$. Notice that X is a node cover of G. Finally, $|X| \leq |T| - 2\hat{n} - 4m$ is a consequence of the fact that the $T(e_j)$'s are disjoint sets of tiles and the $T(i)$'s are disjoint sets of tiles. □

We say that a set of tiles T *weakly covers* a matrix M if for every entry $M[i, j]$ of M there exists a tile t in T that is compatible with row i of M and such that $f(t) \leq j \leq e(t)$.

Lemma 7. *Let $M_{A/B}$ (respectively, $M_{B/A}$) be the $\hat{n} \times \hat{n}$ matrix whose entries are all B (respectively, all A) except for the entries on the diagonal which are all A*

(respectively, all B). Let $M_L = \begin{pmatrix} M_{B/A} \\ M_{A/B} \end{pmatrix}$ *and let* T_L *be a set of tiles which weakly covers* M_L*. Then* $|T_L| \geq 2\hat{n}$*.*

Proof. We prove a stronger claim: Let $\Sigma = \{A, B, C\}$. Let $N_{A/B}$ (respectively, $N_{B/A}$) be the $\hat{n} \times \hat{n}$ matrix obtained from matrix $M_{A/B}$ (respectively, $M_{B/A}$) by replacing with C all the entries below the main diagonal. Let $T_C := \{(i, i+1, C) : i \in \mathbb{N}_{\hat{n}}\}$. Let $N = \begin{pmatrix} N_{B/A} \\ N_{A/B} \end{pmatrix}$ and let T be a set of tiles such that $T \cup T_C$ weakly covers N. Then $|T| \geq 2\hat{n}$.

We prove this by induction on $\hat{n}$. Among the minimum-cardinality sets of tiles T such that $T \cup T_C$ weakly covers N, let T^* be one minimizing $\sum_{t \in T} |s(t)|$. Notice that $(\hat{n}-1, \hat{n}-1, B)$ and $(\hat{n}-1, \hat{n}-1, A)$ both belong to T^*. Indeed, where t is the tile in T compatible with the row $\hat{n}-1$ of N and with $e(t) = \hat{n}-1$, then $t = (\hat{n}-1-i, \hat{n}-1, C^i B)$ for some $i \in \mathbb{N}_{\hat{n}}\}$. Notice however that any tile t of this form can always be substituted by tile $(\hat{n}-1, \hat{n}-1, B)$, plus some tiles in T_C. The same argument also shows that $(\hat{n}-1, \hat{n}-1, A) \in T^*$. Notice now that $(T^* \setminus \{(\hat{n}-1, \hat{n}-1, B), (\hat{n}-1, \hat{n}-1, A)\}) \cup T_C$ weakly covers N', the matrix obtained from N by dropping the last column and by dropping the rows $\hat{n}-1$ and $2\hat{n}-1$. Notice that matrix N' is of the same form as matrix N, but with $\hat{n}' = \hat{n}-1$. Therefore, by induction, $|T^*| \geq 2 + 2(\hat{n}-1) = 2\hat{n}$. □

Theorem 8. *When we allow for general, possibly non-binary matrices, then the* Tile *problem is APX-hard.*

Proof. We proceed as follows: We assume to be given a $(1+\epsilon)$-approximation algorithm A for Tile and design a $(1+31\,\epsilon)$-approximation algorithm for Node-Cover which rests on algorithm A as a subroutine. The APX-harness of Tile then follows from the APX-harness of Node-Cover.

After receiving in input a cubic graph G, we construct the matrix M as described above. Assume the minimum node cover of G has size *opt*. Clearly, $opt \geq \frac{m}{3}$ since G is cubic. Moreover, by Lemma 5, there exists a tiling T_{opt} covering M with $|T_{opt}| = 2\hat{n} + 4m + opt = 10m + opt$. By running the $(1+\epsilon)$-approximation algorithm for Tile we are hence guaranteed to find a solution T_{apx} with $|T_{apx}| \leq (10\,m + opt)(1+\epsilon)$. And Lemma 6 (whose proof can be easily converted into a poly-time algorithm) shows how, starting from this tiling T_{apx}, one can obtain a node cover X of G of size at most

$$|X| \leq (10\,m + opt)(1+\epsilon) - 10\,m \leq 10\,\epsilon\,m + opt + \epsilon\,opt \leq 30\,\epsilon\,opt + opt + \epsilon\,opt \leq (1 + 31\,\epsilon)opt.$$

□

3.2 The power of the binary case

In this subsection, we show that allowing non-binary matrices does not affect the approximability of the problem Tile. Formally stated, we prove the following result.

Lemma 9. *There exists an objective function preserving reduction from the* Tile *problem on general non-binary matrices to the* Tile *problem on binary matrices.*

Notice that, combining Lemma 9 here above with Theorem 8 from the previous section, we obtain the following result.

Theorem 10. *Even when restricted to binary input matrices, problem* Tile *is APX-hard.*

Assume the entries of the input matrix M are taken from the alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$, where $k := |\Sigma|$. We can clearly assume that $k \leq m \cdot n$, where m and n are the number of rows and columns of matrix M. The objective function preserving reduction we are going to propose can be conveniently described as the composition of two objective function preserving transformations to be applied in series. First, an $m \times nm$ matrix M_σ over Σ is obtained from M by echoing each single column of M precisely m times. Notice that, by Fact 3, this does not affect the objective function value. Next and last, M_b is the $m \times nmk$ binary matrix obtained from M_σ by replacing each entry σ_i of M_σ with a row vector of i zero's followed by a row vector of $k - i$ one's. So, where M had m rows and n columns, both numbered starting from 0, then M_b has m rows and nmk columns, and

$$M_b[i,j] = \begin{cases} 0 \text{ if } M[i, j \,.\text{div.}\, km] = \sigma_p \text{ and } p > j \,.\text{mod.}\, k, \\ 1 \text{ otherwise.} \end{cases}$$

It should be clear that a tiling of M directly translates into a tiling of M_b involving the same number of tiles. Indeed, a feasible tiling for M gets converted into a feasible tiling for M_b if all tiles get stretched by a factor of mk. In the tiling of M_b obtained in this way, all tiles t have $f(t)$ which is a multiple of mk and $e(t) \equiv_k k - 1$. We call such a tiling of M_b *standard*.

Conversely, it is also clear that to any standard tiling of M_b corresponds a tiling of M involving the same number of tiles. Therefore, in order to prove Lemma 9, we only need to prove the following lemma.

Lemma 11. *Given any tiling T of M_b, we can produce in poly-time a standard tiling T' of M_b with $|T'| \leq |T|$.*

Proof. Clearly, since M_b has m rows, there is no difficulty in producing a standard tiling of M_b of size m. We can therefore assume that $|T| < m$. We also assume that T is a minimal tiling of M_b, that is, $T \setminus \{t\}$ is also a feasible tiling of M_b for no $t \in T$. Fix attention on any $c_1 = 0, 1, 2, \ldots, n-1$. Since, $|\{f(t) \,.\text{div.}\, k : t \in T\}| \leq |T| < m$, then there exists a $c_2 = c_2(c_1) \in \{0, 1, 2, \ldots, m-1\}$ such that $f(t) \,.\text{div.}\, k \neq c_1 m + c_2$ holds for every $t \in T$. This means that no tile starts within the k column positions of M_b corresponding to position $c_1 m + c_2$ of M_σ. More formally, for no $t \in T$ we have that $(c_1 m + c_2)k \leq f(t) < (c_1 m + c_2 + 1)k$. From this, and by the minimality of T, it also follows that for no $t \in T$ we have that $(c_1 m + c_2)k - 1 \leq e(t) < (c_1 m + c_2 + 1)k - 1$. Based on these facts, we can massage the tiles in T as follows.

1. **left extension** Let t be any tile in T with $f(t) \leq (c_1 m + c_2)k \leq e(t)$. If $f(t) > c_1 mk$, then t is replaced with a tile t' with $f(t') = c_1 mk$, $e(t') = e(t)$, $t'[p] = t[p]$ for each $p = f(t), f(t)+1, \ldots, e(t)$, and $t'[p] = t[(c_1 m + c_2)k + (p \,.\text{mod.}\, k)]$ for each $p < f(t)$ with $p \geq f(t')$.
2. **right extension** Let t be any tile in T with $f(t) \leq (c_1 m + c_2)k \leq e(t)$. If $e(t) < (c_1 + 1)mk - 1$, then t is replaced with a tile t' with $f(t') = f(t)$, $e(t') = (c_1 + 1)mk - 1$, $t'[p] = t[p]$ for each $p = f(t), f(t) + 1, \ldots, e(t)$, and $t'[p] = t[(c_1 m + c_2)k + (p \,.\text{mod.}\, k)]$ for each $p > e(t)$ with $p \leq e(t')$.

3. right trim Let t be any tile in T with $c_1mk \leq e(t) < (c_1m + c_2)k$. Then t is replaced with a tile t' with $f(t') = f(t)$, $e(t') = c_1mk - 1$, and $t'[p] = t[p]$ for each $p = f(t'), f(t) + 1, \ldots, e(t')$.

4. left trim Let t be any tile in T with $(c_1m + c_2 + 1)k \leq f(t) < (c_1 + 1)mk$. Then t is replaced with a tile t' with $e(t') = e(t)$, $f(t') = (c_1 + 1)mk$, and $t'[p] = t[p]$ for each $p = f(t'), f(t) + 1, \ldots, e(t')$.

Clearly, no one of the above four operations can possibly increase $|T|$. Furthermore, it can be checked that if a row r is covered by some sequence of tiles in T, then, after each one of the above 4 operations has been performed, row r can still be covered by a suitable sequence of tiles. Indeed, the new sequence of tiles can be obtained from the original one by performing the following operations:

(1) discard all tiles t with $f(t) \geq c_1mk$ and $e(t) < (c_1m + c_2)k$;

(2) discard all tiles t with $f(t)$ with $e(t) < (c_1 + 1)mk$ and $f(t) \geq (c_1m + c_2)k$;

(3) retain all other tiles; on each one of these remaining tiles, apply each one of the above four operations.

Notice that, after each one of the four operations above has been performed, no tile t can have $c_1mk < f(t) < (c_1 + 1)mk$ or $c_1mk \leq e(t) < (c_1 + 1)mk - 1$. Furthermore, if $f(t)$ (respectively, $e(t)$) has been affected by the above operations, then, after the operations have taken place, $f(t)$ (respectively, $e(t)+1$) is a multiple of mk. It follows that after the above 4 steps have been executed for each $c_1 = 0, 1, 2, \ldots, n$, and for the corresponding $c_2 = c_2(c_1)$, then T has become standard. □

4 ILP formulations and DP

Exponential formulation

As defined in Section 2, let m be the number of rows and n be the number of columns of the input matrix M for which we seek an optimal tiling.

In our first ILP formulation, we introduce a binary variable x_t for each possible tile $t \in \mathcal{T}$, and further, for every $i = 1, 2, \ldots, m$, we introduce a binary variable y_T for each minimal set of tiles which covers row i. For the porpouse of notation, we denote by $\mathcal{M}(i)$ the family of the minimal sets of tiles which cover row i. Notice that a set of tiles belongs to $\mathcal{M}(i)$ if and only if is contained in $\mathcal{T}(i)$ and has the form $\{t_1, \ldots, t_k\}$ with $f(t_1) = 1$, $f(t_i) = e(t_{i-1}) + 1$ for $i = 2, \ldots, k$, and $e(t_k) = n$.

We have

$$\min \sum_{t \in \mathcal{T}} x_t \tag{3}$$

$$\sum_{T \in \mathcal{M}(i)} y_T = 1 \qquad \forall i = 1, \ldots, m \tag{4}$$

$$\sum_{T \in \mathcal{M}(i) \,|\, t \in T} y_T \leq x_t \qquad \forall i = 1, \ldots, m \quad \forall t \in \mathcal{T}(i) \tag{5}$$

$$x_t, y_T \in \{0, 1\} \qquad \forall t \in \mathcal{T}, T \subseteq \cup_i \mathcal{M}(i) \tag{6}$$

Note that, for each i, it is $|\mathcal{T}(i)| = n(n+1)/2$ (we have to decide the first starting and the ending column), while $|\mathcal{M}(i)| = 2^{n-1}$ (we have to decide which of the first $n - 1$ columns are ending columns).

Thus, the above model has an exponential number of y variables. However, the LP relaxation can still be solved in polynomial time provided we can show how to solve the *pricing* problem for the y variables in polynomial time. The resulting approach is called column generation. The idea is to have all x variables in the model, and only a subset of the y variables. Then, given an optimal solution to the current LP, we see if there is any missing y variable that should be priced-in (i.e., added to the current variables).

Let $\gamma_1, \ldots, \gamma_m$ be the dual variables associated with constraints (4) and let λ_t^i, for $i = 1, \ldots, m$ and $t \in \mathcal{T}(i)$, be the dual variables associated with constraints (5).

To each primal variable y_T corresponds an inequality in the dual LP. The variable has negative reduced cost if and only if the corresponding dual constraints is violated by the current optimal dual solution. Assume T is a set in $\mathcal{M}(i)$. Then, the corresponding dual inequality for T is

$$\gamma_i - \sum_{t \in T} \lambda_t^i \leq 0 \tag{7}$$

If we consider λ_t^i to be the cost of tile t (relatively to a particular row i), and define $\lambda^i(T) := \sum_{t \in T} \lambda_t^i$, we have that the dual inequalities, for all $T \in \mathcal{M}(i)$, are of type

$$\lambda^i(T) \geq \gamma_i \tag{8}$$

A set of tiles violates the dual inequality if $\lambda^i(T) < \gamma_i$. If this happens, y_T should be added to the current set of primal variables. To identify a set which violates the dual inequality, it is enough to find the *smallest-cost* set. If $T^* \in \mathcal{M}(i)$ is such that $\lambda^i(T^*) = \min_{T \in \mathcal{M}(i)} \lambda^i(T)$, then, if $\lambda^i(T^*) < \gamma_i$ then y_{T^*} should be added to the LP variables, otherwise, no y_T variables, with $T \in \mathcal{M}(i)$ should be added to the LP. We should repeat this reasoning for all $i = 1, \ldots, m$.

Let us consider then the following problem:

- given i and costs λ_t^i, for $t \in \mathcal{T}(i)$, find T^* in $\mathcal{M}(i)$ with minimum λ-cost.

For each $1 \leq u \leq v \leq n$, denote in short $\Lambda(u, v)$ the value λ_t^i, where $t =< i, u, v >$.

We consider the following dynamic program. Denote by $V(r)$ the optimal (minimum) λ-cost of fragmenting row i in consecutive tiles, up to position r. We are interested in $V(n)$. We have the recurrence:

$$V(r) = \min_{p=\max\{1, r-l_M+1\}}^{r-l_m+1} (V(p-1) + \Lambda(p, r)) \tag{9}$$

where l_m is the minimum possible length of a tile, and l_M is the maximum possible length of a tile. We have $\lambda^i(T^*) = V(n)$. Base case is $V(j) = 0$, for $j \leq 0$, and $V(j) = +\infty$ for $0 < j < l_m$.

Polynomial-size formulation

Here we consider an alternative ILP formulation, which in fact yields the same bound as the previous one. The idea is to formulate the problem as a multicommodity flow problem. In principle, imagine to have a directed graph $G = (V, A)$, in which the vertices V are given by the column indexes, augmented with a dummy node $n + 1$, i.e., $V = 1, \ldots, n + 1$. The arcs are associated to the tiles. There is a directed arc for

each tile t. Assume $t =< i, u, v >$. Then, there is an arc $a_t = (u, v+1)$. Note that there can be parallel arcs, and, for each $1 \leq u \leq v \leq n+1$ there is at least one arc from u to v.

Now, for each commodity $i = 1, \ldots, m$, we want to send a unit of flow out of node 1, through the network as far as it can go (i.e., until it reaches node $n+1$). Each time an arc a_t is used by the flow f_i, for commodity i, it means that the tile t is used in the solution to cover row i.

We can associate flow variables to the arcs, and have flow conservation constraints. Furthermore, the activation variables for the tiles (i.e., the x_t variables) provide capacities for each arc (i.e., x_t is the capacity of the arc a_t).

Instead of actually building the network, we now describe a formulation that achieves the exact same purpose, and "builds" the network only implicitly.

As before, there are variables x_t for each tile $t \in \mathcal{T}$. Furthermore, for each row i and indices $1 \leq a \leq b \leq n$, we have a variable z^i_{ab}. This variable represents the i-th flow along the arc $a_{<i,a,b>}$ (i.e., one of the parallel arcs between a and $b+1$) in G.

We get the following formulation:

$$\min \sum_{t \in \mathcal{T}} x_t \tag{10}$$

$$\sum_{r=1,\ldots,n} z^i_{1r} = 1 \qquad \forall i = 1, \ldots, m \tag{11}$$

$$\sum_{1 \leq j < r} z^i_{jr} = \sum_{r < j \leq n} z^i_{rj} \qquad \forall i = 1, \ldots, m \quad \forall 1 < r < n \tag{12}$$

$$z^i_{ab} \leq x_{<i,a,b>} \qquad \forall i = 1, \ldots, m \quad \forall 1 \leq a \leq b \leq n \tag{13}$$

$$x_{<i,a,b>}, z^i_{ab} \in \{0, 1\} \qquad \forall 1 \leq i \leq m,\ 1 \leq a \leq b \leq n \tag{14}$$

Constraints (12) are flow conservation, saying that in each non-final column, the unit of flow coming in must also go out. Constraints (13) put capacities on the arcs, saying that an arc corresponding to a tile not activated ($x_t = 0$) cannot be used by the flows. Note that the z variables could be just declared real, as, when in a feasible solution the x are integer, there is always a way to make the z integer as well.

This model has $m \times n \times (n+1)/2$ z-variables ($\simeq mn^2/2$) and $|\mathcal{T}|$ x-variables. As for the constraints, there are m flow-out constraints, $m \times (n-2)$ conservation constraints and $m \times n(n+1)/2$ capacity constraints, for a total of $\simeq mn^2/2$ constraints.

Theorem 12. *For each solution (x, y) of the exponential formulation there is a solution (x, z) of the polynomial formulation, which achieves the exact same value, and vice versa.*

Proof. (Sketch) Just use the flow-decomposition theorem. Given a solution (x, y) each admissible set of tiles T for row i corresponds to (the arcs of) a path starting at vertex 1 and ending at vertex $n+1$. Sending along this path y_T units of flow, we get in the end a unit of flow out of 1. Vice versa, given a solution (x, z), each z^i identifies a flow of value 1 out of 1. This flow can be decomposed into paths, and each path identifies an admissible set T. The decomposition is based on finding a minimum-flow arc (say of value δ) and tracing it back to the source and to the sink, thus identifying a path. Then we subtract δ from the flow on the arcs of the path and iterate.

Dynamic Programming for Tiling into Strips

Let M be a binary $m \times n$ matrix. For each $i, j \in \{1, 2, \ldots, n\}$ with $i \leq j$, we denote by $M(i, j)$ the submatrix obtained from M by dropping all columns except those with index p with $i \leq p \leq j$, and let $M(i, -) = M(i, n)$ be obtained by removing only the first $i - 1$ columns.

A tiling T of M is called a *striping* of M if for every two tiles $t_1, t_2 \in T$ with $f(t_1) \leq e(t_2) \leq e(t_1)$ we have that $e(t_2) = e(t_1)$ and $f(t_2) = f(t_1)$.

In this section we show that the problem of finding a striping of minimum size can be solved in polynomial time by Dynamic Programming.

The *starting shadow* of a striping T is the set $F(T) := \{f(t) : t \in T\}$. Notice that $F(T)$ uniquely defines the striping T. Indeed, for every tile $t \in T$ we have that $f(t)$ and $e(t) + 1$ are two consecutive integers in $F(T)$, and, conversely, for every two consecutive integers f_1 and f_2 in $F(T)$, striping T contains precisely $variety(M(f_1, f_2 - 1))$ different tiles t with $f(t) = f_1$ and $e(t) = f_2 - 1$, where $Variety(M(i, j))$ are the equivalence classes over the rows of $M(i, j)$ under the identity relation, and $variety(M(i, j)) = |Variety(M(i, j))|$. In this section we show that the problem of finding a striping of minimum size can be solved by Dynamic Programming.

Denote by opt_i the minimum size of a striping for matrix $M(i, -)$. Then, since $M = M(1, -)$, the size of an optimum striping for M is given by opt_1. Moreover, $opt_n = variety(M(n, n))$, which amounts to the number of different symbols occurring in the last column of M (i.e. either 1 or 2). Finally, for $i = n - 1$ down to $i = 1$ we can iteratively compute opt_i by means of the recurrence

$$opt_i := \min_{j>i} opt_j + variety(M(i, j - 1)) .$$

We now introduce the data structures needed to efficienty compute all the values $variety(M(i, j))$ for $j \geq i$. Clearly, $variety(M(i, i))$ is the number of different symbols occurring in the column vector $M(i, i)$. We next show how $variety(M(i, j+1))$ can be computed from $variety(M(i, j))$ in $O(m)$ steps. Thanks to this, the total running time of the Dynamic Programming algorithm outlined above is clearly $O(mn^2)$. The idea is to store the partition of the rows of $M(i, j)$ associated to the identity equivalence relation as a vector $class$ of m entries. In each entry $class[r]$ the smallest index of a row identical to row r is reported. We now describe how the m-vector $class$ gets updated when going from $M(i, j)$ to $M(i, j + 1)$. This is done by using a second m-vector $newName$, initialized to all -1, and running the following algorithm.

```
for r := 1 to n,
  if M[r, j + 1] ≠ M[class[r], j + 1] then
    if newName[class[r]] = −1 then
        newName[class[r]] := r;
        class[r] := r;
    else class[r] := newName[class[r]].
```

Acknowledgments

R.S. was supported in this work by U.S. National Science Foundation award #0612099.

References

1. J. Akey, L. Jin, and M. Xiong: *Haplotypes versus single marker linkage disequilibrium tests: what do we gain?* European Journal of Human Genetics, 9 2001, pp. 291–300.
2. K. L. Ayers, C. Sabatti, and K. Lange: *Reconstructing ancestral haplotypes with a dictionary model.* Journal of Computational Biology, 13 2006, pp. 767–785.
3. K. L. Ayers, C. Sabatti, and K. Lange: *A dictionary model for haplotyping, genotype calling, and association testing.* Genetic Epidemiology, 31 2007, pp. 672–683.
4. P. Berman and M. Karpinski: *On some tighter inapproximability results*, ECCC report no. 29, University of Trier, 1998.
5. N. Castellana, K. Dhamdhere, S. Sridhar, and R. Schwartz: *Relaxing haplotype block models for association testing*, in Proc. Pacific Symposium on Biocomputing (PSB05), 2005.
6. N. H. Chapman and E. M. Wijsman: *Genome screens using linkage disequilibrium tests: optimal marker characteristics and feasibility.* American Journal of Human Genetics, 63 1998, pp. 1872–1885.
7. D. Falush, M. Stephens, and J. K. Pritchard: *Inference of population structure using multilocus genotype data: linked loci and correlated allele frequencies.* Genetics, 164 2003, pp. 1567–1587.
8. S. B. Gabriel, S. F. Schaffer, H. Nguyen, J. M. Moore, J. Roy, and *et al.*: *The structure of haplotype blocks in the human genome.* Science, 296 2001, pp. 2225–2229.
9. L. Helmuth: *Genome research: Map of the human genome 3.0.* Science, 293(5530) 2001, pp. 583–585.
10. J. N. Hirschhorn and M. J. Daly: *Genome-wide association studies for common diseases and complex traits.* Nature Reviews Genetics, 6 2005, pp. 95–108.
11. M. Jakobsson, S. W. Scholz, P. Scheet, J. R. Gibbs, J. M. VanLiere, and et al.: *Genotype, haplotype and copy-number variation in worldwide human populations.* Nature, 451 2008, pp. 998–1003.
12. J. Kaiser: *A plan to capture human diversity in 1000 genomes.* Science, 319 2008, p. 395.
13. G. Kimmel, R. Sharan, and R. Shamir: *Identifying blocks and sub-populations in noisy SNP data*, in Proceedings of the Third Workshop on Algorithms in Bioinformatics (WABI), 2003, pp. 303–319.
14. M. Koivisto et al.: *An MDL method for finding haplotype blocks and estimating the strength of haplotype block boundaries*, in Proceedings of the Pacific Symposium on Biocomputing (PSB), 2003, pp. 502–513.
15. E. Marshall: *Drug firms to create public database of genetic mutations.* Science Magazine, 284(5413) 1999, pp. 406–407.
16. E. W. Sayers, T. Barret, D. A. Benson, S. H. Bryant, K. Canese, and *et al.*: *Database resources of the National Center for Biotechnology Information.* Nucleic Acids Research, 37 2008, pp. D5–D15.
17. R. Schwartz: *Haplotype motifs: an algorithmic approach to locating evolutionarily conserved patterns in haploid sequences*, in Proceedings of the Second IEEE Computer Society Bioinformatics Conference, 2003, pp. 306–314.
18. R. Schwartz: *Algorithms for association study design using a generalized model of haplotype conservation*, in Proc. IEEE Computer Society Bioinformatics Conference, 2004, pp. 90–97.
19. R. Schwartz, A. Clark, and S. Istrail: *Methods for inferring block-wise ancestral history from haploid sequences: The haplotyping coloring problem*, in Proceedings of Annual Workshop on Algorithms in Bioinformatics (WABI), R. Guigo and D. Gusfield, eds., Lecture Notes in Computer Science, Springer, 2002.
20. R. Schwartz, B. Halldorsson, V. Bafna, A. G. Clark, and S. Istrail: *Robustness of inference of haplotype block structure.* Journal of Computational Biology, 10(1) 2003, pp. 13–20.
21. S. K. Service, D. W. Lang, N. B. Freimer, and L. A. Sandkuijl: *Linkage-disequilibrium mapping of disease genes by reconstruction of ancestral haplotypes in founder populations.* American Journal of Human Genetics, 64 1999, pp. 1728–1738.
22. S. T. Sherry, M.-H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin: *dbSNP:the NCBI database of genetic variation.* Nucleic Acids Research, 29 2001, pp. 308–311.
23. M. D. Shriver and R. A. Kittles: *Genetic ancestry and the search for personalized genetic histories.* Nature Reviews Genetics, 5 2004, pp. 611–618.

24. S. Sridhar, K. Dhamdhere, G. Blelloch, R. Ravi, and R. Schwartz: *Evaluation of the haplotype motif model using the principle of minimum description*, cmu computer science technical report cmu-cs-04-166, Carnegie Mellon University, Pittsburgh, USA, 2004.
25. The International HapMap Consortium: *The international HapMap project.* Nature, 426 2005, pp. 789–796.
26. The International HapMap Consortium: *A second generation human haplotype map of over 3.1 millions SNPs.* Nature, 449 2007, pp. 851–861.
27. K. Zhang, P. Calabrese, M. Nordborg, and F. Sun: *Haplotype block structure and its applications in association studies: power and study design.* The American Journal of Human Genetics, 71 2002, pp. 1836–1894.
28. K. Zhang, M. Deng, T. Chen, M. Watermanm, and F. Sun: *A dynamic programming algorithm for haplotype block partitioning.* PNAS, 99 2002, pp. 7335–7339.

Binary Image Compression via Monochromatic Pattern Substitution: Effectiveness and Scalability

Luigi Cinque[1], Sergio De Agostino[1], and Luca Lombardi[2]

[1] Computer Science Department, Sapienza University, 00198 Rome, Italy
[2] Computer Science Department, University of Pavia, 27100 Pavia, Italy

Abstract. We present a method for compressing binary images via monochromatic pattern substitution. Monochromatic rectangles inside the image are compressed by a variable length code. Such method has no relevant loss of compression effectiveness if the image is partitioned into up to a thousand blocks and each block is compressed independently. Therefore, it can be implemented in parallel on both small and large scale arrays of processors with distributed memory and no interconnections. We experimented the procedure with up to 32 processors of a 256 Intel Xeon 3.06 GHz processors machine (`avogadro.cilea.it`) on a test set of large topographic bi-level images. We obtained the expected speed-up of the compression and decompression times, achieving parallel running times about twenty times faster than the sequential ones. In the theoretical context of unbounded parallelism, we show experimentally that interprocessor communication is needed when we scale up the distributed system. It results that compression effectiveness has a bell-shaped behaviour which is again competitive with the sequential performance when the highest degree of parallelism is reached.

Keywords: lossless compression, binary image, distributed algorithm, scalability

1 Introduction

The best lossless image compressors are enabled by arithmetic encoders based on the model driven method [2]. The *model driven method* consists of two distinct and independent phases: *modeling* [17] and *coding* [16]. Arithmetic encoders are the best model driven compressors both for binary images (JBIG) [13] and for grey scale and color images (CALIC)[24], but they are often ruled out because they are too complex.

As far as the model driven method for grey scale and color image compression is concerned, the modeling phase consists of three components: the determination of the context of the next pixel, the prediction of the next pixel and a probabilistic model for the *prediction residual*, which is the value difference between the actual pixel and the predicted one. In the coding phase, the prediction residuals are encoded. The use of prediction residuals for grey scale and color image compression relies on the fact that most of the times there are minimal variations of color in the neighborood of one pixel. LOCO-I (JPEG-LS) [22] is the current lossless standard in low-complexity applications and involves Golomb-Rice codes [12], [15] rather than the arithmetic ones. A low-complexity application compressing 8x8 blocks of a grey-scale or color image by means of a header and a fixed-length code is PALIC [6], [7] which can be implemented on an arbitrarily large scale system at no communication cost. As explained in [7], parallel implementations of LOCO-I would require more sophisticated architectures than a simple array of processors. PALIC achieves 80 percent of the compression obtained with LOCO-I and is an extremely local procedure which is able to achieve a satisfying degree of compression by working independently

Luigi Cinque, Sergio De Agostino, Luca Lombardi: Binary Image Compression via Monochromatic Pattern Substitution: Effectiveness and Scalability, pp. 103–115 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

on different very small blocks. On the other hand, no local low-complexity binary image compressor has been designed so far. BLOCK MATCHING [18], [19] is the best low-complexity compressor for binary images and extends data compression via textual substitution to two-dimensional data by compressing sub-images rather than substrings [14], [20], achieving 80 percent of the compression obtained with JBIG. However, it does not work locally since it applies a generalized LZ1-type method with an unrestricted window.

In this paper, we present a method for compressing binary images via monochromatic pattern substitution. Monochromatic rectangles inside the image are compressed by a variable length code. Such monochromatic rectangles are detected by means of a *raster* scan (row by row). If the 4 x 4 subarray in position (i, j) of the image is monochromatic, then we compute the largest monochromatic rectangle in that position else we leave it uncompressed. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic rectangle or raw data. The procedure for computing the largest monochromatic rectangle with left upper corner in position (i, j) takes $O(M \log M)$ time, where M is the size of the rectangle. The positions covered by the detected rectangles are skipped in the linear scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. The analysis of the running time of this algorithm involves a *waste factor*, defined as the average number of matches covering the same pixel. We experimented that the waste factor is less than 2 on realistic image data. Therefore, the heuristic takes $O(n \log M)$ time. On the other hand, the decoding algorithm is linear. The compression effectiveness of this technique is about the same as the one of the rectangular block matching technique [19], which still requires $\Omega(n \log M)$ time. However, in practice it is about twice faster.

Compression via monochromatic pattern substitution by the variable length code presented here has no relevant loss of effectiveness if the image is partitioned into up to a thousand blocks and each block is compressed independently. Therefore, it can be implemented in parallel on both small and large scale arrays of processors with distributed memory and no interconnections. We experimented the procedure with up to 32 processors of a 256 Intel Xeon 3.06 GHz processors machine (`avogadro.cilea.it`) on a test set of large topographic bi-level images. We obtained the expected speed-up of the compression and decompression times, achieving parallel running times about twenty times faster than the sequential ones. Similar results were obtained in [10], [8], [9] on the same machine for the version of block matching compressing squares. However, although the square block matching technique [18] has a linear sequential time it has slower compression and decompression running times in practice. Moreover, it has a lower effectiveness and it is not scalable.

In the theoretical context of unbounded parallelism, interprocessor communication is needed when we scale up the distributed system. Parallel algorithms are shown in [1], [4], [5], [10], [9] for the rectangular block matching heuristic. In [5], a variable length coding technique similar to the one presented in this paper is described for the sequential implementation of the rectangular block matching method. The technique encodes bounded size two-dimensional patterns and has not been employed in the theoretical parallel implementations. We employed it in the experimental results of this paper on compression effectiveness since the bound to the pattern dimensions is large enough with respect to the size of realistic image data. It resulted that the compression effectiveness has a bell-shaped behaviour which is again competitive with the sequential performance when the highest degree of parallelism is reached.

Previous work on the square block matching heuristic is reported in section 2. Compression via monochromatic pattern substitution is described in section 3. In section 4, we present the parallel implementations and the experimental results on the speed-up. In section 5, we show how parallel implementations can be realized in the theoretical context of unbounded parallelism by means of interprocessor communication. Section 6 presents the experimental results relating scalability to compression effectiveness. Conclusions and future work are given in section 7.

2 Previous Work

As mentioned in the introduction, the square block matching procedure has been so far the only low-complexity lossless binary image compression technique implemented in parallel at no communication cost [10], [8], [9]. Such technique is a two-dimensional extension of Lempel-Ziv compression [14] and simple practical heuristics exist to implement Lempel-Ziv compression by means of hashing techniques [3], [11], [21], [23].

Image	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	76	39	19	11	6	3
2	81	40	23	11	5	3
3	78	39	24	12	6	3
4	79	44	24	11	5	3
5	77	38	22	10	5	4
Avg.	78.2	40	22.4	11	5.4	3.2

Figure 1. Compression times of the block matching procedure (cs.)

Image	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	43	22	11	6	4	2
2	44	22	12	7	3	2
3	43	22	15	7	4	2
4	43	30	12	7	3	2
5	41	32	15	6	3	2
Avg.	42.8	25.6	13	6.6	3.4	2

Figure 2. Decompression times of the block matching procedure (cs.)

The hashing technique used for the two-dimensional extension is even simpler [18]. The square matching compression procedure scans an m x m' image by a raster scan. A 64K table with one position for each possible 4x4 subarray is the only data structure used. All-zero and all-one squares are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square, a match, or raw data. When there is a match, the 4x4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current

$c = p_{r,j}$;
$r = i$;
$width = m'$;
$length = 0$;
$side1 = side2 = area = 0$;
repeat
    Let $p_{r,j} \cdots p_{r,j+\ell-1}$ be the longest string in (r, j) with color c and $\ell \leq width$;
    $length = length + 1$;
    $width = \ell$;
    $r = r + 1$;
    if $(length * width > area)$ {
        $area = length * width$;
        $side1 = length$;
        $side2 = width$;
    }
until $area \geq width * (i - k + 1)$ or $p_{r,j} <> c$

Figure 3. Computing the largest monochromatic rectangle match in (i, j)

0	0	0	0	0	0	1	step 1
0	0	0	0	1	0	0	step 2
0	0	1	1	1	0	0	step 3
0	0	1	0	1	1	0	step 4
0	0	1	0	1	0	0	step 5
0	1	0	0	1	1	0	step 6
1	0	1	0	1	1	0	

Figure 4. The largest monochromatic match in (0,0) is computed at step 5

greedy match and then replaced in the hash table by a pointer to the current position. The procedure for computing the largest square match with left upper corners in positions (i, j) and (k, h) takes O(M) time, where M is the size of the match. Obviously, this procedure can be used to compute the largest monochromatic square in a given position (i, j) as well. If the 4 x 4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic square in that position. Otherwise, we compute the largest match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then

Figure 5. Image 1

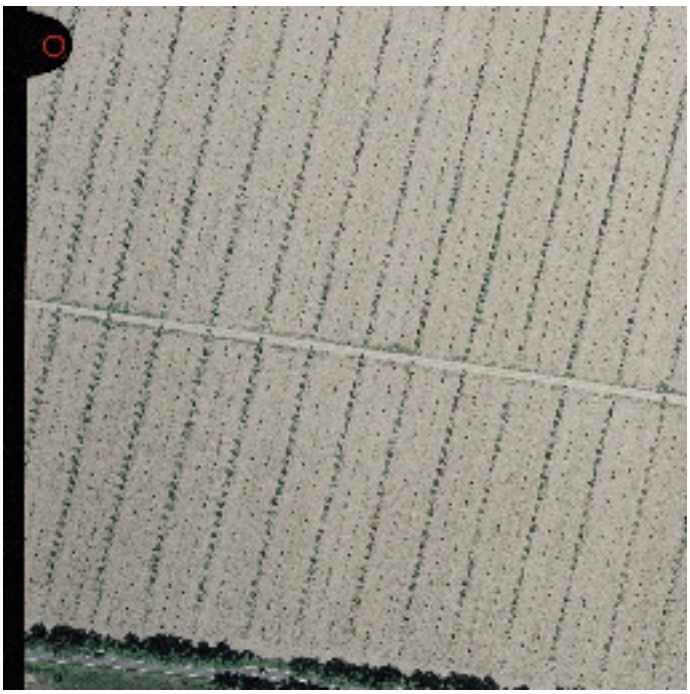


Figure 6. Image 2

it is left uncompressed and added to the hash table with its current position. The positions covered by matches are skipped in the linear scan of the image. We wish to point out that besides the proper matches we call a match every square of the parsing of the image produced by the heuristic. We also call pointer the encoding of

Figure 7. Image 3

Figure 8. Image 4

a match. Therefore, each pointer starts with a flag field indicating whether there is a monochromatic match (0 for the white ones and 10 for the black ones), a proper match (110) or a raw match (111).

Figure 9. Image 5

We were able to partition an image into up to a hundred areas and to apply the square block matching heuristic independently to each area with no relevant loss of compression effectiveness on both the CCITT bi-level image test set and to the set of five 4096 x 4096 pixels half-tone images of figures 5-9. Moreover, in order to implement decompression on an array of processors, we needed to indicate the end of the encoding of a specific area. So, we changed the encoding scheme by associating the flag field 1110 to the raw match so that 1111 could indicate the end of the sequence of pointers corresponding to a given area. Then, the flag field 1110 is followed by sixteen bits uncompressed, flag fields 0 and 10 by the size of the square side while flag field 110 is also followed by the position of the matching copy. The values following the flag fields are represented by a fixed length code.

In [10], [8], [9] we showed the compression and decompression times of the parallel procedure applied to the five images of figures 5-9, doubling up the number of processors of the avogadro.cilea.it machine from 1 to 32 (figures 1 and 2). In the next section, we present a lossless compression technique for binary images which is implementable at no communication cost on both small and large scale parallel systems, differently from the one described in this section.

3 Compression via Monochromatic Pattern Substitution

The technique scans an image row by row. If the 4 x 4 subarray in position (i, j) of the image is monochromatic, then we compute the largest monochromatic rectangle in that position. We denote with $p_{i,j}$ the pixel in position (i, j). The procedure for finding the largest rectangle with left upper corner (i, j) is described in figure 3. At the first step, the procedure computes the longest possible width for a monochromatic rectangle in (i, j) and stores the color in c. The rectangle 1 x ℓ computed at the first

step is the current detected rectangle and the sizes of its sides are stored in *side*1 and *side*2. In order to check whether there is a better match than the current one, the longest sequence of consecutive pixels with color c is computed on the next row starting from column j. Its length is stored in the temporary variable *width* and the temporary variable *length* is increased by one. If the rectangle R whose sides have size *width* and *length* is greater than the current one, the current one is replaced by R. We iterate this operation on each row until the area of the current rectangle is greater or equal to the area of the longest feasible *width*-wide rectangle, since no further improvement would be possible at that point.

For example, in figure 4 we apply the procedure to find the largest monochromatic rectangle in position $(0, 0)$. A monochromatic rectangle of width 6 is detected at step 1. Then, at step 2 a larger rectangle is obtained which is 2 x 4. At step 3 and step 4 the current rectangle is still 2 x 4 since the longest monochromatic sequence on row 3 and 4 comprises two pixels. At step 5, another sequence of two pixels provides a larger rectangle which is 5 x 2. At step 6, the procedure stops since the longest monochromatic sequence is just one pixel and the rectangle can cover at most 7 rows. It follows that the detected rectangle is 5 x 2 since a rectangle of width 1 cannot have a larger area. Such procedure for computing the largest monochromatic rectangle in position (i, j) takes $O(M \log M)$ time, where M is the rectangle size. In fact, in the worst case a rectangle of size M could be detected on row i, a rectangle of size $M/2$ on row $i + 1$, a rectangle of size $M/3$ on row $i + 2$ and so on.

If the 4 x 4 subarray in position (i, j) of the image is not monochromatic, we do not expand it. The positions covered by the detected rectangles are skipped in the linear scan of the image. The encoding scheme for such rectangles uses a flag field indicating whether there is a monochromatic match (0 for the white ones and 10 for the black ones) or not (11). If the flag field is 11, it is followed by the sixteen bits of the 4 x 4 subarray (raw data). Otherwise, we bound by twelve the number of bits to encode either the width or the length of the monochromatic rectangle. We use either four or eight or twelve bits to encode one rectangle side. Therefore, nine different kinds of rectangle are defined. A monochromatic rectangle is encoded in the following way:

- the flag field indicating the color;
- three or four bits encoding one of the nine kinds of rectangle;
- bits for the length and the width.

Four bits are used to indicate when twelve bits or eight and twelve bits are needed for the length and the width. This way of encoding rectangles plays a relevant role for the compression performance. In fact, it wastes four bits when twelve bits are required for the sides but saves four to twelve bits when four or eight bits suffice.

4 The Parallel Implementations

The variable length coding technique expained in the previous section has been applied to the CCITT test set of bi-level images and has provided a compression ratio equal to 0.13 in average. The images of the CCITT test set are 1728 x 2376 pixels. If these images are partitioned into 4^k sub-images and the compression heuristic is applied independently to each sub-image, the compression effectiveness remains about the same for $1 \leq k \leq 5$ with a 1 percent loss for $k = 5$. Raw data are associated

with the flag field 110, so that we can indicate with 111 the end of the encoding of a sub-image. For $k = 6$, the compression ratio is still just a few percentage points of the sequential one. This is because the sub-image is 27 x 37 pixels and it still captures the monochromatic rectangles which belong to the class encoded with four bits for each dimension. These rectangles are the most frequent and give the main contribution to the compression effectiveness. On the other hand, the compression ratio of the square block matching heuristic when applied to the CCITT test set is about 0.16 for $1 \leq k \leq 3$ and it deteriorates in a relevant way for $k \geq 4$.

The compression effectiveness of the variable-length coding technique depends on the sub-image size rather than on the whole image. In fact, if we apply the parallel procedure to the test set of larger binary images as the 4096 x 4096 pixels half-tone topographic images of figures 5-9 we obtain about the same compression effectiveness for $1 \leq k \leq 5$. The compression ratio is 0.28 with a 2 percent loss for $k = 6$. This means that actually the approach without interprocessor communication works in the context of unbounded parallelism as long as the elements of the image partition are large enough to capture the monochromatic rectangles encoded with four bits for each dimension. On the other hand, the compression ratio of the square block matching heuristic is 0.31 and depends on how the match size compares to the whole image. In fact, the compression effectiveness is again about the same for $1 \leq k \leq 3$ and deteriorates in a relevant way for $k \geq 4$.

We wish to remark at this point that parallel models have two types of complexity, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any parallel algorithm and has standard solutions, the communication cost of the computational phase after the distribution of the data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation to design a practical algorithm. The simplest model for this phase is, of course, a simple array of processors with no interconnections and, therefore, no communication cost. Obviously, the compression and decompression procedures described here are suitable for such a model and implementable on both small (about 100 processors) and large (about 1000 processors) scale distibuted systems.

Image	1 proc.	2. proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	41	22	12	7	4	2
2	40	23	14	7	4	2
3	42	22	15	8	4	2
4	42	25	15	7	4	2
5	41	22	13	7	4	3
Avg.	41.2	22.8	13.8	7.2	4	2.2

Figure 10. Compression times of the new procedure (cs.)

We show in figures 10 and 11 the compression and decompression times of the parallel procedure applied to the five images of figures 5-9, doubling up the number of processors of the avogadro.cilea.it machine from 1 to 32. We executed the compression and decompression on each image several times. The variances of both the compression and decompression times were small and we report the greatest run-

Image	1 proc.	2. proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	20	11	6	4	2	1
2	20	11	7	4	2	1
3	21	11	8	4	2	1
4	21	13	8	4	2	1
5	20	11	7	4	2	1
Avg.	20.4	11.4	7.2	4	2	1

Figure 11. Decompression times of the new procedure (cs.)

ning times, conservatively. As it can be seen from the values on the tables, also the variance over the test set is quite small. The decompression times are faster than the compression ones and in both cases we obtain the expected speed-up, achieving running times about twenty times faster than the sequential ones. Images 1 and 5 have the smallest compression time, while image 2 has the greatest one. Image 2 also has the greatest sequential decompression time while image 5 has the smallest one. The greatest compression time with 32 processors is given by image 5 while the decompression time with 32 processors is the same for all images.

5 A Massively Parallel Algorithm

Compression and decompression parallel implementations are presented, requiring $O(\alpha \log M)$ time with $O(n/\alpha)$ processors for any integer square value $\alpha \in \Omega(\log n)$ on a PRAM EREW. As in the previous section, we partition an m x m' image I in x x y rectangular areas, where x and y are $\Theta(\alpha^{1/2})$. In parallel for each area, one processor applies the sequential parsing algorithm so that in $O(\alpha \log M)$ time each area is parsed into rectangles, some of which are monochromatic. In the theoretical context of unbounded parallelism, α can be arbitrarely small and before encoding we wish to compute larger monochromatic rectangles. The monochromatic rectangles are computed by merging adjacent monochromatic areas without considering those monochromatic matches properly contained in some area. We denote with $A_{i,j}$ for $1 \leq i \leq \lceil m/x \rceil$ and $1 \leq j \leq \lceil m'/y \rceil$ the areas into which the image is partitioned. In parallel for $1 \leq i \leq \lceil m/x \rceil$, if i is odd, a processor merges areas $A_{2i-1,j}$ and $A_{2i,j}$ provided they are monochromatic and have the same color. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, $\cdots$ $A_{i2^{k-1},j}$, with i odd, were merged, then they will merge with areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, $\cdots$ $A_{(i+1)2^{k-1},j}$, if they are monochromatic with the same color. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, $\cdots$ $A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, $\cdots$ $A_{i,(j+1)2^{k-1}}$. After $O(\log M)$ steps, the procedure is completed and each step takes $O(\alpha)$ time and $O(n/\alpha)$ processors since there is one processor for each area. Therefore, the image parsing phase is realized in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors on an exclusive read, exclusive write shared memory machine. The interested reader can see how the coding and decoding phase are designed in [1], [10], since similar procedures are subroutines of the parallel block matching compression and decompression algorithms. As in the previous section, the flag field 110 corresponds the raw data so that we can indicate with 111 the end of the sequence of pointers corresponding to a

given area. Since some areas could be entirely covered by a monochromatic rectangle 111 is followed by the index associated with the next area by the raster scan. Under some realistic assumptions, implementations with the same parallel complexity can be realized on a full binay tree architecture [1], [10].

6 Scalability and Compression Effectiveness

Since the images of the CCITT test set are 1728 x 2376 pixels, they can be partitioned into 4^k sub-images for $1 \leq k \leq 8$. For $k = 8$, the sub-image is 7 x 9 pixels. The extremal case is when we partition the image into 4 x 4 subarrays. As we mentioned in section 4, the compression ratio is 0.13 in average for $1 \leq k \leq 4$ and 0.14 for $k = 5$. For greater values of k, the compression effectiveness deteriorates. For $k = 6$, $k = 7$ and $k = 8$ the compression ratio is 0.17, 0.23 and 0.51 respectively. For the extremal case, the compression ratio is about 0.80. For $k \geq 6$ and the extremal case, we obviously have an improvement on the compression results if we use interprocessor communication to compute larger monochromatic rectangles with the procedure explained in the previous section. As shown in figure 12, the improvements are consistent but not satisfactory but for the extremal case. This is because we compute larger monochromatic matches by merging adjacent monochromatic elements of the image partition. This process implies that such matches are likely to be sub-arrays of larger monochromatic rectangles with the margins included in non-monochromatic elements of the partition. Yet, these rectangles are likely to be computed as matches by the sequential procedure. Therefore, the smaller the elements are the better the merging procedure works. A more sophisticated procedure would be to consider the monochromatic matches properly contained in some element but such approach is not very practical and does not provide a parallel decoder [5].

N. Pr.	c. ratio	c. ratio
$\leq 4^4$	.13	.13
4^5	.14	.14
4^6	.17	.16
4^7	.24	.19
4^8	.51	.29
$\sim 4^9$	.80	.15
comm.	without	with

Figure 12. Average compression and scalability on the CCITT images

As pointed out earlier, the compression effectiveness depends on the sub-image size rather than on the whole image. In fact, if we apply the parallel procedures with and without interprocessor communication to the test set of larger binary images of figures 5-9 we obtain the average results of figure 13. As we can see, the compression effectiveness has still a bell-shaped behaviour which mantains the sequential performance to a higher degree of parallelism with respect to the CCITT test set and is again competitive with the sequential procedure when the highest degree of parallelism is reached by the extremal case. This confirms that actually the

approach without interprocessor communication works in the context of unbounded parallelism as long as the elements of the image partition are large enough to capture the monochromatic rectangles encoded with four bits for each dimension. On the other hand, interprocessor communication is effective only in the extremal case.

N. Pr.	c. ratio	c. ratio
$\leq 4^5$	.28	.28
4^6	.30	.29
4^7	.33	.33
4^8	.42	.39
4^9	.76	.60
$\sim 4^{10}$	.95	.30
comm.	without	with

Figure 13. Average compression and scalability on the 4096 x 4096 pixels images

7 Conclusions

We provided a parallel low-complexity lossless compression technique for binary images which is suitable for both small and large distributed memory systems at no communication cost. The degree of locality of the technique is high enough to guarantee no loss of compression effectiveness when it is applied independently to blocks with dimensions about 50 x 50 pixels. When the blocks are smaller, interprocessor communication is needed but it is not effective except for the extremal case of blocks with dimensions 4 x 4 pixels. We presented experimental results with at most 32 processors and obtained the expected linear speed-up. As future work, we wish to experiment with more processors by implementing the procedure on a graphical processing unit.

References

1. S. D. Agostino: *Compressing bi-level images by block matching on a tree architecture*, in Proceedings of the Prague Stringology Conference, 2008, pp. 137–145.
2. T. C. Bell, J. G. Cleary, and I. W. Witten: *Text Compression*, Prentice Hall, 1990.
3. R. P. Brent: *A linear algorithm for data compression.* Australian Computer Journal, 19 1987, pp. 64–68.
4. L. Cinque and S. D. Agostino: *Lossless image compression by block matching on practical massively parallel architectures*, in Proceedings of the Prague Stringology Conference, 2008, pp. 26–34.
5. L. Cinque, S. D. Agostino, and F. Liberati: *A work-optimal parallel implementation of lossless image compression by block matching.* Nordic Journal of Computing, 10 2003, pp. 11–20.
6. L. Cinque, S. D. Agostino, F. Liberati, and B. Westgeest: *A simple lossless compression heuristic for grey scale images*, in Proceedings of the Prague Stringology Conference, 2004, pp. 48–55.
7. L. Cinque, S. D. Agostino, F. Liberati, and B. Westgeest: *A simple lossless compression heuristic for grey scale images.* International Journal of Foundations of Computer Science, 16 2009, pp. 1111–1119.
8. L. Cinque, S. D. Agostino, and L. Lombardi: *Speeding up lossless image compression: Experimental results on a parallel machine*, in Proceedings of the Prague Stringology Conference, 2008, pp. 35–45.

9. L. Cinque, S. D. Agostino, and L. Lombardi: *Lossless image compression by block matching on practical parallel architectures.* Texts in Algorithmics, 11 2009, pp. 136–151.
10. L. Cinque, S. D. Agostino, and L. Lombardi: *Scalability and communication in parallel low-complexity lossless compression.* Mathematics in Computer Science, 3 2010, pp. 391–406.
11. J. Gailly and M. Adler: *The gzip compressor.* http://www.gzip.org, 1991.
12. S. W. Golomb: *Run length encodings.* IEEE Transactions on Information Theory, 12 1966, pp. 399–401.
13. P. G. Howard, F. Kossentini, B. Martinis, S. Forchammer, W. J. Rucklidge, and F. Ono: *The emerging JBIG2 standard.* IEEE Transactions on Circuits and Systems for Video Technology, 8 1998, pp. 838–848.
14. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
15. R. F. Rice: *Some practical universal noiseless coding technique – part I.* Technical Report JPL-79-22 Jet Propulsion Laboratory, 1979.
16. J. Rissanen: *Generalized kraft inequality and arithmetic coding.* IBM Journal on Research and Development, 20 1976, pp. 198–203.
17. J. Rissanen and G. G. Langdon: *Universal modeling and coding.* IEEE Transactions on Information Theory, 27 1981, pp. 12–23.
18. J. A. Storer: *Lossless image compression using generalized LZ1-type methods*, in Proceedings of the IEEE Data Compression Conference, 1996, pp. 290–299.
19. J. A. Storer and H. Helfgott: *Lossless image compression by block matching.* The Computer Journal, 40 1997, pp. 137–145.
20. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of ACM, 29 1982, pp. 928–951.
21. J. R. Waterworth: *Data compression system.* US Patent 4 701 745, 1987.
22. M. J. Weimberger, G. Seroussi, and G. Sapiro: *LOCO-I: A low-complexity, context based, lossless image compression algorithn*, in Proceedings of the IEEE Data Compression Conference, 1996, pp. 140–149.
23. D. A. Whiting, G. A. George, and G. E. Ivey: *Data compression apparatus and method.* US Patent 5016009, 1991.
24. X. Wu and M. D. Memon: *Context-based, adaptive, lossless image coding.* IEEE Transactions on Communications, 45 1997, pp. 437–444.

Practical Fixed Length Lempel Ziv Coding

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il
[2] Dept. of Computer Science, Ashkelon Academic College, Ashkelon 78211, Israel
shapird@ash-college.ac.il

Abstract. We explore the possibility of transforming the standard encodings of the main LZ methods to be of fixed length, without reverting to the originally suggested encodings. This has the advantage of allowing easier processing and giving more robustness against errors, while only marginally reducing the compression efficiency in certain cases.

1 Introduction and Motivation

Some of the most popular compression algorithms are based on the works of J. Ziv and A. Lempel. One of the main features of these methods is their adaptivity: they are dictionary based techniques, in which compression is obtained by replacing parts of the text to be compressed by (shorter) pointers to elements in some dictionary. The ingenious innovation of the LZ methods was to define the dictionary as the text itself, using pointers of the form (offset, length) in the variant known as LZ77 [17], or using a dynamically built dictionary consisting of the strings not encountered so far in the text for the method known as LZ78 [18].

The original Lempel-Ziv algorithms suggested to produce pointers and single characters in strict alternation and thereby enabled the use of a fixed length encoding for the (offset, length, character) items of LZ77 or the (pointer, character) items of LZ78. Later and more efficient implementations then removed the requirement for strict alternation by adding flag-bits, as in [12] for LZ77, or by including the single characters within the dictionary elements, as in [15] for LZ78. This paved the way to using variable length encodings. Indeed, the idea of assigning shorter codewords to items which appear with higher probability has been a main ingredient of some basic compression approaches, such as Huffman coding, which is optimal once the probabilities of the items to be encoded are given, and under the constraint that an integral number of bits should be used for the encoding of each element. But the use of variable length coding has its price:

- The manipulation of variable length codewords, which are thus not always byte aligned, is much more involved, requiring algorithms that are more complicated and usually also more time consuming. While increased time might be acceptable in certain applications for the encoding, the *decoding* is often required to be very fast, and fixed length codes can contribute to this goal.
- The compressed text is more vulnerable to transmission errors.

Another disadvantage of variable length codes is that direct search in the compressed file is not always possible, and even if it is, it might be slower than in the

Shmuel T. Klein, Dana Shapira: Practical Fixed Length Lempel Ziv Coding, pp. 116–126 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

case of fixed length encodings. In the case of LZ schemes, even the processing of fixed length encodings might be problematic, as will be explained below.

These deficiencies of variable length codes have led recently to the investigation of several alternatives. It was first suggested to trade the optimal compression of binary Huffman codes with a faster to process 256-ary variant. The resulting codewords have lengths which are multiples of 8, so consist of 1, 2 or more bytes, and for large enough alphabets, the loss in compression efficiency is of the order of only a few percent [4], and may often be tolerated. The problem of getting large enough alphabets for the method to be worthwhile was overcome, on natural text files, by defining the elements of the "alphabet" to be encoded as the different *words* of the database, instead of just the different characters [11].

A further step was then to use a *fixed* set of codewords, rather than one which has to be derived according to the given probability distribution as for Huffman codes, still adhering to the main idea of variable length codes with codeword lengths that are multiples of bytes [3,2]. Another tradeoff with better compression but slower processing can be obtained by the use of Fibonacci codes [6]. Both (s, c) codes and Fibonacci codes have the additional advantage of lending themselves to searches directly in the compressed file. Finally, at the other end of the spectrum, one may advocate again the use of fixed length codes, achieving compression by using variable length for the strings to be encoded instead for the codewords themselves [13,9].

We now turn to the investigation of how to adapt Lempel and Ziv codes to fit into a framework of fixed length codes without reverting to the original encodings based on alternating pointers and characters, yielding some of the above mentioned advantages. The new variants are presented in the next section, and some experimental results are given in Section 3.

2 Fixed length LZ codes

The two main approaches suggested by Lempel and Ziv have generated a myriad of variants. We shall more specifically refer to LZSS [12] as representative of the LZ77 family, and to LZW [15] for LZ78.

2.1 Fixed length LZSS

LZ77 produces an output consisting of a strictly alternating sequence of single characters and pointers, but no external dictionary is involved and the pointers, coded as (offset, length) pairs, refer to the previously scanned text itself. LZSS suggests to replace strict alternation by a set of flag-bits indicating whether the next element is a single character or an (offset, length) pair. A simple implementation, like in LZRW1 [16], uses 8 bits to encode a character, 12 bits for the offset and 4 bits for the length. Excluding the flag-bits, we thus get codewords of lengths 8 or 16. Though variable length, it still keeps byte alignment, by processing the flag bits by blocks of 16.

One of the challenges of LZ77 is the way to locate the longest matching previously occurring substring. Various techniques have been suggested, approximating the requested longest match by means of trees or hashing as in [16]. In many cases, fixed length codes are preferred in a setting in which encoding and decoding are not

considered to be symmetric tasks: encoding might be done only once, e.g., during the construction of a large textual database, so the time spent on compression is not really relevant. On the other hand, decompression is requested every time the database is being accessed and ought therefore to be extremely fast. For such cases, there is no need to use hashing or other fast approximations for finding the longest possible match, and one might find the true optimum by exhaustive search. Given the limit on the encoding of the offset (12 bits), the size of the window to be scanned is just 4K, but the scan has to be done for each position in the text.

Encoding and decoding A fast way to get a fixed length encoding is to set the length of the shortest character sequence to be copied to 3 (rather than 2 in the original algorithm); thus when looking for the longest previously occurring sequence P which matches the sequence of characters C starting at the current point in the text, if the length of P is less than 3, we encode the first 2 characters of C. In the original algorithm, if the length of P was less than 2, we encoded only the first character of C. The resulting encoding therefore uses only 16 bits elements, regardless of if they represent character pairs or (offset, length) pairs. To efficiently deal also with the flag bits, one can use the same technique as in [16], aggregating them into blocks of 16 elements each.

Decoding could then be done by the following simple procedure:

```
j ⟵ 0
while not EOF
    F[0 · ·15] ⟵ next 2 bytes
    for i ⟵ 0 to 15
        if F[i] = 0 then
            (T[j], T[j + 1]) ⟵ next 2 bytes
            j ⟵ j + 2
        else
            z ⟵ next 2 bytes
            off ⟵ 1 + z mod 2^12
            len ⟵ 3 + ⌊z/2^12⌋
            T[j · · j + len − 1] ⟵ T[j − off · · j − off + len − 1]
            j ⟵ j + len
    end for
end while
```

Compression efficiency At first sight, enforcing fixed length codewords seems quite wasteful with the LZRW1 encoding scheme:

1. The number of characters that are not encoded as part of a pointer will increase, so the average length of a substring represented by a codeword, and thus the compression, will decrease. Moreover, each such character actually adds a negative

contribution to the compression, because the need of the flag bit: a single character is encoded by 9 bits in the original LZSS and by 8.5 bits in the fixed length variant, whereas only 8 bits are needed in the uncompressed file.

2. Elements of the form (off,2) are replaced by a pair of characters, and both are encoded by the same number of bits.
3. If there is a sequence of odd length ℓ of characters encoded on their own in LZSS, followed by a pointer (off, len), with len > 2, one would need $9\ell + 17$ bits to encode them. This can be replaced in the fixed length variant by $(\ell+1)/2$ pairs of characters, followed by the pointer (off, len-1), requiring $17\left(\frac{\ell+1}{2}+1\right)$ bits. There is thus a loss whenever $\ell < 17$, and this will mostly be the case, as the probability of having sequences of single characters of length longer than 17 in the compressed text is extremely low: it would mean that the sequence contains no substring of more than two characters which occurred earlier. In all our experiments, the longest sequence of single characters was of length $\ell = 14$.

On the other hand, the passage to fixed length encoding can even lead to additional savings in certain cases. An extreme example would be an LZSS encoded file in which all the sequences of single characters are of even length. In the corresponding fixed length encoding, all the (off, len) pairs would be kept, and the single characters could be partitioned into pairs, each requiring only 17 bits for its encoding instead of 18 bits if encoded as two 9-bit characters. But the improvement can also originate from a different parsing, as can be seen in the example in Figure 1.

File	Text	Size
Original	b c d e f b b c d a b b c d e f b b	18 bytes
variable LZSS	b c d e f b (6,3) a (5,4) (11,4)	7 chars + 3 pointers + 10 bits → 15 bytes
fixed length	b c d e f b (6,3) a b (5,3) (11,4)	4 pairs + 3 pointers + 7 bits ⟶ 15 bytes
better fixed	b c d e f b (6,3) a b (11,7)	4 pairs + 2 pointers + 6 bits ⟶ 13 bytes

Figure 1. Comparison of LZSS encodings

The first line brings a small example of a text of 18 characters and the next line is its LZSS encoding using the LZRW1 scheme: it parses the text into 10 elements, 7 single characters and 3 (off, len) pointers, for a total of $7 \times 1 + 3 \times 2 + 10$ flag bits (rounded up to 2 bytes) = 15 bytes. The third line shows a parsing which could be derived from the one above it, but does not encode single characters, so that a (5,4) has to be replaced by a b (5,3). There are now 11 elements in the parsing, 4 character pairs and 3 pointers, so they require already 14 bytes, to which one has to add the flags. The last line shows that in this case, another parsing is possible, using only 6 codewords, which yields 12 bytes plus the flag bits. This example shows that one can get a strict improvement with fixed length encoding.

An encoding as in LZRW1 is of course not the only possibility for fixed lengths, but the 16 bit units have been chosen because it is most convenient to process the input by multiples of bytes. If one is willing to abandon byte alignment, but still wants to keep fixed length, one could for example define codewords of 18 bits and incorporate the flag bits into the codewords themselves. The processing could be done by blocks of 18 bytes = 144 bits, each block representing 8 consecutive 18 bit codewords. One could then use 13 bits for offsets of size 1 to 8192, and stay with

4 bits for copy lengths between 3 and 18. Alternatively, the 18 bit codewords could be stored as before in units of 2 bytes, considering the exceeding 2 bits as flag-bits, which could be blocked for each set of 8 consecutive codewords.

For a further refinement, note that only 17 of the 18 bits are used in case of a character pair. For example, the second bit from the left could be unused. To fully exploit the encoding possibilities, decide that this second bit will be 0 when the following 16 bits represent a character pair; this frees 16 bits in the case the second bit is set to 1. One can then use these 16 bits for (off, len) pairs as before in the 16-bit encoding for offsets from 1 to 4096, and shift the range of the offsets encoded by the 18-bit codewords accordingly to represent numbers between 4097 and 12288. The increased size of the search window will lead to improved compression.

The above methods are suitable for alphabets with up to 256 symbols. For the general case of a byte aligned fixed length encoding of LZSS, consider an alphabet Σ, of size $|\Sigma|$, and denote the size of a byte by B. We shall adapt the encoding algorithm, the size W of the sliding window and the maximum copy length L as follows. If $\log(|\Sigma|) = kB$ for some integer k, choose W and L such that

$$\log W + \log L = kB, \tag{1}$$

and use throughout codewords of fixed length k bytes. If $\log(|\Sigma|)$ is not a multiple of B, let $k = \lceil \log(|\Sigma|)/B \rceil$ and again use codewords of k bytes, only that in this case, a codeword for a single character has r spare bits, $1 \leq r < B$, which we shall use to accommodate the required flagbits. The codewords for copy elements are still defined by equation (1).

The first element does not need a flag, as it must be a single character. The r flags in codewords representing single characters refer to the *following* r items that have not yet been assigned a flag. For example, if the second and third elements in the compressed file are single characters, we have already collected $2r$ flag bits, referring to the elements indexed $2, 3, \ldots, 2r + 1$. If at some stage we run out of flag bits, a codeword consisting only of kB flags is inserted. This may happen in the above example if the elements indexed $3, \ldots, 2r + 1$ are all of type (`offset, length`). If there are many items of type single character, this scheme might be wasteful, as a part of the available flag bits will be superfluous, but in a typical scenario, after some initialization phase, an LZSS encoded file consists almost exclusively of a sequence of copy items, with only occasionally interspersed single characters.

Robustness A major reason for the vulnerability of variable length codes to transmission errors — a bit flip from 0 to 1 or 1 to 0, a bit getting lost or a spurious bit being picked up — is that the error might not be locally restricted. If one considers only changes in bit values, but assumes that there are no bit insertions or deletions, then in the case of fixed length codes, only a single codeword will be affected. But for variable length codes, the bit flip might imply that the encoded text will now be parsed into codewords of different lengths, and the error can then propagate indefinitely. Consider, for example, the code $\{01, 10, 000, 001, 110, 111\}$ for the alphabet {A, B, C, D, E, F}, respectively, and suppose the encoded text is EBBB$\cdots$BA = 110101010$\cdots$1001. If the leftmost bit is flipped, the text will erroneously be decoded as AAAA$\cdots$AD, which shows that a single bit error can affect an unlimited number of consecutive codewords.

For the case in which the number of bits can also change, fixed length codes might be the most vulnerable of all. An inserted or deleted bit will cause a shift in the decoding, and all the codeword boundaries after the error could be missed. In variable length encoded files, on the other hand, the error might also spill over to a few consecutive other codewords, but there is always also a chance that synchronization will be regained. This actually happens often after a quite low number of codewords for Huffman codes [8], a property which has several applications, see [10].

As to LZSS encoded texts, bit insertions or deletions are hazardous for both the fixed and the original variable length variants. For bit flips, if they occur in the encoding of a character, it will be changed into another one, so the error will be local; if the bit flip occurs in an offset item, a wrong sequence will be copied, but again, only a restricted range will (generally) be affected; if the error is in a length item, a wrong number of characters will be copied (yet starting from the correct location), which is equivalent to inserting or deleting a few characters; if one of the flag bits is flipped, then the text encoded by the original LZSS could be completely garbelled, while for the fixed length variant, no harm is done other than wrongly decoding a codeword of one type as if it were of the other, since both types are encoded by 16 bits.

There might be the danger of an unbounded propagation of the error even in the case of a bit flip in an offset or length item: the wrong characters could be referenced later by subsequent (off, len) pairs, which would again cause the insertion of wrong characters, etc. Such a chain of dependent errors is not unlikely in natural language texts: a rare word might be locally very frequent, e.g., some proper name, and each occurrence could be encoded by a reference to the preceding one. If there is an error in the first occurrence, all the subsequent ones might be affected.

In fact, LZSS encodings are error prone on two accounts: because of the variable lengths, (1) the parsing into codewords could be wrong, but in addition, even if the parsing and thus the codewords are correct, (2) their interpretation might suffer when previous errors caused an erroneous reconstitution of the referenced text. Fixed length LZSS is robust against the first type of error, but not against the second.

Compressed Pattern Matching Given a pattern of size m and a compressed file of size n, *Compressed Pattern Matching* is the problem of locating a pattern directly in the compressed file without any decompression, in time proportional to the size of the input. Performing pattern matching in LZSS compressed files is not trivial, as the encoding of the same subpattern is not necessarily the same throughout the file and depends also on its location. A variant of LZSS that is suitable for compressed matching, replacing the backward by forward pointing copy elements, has been suggested in [7], but the problems remain essentially the same for fixed length as for variable length encodings.

2.2 Fixed length LZW

LZW is based on parsing the text into phrases belonging to a dictionary, which is dynamically built during the parsing process itself. The output of LZW consists of a sequence of pointers, and the size of the encoding of the pointers is usually adapted to the growing dictionary: starting with a dictionary of 512 entries (256 for a basic

ASCII alphabet as the single characters have to be included, and room for 256 additional items), one uses 9-bit pointers to refer to its elements, until the dictionary fills up. At that stage, the number of potential entries is doubled and the length of the pointers is increased by 1. This procedure is repeated until the size of the dictionary reaches some predetermined limit, say 64K with 16-bit pointers. In principle, the dictionary could be extended even further, but erasing it and starting over from scratch has the advantage of allowing improved adaptation to dynamically changing texts, while only marginally hurting the compression efficiency on many typical texts.

Encoding and decoding A fixed length encoding variant of LZW could thus fix the size S of the dictionary in advance and use throughout $\log S$ bits for each of the pointers. This would have almost no effect on the encoding and decoding procedures, besides that there is no need to keep track of the number of encoded elements, since their size does not change.

Compression efficiency The exact loss incurred by passing from the variable to fixed length LZW can be evaluated based on the fact that after each substring parsed from the text, a new element is adjoined to the dictionary. Suppose we let the dictionary grow up to a size of 2^k entries. For the first 256 elements of the encoded text, the loss of using fixed instead of variable length is $256(k-9)$ bits; for the next 512 elements, the loss is $512(k-10)$, etc. The penultimate block has 2^{k-2} elements, for each of which only 1 bit is lost, and in the last block, of 2^{k-1} elements, there is no loss as the maximum of k bits are needed anyway. The total loss in bits is thus

$$\sum_{i=1}^{k-9} 2^{k-i-1} i = 2^k - (k-7)2^8. \tag{2}$$

The size of the fixed length compressed file at this stage is k times the total number of encoded elements,

$$k \sum_{i=8}^{k-1} 2^i = k(2^k - 2^8). \tag{3}$$

The relative loss, expressed as a fraction of the file size, is then the ratio of (2) to (3), which is plotted in Figure 2. Typical values are about 6.05% for $k = 16$ and 4.17% for $k = 24$, and in any case the loss does not exceed 6.463% (for $k = 14$).

For larger files, if the dictionary is rebuilt each time it reaches a size of 2^k elements, one may consider it as a sequence of blocks, each of which, except perhaps the last, representing the encoding of 2^k elements. The overall relative loss would thus approach the values above for large enough files. Another option, which can also be useful for the regular LZW, is to consider the dictionary as constant once it fills up. This approach is actually a good one when the file is homogeneous, like, e.g., some natural language text. In that case, the relative loss of using fixed length codewords already from the beginning, and not only after 2^{k-1} elements have been processed, will decrease to zero with increasing file size.

Robustness From the point of view of robustness to errors, fixed and variable length LZW are identical. Note that a bit flip cannot change the length of one of the codewords, even if those are of variable length, because the length is determined by an

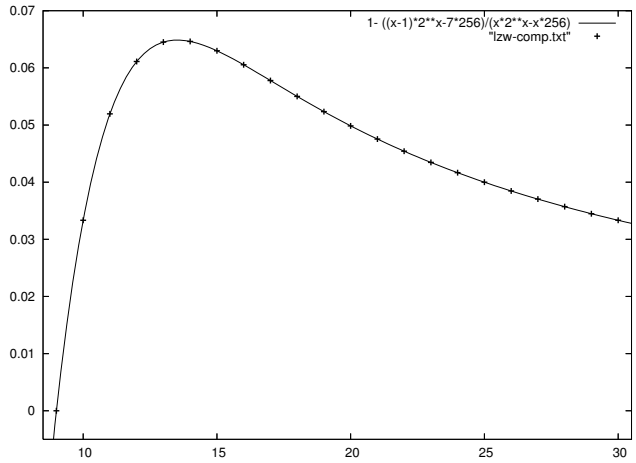


Figure 2. Fraction of loss by using fixed length LZW as function of codeword length

external mechanism (the number of elements processed so far) and does not depend on the value of the codeword itself. Therefore a bit flip will not affect subsequent codewords, and the error seems at first sight to be locally restricted. There might, however, be a snowball effect in case the error occurs during the construction of the dictionary: a wrong bit implies an erroneously decoded codeword, which may cause one or more wrong codewords to be inserted into the dictionary; if these are later referenced, they will cause even more such wrong insertions, and in the long run, the text may become completely garbelled.

If the dictionary is erased when it reaches 2^k entries, such errors cannot propagate beyond these new initialization points. On the other hand, if the dictionary is considered as fixed once it reaches it limiting size, a bit flip in the construction phase can cause considerable damage, but a bit error occurring later will only destroy a single codeword.

Compressed Pattern Matching Amir, Benson and Farach [1], were the first to perform Compressed Pattern Matching in LZW. After preprocessing the pattern, a so-called LZW trie is built, and used to check at each stage whether the pattern has been found. Since the algorithm requires the extraction of all the codewords for building the trie, the difference between fixed and variable length encodings depends on the number of accesses to the encoded file. One should distinguish between the efficient byte aligned operations and the more expensive operations requiring bit manipulations. If the size of the dictionary is 2^{8k} for some integer k, each codeword of the fixed length LZW encoding requires a single byte oriented operation (by fetching k bytes at a time). However, the codewords of the variable length LZW encoding require bit manipulations in addition to byte extractions. Although the size of the compressed file in fixed length LZW is larger than for variable length LZW, the compressed matching algorithm depends on the number of codewords, which is identical in the two schemes. Therefore, compressed pattern matching in fixed length encoding is less time consuming than pattern matching in variable length LZW.

Figure 3 gives a schematic view of the layout of LZW codewords in the case of variable length. The lower part (b) shows the increase of the codeword sizes from left to right, and the upper part (a) zooms in on the subfile in which only 10-bit

codewords are used. Solid lines refer to byte boundaries and the broken lines to codeword boundaries. The appearance of a broken line which does not overlap with a solid line indicates that bit manipulations are required. In the case of fixed length encodings, the codeword length can be chosen so as to minimize the bit specific operations, whereas for the variable length encodings, the non-synchronization of the byte and codeword boundaries can not be avoided.

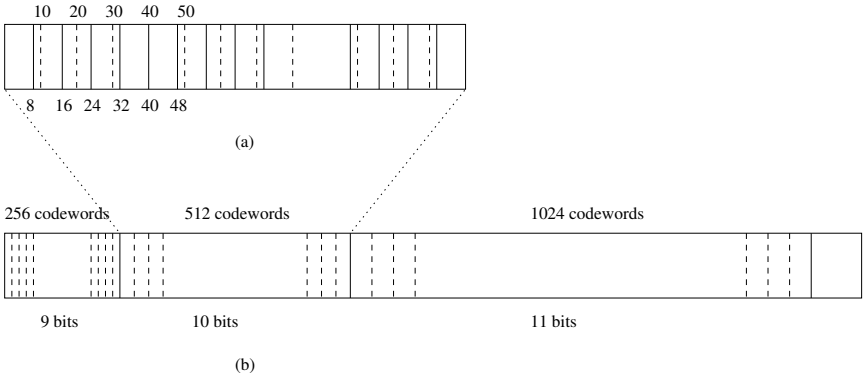


Figure 3. Variable length LZW encoding

3 Experimental results

To empirically compare the compression efficiency, we chose the following input files of different sizes and languages: the Bible (King James version) in English, 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 [14], and the concatenated text of all the XML files of the INEX database [5]. To get also alphabets of different sizes, the Bible text was stripped of all punctuation signs, whereas the French text and the XML file have not been altered.

File	Size (MB)	gzip bzip
Bible	2.96	0.279 0.205
JOC corpus	7.26	0.306 0.212
XML	494.7	0.278 0.202

Table 1. Test file statistics

Table 1 brings basic statistics on the files, their sizes in MB and some measure of compressibility, in terms of bzip2 and gzip compression. All compression figures are given as the ratio between the size of the compressed file to that of the uncompressed one. Table 2 deals with LZSS and brings results for 16 and 18 bit codewords. The columns headed hash refer to the approximate method of [16], in which the matching substring is located by hashing character pairs. In our case, we used a full table of 2^{16} entries for each possible character pair; when a previous occurrence was found, it was extended as much as possible. Much better compression could be obtained by using an optimal variant, searching the full addressable window for the longest match, the column headed fix referring to the fixed length variant, and the column headed var

to the original one using variable length. The column entitled loss shows the relative loss in percent when using fixed length LZSS, which can be seen to be low.

LZSS	16 bit				18 bit			
	hash	fix	var	loss	hash	fix	var	loss
Bible	0.664	0.398	0.398	0.05%	0.694	0.345	0.331	4.0%
JOC corpus	0.732	0.452	0.451	0.42%	0.760	0.388	0.372	4.1%
XML	0.637	0.412	0.409	0.65%	0.655	0.357	0.340	4.8%

Table 2. Comparing fixed with variable length compression for LZSS

Table 3 is then the corresponding table for LZW, with dictionaries addressed by pointers of 12, 16 and 18 bits. The test files being homogeneous, we used the variant building the dictionary until it fills up, and keeping it constant thereafter. This explains why the loss incurred by using fixed length is decreasing with the size of the input file.

LZW	12 bit			16 bit			18 bit		
	fix	var	loss	fix	var	loss	fix	var	loss
Bible	0.444	0.444	0.02%	0.341	0.339	0.75%	0.313	0.303	3.35%
JOC corpus	0.482	0.482	0.009%	0.346	0.345	0.30%	0.306	0.302	1.38%
XML	0.605	0.605	0.001%	0.484	0.484	0.004%	0.436	0.436	0.01%

Table 3. Comparing fixed with variable length compression for LZW

4 Conclusion

We saw that there is only a small increase in the size of the compressed file when passing from the standard variable length LZ encodings to fixed length variants. In many applications, it may thus be worthwhile to consider this option, which gives some known advantages, like more robustness and easier processing.

References

1. A. Amir, G. Benson, and M. Farach: *Let sleeping files lie: Pattern matching in Z-compressed files.* Journal of Computer and System Sciences, 52 1996, pp. 299–307.
2. N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller: *(S,C)-dense coding: an optimized compression code for natural language text databases*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'03, vol. 2857 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 122–136.
3. N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá: *An efficient compression code for text databases.* Proc. European Conference on Information Retrieval ECIR'03, 2633 2003, pp. 468–481.
4. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Trans. on Information Systems, 18 2000, pp. 113–139.
5. G. Kazai, N. Gövert, M. Lalmas, and N. Fuhr: *The INEX evaluation initiative*, in Intelligent search on XML data, LNCS 2818, Springer-Verlag, 2003, pp. 279–293.
6. S. T. Klein and M. Kopel Ben-Nissan: *Using Fibonacci compression codes as alternatives to dense codes.* Proc. Data Compression Conference DCC–2008, 2008, pp. 472–481.

7. S. T. Klein and D. Shapira: *A new compression method for compressed matching.* Proc. Data Compression Conference DCC–2000, 2000, pp. 400–409.
8. S. T. Klein and D. Shapira: *Pattern matching in Huffman encoded texts.* Information Processing & Management, 41(4) 2005, pp. 829–841.
9. S. T. Klein and D. Shapira: *Improved variable-to-fixed length codes*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'08, Lecture Notes in Computer Science, Melbourne, 2008, Springer-Verlag, pp. 39–50.
10. S. T. Klein and Y. Wiseman: *Parallel Huffman decoding with applications to JPEG files.* The Computer Journal, 46(5) 2003, pp. 487–497.
11. A. Moffat: *Word-based text compression.* Software — Practice & Experience, 19 1989, pp. 185–198.
12. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of the ACM, 29(4) 1982, pp. 928–951.
13. B. P. Tunstall: *Synthesis of noiseless compression codes*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1967.
14. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The arcade project.* Parallel Text Processing, J. Véronis, ed., 2000, pp. 369–388.
15. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
16. R. N. Williams: *An extremely fast Ziv-Lempel data compression algorithm.* Proc. Data Compression Conference DCC–91, 1991, pp. 362–371.
17. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. on Information Theory, 23 1977, pp. 337–343.
18. J. Ziv and A. Lempel: *Compression of individual sequence via variable rate coding.* IEEE Trans. on Information Theory, 24 1978, pp. 530–536.

Tight and Simple Web Graph Compression

Szymon Grabowski and Wojciech Bieniecki

Computer Engineering Department, Technical University of Łódź,
Al. Politechniki 11, 90–924 Łódź, Poland
{sgrabow,wbieniec}@kis.p.lodz.pl

Abstract. Analysing Web graphs has applications in determining page ranks, fighting Web spam, detecting communities and mirror sites, and more. This study is however hampered by the necessity of storing a major part of huge graphs in the external memory, which prevents efficient random access to edge (hyperlink) lists. A number of algorithms involving compression techniques have thus been presented, to represent Web graphs succinctly but also providing random access. Those techniques are usually based on differential encodings of the adjacency lists, finding repeating nodes or node regions in the successive lists, more general grammar-based transformations or 2-dimensional representations of the binary matrix of the graph. In this paper we present a Web graph compression algorithm which can be seen as engineering of the Boldi and Vigna (2004) method. We extend the notion of similarity between link lists, and use a more compact encoding of residuals. The algorithm works on blocks of varying size (in the number of input lines) and sacrifices access time for better compression ratio, achieving more succinct graph representation than other algorithms reported in the literature. Additionally, we show a simple idea for 2-dimensional graph representation which also achieves state-of-the-art compression ratio.

Keywords: graph compression, random access

1 Introduction

Development of succinct data structures is one of the most active research areas in algorithmics in the last years. A succinct data structure shares the interface with its classic (non-succinct) counterpart, but is represented in much smaller space, via data compression. Successful examples along these lines include text indexes [17], dictionaries, trees [12,16] and graphs [16]. Queries to succinct data structures are usually slower (in practice, although not always in complexity terms) than using non-compressed structures, hence the main motivation in using them is to allow to deal with huge datasets in the main memory. For example, indexed exact pattern matching in DNA would be limited to sequences shorter than 1 billion nucleotides on a commodity PC with 4 GB of main memory, if the indexing structure were the classic suffix array (SA), and even less than half of it, if SA were replaced with a suffix tree. On the other hand, switching to some compressed full-text index (see [17] for a survey) shifts the limit to over 10 billion nucleotides, which is more than enough to handle the whole human genome.

Another huge object of significant interest seems to be the Web graph. This is a directed unlabeled graph of connections between Web pages (i.e., documents), where the nodes are individual HTML documents and the edges from a given node are the outgoing links to other nodes. We assume that the order of hyperlinks in a document is irrelevant. Web graph analyses can be used to rank pages, fight Web spam, detect communities and mirror sites, etc. [11,20].

It was estimated that the graph of the Web index by *Yahoo!*, *Google*, *Bing* and *Ask* has between 21 and 59 billion nodes (`http://www.worldwidewebsize.com/`, May

Szymon Grabowski, Wojciech Bieniecki: Tight and Simple Web Graph Compression, pp. 127–137 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

2010), but the top figure is more likely. Therefore assuming 50 billion nodes and 20 outgoing links per node, we have about 1 trillion links. Using plain adjacency lists, representation of this graph would require about 8 TB, if the edges are represented with 64-bit pointers (note that 32-bit pointers may simply be too small). In a slightly less naïve variant, with 5-byte pointers (note that 40 bits are just enough to represent 1 trillion values, but cannot scale any longer), the space occupancy drops to 5 TB, i.e., is still ways beyond the capacities of the current RAM memories. We believe that, confronted with the given figures, the reader is now convinced about the necessity of compression techniques for Web graph representation.

2 Related work

We assume that a directed graph $G = (V, E)$ is a set of $n = |V|$ vertices and $m = |E|$ edges. The earliest works on graph compression were theoretical, and they usually dealt with specific graph classes. For example, it is known that planar graphs can be compressed into $O(n)$ bits [13,21]. For dense enough graphs, it is impossible to reach $o(m \log n)$ bits of space, i.e., go below the space complexity of the trivial adjacency list representation. Since the seminal Jacobson's thesis [14] on succinct data structures, there appear papers taking into account not only the space occupied by a graph, but also access times.

There are several works dedicated to Web graph compression. Bharat et al. [3] suggested to order documents according to their URL's, to exploit the simple observation that most outgoing links actually point to another document within the same Web site. Their Connectivity Server provided linkage information for all pages indexed by the AltaVista search engine at that time. The links are merely represented by the node numbers (integers) using the URL lexicographical order. We noted that we assume the order of hyperlinks in a document irrelevant (like most works on Web graph compression do), hence the link lists can be sorted, in ascending order. As the successive numbers tend to be close, differential encoding may be applied efficiently.

Randall et al. [19] also use this technique (stating that for their data 80% of all links are local), but they also note that commonly many pages within the same site share large parts of their adjacency lists. To exploit this phenomenon, a given list may be encoded with a reference to another list from its neighborhood (located earlier), plus a set of additions and deletions to/from the referenced list. Their encoding, in the most compact variant, encodes an outgoing link in 5.55 bits on average, a result reported over a Web crawl consisting of 61 million URL's and 1 billion links.

One of the most efficient compression schemes for Web graph was presented by Boldi and Vigna [4] in 2003. Their method is likely to achieve around 3 bits per edge, or less, at link access time below 1 ms at their 2.4 GHz Pentium4 machine. Of course, the compression ratios vary from dataset to dataset. We are going to describe the Boldi and Vigna algorithm in detail in the next section as this is the main inspiration for our solution.

Claude and Navarro [7,9] took a totally different approach of grammar-based compression. In particular, they focus on Re-Pair [15] and LZ78 compression schemes, getting close, and sometimes even below, the compression ratios of Boldi and Vigna, while achieving much faster access times. To mitigate one of the main disadvantages of Re-Pair, high memory requirements, they develop an approximate variant of this algorithm.

When compression is at a premium, one may acknowledge the work of Asano et al. [2] in which they present a scheme creating a compressed graph structure smaller by about 20–35% than the BV scheme with extreme parameters (best compression but also impractically slow). The Asano et al. scheme perceives the Web graph as a binary matrix (1s stand for edges) and detects 2-dimensional redundancies in it, via finding six types of blocks in the matrix: horizontal, vertical, diagonal, L-shaped, rectangular and singleton blocks. The algorithm compresses the data of intra-hosts separately for each host, and the boundaries between hosts must be taken from a separate source (usually, the list of all URL's in the graph), hence it cannot be justly compared to other algorithms mentioned here. Worse, retrieval times per adjacency list are much longer than for other schemes: on a order of a few milliseconds (and even over 28 ms for one of three tested datasets) on their Core2 Duo E6600 (2.40 GHz) machine running Java code. We note that 28 ms is at least twice more than the access time of modern hard disks, hence working with a naïve (uncompressed) external representation would be faster for that dataset (on the other hand, excessive disk use from very frequent random accesses to the graph can result in a premature disk failure). It seems that the retrieval times can be reduced (and made more stable across datasets) if the boundaries between hosts in the graph are set artificially, in more or less regular distances, but then also the compression ratio is likely to drop.

Also excellent compression results were achieved by Buehrer and Chellapilla [6], who used grammar-based compression. Namely, they replace groups of nodes appearing in several adjacency lists with a single "virtual node" and iterate this procedure; no access times were reported in that work, but according to findings in [8] they should be rather competitive and at least much shorter than of the algorithm from [2], with compression ratio worse only by a few percent.

Anh and Moffat [1] devised a scheme which seems to use grammar-based compression in a local manner. They work in groups of h consecutive lists and perform some operations to reduce their size (e.g., a sort of 2-dimensional RLE if a run of successive integers appears on all the h lists). What remains in the group is then encoded statistically. Their results are very promising: graph representations by about 15–30% (or even more in some variant) smaller than the BV algorithm with practical parameter choice (in particular, Anh and Moffat achieve 3.81 bpe and 3.55 bpe for the graph EU) and reported comparable decoding speed. Details of the algorithm cannot however be deduced from their 1-page conference poster.

Recent works focus on graph compression with support for bidirectional navigation. To this end, Brisaboa et al. [5] proposed the *k^2-tree*, a spatial data structure, related to the well-known quadtree, which performs a binary partition of the graph matrix and labels empty areas with 0s and non-empty areas with 1s. The non-empty areas are recursively split and labeled, until reaching the leaves (single nodes). An important component in their scheme is an auxiliary structure to compute *rank* queries [14] efficiently, to navigate between tree levels. It is easy to notice that this elegant data structure supports handling both forward and reverse neighbors, which implies from its symmetry. Experiments show that this approach uses significantly less space (3.3–5.3 bits per edge) than the Boldi and Vigna scheme applied for both direct and transposed graph, at the average neighbor retrieval times of 2–15 microseconds (Pentium4 3.0 GHz).

Even more recently, Claude and Navarro [8] showed how Re-Pair can be used to compress the graph binary relation efficiently, enabling also to extract the reverse

neighbors of any node. These ideas let them achieve a number of Pareto-optimal space-time tradeoffs, usually competitive to those from the k^2-tree.

3 The Boldi and Vigna scheme

Based on WebGraph datasets (`http://webgraph.dsi.unimi.it/`), Boldi and Vigna noticed that similarity is strongly concentrated; typically, either two adjacency (edge) lists have nothing or little in common, or they share large subsequences of edges. To exploit this redudancy, one bit per entry on the referenced list could be used, to denote which of its integers are copied to the current list, and which are not. Those bit-vectors are dubbed *copy lists*. Still, Boldi and Vigna go further, noticing that copy lists tend to contain runs of 0s and 1s, thus they compress them using a sort of run-length encoding. They assume the first run consists of 1s (if the copy list actually starts with 0s, the length of the first run is simply zero), and then it allows to represent a copy list as only a sequence of run lengths, encoded e.g. with Elias coding.

The integers on the current list which didn't occur on the referenced list must be stored too, and how to encode them is another novelty of the described algorithm. They detect intervals of consecutive (i.e., differing by 1) integers and encode them as pairs of the left boundary and the interval length; the left boundary of the next interval on a given list will be encoded as the difference to the right boundary of the previous interval minus two (this is because between the end of one interval and the beginning of another there must be at least one integer). The numbers which do not fall into any interval are called *residuals* and are also stored, encoded in a differential manner.

Finally, the algorithm allows to select as the reference list one of several previous lines; the size of the *window* is one of the parameters of the algorithm posing a tradeoff between compression ratio and compression/decompression time and space. Another parameter affecting the results is the maximum reference count, which is the maximum allowed length of a chain of lists such that one cannot be decoded without extracting its predecessor in the chain.

4 Our algorithm

Our algorithm (Alg. 1) works in blocks consisting of multiple adjacency lists. The blocks in their compact form are approximately equal, which means that the number of adjacency lists per block varies; for example, in graph areas with dominating short lists the number of lists per block is greater than elsewhere.

We work in two phases: preprocessing and final compression, using a general-purpose compression algorithm. The algorithm processes the adjacency lines one-by-one and splits their data into two streams.

One stream holds copy lists, in an extended sense compared to the Boldi and Vigna solution. Our copy lists are no longer binary but consist of four different flag symbols: 0 denotes an exact match (i.e., value j from the reference list occurs somewhere on the current list), 2 means that the current list contains integer $j + 1$, 3 means that the current list contains integer $j + 2$, if the corresponding integer from the reference list is j. Finally, the bits 1 correspond to the items from the reference list which have not been earlier labeled with 0, 2 or 3.

Alg. 1 GraphCompress($G, BSIZE$).

```
1   firstLine ← true
2   prev ← [ ]
3   outB ← [ ]
4   outF ← [ ]
5   for line ∈ G do
6       residuals ← line
7       if firstLine = false then
8           f[1 . . . |prev|] ← [1, 1, . . . , 1]
9           for i ← 1 to |prev| do
10              if prev[i] ∈ line then f[i] ← 0
11              else if prev[i] + 1 ∈ line then f[i] ← 2
12              else if prev[i] + 2 ∈ line then f[i] ← 3
13          append(outF, f)
14          for i ← 1 to |prev| do
15              if f[i] ≠ 1 then
16                  remove(residuals, prev[i])
17      residuals′ ← RLE(diffEncode(residuals)) + [0]
18      append(outB, byteEncode(residuals′))
19      prev ← line
20      firstLine ← false
21      if |outB| ≥ BSIZE then
22          compress(outB)
23          compress(outF)
24          outB ← [ ]
25          outF ← [ ]
26          firstLine ← true
```

Of course, several events may happen for a single element, e.g., the integer 34 from the reference list triggers three events if the current list contains 34, 35 and 36. In such case, the flag with the smallest value is chosen (i.e., 0 in our example).

Moreover, we make things even simpler than in the Boldi–Vigna scheme and our reference list is always the previous adjacency list.

The other stream stores residuals, i.e., the values which cannot be decoded with flags 0, 2 or 3 on the copy lists. First differential encoding is applied and then an RLE compressor for differences 1 only (with minimum run length set experimentally to 5) is run. The resulting sequence is terminated with a unique value (0) and then encoded using a byte code.

For this last step, we consider two variants. One is similar to *two-byte dense code* [18] in spending one bit flag in the first codeword byte to tell the length of the current codeword. Namely, we choose between 1 and b bytes for encoding each number, where b is the minimum integer such that $8b - 1$ bits are enough to encode any node value in a given graph. In practice it means that $b = 3$ for EU and $b = 4$ for the remaining available datasets.

The second coding variant can be classified as a prelude code [10] in which two bits in the first codeword byte tell the length of the current codeword; originally the lengths are 1, 2, 3 and 4 but we take 1, 2 and b such that $8b - 2$ bits are enough to encode the largest value in the given graph (i.e., b could be 5 or 6 for really huge graphs).

Once the residual buffer reaches at least BSIZE bytes, it is time to end the current block and start a new one. Both residual and flag buffers and then (independently) compressed (we used the well-known Deflate algorithm for this purpose) and flushed.

The code at Alg. 1 is slightly simplified; we omitted technical details serving for finding the list boundaries in all cases (e.g., empty lines).

5 Experimental results

We conducted experimented on the crawls EU-2005 and Indochina-2004, downloaded from the WebGraph project (`http://webgraph.dsi.unimi.it/`), using both direct and transposed graphs. The main characteristics of those datasets are presented in Table 1.

Dataset	EU-2005		Indochina-2004	
	direct	transposed	direct	transposed
Nodes	862664		7414866	
Edges	19235140		19235140	
Edges / nodes	22.30		26.18	
% of empty lists	8.31	0.000	17.66	0.004
Longest list length	6985	68922	6985	256425

Table 1. Selected characteristics of the datasets used in the experiments.

The main experiments (Sect. 5.1) were run on a machine equipped with an Intel Core 2 Quad Q9450 CPU, 8 GB of RAM, running Microsoft Windows XP (64-bit). Our algorithms were implemented in Java (JDK 6). A single CPU core was used by all implementations. As seemingly accepted in most reported works, we measure access time per edge, extracting many (100,000 in our case) randomly selected adjacency lists and summing those times, and dividing the total time by the number of edges on the required lists. The space is measured in bits per edge (bpe), dividing the total space of the structure (including entry points to blocks) by the total number of edges.

Throughout this section by 1 KB we mean 1000 bytes.

5.1 Compression ratios and access times

Our routine has three parameters: the number of flags used (either 2 or 4, where 2 flags mimic the Boldi–Vigna scheme and 4 correspond to Alg. 1), the byte encoding scheme (either using 2 or 3 codeword lengths), and the residual block size threshold BSIZE. As for the last parameter, we initially set it to 8192, which means that the residual block gets closed and is submitted to the Deflate compression once it reaches at least 8192 bytes. Experiments with the block size are presented in the next subsection. The remaining parameters constitute four variants:

2a Two flags and two codeword lengths are used.
2b Two flags and three codeword lengths are used.
4a Four flags and two codeword lengths are used.
4b Four flags and three codeword lengths are used.

Dataset	EU-2005		Indochina-2004	
	direct	transposed	direct	transposed
2a	2.286	2.345	1.101	1.087
2b	2.199	2.290	1.062	1.065
4a	1.735	1.809	0.936	0.903
4b	1.696	1.782	0.909	0.890

Table 2. Compression ratios in bits per edge.

As expected, the compression ratios improve with using more flags and more dense byte codes (Table 2). Tables 3 and 4 present the compression and access time results for the two extreme variants: 2a and 4b. Here we see that using more aggressive preprocessing is unfortunately slower (partly because of increased amount of flag data per block) and the difference in speed between variants 2a and 4b is close to 50%. Translating the times per edge into times per neighbor list, we need from 410 μs to 550 μs for 2a and from 620 μs to 760 μs for 4b. This is about 10 times less than the access time of 10K or 15K RPM hard disks.

	direct graph		transposed graph	
	bpe	time [μs]	bpe	time [μs]
BV (7,3)	5.169	0.24	–	–
2a	2.286	18.59	2.345	18.88
4b	1.696	28.93	1.782	27.83

Table 3. EU-2005 dataset. Compression ratios (bpe) and access times per edge. To the results of BV (7,3) the amount of 0.510 bpe should be added, corresponding to extra data required to access the graph in random order.

	direct graph		transposed graph	
	bpe	time [μs]	bpe	time [μs]
BV (7,3)	2.063	0.21	–	–
2a	1.101	20.77	1.087	21.10
4b	0.909	29.03	0.890	27.43

Table 4. Indochina-2004 dataset. Compression ratios (bpe) and access times per edge. To the results of BV (7,3) the amount of 0.348 bpe should be added, corresponding to extra data required to access the graph in random order.

5.2 Varying the block size

Obviously, the block size should seriously affect the overall space used by the structure and the access time. Larger blocks mean that the Deflate algorithm is more successful in finding longer matches and the overhead from encoding first lines in a block without any reference is smaller. On the other hand, more lines have to be usually decoded before extracting the queried adjacency list.

In this experiment we run the 2a algorithm (the same implementation in Java) with each block of residuals terminated (and later Deflate-compressed) after reaching BSIZE of 1024, 2048, 4096, 8192 and 16384 bytes, respectively. The test computer had an Intel Pentium4 HT 3.0 GHz CPU, 1 GB of RAM, and was running Microsoft Windows XP Home SP3 (32-bit). The results (Table 5) show that doubling the block size implies space reduction by about 10% while the access time grows less than twice (in particular, using 8K blocks is only 2.0–2.5 times slower than using 2K blocks). Still, as the block size gets larger (compare the last two rows in the table), the improvement in compression starts to drop while the slowdown grows. For a reference, the access times of a practical Boldi–Vigna variant, BV (7,3), are 0.47 μs and 0.42 μs on the test machine.

	EU-2005		Indochina-2004	
	bpe	time [μs]	bpe	time [μs]
1024	3.398	6.50	1.485	8.99
2048	2.869	8.91	1.292	12.05
4096	2.513	15.93	1.172	17.87
8192	2.286	27.60	1.101	29.83
16384	2.129	48.77	1.061	57.39

Table 5. Compression ratios and access times in function of the block size. 2a variant used. Tests run on the non-transposed graphs.

6 Obtaining forward and reverse neighbors

Sometimes one is interested in grasping not only the (forward) neighbors of a given node but also the nodes that point to the current node (also called its *reverse neighbors*). A naïve solution to this problem is to store a twin data structure built for the transposed graph, which more or less doubles the required space. Interestingly, as pointed out in Sect. 2, more sophisticated ideas are already known, using 2D structures that support bidirectional navigation over the graph.

In this section we propose two simple techniques for this problem scenario. One of them reduces the size of the compressed transposed graph for the price of moderate increase in search time. Basically, the idea is to remove parts of some adjacency lists from the transposed graph and refer to the compressed structure for the direct graph when there is a need to extract those removed reverse neighbors. In our preliminary experiments the transposed graph compressed component was reduced by less than 10% while for many lists the access time had to be approximately doubled (instead of extracting one compressed block, two randomly accessed blocks had to be extracted). Even if more can be done along these lines, we do not anticipate this approach being competitive.

The other algorithm partitions the binary matrix of the EU graph into squares, in the manner of the k^2-tree, but without any hierarchy, i.e., using only one level of blocks. Although seemingly very primitive, this idea let us attain the smallest space ever reported in the literature, for the EU dataset, among the algorithms supporting bidirectional navigation, namely 1.76 bpe, but the average extraction time per adjacency list is now on the order of a few milliseconds, i.e., close to hard disk access time. This is, in a way, an extreme result; a slower algorithm could already lose in speed to a plain external representation.

In an experiment, we partitioned the binary matrix M of the EU graph ($n = 862,664$ nodes) into boxes (squares) of size $B = 1024$ (the boundary areas may be rectangular). Each box is identified with a single bit (totalling 89 KB) where 1s stand for the non-empty boxes (those that contain at least one edge). The non-empty boxes, obtained in a row-wise scan, are labeled with successive integers, which are offsets in an array $A[1 \ldots |A|]$ of pointers to the actual (compressed) content of the corresponding boxes. Now we present how forward and reverse neighbors of a given page are found.

To find the forward neighbors of page j, we must retrieve and decode all the non-empty boxes overlapping the jth row of the matrix. Note that for efficient retrieval we need only to find quickly in the array A the pointer to first (leftmost) such box as all its successors will be pointed from the following cells of A. A trivial yet satisfying

solution is to store in an extra array the indexes in A of the leftmost non-empty boxes for the following rows of boxes. This needs n/B indexes (about 3.3 KB for the EU graph if 4-byte indexes are used).

Finding the reverse neighbors is harder but we avoid the challenge and solve it trivially, storing an array analogous to A, only built according to the column-wise scan. For the EU graph and our choice of B, the number of non-empty blocks is about 24,700, i.e., the extra cost in space is just above 100 KB (4-byte pointers and the 3.3 KB of the auxiliary array).

As mentioned, the non-empty boxes are stored in compressed form. We (conceptually) flatten each box, writing it row after row, and encode the gaps between the successive 1s. The gaps are represented with a byte code (1, 2 or 3 bytes per gap). Finally, the sequence of encoded gaps is compressed with the Deflate algorithm. To improve compression, for each non-empty box we check if transposing it results in a smaller Deflate-compressed size and also if it is better not to compress it at all (data expansion is typical if the box contains only a few items). This adds two extra bits per non-empty box.

Note that accessing the (forward or reverse) neighbors of a given page requires decoding many boxes, even those that have no item in common with the desired neighbor list. For the EU graph a single list passes through about 29 non-empty boxes, on average. The average non-empty box occupies a little over 1.3 KB before Deflate compression (their size variance is however very large), which means that retrieving the neighbor list requires extracting compressed data to about 39 KB, on average. This estimation is optimistic since decompressing a single chunk of data is usually faster than of several chunks totalling the same size, because of the locality of memory accesses. Moreover, as said, those average-case estimations are far from the worst case. Yet another factor is that the decoded boxes must be filtered to return only those values which belong to the desired list. Overall, we however believe that one can retrieve the neighbor list in about 2–3 ms (i.e., about 100 microseconds per neighbor) on modern hardware in an average case.

The total size of the EU graph compressed in the presented way is about 4,225 KB, which translates to 1.76 bpe. This contrasts with 3.93 bpe presented as the most succinct result in [8], for which graph representation the average reported direct and reverse edge retrieval time is about 35 and 55 microseconds, respectively. As the number of edges per adjacency list is about 22 for this graph, the times to extract the whole list are close to 1 ms which is not that far from our (very crudely estimated) retrieval times. This leads us to the conclusion that even simple heuristics and off-the-shelf tools (like the Deflate compression algorithm) may help one get close to the state of the art and should encourage researchers to rethink the problem.

7 Conclusions

We presented two algorithms for Web graph compression, one encoding blocks consisting of whole lines and the other working on boxes (squares) of the graph binary matrix. Both algorithms achieve much better compression results than those presented in the literature, although for the price of relatively slow access time. We point out, however, that one extreme tradeoff in succinct in-memory data structures is when accessing the structure is only slightly faster than reading data from disk. The niche for such a solution is when the given Web crawl cannot fit in RAM memory using less tight compressed representation and the stronger compression is already

enough. The disk transfer rate is of relatively small imporantance here and what matters is the access time, which is about 10 ms or more for commodity 7200 RPM hard disks. Our algorithms spend significantly less time for extracting an average adjacency list, even if they are 1 or 2 orders of magnitude slower than the solutions from [4,7,8]. Another challenge is to compete with SSD disks which are not much faster than conventional disks in reading or writing sequential data but their access times are two orders of magniture smaller.

Our future work will focus on improving the access times; some possibilities lie in more aggressive reference list encoding via referring to several (cf. [4]) rather than a single previous list, using smaller independently compacted blocks with backend compression applied over many of them, and replacing Deflate with alternative compressors from LZ77 family, e.g. LZMA (`http://www.7-zip.org/`).

8 Acknowledgments

The work was partially supported by the Polish Ministry of Science and Higher Education under the project N N516 477338 (2010–2011).

References

1. V. N. Anh and A. F. Moffat: *Local modeling for webgraph compression*, in DCC, J. A. Storer and M. W. Marcellin, eds., IEEE Computer Society, 2010, p. 519.
2. Y. Asano, Y. Miyawaki, and T. Nishizeki: *Efficient compression of web graphs*, in COCOON, X. Hu and J. Wang, eds., vol. 5092 of Lecture Notes in Computer Science, Springer, 2008, pp. 1–11.
3. K. Bharat, A. Z. Broder, M. R. Henzinger, P. Kumar, and S. Venkatasubramanian: *The Connectivity Server: Fast access to linkage information on the Web.* Computer Networks, 30(1–7) 1998, pp. 469–477.
4. P. Boldi and S. Vigna: *The webgraph framework I: Compression techniques*, in WWW, S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, eds., ACM, 2004, pp. 595–602.
5. N. Brisaboa, S. Ladra, and G. Navarro: *K2-trees for compact web graph representation*, in Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 5721, Springer, 2009, pp. 18–30.
6. G. Buehrer and K. Chellapilla: *A scalable pattern mining approach to web graph compression with communities*, in WSDM, M. Najork, A. Z. Broder, and S. Chakrabarti, eds., ACM, 2008, pp. 95–106.
7. F. Claude and G. Navarro: *Fast and compact Web graph representations*, Tech. Rep. TR/DCC-2008-3, Department of Computer Science, University of Chile, April 2008.
8. F. Claude and G. Navarro: *Extended compact web graph representations*, in Algorithms and Applications, T. Elomaa, H. Mannila, and P. Orponen, eds., vol. 6060 of Lecture Notes in Computer Science, Springer, 2010, pp. 77–91.
9. F. Claude and G. Navarro: *Fast and compact web graph representations.* ACM Transactions on the Web (TWEB), 2010, To appear.
10. J. S. Culpepper and A. Moffat: *Enhanced byte codes with restricted prefix properties*, in SPIRE, M. P. Consens and G. Navarro, eds., vol. 3772 of Lecture Notes in Computer Science, Springer, 2005, pp. 1–12.
11. D. Donato, L. Laura, S. Leonardi, U. Meyer, S. Millozzi, and J. F. Sibeyn: *Algorithms and experiments for the webgraph.* J. Graph Algorithms Appl., 10(2) 2006, pp. 219–236.
12. R. F. Geary, N. Rahman, R. Raman, and V. Raman: *A simple optimal representation for balanced parentheses*, in Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings, S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, eds., vol. 3109 of Lecture Notes in Computer Science, Springer–Verlag, 2004, pp. 159–172.

13. X. He, M.-Y. Kao, and H.-I. Lu: *A fast general methodology for information-theoretically optimal encodings of graphs*. SIAM J. Comput., 30(3) 2000, pp. 838–846.
14. G. Jacobson: *Succinct Static Data Structures*, PhD thesis, Carnegie Mellon University, 1989.
15. N. J. Larsson and A. Moffat: *Off-line dictionary-based compression*. Proceedings of the IEEE, 88(11) Nov. 2000, pp. 1722–1732.
16. J. I. Munro and V. Raman: *Succinct representation of balanced parentheses, static trees and planar graphs*, in IEEE Symposium on Foundations of Computer Science (FOCS), 1997, pp. 118–126.
17. G. Navarro and V. Mäkinen: *Compressed full-text indexes*. ACM Computing Surveys, 39(1) 2007, p. article 2.
18. P. Procházka and J. Holub: *New word-based adaptive dense compressors*, in IWOCA, J. Fiala, J. Kratochvíl, and M. Miller, eds., vol. 5874 of Lecture Notes in Computer Science, Springer, 2009, pp. 420–431.
19. K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener: *The link database: Fast access to graphs of the Web*, 2001.
20. H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara: *A large-scale study of link spam detection by graph algorithms*, in AIRWeb '07: Proceedings of the 3rd international workshop on Adversarial information retrieval on the web, New York, NY, USA, 2007, ACM, pp. 45–48.
21. G. Turán: *On the succinct representation of graphs*. Discrete Applied Math, 15(2) May 1984, pp. 604–618.

New Simple Efficient Algorithms Computing Powers and Runs in Strings

Maxime Crochemore[1,3], Costas Iliopoulos[1,4], Marcin Kubica[2], Jakub Radoszewski[⋆,2], Wojciech Rytter[2,5], Krzysztof Stencel[2,5], and Tomasz Waleń[2]

[1] King's College London, London WC2R 2LS, UK
maxime.crochemore@kcl.ac.uk, csi@dcs.kcl.ac.uk
[2] Dept. of Mathematics, Computer Science and Mechanics, University of Warsaw, Warsaw, Poland
[kubica,jrad,rytter,stencel,walen]@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics, Copernicus University, Toruń, Poland

Abstract. Three new simple $O(n \log n)$ time algorithms related to repeating factors are presented in the paper. The first two algorithms employ only a basic textual data structure called the Dictionary of Basic Factors. Despite their simplicity these algorithms not only detect existence of powers but also find all primitively rooted cubes (as well as higher powers) and all cubic runs. Our third $O(n \log n)$ time algorithm computes all runs and is probably the simplest known efficient algorithm for this problem. It uses additionally the Longest Common Extension function, however, due to relaxed running time constraints, a simple $O(n \log n)$ time implementation can be used. At the cost of logarithmic factor (in time complexity) we have novel algorithmic solutions for several classical string problems which are much simpler than (usually quite sophisticated) linear time algorithms.

Keywords: run, repetition, square, cube (in a string), Dictionary of Basic Factors

1 Introduction

In this paper, we present algorithms finding various types of repetitions in a string: powers (e.g. squares or cubes), cubic runs and runs. Finding repetitions is a fundamental problem in text processing and has numerous applications. Examples of such applications, an explanation of the motivation and related topics can be found in the survey [8].

Various problems related to finding repetitions in a string have already been studied. For the problem of finding all distinct squares, a linear time algorithms are known [14,17,18]. It is also known, that the maximal number of distinct squares in a string is linear [13].

Multiple approaches to searching for squares in a string can be found in the literature, however most of the existing algorithms are rather complex. The first approach, is to check if a string is square-free. $O(n \log n)$ time algorithms for this problem have been presented in [23,24] (the latter one is randomized). The optimal $O(n)$ time algorithms are described in [5,23].

⋆ Corresponding author. Some parts of this paper were written during the corresponding author's Erasmus exchange at King's College London

Maxime Crochemore, Costas Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Krzysztof Stencel, Tomasz Waleń: New Simple Efficient Algorithms Computing Powers and Runs in Strings, pp. 138–149 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

Another approach is to find all occurrences of primitively rooted squares in a string. A number of $O(n \log n)$ time algorithms reporting all such occurrences can be found in [2,4,19,22,25,26]. Due to the lower bound shown in [6] these algorithms are optimal.

Yet another approach is to report all occurrences of squares in a string. If we denote the number of such occurrences by z, then both $O(n \log n + z)$ time algorithms [20,22,26] and $O(n + z)$ time algorithms [14,17,18] are known for this problem.

Finally, there are recent results related to on-line square detection (that is, when letters of u are given one by one), improving the time complexity from $O(n \log^2 n)$ [21] to $O(n \log n)$ [16] and $O(n)$ [3].

Let u be a string of length n over a bounded alphabet. In Section 3 a very simple $O(n \log n)$ time algorithm checking whether u contains any kth string power is presented. The algorithm reports all occurrences of primitively rooted kth powers for any $k \geq 3$, in particular, primitively rooted cubes. As a by-product we obtain an alternative, algorithmic proof of the fact [6] that the maximal number of such occurrences is $O(n \log n)$. The output of the algorithm is later on used to list all cubic runs in u, within the same time complexity.

From the aforementioned literature, the papers [2,4,22] deal also with powers of arbitrary (integer) exponent, however the techniques used there (e.g., suffix trees, Hopcroft's factor partitioning) are much more sophisticated than the techniques applied in this paper. The $O(n \log n)$ time algorithm for a single square detection from [23] is in some sense similar to the algorithm presented in this paper. However it is less versatile than ours: we see no simple modification adapting it to detect all occurrences of primitively rooted higher powers.

In Section 4, we present an application of the algorithm finding all occurrences of primitively rooted cubes to find all cubic runs. This algorithm also runs in $O(n \log n)$ time and it does not use any additional advanced techniques.

Finally, in Section 5, we give an algorithm reporting all runs in a string, in $O(n \log n)$ time. It is significantly simpler than all known $O(n \log n)$ time algorithms present implicitly in [2,4,22] and than the optimal $O(n)$ time algorithm [17,18]. The only non-trivial technique used in our algorithm is the Longest Common Extension function. It can be either implemented as described in [10,15] — very efficiently, but using quite sophisticated machinery, or less efficiently, but in a much simpler way, what is sufficient to obtain $O(n \log n)$ time complexity.

2 Preliminaries

We consider *words* (*strings*) over a bounded alphabet Σ, $u \in \Sigma^*$. The empty word is denoted by ε. The positions in u are numbered from 1 to $|u|$. For $u = u_1 u_2 \dots u_n$, by $u[i\,..\,j]$ we denote a *factor* of u equal to $u_i \dots u_j$ (in particular $u[i] = u[i\,..\,i]$). Words $u[1\,..\,i]$ are called prefixes of u, words $u[i\,..\,n]$ suffixes of u, whereas words that are both a prefix and a suffix of u are called *borders* of u.

We say that a positive integer p is a *period* of the word $u = u_1 \dots u_n$ if $u_i = u_{i+p}$ holds for all i, $1 \leq i \leq n - p$. Periods and borders correspond to each other, i.e. u has a period p if and only if it has a border of length $n - p$, see e.g. [7,12].

A *run* (also called a maximal repetition) in a string u is such an interval $[i\,..\,j]$, that:

- the shortest period p of the associated factor $u[i\,..\,j]$ satisfies $2p \leq j - i + 1$,

– the interval can be extended neither to the left nor to the right, without violating the above property, that is, $u[i-1] \neq u[i+p-1]$ and $u[j-p+1] \neq u[j+1]$, provided that the respective characters exist.

A *cubic run* is a run $[i\,..\,j]$ for which the shortest period p satisfies $3p \leq j-i+1$. We identify a run (or a cubic run) with a corresponding triple (i, j, p).

If $w^k = u$ (k is a positive integer) then we say that u is the kth power of the word w. A *square* (*cube*) is the 2nd (3rd) power of some nonempty word.

2.1 Dictionary of Basic Factors

Dictionary of Basic Factors is a simple, yet powerful data-structure. It is widely used in this paper. For a word u of length n, the *Dictionary of Basic Factors* of u (denoted by $\mathsf{DBF}(u)$) consists of a sequence of arrays $Name_t[\,]$, for $0 \leq t \leq \lfloor \log n \rfloor$. Array $Name_t[\,]$ contains information about factors of u of length 2^t — $Name_t[i]$ contains information about word $u[i\,..\,i+2^t-1]$, for $1 \leq i \leq n-2^t+1$. More precisely, value of $Name_t[i]$ is the rank of $u[i\,..\,i+2^t-1]$ among other factors of length 2^t. Hence, values of elements of all the arrays $Name_t[\,]$ are in the range from 1 to n. The important property of $\mathsf{DBF}(u)$, that we exploit, is that $u[i\,..\,i+2^t-1] \leq u[j\,..\,j+2^t-1]$ if and only if $Name_t[i] \leq Name_t[j]$. DBF has a variety of known applications in the field of text and sequence algorithms, see e.g. [11].

$\mathsf{DBF}(u)$ requires $O(n \log n)$ space and can be constructed in $O(n \log n)$ time [12]. $Name_0[\,]$ contains information about consecutive characters of u. So, $Name_0[\,]$ can be computed in $O(n)$ time, by sorting all the letters appearing in u and mapping characters of u to numbers from 1 on. Having computed $Name_t[\,]$, one can easily compute $Name_{t+1}[\,]$ in $O(n)$ time. Factor $u[i\,..\,i+2^{t+1}-1]$ is a concatenation of factors $u[i\,..\,i+2^t-1]$ and $u[i+2^t\,..\,i+2^{t+1}-1]$. Hence, it can be represented by a pair $(Name_t[i], Name_t[i+2^t])$. Then, all such pairs can be sorted lexicographically (in $O(n)$ time) and mapped onto their ranks, that is integers from 1 on. Figure 1 shows the DBF for an example string.

Text	a	b	b	a	a	b	b	a	b	b	a
$Name_0[\,]$	1	2	2	1	1	2	2	1	2	2	1
	(1,2)	(2,2)	(2,1)	(1,1)	(1,2)	(2,2)	(2,1)	(1,2)	(2,2)	(2,1)	
$Name_1[\,]$	2	4	3	1	2	4	3	2	4	3	
	(2,3)	(4,1)	(3,2)	(1,4)	(2,3)	(4,2)	(3,4)	(2,3)			
$Name_2[\,]$	2	5	3	1	2	6	4	2			
	(2,2)	(5,6)	(3,4)	(1,2)							
$Name_3[\,]$	2	4	3	1							

Figure 1. Example of DBF computation for word *abbaabbabba*. Factor *abba* appears three times in this word and is represented in $Name_2[\,]$ by 2.

Using DBF, one can compare factors of arbitrary length, as given in the following Lemma, see [12].

Lemma 1. *Having precomputed* $\mathsf{DBF}(u)$*, any two factors of* u *can be compared in* $O(1)$ *time.*

Proof. Let $u[i\,..\,j]$ and $u[i'\,..\,j']$ be the two factors that should be compared. We can assume, that $j-i = j'-i'$, since otherwise they have different lengths, cannot be equal and can be compared by trimming the longer factor.

Let t be such an integer, that $2^t \leq j - i < 2^{t+1}$. Then, it is enough to compare $u[i\,..\,i+2^t-1]$ with $u[i'\,..\,i'+2^t-1]$, and $u[j-2^t+1\,..\,j]$ with $u[j'-2^t+1\,..\,j']$. This however can be done by comparing $Name_t[i]$ with $Name_t[i']$, and $Name_t[j-2^t+1]$ with $Name_t[j'-2^t+1]$. □

Later on, we show, that the memory complexity of the presented algorithm can be reduced to $O(n)$, although it uses $\mathsf{DBF}(u)$. To do this, we cannot store all the arrays $Name_t[\,]$. Instead, we should store just a fixed number (e.g. one) of such arrays, and design the algorithm in such a way, that are used in the ascending order of t. Then, new arrays can be computed when needed, replacing previously used arrays. Still, it is possible to compare factors in $O(1)$ time, provided that the ratio between the length of the compared factors and 2^t is bounded, as expressed by the following Lemma.

Lemma 2. *Let t be a fixed number between 0 and $\lfloor \log n \rfloor$, and let $Name_t[\,]$ be one of the arrays constituting $\mathsf{DBF}(u)$. It is possible to compare factors of u of length l, using just $Name_t[\,]$, in constant time, provided that: $l \geq 2^t$ and $\frac{l}{2^t} = O(1)$.*

Proof. The proof is similar to the proof of the previous Lemma. The compared factors can be covered using $O(1)$ factors of length 2^t. Hence, it is enough to compare $O(1)$ pairs of elements of $Name_t[\,]$. □

3 Detecting String Powers

Let u be a word of length n. The following algorithm tests if u contains a kth power, for $k \geq 2$. It exploits $\mathsf{DBF}(u)$ and two other auxiliary data-structures, denoted by POWERS and $Prev$.

POWERS is a list on which the result is accumulated. Each occurrence of a power of the form $(u[pos\,..\,pos+root-1])^k$ (i.e. kth power of a factor of length $root$, starting at position pos) is represented by a pair $(root, pos)$. The output of the algorithm is a list of pairs denoting kth powers. We allow the same power to be inserted multiple times — at the end the list is sorted and the repetitions are removed.

$Prev[1\,..\,n]$ is an array of positions in the text, such that $Prev[Name_t[j]]$ is the most recent occurrence of $u[j\,..\,j+2^t-1]$ preceding j, or -1 if there is none.

For all values of t, the algorithm scans the text, and for each position j it checks (in constant time) if factors of u of length 2^t, starting at $Prev[Name_t[j]]$ and j generate a power. Examples of how the algorithm works can be found in Fig. 2 and Table 1.

```
Algorithm DetectPowers(u, n, k)
 1: {detect kth string powers in a word u, |u| = n}
 2: Name ← DBF(u)
 3: POWERS ← ∅
 4: for t ← 0 to ⌊log n⌋ do
 5:    Prev ← (0, 0, ..., 0)
 6:    for j ← 1 to n − 2^t + 1 do
 7:       name ← Name_t[ j ]
 8:       pos ← Prev[ name ]
 9:       root ← j − pos
10:       if u[pos..pos + k · root − 1] is (really) a kth power
             {constant time test due to DBF} then
11:          POWERS.insert((root, pos))
12:       Prev[ name ] ← j
13: RadixSort(POWERS) with repetitions removed
14: return POWERS
```

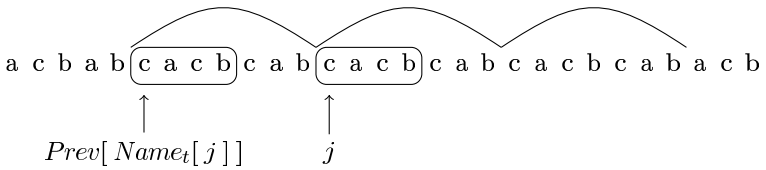


Figure 2. The basic factor *cacb* of rank $t = 2$ at position $j = 13$ generates a cube $(cacbcab)^3$ starting at position $pos = 6$. The same cube is generated for $t = 3$ and $j = 13$, for the basic factor *cacbcabc*.

a	b	b	a	b	a	a	b	b	a	a	b	a	b	b	a	b	a	a	b	a	b	b	a	a	b	b	a	b	a	a	b
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Table 1. For the Thue-Morse word of length 32 the algorithm reports the following squares (pairs $(root, pos)$): $t = 0$: (1, 2), (2, 3), (1, 6), (1, 8), (1, 10), (2, 11), (1, 14), (2, 15), (1, 18), (2, 19), (1, 22), (1, 24), (1, 26), (2, 27), (1, 30); $t = 1$: (4, 5), (3, 12), (3, 16), (4, 21); $t = 2$: (8, 9). In particular, the algorithm reports all squares in the Thue-Morse words.

In the analysis of the algorithm we use some combinatorics of primitive words. The *primitive root* of a word u is the shortest word w, such that $w^k = u$ for some positive integer k. We call a word u *primitive* if it equals its primitive root, otherwise it is called *non-primitive*. Primitive words admit a so-called *synchronizing property*, as given in the following Lemma, see [7].

Lemma 3 (Synchronizing property of primitive words). *A nonempty word is primitive if and only if it occurs as a factor in its square only as a prefix and a suffix.*

Theorem 4. *DetectPowers algorithm reports only primitively rooted powers in the word u.*

Proof. Obviously, all positions reported by the algorithm represent kth powers. Thus we only need to show that no non-primitively-rooted powers are reported.

Consider lines 7–12 of the algorithm, for some t and j. Assume that $pos \neq 0$. To conclude the proof of the theorem, it suffices to show that the word $w \stackrel{\text{def}}{=} u[pos\,..\,j-1]$ is always primitive.

Assume to the contrary, that $w = v^m$, for some v and $m \geq 2$. Let $z = u[j\,..\,j + 2^t - 1] = u[pos\,..\,j + 2^t - 1]$. There are two cases (see Fig. 3):

a) Let us assume, that $|w| \geq 2^t$. Then z is a prefix of of w and $|v|$ is a period of z.
b) Let us assume, that $|w| < 2^t$. Then w is a prefix of z and $u[j\,..\,j + 2^t - |w| - 1]$ is a border of z.

In both cases z appears also at position $j-|v|$. Hence, $Prev[Name_t[j]] \geq j-|v| > pos$, this contradiction concludes the proof. □

The following two theorems conclude that the algorithm correctly checks if the word contains any kth power (i.e., whether the word is kth-power-free or not), and also reports (among others) all kth powers of specific type, depending on the value of parameter k (2 or ≥ 3).

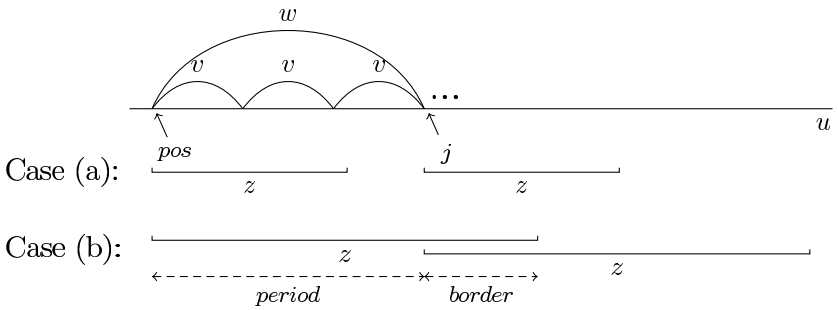


Figure 3. Illustration of the proof of Theorem 4; case (a): $|z| \leq |w|$, case (b): $|z| > |w|$.

Theorem 5. *For $k = 2$, the DetectPowers algorithm finds all occurrences of shortest squares in u.*

Proof. Let v^2 be any shortest square occurring in u at position i. Note that v must be primitive. Let s be such an integer, that $2^s \leq |v| < 2^{s+1}$. Consider the step of the algorithm in which $t = s$, $j = i + |v|$. We show that the algorithm reports the square v^2 in this step, i.e., $pos = i$. Obviously $pos \geq i$, hence it suffices to show that this value cannot be greater than i.

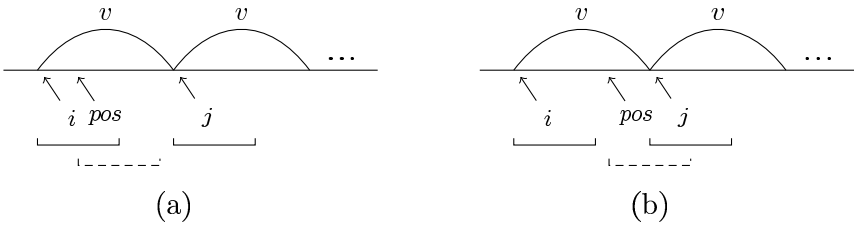


Figure 4. Illustration of the proof of Theorem 5. (a) If $pos - i < 2^t$ then the occurrences of w at positions i and pos overlap. (b) If $pos - i \geq 2^t$ then the occurrences of w at positions pos and j overlap.

If this was the case, the factor $w \stackrel{\text{def}}{=} u[j\,..\,j + 2^t - 1]$ would occur in u at positions i, pos and j, thus forming an overlap, see Fig. 4. However, an overlap of a string of length 2^t corresponds to a square in u with primitive root shorter than 2^t, what contradicts the fact that v^2 is the shortest square in u. □

Theorem 6. *For a given $k \geq 3$, the DetectPowers algorithm finds all occurrences of primitively rooted kth powers in u.*

Proof. Assume that there is an occurrence of v^k, for v primitive, which starts at position i in u. Let integer s be defined as $2^{s-1} < |v| \leq 2^s$. Let us consider the step of the algorithm in which $t = s$, $j = i + |v|$. We show that in this step $pos = i$, this concludes that the considered power is reported by the algorithm.

Let us note that $pos \geq i$, since $2^t < 2|v|$, and therefore:

$$u[i\,..\,i + 2^t - 1] \;=\; u[j\,..\,j + 2^t - 1] \;\stackrel{\text{def}}{=}\; w,$$

see Fig. 5a. We prove the inequality $pos \leq i$ by contradiction. Assume that $i < pos < j$. Then the prefix of w of length $|v|$, that is the word v, would occur in u at positions:

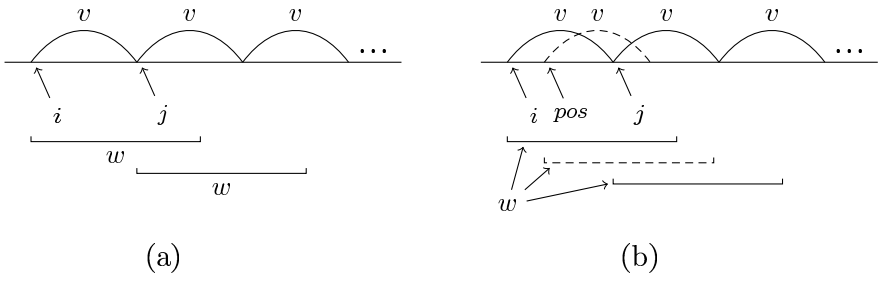


Figure 5. Illustration of the proof of Theorem 6

i, pos and j (see Fig. 5.b). This is not possible, however, due to the synchronizing property of primitive words (Lemma 3). □

Remark. It is easy to see that the stronger claim (from Theorem 6) does not hold in the case of $k = 2$ (Theorem 5). That is, not all primitively rooted squares are detected by the algorithm. Among others, the squares for which the primitive root admits a very long border, e.g., $((ab)^m a)^2$, may not be reported.
Finally, let us consider the complexity of the algorithm DetectPowers.

Theorem 7. *The time complexity of the DetectPowers algorithm is $O(n \log n)$. Moreover, for $k = O(1)$ the algorithm can be modified to require only $O(n)$ space and still satisfy the properties from Theorems 4–6.*

Proof. The analysis of the time complexity is straightforward — the outer loop of the algorithm makes $O(\log n)$ iterations and in each iteration the inner loop runs in $O(n)$ time.

The space complexity of the presented implementation is also $O(n \log n)$ due to the space requirements of the DBF, however it can be reduced to $O(n)$ if only two consecutive rows of the table $Name$ are stored in the memory.

This causes a difficulty only in the kth-power-test in line 8, since the value of $root$ can be arbitrary. However, along with the proofs of Theorems 5 and 6, we can immediately return $false$ in the test if the parameter $root$ is not in the interval $[2^t, 2^{t+1})$ (for squares) or the interval $(2^{t-1}, 2^t]$ (for higher powers), and still the output of the algorithm will fulfill the requirements.

Thus the predicate reduces to testing equality of words of length $(k-1) \cdot root = c \cdot 2^{t-1}$, where $1 \leq c = O(1)$ for $k = O(1)$, thus can be performed using only $Name_{t-1}[\,]$ in constant time (Lemma 2). □

4 Application of the DetectPowers Algorithm for Cubic Runs

In this section we show how to use the output of the DetectPowers algorithm to compute, in a simple manner, all cubic runs in a string u of length n in $O(n \log n)$ time. Cubic runs [9] are special type of runs in which the period is at least 3 times shorter than the run, hence they characterize strong periodic properties of a word.

Let L be the output of the DetectPowers algorithm for u and $k = 3$. It is a sorted list of pairs with repetitions removed. Moreover, without the loss of generality, we can assume, that it is sorted in ascending lexicographical order of pairs. Let us

define a *special sublist* of L as a maximal continuous subsequence of L of the form $(per, i),\ (per, i+1),\ \ldots,\ (per, i+s)$. Note, that such a sublist corresponds to a cubic run $(i,\ i+s+3\cdot per-1,\ per)$.

Example 8. For the following list of pairs $(root, pos)$:

$$L = (2,3),\ (2,4),\ (2,5),\ (4,8),\ (4,9),\ (4,28),\ (4,29),\ (4,30),\ (4,31),\ (5,18)$$

the corresponding cubic runs are:

$$(3,10,2),\ (8,20,4),\ (28,42,4),\ (18,32,5).$$

Thus we obtain:

Restriction 9. *There is a bijection between special sublists of L and cubic runs.*

The following algorithm scans the list of cubes and glues together its special sublists into cubic runs, utilizing Observation 9.

```
Algorithm DetectCubicRuns(u, n)
 1: {list all cubic runs in the word u, |u| = n}
 2: L ← DetectPowers(u, n, 3)
 3: L.append((−1, −1))
 4: CRUNS ← ∅
 5: prev_root ← prev_pos ← start ← −1
 6: for all (root, pos) ∈ L do
 7:     if (root, pos) = (prev_root, prev_pos + 1) then
 8:         prev_pos ← pos
 9:     else if start ≥ 0 then
10:         CRUNS.insert((start, prev_pos + 3 · prev_root − 1, prev_root))
11:         start ← prev_pos ← pos
12:         prev_root ← root
13: return CRUNS
```

Theorem 10. *The DetectCubicRuns algorithm computes all cubic runs in a string u of length n in $O(n \log n)$ time.*

Proof. Due to Theorem 7, line 2 of the algorithm runs in $O(n \log n)$ time. The time complexity of the rest of the algorithm is $O(|L|+n)$, where $|L|$ denotes the number of elements in the list L. Time complexity of lines 6–13 is clearly $O(|L|)$. Finally, again due to Theorem 7, $|L| = O(n \log n)$, which yields $O(n \log n)$ total time complexity of the DetectCubicRuns algorithm.

Due to Theorems 4 and 6, the list L contains all occurrences of all primitively rooted cubes in the word u. In the for-all-loop (lines 6–12) the algorithm glues together cubes forming special sublists of L, thus forming the same cubic run. Hence, the output of the algorithm comprises exactly all the cubic runs in u. □

5 Detecting Runs

In this section we describe a different, however still very simple algorithm which reports all (ordinary) runs in a string u of length n in $O(n \log n)$ time. In the following pseudocode, in the for-loop we consider candidates for runs with period *per*. Verification of existence of runs is performed using *the longest common prefix* (*lcpref* in

short) and *the longest common suffix* ($lcsuf$ in short) queries, also called *longest common extension* queries (see also Fig. 6). Here, $lcpref(a, b)$ denotes the length of the longest common prefix of suffixes $u[a\,..\,n]$ and $u[b\,..\,n]$, similarly $lcsuf(a, b)$ denotes the length of the longest common suffix of prefixes $u[1\,..\,a]$ and $u[1\,..\,b]$.

The obtained list of candidates RUNS may contain the same run listed several times and additionally with periods being multiples of its shortest period. Therefore, in the end (lines 11–14) we remove such repetitions, leaving at most one triple (i, j, per) for given i, j, with the smallest corresponding value of period per.

```
Algorithm DetectRuns(u, n)
 1: {list all runs in the word u, |u| = n}
 2: RUNS ← ∅
 3: for per ← 1 to n div 2 do
 4:    pos ← per
 5:    while pos + per ≤ n do
 6:       left ← lcsuf(pos, pos + per)
 7:       right ← lcpref(pos, pos + per)
 8:       if left + right > per then
 9:          RUNS.insert((pos − left + 1, pos + per + right − 1, per))
10:       pos ← pos + per
11: RadixSort(RUNS) {triples sorted lexicographically}
12: prev ← (−1, −1);
13: for all (i, j, per) ∈ RUNS do
14:    if prev = (i, j) then RUNS.delete((i, j, per)); else prev ← (i, j)
15: return RUNS
```

The following theorem shows correctness of the DetectRuns algorithm, i.e., that it computes exactly all distinct runs in a string.

Theorem 11.

(a) Each run (a, b, q) in the word u is inserted to the list RUNS (line 9) at least once.
(b) Every triple (a, b, p) inserted to RUNS in line 9 of the algorithm corresponds to a run (a, b, q) in u with $q \mid p$.

Proof. (a) Let $a + r$, for $0 \le r < q$, be any of the first q positions of the run. Then, by the definition of a run, the following inequalities hold, see Fig. 6:

$$\begin{aligned} lcsuf(a + r, a + r + q) &= r + 1 \\ lcpref(a + r, a + r + q) &\ge q - r. \end{aligned}$$

Hence, if $per = q$ and $pos = a + r$ then the condition in line 8 of the algorithm is true and the run (a, b, q) is reported. However, this happens for $r \equiv -a \pmod{per}$, i.e., in the mth step of the while-loop, where $m = \lceil a/per \rceil$.

(b) Clearly any triple (a, b, p) inserted into the list RUNS in line 9 of the algorithm corresponds to an interval $[a\,..\,b]$ in u with period (not necessarily shortest) equal to p and repeating at least twice within the interval, i.e., $2p \le b - a + 1$. Moreover, this interval is not extendable to either side without violating this periodicity.

Let q be the shortest period of the factor $u[a\,..\,b]$. Note that $q \mid p$, since otherwise, by Fine & Wilf's Periodicity Lemma [7,12], $\gcd(p, q)$ would be a shorter period of this factor. To show that $[a\,..\,b]$ is a run with period q, it suffices to prove that this interval is not extendable to either size with regard to the period q.

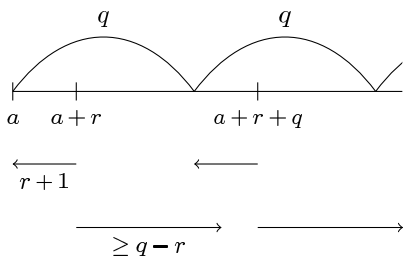


Figure 6. Graphical interpretation of $lcsuf(a+r, a+r+q)$ and $lcpref(a+r, a+r+q)$ for a run (a, b, q) and $0 \leq r < q$

Assume to the contrary that the interval is extendable to the left (the other case is analogical). Then we have:

$$u[a-1] \;=\; u[a-1+q] \;=\; u[a-1+q+(p/q-1)\cdot q] \;=\; u[a-1+p]$$

and consequently $[a\,..\,b]$ would be extendable to the left w.r.t. the period p, a contradiction. □

Now let us analyze the time complexity of the algorithm. It mostly depends on the time complexity of the $lcpref$ and $lcsuf$ queries. Their efficient implementation involves the Longest Common Prefix (LCP) and Suffix Arrays (SUF) (computed in $O(n)$ time), and Range Minimum Queries (RMQ) (with $O(n)$ preprocessing time and $O(1)$ query time) [10]. Techniques used in the efficient implementation of these data structures are rather complex. However, without increasing overall time complexity of the algorithm, we can compute the Suffix Array and preprocess RMQ in $O(n \log n)$ time. Hence, we can use much simpler machinery.

The Suffix Array of u can be computed in $O(n \log n)$ time using $\mathsf{DBF}(u)$ — all the suffixes are sorted lexicographically and can be compared in $O(1)$ time using $\mathsf{DBF}(u)$. Then, the LCP array can be simply computed in $O(n)$ time using the Suffix Array.

Preprocessing of RMQ data-structure in $O(n \log n)$ time resembles computation of DBF a lot. The main difference is that instead of computing ranks of factors we compute positions of minimal elements in ranges. Then, we can find a minimum in the given range by covering the given range by two ranges whose size is a power of two, and comparing their minimal elements. Hence, $O(1)$ query time is preserved.

Thus we obtain $O(n \log n)$ preprocessing time and $O(1)$ query time for the $lcpref$ and $lcsuf$ queries, what yields the time complexity of the algorithm specified in the following theorem.

Theorem 12. *The time complexity of the DetectRuns algorithm is* $O(n \log n)$.

Proof. For a given value of per, the while-loop performs at most n/per steps, each in constant time. The time complexity of the for-loop is therefore

$$O\left(\sum_{per=1}^{\lfloor n/2 \rfloor} \frac{n}{per}\right) = O(n \log n)$$

and this is also the maximum size of the list RUNS.

All remaining operations in the algorithm are: $lcpref/lcsuf$ preprocessing which is performed in $O(n \log n)$ time, and sorting and removing duplicates from RUNS, both performed in $O(|\mathsf{RUNS}|) = O(n \log n)$ time. In total, we obtain the aforementioned time complexity of the algorithm. □

6 Acknowledgements

The authors thank Tomasz Kociumaka for the idea of the proof of Theorem 6.

References

1. A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, vol. F12 of NATO ASI Series, Springer-Verlag, 1985.
2. A. Apostolico and F. P. Preparata: *Optimal off-line detection of repetitions in a string.* Theor. Comput. Sci., 22 1983, pp. 297–315.
3. G.-H. Chen, J.-J. Hong, and H.-I. Lu: *An optimal algorithm for online square detection*, in CPM, A. Apostolico, M. Crochemore, and K. Park, eds., vol. 3537 of Lecture Notes in Computer Science, Springer, 2005, pp. 280–287.
4. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inf. Process. Lett., 12(5) 1981, pp. 244–250.
5. M. Crochemore: *Transducers and repetitions.* Theor. Comput. Sci., 45(1) 1986, pp. 63–86.
6. M. Crochemore, S. Z. Fazekas, C. S. Iliopoulos, and I. Jayasekera: *Bounds on powers in strings*, in Developments in Language Theory, M. Ito and M. Toyama, eds., vol. 5257 of Lecture Notes in Computer Science, Springer, 2008, pp. 206–215.
7. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, 2007.
8. M. Crochemore, L. Ilie, and W. Rytter: *Repetitions in strings: Algorithms and combinatorics.* Theor. Comput. Sci., 410(50) 2009, pp. 5227–5235.
9. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *On the maximal number of cubic runs in a string*, in LATA, A. H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 227–238.
10. M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen: *Efficient algorithms for two extensions of LPF table: The power of suffix arrays*, in SOFSEM, J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, eds., vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 296–307.
11. M. Crochemore and W. Rytter: *Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays.* Theor. Comput. Sci., 88(1) 1991, pp. 59–82.
12. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2003.
13. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* J. of Combinatorial Theory Series A, 82 1998, pp. 112–120.
14. D. Gusfield and J. Stoye: *Linear time algorithms for finding and representing all the tandem repeats in a string.* J. Comput. Syst. Sci., 69(4) 2004, pp. 525–546.
15. L. Ilie and L. Tinta: *Practical algorithms for the longest common extension problem*, in SPIRE, J. Karlgren, J. Tarhio, and H. Hyyrö, eds., vol. 5721 of Lecture Notes in Computer Science, Springer, 2009, pp. 302–309.
16. J. Jansson and Z. Peng: *Online and dynamic recognition of squarefree strings*, in MFCS, J. Jedrzejowicz and A. Szepietowski, eds., vol. 3618 of Lecture Notes in Computer Science, Springer, 2005, pp. 520–531.
17. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proceedings of the 40th Symposium on Foundations of Computer Science, 1999, pp. 596–604.
18. R. M. Kolpakov and G. Kucherov: *On maximal repetitions in words.* J. of Discr. Alg., 1 1999, pp. 159–186.
19. S. R. Kosaraju: *Computation of squares in a string (preliminary version)*, in CPM, M. Crochemore and D. Gusfield, eds., vol. 807 of Lecture Notes in Computer Science, Springer, 1994, pp. 146–150.
20. G. M. Landau and J. P. Schmidt: *An algorithm for approximate tandem repeats*, in CPM, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., vol. 684 of Lecture Notes in Computer Science, Springer, 1993, pp. 120–133.
21. H.-F. Leung, Z. Peng, and H.-F. Ting: *An efficient online algorithm for square detection*, in COCOON, K.-Y. Chwa and J. I. Munro, eds., vol. 3106 of Lecture Notes in Computer Science, Springer, 2004, pp. 432–439.

22. M. G. Main and R. J. Lorentz: *An O(n log n) algorithm for finding all repetitions in a string.* J. Algorithms, 5(3) 1984, pp. 422–432.
23. M. G. Main and R. J. Lorentz: *Linear time recognition of squarefree strings*, in Apostolico and Galil [1], pp. 271–278.
24. M. O. Rabin: *Discovering repetitions in strings*, in Apostolico and Galil [1], pp. 279–288.
25. A. O. Slisenko: *Detection of periodicities and string matching in real time.* J. Soviet Math., 22 1983, pp. 1316–1386.
26. J. Stoye and D. Gusfield: *Simple and flexible detection of contiguous repeats using a suffix tree.* Theor. Comput. Sci., 270(1-2) 2002, pp. 843–856.

Inferring Strings from Runs

Wataru Matsubara*[1], Akira Ishino[2], and Ayumi Shinohara[1]

[1] Graduate School of Information Science, Tohoku University, Japan
{matsubara@shino., ayumi@}ecei.tohoku.ac.jp
[2] Google Japan Inc.
ishino@google.com

Abstract. A run in a string is a nonextendable periodic substring in the string. Detecting all runs in a string is important and studied both from theoretical and practical points of view. In this paper, we consider the reverse problem of it. We reveal that the time complexity depends on the alphabet size k of the string to be output. We show that it is solvable in polynomial time for both binary alphabet and infinite alphabet, while it is NP-complete for finite $k \geq 4$. We also consider a variant of the problem where only a subset of runs are given as an input. We show that it is solvable in polynomial time for infinite alphabet, while it is NP-complete for finite $k \geq 3$.

Keywords: repetition, runs, inferring problem

1 Introduction

A reverse problem on strings is for a given data structure, inferring a string that does not conflict with the input information. A motivation of considering reverse problem is to characterize if-and-only-if conditions on the data structures. It is also of interest for design methods for the data structures.

Reverse problems on strings have been considered for various data structures. Franek et al. [8] initiated a linear time algorithm for testing whether an integer array is the Border Table of a string on unbounded size alphabet (infinite alphabet). Duval et al. [7] solves the same question for a bounded-size alphabet (finite alphabet). I et al. [10] considered for parametrized border array. Bannai et al. [1] solved three other data structures: Directed Acyclic Subsequence Graph, Directed Acyclic Word Graph, and Suffix Array. All of their three testing algorithms run in linear time. Clement et al. [3] showed a linear time algorithm for solving the reverse problem for Prefix Table. The reverse problem for the Longest Previous Factor Table is an open question.

Repetitions is one of most fundamental property of strings, it is important both theoretical and practical point of view. Detecting repetition in strings is an important element of several questions: pattern matching, text compression.

Kucherov and Kolpakov showed that considering maximal repetitions, or runs, the number of runs in any string of length n is $O(n)$. Although they were not able to give bounds for the constant factor, there have been several works to this end [13,14,12,4,2,9,5,15]. The known results in the topic and a deeper description of the motivation can be found in a survey by Crochemore et al. [6]. The currently known best upper bound[1] and lower bound are as follows:

$$0.944575 \leq \frac{\rho(n)}{n} \leq 1.029$$

* Supported in part by Grant-in-Aid for JSPS Fellows

[1] Presented on the website http://www.csd.uwo.ca/faculty/ilie/runs.html

Wataru Matsubara, Akira Ishino, Ayumi Shinohara: Inferring Strings from Runs, pp. 150–160 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

The upper bound obtained by calculations based on the proof technique of [4,5]. The technique bounds the number of runs for each string by considering runs in two parts: runs with long periods, and runs with short periods. The former is more sparse and easier to bound. The latter is bounded by an exhaustive calculation concerning how runs of different periods can overlap in an interval of some length. Considering all possibility, they show that a string of length n contains at most $0.93n$ runs with period up to 60.

We are inspired by their work, and we tackle to the reverse problem of detecting all runs in a string. That is, for a given set S of runs, we will find a string whose runs are consitent with S. We consider the following two situations.

1. Inferring strings whose runs are equal to S (the perfect input problem).
2. Inferring strings whose runs subsume S (the imperfect input problem).

We show that inferring strings over infinite alphabet is tractable in both settings. We show that it is also tractable for binary alphabet in the perfect input setting. On the other hand, the inferring string over ternary alphabet is intractable. For alphabet size k, we show that the perfect input problem is NP-hard for $k \geq 4$ and the inperfect input problem is NP-hard for $k \geq 3$.

2 Preliminary

2.1 Notations on Strings

Let $\mathcal{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers. Let Σ be a set of symbols. An element of Σ^* is called a *string*. Σ^n denotes the set of strings of length n. The length of a string w is denoted by $|w|$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. A string w has period p if $w[i] = w[i + p]$ for $1 \leq i \leq |w| - p$. A string w is called *primitive* if w cannot be written as u^k, where k is a positive integer, $k \geq 2$.

Definition 1. *A* run *(also called a maximal repetition) of period p in a string w is a substring $w[i : j]$ such that:*

(1) $w[i : j]$ *has period p and satisfies $j - i + 1 \geq 2p$,*
(2) $w[i - 1] \neq w[i + p - 1]$ *(if $w[i - 1]$ is defined),*
$w[j + 1] \neq w[j - p + 1]$ *(if $w[j + 1]$ is defined), and*
(3) $w[i : i + p - 1]$ *is primitive.*

We denote the run $u = w[i : j]$ of period p in w by a triple $\langle i, j, p\rangle \in \mathcal{N}^3$ consisting of the begin position i, the end position j and the minimul period p of u. For a string w, we define that $\mathit{Runs}(w) = \{\langle i, j, p\rangle \mid w[i : j]$ is a run of period p in $w\}$. For instance, $\mathit{Runs}(\texttt{ababcbcca}) = \{\langle 1, 4, 2\rangle, \langle 4, 7, 2\rangle, \langle 7, 8, 1\rangle\}$.

Theorem 2 ([11]). *Given a string w of length n, the set of all runs in string w can be calculated in $O(n)$ time.*

3 Easiness Results

At first, we give the definition of the problem.

Problem 3. Inferring strings of alphabet size k from runs (k-InvRunsEq).
Input: $S \subseteq \mathcal{N}^3$ and $n \in \mathcal{N}$.
Output: A string $w \in \Sigma_k^n$ such that $Runs(w) = S$ if any, and *None* otherwise.

Problem 4. Inferring consistent strings from runs (k-InvRunsSubset).
Input: $S \subset \mathcal{N}^3$ and $n \in \mathcal{N}$.
Output: A string $w \in \Sigma_k^n$ such that $Runs(w) \supseteq S$ if any, and *None* otherwise.

For convenience, ∞-InvRunsEq and ∞-InvRunsSubset denotes the problem for infinite alphabet. In this section, we show k-InvRunsEq can be solved in polynomial time if either $k \leq 2$ or $k = \infty$.

Theorem 5. *2-*InvRunsEq *is solvable in linear time.*

Proof. First we give a simple observation on the relationship between consecutive two symbols $w[k], w[k+1]$ of string w and runs of period 1 in $Runs(w)$. If $w[k] = w[k+1]$ for some k, then the interval $(k, k+1)$ must be included in some run $\langle i, j, 1\rangle \in Runs(w)$ such that $i \leq k$ and $k+1 \leq j$. The converse is also holds. As a result, we can determine whether $w[k] = w[k+1]$ or not for every $1 \leq k \leq n-1$, by simply checking the intervals in all runs of period 1 in $Runs(w)$.

For binary alphabet $\Sigma_2 = \{\mathtt{a}, \mathtt{b}\}$, after choosing the first symbol $w[1]$ either `a` or `b` arbitrarily, we can uniquely determine the next symbol one by one consecutively, depending on whether $w[k] = w[k+1]$ or not, for each $k = 1, 2, \ldots, n-1$. It can be done in $O(n)$ time. The resulting string w is the only possible candidate for the solution of a given set S of runs, up to isomorphism. We can verify that $Runs(w) = S$ holds or not, in $O(n)$ time. If it holds, return w as a solution; otherwise, return *None*. □

Let $G = (V, E)$ be an unordered graph, where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. A proper graph coloring on G is an assignment of colors to its nodes such that no two adjacent nodes receive the same color.

Problem 6. [k-Color problem]
Input: Graph G
Decide: The nodes of G can be colored with k colors such that no two nodes jointed by an edge have the same color.

We can regard the problem as identifying an equivalence relation over n elements $V = \{1, 2, \ldots, n\}$, so that it is consistent with the structural information of runs in S. Some constraints of equivalence and inequivalence are easily extracted from each run in S. For example, if $\langle 4, 7, 2\rangle$ is a run of string w of length 9, we know that $w[4] = w[6]$ and $w[5] = w[7]$ from condition (1) in Definition 1, as well as $w[3] \neq w[5]$ and $w[6] \neq w[8]$ from condition (2). Based on these observations, we will reduce the problem to the graph-coloring problem of a graph (V_S, E_S), where V_S is an equivalence class of V and E_S represents the inequivalence relations, as we will show the details below.

We define a binary relation R over V by

$$R = \{(k, k+p) \mid i \leq k \leq j-p \quad \text{for } \langle i, j, p\rangle \in S\},$$

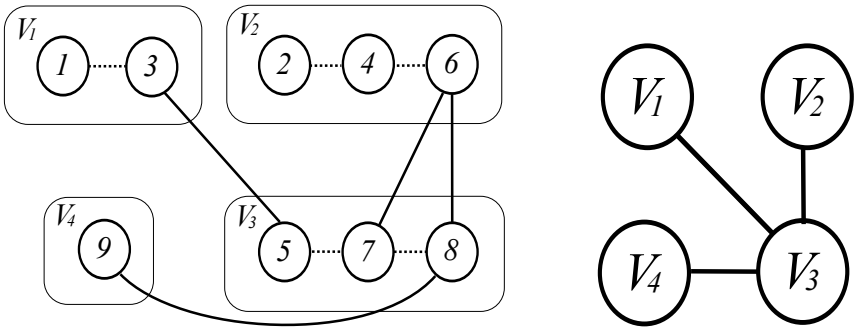


Figure 1. Graph G_S which represents $S = \{\langle 1, 4, 2\rangle, \langle 4, 7, 2\rangle, \langle 7, 8, 1\rangle\}$.

which stands for the equivalence $w[k] = w[k + p]$ due to the condition (1) in Definition 1. Let $R^{\equiv}$ be the reflexive transitive symmetric closure of R. That is, $R^{\equiv}$ is the smallest equivalence relation over V containing R. We define $V_S = \{[v]_{R^{\equiv}} \mid v \in V\}$, where $[v]_{R^{\equiv}}$ denotes the equivalence class of v in V with respect to $R^{\equiv}$.

We also define a binary relation D by

$$D = \{(i-1, i-1+p) \mid 1 < i,\ \langle i, j, p\rangle \in S\} \cup \{(j+1, j+1-p) \mid j < n,\ \langle i, j, p\rangle \in S\}.$$

D represents the inequalities $w[i-1] \neq w[i-1+p]$ and $w[j+1] \neq w[j+1-p]$ that come from condition (2) in Definition 1. We now define the set $E_S \subseteq V_S \times V_S$ of edges by

$$E_S = \{([v_1]_{R^{\equiv}}, [v_2]_{R^{\equiv}}) \mid (v_1, v_2) \in D\}.$$

Using a graph (V_S, E_S), we give a following theorem.

Theorem 7. ∞-InvRunsEq *and* ∞-InvRunsSubset *are solvable in* $O(n^2)$ *time.*

Proof. Since the equivalence relation $R^{\equiv}$ is disjoint with D, if G_S contains a self loop, then there exists no string w that satisfies $\mathit{Runs}(w) \supseteq S$. Otherwise, let ψ be a coloring of G_S. Since the number of colors is unbounded, it is straightforward to get such ψ; we may associate a different color to each node in V_S. By using ψ, we construct a string $w = \psi([1])\psi([2])\ldots\psi([n])$, where we abbreviated $[i]_{R^{\equiv}}$ by $[i]$. It is not hard to verify that $\mathit{Runs}(w)$ satisfy the problem condition. We now consider the time complexity. The graph G_S can be constructed in $O(n^2)$ time, and we can check whether G_S contains a self loop or not in linear time. Therefore, ∞-InvRunsEq and ∞-InvRunsSubset are solvable in $O(n^2)$ time. □

Example 8. Let us consider an instance $S = \{\langle 1, 4, 2\rangle, \langle 4, 7, 2\rangle, \langle 7, 8, 1\rangle\}$ and 9 for ∞-InvRunsEq. We will find a string w of length 9 over infinite alphabet satisfying $\mathit{Runs}(w) = S$. We construct the graph $G_S = (V_S, E_S)$ from S as follows: Since $R = \{(1, 3), (2, 4), (4, 6), (5, 7), (7, 8)\}$, we have a set of nodes $V_S = \{V_1, V_2, V_3, V_4\}$ with $V_1 = \{1, 3\}$, $V_2 = \{2, 4, 6\}$, $V_3 = \{5, 7, 8\}$, $V_4 = \{9\}$. Moreover, since $D = \{(3, 5), (6, 7), (6, 8), (8, 9)\}$, we have a set of edges $E_S = \{(V_1, V_3), (V_2, V_3), (V_3, V_4)\}$. (Figure 1 shows G_S.) Since G_S contains no self loop, it is always colorable if the number of colors is unlimited. By considering a trivial coloring function ϕ such that $\phi(V_1) = \mathtt{a}$, $\phi(V_2) = \mathtt{b}$, $\phi(V_3) = \mathtt{c}$, and $\phi(V_4) = \mathtt{d}$, we get the string $w = \psi([1])\psi([2])\ldots\psi([9]) = \mathtt{ababcbccd}$, and we can verify that $\mathit{Runs}(w) = S$, indeed.

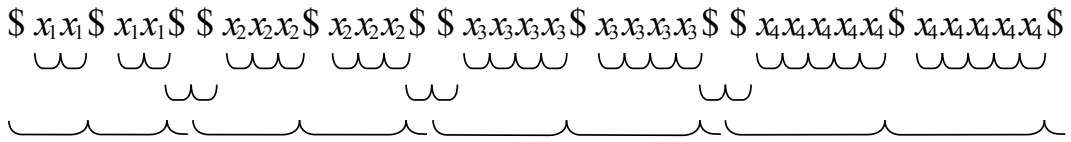


Figure 2. The string v representing the node set $V = \{1, 2, 3, 4\}$. Bows indicate all runs in v.

Remark that in Example 8, the graph G_S actually can be colored by 3 colors as $\phi(V_1) = \phi(V_4) = \mathtt{a}$, $\phi(V_2) = \mathtt{b}$, and $\phi(V_3) = \mathtt{c}$, so that we have another solution string $w = \mathtt{ababcbcca}$ over three symbols. In this way, k-InvRunsEq problem is reduced to k-Color problem. However, unfortunately, k-Color is NP-complete for finite $k \geq 3$ so that it is intractable. In the next section, we show an opposite reduction.

4 Hardness result for perfect input

In this section we show the NP-completeness of k-InvRunsEq for any fixed $k \geq 4$.

4.1 Instance transformation

Let $G = (V, E)$ be an input of the 3-Color problem, where $V = \{1, 2, \ldots, m\}$. We will construct an instance $\langle S, n\rangle$ for 4-InvRunsEq.

At first, we construct a string g over $\Delta = \{x_1, x_2, \ldots, x_m, \$\}$ which represents G as follows. For nodes V, let v be the string

$$v = v_1 v_2 \ldots v_k \ldots v_m,$$

where each substring v_k corresponding to node k is defined by

$$v_k = \$x_k^{(k+1)}\$x_k^{(k+1)}\$.$$

For example, Figure 2 shows the string v for the case $m = 4$.

Next we give the transformation which represents the edges E. For each $1 \leq k \leq n$, we define ℓ_k and r_k by

$$\ell_k = v_1 v_2 \ldots v_{k-1}\$x_k^{k+1},$$
$$r_k = x_k^{k+1}\$v_{k+1} \ldots v_m.$$

You see that ℓ_k is a prefix of v, and r_k is a suffix of v, so that $\ell_k\$r_k = v$. String e_{ij} which represents the edge $(i, j) \in E$ is defined as $e_{ij} = \ell_i r_j$.

Using the above gadgets, we give the function *enc* which encodes the graph G to the string:

$$g = B_1 B_2 \ldots B_{|E|+1},$$

where blocks B_t's are defined by $B_1 = vv\ell_{i_1}$, $B_{|E|+1} = r_{j_{|E|}}vv$, and for $t = 2 \ldots |E|$, $B_t = r_{j_{t-1}}vv\ell_{i_t}$ if t is odd $r_{j_{t-1}}vvv\ell_{i_t}$ otherwise. For example, Figure 3 shows the string g for the case $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$.

Critical points of the construction B_t are the following (see Figure 4):

(B1) Each B_t has period $|v|$ since ℓ_k (r_k, resp.) is a prefix (suffix, resp.) of v.

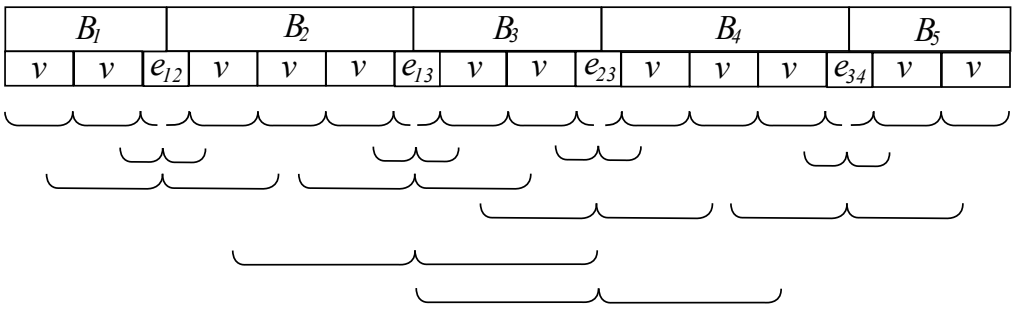


Figure 3. The string g representing the graph $G = (V = \{1, 2, 3, 4\}, E = \{(1, 2), (1, 3), (2, 3), (3, 4)\})$. Bows indicate some of runs in g.

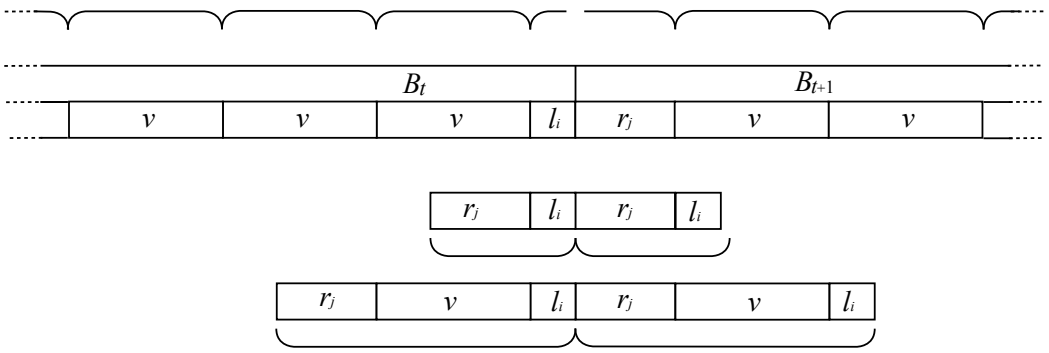


Figure 4. The substring of g represents the edge $E_t = (i, j)$. The top bows indicate that both B_t and B_{t+1} have period $|v|$, although these two periodicities are disconnected.

(B2) On the border between B_t and B_{t+1} there exists a run of period $|r_j v \ell_i|$ (the bottom bow in Figure 4), as well as a run of period $|r_j \ell_i|$ (the middle bow in Figure 4).

Therefore, string g is a concatenation of e_{i_k,j_k}'s and v's. Using this string, we can add some appropriate restrictions to the substitution φ depending on the input graph G. Finally, we calculate the set of runs and the length of string g, and output the instance $\langle \mathit{Runs}(g), |g| \rangle$ of the inverse runs problem.

4.2 Correctness of the reduction

For a given graph G, let $\langle S, n \rangle$ be the instance generated by the above reduction. By reconstructing a graph G_S from S and n in the same way in Section 3, we will show the relation between a coloring function for G and a string w of length n satisfying $\mathit{Runs}(w) = S$.

Let $V' = \{1, 2, \ldots, n\}$ be the set of positions of w. Because of the conditions (B1) and (B2), for any position $i \in V'$, there exists position $j \in \{1, \ldots, |v|\}$ such that $[i]_{R^{\equiv}} = [j]_{R^{\equiv}}$. We consider V_S as the quotient set of V by the equivalence relation $R^{\equiv}$, that is represented using the base string g as $V_S = \{V_{x_1}, V_{x_2}, \ldots, V_{x_{|V|}}, V_{\$}\}$ where $V_c = \{i \mid g[i] = c\}$ for $c \in \Delta$.

Next we consider the edges $E_S \subset V_S \times V_S$ yielded by the binary relation D representing inequivalences extracted from S. We show $E_S = \{(V_{x_i}, V_{x_j}) \mid (i,j) \in E\} \cup \{(V_{x_t}, V_\$), \mid 1 \le t \le |V|\}$ by enumerating the all runs in the string g. That is, we show the substring $g[i-1 : j+1]$ for each run $\langle i, j, p\rangle \in Runs(g)$:

- period 1
  Since $\$x_k^{k+1}\$$ is a substring of g, we have $(V_\$, V_{x_k}) \in E_S$ for each $k = 1 \dots m$.
- period 1
  Since $\$x_i^{i+1}x_j$ and $x_i x_j^{j+1}\$$ are substrings of g, we have
  $$\{(V_{x_j}, V_\$), (V_{x_i}, V_\$), (V_{x_i}, V_{x_j})\} \subset E_S$$
  for each $(i,j) \in E$.
- period $(k+2)$
  Since $\$\$x_k^{k+1}\$x_k^{k+1}\$\$$ is a substring of g, we have $(V_{x_k}, V_\$) \in E_S$ for each $k = 1 \dots m$.
- period $|r_j v^t \ell_j|$ for $t = 0, 1$
  Since $\$r_j v^t \ell_i r_j v^t \ell_i\$$ is a substring of g, we have
  $$\{(V_{x_j}, V_\$), (V_{x_i}, V_\$)\} \subset E_S$$
  for each $(i,j) \in E$,
- period $|r_j v v \ell_j|$
  Since $x_j(r_j v v \ell_i)^2\$$ and $\$(r_j v v \ell_i)^2 x_i$ are substrings of g, we have
  $$\{(V_{x_j}, V_\$), (V_{x_i}, V_\$), (V_{x_i}, V_{x_j})\} \subset E_S$$
  for each $(i,j) \in E$,

  For this graph G_S, we have the following lemma.

Lemma 9. *The following three propositions are equivalent for any integer $k \ge 1$:*

(1) G is k-Colorable.
(2) G_S is $(k+1)$-Colorable.
(3) A string $w \in \Sigma_{k+1}^n$ exists such that $Runs(w) = S$.

Proof. (1) $\Leftrightarrow$ (2) From the definition of G_S, the induced subgraph $G' = (V_S - \{V_\$\}, E'_S)$ of G_S is isomorphic to G. Moreover $V_\$$ is connected with all the other nodes in V_S. Therefore $(k+1)$-coloring for G_S is a k-coloring for G'. It means (2) $\Rightarrow$ (1). On the other hand, by assigning a new color to the node $V_\$$, we obtain the $(k+1)$-coloring for G_S from k-coloring for G. It means (1) $\Rightarrow$ (2). See Figure 5.

(2) $\Rightarrow$ (3) Assume that $\psi : V_S \to \{1, \dots, k\}$ is a k coloring function of G. We can construct the substitution $\varphi : \{x_1, \dots, x_n, \$\} \to \{\mathtt{a}_1, \dots, \mathtt{a}_k, \$\}$ as $\varphi(c) = \mathtt{a}_{\psi(V_c)}$ for $c \in \Delta$. Since it satisfies $Runs(\varphi(g)) = S$, there exists a string $w = \varphi(g)$ such that $Runs(w) = S$.

(3) $\Rightarrow$ (2) Assume that string w satisfies $Runs(w) = S$. There exists a substitution $\varphi : \{x_1, \dots, x_n, \$\} \to \{\mathtt{a}_1, \dots, \mathtt{a}_k, \$\}$ and it holds $w = \varphi(g)$. Then, we can construct a coloring function ψ of G as follows; for each $c \in \{x_1, \dots, x_n\}$, $\psi(V_c) = i$ such that $\varphi(c) = \mathtt{a}_i$. Therefore G is k-colorable. □

Theorem 10. *4-*InvRunsEq *is NP-complete.*

Proof. From the above section, for any fixed $k \ge 1$, k-Color is polynomial time reducible to $(k+1)$-InvRunsEq. Since 3-Color is NP-Complete, 4-InvRunsEq is also NP-complete. □

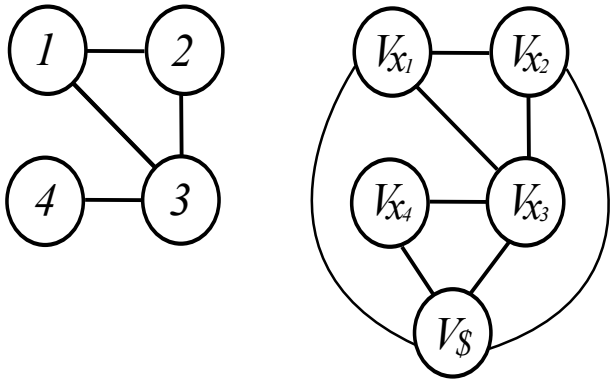


Figure 5. Input graph G and reconstructed graph G_S

5 Hardness result for Imperfect input

In this section we consider a variant of k-InvRunsEq. We consider the problem to infer a string from a set of runs that are given *imperfectly*.

We show the NP-completeness of 3-InvRunsSubset.

5.1 Instance reduction

Let $G = (V, E)$ be an input of the k-Color problem, where $V = \{1, 2, \ldots, m\}$. we construct the instance for the k-InvRunsSubset problem.

At first, we construct a string g over $\Delta' = \{x_1, x_2, \ldots x_m, \$_1, \$_2, \ldots, \$_m\}$ which represents G as follows: for nodes V, let v be the string,

$$v = x_1 x_1 \$_1 x_2 x_2 \$_2 \ldots x_m x_m \$_m.$$

The different point from k-InvRunsEq is that we cannot use a single symbol \$ as a separator. Instead, we use m separator symbols $\$_1, \$_2, \ldots, \$_m$, and we construct a string on Δ' and substitution $\varphi : \Delta' \to \{\mathtt{a}_1, \ldots, \mathtt{a}_k\}$.

For all $k = 2 \ldots m$, it satisfies $\varphi(x_{k-1}) \neq \varphi(\$_k) \neq \varphi(x_{k+1})$ and $\varphi(x_m) \neq \varphi(\$_m) \neq \varphi(x_1)$.

Since the alphabet size is three or more, for any substitution on $\{x_1, x_2, \ldots, x_m\}$, we can choose a substitution on $\{\$_1, \ldots, \$_m\}$. We can use the variables $\{\$_1, \ldots, \$_m\}$ as a separator.

Next we give the transformation which represents the edge of graph. For each $k = 1 \ldots m$, we define ℓ_k and r_k as follows:

$$\ell_k = x_1 x_1 \$_1 x_2 x_2 \$_2 \ldots x_{k-1} x_{k-1} \$_{k-1} x_k,$$
$$r_k = x_k \$_k x_{k+1} x_{k+1} \$_{k+1} \ldots x_m x_m \$_m.$$

You can see that ℓ_k is a prefix of v and r_k is a suffix v. The string e_{ij} represents the edge $(i, j) \in E$, where $e_{ij} = \ell_i r_j$.

Using the above gadgets, we give the definition of string which represents graph G by

$$g = B_1 B_2 \ldots B_{|E|+1},$$

where blocks B_t's are defined by $B_1 = vv\ell_{i_1}, B_{|E|+1} = r_{j_{|E|}} vv$, and for $t = 2 \ldots |E|$, $B_t = r_{j_{t-1}} vv\ell_{i_t}$.

By picking up runs from $Runs(g)$ we construct S as follows:

$$\begin{aligned} S = &\{\langle 3t-2, 3t-1, 1\rangle \mid 1 \le t \le m\} \\ &\cup \{\langle pos_t + 1, pos_{t+1}, |v|\rangle \mid 0 \le t \le |E|\} \\ &\cup \{\langle pos_t - p_t + 1, pos_t + p_t, p_t\rangle \mid 1 \le t \le |E|\}, \end{aligned}$$

where $pos_t = |B_1 B_2 \dots B_t|$ and $p_t = |r_{i_t} v \ell_{j_t}|$. Note that $S \subset Runs(g)$.

We output $\langle S, |g|\rangle$ as the instance of k-InvRunsSubset.

5.2 Correctness of the reduction

For a given a graph G, let $\langle S, n\rangle$ be the instance generated by the above reduction. By reconstructing a graph G_S from S and n in the same way in Section 3 and Section 4, we will show the relation between a coloring function for G and a string w of length n satisfying $Runs(w) \supseteq S$.

Let $V' = \{1, 2, \dots n\}$ be a set of positions. At first we construct V_S from using the equivalence relation R of S. Similar to the case of k-InvRunsEq, because the conditions (B1) and (B2) in Section 4, for any position $i \in V'$, there exists position $j \in \{1, \dots, |v|\}$ such that $[i]_{R^{\equiv}} = [j]_{R^{\equiv}}$. We consider V_S as the quotient set of V by the equivalence relation $R^{\equiv}$, that is represented using the base string g as $V_S = \{V_{x_1}, \dots, V_{x_m}, V_{\$_1}, \dots, V_{\$_m}\}$ where $V_c = \{i \mid g[i] = c\}$ for $c \in \Delta'$.

Next we consider the edges $E_S \subset V_S \times V_S$ yielded by the binary relation D representing inequivalences extracted from S. We show $E_S = \{(V_{x_i}, V_{x_j} \mid (i,j) \in E\} \cup \{(V_{x_t}, V_{\$_{t-1}}), (V_{x_t}, V_{\$_t}) \mid 1 \le t \le |E|\}$:

- Since $\langle 3t-2, 3t-1, 1\rangle \in S$, and the fact that $g[3t-3] = \$_{t-1}, g[3t-2] = x_t$ and $g[3t-1] = x_t, g[3t] = \$_t$, we have $(V_{\$_{t-1}}, V_{x_t}), (V_{\$_t}, V_{x_t}) \in E_S$ for $t = 1 \dots m$.
- Since $\langle pos_t + 1, pos_{t+1}, |v|\rangle \in S$, and the fact that $g[pos_t] = x_{j_t}, g[pos_t + |v|] = x_{i_t}$ and $g[pos_t + 1] = x_{i_t}, g[pos_t + 1 - |v|] = x_{j_t}$, we have $(V_{x_{i_t}}, V_{x_{j_t}}) \in E_S$ for $t = 1 \dots |E|$.
- Since $\langle pos_t - p_t + 1, pos_t + p_t, p_t\rangle \in S$ and the fact that $g[pos_t - p_t] = x_{j_t}, g[pos_t] = x_{i_t}$ and $g[pos_t + 1] = x_{j_t}$ and $g[pos_t + p_t + 1] = x_{i_t}$, we have $(V_{x_{i_t}}, V_{x_{j_t}}) \in E_S$ for $t = 1 \dots |E|$.

For this graph G_S, we have the following lemma.

Lemma 11. *The following three observation are equivalent for any fixed $k \ge 3$:*

(1) G is k-Colorable.
(2) G_S is k-Colorable.
(3) A string $w \in \Sigma_k^n$ exists such that $Runs(w) \supseteq S$.

Proof. (1) $\Leftrightarrow$ (2) From the definition of G_S, the induced subgraph $G' = (V'_S, E'_S)$ of G_S is isomorphic to G, where $V'_S = V_S - \{V_{\$_1}, \dots, V_{\$_m}\}$. Therefore k-coloring for G_S is a k-coloring for G'. It means (2) $\Rightarrow$ (1). On the other hand, Since each nodes $\{V_{\$_1}, \dots, V_{\$_m}\}$ is connected to two nodes in G_S, we can assign an another color for any fixed $k \ge 3$

Therefore we obtain the k-coloring for V_S from k-coloring for G, where $k \ge 3$. It means (1) $\Rightarrow$ (2). See Figure 6.

(2) $\Rightarrow$ (3) Assume that $\psi : V_S \to \{1 \dots k\}$ is k coloring function of G. We can construct the substitution $\varphi : \Delta' \to \{\mathtt{a}_1, \dots, \mathtt{a}_k\}$ as $\varphi(c) = \mathtt{a}_{\psi(V_c)}$ for $c \in \Delta'$. Since it satisfies $Runs(\varphi(g)) \supseteq S$, there exists the string $w = \varphi(g)$ such that $Runs(w) \supseteq S$.

(3) ⇒ (2) Assume that string w satisfies $Runs(w) \supseteq S$. there exists some substitution $\varphi : \Delta' \to \{\mathtt{a}_1, \ldots, \mathtt{a}_k\}$ and it holds $Runs(w) \supseteq Runs(\varphi(g))$. And then, we can construct the coloring function ψ of G as follows: For each $c \in \{x_1, \ldots, x_n\}$, $\psi(V_c) = i$ such that $\varphi(c) = \mathtt{a}_i$. Therefore G is k-colorable.

□

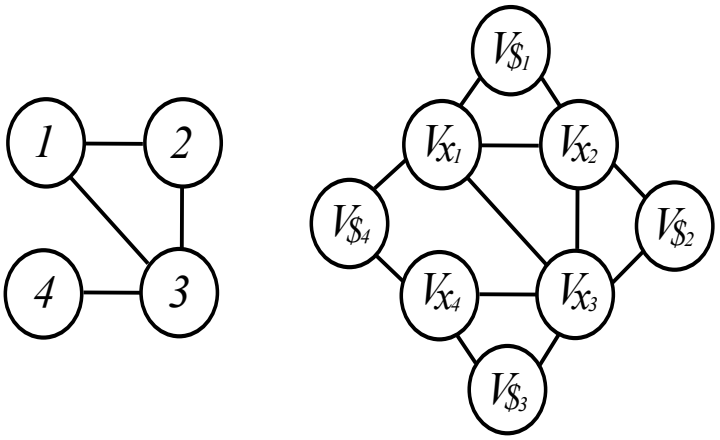


Figure 6. Input graph G and reconstructed graph G_S

Theorem 12. *3-*InvRunsSubset *is NP-complete.*

Proof. From above section, for any $k \geq 3$, k-Color is polynomial time reducible to k-InvRunsSubset. Since 3-Color is NP-Complete, 3-InvRunsSubset is also NP-complete. □

6 Conclusion

In this paper, we considered reverse problems of detecting all runs in a string. We showed that the computational complexity depends on the alphabet size of the output string. we also consider a variant of the problem, where the information on runs is incomplete. The result is summarized as the following table.

alphabet size k	k-InvRunsEq	k-InvRunsSubset
2	$O(n)$	open
3	open	NP-Complete
≥ 4	NP-Complete	NP-Complete
∞	$O(n^2)$	$O(n^2)$

It would be interesting to find out whether 3-InvRunsEq and 2-InvRunsSubset are NP-complete or in P. For 3-InvRunsEq, it is needed to improve the reduction from 3-Color without using the separator symbol "$". On the other hand, for 2-InvRunsSubset, it seems that we should develop a reduction from another NP-Complete problem.

References

1. H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda: *Inferring strings from graphs ans arrays*, in Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003), vol. 2747 of LNCS, Springer, 2003, pp. 208–217.
2. P. Baturo, M. Piatkowski, and W. Rytter: *The number of runs in Sturmian words*, in Proc. 13th International Conference on Implementation and Application of Automata (CIAA 2008), vol. 5148 of LNCS, Springer, 2008, pp. 252–261.
3. J. Clément, M. Crochemore, and G. Rindone: *Reverse engineering prefix tables*, in 26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 289–300.
4. M. Crochemore and L. Ilie: *Maximal repetitions in strings*. J. Comput. Syst. Sci., 74 2008, pp. 796–807.
5. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture*, in Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008), vol. 5029 of LNCS, Springer, 2008, pp. 290–302.
6. M. Crochemore, W. Rytter, and L. Ilie: *Repetitions in strings: Algorithms and combinatorics*. Theoretical Computer Science, 410(50) 2009, pp. 5227–5235.
7. J. Duval, T. Lecroq, and A. Lefevre: *Border array on bounded alphabet*, in Proc. Prague Stringology Conference 2002, 2002, pp. 28–35.
8. F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang: *Verifying a border array in linear time*. J. Comb. Math. Comb. Comput, 42 2000, pp. 223–236.
9. M. Giraud: *Not so many runs in strings*, in Proc. 2nd International Conference on Language and Automata Theory and Applications (LATA 2008), vol. 5196 of LNCS, Springer, 2008, pp. 245–252.
10. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Counting parameterized border arrays for a binary alphabet*, in Proc. 3nd International Conference on Language and Automata Theory and Applications (LATA 2009), vol. 5457 of LNCS, Springer, 2009, pp. 422–433.
11. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS 1999), 1999, pp. 596–604.
12. S. J. Puglisi, J. Simpson, and W. F. Smyth: *How many runs can a string contain?* Theoretical Computer Science, 401(1–3) 2008, pp. 165–171.
13. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006), vol. 3884 of LNCS, Springer, 2006, pp. 184–195.
14. W. Rytter: *The number of runs in a string*. Inf. Comput., 205(9) 2007, pp. 1459–1469.
15. J. Simpson: *Modified padovan words and the maximum number of runs in a word*. The Australasian Journal of Combinatorics, 42 2010, pp. 129–145.

Bounded Number of Squares in Infinite Repetition-Constrained Binary Words

Golnaz Badkobeh[1] and Maxime Crochemore[1,2]

[1] King's College London, London, UK
[2] Université Paris-Est, France
Maxime.Crochemore@kcl.ac.uk

Abstract. A square is the concatenation of a nonempty word with itself. A word has period p if its letters at distance p match. The exponent of a nonempty word is the quotient of its length over its smallest period.
In this article we give a sketch of the new proof of the fact that there exists an infinite binary word which contains finitely many squares and simultaneously avoids words of exponent larger than 7/3.
Our infinite word contains 12 squares, which is the smallest possible number of squares to get the property, and 2 factors of exponent 7/3. These are the only factors of exponent larger than 2.
Keywords: combinatorics on words, repetitions, word morphisms.

1 Introduction

Repetitions in words is a basic question in Theoretical Informatics, certainly because it is related to many applications although it is first been studied by Thue at the beginning of the twentieth century [10] with a pure theoretical objective. Related results apply to the design of efficient string pattern matching algorithm, to text compression methods and entropy analysis, as well as to the study of repetitions in biological molecular sequences among others.

The knowledge of the strongest constraints an infinite word can tolerate helps the design and analysis of efficient algorithms. The optimal bound on the maximal exponent of factors of the word has been studied by Thue and many other authors after him. One of the first findings is that an infinite binary word can avoid factors with an exponent larger than 2, called 2^+-powers. This has been extended by Dejean [2] to the ternary alphabet and her famous conjecture on the repetitive threshold for larger alphabets has eventually been proved recently after a series of partial results by different authors (see [8] and references therein).

Another constraint is considered by Fraenkel and Simpson [3]: their parameter to the complexity of binary infinite words is the number of squares occurring in them without any restriction on the number of occurrences. It is fairly straightforward to check that no infinite binary word can contain less than three squares and they proved that some of them contain exactly three. Indeed all factors of exponent at least 2 occurring in their word should be considered, which adds 2 cubes. Their proof uses a pair of morphisms, one morphism to get an infinite string by iteration, the other morphism to produce the final translation on the binary alphabet. Their result has been proved with different pairs of morphism by Rampersad et al. [7] (the first morphism is uniform), by Harju and Nowotka [4] (the second morphism accepts any infinite square-free word), and by Badkobeh et al. [1] (the simplest morphisms).

In this article we show that we can combine the two types of constraints for the binary alphabet: producing an infinite word whose maximal exponent of its factor

Golnaz Badkobeh, Maxime Crochemore: Bounded Number of Squares in Infinite Repetition-Constrained Binary Words, pp. 161–166 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

is the smallest possible while containing the smallest number squares. The maximal exponent is 7/3 and the number of squares is 12 to which can be added two words of exponent 7/3.

It is known from Karhumäki and Shallit [5] that if an infinite binary avoids 7/3-powers it contains an infinite number of squares. Proving that it contains more than 12 squares is indeed a matter of simple computation.

Shallit [9] has built an infinite binary word avoiding $7/3^+$-powers and all squares of period at least 7. His word contains more than 18 squares.

Our infinite binary word avoids the same powers but contains only 12 squares, the largest having period 8. As before the proof relies on a pair of morphisms satisfying suitable properties. Both morphisms are almost uniform (up to one unit). The first morphism is weakly square-free on a 6-letter alphabet, and the second does not even corresponds to a uniquely-decipherable code but admits a unique decoding on the words produced by the first.

2 Repetitions in binary words

A word is a sequence of letters drawn from a finite alphabet. We consider the binary alphabet $B = \{\mathtt{0}, \mathtt{1}\}$, the ternary alphabet $A_3 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and the 6-letter alphabet $A_6 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$.

A square is a word of the form uu where u is a nonempty (finite) word. A word has period p if its letters at distance p are equal. The exponent of a nonempty word is the quotient of its length over its smallest period. Thus, a square is any word with an even integer exponent.

In this article we consider infinite binary words in which a small number of squares occur.

The maximal length of a binary word containing less than three square is finite. Indeed, it is 3 if it contains no square (e.g. 010), 7 if it contains 1 square (e.g. 0001000), and it is 18, e.g. 010011000111001101 contains only 00 and 11. But, as recalled above, this length is infinite if 3 squares are allowed to appear in the word. A simple proof of it relies on two morphisms f and h_0 defined as follows. The morphism f is defined from A_3^* to itself by

$$\begin{cases} f(\mathtt{a}) = \mathtt{abc}, \\ f(\mathtt{b}) = \mathtt{ac}, \\ f(\mathtt{c}) = \mathtt{b}. \end{cases}$$

It is known that the infinite word $\mathbf{f} = f(\mathtt{a})^\infty$ is square-free (see [6, Chapter 2]). It can additionally be checked that all square-free words of length 3 occur in $\mathbf{f}$ except aba and cbc. The morphism h_0 is from A_3^* to B^* and defined by

$$\begin{cases} h(\mathtt{a}) = \mathtt{01001110001101}, \\ h(\mathtt{b}) = \mathtt{0011}, \\ h(\mathtt{c}) = \mathtt{000111}. \end{cases}$$

This morphism is not uniform but the three codewords form a uniquely-decipherable code. Then the above result is a consequence of the next statement.

Theorem 1 ([1]). *The infinite word* $\mathbf{h_0} = h_0(f(\mathtt{a})^\infty)$ *contains the* 3 *squares* 00, 11 *and* 1010 *only. The cubes* 000 *and* 111 *are the only factors occurring in* $\mathbf{h}$ *and of exponent larger than* 2.

It is impossible to avoid 2^+-powers and keep a bounded number of squares. As proved by Karhumäki and Shallit [5], the exponent has to go up to 7/3 to allow the property.

In the two following sections we define two morphisms and derive their properties used to prove the next statement.

Theorem 2. *There exist an infinite binary word whose factors have an exponent at most* 7/3 *and that contains* 12 *squares, the fewest possible.*

Our infinite binary word contain the 12 squares $\mathtt{0}^2$, $\mathtt{1}^2$, $(\mathtt{01})^2$, $(\mathtt{10})^2$, $(\mathtt{001})^2$, $(\mathtt{010})^2$, $(\mathtt{011})^2$, $(\mathtt{100})^2$, $(\mathtt{101})^2$, $(\mathtt{110})^2$, $(\mathtt{01101001})^2$, $(\mathtt{10010110})^2$, and the two words 0110110 and 1001001 of exponent 7/3.

3 A weakly square-free morphism on six letters

In this section we consider a specific morphism used for the proof of Theorem 2. It is called g and defined from $A_6^* = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}^*$ to itself by:

$$\begin{cases} g(\mathtt{a}) = \mathtt{abac}, \\ g(\mathtt{b}) = \mathtt{babd}, \\ g(\mathtt{c}) = \mathtt{eabdf}, \\ g(\mathtt{d}) = \mathtt{fbace}, \\ g(\mathtt{e}) = \mathtt{bace}, \\ g(\mathtt{f}) = \mathtt{abdf}. \end{cases}$$

It can be shown that the morphism is weakly square-free in the sense that $\mathbf{g} = g^\infty(\mathtt{a})$ is an infinite square-free word, that is, all its finite factors have an exponent smaller than 2. Note that however it is not square-free since for example $g(\mathtt{cf}) = \mathtt{eabdfabdf}$ contains the square $(\mathtt{abdf})^2$. Moreover there is no known characterisation of weakly square-free morphisms defined on more than three letters (unless of course if only three letters occur in the infinite word).

The set of codewords $g(a)$'s ($a \in A_6$) is a prefix code and therefore a uniquely-decipherable code. Note also that any occurrence of abac in $g(w)$, for $w \in A_6^*$, uniquely corresponds to an occurrence of a in w.

Lemma 3. *The set of doublets occurring in* $\mathbf{g}$ *is*

$$D = \{\mathtt{ab}, \mathtt{ac}, \mathtt{ba}, \mathtt{bd}, \mathtt{cb}, \mathtt{ce}, \mathtt{da}, \mathtt{df}, \mathtt{ea}, \mathtt{fb}\}.$$

Proof. Note that all letters of A_6 appear in $\mathbf{g}$. Then doublets ab, ac, ba, bd, ce, df, ea, fb appear in $\mathbf{g}$ because they appear in the images of one letter. The images of these doublets generate two more doublets, cb and da, whose images do not create new doublets. □

Lemma 4.
The set of triplets in $g^\infty(\mathtt{a})$ *is*

$$T = \{\mathtt{aba}, \mathtt{abd}, \mathtt{acb}, \mathtt{ace}, \mathtt{bab}, \mathtt{bac}, \mathtt{bda}, \mathtt{bdf}, \mathtt{cba}, \mathtt{cea}, \mathtt{dab}, \mathtt{dfb}, \mathtt{eab}, \mathtt{fba}\}.$$

Proof. Triplets appear in the images of a letter of a doublet. Found in images of one letter are: aba, abd, ace, bab, bac, bdf, eab, fba. The images of doublets occurring in $\mathbf{g}$, in set D of Lemma 3, contain the extra triplets: acb, bda, cba, cea, dab, dfb. □

To prove the infinite word $g^\infty(\mathtt{a})$ is square-free first we discard squares containing less than four occurrences of the word $g(\mathtt{a}) = \mathtt{abac}$, Then squares containing at least four. The word $\mathtt{abac}$ is chosen because its occurrences in $g^\infty(\mathtt{a})$ correspond to $g(\mathtt{a})$ only, so they are used to synchronise the parsing of the word according to the codewords $g(a)$'s.

Lemma 5. *No square in $g^\infty(\mathtt{a})$ can contain less than four occurrences of $\mathtt{abac}$.*

Proof. Assume by contradiction that a square ww in $g^\infty(\mathtt{a})$ contains less than four occurrences of $\mathtt{abac}$. Let x be the shortest word whose image by g contains ww.

Then x is a factor of $g^\infty(\mathtt{a})$ that belongs to the set $\mathtt{a}((A_6 \setminus \{\mathtt{a}\})^*\mathtt{a})^4$. Since two consecutive occurrences of $\mathtt{a}$ in $g^\infty(\mathtt{a})$ are separated by a string of length at most 4 (the largest such string is indeed $\mathtt{bdfb}$ as a consequence of Lemma 3), the set is finite.

The square-freeness of all these factors has been checked via an elementary implementation of the test, which proves the result. □

Proposition 6. *No square in $g^\infty(\mathtt{a})$ can contain at least four occurrences of $\mathtt{abac}$.*

Table 1. Gaps: words between consecutive occurrences of $\mathtt{abac}$ in $g^\infty(\mathtt{a})$. They are images of gaps between consecutive occurrences of $\mathtt{a}$.

$g(\mathtt{b})$	$= \mathtt{babd}$	4
$g(\mathtt{cb})$	$= \mathtt{eabdfbabd}$	9
$g(\mathtt{bd})$	$= \mathtt{babdfbace}$	9
$g(\mathtt{ce})$	$= \mathtt{eabdfbace}$	9
$g(\mathtt{bdfb})$	$= \mathtt{babdfbaceabdfbabd}$	17

Proof (Sketch). The complete proof is by contradiction: let k be the maximal integer k for which $g^k(\mathtt{a})$ is square-free and let ww be a square occurring in $g^{k+1}(\mathtt{a})$. Distinguishing several cases according to the words between consecutive occurrences of $\mathtt{abac}$ (see Table 1), we deduce that $g^k(\mathtt{a})$ is not square-free, the contradiction. □

Corollary 7. *The infinite word $g^\infty(\mathtt{a})$ is square-free, or equivalently, the morphism g is weakly square-free.*

4 Binary translation

The second part of the proof of Theorem 2 consists in showing that the special square-free words on 6 letters introduced in the previous section can be transformed into the desired binary word. This is done with a second morphism h from A_6^* to B^* defined by

$$\begin{cases} h(\mathtt{a}) = \mathtt{10011}, \\ h(\mathtt{b}) = \mathtt{01100}, \\ h(\mathtt{c}) = \mathtt{01001}, \\ h(\mathtt{d}) = \mathtt{10110}, \\ h(\mathtt{e}) = \mathtt{0110}, \\ h(\mathtt{f}) = \mathtt{1001}. \end{cases}$$

Note that the codewords of h do not form a prefix code, nor a suffix code, nor even a uniquely-decipherable code! We have for example $g(\mathtt{ae}) = \mathtt{10011} \cdot \mathtt{0110} =$

$\mathtt{1001} \cdot \mathtt{10110} = g(\mathtt{fd})$. However, parsing the word $h(y)$ when y is a factor of $g^\infty(\mathtt{a})$ is unique due to the absence of some doublets in it (see Lemma 3). For example $\mathtt{fd}$ does not occur, which induces the unique parsing of $\mathtt{100110110}$ as $\mathtt{10011} \cdot \mathtt{0110}$.

Proposition 8. *The infinite word* $\mathbf{h} = h(g^\infty(\mathtt{a}))$ *contains no factor of exponent larger than* $7/3$. *It contains the* 12 *squares* $\mathtt{0}^2$, $\mathtt{1}^2$, $(\mathtt{01})^2$, $(\mathtt{10})^2$, $(\mathtt{001})^2$, $(\mathtt{010})^2$, $(\mathtt{011})^2$, $(\mathtt{100})^2$, $(\mathtt{101})^2$, $(\mathtt{110})^2$, $(\mathtt{01101001})^2$, $(\mathtt{10010110})^2$ *only. Words* $\mathtt{0110110}$ *and* $\mathtt{1001001}$ *are the only factors with an exponent larger than* 2.

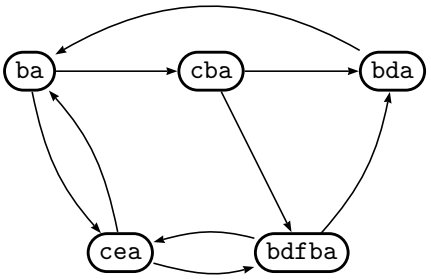


Figure 1. Graph showing immediate successors of gaps in the word $g^\infty(\mathtt{a})$: a suffix of it following an occurrence of $\mathtt{a}$ is the label of an infinite path.

The proof is far beyond this extended abstract. It is based on the fact that occurrences of $\mathtt{10011}$ in $\mathbf{h}$ identify occurrences of $\mathtt{a}$ in $\mathbf{g}$ and on the unique parsing mentioned above. It proceeds by considering several cases according to the gaps between consecutive occurrences of $\mathtt{a}$, which leads to analyse paths in the graph of Figure 1.

5 Conclusion

The constraint on the number squares imposed on binary words slightly differs from the constraint considered by Shallit [9]. The squares occurring in his word have period smaller than 8. Our word contains less squares but their maximal period is 8. Indeed it is impossible to have both constrains simultaneously for an infinite binary strings.

Looking at repetitions in words on larger alphabets, the subject introduces a new type of threshold, that we call the *bounded-repetitions threshold*. For the alphabet of a letters, it is defined as the smallest rational number t_a for which there exist an infinite word avoiding t^+-powers and containing a finite number of r-powers, where r is Dejean's repetitive threshold. Karhumäki and Shallit results as well as ours show that $t_2 = 7/3$. Values for larger alphabets remains to explore.

References

1. G. Badkobeh and M. Crochemore: *An infinite binary word containing only three distinct squares*, 2010, submitted.
2. F. Dejean: *Sur un théorème de Thue.* J. Comb. Theory, Ser. A, 13(1) 1972, pp. 90–99.
3. A. S. Fraenkel and J. Simpson: *How many squares must a binary sequence contain?* Electr. J. Comb., 2 1995.
4. T. Harju and D. Nowotka: *Binary words with few squares.* Bulletin of the EATCS, 89 2006, pp. 164–166.

5. J. Karhumäki and J. Shallit: *Polynomial versus exponential growth in repetition-free binary words*. J. Comb. Theory, Ser. A, 105(2) 2004, pp. 335–347.
6. M. Lothaire, ed., *Combinatorics on Words*, Cambridge University Press, second ed., 1997.
7. N. Rampersad, J. Shallit, and M. wei Wang: *Avoiding large squares in infinite binary words*. Theor. Comput. Sci., 339(1) 2005, pp. 19–34.
8. M. Rao: *Last cases of Dejean's conjecture*, in WORDS 2009, A. Carpi and C. de Felice, eds., University of Salerno, Italy, 2009.
9. J. Shallit: *Simultaneous avoidance of large squares and fractional powers in infinite binary words*. Int'l. J. Found. Comput. Sci., 15 2004, pp. 317–327.
10. A. Thue: *Über unendliche Zeichenreihen*. Norske vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana, 7 1906, pp. 1–22.

Average Number of Runs and Squares in Necklace

Kazuhiko Kusano* and Ayumi Shinohara

Graduate School of Information Science, Tohoku University,
Aramaki aza Aoba 6-6-05, Aoba-ku, Sendai-shi 980-8579, Japan
{kusano@shino., ayumi@}ecei.tohoku.ac.jp

Abstract. A repetition is an important property of a string. In this paper we consider the average number of occurrences of primitively rooted repetitions in necklace. First, we define *circular square* and *circular run* for a string and show the average number of them. Using these results, we obtain the average number of squares, the average number of runs and the average sum of exponents of runs in a necklace, exactly.

Keywords: repetition, run, combinatorics on words

1 Introduction

A repetition is a fundamental property of a string. It can be applied to string processing or data compression. We are interested in run (as known as maximal repetition), which is non-extendable repetition. Kolpakov and Kucherov showed that the maximal number $\rho(n)$ of runs in a string of length n is $\rho(n) \leq cn$ for some constant c [6]. The exact value of $\rho(n)$ is still unknown and it is conjectured that $\rho(n) < 1$. The current best upper bound is $\rho(n) < 1.029n$ [3,4]. On the other hand, there are approaches to show the lower bounds of $\rho(n)$ constructing run-rich strings. The best lower bound is $\rho(n) > 0.945$ [9,11]. A repetition count of a run is called an exponent. It is proved that the maximal sum of exponents is also linear and the current best upper bound is $2.9n$ [2]. It is conjectured that the sum is less than $2n$ [7].

A square is a substring of the form u^2. We consider the primitively rooted square and count occurrences of squares instead of distinct squares. Counting squares in this way, it is known that the maximal number of squares is $O(n \log n)$ [1].

Although the maximal number of runs is unknown, the average number of runs in a string of length n is shown exactly as follows [10]:

$$R_s(n, \sigma) = \sum_{p=1}^{\frac{n}{2}} \sigma^{-2p-1} \left((n-2p+1)\sigma - (n-2p)\right) \sum_{d|p} \mu\left(\frac{p}{d}\right) \sigma^d,$$

where σ is alphabet size and $\mu(n)$ is the Möbius function. The average number of squares and the average sum of exponents of runs are also presented [8]:

$$S_s(n, \sigma) = \sum_{p=1}^{\frac{n}{2}} \sigma^{-2p}(n-2p+1) \sum_{d|p} \mu\left(\frac{p}{d}\right) \sigma^d,$$

$$E_s(n, \sigma) = \sum_{p=1}^{\frac{n}{2}} \sigma^{-2p-1} \left(2(n-2p+1)\sigma - \left(2 - \frac{1}{p}\right)(n-2p)\right) \sum_{d|p} \mu\left(\frac{p}{d}\right) \sigma^d.$$

* Supported in part by Grant-in-Aid for JSPS Fellows

Kazuhiko Kusano, Ayumi Shinohara: Average Number of Runs and Squares in Necklace, pp. 167–177 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

In [9,11], to construct run-rich strings they considered repeated strings or necklace. Therefore we focus on the average number of repetitions in necklace. To obtain the average number of repetitions in necklace we define *circular square* and *circular run* for string and show the average number of them, exactly.

In Section 2 we give some definitions and basic facts. In Section 3 we show the average number of circular squares and circular runs and the average sum of exponents of circular runs in a string. In Section 4 we derive the average numbers of squares, the average number of runs and the average sum of exponents of runs in a necklace.

2 Preliminary

Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \dots\}$ be an alphabet of size σ. We denote the set of all strings of length n on Σ by Σ^n and the length of a string w by $|w|$. For a string $w = xyz$, strings x, y and z are called *prefix*, *substring* and *suffix* of w, respectively. We denote ith letter of a string w by $w[i]$ and a substring $w[i]w[i+1]\dots w[j]$ of w by $w[i..j]$.

A *necklace* is a word which can be obtained by joining the ends of a string. We denote a necklace of a string w by $\langle w \rangle$.

For a string w of length n and positive integer $p < n$, we say that w has a period p if and only if $w[i] = w[i+p]$ holds for any i, $1 \le i \le n-p$. We denote the set of periods of w by $\mathit{period}(w)$. For periods of strings, the following lemma is known [5].

Lemma 1. *Let p and q be periods of a string w. If $|w| \ge p + q - \gcd(p,q)$, w has also period $\gcd(p,q)$.*

A string w is *primitive* if w can not be written as $w = u^k$ by string u and integer $k \ge 2$.

We call a substring $w[i..j]$ a *repetition* if $w[i..j]$ has the smallest period $p \le \frac{j-i+1}{2}$ and denote the substring by triplet $\langle i, j, p \rangle$. We say that $w[i..p]$ is the *root* of the repetition. By Lemma 1, the root of a repetition is primitive. The *exponent* of the repetition is $\frac{j-i+1}{p}$.

A *square* is a repetition whose exponent is exactly 2. We consider only squares which have a primitive root. A *run* is a repetition which has non-extendability, that is, a run $\langle i, j, p \rangle$ in w satisfies the following two conditions:

$$\begin{aligned} i = 1 \quad &\text{or} \quad w[i-1] \neq w[i+p-1], \\ j = n \quad &\text{or} \quad w[j+1] \neq w[j-p+1]. \end{aligned}$$

We denote a string of infinite length, obtained by repeating string w to both left and right, by w^ω. For a string w of length n and integer i, $w^\omega[i] = w[i\%n]$, where the operator $x\%y$ represents a number z such that $1 \le z \le y$ and $z \equiv x \pmod{y}$. In this paper, we define a *circular run* (*circular square*, resp.) for a string w as a run (square, resp.) in w^ω and which starts between 1 and $|w|$. We denote the number of circular squares by $\mathit{csqr}(w)$, the number of circular runs in a string w by $\mathit{crun}(w)$ and the sum of exponents of runs by $\mathit{cexp}(w)$. For a necklace $\langle w \rangle$, we define number of runs $\mathit{run}(\langle w \rangle)$, number of squares $\mathit{sqr}(\langle w \rangle)$ and sum of exponents of runs $\mathit{sqr}(\langle w \rangle)$ as follows:

$$\begin{aligned} \mathit{run}(\langle w \rangle) &= \mathit{crun}(w), \\ \mathit{sqr}(\langle w \rangle) &= \mathit{csqr}(w), \\ \mathit{exp}(\langle w \rangle) &= \mathit{cexp}(w). \end{aligned}$$

3 Average number of circular repetitions in a string

For a string of length n and alphabet size σ, the average number of circular squares, the average number of circular runs and the average sum of exponents of circular runs are defined as:

$$S_c(n,\sigma) = \frac{1}{|\Sigma^n|}\sum_{w\in\Sigma^n} csqr(w),$$
$$R_c(n,\sigma) = \frac{1}{|\Sigma^n|}\sum_{w\in\Sigma^n} crun(w),$$
$$E_c(n,\sigma) = \frac{1}{|\Sigma^n|}\sum_{w\in\Sigma^n} cexp(w).$$

3.1 Average number of circular squares

To obtain these values, we count repetitions in all strings of length n. We consider repetitions classified according to their position and period. For the position, it is sufficient to consider only repetitions at one position. The total number of occurrences can be obtained as the product of this number and the length of strings.

Lemma 2. *For a string f and integer i, let $\Sigma_{f,i}$ be the set of string w of length n such that w^ω contains f at i. For any integer i and j, $|\Sigma_{f,i}| = |\Sigma_{f,j}|$.*

Proof. We may assume without loss of generality that $i \le j$. If w is an element of $\Sigma_{f,i}$, then $w[i\%n]w[(i+1)\%n]\dots w[(i+|f|)\%n] = f$. Let $w' = w[n-(j-i)+1..n]w[1..j-i]$. Since w' satisfies the condition $w'[j\%n]w'[(j+1)\%n]\dots w'[(i+|f|)\%n] = f$, w' is in $\Sigma_{f,j}$.

Although the circular repetition is defined as the repetition in an infinity string, the period of the primitive rooted repetition is not so long.

Lemma 3. *Let w be the string of length n. The period of circular square in w is at most n.*

Proof. Let $\langle i,j,p\rangle$ be the circular run in w. If we suppose that $p > n$, the substring $w^\omega[i..j]$ of length $2p$ has two periods n and p. By Lemma 1, it also has period $\gcd(n,p)$, which is less than p and the divisor of p. So the primitive root $w^\omega[i..i+p-1]$ can be written as $w^\omega[i..i+p-1] = u^k$ using a string u and integer $k = \frac{p}{q} > 1$, a contradiction.

The length of the circular square in string of length n can be longer than n. For example, the string `abaab` of length 5 contains the circular square $\langle 1,6,3\rangle$ of length 6.

We consider the number of circular squares in all strings of length n at the position 1. Let $S_{f1}(p,\sigma)$ be the set of squares of period p and alphabet size σ; that is,

$$S_{f1}(p,\sigma) = \{vv : v \in Prim_{p,\sigma}\}.$$

Since a string w^ω may contain at most one elements of $S_{f1}(p,\sigma)$ at the position 1, the number of circular squares of period p in Σ^n equals to the number of the strings w such that w^ω contains the element of $S_{f1}(p,\sigma)$ at the position 1. More generally, we consider the set of strings of length l, instead of $S_{f1}(p,\Sigma)$.

Lemma 4. *Let F be a subset of Σ^l. For the number $N_F(n,\sigma)$ of strings w such that $|w| = n$ and $w^\omega[1..l] \in F$,*

$$N_F(n,\sigma) = \begin{cases} |F|\sigma^{n-l} & \text{if } l \le n, \\ |G| & \text{if } l > n, \end{cases}$$

where the set G is a subset of F and whose elements have the period n; that is

$$G = \{w \in F : n \in period(w)\}.$$

Proof.

1. Case $l \le n$
   The string w^ω contains the element of F at the position 1 if and only if $w[1..l] \in F$. Let C be the set of such strings. We have
   $$C = \{uv : u \in F, v \in \Sigma^{n-l}\}.$$
   The number of elements of C is $|F|\,\sigma^{n-l}$.
2. Case $l > n$
   Since w^ω has the period n, the substring $w^\omega[1..l]$ also has period n. For the element g of G, the suffix $g[n+1..l]$ is the repetition of $g[1..n]$. Therefore, the prefixes of length n of the elements of G are different. So, $N_F(n,\sigma) = |G|$.

Let $S_{f2}(n,p,\sigma)$ be the set of squares of length p and alphabet size σ and which can be contained in a repetition of string of length n; that is

$$S_{f2}(n,p,\sigma) = \{w : w \in S_{f1}(p,\sigma), n \in period\,(w)\}.$$

To obtain the size of $S_{f2}(n,b,\sigma)$, for integer d of divisor of p, we define $S_{f3}(n,p,d,\sigma)$ and $S_{f4}(n,p,d,\sigma)$ as follows:

$$S_{f3}(n,p,d,\sigma) = \left\{u^{\frac{2p}{d}} : u \in Prim_{d,\sigma}, n \in period\left(u^{\frac{2p}{d}}\right)\right\},$$
$$S_{f4}(n,p,d,\sigma) = \left\{u^{\frac{2p}{d}} : u \in \Sigma^d, n \in period\left(u^{\frac{2p}{d}}\right)\right\}.$$

We see that $S_{f2}(n,p,\sigma) = S_{f3}(n,p,p,\sigma)$. Since any string can be written uniquely as an integer power of a primitive string, $S_{f4}(n,p,d,\sigma) = \bigcup_{d|p} S_{f3}(n,p,d,\sigma)$.

First, we consider $S_{f4}(n,p,d,\sigma)$.

Lemma 5. *For the element $u^{\frac{2p}{d}}$ of $S_{f4}(n,p,d,\sigma)$, $u^{\frac{p}{d}}$ has a period $n-p$.*

Proof. Let $v = u^{\frac{p}{d}}$. By the definition of $S_{f4}(n,p,d,\sigma)$, for any position $1 \le i \le 2p-n$, $v^2[i] = v^2[i+n]$. For any position $1 \le j \le p-(n-p)$, $v[j] = v^2[j] = v^2[j+n] = v[j+n-p]$.

For $u^{\frac{2p}{d}} \in S_{f4}(n,p,d,\sigma)$, the string $u^{\frac{p}{d}}$ of length p has two periods d and $n-p$. If $d+(n-p) \le p$ such that $d \le 2p-n$, from lemma 1, $u^{\frac{p}{d}}$ also has a period $\gcd(d, n-p)$. In the other case, $u^{\frac{p}{d}}$ has another period.

Lemma 6. *For the element $u^{\frac{2p}{d}}$ of $S_{f4}(n,p,d,\sigma)$, if $d > 2p-n$, $u^{\frac{p}{d}}$ has a period $d-(2p-n)$.*

Proof. For any position $1 \leq i \leq d-(d-(2p-n))$, $u[i] = u^{\frac{2p}{d}}[i] = u^{\frac{2p}{d}}[i+n] = u[i+n-(2p-d)] = u[i+d-(2p-n)]$.

Therefore,

$$S_{f4}(n,p,d,\sigma) = \begin{cases} \{s^{\frac{2p}{\gcd(d,n-p)}} : s \in \Sigma^{\gcd(d,n-p)}\} & \text{if } d \leq 2p-n, \\ \{s^{\frac{2p}{d-2p+n}} : s \in \Sigma^{d-2p+n}\} & \text{if } d > 2p-n. \end{cases}$$

The number of elements of $S_{f4}(n,p,d,\sigma)$ can be written as

$$|S_{f4}(n,p,d,\sigma)| = \delta_s(n,p,d,\sigma),$$

where

$$\delta_s(n,p,d,\sigma) = \begin{cases} \sigma^{\gcd(d,n-p)} & \text{if } d \leq 2p-n, \\ \sigma^{d-2p+n} & \text{if } d > 2p-n. \end{cases}$$

Lemma 7. *The number of elements of $S_{f2}(n,p,\sigma)$ is as follows:*

$$|S_{f2}(n,p,\sigma)| = \sum_{d|p} \mu\left(\frac{p}{d}\right) \delta_s(n,p,d,\sigma).$$

Proof. Since

$$S_{f4}(n,p,p,\sigma) = \bigcup_{d|p} S_{f3}(n,p,d,\sigma),$$

we see that

$$|S_{f4}(n,p,p,\sigma)| = \sum_{d|p} |S_{f3}(n,p,d,\sigma)|.$$

Applying the Möbius inversion formula to this equation we have that

$$\begin{aligned} |S_{f2}(n,p,\sigma)| &= |S_{f3}(n,p,p,)| \\ &= \sum_{d|p} \mu\left(\frac{p}{d}\right) |S_{f4}(n,p,d,\sigma)| \\ &= \sum_{d|p} \mu\left(\frac{p}{d}\right) \delta_s(n,p,d,\sigma). \end{aligned}$$

By Lemma 4 and 7 we can derive the following theorem.

Theorem 8. *For any positive integer n and σ, the average number of circular squares in a string of length n and alphabet size σ is*

$$S_c(n,\sigma) = \frac{n}{\sigma^n} \sum_{p=1}^{n} \sum_{d|p} \mu\left(\frac{p}{d}\right) \delta_s(n,p,d,\sigma).$$

3.2 Average number of circular runs

In this subsection, we show the average number of circular runs in string of length n and alphabet size σ.

Unlike circular squares, whether a substring is a circular run or not depends on the characters next to the substring. For example, the repetition $\langle 2, 5, 2\rangle$ is a run in `aabab`, while the repetition is not a run in `babab` since the repetition can be extended to left. Instead of the set $S_{f1}(n,p)\sigma$, we consider a set $R_{f1}(n,p)\sigma$ of string such that

$$R_{f1}(p,\sigma) = \{cvv : c \neq v[p], v \in Prim_{p,\sigma}\}.$$

There is a circular run in w at the position i if and only if w^ω contains an element of $R_{f1}(n,p)\sigma$ at the position $i-1$.

The Lemma 2 can be applied to the element of $R_{f1}(p,\sigma)$. From Lemma 3 the period of a circular run in a string of length n does not exceed n, since a circular run contains at least one circular square. We consider the number of occurrences of the element of $R_{f1}(p,\sigma)$ in w^ω for all strings w of length n.

We define, for d of divisor of p, $R_{f2}(n,p,\sigma)$, $R_{f3}(n,p,d,\sigma)$ and $R_{f4}(n,p,d,\sigma)$ as follows:

$$\begin{aligned}
R_{f2}(n,p,\sigma) &= \{w \in S_{f1}(p,\sigma) : n \in period\,(w)\},\\
R_{f3}(n,p,d,\sigma) &= \left\{cu^{\frac{2p}{d}} : c \neq u[d], u \in Prim_{d,\sigma}, n \in period\left(cu^{\frac{2p}{d}}\right)\right\},\\
R_{f4}(n,p,d,\sigma) &= \left\{cu^{\frac{2p}{d}} : c \neq u[d], u \in \Sigma^d, n \in period\left(cu^{\frac{2p}{d}}\right)\right\}.
\end{aligned}$$

Since d is a divisor of p, we see that $u^{\frac{p}{d}} = u[d]$.

The Lemma 5 and 6 also hold for $R_{f4}(n,p,d,\sigma)$. The condition $c \neq u[d]$ sometimes makes $R_{f4}(n,p,d,\sigma)$ be empty.

Lemma 9. *If either $d \leq 2p-n$ or $d \equiv 0 \pmod{d-(2p-n)}$, the set $R_{f4}(n,p,d,\sigma)$ is empty.*

Proof. For the element $cu^{2p}d \in R_{f4}(n,p,d,\sigma)$, $u^{\frac{p}{d}}$ has the period $n-p$ and d. If $d \leq 2p-n$ that is $d+(n-p) \leq p$, from Lemma 1, $u^{\frac{p}{d}}$ also has period $t = gcd(d, n-p)$. In this case, $c = u^{\frac{p}{d}}[n-p] = u[d]$ and the condition $c \neq u[d]$ does not hold.

For the case $d > 2p-n$, $c = u^{\frac{2p}{d}}[n] = u[n-(2p-d)] = u[d-(2p-n)]$. Lemma 6 says that u has the period $d-(2p-n)$ such that $u[d] = u[d\%(d-(2p-n))]$. If $d \equiv 0 \pmod{d-(2p-n)}$, we have that $d-(2p-n) = d\%(d-(2p-n))$ and $c = u[d-(2p-n)] = [d\%(d-(2p-n))] = u[d]$.

For the case $d > 2p-n$ and $d \not\equiv 0 \pmod{d-(2p-n)}$, the set $R_{f4}(n,p,d,\sigma)$ can be written as:

$$R_{f4}(n,p,d,\sigma) = \left\{cs^{\frac{2p}{d-2p+n}} : c \neq s[d-2p+n], s \in \Sigma^{d-2p+n}\right\}.$$

Therefore,

$$|R_{f4}(n,p,d,\sigma)| = \delta_r\,(n,p,d,\sigma)\,,$$

where

$$\delta_r\,(n,p,d,\sigma) = \begin{cases} (\sigma-1)\sigma^{d-2p+n-1} & \text{if } d > 2p-n \text{ and } d \not\equiv 0 \pmod{d-(2p-n)},\\ 0 & \text{otherwise.}\end{cases}$$

We can derive $|R_{f2}(n,p,\sigma)|$ as follows:

$$\begin{aligned}|R_{f2}(n,p,\sigma)| &= |R_{f3}(n,p,p,)\,| \\ &= \sum_{d|p} \mu\left(\frac{p}{d}\right) |R_{f4}(n,p,d,\sigma)| \\ &= \sum_{d|p} \mu\left(\frac{p}{d}\right) \delta_r\left(n,p,d,\sigma\right).\end{aligned}$$

Theorem 10. *For any positive integers n and σ, the average number of circular runs in a string of length n and alphabet size σ is*

$$R_c\left(n,\sigma\right) = \frac{n}{\sigma^n} \sum_{p=1}^{n} \sum_{d|p} \mu\left(\frac{p}{d}\right) \delta_r\left(n,p,d,\sigma\right).$$

3.3 Average sum of exponents of circular runs

A circular run contains circular squares of same period. For example, for string $w =$ `abaabaab`, a circular run $\langle 1,8,3\rangle$ contains three circular squares $\langle 1,6,3\rangle$, $\langle 2,7,3\rangle$ and $\langle 3,8,3\rangle$. The number of circular squares depends on period and exponent of the run.

Lemma 11. *A circular run of period p and exponent e contains $(e-2)p+1$ circular squares of period p.*

Proof. Let $\langle i,j,p\rangle$ be a circular run in string w. For any position $i \le k \le j-p$, $w[k] = w[k+p]$ and a substring $w[k..k+p]$ is primitive since $w[k..k+p]$ is a conjugate of primitive string $w[i..i+p]$. The number of circular squares contained the run is $j-2p-i+2$. The exponent of the run is $e = \frac{j-i+1}{p}$. The number of circular squares can be written as $(e-2)p+1$.

Although any circular run contains circular squares, some circular squares are not contained in a run of the same period. For example, for a string $w =$ `abc`, there are circular runs $\langle 1,6,3\rangle$, $\langle 2,7,3\rangle$ and $\langle 3,8,3\rangle$ and there are no circular run containing the squares. For a string $w =$ `abab`, such circular squares are $\langle 1,4,2\rangle$, $\langle 2,5,2\rangle$, $\langle 3,6,2\rangle$ and $\langle 4,7,2\rangle$.

Lemma 12. *For a primitive string u of length p and integer k, u^k contains n circular squares of period p which is not contained in a circular run of period p.*

Proof. Let $w = u^k$. For any position i, $\langle i, i+2p-1, p\rangle$ is a circular square, since $w[i..i+p-1] = w[i+p..i+2p-1]$ and $w[i..i+p-1]$ is primitive by the definition of w. There is n circular squares of length p. There is no circular run of period p in w, since, for any position i, $w^\omega[i] = w^\omega[i+p]$ and a repetition of period p cannot satisfy non-extendability.

By contradiction, we show that there is no circular square of period $p' \neq p$ which contained in a circular run of period p'. Assume that w contains a circular square $\langle i,j,p'\rangle$. If there is no circular run of period p' containing $\langle i,j,p'\rangle$, the repetition $\langle i,j,p'\rangle$ can extend to both left and right infinitely. It mean that w^ω has the period p'. From Lemma 1 w^ω also has a period $t = \gcd(p',p)$. The period p' is not multiple of p since p' is the period of a circular square. The period t is less than p and a divisor of p, a contradiction.

From Lemma 11 and 12, we can derive the average sum of exponents of circular runs.

Theorem 13. *For positive integers n and σ, the average sum of exponents of circular runs in a string of length n and alphabet size σ is*

$$E_c(n,\sigma) = \frac{n}{\sigma^n}\left(\sum_{p=1}^{n}\frac{1}{p}\sum_{d|p}\mu\left(\frac{p}{d}\right)\delta_s(n,p,d,\sigma) + \sum_{p=1}^{n}\left(2-\frac{1}{p}\right)\sum_{d|p}\mu\left(\frac{p}{d}\right)\delta_r(n,p,d,\sigma) - \sum_{p|n}\frac{1}{p}\sum_{d|p}\mu\left(\frac{p}{d}\right)\sigma^d\right).$$

Proof. Consider a string w of length n. A string w can be uniquely written as $w = u^k$ where u is primitive string and k is integer. Let $csqr_p(w)$, $crun_p(w)$ and $cexp_p(w)$ be the number of circular squares of period p in w, the number of circular runs of period p in w and the sum of exponents of circular runs of period p in w, respectively. Applying Lemma 11 for each circular runs in w we have

$$csqr_p(w) - n[k=p] = (cexp_p(w) - 2crun_p(w))p + crun_p(w)$$
$$cexp_p(w) = \frac{1}{p}csqr_p(w) + \left(2-\frac{1}{p}\right)crun_p(w) - \frac{1}{p}n[k=p],$$

where $[k=p]$ is defined as 1 if $k=p$ and 0 if $k \neq p$. The number of circular squares not to be contained circular run of the same period is $n[k=p]$. Summing them up for each strings and each periods, from Theorem 8 and 10, we can obtain $E_c(n,\sigma)$. The number of strings of length n which can be written as $w = u^k$ equals to the number of primitive strings of length $p = \frac{n}{k}$. It is known that the number is $\sum_{d|p}\mu\left(\frac{p}{d}\right)\sigma^d$.

4 Average number of repetitions in necklace

Although we defined the number of repetitions in a necklace $\langle w\rangle$ equals to the number of circular repetitions in the string w, the average number of repetitions in a necklace of length n and the average number of circular repetitions in a string of length n are different.

Example 14. Let length $n = 4$ and alphabet size $\sigma = 2$. All strings of length n and the numbers of circular runs they contain are as follows:

aaaa 0	aaab 1	aaba 1	aabb 2	abaa 1	abab 0	abba 2	abbb 1
baaa 1	baab 2	baba 0	babb 1	bbaa 2	bbab 1	bbba 1	bbbb 0

Thus, the average number of circular runs in string is $\frac{16}{16} = 1$.

All necklaces of length n and the numbers of runs they contain are as follows:

$\langle$aaaa$\rangle$ 0 $\langle$aaab$\rangle$ 1 $\langle$aabb$\rangle$ 2 $\langle$abab$\rangle$ 0 $\langle$abbb$\rangle$ 1 $\langle$bbbb$\rangle$ 0

Thus, the average number of runs in necklace is $\frac{4}{6} = \frac{2}{3}$.

If and only if a string w of length n is primitive, there is n strings v such that $\langle v\rangle = \langle w\rangle$. Consider the number of repetitions in non-primitive string.

Lemma 15. *For string w and integer k, $csqr\left(w^k\right) = k\ csqr\left(w\right)$, $crun\left(w^k\right) = k\ crun\left(w\right)$ and $cexp\left(w^k\right) = k\ cexp\left(w\right)$.*

Proof. By the definition of w^ω, w^ω and $\left(w^k\right)^\omega$ are the same strings. Shifting w^ω to left or right by $|w|$, we get the same string. If there is a repetition in w^ω at the position i, repetitions also exist at the positions $i+|w|$, $i+2|w|$, $\ldots$, $i+(k-1)|w|$.

It is known that the number $|NL_{n,\sigma}|$ of necklaces of length n and alphabet size σ is

$$|NL_{n,\sigma}| = \frac{1}{n}\sum_{d|n}\phi\left(\frac{n}{d}\right)\sigma^d,$$

where $\phi\left(n\right)$ is the Euler's phi function. The function $\phi\left(n\right)$ is defined to be the number of integers less or equal to n which are coprime to n and can be written as:

$$\phi\left(n\right) = \sum_{d|n}\mu\left(\frac{n}{d}\right)d.$$

Using the method calculating the number of necklaces and Lemma 15, we can obtain the number of squares in all necklaces.

Lemma 16. *The number of squares in all necklaces of length n and alphabet size σ is*

$$|NL_{n,\sigma}|\,S_n\left(d,\sigma\right) = \frac{1}{n}\sum_{d|n}\phi\left(\frac{n}{d}\right)\frac{n}{d}\sigma^d S_c\left(d,\sigma\right).$$

Proof. Let T be a multi set of strings obtained by cutting necklaces $NL_{n,\sigma}$ in n ways. For example, for

$$NL_{4,2} = \{\langle\mathtt{aaaa}\rangle, \langle\mathtt{aaab}\rangle, \langle\mathtt{aabb}\rangle, \langle\mathtt{abab}\rangle, \langle\mathtt{abbb}\rangle, \langle\mathtt{bbbb}\rangle\}.$$

T is as follows:

$$T = \left\{\begin{array}{ll}\mathtt{aaaa\ aaaa\ aaaa\ aaaa} & \mathtt{abab\ baba\ abab\ baba}\\ \mathtt{aaab\ aaba\ abaa\ baaa} & \mathtt{abbb\ bbba\ bbab\ babb}\\ \mathtt{aabb\ abba\ bbaa\ baab} & \mathtt{bbbb\ bbbb\ bbbb\ bbbb}\end{array}\right\}.$$

We see that $|T| = n|NL_{n,\sigma}|$ and the number of circular squares in T is $n\,|NL_{n,\sigma}|S_n\left(d,\sigma\right)$. The number of $w \in \Sigma^n$ in T equals to the number of k such that $1 \le k \le n$ and $w = w[k+1..n]w[1..k]$. This equations holds if w can be written as $w = u^{\frac{n}{|u|}}$ using $u \in \Sigma^{\gcd(k,n)}$. Thus, from Lemma 15,

$$n\,|NL_{n,\sigma}|\,S_n\left(d,\sigma\right) = \sum_{k=1}^{n}\frac{n}{\gcd(k,n)}\sigma^{\gcd(k,n)}S_c\left(\gcd(k,n),\sigma\right).$$

Since $\gcd(k,n)$ is a divisor of n, this equation can be transformed, with $d = \gcd(k,n)$, as follows:

$$\begin{aligned}
|NL_{n,\sigma}|\, S_n(p,\sigma) &= \frac{1}{n}\sum_{d|n}\sum_{k=1}^{n}\frac{n}{d}\sigma^d S_c(d,\sigma)\,[d=\gcd(k,n)] \\
&= \frac{1}{n}\sum_{d|n}\left(\frac{n}{d}\sigma^d S_c(d,\sigma)\sum_{k=1}^{n}\left[\frac{k}{d}\perp\frac{n}{d}\right]\right) \\
&= \frac{1}{n}\sum_{d|n}\left(\frac{n}{d}\sigma^d S_c(d,\sigma)\sum_{k'=1}^{\frac{n}{d}}\left[k'\perp\frac{n}{d}\right]\right) \\
&= \frac{1}{n}\sum_{d|n}\frac{n}{d}\sigma^d S_c(d,\sigma)\,\phi\left(\frac{n}{d}\right).
\end{aligned}$$

From the number of necklaces and Lemma 16, we can derive the following theorem.

Theorem 17. *For integers n and σ, the average number of squares in necklace of length n and alphabet size σ is*

$$S_n(p,\sigma) = \frac{\sum_{d|n}\phi\left(\frac{n}{d}\right)\frac{n}{d}\sigma^d S_c(d,\sigma)}{\sum_{d|n}\phi\left(\frac{n}{d}\right)\sigma^d}.$$

Similarly we obtain the average number and the average sum of exponents of runs in necklace.

Theorem 18. *For integers n and σ, the average number of runs in necklace of length n and alphabet size σ is*

$$R_n(p,\sigma) = \frac{\sum_{d|n}\phi\left(\frac{n}{d}\right)\frac{n}{d}\sigma^d R_c(d,\sigma)}{\sum_{d|n}\phi\left(\frac{n}{d}\right)\sigma^d},$$

and the average sum of exponents of runs in necklace is

$$E_n(p,\sigma) = \frac{\sum_{d|n}\phi\left(\frac{n}{d}\right)\frac{n}{d}\sigma^d E_c(d,\sigma)}{\sum_{d|n}\phi\left(\frac{n}{d}\right)\sigma^d}.$$

5 Conclusion

In this paper we defined circular squares and circular runs in a string and considered squares and runs in a necklace. They are useful for analysing ordinary squares and runs, especially a lower bound of the number of them. We showed the average number of runs, the average number of squares and the average number of sum of exponents of runs in a necklace. It would also be interesting problem to analyse the average number of distinct repetitions instead of their occurrences.

References

1. M. Crochemore: *An optimal algorithm for computing the repetitions in a word*. Information Processing Letters, 12 1981, pp. 244–250.
2. M. Crochemore and L. Ilie: *Analysis of maximal repetitions in strings*, in Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2007), vol. 4708 of LNCS, Springer-Verlag, 2007, pp. 465–476.
3. M. Crochemore, L. Ilie, and L. Tinta: *The "runs" conjecture*. http://www.csd.uwo.ca/~ilie/runs.html.
4. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture*, in Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008), vol. 5029 of LNCS, Springer-Verlag, 2008, pp. 290–302.
5. N. Fine and H. Wilf: *Uniqueness theorems for periodic functions*. Proceedings of the American Mathematical Society, 1965, pp. 109–114.
6. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 1999), IEEE Computer Society, 1999, pp. 596–604.
7. R. Kolpakov and G. Kucherov: *On the sum of exponents of maximal repetitions in a word*, Tech. Rep. 99-R-034, LORIA, France, 1999.
8. K. Kusano, W. Matsubara, A. Ishino, and A. Shinohara: *Average value of sum of exponents of runs in strings*, in Proceedings of the Prague Stringology Conference 2008, Czech Technical University in Prague, 2008, pp. 185–192.
9. W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara: *A series of run-rich strings*, in Proceedings of the 3rd International Conference on Language and Automata Theory and Applications (LATA 2009), Springer, 2009, pp. 578–587.
10. S. J. Puglisi and J. Simpson: *The expected number of runs in a word*. Australasian Journal of Combinatorics, 42 2008, pp. 45–54.
11. J. Simpson: *Modified padovan words and the maximum number of runs in a word*. Australasian Journal of Combinatorics, 46 2010, pp. 129–146.

The Number of Runs in a Ternary Word

Hideo Bannai[1], Mathieu Giraud[2], Kazuhiko Kusano[3], Wataru Matsubara[3], Ayumi Shinohara[3], and Jamie Simpson[4]

[1] Kyushu University, Japan
[2] Université Lille, France
[3] Tohoku University, Japan
[4] Curtin University, Australia

Abstract. It is not known that the maximum number of runs in a word of length n is attained by a binary word though it seems likely that this is the case. In this note, we report observations on runs in *ternary words*, in which every small factor contains all three letters.

1 Introduction

A *run* of period p in a word x is a factor $\mathbf{x}[i..j]$ with least period p, length at least $2p$ and such that neither $\mathbf{x}[i-1..j]$ nor $\mathbf{x}[i..j+1]$ has period p. Runs are also called *maximal repetitions*. Let $r(x)$ be the number of runs in x. In recent years there has been great interest in the maximum number of runs that can occur in a word of length n which we call $\rho(n)$. In 2000 Kolpakov and Kucherov [5] showed that $\rho(n) = O(n)$ but their method gave no information about the size of the implied constant. Since then a number of authors have obtained upper and lower bounds on $\rho(n)$, the best to date being that

$$0.944 < \lim_{n\to\infty} \frac{\rho(n)}{n} < 1.029 \tag{1}$$

The upper bound here is due to Crochemore, Ilie and Tinta [1,2], and the lower bound to Kusano et al [6] and Simpson [8]. It is known [4] that that $\lim_{n\to\infty} \rho(n)/n$ exists. It is also known that the expected number of runs in binary words of length n tends to about $0.412n$ as n goes to infinity and for ternary words to about $0.305n$ [7].

It is not known that the maximum number of runs in a word of length n is attained by a binary word though it seems likely that this is the case. Between lengths 17 and 35, all the words with the maximum number of runs are binary [3]. More generally, the high density words we know about are binary and [7] showed the expected density decreases with alphabet size.

Part of the problem is that whatever you can do with a two letter alphabet you can do with a three letter alphabet by just not using the third letter. Even if we insist that each letter be used we can achieve the same asymptotic run density with three letters as with two by taking a good binary word on the alphabet $\{a, b\}$ and attaching c to the front. Insisting that the frequency of each letter is greater than some bound doesn't help. We could take a binary word of length $n/3$, make three copies of it using the alphabets $\{a, b\}$, $\{b, c\}$ and $\{c, a\}$ then concatenate them. This will give a word of length n with the same run density as the original word and with each letter having frequency $n/3$. In both these cases we are dealing with binary words in disguise.

Hideo Bannai, Mathieu Giraud, Kazuhiko Kusano, Wataru Matsubara, Ayumi Shinohara, Jamie Simpson: The Number of Runs in a Ternary Word, pp. 178–181 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

To make a word really ternary, we use the following definition. Let k be an integer with $k \geq 3$. A *ternary word of order k* is a word on the alphabet $\{a, b, c\}$ in which each factor of length k contains at least one occurrence of each letter.

Say that the maximum number of runs in such a word of length n is $\rho(k, n)$. We have

$$\rho(n) \geq \rho(k+1, n) \geq \rho(k, n)$$

and, when the order goes to the infinity, $\rho(k, n)$ will approach $\rho(n)$. Showing that

$$\lim_{n\to\infty} \frac{\rho(n)}{n} > \lim_{n\to\infty} \frac{\rho(k, n)}{n}$$

for all k would be equivalent to showing that, for bounded order, two letters are better than three. Towards this aim we have obtained upper and lower bounds for $\rho(k, n)$ for various values of k.

2 Lower bound

In [6] some of us used a search technique to find long run-rich words. The basic idea of this was to start with n run-rich words, augment each by adding a or b to its end giving $2n$ words. Then select the n most run-rich words from this collection and repeat the process. Using these techniques, we constructed a ternary word of order 9 and length 9686 containing 7728 runs. Concatenating this with itself infinitely often (which gives an extra 11 runs for each copy) produces an infinite order 9 word having run density $7739/9686 \approx 0.798988$. The word begins:

aabbccaabbccbbccaabbccaabbccbbccaabbccaacaabbccaabbccbbccaabbc
caabbccbbccaabbccaaccaabbccaabbccbbccaabbccaabbccbbccaabbccaac
aabbccaabbccbbccaabbccaabbccbbccaabbccaaccaabbccaabbccbbccaabb
ccaacaabbccaabbccbbccaabbccaabbccbbccaabbccaacaabbccaabbccbbcc
aabbccaabbccbbccaabbccaaccaabbccaabbccbbccaabbccaabbccbbccaabb
ccaacaabbccaabbccbbccaabbccaabbccbbccaabbccaaccaabbccaabbccbbc
caabbccaacaabbccaabbccbbccaabbccaabbccbbccaabbccaaccaabbccaabb
ccbbccaabbccaabbccbbccaabbccaacaabbccaabbccbbccaabbccaabbccbbc
caabbccaaccaabbccaabbccbbccaabbccaabbccbbccaabbccaacaabbccaabb
ccbbccaabbccaabbccbbccaabbccaaccaabbccaabbccbbccaabbccaacaabbc
caabbccbbccaabbccaabbccbbccaabbccaacaabbccaabbccbbccaabbccaabb
ccbbccaabbccaaccaabbccaabbccbbccaabbccaabbccbbccaabbccaacaabbc

Thus we have:

$$\lim_{n\to\infty} \frac{\rho(9, n)}{n} \geq 0.798988$$

3 Upper bound

To get an upper bound on $\rho(k, n)$, we used the techniques of [4] for various values of k. These techniques give an exact bound on the number of runs with period less than some bound p and then use a result of Crochemore and Ilie [1] which states that the number of runs with period greater than or equal to p in a word of length n is at most $6n/p$. Results for different values of k are given in the following table, which used $p = 9$. Concerning runs with period less than 9, even ternary words of order $k = 19$ do not achieve the 11/13 maximal run density for binary words [4].

Order k	Exact bound (period ≤ 9)	Upper bound on $\rho(k, n)/n$
3	0/1	0.6000
4	7/13	1.1385 *cabcabccabcabbcabcabbcabcaabcabcaabcabc*
5	7/11	1.2364 *abbccaabbca*
6	7/11	1.2364
7	2/3	1.2667 *aacaabbcaabbcbbcaabbc*
8	17/24	1.3083 *bbccaabaabbccaabbccbccaa*
9	17/24	1.3083
10	17/24	1.3083
11	17/24	1.3083
12	17/24	1.3083
13	17/24	1.3083
14	17/24	1.3083
15	3/4	1.3500 *acaaccaacaabaabbaabaabbcbbccbbcbbcca*
16	3/4	1.3500
17	3/4	1.3500
18	3/4	1.3500
19	25/33	1.3576
$+\infty$	11/13	1.4462 *aababbabaabab*

Table 1. Upper bounds on $\rho(k, n)$ for various orders k. The second column gives the exact bound on the number of such runs with period at most 9. The third column adds this to 6/(9+1), the bound on the run density of runs with period greater than 9, to give the required upper bound. The last column shows examples of strings, that, concatenated with themselves infinitely often, give the exact bound.

For example, we have:

$$\lim_{n\to\infty} \frac{\rho(9, n)}{n} \leq 1.3083$$

Our results are not strong enough to be more than suggestive. To get more convincing evidence for the superiority of binary words, we would need to get the upper bound on $\rho(k, n)$ to be less than the lower bound on $\rho(n)$ given in (1). Perhaps this can be done using the more powerful techniques of Crochemore and Ilie.

4 Other remarks

We mention another condition on run-rich words which is suggested by experimental evidence. This is that run-rich words do not need to contain cubes, in particular the cubes *aaa* and *bbb*. This is not necessary since, for example, the word *aaa* contains the maximum possible number of runs for a 3 letter word, but we could use instead the word *aab* which has the same number of runs but no cube.

We also note the following observation which is in the other direction to our earlier results. Let x be a string of size n on the binary alphabet $\{a, b\}$. Let $\bar{x}$ be the string obtained by rewriting x with the as replaced by bs and the bs by as. The words x and $\bar{x}$ have the same number of runs.

For a binary word y of length p, consider the word $\tau(y)$ of size $(n+1)p$ obtained by the rewriting $\tau(a) = cx$ and $\tau(b) = c\bar{x}$. The word $\tau(y)$ is of ternary order $n+2$. It can be shown that no run is lost with such rewriting, thus

$$r(\tau(y)) \geq r(y) + p \cdot r(x)$$

If we select x and y to be run-maximal, that is $r(x) = r(\bar{x}) = \rho(n)$ and $r(y) = \rho(p)$, then we have

$$\rho(n+2, (n+1)p) \geq r(\tau(y)) \geq \rho(p) + p \cdot \rho(n)$$

thus

$$\lim_{n\to\infty} \frac{\rho(n+2,(n+1)p)}{(n+1)p} \geq \lim_{n\to\infty} \frac{\rho(n)}{n+1} = \lim_{n\to\infty} \frac{\rho(n)}{n}.$$

References

1. M. Crochemore and L. Ilie: *Maximal repetitions in strings.* J. Comput. Systems Sci., 74(5) 2008, pp. 796–807.
2. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the runs conjecture*, in Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008), P. Ferragina and G. M. Landau, eds., vol. 5029 of LNCS, 2008, pp. 290–302, See also `http://www.csd.uwo.ca/~ilie/runs.html`.
3. F. Franek: `http://www.cas.mcmaster.ca/~franek/research.html`.
4. M. Giraud: *Asymptotic behavior of the numbers of runs and microruns.* Information and Computation, 207(11) 2009, pp. 1221–1228.
5. R. Kolpakov and G. Kucherov: *On maximal repetitions in words.* Journal of Discrete Algorithms, 1(1) 2000, pp. 159–186.
6. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proceedings of the Prague Stringology Conference 2008 (PSC 2008), J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 140–145.
7. S. J. Puglisi and J. Simpson: *The expected number of runs in a word.* Australasian J. Combinatorics, 42 2008, pp. 45–54.
8. J. Simpson: *Modified Padovan words and the maximum number of runs in a word.* Australasian J. Combinatorics, 46 2010, pp. 129–146.

Formal Characterizations of FA-based String Processors

Ernest Ketcha Ngassam[1,2,⋆], Bruce W. Watson[3], and Derrick G. Kourie[3]

[1] SAP Meraka UTD, Pretoria, South Africa
[2] School of Computing University of South Africa
Pretoria 0001
ernest.ngassam@sap.com
[3] Department of Computer Science, University of Pretoria
South Africa, Pretoria 0002
{dkourie,bwatson}@cs.up.ac.za

Abstract. We rely on the denotational semantics of algorithms to suggest an abstraction of a string recognizer. The abstraction provides a unified formalism for representing FA-based string recognizers as an instance of a parameterized function. It also forms the basis for theoretical investigations on implementations of FA-based recognizers, and represents a framework for the identification of new algorithms for further studies.

1 Preliminaries and Characterization

An acceptor (or a string recognizer) of a finite automaton is an algorithm that relies on the finite automaton's transition function in order to determine whether a string is part of the language modeled by the FA or not. An acceptor of the automaton $\mathrm{M} = (\mathcal{Q}, \mathcal{V}, \Delta, \mathcal{S}, \mathcal{F})$, where: $\mathcal{V}$ and Δ are the alphabet and transition function respectively; $\mathcal{L}(\mathrm{M}) \subseteq \mathcal{V}^*$ is the language of M; and $\mathcal{P}(X)$ is used to represent the power set of a set X, can be characterized by the following function:
$\rho : \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q}) \times \mathcal{V}^* \nrightarrow \mathbb{B} = \{true, false\}$ such that for $\Delta \in \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q})$, $\rho(\Delta, s) = true$ if $s \in \mathcal{L}(\mathrm{M})$ or $\rho(\Delta, s) = false$ if $s \notin \mathcal{L}(\mathrm{M})$.
In fact, ρ is the *denotational semantics* of the acceptor [4] and is a partial function, hence the mapping using the relation $\nrightarrow$. The denotational semantics indicates the "meaning" of the algorithm in functional terms, but hides details about how the algorithm that performs acceptance testing should actually work. At this level of description, the acceptor is viewed as a "black box" that receives as input a transition set and a string, and later produces a boolean as output. There are, in fact, a large number of ways in which this processing can take place, as extensively discussed and implemented in [5]. In fact, three core algorithms were discussed; namely the table-driven (TD), hardcoded (HC), and a hybrid version of the two referred to as mixed-mode (MM). Furthermore, a range of implementation strategies were investigated and implemented with their performance analyzed. Those strategies intended to optimize cache memory usage in order to improve the performance of the recognizer. The strategies discussed were Dynamic State Allocation (DSA), Allocated Virtual Caching (AVC) and State pre-Ordering (SpO). It was proven that by implementing each strategy or their combination based on any given core algorithm, the performance of the recognizer would improve [5].

In order to refine the generic definition of ρ above, let Δ_t denotes the transition set that is used in the TD part of an MM implementation. Similarly, let Δ_h be the

⋆ Supported in part by the South African National Research Foundation (NRF).

Ernest Ketcha Ngassam, Bruce W. Watson, Derrick G. Kourie: Formal Characterizations of FA-based String Processors, pp. 183–185 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8

transition set that is used in the TD part. Clearly, Δ_t and Δ_h must be a partition of the original transition set, Δ, i.e. $\Delta_t \cup \Delta_h = \Delta$ and $\Delta_t \cap \Delta_h = \emptyset$. Now let $\rho_{_C}$ be the function to represent the denotational semantics of a recognizer based on one of the core algorithms, TD, HC or MM. Letting $\mathcal{T} = \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q})$, this function can be defined as follows: $\rho_{_C} : \mathcal{T} \times \mathcal{T} \times \mathcal{V}^* \nrightarrow \mathbb{B}$ such that $\rho_{_C}(\Delta_t, \Delta_h, s) = \rho(\Delta, s)$.

In order to obtain a generic formalism that takes into consideration the various foregoing implementation strategies, appropriate parameters are necessary. In fact, DSA and AVC would each require two variables D_t and D_h (resp. V_t and V_h) which are natural numbers that reflect the extent to which the TD part (resp. HC part) of the recognizer will be table-driven (resp. hardcoded). Similarly, for the SpO strategy, two boolean parameters P_t and P_h are required. They indicate whether the TD part (resp. the HC part) of the algorithm requires pre-ordering.

By putting all the strategies together, we may now characterize a recognizer as a new function, $\rho_{_{CDPV}}$, that is parameterised by: its transition sets Δ_t and Δ_h; all its strategies (D_t, D_h, P_t, P_h, V_t and V_h); and its input string s. Therefore, once the transitions sets have been specified, given an arbitrary input string, one may choose to implement $\rho_{_{CDPV}}$ using any of the strategies or some combination of them, as long as the necessary conditions on strategies are respected. A recognizer's denotational semantics is now formally expressed in general as follows:
$\rho_{_{CDPV}} : \mathcal{T} \times \mathcal{T} \times \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B}$ such that

$$\text{if } \begin{cases} (\Delta_t \cup \Delta_h = \Delta) \wedge (\Delta_t \cap \Delta_h = \emptyset) \\ (0 \leq D_t \leq |\mathcal{Q}_t|) \wedge (0 \leq D_h \leq |\mathcal{Q}_h|) \\ (P_t \in \mathbb{B}) \wedge (P_h \in \mathbb{B}) \\ (0 \leq V_t < |\mathcal{Q}_t|) \wedge (0 \leq V_h < |\mathcal{Q}_h|) \end{cases}$$

$$\text{then } \rho_{_{CDPV}}(\Delta_t, \Delta_h D_t, D_h, P_t, P_h, V_t, V_h, s) = \rho(\Delta, s)$$

The recognizer defined as such shows that, the strategies used depend on the nature of the transition set itself. Arguments subscripted with $_t$ are associated to the TD algorithm, whereas those subscripted with $_h$ are associated to the HC algorithm. The high-level formalisms associated with each type of algorithm may be used in obtaining of new algorithms using appropriate instances of their associated strategy variables.

2 Conclusion and Future Work

This paper suggested a formal characterization of FA-based string processor taking into consideration a range of strategies that could be explored for leveraging the performance of the recognizer at run-time. Such characterization could be explored theoretically in order to determine appropriate properties of each strategic variables employed. The suggested formalism relied only on investigations conducted in [5] for cache optimization strategies. As a matter of future work, we consider exploring other aspects such computing platform (OS) and appropriate computational medium. It also worth mentioning that many of the algorithms formalized in this paper based on implementation strategies have proven to be more efficient that their core counterparts. Work on further investigation on other algorithms is still ongoing and results presenting their performance can be found in [1,2,3,5].

References

1. E. N. Ketcha, D. G. Kourie, and B. W. Watson: *Reordering finite automatata states for fast string recognition*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 2005, Czech Technical University in Prague.
2. E. N. Ketcha, D. G. Kourie, and B. W. Watson: *A taxonomy of DFA-based string processors*, in Proceedings of the SAICSIT Conference, Gordon's Bay, South Africa, October 2006, ACM, pp. 111–121.
3. E. N. Ketcha, B. W. Watson, and D. G. Kourie: *On implementations and performances of table-driven FA-based string processors*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 2006, Czech Technical University in Prague.
4. B. Meyer: *Introduction to the Theory of Programming Languages*, Prentice Hall, C.A.R Hoare series ed., 1990.
5. E. K. Ngassam: *Towards Cache Optimization in Finite Automata Implementations*, PhD thesis, Department of Computer Science, Pretoria, South Africa, November 2006.

Author Index