Proceedings of the Prague Stringology Conference 2014

Edited by Jan Holub and Jan Žďárek

September 2014

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

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

Organising Committee

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

External Referees

Jérémy Barbay	Juan Mendivelso	Elise Prieur-Gaston
Loek Cleophas	Yuto Nakashima	Ayumi Shinohara
Arnaud Lefebvre		

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2014 (PSC 2014) at the Czech Technical University in Prague, which organizes the event. The conference was held on September 1–3, 2014 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. Eighteen papers were selected, based on originality and quality, as regular papers for presentations at the conference. This volume contains not only these selected papers but also an abstract of one invited talk "On the Number of Distinct Squares".

The Prague Stringology Conference has a long tradition. PSC 2014 is the eighteenth 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, 2008–2013 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions were published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, the International Journal of Foundations of Computer Science, and the Discrete Applied Mathematics.

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

We would like to thank all those who had submitted papers for PSC 2014 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 2014. Last, but not least, our thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on September 2014

Jan Holub and Thierry Lecroq

Table of Contents

Invited Talk

Contributed Talks

On the Number of Distinct Squares
Abstract

Frantisek Franek

Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
franek@mcmaster.ca

Abstract. Counting the number of types of squares rather than their occurrences, we consider the problem of bounding the maximum number of distinct squares in a string. Fraenkel and Simpson showed in 1998 that a string of length n contains at most $2n$ distinct squares and indicated that all evidence pointed to n being a natural universal upper bound. Ilie simplified the proof of Fraenkel-Simpson's key lemma in 2005 and presented in 2007 an asymptotic upper bound of $2n\Theta(\log\ n)$. We show that a string of length n contains at most $\lfloor 11n/6 \rfloor$ distinct squares for any n. This new universal upper bound is obtained by investigating the combinatorial structure of FS-double squares (named so in honour of Fraenkel and Simpson's pioneering work on the problem), i.e. two rightmost-occurring squares that start at the same position, and showing that a string of length n contains at most $\lfloor 5n/6 \rfloor$ FS-double squares. We will also discuss a much more general approach to double-squares, i.e. two squares starting at the same position and satisfying certain size conditions. A complete, so-called canonical factorization of double-squares that was motivated by the work on the number of distinct squares is presented in a separate contributed talk at this conference. The work on the problem of the number of distinct squares is a joint effort with Antoine Deza and Adrien Thierry.

At the time of the presentation of this talk, the slides of the talk are also available at
`http://www.cas.mcmaster.ca/~franek/PSC2014/invited-talk-slides.pdf`

This work was supported by the Natural Sciences and Engineering Research Council of Canada

References

1. M. Crochemore and W. Rytter: *Squares, cubes, and time-space efficient string searching.* Algorithmica, 13:405–425, 1995.
2. A. Deza and F. Franek: *A d-step approach to the maximum number of distinct squares and runs in strings.* Discrete Applied Mathematics, 163:268–274, 2014.
3. A. Deza, F. Franek, and M Jiang: *A computational framework for determining square-maximal strings.* In J. Holub and J. Žďárek, editors, Proceedings of the Prague Stringology Conference 2012, pp. 111–119, Czech Technical University in Prague, Czech Republic, 2012.
4. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* Journal of Combinatorial Theory, Series A, 82(1):112–120, 1998.
5. F. Franek, R. C. G. Fuller, J. Simpson, and W. F. Smyth: *More results on overlapping squares.* Journal of Discrete Algorithms, 17:2–8, 2012.
6. L. Ilie: *A simple proof that a word of length n has at most 2n distinct squares.* Journal of Combinatorial Theory, Series A, 112(1):163–163, 2005.
7. L. Ilie: *A note on the number of squares in a word.* Theoretical Computer Science, 380(3):373–376, 2007.
8. E. Kopylova and W. F. Smyth: *The three squares lemma revisited.* Journal of Discrete Algorithms, 11:3–14, 2012.

Frantisek Franek: On the Number of Distinct Squares, pp. 1–2.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

9. M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń: *On the maximum number of cubic subwords in a word.* European Journal of Combinatorics, 34:27–37, 2013.
10. N. H. Lam: *On the number of squares in a string.* AdvOL-Report 2013/2, McMaster University 2013.
11. M. J. Liu: *Combinatorial optimization approaches to discrete problems.* Ph.D. thesis, Department of Computing and Software, McMaster University 2013.

Fast Regular Expression Matching Based On Dual Glushkov NFA

Ryutaro Kurai[1,2], Norihito Yasuda[1], Hiroki Arimura[2], Shinobu Nagayama[3], and Shin-ichi Minato[1,2]

[1] JST ERATO MINATO Discrete Structure Manipulation System Project
060-0814 Sapporo, Japan
{kurai, yasuda, minato}@erato.ist.hokudai.ac.jp
[2] Graduate School of Information Science and Technology
Hokkaido University, 060-0814 Sapporo, Japan
arim@ist.hokudai.ac.jp
[3] Department of Computer and Network Engineering
Hiroshima City University, 731-3194 Hiroshima, Japan
s_naga@hiroshima-cu.ac.jp

Abstract. This paper presents a new regular expression matching method by using Dual Glushkov NFA. Dual Glushkov NFA is the variant of Glushkov NFA, and it has the strong property that all the outgoing transitions to a state of it have the same labels. We propose the new matching method Look Ahead Matching that suited to Dual Glushkov NFA structure. This method executes NFA simulation with reading two input symbols at the one time. We use information of next symbol to narrow down the active states on NFA simulation. It costs additional working memory to apply Look Ahead Matching to ordinal Thompson NFA. However, we can use this method with no additional memory space if use it with Dual Glushkov NFA. Experiments also indicate that the combination of Dual Glushkov NFA with Look Ahead Matching outperforms the other methods on NFAs converted from practical regular expressions.

Keywords: regular expression matching, non-deterministic finite automata, ε-transition removal, Thompson NFA, Glushkov NFA

1 Introduction

1.1 Background

Regular expression matching is one of the fundamental research topics in computer science [13], since it plays such important role in emerging applications in large-scale information processing fields, such as: Network Intrusion Detection System (NIDS), Bioinformatics search engines, linguistic vocabulary, and pattern matching in cloud computing [10, 12, 15].

1.2 Problems with previous approaches

For regular expression matching, there are three well-known approaches: *backtracking*, *DFA*, and *NFA*. Among them, backtracking is the most widely used in practical applications. However, this approach is so slow if it manipulates some difficult patterns and texts, like $a?^n a^n$ as pattern and a^n as text, which triggers many backtracking on the input text [4]. The deterministic finite automaton (DFA) approach is extremely fast if the input regular expression can be compiled into a DFA of small size, but it is not practical if a given regular expression causes the exponential explosion of the number of states.

Ryutaro Kurai, Norihito Yasuda, Hiroki Arimura, Shinobu Nagayama, Shin-ichi Minato: Fast Regular Expression Matching Based On Dual Glushkov NFA, pp. 3–16.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

The Nondeterministic Finite Automaton (NFA) approach can avoid such explosion in the number of states, and is shown to be faster than the naive backtrack approach for the case that the backtracking approach suffer from many near-misses. Unfortunately, NFA approach is not so fast in practice. One of the major reasons is the cost of maintaining a set of active states; every time next input symbol comes, NFA has to update the all the active states to the next states or to just discard them. If the number of active states becomes large, the updating cost will also increase.

We can further classify the NFA into three types; Thompson NFA, Glushkov NFA and Dual Glushkov NFA. The most popular one is Thompson NFA, which is easy to construct, and its number of transitions and states are constant multiple of associated regular expression's length. Thompson NFA includes many of epsilon transitions. Those transitions make many of active states when we simulate such NFA.

Glushkov NFA is the other popular NFA. It has strong property that it has no epsilon transition and its all the incoming transitions to a state of Glushkov NFA have the same labels. Dual Glushkov NFA is a variant of Glushkov NFA, but has a special feature that is worth our attention. In an opposed manner of the Glushkov NFA, all the outgoing transitions from a state of Dual Glushkov NFA have the same labels.

1.3 Speed-up methods

We can simulate Glushkov NFA faster than Thompson NFA because of its property that it has no epsilon transition. The property causes less active states. Nevertheless, we have to manipulate amount of active state, and it slows down matching speed, if we treat complex regular expression. To cope with this problem, we propose a new method Look Ahead Matching.

That is the new matching method that reads two symbols of the input text at one time. We call the first of the two symbols the "current symbol", and second one the "next symbol". Ordinary matching methods read only current symbol, and calculate NFA active states from current active states and the symbol. Our method uses the next symbol to narrow down the active state size. We set states active only if the states have incoming transition labeled by first symbol and outgoing transition labeled by second symbol. However, fast matching by two input symbols creates large memory demands if we use the Glushkov NFA. To treat this problem, we employ a Dual Glushkov NFA. The structure of Dual Glushkov NFA is similar to that of Glushkov NFA, but is better suited to building an index of transitions for look ahead matching. We have to make transitions table for combination of two symbols to enable this new matching method. We can generate such table without additional space if we use the above Dual Glushkov NFA's property. Therefore, we propose the new look ahead matching method by using Dual Glushkov NFA.

1.4 Main Results

In this paper we propose a new matching method created by combining Dual Glushkov NFA and look ahead matching. Then we compare our proposal against other methods such as original Thompson NFA, Glushkov NFA, and a combination of Glushkov NFA and look ahead matching. For reference, we also compare our method with NR-grep [11]. In most cases our method is faster than Thompson NFA or Glushkov NFA.

Our method is only slightly slower than the combination of Glushkov NFA and look ahead matching, but it uses far less memory.

1.5 Related Works

Many regular expression matching engines use the backtracking approach. They traverse the syntax tree of a regular expression, and backtrack if they fail to find a match. Backtracking has a small memory footprint and high throughput for small and simple regular expressions. However, in worst case, it takes exponential time in the size of the regular expression [4].

Another approach, compiling regular expression has been used from the 1970s [1]. Such an algorithm converts a regular expression into an NFA, which is then converted into a DFA. This intermediate NFA (called Thompson NFA) has linear size memory in the length of the input regular expression. However, the final subset-constructed DFA takes exponential space in the size of the NFA and overflows the main memory of even recent computers.

In recent years, NFA evaluation for regular expression matching has been attracting much attention. Calculations performed on GPU, FPGA, or some special hardware cannot use abundant memory, but their calculation speed is much faster and concurrency is larger than typical computers. Therefore, using just the basic approach is adopted in some fields [3,7]. Cox proposed the NFA based regular expression library RE2 [14]. For fast evaluation, it caches a DFA generated from an NFA on the fly. RE2 seems to be the NFA based library that is being used widely; it has guaranteed computation time due to the NFA-oriented execution model.

Berry and Sethi indeed popularized Glushkov NFA [2]. Watson has finely researched Glushkov NFA and its application. He also showed the relationship between Thompson NFA and Glushkov NFA [16–18].

1.6 Organization

Sec. 2 briefly introduces regular expression matching, non-deterministic automata, and their evaluation. Sec. 3 presents our methods including preprocessing and runtime methods. Sec. 4 shows the results of computer experiments on NFA evaluation. Sec. 5 concludes this paper.

2 Preliminaries

2.1 Regular Expression

In this study, we consider regular expressions as follows. This definition is from [9].

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ε is a regular expression and denotes the set ε.
3. For each symbol a in Σ is a regular expression and denotes the set a.
4. If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$, respectively then $(r|s)$, (rs), and $(r*)$ are regular expressions that denote the sets $L(r) \cup L(s)$, $L(r)L(s)$ and $L(r)*$, respectively.

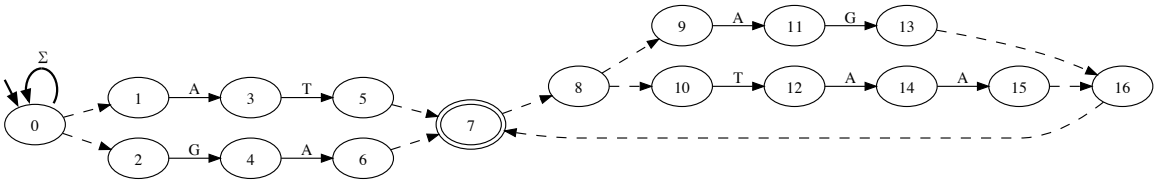


Figure 1. T-NFA for $R = (AT|GA)((AG|TAA)^*)$

2.2 Thompson NFA

The NFA constructed by Thompson's algorithm for regular expression R is called the *Thompson NFA* (or *T-NFA*, for short). It precisely handles the language statement $\Sigma^* L(R)$, which represents the substring match of R against a substring of a text.

Formally, T-NFA for R is a 5-tuple $N_R = (V, \Sigma, E, I, F)$, where V is a set of *states*, Σ is an *alphabet* of symbols, $E \subseteq V \times \Sigma \cup \{\varepsilon\} \times V$ is a set of symbol- and ε-transitions, called the *transition relation*, I and $F \subseteq V$ are the sets of *initial* and *final states* respectively. Each transition e in E is called a *symbol-transition* if its label is a symbol c in Σ, and an *ε-transition* if the label is ε. Each transition is described as $(s, char, t) \in E$, in this expression, s and t mean source state and target state. *char* is a label of the transition.

The T-NFA N_R for R has nested structure associated with the syntax tree for R.

Let the length of associated regular expression be m. This length means the number of all symbols that appeared in the associated regular expression. The number includes the special symbols like "*", "(", or "+". For instance, the length of a regular expression "abb*" is 4.

It has at most $2m$ states, and every state has in-degree and out-degree of two or less. Specifically, state s in V can have at most one symbol-transition or two ε-transitions from s. We show an example of T-NFA when $R = (AT|GA)((AG|TAA)^*)$ in Fig. 1.

2.3 Glushkov NFA

Another popular method of constructing an NFA from a regular expression is Glushkov's algorithm [2]. We call the automaton constructed by this method Glushkov NFA (also known as Position Automata); abbreviated here to G-NFA. However, G-NFA can also be converted from T-NFA by using the algorithm in Fig. 4 (Watson showed in [18]). This algorithm removes the ε-transitions from T-NFA. For instance, we show the new transitions that skip ε-transition by the bold transitions in Fig. 2 and the fully converted G-NFA from T-NFA in Fig. 3. Both examples show NFAs that precisely handle $R = (AT|GA)((AG|TAA)^*)$.

2.4 G-NFA Properties

G-NFA has some very interesting properties.

- It has no ε-transitions. We call this property ε-free.
- For any state, the state's incoming transitions are labeled by the same symbol.
- It has only one initial state.
- It has one or more final states.

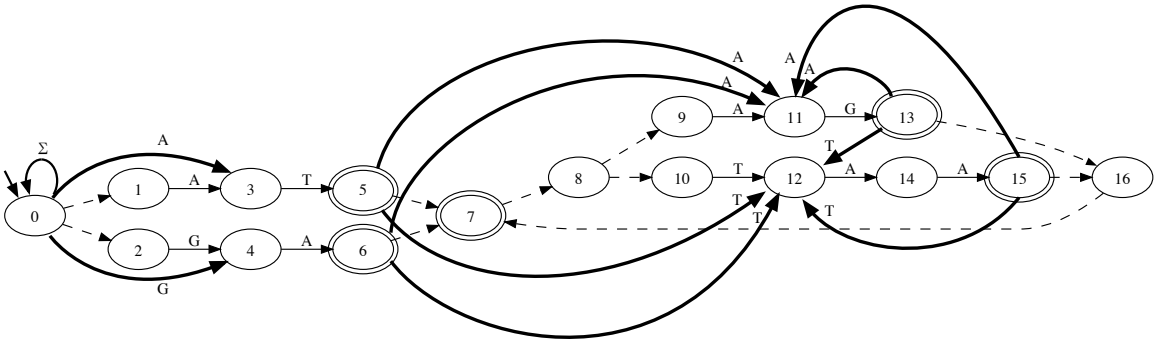


Figure 2. T-NFA and skip transitions of ε-transitions

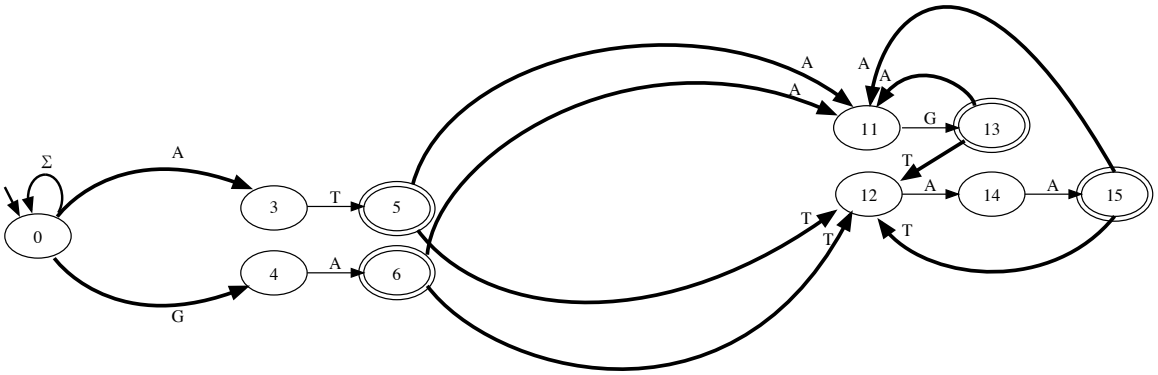


Figure 3. G-NFA for $R = (AT|GA)((AG|TAA)^*)$

- Its number of states is $\tilde{m}+1$. $\tilde{m}$ is the length of the associated regular expression, but the number excludes the special symbols like "$*$", "(", or "+".
- The number of transitions is $\tilde{m}^2$ at worst.

2.5 Dual Glushkov NFA

As a variation of G-NFA, Dual Glushkov NFA is known [16]. We call it Dual G-NFA for short. The algorithm that converts T-NFA into Dual G-NFA (Fig. 5) is similar to the algorithm that converts T-NFA into G-NFA (fig. 4). The base algorithm of Fig. 5 was also shown by Watson [18].

When we convert T-NFA into G-NFA, we generate a skip transition as follows. First, we search the path that started by epsilon-path and ended only one symbol-transition. Then we create skip transition from the start state to the end state for each such path. The label of new skip transition is taken from the last transition of the path.

When we convert T-NFA into Dual G-NFA, we generate a skip transition as follows. First, we search the path that started only one symbol-transition and ended epsilon-path. Then we create skip transition from the start state to the end state for each such path. The label of new skip transition is taken from the first transition of the path.

In addition, an original T-NFA has same number of outgoing and incoming transitions for all section of T-NFA. In fact, Conjunction, Concatenation, and Kleene

```
procedure BuildG-NFA(N = (V, Σ, E, I, F))
    V' ← ∅, E' ← ∅, F' ← F
    GlushkovState ← V \ ⋃_{s', ∃s∈V, (s,ε,s')∈E} s'
    for all s ∈ GlushkovState do
        for s' ∈ Eclose(s) do
            if s' ∈ F then
                F' ← F' ∪ {s'}
            end if
            for (s', char, t) ∈ E do
                if char ≠ ε then
                    E' ← E' ∪ {(s, char, t)}
                end if
            end for
        end for
    end for
    return (V', Σ, E', I, F')
end procedure
procedure Eclose(s ∈ E)
    Closure ← {s}
    for (s, char, t) ∈ E do
        if char = ε then
            Closure ← Closure ∪ Eclose(t)
        end if
    end for
    return Closure
end procedure
```

Figure 4. Algorithm Converting T-NFA to G-NFA

```
procedure BuildDualG-NFA(N = (V, Σ, E, I, F))
    V' ← ∅, E' ← ∅, I' ← ∅
    DualGlushkovState ← V \ ⋃_{s', ∃s∈V, (s',ε,s)∈E} {s'}
    for s ∈ Eclose(I) do
        if s ∈ DualGlushkovState then
            I' ← I' ∪ {s}
        end if
    end for
    for all s ∈ DualGlushkovState do
        for (s, char, t) ∈ E do
            for t' ∈ Eclose(t) do
                if t ≠ t' and char ≠ ε and t' ∈ DualGlushkovState then
                    E' ← E' ∪ {(s, char, t')}
                end if
            end for
            E' ← E' ∪ {(s, char, t)}
        end for
    end for
    return (V', Σ, E', I, F')
end procedure
```

Figure 5. Algorithm Converting T-NFA to Dual G-NFA

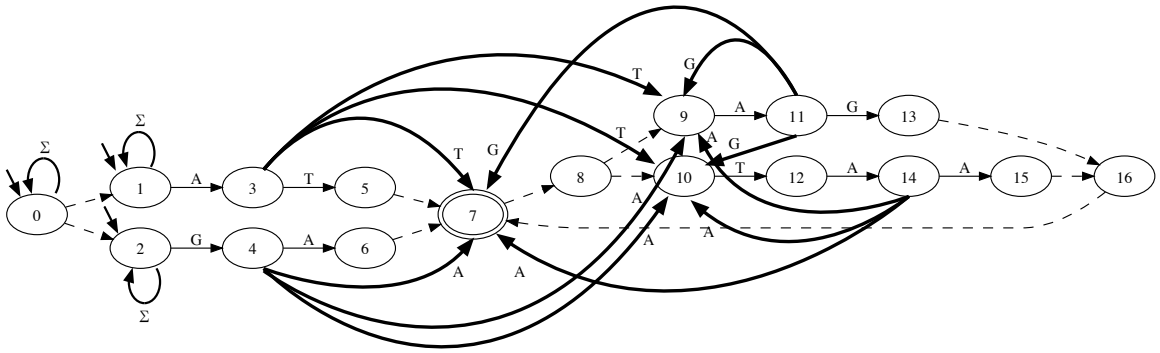


Figure 6. T-NFA to Dual G-NFA

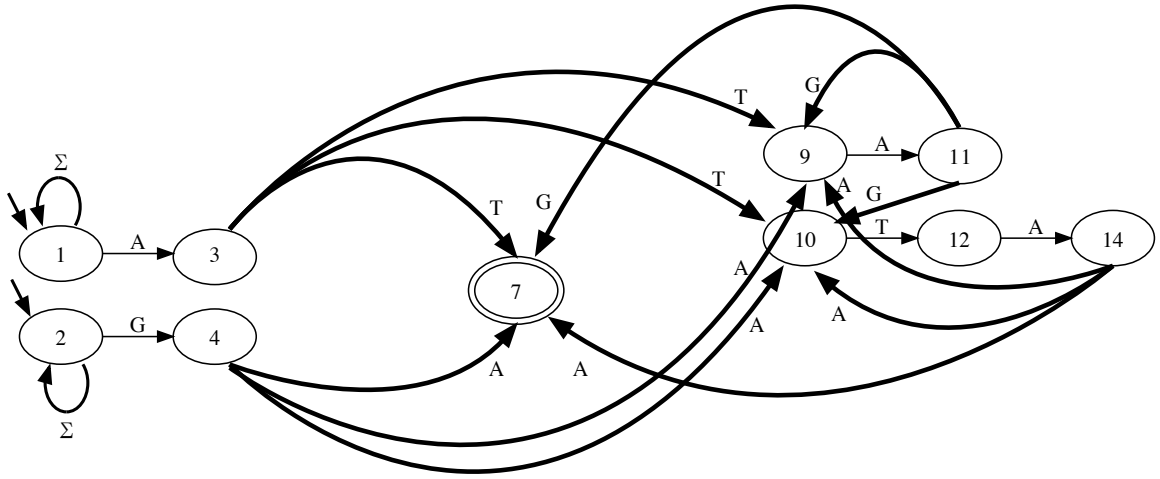


Figure 7. Dual G-NFA for $R = (AT|GA)((AG|TAA)^*)$

Closure section of T-NFA have same in-degree and out-degree. Because of this T-NFA's property, we can consider that above "Dual G-NFA" is dual of "G-NFA".

For instance, we show the new transitions that skip ε-transition by bold transitions in Fig. 6, and the fully converted Dual G-NFA from T-NFA in Fig. 7. Both examples show NFAs that precisely handle $R = (AT|GA)((AG|TAA)^*)$.

2.6 Dual G-NFA Properties

Dual G-NFA has properties similar to those of G-NFA.

- It is ε-free.
- For any state, the state's outgoing transitions are labeled by the same symbol.
- It has only one final state.
- It has one or more initial states.
- Its number of states is $\tilde{m} + 1$.
- The number of transitions is $\tilde{m}^2$ at worst.

There is a duality between G-NFA and Dual G-NFA in the sense of the properties of initial states, final states, and labels of transitions.

```
procedure G-NFACountMatching(N = (V, Σ, E, I, F), T = t1t2t3...tn)
    CurrentActive ← ∅
    NextActive ← ∅
    MatchCount ← 0
    Index ← BuildIndex(E)
    for pos ∈ 1, ..., n do
        CurrentActive ← CurrentActive ∪ I
        for s ∈ CurrentActive do
            NextActive ← NextActive ∪ Index[t_pos][s]
        end for
        if NextActive ∩ F ≠ ∅ then
            MatchCount ← MatchCount + 1
        end if
        CurrentActive ← NextActive
        NextActive ← ∅
    end for
    return MatchCount
end procedure
procedure BuildIndex(V, Σ, E)
    for s ∈ V do
        for char ∈ Σ do
            Index[char][s] = ∅
        end for
    end for
    for (s, char, t) ∈ E do
        Index[char][s] = Index[char][s] ∪ {t}
    end for
    return Index
end procedure
```

Figure 8. Regular Expression Matching Using NFA

2.7 Regular Expression Matching Method

For both G-NFA and Dual G-NFA, ε-free NFAs have the same simulation algorithm like that of Fig. 8. The basic idea of this algorithm was also shown by Watson [17].

This algorithm reads input symbol t_i one by one, then searches for a state that has outgoing transition labeled t_i from current active state set (*CurrentActive* in Fig. 8). For fast search we use the index created by *BuildIndex*. If such states are found, we add a transitive state to next state set (*NextActive* in Fig. 8). At the end of a step, we check if the *NextActive* includes a final state. If a final state is found, we recognize that the input symbols match a given regular expression.

3 Our Method

3.1 Look ahead matching

The above NFA simulation method reads input symbols one by one, and calculates state transitions. However, it is quite easy to read a next input symbol. We consider how to more effectively calculate state transitions. Let the current input symbol be t_i, next input symbol t_{i+1}. When we know t_{i+1}, we want to treat the states that satisfy the next formula as active states.

$$LookAheadActive(s, t_i, t_{i+1}) = \{s' : (s, t_i, s') \in E, (s', t_{i+1}, s'') \in E\}$$

And we formally define normal active states as follows.

$$Active(s, t_i) = \{s' : (s, t_i, s') \in E\}$$

For any $LookAheadActive(s, t_i, t_{i+1})$, the size of $LookAheadActive(s, t_i, t_{i+1})$ is equal or less than the size of $Active(s, t_i)$. Because of this difference in size of active states, we consider that look ahead matching can calculate transitions faster than normal matching. We formally show this algorithm in Fig. 9. This look ahead mathing idea have been used in some studies [5, 6].

The problem of this matching method is the large size of the state transition table associated with t_i and t_{i+1}. The state transition table has duplicate transitions and costs $O(|E|^2)$ space to build from G-NFA.

For example, we show the transition table of Fig. 1 as Table 1. This table has 19 records, more than the number of original G-NFA's transitions. The difference is due to the duplication of transitions.

id	t_i	t_{i+1}	source state	target state
1	T	A	3	5
2	T	A	5	12
3	T	A	6	12
4	T	A	13	12
5	T	A	15	12
6	T	T	3	5
7	A	A	4	6
8	A	A	12	14
9	A	A	14	15
10	A	T	4	6
11	A	T	14	15
12	A	T	0	3
13	G	A	11	13
14	G	A	0	4
15	G	T	11	13
16	A	G	5	11
17	A	G	6	11
18	A	G	13	11
19	A	G	15	11

Table 1. Look Ahead Transition Table for G-NFA and Dual G-NFA

id	t_i	t_{i+1}	source state	target state
1	A	T	1	3
2	A	T	4	10
3	A	T	14	10
4	G	A	2	4
5	G	A	11	9
6	A	G	9	11
7	T	A	10	12
8	T	A	3	9
9	A	A	12	14
10	A	A	4	9
11	A	A	14	9
12	T	*	3	7
13	T	T	3	10
14	A	*	4	7
15	A	*	14	7
16	G	*	11	7
17	G	T	11	10

Table 2. Look Ahead Transition Table for Dual G-NFA

3.2 Dual G-NFA Look Ahead Transition Function

As shown in the above section, Dual G-NFA has the very desirable property that all outgoing transition of a state have the same label. Because of this property, when the source state and t_i are given, the pairs of t_{i+1} and the target state are determined uniquely. Therefore, the transition tables size is $O(|E|)$. This is effectively smaller than G-NFA's size of $O(|E|^2)$.

For instance, we show the transition table of Fig. 7 in Table 2. This table has 17 records, equaling the number of original Dual G-NFA's transitions. A final state of Dual G-NFA has no outgoing transition, so we show the "$*$" on t_{i+1} column for the transitions that go to final state.

```
procedure DualG-NFACountLookAheadMatching(N = (V, Σ, E, I, F), T = t_1 t_2 t_3 ... t_n)
    CurrentActive ← ∅
    NextActive ← ∅
    MatchCount ← 0
    (Index, FinalIndex) ← BuildLookAheadIndex(E)
    for pos ∈ 1, ..., n − 1 do
        CurrentActive ← CurrentActive ∪ I
        for s ∈ CurrentActive do
            for (t ∈ Index[t_pos][t_pos+1][s]) do
                NextActive ← NextActive ∪ {t}
            end for
        end for
        for s ∈ CurrentActive do
            for (t ∈ FinalIndex[t_pos+1][s]) do
                NextActive ← NextActive ∪ {t}
            end for
        end for
        if NextActive ∩ F ≠ ∅ then
            MatchCount ← MatchCount + 1
        end if
        CurrentActive ← NextActive
        NextActive ← ∅
    end for
    return MatchCount
end procedure
procedure BuildLookAheadIndex(V, Σ, E, F)
    for s ∈ V do
        for char_1 ∈ Σ do
            for char_2 ∈ Σ do
                Index[char_1][char_2][s] = ∅
            end for
            FinalIndex[char_1][s] = ∅
        end for
    end for
    for (s, char_1, t) ∈ E do
        for (t, char_2, t') ∈ E do
            Index[char_1][char_2][s] = Index[char_1][char_2][s] ∪ {t}
        end for
        if t ∩ F ≠ ∅ then
            FinalIndex[char_1][s] = FinalIndex[char_1][s] ∪ {t}
        end if
    end for
    return (Index, FinalIndex)
end procedure
```

Figure 9. Look Ahead Regular Expression Matching Using Dual G-NFA

4 Experiments and Results

To confirm the efficiency of Dual G-NFA with Look Ahead matching (for short, Dual G-NFA with LA), we conducted three experiments. All experiments use "English.100MB" text in Pizza&Chili Corpus [8] as the input texts, and we compared our method with G-NFA, and G-NFA with Look ahead matching (for short G-NFA with LA). For reference, the results of a simple T-NFA implementation by Russ Cox [4], and NR-grep, a bit parallel implementation of G-NFA by Gonzalo Navarro are shown. All experiments were executed 10 times and the average time is shown.

The first experiment examines fixed string patterns. In this experiment, patterns were generated as follows. n fixed strings were randomly chosen from a fixed strings dictionary and then patterns were joined by conjunction symbol "|". We used `/usr/share/dict/words` file on Mac OS X 10.9.2 as the fixed strings dictionary. None of patterns included special symbols of regular expressions like "$*$", "?", or "+". Thus, the Aho-Corasick algorithm is clearly the most suited method for this problem. However, to measure trends of our methods, we make this experiment.

Table 3 shows the time (in seconds) needed to convert regular expression to NFAs. From Table 3, the converting time is so shorter than matching time. T-NFA (by Cox) was so fast to measure the converting time accurately (It was under micro seconds).

n	G-NFA	Dual G-NFA
20	immeasurable	immeasurable
40	0.01	0.01
60	0.01	0.01
80	0.02	0.02
100	0.02	0.02
120	0.03	0.03
140	0.04	0.04
160	0.05	0.05
180	0.06	0.06
200	0.07	0.07

Table 3. Needed time converting regular expression to NFAs in seconds

Fig. 10 shows the time (in seconds) needed to match with the whole text of "English.100MB". From Fig. 10, the time taken linearly increases with the number of patterns with T-NFA, G-NFA or Dual G-NFA. In contrast, G-NFA with LA or Dual G-NFA with LA, which uses look ahead matching method took almost constant time regardless of n. We assume this is because look ahead matching kept the active state size small.

NR-grep could not treat large regular expressions, so we only measured patterns for $n = 20$, 40, and 60;

Fig. 11 shows the average active state size, total number of active states divided by the number of all input symbols. As the graph shows, there is strong correlation between the average active state size and matching time.

In the second experiment, we generated patterns as follows. We inserted special symbols of regular expressions such as "$*$","?", or "+" into the patterns used in the first experiment. Insert positions were randomly selected excluding the first and last pattern positions. We then joined these generated regular expression patterns by conjunction. In this case, the Aho-Corasick algorithm is clearly the most suited

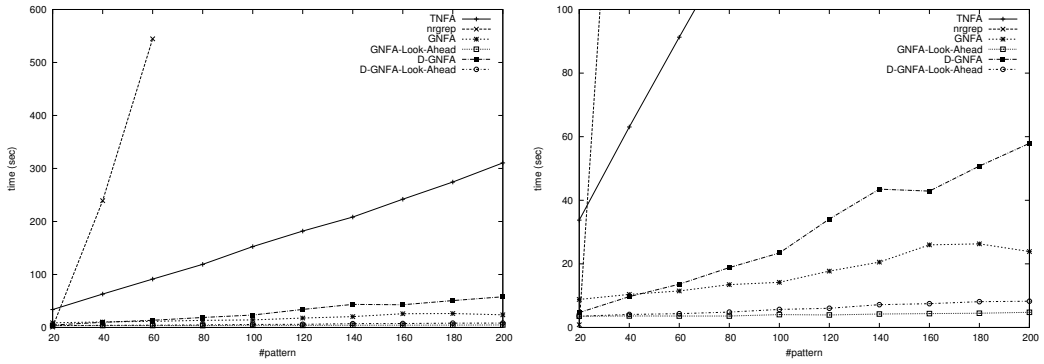


Figure 10. Needed time (sec) to matching whole text of "English.100MB" for 1st experiment. Right part is the part of left part.(Right pert is scaled-up)

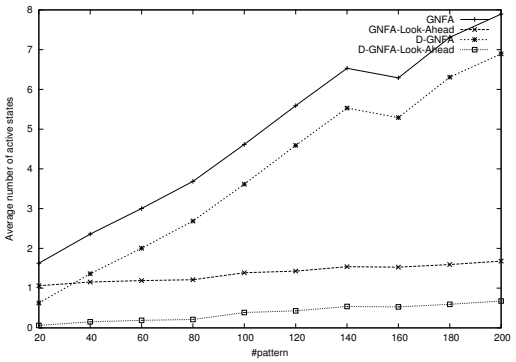


Figure 11. Average number of active states for 1st experiment.

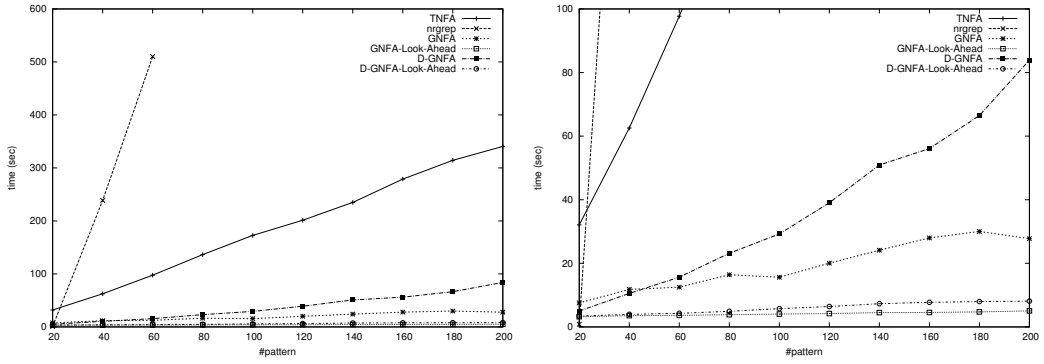


Figure 12. Needed time (sec) to matching whole text of "English.100MB" for 2nd experiment. Right part is the part of left part.(Right pert is scaled-up)

method since the pattern is a set of fixed string. However, we can see the basic speed of pattern matching by treating the pattern as a regular expression.

Fig. 12 shows the results. The trends resemble those of first experiment. G-NFA-Look-Ahead or Dual G-NFA-Look-Ahead was superior in terms of calculation time.

In the third experiment, we challenged our method with some actual regular expression patterns in Table 4. First pattern "suffix" matches to words that have some specific suffixes. There were 35 suffixes. Second pattern "prefix" matches to words that have some specific prefixes. There were 32 prefixes. Third pattern "name" matches

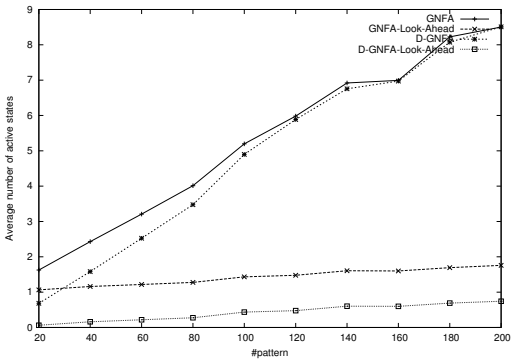


Figure 13. Average number of active states for 2nd experiment.

to some people's names. The names were combination of ten common given names and ten common surnames. Fourth pattern "user" matches to popular expression of user and computer name. Fifth pattern "title" matches strings that composed of capitalized words like a chapter title in books. These patterns include symbol classes like "[a-zA-Z]".

name	pattern sample
suffix	`[a-zA-Z]+(able|ible|al|...|ise)`
prefix	`(in|il|im|infra|...|under)[a-zA-Z]+`
names	`(Jackson|Aiden|...|Jack) (Smith|Johnson|...|Rodriguez|Wilson)`
user	`[a-zA-Z]+@[a-zA-Z]+`
title	`([A-Z]+ )+`

Table 4. Regular expression patterns used in third experiment.

As shown in Table 5, Dual G-NFA with LA is the fastest in some cases, once again to the reduction in active state size. Look ahead methods never match slower than T-NFA, G-NFA and Dual G-NFA. If that input consists of only small regular expression like pattern "name", "user" or "title", NR-grep is the fastest. For such patterns, bit parallel method implemented in NR-grep can manipulate G-NFAs effectively.

pattern	T-NFA (by Cox)	ngrep	G-NFA	G-NFA with LA	Dual G-NFA	Dual G-NFA with LA
suffix	113.48	20.24	9.74	7.51	106.64	3.35
prefix	14.33	5.295	2.74	3.97	78.39	3.82
names	12.95	0.216	2.97	2.74	3.21	2.76
user	78.14	0.08	12.11	7.41	185.22	3.36
title	38.88	0.186	2.93	2.38	2.68	2.21

Table 5. Needed time (sec) to matching with whole text of "English.100MB"

5 Conclusion

We proposed the new regular expression matching method that based on Dual G-NFA and Look Ahead Matching. We have shown that Dual G-NFA can construct a look ahead matching index without additional space. Simulations have shown the effectiveness of look ahead matching in accelerating NFA. From the experimental

pattern	G-NFA	G-NFA with LA	Dual G-NFA	Dual G-NFA with LA
suffix	2.33	1.65	50.44	1.15
prefix	1.50	1.15	8.11	0.33
names	1.01	1.00	0.01	0.001
user	1.77	1.59	40.67	0.59
title	1.03	1.01	0.75	0.01

Table 6. Average number of active states

results, our method can be useful for regular expression matching in practical usage. G-NFAs are used in some bit parallel methods, so we now plan to apply bit parallel techniques to Dual G-NFA methods.

References

1. A. V. Aho and J. E. Hopcroft: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1st ed., 1974.
2. G. Berry and R. Sethi: *From regular expressions to deterministic automata.* Theoretical computer science, 48 1986, pp. 117–126.
3. N. Cascarano, P. Rolando, F. Risso, and R. Sisto: *iNFAnt: NFA pattern matching on GPGPU devices.* ACM SIGCOMM Computer Comm. Review, 40(5) 2010, pp. 20–26.
4. R. Cox: *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).* `http://swtch.com/~rsc/regexp/regexp1.html`, January 2007.
5. N. de Beijer: *Stretching and jamming of automata.* Masters thesis, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 2004.
6. N. de Beijer, L. G. Cleophas, D. G. Kourie, and B. W. Watson: *Improving automata efficiency by stretching and jamming*, in Proceedings of the Fifteenth Prague Stringologic Conference, September 2010, pp. 9–24.
7. P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes: *An efficient and scalable semiconductor architecture for parallel automata processing.* IEEE Transactions on Parallel and Distributed Systems, 2013.
8. P. Ferragina and G. Navarro: *The Pizza & Chili Corpus.* `http://pizzachili.dcc.uchile.cl/`.
9. J. E. HOPCROFT, R. MOTWANI, and J. D. ULLMAN: *Introduction to Automata Theory, Languages, and Computation, Third Edition*, Addison Wesley, 2006.
10. Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga: *Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching*, in Proc. FPT'10, Dec 2010, pp. 21–28.
11. G. Navarro: *Nr-grep: a fast and flexible pattern-matching tool.* Software: Practice and Experience, 31(13) 2001, pp. 1265–1312.
12. G. Navarro and M. Raffinot: *Flexible pattern matching in strings — practical on-line search algorithms for texts and biological sequences.*, Cambridge, 2002.
13. D. Perrin: *Finite automata*, in Handbook of Theor. Comput. Sci, Vol.B, Chap. 1, J. van Leeuwen, ed., 1990, pp. 1–57.
14. *RE2 an efficient, principled regular expression library*: `https://code.google.com/p/re2/`.
15. Y. Wakaba, M. Inagi, S. Wakabayashi, and S. Nagayama: *An efficient hardware matching engine for regular expression with nested kleene operators*, in Proc. FPL'11, 2011, pp. 157–161.
16. B. W. Watson: *A taxonomy of finite automata construction algorithms*, tech. rep., Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 1993.
17. B. W. Watson: *The design of the FIRE engine: A C++ toolkit for FInite automata and Regular Expressions*, tech. rep., Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 1994.
18. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, September 1995.

A Process-Oriented Implementation of Brzozowski's DFA Construction Algorithm

Tinus Strauss[1], Derrick G. Kourie[2], Bruce W. Watson[2], and Loek Cleophas[1]

[1] FASTAR Research group
University of Pretoria
South Africa
{tinus,loek}@fastar.org
[2] FASTAR Research group
University of Stellenbosch
South Africa
{derrick,bruce}@fastar.org

Abstract. A process-algebraic description of Brzozowski's deterministic finite automaton construction algorithm, slightly adapted from a previous version, shows how the algorithm can be structured as a set of communicating processes. This description was used to guide a process-oriented implementation of the algorithm.
The performance of the process-oriented algorithm is then compared against the sequential version for a statistically significant number of randomly generated regular expressions. It is shown that the concurrent version of the algorithm outperforms the sequential version both on a multi-processor machine as well as on a single-processor multi-core machine. This is despite the fact that processor allocation and process scheduling cannot be user-optimised but are, instead, determined by the operating system.

Keywords: automaton construction, concurrency, CSP, regular expressions

1 Introduction

Although contemporary computers commonly have multicores, the processor allocation and scheduling is not in the hands of the application software developer but, instead, determined by the operating system. This fact raises numerous questions. What are the implications of parallelising algorithms that have traditionally been expressed sequentially? The strengths and weaknesses of the sequential algorithms are generally well-known, and often a lot of effort has gone into sequential optimisations. Furthermore, for most software developers parallel thinking is unfamiliar, difficult and error-prone compared to sequential algorithmic thinking. Is parallel thinking inherently difficult for software developers or is the relative scarcity of parallel versions of sequential software simply a matter of inertia? Is it worth the effort to convert traditional sequential algorithms into parallel format when the fate of the software—the processor allocation and process scheduling—is largely out of the developer's control? Perhaps questions such as these explain, at least in part, why there has not been a mushrooming of parallel software algorithm development, notwithstanding more than a decade of hype about the future of computing being in parallelism. These observations apply not only to algorithmic software development in general, but also to the specific case of algorithmic software related to stringology.

This paper makes a start in assessing the practical implications of developing and implementing parallel algorithmic versions of well-known stringological sequential algorithms in contexts where we have no direct control over processor allocation and

Tinus Strauss, Derrick G. Kourie, Bruce W. Watson, Loek Cleophas: A Process-Oriented Implementation of Brzozowski's DFA Construction Algorithm, pp. 17–29.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

scheduling. A process algebraic description of a Brzozowski's deterministic finite automaton construction algorithm, slightly adapted from a previous version [15], shows how the algorithm can be structured as a set of communicating processes in Hoare's CSP [8,7]. This description was used to guide a process-oriented implementation of the algorithm in Go [17], as Go's concurrency features (inspired by CSP) allowed us to easily map from CSP to Go. A scheme is described to randomly generate regular expressions within certain constraints. The performance of the process-oriented algorithm is then compared against the sequential version for a statistically significant number of randomly generated regular expressions. It is shown that the concurrent version of the algorithm outperforms the sequential version on a multi-processor machine, despite the fact that processor allocation and process scheduling cannot be user-optimised but is, instead, determined by the operating system.

Of course, [15] is one of several efforts at developing parallel algorithms for stringological problems. Some previous efforts include [4,18] for finite automaton construction, [12,3,9] for membership testing, and [16,10,14] for minimization. In [6] Watson and Hanneforth consider the parallelisation of finite automaton determinisation and in [15], a high-level CSP-based specification of Brzozowski's DFA construction algorithm was proposed.

The next section discusses Brzozowski's classical sequential DFA construction algorithm. Section 3 then presents a process-oriented implementation of the algorithm, suitable for concurrent execution, and is followed by a performance comparison between the two approaches in Section 4. Section 5 presents some concluding remarks and ideas for future work.

2 Sequential algorithm

Brzozowski's DFA construction algorithm [2] employs the notion of derivatives of regular expressions to construct a DFA. The algorithm takes a regular expression E as input and constructs an automaton that accepts the language represented by E.

The automaton is represented using the normal five-tuple notation $(D, \Sigma, \delta, S, F)$ where D is the set of states; Σ the alphabet; δ the transition relation mapping a state and an alphabet symbol to a state; and $S, F \subseteq D$ are the start and final states, respectively. $\mathcal{L}$ is an overloaded function giving the language of a finite automaton or a regular expression.

Brzozowski's algorithm identifies each DFA state with a regular expression. Apart from the start state, this regular expression is the derivative of a parent state's associated regular expression[1]. Elements of D may therefore interchangeably be referred to either as regular expressions or as states, depending on the context of the discussion.

The well-known sequential version of the algorithm is given in Dijkstra's guarded command language [5] in Figure 1. The notation assumes that the set operations ensure 'uniqueness' of the elements at the level of *similarity* [2, Def 5.2], i.e. $a \in A$ implies that there is no $b \in A$ such that a and b are similar regular expressions.
The algorithm maintains two sets of regular expressions (or states): a set T ('to do') containing the regular expressions for which derivatives need to be calculated; and another set D ('done') containing the regular expressions for which derivatives have

[1] In fact, it can be shown that the language of each state's associated regular expression is also the right language of that state.

```
func Brz(E, Σ) →
    δ, S, F := ∅, {E}, ∅;
    D, T := ∅, S;
    do (T ≠ ∅) →
        let q be some state such that q ∈ T;
        D, T := D ∪ {q}, T \ {q};
        { build out-transitions from q on all alphabet symbols }
        for (a : Σ) →
            { find derivative of q with respect to a }
            d := a⁻¹q;
            if d ∉ (D ∪ T) → T := T ∪ {d}
            [] d ∈ (D ∪ T) → skip
            fi;
            { make a transition from q to d on a }
            δ(q, a) := d
        rof;
        if ε ∈ L(q) → F := F ∪ {q}
        [] ε ∉ L(q) → skip
        fi
    od;
    return (D, Σ, δ, S, F)
cnuf
```

Figure 1. Brzozowski's DFA construction algorithm

been found already. When the algorithm terminates, T is empty and D contains the states of the automaton which recognises $\mathcal{L}(E)$.

The algorithm iterates through all the elements $q \in T$, finding derivatives with respect to all the alphabet symbols and depositing these new states (regular expressions) into T in those cases where no similar regular expression has already been deposited into $T \cup D$.

Each q, once processed in this fashion, is then removed from T and added into D.

In each iteration of the inner **for** loop (i.e. for each alphabet symbol), the δ relation is updated to contain the mapping from state q to its derivative with respect to the relevant alphabet symbol.

Finally if state q represents a regular expression whose language contains the empty string[2], then that state is included in the set of final states F.

In the forthcoming section we present a process-oriented implementation of the algorithm in which we attempt to structure the algorithm around a number of communicating sequential processes which may benefit from concurrent execution.

3 Concurrent description

We present here an approach to decompose the algorithm into communicating processes. These processes may then be executed concurrently which may result in im-

[2] Such a regular expression is called "nullable".

proved runtimes on multi-processor platforms. Of the many process algebras that have been developed to concisely and accurately model concurrent systems, we have selected CSP [8,7] as a fairly simple and easy to use notation. It is arguably better known and more widely used than most other process algebras. Below, we provide a brief introduction to the CSP operators that are used in the subsequent process definitions.

3.1 Introductory Remarks

CSP is concerned with specifying a system of communicating sequential processes (hence the CSP acronym) in terms of sequences of events, called traces. Various operators are available to describe the sequence in which events may occur, as well as to connect processes. Table 1 briefly outlines the main operators used in this article.

$a \rightarrow P$	event a then process P
$a \rightarrow P \vert b \rightarrow Q$	a then P choice b then Q
$x : A \rightarrow P(x)$	choice of x from set A then $P(x)$
$P \parallel Q$	P in parallel with Q
	Synchronize on common events in the alphabet of P and Q
$b!e$	on channel b output event e
$b?x$	from channel b input to variable x
$P \not< C \not> Q$	if C then process P else process Q
$P; Q$	process P followed by process Q
$P \,\Box\, Q$	process P choice process Q

Table 1. Selected CSP notation

Full details of the operator semantics and laws for their manipulation are available in [8,7]. Note that *SKIP* designates a special process that engages in no further event, but that simply terminates successfully. Parallel synchronisation of processes means that if $A \cap B \neq \emptyset$, then process $(x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y))$ engages in some nondeterministically chosen event $z \in A \cap B$ and then behaves as the process $P(z) \parallel Q(z)$. However, if $A \cap B = \emptyset$ then deadlock results. A special case of such parallel synchronisation is the process $(b!e \rightarrow P) \parallel (b?x \rightarrow Q(x))$. This should be viewed a process that engages in the event $b.e$ and thereafter behaves as the process $P \parallel Q(e)$. This can also be interpreted as processes communicating over a channel b. The one process writes e onto channel b and the other process reads from channel b into variable x.

3.2 Process descriptions

The concurrent version of the algorithm can be modeled as a process *BRZ* which is composed of four concurrent processes which may themselves be composed of more processes. The first of these processes is named *OUTER* and corresponds to the outer loop of the algorithm in Figure 1. It is responsible for maintaining the sets D, T, and F. The second process computes the derivatives of regular expressions and is called *DERIV*. A *FANOUT* process is responsible for distributing a regular expression to the components of *DERIV*. The final process *UPDATE* modifies the transition relation δ. The process definition for *BRZ(T,D,F,δ)* is thus:

$$BRZ(T, D, F, \delta) = OUTER(T, D, F) \parallel FANOUT \parallel DERIV \parallel UPDATE(\delta)$$

Process $OUTER$ is modelled as a process which initialises its local state and then repeatedly performs actions to modify T, D, and F as well as its local state. This repetition is modelled as process $LOOP$.

$$OUTER(T, D, F) = init \rightarrow LOOP(T, D, F)$$

$LOOP$ first modifies the local state and has a choice between the following behaviours. It may write $q \in T$ to channel $outNode$ and repeat, it may receive a new regular expression d from channel $inNode$ and repeat, or it may terminate if the local state is such that no more states need to be processed. When $LOOP$ sends q out, q is removed from T and added to D and if $\varepsilon \in \mathcal{L}(q)$ then q is also added into F. In the case when a new state is received it is added into T if no similar state is in $T \cup D$.

$$\begin{aligned} LOOP(T, D, F) = {} & modifyLocalState \rightarrow \\ & ((q : T \rightarrow outNode!q \rightarrow \\ & \quad LOOP(T \setminus q, D \cup q, F \cup q) \not< \varepsilon \in \mathcal{L}(q) \not> LOOP(T \setminus q, D \cup q, F)) \\ & \Box \\ & (inNode?d \rightarrow LOOP(T \cup d, D, F) \not< d \notin T \cup D \not> LOOP(T, D, F)) \\ & \Box \\ & SKIP) \end{aligned}$$

$DERIV$ is responsible for concurrently calculating the derivatives of a regular expression with respect to each alphabet symbol. This corresponds to the inner `for` loop in Figure 1. $DERIV$ is thus modelled as the concurrent composition of $|\Sigma|$ processes, each responsible for calculating the derivative with respect to a given $i \in \Sigma$.

$$DERIV = \|_{i:\Sigma}\ DERIV_i$$

Each $DERIV_i$ repeatedly reads a regular expression re from its input channel $dOut_i$, calculates the derivative and then sends the result as a triple $\langle re, i, i^{-1}re \rangle$ out on a shared channel $derivChan$.

$$DERIV_i = dOut_i?re \rightarrow computeDeriv.re \rightarrow derivChan!\langle re, i, i^{-1}re \rangle \rightarrow DERIV_i$$

The $FANOUT$ process connects $OUTER$ and $DERIV$ and is responsible for communicating the regular expressions from $OUTER$ to each $DERIV_i$. It repeatedly reads a regular expression from its input channel $outNode$ and concurrently replicates it to the $|\Sigma|$ output channels $dOut_i$.

$$FANOUT = (outNode?re \rightarrow \|_{i:\Sigma}\ (dOut_i!re \rightarrow SKIP)); FANOUT$$

In order to complete the DFA, we need to record all the state transitions in δ. This is the responsibility of $UPDATE$. It is modelled as a repeating process which reads a triple $\langle re, i, d \rangle$ from its input channel $derivChan$ and records $\delta(re, i) = d$. It also sends one element of the triple, d, on to $OUTER$ via channel $inNode$. This d is potentially a new state from which transitions should be calculated and hence it should be added into T if a similar node has not been processed before.

$$UPDATE(\delta) = derivChan?\langle re, i, d \rangle \rightarrow inNode!d \rightarrow UPDATE(\delta \cup \langle re, i, d \rangle)$$

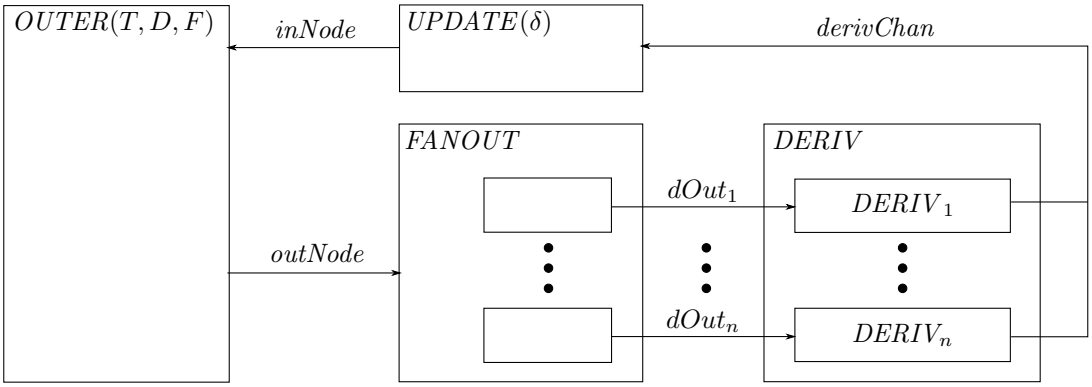


Figure 2. The communications network of the *BRZ* process.

Figure 2 shows the communicating processes along with their associated input and output channels.

Termination is not addressed completely in the above process models. Notably, processes which repeatedly read from an input channel live until their input channels are closed. Consequently they can be modelled as a choice between reading from the channel and *SKIP*. These choices were omitted above to simplify the presentation.

In the following section we compare the performance of the concurrent implementation against the sequential algorithm.

4 Performance comparison

In the preceding section the Brzozowski DFA construction algorithm was decomposed into a network of communicating processes. The next step is then to implement the CSP descriptions as an executable program and compare the performance of the sequential and concurrent algorithms.

It was decided to use the Go programming language [17] for the implementation. Go is a compiled language with concurrency features inspired by Hoare's CSP. Particularly relevant to the present context is the fact that Go has channels as first class members of the language. The CSP processes in the process network from Section 3 map to so-called go-routines and the communication channels map to Go channels. An alternative language that implements CSP-like channels of which we are aware is occam-π [1]. Go was chosen over occam-π since Go allows us to implement data structures more easily and documentation is also more readily available.

4.1 Experimental setup

The aim of the present experiment is simply to test the hypothesis that the concurrent implementation can construct DFAs faster than the sequential version. No attempt was made to investigate completely the performance characteristics of the process-oriented implementation.

In order to compare the performance the following approach was followed. A regular expression was generated and both the sequential and concurrent algorithms were executed with this regular expression. The respective execution times were recorded. In order to reduce the effects of transient operating system events, each construction

was executed 30 times and the minimum duration was used as the data point for that regular expression. Various regular expressions were used as input to observe the performance of the algorithms over a range of input.

Regular expressions were randomly generated via a simple recursive procedure $gen(\Sigma, d)$. The procedure takes as input two parameters: an alphabet Σ and an integer d. If $d = 0$ the procedure returns a random symbol from Σ. If $d > 0$ then $gen(\Sigma, d)$ randomly chooses a regular expression operator and then recursively generates the required operands for the operator by calling $gen(\Sigma, d-1)$. The size of the regular expression is thus controlled by d since d defines the depth of the expression tree for the regular expression. The upper bound for the number of operators in the tree is $2^d - 1$ and for the number of symbols (leaves) it is 2^d. Many generated regular expressions will be smaller since some regular expression operators are unary operators which result in a tree that is smaller than a complete binary tree.

We decided to consider the performance of the algorithms with regular expressions over both a small alphabet and a larger alphabet. The small alphabet contained 4 symbols and the larger alphabet 85 symbols.

Regular expression were generated with depths $d = 5, 6, \ldots, 10$. For each of the 12 elements in $\{4, 85\} \times \{5, 6, \ldots, 10\}$ we generated 50 regular expressions.

The implementations were compiled in Go version 1.2.2 and initially executed in Mac OS X 10.7.5 on a MacPro1,1 with two Dual-Core Intel Xeon 2.66 GHz processors and 5 GB RAM. The runtimes of the programs on this platform are analysed in the next section.

4.2 Observations

Let us now consider the results of the experiment. The results will be communicated mainly through graphical plots and a few statistical calculations. These plots and statistical calculations were produced using the statistical system R version 3.1.0 [13]. In most cases the two alphabet cases were considered separately.

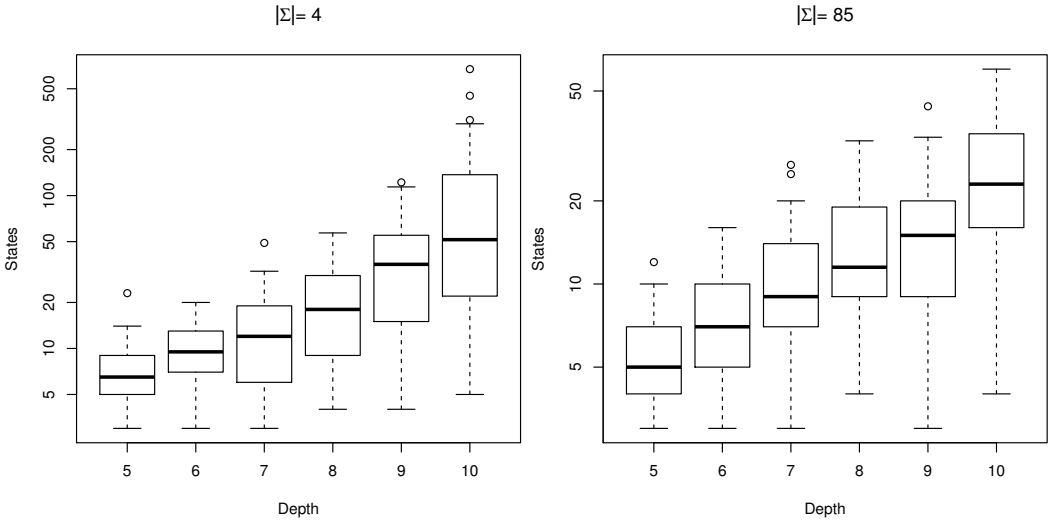


Figure 3. Sizes of automata generated.

First we consider the sizes of the automata constructed by the algorithms. The plots in Figure 3 show the number of states (on a logarithmic scale) in the resultant

automata. The plots in the figure provide a visualisation of the distribution of the 50 values for each of the depths. As expected, the regular expressions with larger depths generally yielded larger automata. It should also be noted that for the small alphabet case a few very large automata were constructed. The data confirm that the input regular expressions did indeed vary significantly and hence the algorithms were executed with a variety of input. In future work a more sophisticated approach to obtain input should be considered so that one may better control the nature of the input regular expressions.

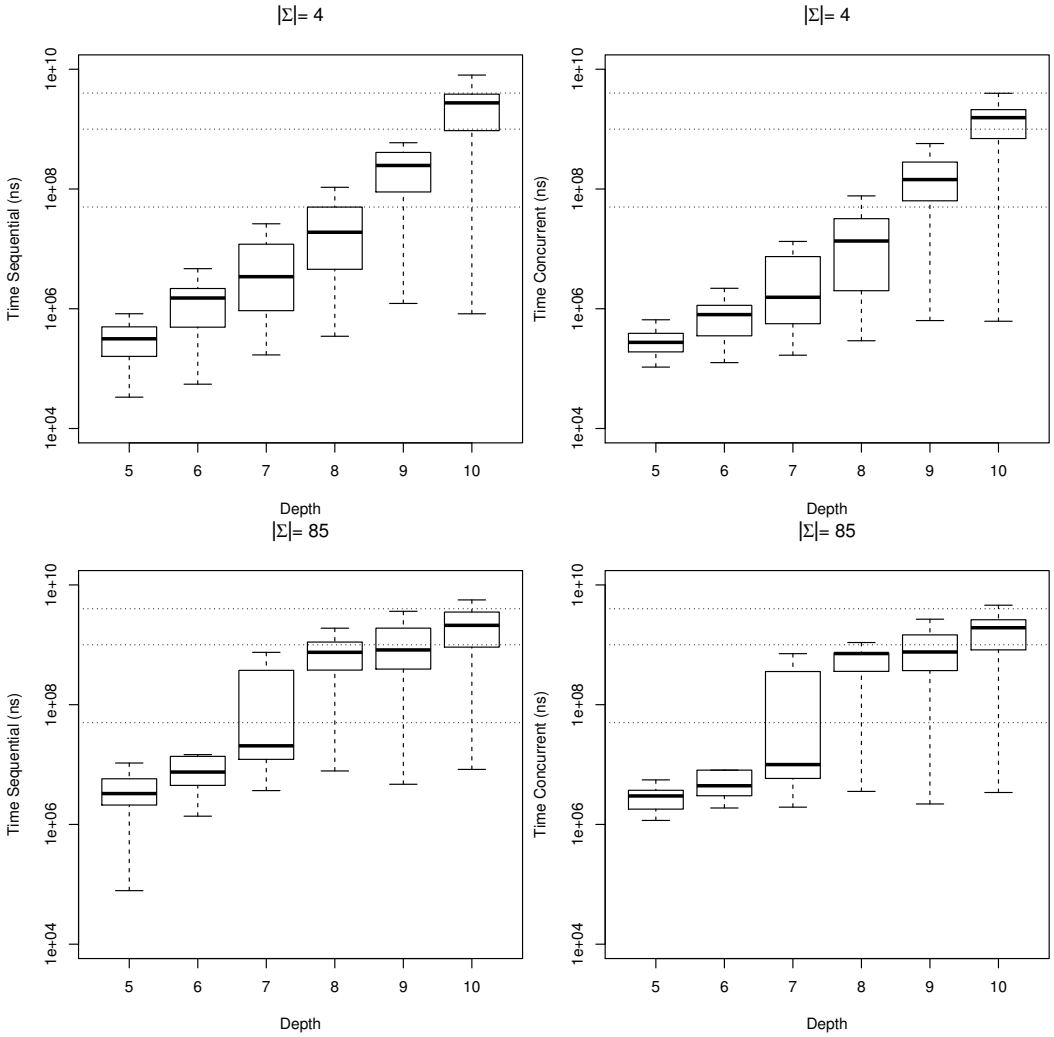


Figure 4. Construction times against problem size.

Let us now consider the construction times. Figure 4 shows the construction times for the sequential algorithm and the concurrent algorithm in both of the alphabet cases for the various regular expression sizes. Note that in each plot the y-axis is logarithmic. Outlying observations were omitted from the plots to make them clearer.

From the plots it is clear that in all cases the time increased as the regular expression grew. It can also be seen – although less clearly due to the logarithmic scale – that the construction time for the concurrent algorithm tends to be lower than that of the sequential algorithm. The construction time difference is less pronounced in the large alphabet case. This could be due to the larger overhead involved in this

case. For example, when one considers the process *FANOUT* from Figure 2 it will be seen that it creates a process for each alphabet symbol. As the alphabet grows, this creation and scheduling overhead will also increase.

The data from the plots suggest that the concurrent algorithm may indeed be faster. To confirm this we performed the Wilcoxon signed rank test for paired observations. The test tests the null hypothesis that the median difference between the pairs of construction times is zero against the alternative that the median difference is greater than zero:

$$H_0 : \text{median difference between runtimes is } 0$$
$$H_a : \text{median difference between runtimes is greater than } 0$$

The results of the tests are as follows.

	N	Test statistic	$p < 0.01$
$\lvert\Sigma\rvert = 4$	300	44482	Yes
$\lvert\Sigma\rvert = 85$	300	44141	Yes

In both the small alphabet and large alphabet cases the null hypothesis can be rejected at 99% level of confidence. Our data thus provides evidence in support of our hypothesis that the concurrent algorithm can outperform the sequential one.

To explore further the relationship between the sequential and concurrent construction times scatterplots were constructed. These can be found in Figure 5. Each point represents a pair of sequential and concurrent runtimes for a given regular expression. The x-coordinate of the point is the sequential runtime and the y-coordinate is the concurrent runtime. Each plot contains $50 \times 6 = 300$ points. From the plots

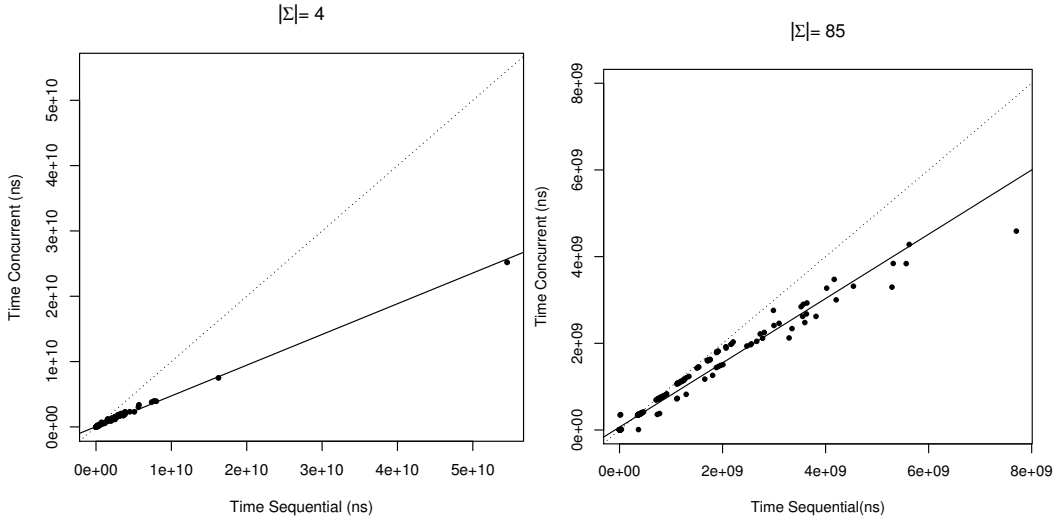


Figure 5. Scatter plots of sequential time against concurrent time.

it can be seen that there appears to be a linear relationship between the sequential and concurrent runtimes. Linear regression lines were fitted to data and also plotted on the graph as solid lines. The dotted line in each plot is simply a line through the origin with slope 1. From the graph it can be seen that the slope of the regression line for the small alphabet case is less steep than that of the regression line for the large alphabet case.

If we let T_c and T_s be the construction times for the concurrent and sequential algorithms respectively, then the regression lines that were fitted are as follows.

$$T_c = 39.6\,\text{ms} + 0.47 \cdot T_s \qquad \text{for } |\Sigma| = 4$$
$$T_c = 65.7\,\text{ms} + 0.74 \cdot T_s \qquad \text{for } |\Sigma| = 85$$

The fact that the slopes are less than one is consistent with the fact that the concurrent construction times are smaller than the sequential times. From the slope terms in the equations above it is clear that the performance increase for the small alphabet case was greater than for the large alphabet case. As mentioned earlier this can be explained by the greater amount of overhead present in the large alphabet case.

Speedup and efficiency are well-known metrics for characterising parallel algorithmic performance [11]. Speedup is defined as the execution time of the sequential program divided by the execution time of the parallel program. Efficiency is defined as the speedup divided by the number of processes. Table 2 contains the observed speedup and efficiency for our experiment. Each entry shows the median for the relevant subset of the data.

	Speedup		Efficiency	
Depth	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$
All	1.72	1.09	0.43	0.27
5	1.15	1.21	0.29	0.30
6	1.84	1.45	0.46	0.36
7	1.82	1.43	0.46	0.36
8	1.80	1.06	0.45	0.27
9	1.71	1.09	0.43	0.27
10	1.83	1.21	0.46	0.30

Table 2. Speedup and efficiency overall and for different problem sizes.

Ideal speedup in a p processor environment is p and efficiency equal to 1 is very good. The median speedup for the small alphabet case is 1.72 and for the large alphabet case it is 1.09. Recall that the total number of cores in our experimental platform was four. We have clearly not achieved optimal speedup, but we have a median speedup greater than one. From this and from the relatively low efficiency numbers it is clear that the process-oriented approach is promising especially if we can reduce the amount of overhead.

Finally, let us consider whether the problem size influences the speedup. Figure 6 shows the plots for speedup against the depths used to generate the regular expressions. The plots show that the speedup for smaller regular expressions is sometimes less than 1. This implies that the concurrent version is sometimes slower than the sequential version for these smaller regular expressions. This effect is more pronounced in the large alphabet case. In the small alphabet case, the effect is seen at depths 5 and 6, but in the large alphabet case the speedup less than 0 is also found at depth 7. In the smaller regular expressions the overhead removes entirely the performance gain of the concurrent processes. In larger expressions the nett gain is still positive.

4.3 A second experiment

In order to gain insight into whether or not the foregoing results are reasonably robust across different platforms, we repeated the experiment on a newer machine with a

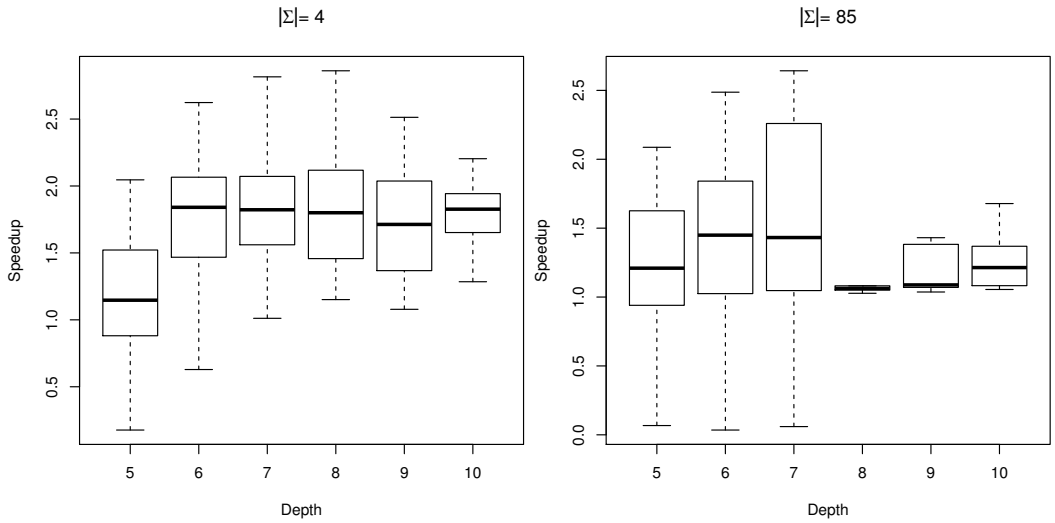


Figure 6. Speedup against problem size.

slightly different configuration. This machine is a MacBookPro11,3. It has a four-core *single* Intel i7 processor running at 2.3 GHz, whereas the earlier machine had *two* Dual-Core Intel Xeon processors, each running at 2.66 GHz. The newer machine ran Mac OS X 10.9.3 as opposed to the earlier machine which ran Mac OS X 10.7.5. Finally, the newer machine had considerably more RAM—16 GB compared to the earlier machine's 5 GB.

The data produced by this experiment results in figures that are broadly similar to those in Figures 3, 4 and 5, but on a somewhat different scale—i.e. there is a general increase in speedup and efficiency. The best speedup attained was 2.2, representing an efficiency of 0.55. This was for the large alphabet when the parameter d for the depth of regular expression trees was set to 5. Rather than providing all the raw values, Table 3 shows the increases in speedup and efficiency as a *percentage* of the corresponding data in Table 2.

Note that the improvement is largest for the large alphabet case where overall speedup and efficiency increases are attained of approximately 40%. The table also illustrates that these gains tend to diminish as the problem size increases. However, there is no obvious relationship between problem size and the extent to which the gains diminish. For the largest problem size, the speedup and efficiency increases on the large alphabet were around 20%. By way of comparison, there was a mere 2% to 3% increase in the case of the small alphabet.

It has already been pointed out that in the concurrent design that has been implemented, an increase in alphabet size results in an increase in process creation and scheduling. There results suggest that the overhead required to create and schedule processes over four cores on the same CPU is somewhat more efficient than achieving the same task across two dual-core CPUs. It has been left, however, to future research to carry out a more fine-grained analysis to determine the contribution of other factors to improved performance, such as the increase in RAM size, the small increase in clock speed and the more recent version of the operating system.

	Speedup Increase		Efficiency Increase	
Depth	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$	$\|\Sigma\| = 4$	$\|\Sigma\| = 85$
All	12.8%	40.4%	14.0%	40.7%
5	22.6%	54.5%	20.7%	56.7%
6	2.7%	51.7%	2.2%	52.8%
7	15.9%	44.8%	15.2%	44.4%
8	15.6%	13.2%	15.6%	11.1%
9	18.1%	16.5%	16.3%	14.8%
10	3.3%	18.2%	2.2%	20.0%

Table 3. Speedup and efficiency increase on the four core machine.

5 Conclusion

We set out to test whether a process-oriented implementation of Brzozowski's DFA construction algorithm could outperform the sequential implementation in a multi-processor environment. The results of our experiment, carried out on two different (but similar) platforms, confirm that it is indeed possible. In neither case was the speedup close to ideal. Nevertheless, there were instances where double the speed of the sequential algorithm was reached. Even though this represents an efficiency of about 50%, the results are a big improvement over the sequential algorithm's runtime in light of typical under-utilization of multi-core capabilities of present-day CPUs.

Inefficiencies in the process-oriented implementation are, no doubt, part of the reason why efficiency measurements are not higher. The *FANOUT* process, in particular, could be enhanced to be more efficient by creating fewer processes. Future work would include improving on implementation efficiency and exploring further algorithms to implement in a process-oriented manner. It will also be interesting to verify results to date on a wider variety of platforms.

In the first experimental setup we used a machine with two CPUs, each with two cores. We conjecture that the operating system may schedule the threads of the executable to run only on one of the two processors—effectively utilising only two cores for the execution. If this turns out to be true, the observed speedup is, especially in the small alphabet case, rather closer to the ideal. This matter should be investigated further. As a simple first step, we repeated the experiment on a machine with a single processor with four cores and compared the results, obtaining somewhat improved efficiencies. More sophisticated profiling tools are, however, needed to examine the behaviour of the running processes in finer detail.

This uncertainty regarding the operating system's scheduling behaviour raises the theme of control over scheduling of tasks. A number of questions immediately come to mind. Would greater speedups be possible if such control was readily available? What (if any) are the disadvantages of granting greater control to software developers? Could such control mechanisms not be built into the operating systems, accompanied by appropriate escape measures in case the user abuses these mechanisms? These questions open up various avenues for further research.

References

1. F. Barnes and P. Welch: *occam-pi: blending the best of CSP and the pi-calculus.* `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
2. J. A. Brzozowski: *Derivatives of regular expressions.* Journal of the ACM, 11(4) 1964, pp. 481–494.
3. B. Burgstaller, Y.-S. Han, M. Jung, and Y. Ko: *On the parallelization of DFA membership tests*, tech. rep., Technical Report. TR-0003, Department of Computer Science, Yonsei University, Seoul 120–749, Korea. http://elc. yonsei. ac. kr/PDFA. html, 2011.
4. H. Choi and B. Burgstaller: *Non-blocking parallel subset construction on shared-memory multicore architectures*, in Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140, Australian Computer Society, Inc., 2013, pp. 13–20.
5. E. W. Dijkstra: *A Discipline of Programming*, Prentice Hall, 1976.
6. T. Hanneforth and B. W. Watson: *An efficient parallel determinisation algorithm for finite-state automata*, in Stringology, J. Holub and J. Žďárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2012, pp. 42–52.
7. C. A. R. Hoare: *Communicating sequential processes.* Communications of the ACM, 26(1) 1983, pp. 100–106.
8. C. A. R. Hoare: *Communicating sequential processes (electronic version)*, 2004, `http://www.usingcsp.com/cspbook.pdf`.
9. J. Holub and S. Štekr: *On parallel implementations of deterministic finite automata*, in Implementation and Application of Automata, S. Maneth, ed., vol. 5642 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 54–64.
10. J. JáJá and K. W. Ryu: *An optimal randomized parallel algorithm for the single function coarsest partition problem.* Parallel Processing Letters, 6(2) 1996, pp. 187–193.
11. A. H. Karp and H. P. Flatt: *Measuring parallel processor performance.* Commun. ACM, 33(5) May 1990, pp. 539–543.
12. Y. Ko, M. Jung, Y.-S. Han, and B. Burgstaller: *A speculative parallel DFA membership test for multicore, simd and cloud computing environments.* International Journal of Parallel Programming, 2012, pp. 1–34.
13. R Core Team: *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014.
14. B. Ravikumar and X. Xiong: *A parallel algorithm for minimization of finite automata*, in IPPS, IEEE Computer Society, 1996, pp. 187–191.
15. T. Strauss, D. G. Kourie, and B. W. Watson: *A concurrent specification of Brzozowski's DFA construction algorithm.* Int. J. Found. Comput. Sci., 19(1) 2008, pp. 125–135.
16. A. Tewari, U. Srivastava, and P. Gupta: *A parallel DFA minimization algorithm*, in HiPC, S. Sahni, V. K. Prasanna, and U. Shukla, eds., vol. 2552 of Lecture Notes in Computer Science, Springer, 2002, pp. 34–40.
17. The Go Authors: *The Go programming language.* `http://golang.org/`.
18. D. Ziadi and J.-M. Champarnaud: *An optimal parallel algorithm to convert a regular expression into its Glushkov automaton.* Theoretical Computer Science, 215(1-2) February 1999, pp. 69–87.

Efficient Online Abelian Pattern Matching in Strings by Simulating Reactive Multi-Automata

Domenico Cantone and Simone Faro

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

Abstract. The *abelian pattern matching problem* consists in finding all substrings of a text which are permutations of a given pattern. This problem finds application in many areas and can be solved in linear time by a naïve sliding window approach. In this paper we introduce a new approach to the problem which makes use of a reactive multi-automaton modeled after the pattern, and provides an efficient nonstandard simulation of the automaton based on bit-parallelism.

Keywords: string permutations, nonstandard pattern matching, combinatorial algorithms on words, bit-parallelism, reactive multi-automata

1 Introduction

Given a pattern `p` and a text `t`, the *abelian pattern matching* problem [11] (also known as *jumbled matching* [8,7]) consists in finding all substrings of the text `t`, whose characters have the same multiplicities as in `p`, so that they could be converted into the input pattern just by permuting their characters.

It is a special case of the *approximate string matching* problem and naturally finds applications in many areas, such as string alignment [4], SNP discovery [5], and also in the interpretation of mass spectrometry data [6].

In the field of text processing and in computational biology, algorithms for abelian pattern matching are used as a filtering technique [3], usually referred to as *counting filter*, to speed up complex combinatorial searching problems. For instance, the counting filter technique has been used in the solution to the k-mismatches [15] and k-differences [17] problems. More recently, it has also been used in a solution to the approximate string matching problem allowing for inversions [9] and translocations [14]. A detailed analysis of the abelian pattern matching problem and of its solutions is presented in [11].

In this paper we are interested in the *online* version of the problem, whose worst-case time complexity is well known to be $\mathcal{O}(n)$, which assumes that the input pattern and text are given together for a single instant query, so that no preprocessing is possible.

Specifically, after introducing in Section 2 the relevant notations and describing in Section 3 the related literature, we present in Section 4 a new solution of the online abelian pattern matching problem in strings, based on a generalization of reactive automata [10,13] in which multiple links are allowed. In addition, we propose a nonstandard simulation of the automaton based on bit-parallelism. Despite its quadratic worst-case complexity, the resulting algorithm performs very well in practice, better than existing solutions in most practical cases (especially when the alphabet is large), as can be inferred from the experimental results reported in Section 5. Finally, the paper is closed with some concluding remarks in Section 6.

Domenico Cantone and Simone Faro: Efficient Online Abelian Pattern Matching in Strings by Simulating Reactive Multi-Automata, pp. 30–42.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

2 Notations and Definitions

We represent a string p of length $|\mathsf{p}| = m > 0$ as a finite array $\mathsf{p}[0\,..\,m-1]$ of characters from a finite alphabet Σ of size σ. Thus, $\mathsf{p}[i]$ will denote the $(i+1)$-st character of p, for $0 \leq i < m$, whereas $\mathsf{p}[i\,..\,j]$ will denote the substring of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p.

For a character $c \in \Sigma$, we denote by $\rho_{\mathsf{p}}(c)$ the rightmost position in p of the character c, if present, -1 otherwise. Likewise, $\lambda_{\mathsf{p}}(c)$ will denote the leftmost position in p of the character c, if present, m otherwise. More formally, for $c \in \Sigma$, we have

$$\rho_{\mathsf{p}}(c) := \max\big(\{i \mid 0 \leq i < m \text{ and } \mathsf{p}[i] = c\} \cup \{-1\}\big)$$
$$\lambda_{\mathsf{p}}(c) := \min\big(\{i \mid 0 \leq i < m \text{ and } \mathsf{p}[i] = c\} \cup \{m\}\big)\,.$$

For any index $0 \leq i < m$, we let $\nu_{\mathsf{p}}(i)$ denote the smallest index $i < j < m$ such that $\mathsf{p}[j] = \mathsf{p}[i]$, if such an index exists, m otherwise. In addition, we extend the definition of ν_{p} to m by putting $\nu_{\mathsf{p}}(m) := -1$. In the rest of the paper, when the pattern p is understood, we will simply write λ, ρ, and ν in place of λ_{p}, ρ_{p}, and ν_{p}, respectively. For a function f, we use the notation f^j, with $j \geq 0$, for the j-th *iterate* of f.[1] Thus, for instance, $f^3(i) = f(f(f(i)))$.

It is easy to see that, for any index $0 \leq i < m$, the sequence of indices

$$\big\langle \lambda(\mathsf{p}[i]), \nu(\lambda(\mathsf{p}[i])), \nu^2(\lambda(\mathsf{p}[i])), \ldots, \nu^r(\lambda(\mathsf{p}[i]))\big\rangle,$$

where $r+1$ is the multiplicity of $\mathsf{p}[i]$ in p (so that $\nu^r(\lambda(\mathsf{p}[i])) = \rho(\mathsf{p}[i]))$, is the sequence of the positions of the character $\mathsf{p}[i]$ in p, in increasing order.

Example 1. Let $\mathsf{p} = gactaagtac$ be a pattern of length $m = 10$ over the alphabet $\Sigma = \{a, c, g, t\}$. Then we have $\lambda(a) = 1$ and $\rho(a) = 8$. Moreover, $\nu(1) = 4$, $\nu^2(1) = \nu(4) = 5$, and $\nu^3(1) = \nu(5) = 8 = \rho(a)$. Thus, $\langle 1, 4, 5, 8\rangle$ is the increasing sequence of the positions of the character a in p.

The *Parikh vector* [1,18] of p (denoted by pv_{p} and also known as *compomer* [6], *permutation pattern* [12], and *abelian pattern* [11]) is the vector of the multiplicities of the characters in p. More precisely, for each $c \in \Sigma$, we have

$$pv_{\mathsf{p}}[c] := |\{i : 0 \leq i < m \text{ and } \mathsf{p}[i] = c\}|\,.$$

In the following, the Parikh vector of the substring $\mathsf{p}[i\,..\,i+h-1]$ of p, of length h and starting at position i, will be denoted by $pv_{\mathsf{p}(i,h)}$.

In terms of Parikh vectors, the abelian pattern matching problem can be formally expressed as the problem of finding the set $\Gamma_{\mathsf{p},\mathsf{t}}$ of positions in t, defined as

$$\Gamma_{\mathsf{p},\mathsf{t}} := \{s :\ 0 \leq s \leq n-m \text{ and } pv_{\mathsf{t}(s,m)} = pv_{\mathsf{p}}\}.$$

We close the section by recalling that a *finite automaton* is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the collection of *final states*, Σ is an *alphabet*, and $\delta \subseteq (Q \times \Sigma \times Q)$ is the *transition relation* of $\mathcal{A}$. We also recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", and the `left shift` "$\ll$" operator (which shifts its first argument to the left by a number of bits equal to its second argument): in this context, we will say that a bit *is set* to indicate that its value is equal to 1.

[1] Formally, we put $f^0(x) := x$ and, recursively, $f^{j+1}(x) := f(f^j(x))$, provided that f is defined on $f^j(x)$.

3 Previous Results

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *online abelian pattern matching problem* can be solved in $\mathcal{O}(n)$ time and $\mathcal{O}(\sigma)$ space by using a naïve *prefix based approach* [11], which slides a window of size m over the text while updating in constant time the corresponding Parikh vector. Indeed, for each position $s = 0, 1, \ldots, n-m-1$ and character $c \in \Sigma$, we have

$$pv_{\mathsf{t}(s+1,m)}[c] = pv_{\mathsf{t}(s,m)}[c] - \left|\{c\} \cap \{\mathsf{t}[s]\}\right| + \left|\{c\} \cap \{\mathsf{t}[s+m]\}\right|,$$

so that the vector $pv_{\mathsf{t}(s+1,m)}$ can be computed from $pv_{\mathsf{t}(s,m)}$ by incrementing the value of $pv_{\mathsf{t}(s,m)}[\mathsf{t}[s+m]]$ and by decrementing the value of $pv_{\mathsf{t}(s,m)}[\mathsf{t}[s]]$. Thus, the test "$pv_{\mathsf{t}(s+1,m)} = pv_{\mathsf{t}(s,m)}$" can be easily performed in constant time.

A more efficient prefix-based approach, which uses less branch conditions, has been recently proposed in [14]. Specifically, for each position $0 \leq s \leq n-m$, a function $G_s : \Sigma \to \mathbb{Z}$ is defined by putting $G_s(c) := pv_{\mathsf{p}}[c] - pv_{\mathsf{t}(s,m)}[c]$, for $c \in \Sigma$. Also, a distance value δ_s can be defined as $\delta_s := \sum_{c\in\Sigma} \left|G_s(c)\right|$. Then the set $\Gamma_{\mathsf{p},\mathsf{t}}$ takes on the form $\Gamma_{\mathsf{p},\mathsf{t}} = \{s \ : \ 0 \leq s \leq n-m \text{ and } \delta_s = 0\}$. Observe that $G_{s+1}(c)$ and δ_{s+1} can be computed in constant time from $G_s(c)$ and from δ_s, respectively. Hence, it follows that all values δ_s, for $s = 0, \ldots, n-m$, can be computed in $\mathcal{O}(n)$ time.

A *suffix-based approach* to the problem has been presented in [11], as an adaptation of the Horspool strategy [16]. Rather than reading the characters of the window from left to right, characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $\mathcal{O}(nm)$ worst-case time complexity but performs well in practical cases.

Experimental results show that the prefix based algorithm outperforms the suffix based algorithm only for abelian patterns over small alphabets (as, for instance, in the case of binary data or DNA sequences) and for patterns whose characters have a frequency distribution similar to that of the input text. In all other cases, the suffix based approach achieves better results than the prefix based approach. The gap becomes more significant in the case of very large alphabets as is the case, for instance, in natural language texts.

In [11], a *parameterized suffix based approach* has been presented in which the current frequency vector is reset only if the number of the characters read before an overflow does not exceed εm, where ε is a user defined parameter. The worst-case time complexity of the resulting algorithm is $\mathcal{O}(\frac{n}{1-\varepsilon})$. However, experimental results show that the algorithm never outperforms the prefix- and the suffix-based algorithms.

For the sake of completeness, we notice that recently the problem has also been solved in its *offline* form, where one has to search for several patterns in the same text, so that it makes sense to perform in advance a suitable preprocessing of the text. We mention also a solution presented in [7,8], in which a useful data structure over the input text is constructed beforehand in $\mathcal{O}(n)$ time and space. As a result, each query can be answered in $\mathcal{O}(n)$ worst-case time complexity, though with a sublinear expected time complexity.

4 A New Algorithm Based on Reactive Multi-Automata

Reactive automata, introduced in [10,13], are ordinary automata (deterministic or nondeterministic) augmented with a switching mechanism to turn links on or off

during computation. Thanks to the switching mechanism, the number of states in an ordinary automaton can be dramatically reduced.

For our purposes, we will need to slight generalize the notion of reactive automata as given in [10,13], by also allowing multiple links labeled by a same character between any two states.[2] We will therefore provide in Section 4.1 a formal definition of *reactive multi-automata* and of the related acceptance notion. Then, in Section 4.2, we show how to construct a compact reactive multi-automaton which recognizes all abelian occurrences of a given input pattern, and prove its correctness in Section 4.3. Subsequently, in Section 4.4, we present an algorithm for the online abelian pattern matching problem which makes use of such an automaton and, finally, in Section 4.5 we describe how to efficiently simulate it using bit-parallelism.

4.1 Reactive Multi-Automata

A reactive automaton is an ordinary automaton extended with *reactive links* between its (ordinary) links. These can be of two types, namely *activation* and *deactivation* reactive links. At any step of the computation of a reactive automaton on a given input string S, states and links are distinguished as active and non-active. At start (step 0), the initial state is the only active state and all links of a given *initial transition relation* are active.[3] Active states at step $h+1$ are all states which are reachable by a direct *active* link (at step h), labeled by the character $S[h]$, from any *active* state (at step h). Active links at step $h+1$ are all links which are active at step h and are not deactivated in the transition from step h to step $h+1$, plus all links which are activated in the transition from step h to step $h+1$. A link is activated in the transition from step h to step $h+1$, if it is the endpoint of an activation reactive link from an active (ordinary) link at step h labeled by the character $S[h]$. A link is deactivated in the transition from step h to step $h+1$, if it is the endpoint of a deactivation reactive link from an active (ordinary) link at step h labeled by the character $S[h]$ *and* it does not get activated at the same time (in other words, we stipulate that when a link is both activated and deactived, activation prevails).

Reactive multi-automata extend reactive automata in that they allow the presence of multiple links labeled by a same character between any two states. We choose to represent multiplicity by means of *multiplicity labels* drawn from a finite set of labels L. Thus, a link in a multi-automata is a quadruple (q, c, ℓ, q'), where q, q' are states, c is an alphabet character, and ℓ is a multiplicity label. From an operational point of view, two links differing only on their multiplicity label are regarded just the same.

Let us be more formal. Let Q, Σ, L be finite sets of states, of characters, and of labels, respectively, and let $\mathcal{D} := Q \times \Sigma \times L \times Q$ denote the collection of all possible labeled links on Q, Σ, and L. Also, let $T^+, T^- \subseteq \mathcal{D} \times \mathcal{D}$ be two collections of activation and deactivation reactive links, respectively. Given a set $\psi \subseteq \mathcal{D}$ of links (which are supposed to be the active links at a certain step h) and a subset $\varphi \subseteq \psi$ (of the links in ψ from active states and labeled by the input word character which is being read at step h), then the set of active links (at the subsequent step $h+1$) relative to φ and to the collections T^+, T^- of reactive links, denoted by $\psi^{(\varphi,T^+,T^-)}$, is

$$\begin{aligned}\psi^{(\varphi,T^+,T^-)} := \big(\psi \setminus \{\gamma \mid \exists\, \tau \in \varphi \text{ such that } (\tau,\gamma) \in T^-\}\big) \\ \cup\ \{\gamma \mid \exists\, \tau \in \varphi \text{ such that } (\tau,\gamma) \in T^+\}\,.\end{aligned}$$

[2] In fact, we will only need multiple self-loops.

[3] As we will see, the initial transition is a subset of the transition relation of the underlying automaton.

The map $\psi \mapsto \psi^{(\varphi,T^+,T^-)}$ just defined is the *switch reactive transformation relative to* T^+, T^-.

We are now ready to give a precise definition of reactive multi-automata and of their nondeterministic runs.

Definition 2 (Reactive multi-automata). *Let Q, Σ, L be finite sets of states, of characters, and of labels, respectively.*

A reactive multi-automaton *is a nonuple $\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$, where*

- *$(Q, \Sigma, L, q_0, \delta, F)$ is a* multi-automaton *(called the* multi-automaton underlying *$\mathcal{R}$), with $q_0 \in Q$ (initial state), $F \subseteq Q$ (set of final states), and $\delta \subseteq Q \times \Sigma \times L \times Q$ (transition relation);*
- *$T^+, T^- \subseteq \delta \times \delta$ are the sets of activation and deactivation reactive links;*
- *$\overline{\delta} \subseteq \delta$ is the set of initially active links (initial transition relation).*

Definition 3 (Nondeterministic runs). *Let $\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$ be a reactive multi-automaton and let $S = s_0 s_1 \cdots s_{n-1}$ be a word on the alphabet Σ.*

The nondeterministic run of $\mathcal{R}$ over S is a sequence of pairs (Q_h, δ_h), for $h = 0, \ldots, n$, where $Q_h \subseteq Q$ and $\delta_h \subseteq \delta$ are respectively the set of active states *and the set of* active transitions *at step h, where, for $h = 0$,*

$$(Q_0, \delta_0) := \big(\{q_0\}, \overline{\delta}\big)$$

and, recursively, for $0 < k \leq n$,

$$\begin{aligned} Q_h &:= \big\{q \mid (r, s_{h-1}, \ell, q) \in \delta_{h-1}, \text{ for some } r \in Q_{h-1}, \ell \in L\big\} \\ \delta_h &:= \delta_{h-1}^{(\varphi_{h-1},T^+,T^-)}, \end{aligned}$$

where $\varphi_{h-1} := \{(r, s_{h-1}, \ell, q) \mid (r, s_{h-1}, \ell, q) \in \delta_{h-1} \text{ and } r \in Q_{h-1}\}$ and $\delta_{h-1}^{(\varphi_{h-1},T^+,T^-)}$ is the result of a switch reactive transformation applied to δ_{h-1}, relative to φ_{h-1}, T^+, T^-.

We say that the word S is accepted *by $\mathcal{R}$ provided that the nondeterministic run $\big\langle (Q_0, \delta_0), (Q_1, \delta_1), \ldots, (Q_n, \delta_n)\big\rangle$ of $\mathcal{R}$ over S is such that $Q_n \cap F \neq \emptyset$.*

Remark 4. The above definitions of switch reactive transformation, reactive multi-automaton, and nondeterministic run can be easily extended to the case in which ε-transitions are present, at least when no reactive link is allowed to have an ε-transition as its first component, which is what we will assume in the rest of the paper. In the context of multi-automata, ε-transitions take the form $(q, \varepsilon, \ell, q')$, where q, q' are states and ℓ is a label. In the nondeterministic run over a word S, if at a certain step h the ε-transitions

$$(q, \varepsilon, \ell, q'),\ (q', \varepsilon, \ell', q''),\ \ldots,\ (q^{(r-1)}, \varepsilon, \ell^{(r-1)}, q^{(r)})$$

are active and the states $q, q', \ldots, q^{(r-1)}$ are also active, the state $q^{(r)}$ will become active at step $h + 1$, independently of the $(h + 1)$-st character of S.

In view of the above observation, it is not hard to extend formally Definitions 2 and 3 to the case in which ε-transition are allowed.

4.2 The Abelian Reactive Multi-Automaton

Next we define the *abelian reactive multi-automaton* for a given pattern p of length m over an alphabet Σ, which accepts all and only the $\frac{m!}{\prod_{c\in\Sigma}(pv_{\mathsf{p}}[c])!}$ distinct permutations of p, where pv_{p} is the Parikh vector of p.

Definition 5 (Abelian Reactive Multi-Automaton). *Let* p *be a pattern of length* m *over an alphabet* Σ *and let* $\langle b_0, b_1, \ldots, b_{k-1}\rangle$ *be the sequence of the distinct characters occurring in* p*, ordered by their first occurrence. The* abelian reactive multi-automaton *(ARMA) for* p *is the reactive multi-automaton with* ε*-transitions*

$$\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$$

such that

- $Q = \{q_0, q_1, \ldots, q_k, \omega\}$ *is the set of states, where* q_0 *is the initial state and* ω *is a special state called the* overflow state*;*
- $F = \{q_k\}$ *is the set of final states;*
- $L = \{\ell_0, \ell_1, \ldots, \ell_{m-1}\}$ *is a set of labels of size* m*;*
- *the transition relation* δ *of* $\mathcal{R}$ *and its subset* $\overline{\delta} \subseteq \delta$ *of the links initially active (initial transition relation) are defined as follows*

$$\begin{aligned}
\delta \;:=\; & \{(q_i, \varepsilon, \ell_0, q_{i+1}) \mid 0 \le i < k\} && \textit{(ε-transitions)}\\
& \cup \{(q_0, p[i], \ell_i, q_0) \mid 0 \le i < m\} && \textit{(self-loops)}\\
& \cup \{(q_0, c, \ell_0, \omega) \mid c \in \Sigma\} && \textit{(overflow transitions)}\\
& \cup \{(\omega, c, \ell_0, \omega) \mid c \in \Sigma\} && \textit{(overflow self-loops)}\\
\overline{\delta} \;:=\; & \{(q_0, c, \ell_{\lambda(c)}, q_0) \mid c \in \Sigma_{\mathsf{p}}\}\\
& \cup \{(q_0, c, \ell_0, \omega) \mid c \in \Sigma \setminus \Sigma_{\mathsf{p}}\}\\
& \cup \{(\omega, c, \ell_0, \omega) \mid c \in \Sigma\}
\end{aligned}$$

- *the sets* T^+ *and* T^- *of activation and deactivation reactive links are defined as follows*

$$\begin{aligned}
T^+ \;:=\; & \{((q_0, \mathsf{p}[\rho(b_i)], \ell_{\rho(b_i)}, q_0), (q_i, \varepsilon, \ell_0, q_{i+1})) \mid 0 \le i < k\}\\
& \cup \{((q_0, \mathsf{p}[\rho(b_i)], \ell_{\rho(b_i)}, q_0), (q_0, \mathsf{p}[\rho(b_i)], \ell_{\rho(b_i)}, \omega)) \mid 0 \le i < k\}\\
& \cup \{((q_0, \mathsf{p}[i], \ell_i, q_0), (q_0, \mathsf{p}[\nu(i)], \ell_{\nu(i)}, q_0)) \mid 0 \le i < m \text{ and } i \ne \rho(p_i)\}\\
T^- \;:=\; & \{((q_0, p[i], \ell_i, q_0), (q_0, p[i], \ell_i, q_0)) \mid 0 \le i < m\}\,.
\end{aligned}$$

Fig. 1 shows the general structure of a portion of an abelian reactive automaton, whereas Fig. 2 shows the complete abelian reactive automaton for the pattern $P = \mathsf{acca}$, up to deactivation reactive links, which are not shown.

The following property states that the size of the abelian reactive automaton for a given pattern p of length m is linear in the size of p and of the underlying alphabet. This contrasts with the $\mathcal{O}(2^m)$ size of the minimal standard automaton accepting the same language.

Property 6. The abelian reactive automaton for a pattern p of length m, with $k \le m$ distinct characters, over an alphabet of size σ has size $\mathcal{O}(m + \sigma)$. Specifically it has $k + 2$ states, $k + m + 2\sigma$ transitions, and $2m + k$ reactive links. In addition, it can be constructed and initialized in $\mathcal{O}(m + \sigma)$ time and space.

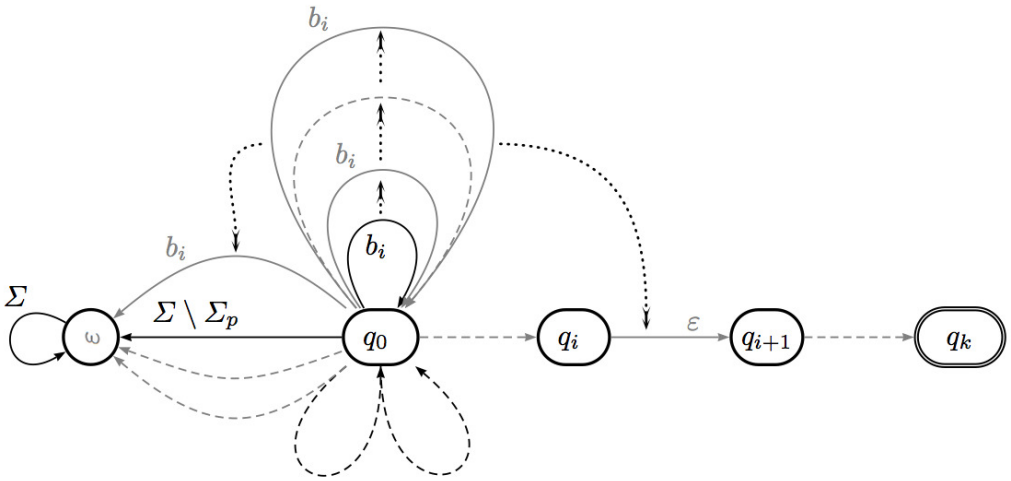


Figure 1. A portion of the general structure of an abelian reactive automaton. Standard transitions in δ are represented with solid and dashed lines, reactive links in T^+ are represented with dotted lines while reactive links in T^- are not represented. Non active links are represented in gray color.

Given a pattern p of length m with k distinct characters $b_0, b_1, \ldots, b_{k-1}$ (ordered by their first occurrence in p), the abelian reactive multi-automaton for p contains $k+1$ 'ordinary' states $q_0, q_1, \ldots, q_k$ and a path of k consecutive ε-transitions $(q_i, \varepsilon, \ell_0, q_{i+1})$, for $i = 0, 1, \ldots, k-1$, starting from the initial state q_0 and ending on its final state q_k. For $i = 0, 1, \ldots, k-1$, we will refer to $(q_i, \varepsilon, \ell_0, q_{i+1})$ as the *ε-transition of the automaton for the character b_i*. Initially, all such transitions are non-active.[4]

An additional state ω, named *overflow state*, is used to detect when the number of occurrences of a character in the current text window exceeds its multiplicity in the pattern. For each character c in the alphabet, the automaton contains a transition labeled by c from the initial state to the overflow state, called the *overflow transition for c*, and from the overflow state to itself, called the *overflow self-loop for c*. Initially, all overflow self-loops and all overflow transitions for the characters *not* occurring in the pattern are active,[5] whereas the overflow transitions for the characters occurring in the pattern are non-active.

For each character c occurring in the pattern p with multiplicity m_c (and, specifically, at positions $0 \leq h_0 < h_1 < \cdots < h_{m_c-1} < m$), the automaton contains also a set $\mathfrak{M}_c := \{(q_0, c, \ell_{h_i}, q_0) | i = 0, 1, \ldots, m_c - 1\}$ (called the *monad of c*) of m_c self-loops labeled by c, from state q_0 into itself. Initially, only the first self-loop in $\mathfrak{M}_c$, corresponding to the leftmost occurrence of c in the pattern, is active, whereas the remaining ones are all non-active. Each self-loop in $\mathfrak{M}_c$ has a deactivation reactive link pointing to itself. In addition, each of the first $m_c - 1$ self-loops in $\mathfrak{M}_c$ has an activation reactive link pointing to the next self-loop in the monad, whereas the last self-loop in $\mathfrak{M}_c$ has two activation reactive links, one pointing to the overflow transition for c and one pointing to the ε-transition relative to c.

[4] As we will see, the final state becomes reachable only when all such ε-transitions have been activated during the recognition process.

[5] In fact, all overflow self-loops and all overflow transitions for the characters *not* occurring in the pattern remain active during the whole recognition process.

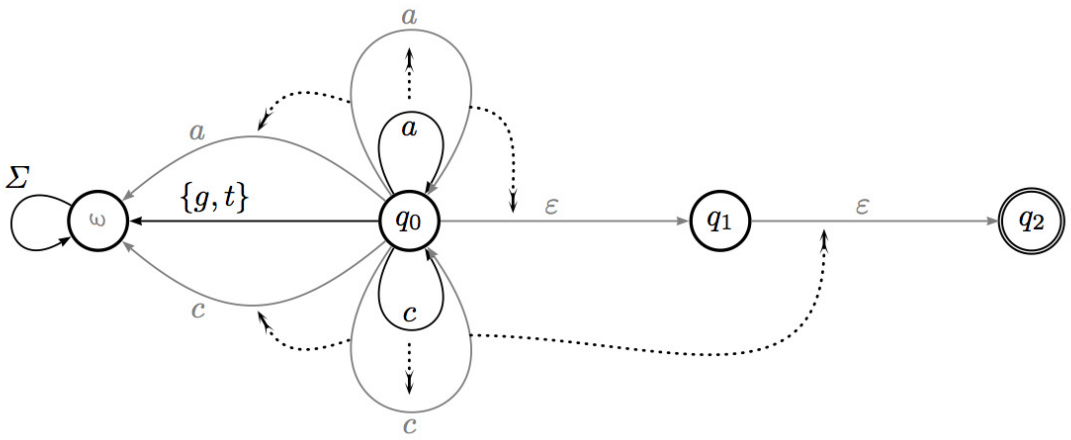


Figure 2. The complete abelian reactive automaton for the pattern $P = \mathsf{acca}$ over the DNA alphabet $\Sigma = \{a, c, g, t\}$. Standard transitions are represented with solid lines while reactive links in T^+ are represented with dashed lines. Reactive links in T^- are not represented. Non active transitions are represented in gray color.

4.3 Correctness

Next we show that the language accepted by the abelian reactive multi-automaton for a pattern p is exactly the set of all the permutations of p.

As in the previous section, let p be a pattern of length m with k distinct characters $b_0, b_1, \ldots, b_{k-1}$ (ordered by their first occurrence in p) and let

$$\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, \{q_k\}\big)$$

be the ARMA for p, with $Q = \{q_0, q_1, \ldots, q_k, \omega\}$ and $L = \{\ell_0, \ell_1, \ldots, \ell_m\}$. In addition, let s be an input string to be recognized by $\mathcal{R}$.

To begin with, we observe that as soon as an overflow transition is followed (in which case we say that an *overflow condition* has occurred), the computation gets trapped in the overflow state ω, so that q_0 is no longer active and the final state q_k cannot be reached anymore. As we will soon see, this happens when it is detected that s contains some character whose multiplicity in s exceeds that in the pattern p.

As long as the initial state q_0 is active, the transitions in the monad $\mathfrak{M}_b$ of b, for each character b in p, allow one to count the number of occurrences of b which have been read so far from the string s, when this number does not exceed the multiplicity m_b of b in p. Specifically, as a result of the interplay of the deactivation and activation reactive links on each of the first $m_b - 1$ transitions of the monad, when exactly $0 \leq i < m_b$ occurrences of b have been read from the string s, it turns out that $(q_0, b, \ell_{h_i}, q_0)$ is the only active transition in the monad $\mathfrak{M}_b$, where $0 \leq h_0 < h_1 < \cdots < h_{m_b-1} < m$ are the positions of the occurrences of b in p in increasing order. In addition, just after the m_b-th occurrence of b is read from the string s, all transitions in $\mathfrak{M}_b$ are non-active (and remain so for the rest of the recognition process), whereas the overflow transition and the ε-transition for b (which initially were non-active) become active and stay active until the end. Thus, if a further occurrence of b is read, the overflow transition for b is followed, leading to the overflow state ω, where the computation gets trapped.

Since the overflow transition for any character not occurring in p is active for the whole recognition process, as soon as an occurrence of a character not in p is found in s, the overflow transition associated to it is followed, leading again to the overflow state ω. This corresponds to having an empty monad for each character not occurring in p.

The above considerations allow us to conclude that if the string s contains any character whose multiplicity in s exceeds that in p, then the recognition process gets trapped in the overflow state ω, so that s is correctly rejected by the automaton $\mathcal{R}$.

On the other hand, if the multiplicity of no character in s exceeds that in p, then the state q_0 remains active until the end of the recognition process by $\mathcal{R}$. However, at termination, the accepting state q_k is active only if all the ε-transitions from q_0 to q_k (initially non-active) have been activated. As seen above, since q_0 is always active, this happens only if the string s and the pattern p contain the same characters and each of them occurs in s and in p with the same multiplicity; in other words, only if s is a permutation of p.

In conclusion, the language accepted by the ARMA $\mathcal{R}$ for p is the set of all the permutations of p.

4.4 The Algorithm

The algorithm that we present in this section makes use of the abelian reactive multi-automaton defined above for locating all occurrences of the permutations of a given pattern p of length m in a text t of length n.

In the preprocessing phase, the algorithm computes in $\mathcal{O}(m+\sigma)$ time and space the Parikh vector pv_{p} of the pattern and constructs the corresponding reactive multi-automaton.[6]

The algorithm works by sliding a window of size m over the text. At start, the left ends of the window and of the text are aligned. An *attempt* consists in checking whether the current window is a permutation of the pattern. This is done by executing the ARMA for the pattern over the window text. When the whole window has been read (or as soon as an overflow condition occurs) the window is shifted to the right of the last character examined. The attempts take place in sequence, until the right end of the window goes past the right end of the text.

Let us consider a generic attempt at position s of the text t, so that the current window is the substring $\mathsf{t}[s\,..\,s+m-1]$. At the beginning of the attempt, the automaton is initialized in time $\mathcal{O}(m)$. Then, during the attempt, the algorithm scans the window from right to left, while executing the corresponding automaton transitions.

If the whole text window has been scanned and no overflow condition has occurred, an occurrence of a permutation of the pattern is reported at position s. In this case the window is advanced by one position to the right.

On the other hand, if an overflow condition occurs while reading the character at position j in the text, with $s \leq j < s+m$, then the substring $\mathsf{t}[j\,..\,s+m-1]$ cannot be a permutation of the pattern, as it contains too many occurrences of the character $\mathsf{t}[j]$. Thus, it is safe to shift the window by $j-s+1$ positions to the right.

Each attempt takes $\mathcal{O}(m)$ worst-case time. Since the minimum advancement performed at the end of each attempt is by one position, the worst-case time complexity of the whole algorithm is $\mathcal{O}(nm)$.

[6] The construction of the reactive multi-automaton is straightforward and details have been omitted.

```
BAM(p, m, t, n, Σ)
 1. for each c ∈ Σ do M[c] ← pv_p[c] ← 0
 2. I ← F ← sh ← 0
 3. for i ← 0 to m − 1 do pv_p[p[i]] ← pv_p[p[i]] + 1
 4. for each c ∈ Σ do
 5.     if pv_p[c] > 0 then
 6.         M[c] ← M[c] | (1 ≪ sh)
 7.         I ← I | (((1 ≪ log m) − pv_p[c] − 1) ≪ sh)
 8.         F ← F | (1 ≪ (sh + log m))
 9.         sh ← sh + log m + 1
10. F ← F | (1 ≪ sh)
11. for each c ∈ Σ do
12.     if pv_p[c] = 0 then M[c] ← M[c] | (1 ≪ sh)
13. s ← 0
14. while s ≤ n − m do
15.     D ← I; j ← s + m − 1
16.     while j ≥ s do
17.         D ← D + M[t[j]]
18.         if (D & F) then break
19.         j ← j − 1
20.     if j < s then
21.         OUTPUT(s)
22.         s ← s + 1
23.     else s ← j + 1
```

Figure 3. The Bit-Parallel Abelian Matcher for the abelian pattern matching problem (BAM).

4.5 An Efficient Bit-Parallel Simulation

In this section we show how to simulate efficiently the abelian reactive multi-automaton for an input pattern p (cf. Definition 5), by using the bit-parallelism technique [2].

Let again $b_0, b_1, \ldots, b_{k-1}$ be the distinct characters in p.

The underlying idea is to associate a counter to each distinct character in p, plus a single 1-bit counter for the remaining characters of the alphabet which do not occur in p, maintaining them in the same computer word. In particular, the counter associated to the character b_i in p, for $i = 0, 1, \ldots, k-1$, will be represented by a group of l_i bits, where $l_i := \lceil \log(pv_{\mathsf{p}}[b_i]) \rceil + 1$. These are just enough to allocate the multiplicity $pv_{\mathsf{p}}[b_i]$ of b_i in p, plus an extra bit called the *i-th overflow bit*. Whenever an occurrence of the character b_i is read in the current text window (which, as before, is scanned backwards), its counter is incremented. Initially, the counter for b_i is set to the value $2^{l_i} - pv_{\mathsf{p}}[b_i] - 1$, so that its overflow bit is 0 and it remains so for up to $pv_{\mathsf{p}}[b_i]$ increments. Hence, the overflow bit gets set only when the $(pv_{\mathsf{p}}[b_i] + 1)$-st occurrence of b_i is encountered in the text window, if it exists, at which point it becomes clear that the text window cannot be a permutation of the pattern p. Likewise, the 1-bit counter reserved for all the characters not occurring in p is initially null and it gets set as soon as any character not in p is encountered in the text window, at which point, again, it becomes clear that the text window cannot be a permutation of the pattern p. By suitably masking the computer word allocating all the counters, it is possible to check in a single pass whether the character of the text window that has just been read has caused any of the $k+1$ overflow bits to be set. If this is the case, the window text is advanced just past the last character read. Otherwise, when the current text window has been scanned completely and no overflow bit has been set, a matching is reported and the window text is advanced one position to the right.

The resulting algorithm, named Bit-Parallel Abelian Matcher (BAM) is shown in Fig. 3. It works in a similar way as the ARMA algorithm.

During the preprocessing phase (lines 4-12), for each distinct character b_i occurring in p, a bit mask $M[b_i]$ of $l+1$ bits is computed, where

$$l := \sum_{i=0}^{k-1} l_i \qquad \text{and} \qquad M[b_i] := 1 \ll \Big(\sum_{j=0}^{i-1} l_j \Big) .$$

The bit mask $M[b_i]$ is then used in line 17 to increment the counter in D associated to the character b_i.

Two additional bit masks of $l+1$ bits are used: the bit mask I, which contains the initial values for each counter, and the bit mask F, whose bits set are exactly the overflow bits. These are defined by

$$I := \sum_{i=0}^{k-1} \Big[\Big(2^{l_i} - pv_{\mathsf{p}}[b_i] - 1 \Big) \ll \sum_{j=0}^{i-1} l_j \Big] \quad \text{and} \quad F := \sum_{i=0}^{k-1} \Big[1 \ll \Big(\sum_{j=0}^{i} l_j - 1 \Big) \Big] .$$

Let us consider a generic attempt at position s of the text (lines 14-23), so that the current text window is the substring $\mathsf{t}[s \,..\, s+m-1]$. At the beginning of each attempt, a bit mask D of $l+1$ bits (intended to represent the Parikh vector of the text window) is initialized to I (line 15). Then, during the attempt, the window is read character by character, proceeding from right to left (lines 16-19). When reading the character $\mathsf{t}[j]$ of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathsf{t}[j]]$ (line 17).

The attempt stops when the left end of the window is reached or when an overflow bit in D is set. In the first case, an occurrence is reported at position s and the window is advanced to the right by one position (lines 20-22). In the second case, i.e., when the counter update for a character $\mathsf{t}[j]$ has set an overflow bit in D (and therefore $D \& F \neq 0$ holds), the substring $\mathsf{t}[j \,..\, s+m-1]$ cannot be involved in any match, as it contains too many occurrences of the character $\mathsf{t}[j]$, and therefore it is safe to shift the window to the right by $j - s + 1$ positions (line 23).

As in the case of the ARMA algorithm, each attempt takes $\mathcal{O}(m)$-worst case time and at most n attempts take place during the whole execution. Thus the worst-case time complexity of the BAM algorithm is $\mathcal{O}(nm)$, whereas the space requirement for maintaining a bit mask for each character of the alphabet is $\mathcal{O}(\sigma)$.

So far we have implicitly assumed that $l + 1 \leq w$, where w is the size of a computer word, so that each of the vectors D, I, F, and $M[b_i]$, for b_i in p, fits in a single computer word.

When $l + 1 > w$, we must content ourselves to maintain the counters only for a proper selection Σ'_{p} of the set of characters occurring in p. In this case, when a match relative to the characters in Σ'_{p} is reported, an additional verification phase must be run, in order to discard possible *false positives*.

5 Experimental Results

In this section we evaluate the performance of the bit-parallel simulation BAM described in the previous section and compare it with some standard solutions known in literature. In particular we compare the performances of the following three algorithms: the prefix based algorithm due to Grabowsky *et al.* (GFG) [14], the algorithm using the suffix based approach (SBA) [11], and the Bit-parallel Abelian Matcher (BAM) described in Section 4.5.

m	GFG	SBA	BAM
2	**23.56**	39.20	27.03
4	**23.56**	33.27	23.17
8	23.54	27.54	**19.01**
16	23.49	24.05	**16.21**
32	23.52	23.78	**15.63**
64	23.50	25.33	**16.12**
128	23.57	28.74	**17.69**
256	23.53	33.14	**19.63**

m	GFG	SBA	BAM
2	23.08	**18.07**	12.51
4	23.00	**15.39**	10.36
8	22.96	13.67	**9.40**
16	23.03	11.91	**8.44**
32	23.04	9.58	**7.16**
64	23.01	8.46	**6.64**
128	22.97	7.82	**6.49***
256	22.96	7.84	**7.69***

Table 1. Experimental results on a genome sequence (on the left) and a on a protein sequence (on the right). An asterisk symbol (*) indicates those runs where false positives have been detected. All best results have been boldfaced.

All algorithms have been implemented in C and compiled with the `GNU C Compiler 4.2.1`, using the optimization option `-O3`. The experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3. The three algorithms have been compared in terms of their running times, including any preprocessing time, measured with a hardware cycle counter, available on modern CPUs.

In our tests we used a genome sequence, with an alphabet of size $\sigma = 4$ (Table 1, on the left) and a protein sequence, with an alphabet of size $\sigma = 20$ (Table 1, on the right), both of 4 MB length.[7] For each input file, we have generated sets of 500 patterns of fixed length m randomly extracted from the text, where $m \in \{2, 4, 8, 16, 32, 64, 128, 256\}$, and reported the mean time over the 500 runs, expressed in milliseconds.

From the experimental results it turns out that the GFG algorithm has a linear behavior in practice and is almost insensitive to the size of the pattern, whereas the algorithms based on a backward approach, such as SBA and BAM, show a sublinear behavior although their theoretical worst-case time complexity is quadratic.

In all cases, when the pattern is longer than 4 characters, the BAM algorithm outperforms the other two algorithms.

The SBA and the BAM algorithms improve their performances in the case of larger alphabets and turn out to be the best solutions when searching for a protein sequence. In this case their performances are up to 3 times faster than the GFG algorithm.

In the case of the protein sequence, we observed some cases where false positive occurrences were detected by an additional verification. Such events are indicated in Table 1 with an asterisk (*). However, in all cases the average number of additional verification runs, for each text position, turned out to be less than 10^{-4}.

6 Conclusions

We have presented a new approach to solve the abelian pattern matching problem for strings which is based on a reactive multi-automaton with only $\mathcal{O}(k)$ states and $\mathcal{O}(m)$ transitions. We have also proposed an efficient simulation of such automaton using bit-parallelism. Our solution is based on a backward approach and, despite its quadratic worst-case time complexity, shows a sublinear behavior in practical cases.

[7] The text buffers are described and available for download at the SMART web page (`http://www.dmi.unict.it/~faro/smart/`).

References

1. A. Amir, A. Apostolico, G. M. Landau, G. Satta: Efficient Text Fingerprinting Via Parikh Mapping. Journal of Discrete Algorithms, 1(5–6) 2003, pp. 409–421.
2. R. A. Baeza-Yates, G. H. Gonnet: A new approach to text searching. Commun. ACM, 35 (10) 1992, pp. 74–82.
3. R. A. Baeza-Yates, G. Navarro: New and faster filters for multiple approximate string matching. Random Struct. Algorithms 20 (1) 2002, pp. 23–49.
4. G. Benson: Composition alignment. In: WABI 2003, pp. 447–461.
5. S. Böcker: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics 23 (2) 2007, pp. 5–12. `http://dx.doi.org/10.1093/bioinformatics/btl291`
6. S. Böcker: Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. Journal of Computational Biology 11 (6) 2004, pp. 1110–1134.
7. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. 23 (2) 2012, pp. 357–374.
8. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: On approximate jumbled pattern matching in strings. Theory Comput. Syst. 50 (1) 2012, pp. 35–51.
9. D. Cantone, S. Cristofaro, S. Faro: Efficient matching of biological sequences allowing for non-overlapping inversions. In: CPM 2011, pp. 364–375.
10. M. Crochemore, D. M. Gabbay: Reactive automata. Inf. Comput., 209(4) 2011, pp. 692–704.
11. E. Ejaz: Abelian pattern matching in strings. Ph.D. Thesis, Dortmund University of Technology (2010), `http://d-nb.info/1007019956`.
12. R. Eres, G. M. Landau, L. Parida: Permutation Pattern Discovery in Biosequences. Journal of Computational Biology, 11(6) 2004, pp. 1050–1060.
13. D. M. Gabbay: Pillars of computer science. Springer-Verlag 2008. Ch. Introducing reactive Kripke semantics and arc accessibility, pp. 292–341.
14. S. Grabowski, S. Faro, E. Giaquinta: String matching with inversions and translocations in linear average time (most of the time). Inf. Process. Lett. 111 (11) 2011, pp. 516–520.
15. R. Grossi, F. Luccio: Simple and efficient string matching with k mismatches. Inf. Process. Lett. 33 (3) 1989, pp. 113–120.
16. R. N. Horspool: Practical fast searching in strings. Software – Practice & Experience 10 (6) 1980, pp. 501–506.
17. P. Jokinen, J. Tarhio, E. Ukkonen: A comparison of approximate string matching algorithms. Softw. Pract. Exp. 26 (12) 1996, pp. 1439–1458.
18. A. Salomaa: Counting (scattered) subwords. Bulletin of the EATCS 81 (2003), pp. 165–179.

Computing Abelian Covers and Abelian Runs

Shohei Matsuda, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
{shohei.matsuda,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. Two strings u and v are said to be *Abelian equivalent* if u is a permutation of the characters of v. We introduce two new regularities on strings w.r.t. Abelian equivalence, called *Abelian covers* and *Abelian runs*, which are generalizations of covers and runs of strings, respectively. We show how to determine in $O(n)$ time whether or not a given string w of length n has an Abelian cover. Also, we show how to compute an $O(n^2)$-size representation of (possibly exponentially many) Abelian covers of w in $O(n^2)$ time. Moreover, we present how to compute all Abelian runs in w in $O(n^2)$ time, and state that the maximum number of all Abelian runs in a string of length n is $\Omega(n^2)$.

Keywords: Abelian equivalence on strings, Parikh vectors, Abelian repetitions, covers of strings, string algorithms

1 Introduction

The study of *Abelian equivalence* of strings dates back to at least the early 60's, as seen in the paper by Erdös [6]. Two strings u, v are said to be *Abelian equivalent* if u is a permutation of the characters appearing in v. For instance, strings `aabba` and `baaba` are Abelian equivalent. Abelian equivalence of strings has attracted much attention and has been studied extensively in several contexts.

A variant of the pattern matching problem called the *jumbled pattern matching problem* is to determine whether there is a substring of an input text string w that is Abelian equivalent to a given pattern string p. There is a folklore algorithm to solve this problem in $O(n + m + \sigma)$ time using $O(\sigma)$ space, where n is the length of w, m is the length of p, and σ is the alphabet size. Assuming $m \leq n$ and all characters appear in w, the algorithm runs in $O(n)$ time and $O(\sigma)$ space. The indexed version of the jumbled pattern matching problem is more challenging, where the task is to preprocess an input text string w so that, given a query pattern string p, we can quickly determine whether or not there is a substring of w that is Abelian equivalent to p. For binary strings, there exists a data structure which occupies $O(n)$ space and answers the above query in $O(1)$ time. Burcsi et al. [2] and Moosa and Rahman [16] independently developed an $O(n^2/\log n)$-time algorithm to construct this data structure, and later Gagie et al. [9] showed an improved $O(n^2/\log^2 n)$-time algorithm. Very recently, Hermelin et al. [10] proposed an $n^2/2^{\Omega(\log n/\log\log n)^{\frac{1}{2}}}$-time solution to the problem for binary strings. For any constant-size alphabets, Kociumaka et al. [12] showed an algorithm that requires $O(n^2 \log^2 \log n/\log n)$ preprocessing time and $O((\log n/\log\log n)^{2\sigma-1})$ query time. Amir et al. [1] showed lower bounds on the indexing version of the jumbled pattern matching problem under a 3SUM-hardness assumption.

Abelian periodicity of strings has also been extensively studied in string algorithmics. A string w is said to have a full Abelian period if w is a concatenation $w_1 \cdots w_k$ of k Abelian equivalent strings $w_1, \ldots, w_k$ with $k \geq 2$, and the length of w_1 is called a full Abelian period of w. A string w is said to have an Abelian period if $w = yz$,

Shohei Matsuda, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Computing Abelian Covers and Abelian Runs, pp. 43–51.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

where y has some full Abelian period d, and z is a non-empty string shorter than d such that the number of each character a contained in z is no more than that contained in the prefix $y[1..d]$ of length d of y. A string w is said to have a weak Abelian period d if $w = xy$, where y has some Abelian period d, and x is a non-empty string shorter than d such that the number of each character a contained in x is no more than that contained in the prefix $y[1..d]$ of length d of y. Fici et al. [7] proposed an $O(n \log \log n)$-time algorithm to compute all full Abelian periods and $O(n^2)$-time algorithm to compute all Abelian periods for a given string of length n. Recently, Kociumaka et al. [13] showed an optimal $O(n)$-time algorithm to compute all full Abelian periods, and an improved $O(n(\log \log n + \log \sigma))$-time algorithm to compute all Abelian periods, where σ is the alphabet size. Fici et al. [8] presented an $O(n^2\sigma)$-time algorithm to compute all weak Abelian periods, and later Crochemore et al. [3] gave an improved $O(n^2)$-time solution to the problem.

In the field of word combinatorics, Erdös [6] posed a question whether there exists an infinitely long string which contains no Abelian squares. A substring s of a string w is called an Abelian square if $s = s_1 s_2$ such that s_1 and s_2 are Abelian equivalent. Entringer et al. [5] proved that any infinite word over a binary alphabet contains arbitrary long Abelian squares. On the other hand, Pleasants [18] showed a construction of an infinitely long string which contains no Abelian squares over an alphabet of size 5, and later, Keränen [11] showed a construction over an alphabet of size 4. An interesting question in the field of string algorithmics is how efficiently we can compute the Abelian repetitions that occur in a given string of finite length n. Cummings and Smyth [4] presented an algorithm to compute all Abelian squares in $O(n^2)$ time. They also showed that there exist $\Omega(n^2)$ Abelian squares in a string of length n. Crochemore et al. [3] showed another $O(n^2)$-time algorithm to compute all squares in a string of length n.

In this paper, we introduce two new regularities on strings w.r.t. Abelian equivalence, called *Abelian covers* and *Abelian runs*, which are generalizations of covers [15] and runs [14] of strings, respectively, and we propose non-trivial algorithms to compute these new string regularities. A set C of intervals is called an Abelian cover of a string w if the substrings corresponding to the intervals in C are all Abelian equivalent, and every position in w is contained in at least one interval in C. We show that, given a string w of length n, we can determine whether or not w has an Abelian cover in optimal $O(n)$ time. Also, we present an $O(n^2)$-time algorithm to compute an $O(n^2)$-size representation of all (possibly exponentially many) Abelian covers of w. A substring s of w is said to be an Abelian run of w if s is a maximal substring which has a weak Abelian period. As a direct consequence from the result by Cummings and Smyth [4], it is shown that the maximum number of all Abelian runs in a string of length n is $\Omega(n^2)$. Then, we propose an $O(n^2)$-time algorithm to compute all Abelian runs in a given string of length n.

2 Preliminaries

Let $\Sigma = \{c_1, \ldots, c_\sigma\}$ be an ordered *alphabet*. We assume that for each $c_i \in \Sigma$, its rank i in Σ is already known and can be computed in constant time. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, strings x, y, and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The i-th character of a string w of length n is denoted by $w[i]$ for $1 \leq i \leq n$. For $1 \leq i \leq j \leq n$, let $w[i..j] = w[i] \cdots w[j]$,

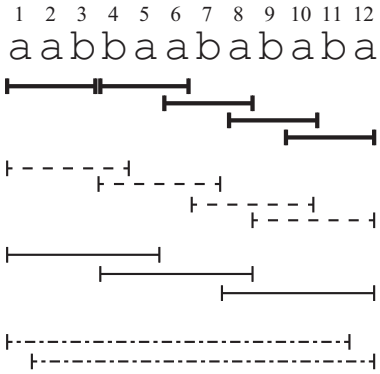


Figure 1. String aabbaababababa over a binary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ has an Abelian cover $\{[1,3],[4,6],[6,8],[8,10],[10,12]\}$ of length 3 with Parikh vector $\langle 2,1\rangle$, an Abelian cover $\{[1,4],[4,7],[7,10],[9,12]\}$ of length 4 with Parikh vector $\langle 2,2\rangle$, an Abelian cover $\{[1,5],[4,8],[8,12]\}$ of length 5 with Parikh vector $\langle 3,2\rangle$, and an Abelian cover $\{[1,11],[2,12]\}$ of length 11 with Parikh vector $\langle 6,5\rangle$. We remark that this string has other Abelian covers than the above ones.

i.e., $w[i..j]$ is the substring of w starting at position i and ending at position j in w. For convenience, let $w[i..j] = \varepsilon$ if $j < i$. For any $0 \le i \le n$, strings $w[1..i]$ and $w[i..n]$ are called prefixes and suffixes of w, respectively.

For any string w of length $n \ge 2$, a set $I = \{[b_1, e_1], \ldots, [b_{|I|}, e_{|I|}]\}$ of intervals is called a *cover* of w if $\bigcup_{1 \le k \le |I|}[b_k, e_k] = [1, n]$ and $[b_k, e_k] \neq [1, n]$ for every $1 \le k \le |I|$. Whenever we write $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ for a cover C of a string w, then we assume that $b_j < b_{j+1}$ for all $1 \le k < |C|$.

Two strings $v, w \in \Sigma^*$ are said to be *Abelian equivalent* if v is a permutation of the characters in w. A Parikh vector [17] of a string $w \in \Sigma^*$, denoted P_w, is an array of length σ such that for any $1 \le i \le \sigma$, $P_w[i]$ stores the number of occurrences of character c_i in w. Let $\preceq$ be a partial order of Parikh vectors P_v and P_w for any strings $v, w \in \Sigma^*$ such that

$$P_v = P_w \text{ if } P_v[i] = P_w[i] \text{ for all } 1 \le i \le \sigma, \text{ and}$$
$$P_v \prec P_w \text{ if } P_v \neq P_w \text{ and } P_v[i] \le P_w[i] \text{ for all } 1 \le i \le \sigma.$$

For instance, for strings $v = \mathtt{aababc}$ and $w = \mathtt{baba}$ over an ordered alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $P_v = [3, 2, 1]$ and $P_w = [2, 2, 0]$, and therefore $P_w \prec P_v$. Clearly, strings v, w are Abelian equivalent iff $P_v = P_w$. For any two strings $v, w \in \Sigma^*$, let $P_v \oplus P_w = P_{vw}$, namely, $(P_v \oplus P_w)[i] = P_v[i] + P_w[i]$ for each $1 \le i \le \sigma$.

A cover $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ of a string w is called an *Abelian cover* of w if $P_{w[b_1..e_1]} = P_{w[b_j..e_j]}$ for all $1 < j \le |C|$. Clearly $e_j - b_j = e_1 - b_1$ holds for all $1 < j \le |C|$. The *length* and *size* of an Abelian cover $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ of a string are $e_1 - b_1 + 1$ and $|C|$, respectively. See Figure 1 for examples of Abelian covers of a string.

A non-empty substring u of a string w is called an *Abelian repetition* with period d if $|u|$ is a multiple of an integer d $(1 \le d \le \frac{|u|}{2})$ and $P_{u[(k-1)d+1..kd]} = P_{u[kd+1..(k+1)d]}$ for all $1 \le k < \frac{|u|}{d}$. If $d = \frac{|u|}{2}$, then u is called an *Abelian square* of w. A substring $w[i..j]$ of a string w is called a *maximal Abelian repetition* of w if $w[i..j]$ is a non-extensible Abelian repetition with period d in w, namely, if $w[i..j]$ is an Abelian repetition satisfying (1) $P_{w[i-d..i-1]} \neq P_{w[i..i+d-1]}$ or $i - d < 0$ and (2) $P_{w[j-d+1..j]} \neq P_{w[j+1..j+d]}$ or $j + d > n$. A substring $w[i - h..j + h']$ of a string w is called an *Abelian run*

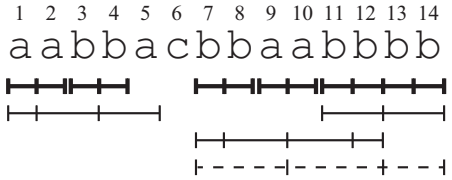


Figure 2. String aabbacbbaabbbb over a ternary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ has Abelian runs $(0, 1, 2, 1, 0)$, $(0, 3, 4, 1, 0)$, $(0, 7, 8, 1, 0)$, $(0, 9, 10, 1, 0)$, and $(0, 11, 14, 1, 0)$ of period 1, Abelian runs $(1, 2, 5, 2, 0)$, $(1, 8, 11, 2, 1)$, and $(0, 11, 14, 2, 0)$ of period 2, and an Abelian run $(0, 7, 12, 3, 2)$ of period 3.

of w if $w[i..j]$ is a maximal Abelian repetition with period d of w and $h, h' \geq 0$ are the largest integers satisfying $P_{w[i-h..i-1]} \prec P_{w[i..i+d]}$ and $P_{w[j+1..j+h']} \prec P_{w[j-d..j]}$, respectively. Each Abelian run $w[i-h..j+h']$ of period d in w is represented by a 5-tuple (h, i, j, d, h'), where h and h' are called the left hand and the right hand of the Abelian run, respectively. See Figure 2 for examples of Abelian runs in a string.

In this paper, we consider the following problems.

Problem 1 (Abelian cover existence). Given a string w, determine whether or not w has an Abelian cover.

Problem 2 (All Abelian covers). Given a string w, compute all Abelian covers of w.

Problem 3 (All Abelian runs). Given a string w, compute all Abelian runs in w.

3 Algorithms

In this section, we present our algorithms to solve the problems stated in the previous section. For simplicity, we assume that all characters in Σ appear in a given string w of length n, which implies $\sigma \leq n$.

3.1 Abelian cover existence

In this subsection, we consider Problem 1 of determining whether there exists an Abelian cover of a given string w of length n. Note that there exists an infinite sequence of strings over a binary alphabet which have no Abelian covers (e.g., $\mathtt{a}^{n-1}\mathtt{b}$ has no Abelian covers), and therefore Problem 1 is of interest. The following lemma is a key to our solution to the problem.

Lemma 4. *If there exists an Abelian cover $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ of arbitrary size for a string w, then there exists an Abelian cover C' of size exactly 2 for w.*

Proof. It is clear when $|C| = 2$. Consider the case where $|C| \geq 3$. Since C is an Abelian cover of w, $P_{w[b_1,e_1]} = P_{w[b_{|C|}..e_{|C|}]}$. This implies that $P_{w[b_1..e_1]} \oplus P_{w[e_1+1..b_{|C|}-1]} = P_{w[e_1+1..b_{|C|}-1]} \oplus P_{w[b_{|C|}..e_{|C|}]}$. Therefore, $C' = \{[b_1..b_{|C|} - 1], [e_1 + 1, e_{|C|}]\} = \{[1..b_{|C|} - 1], [e_1 + 1, |w|]\}$ is an Abelian cover of size 2 of w. (See also Figure 3.) □

Using the above lemma, we obtain the following.

Theorem 5. *Given a string w of length n, we can determine whether w has an Abelian cover or not in $O(n)$ time with $O(\sigma)$ working space.*

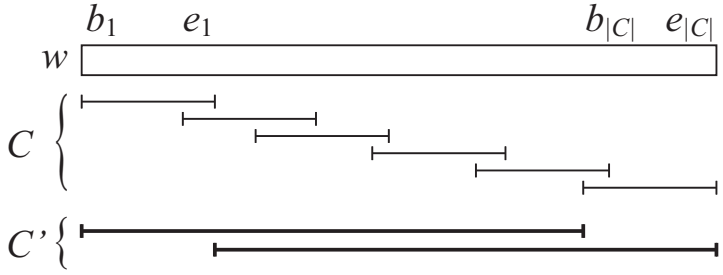


Figure 3. Illustration for Lemma 4. If a string w has a cover C, then w always has a cover C' of size 2.

Proof. By Lemma 4, Problem 1 of deciding whether there exists an Abelian cover of a given string w reduces to finding an Abelian cover of size 2 of w. Therefore, it suffices to find a prefix and a suffix of the same length ℓ such that $P_{w[1..\ell]} = P_{w[n-\ell+1..n]}$. To find such a prefix and a suffix, for each $1 \leq j \leq \lfloor \frac{n}{2} \rfloor$ in increasing order, we maintain an invariant d_j which represents the number of entries of $P_{w[1..\ell]}$ and $P_{w[n-\ell+1..n]}$ whose values differ, i.e.,

$$d_j = \{k \mid P_{w[1..j]}[k] \neq P_{w[n-j+1..n]}[k], 1 \leq k \leq \sigma\}.$$

Clearly w has an Abelian cover of size 2 iff $d_j = 0$ for some j. For any $1 \leq j \leq \lfloor \frac{n}{2} \rfloor$, let $w[j] = c_s$ and $w[n-j+1] = c_t$. We can update $P_{w[1..j-1]}$ (resp. $P_{w[n-j..n]}$) to $P_{w[1..j]}$ (resp. $P_{w[n-j+1..n]}$) in $O(1)$ time, increasing the value stored in the sth entry (resp. the tth entry) by 1. Also, d_j can be computed in $O(1)$ time from d_{j-1}, $P_{w[1..j-1]}[s]$, $P_{w[1..j]}[s]$, $P_{w[n-j..n]}[t]$, and $P_{w[n-j+1..n]}[t]$. Hence, the algorithm runs in a total of $O(n)$ time. The extra working space of the algorithm is $O(\sigma)$, due to the two Parikh vectors we maintain. □

The following corollary is immediate from Lemma 4 and Theorem 5:

Corollary 6. *We can compute the longest Abelian cover of a given string of length n in $O(n)$ time with $O(\sigma)$ working space, if it exists.*

3.2 All Abelian covers

In this subsection, we consider Problem 2 of computing all Abelian covers of a given string w of length n. Note that the number of all Abelian covers of a string can be exponentially large w.r.t. n. For instance, string a^n has $\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} 2^{n-k-1}$ Abelian covers of length at least $\lceil \frac{n}{2} \rceil$. This is because, for any $k \geq \lceil \frac{n}{2} \rceil$, the union of $\{[1,k],[n-k+1,n]\}$ and any subset of $\{[2,k+1],[3,k+2],\ldots,[n-k,n-1]\}$ is an Abelian cover of length k for a^n. Therefore, we consider to compute a "compact" representation of all Abelian covers of a given string.

Theorem 7. *Given a string w of length n, we can compute an $O(n^2)$-size representation of all Abelian covers of w in $O(n^2)$ time and $O(n)$ working space. Given a set I of s intervals sorted by the beginning positions of the intervals, the representation allows us to check if I is an Abelian cover of w in $O(s)$ time.*

Proof. For each $1 \leq \ell \leq n-1$, we compute a subset S_ℓ of positions in w such that $S_\ell = \{i \mid P_{w[i..i+\ell-1]} = P_{w[1..\ell]}, 1 \leq i \leq n-\ell+1\}$. Then, there exists an Abelian cover

of length ℓ for w iff the distance between any two adjacent positions in S_ℓ is at most ℓ. If S_ℓ satisfies the above condition, then we represent S_ℓ as a bit vector B_ℓ of length n such that $B_\ell[i] = 1$ if $i \in S_\ell$, and $B_\ell[i] = 0$ otherwise. If S_ℓ does not satisfy the above condition, then we discard it. Now, given a set I of s intervals sorted by the beginning positions of the intervals, we first check if I is a cover of w and if each interval is of equal length ℓ in a total of $O(s)$ time. If I satisfies both conditions, then we can check if I is a subset of S_ℓ in $O(s)$ time, using the bit vector B_ℓ. Using a similar method to Theorem 5, for each $1 \leq \ell \leq n-1$, S_ℓ and its corresponding bit vector B_ℓ can be computed in $O(n)$ time. Hence, the overall time complexity of the algorithm is $O(n^2)$. The working space (excluding the output) is $O(n)$, since $|S_\ell| = O(n)$ for any ℓ and $\sigma = O(n)$. □

Given a set I of s intervals, a naïve algorithm to check whether I is an Abelian cover of length ℓ requires $O(s\ell)$ time. Therefore, the solution of Theorem 7 with $O(s)$ query time is more efficient than the naïve method.

3.3 All Abelian runs

In this subsection, we consider Problem 3 of computing all Abelian runs in a given string w of length n. We follow and extend the results by Cummings and Smyth [4] on the maximum number of all maximal Abelian repetitions in a string, and an algorithm to compute them. We firstly consider a lower bound on the maximum number of Abelian runs in a string.

Lemma 8 ([4]). *String* $(\mathtt{aababbaba})^n$ *of length* $8n$ *has* $\Theta(n^2)$ *maximal Abelian repetitions (in fact maximal Abelian squares).*

Since the number of Abelian runs in a string is equal to that of maximal Abelian repetitions in that string, the following theorem is immediate:

Theorem 9. *The maximum number of Abelian runs in a string* w *of length* n *is* $\Omega(n^2)$.

Next, we show how to compute all Abelian runs in a given string.

Theorem 10. *Given a string* w *of length* n*, we can compute all Abelian runs in* w *in* $O(n^2)$ *time and space.*

Proof. We firstly compute all Abelian squares in w using the algorithm proposed by Cummings and Smyth [4]. For each $1 \leq i \leq n$, we compute a set L_i of integers such that

$$L_i = \{j \mid P_{w[i-j..i]} = P_{w[i+1..i+j+1]}, 0 \leq j \leq \min\{i, n-i\}\}.$$

Note that substring $w[i-\ell..i+\ell+1]$ is an Abelian square centered at position i iff $\ell \in L_i$. After computing all L_i's, we store them in a two dimensional array L of size $\lfloor \frac{n}{2} \rfloor \times n-1$ such that $L[\ell, i] = 1$ if $\ell \in L_i$ and $L[\ell, i] = 0$ otherwise. All entries of L are initialized *unmarked*. Then, for each $1 \leq \ell \leq n-1$, all maximal Abelian repetitions of period ℓ can be computed by in $O(n)$ time, as follows. We scan the ℓth row of L from left to right for increasing $i = 1, \ldots, n-1$, and if we encounter an unmarked entry (ℓ, i) such that $L[\ell, i] = 1$, then we compute the largest non-negative integer k such that $L[\ell, i + p\ell + 1] = 1$ for all $1 \leq p \leq k$ in $O(k)$ time, by skipping every $\ell - 1$ entries in between. This gives us a maximal Abelian repetitions with period ℓ

L

l \ i	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	0	0	0	0	0	0
2	0	0	0	0	1	1	1	1	0	0
3	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0

Figure 4. The two dimensional array L for string `caaababababac`. The maximal Abelian repetitions `aaa` of period 1 starting at position 3 is found by concatenating two Abelian squares represented by $L[1,2]$ and $L[1,3]$. The maximal Abelian repetition `ababab` of period 2 starting at position 4 is found by concatenating two Abelian squares represented by $L[2,5]$ and $L[2,7]$. The maximal Abelian repetition `bababa` of period 2 starting at position 5 is found by concatenating two Abelian squares represented by $L[2,6]$ and $L[2,8]$. Finally, the maximal Abelian repetition `aababa` of period 3 starting at position 3 is found from $L[3,5]$ (this is not extensible to the right). Every concatenation procedure (represented by an arrow) starts from an unmarked entry, and once an entry is involved in computation of a maximal Abelian repetition, it gets marked. This way the algorithm runs in time linear in the size of L, which is $O(n^2)$.

starting at position $i-\ell+1$ and ending at position $i+(k+1)\ell$. After computing the largest integer k, we mark the entries $L[\ell, i+p\ell+1]$ for all $-1 \le p \le k$ in $O(k)$ time. Since each unmarked entry of the ℓth row is marked at most once and is accessed by a constant number of times, and since the above procedure starts only from unmarked entries, it takes a total of $O(n)$ time for each ℓ. Therefore, this takes a total of $O(n^2)$ time for all $1 \le \ell \le \lfloor \frac{n}{2} \rfloor$. (See also Figure 4 for a concrete example of the two dimensional array L and how to compute all maximal Abelian repetitions from L).

What remains is how to compute the left and right hands of each maximal Abelian runs. If we compute the left and right hands naïvely for all the maximal Abelian repetitions, then it takes a total of $O(n^3)$ time due to Theorem 9. To compute the left and right hands in a total of $O(n^2)$ time, we use the following property on Abelian repetitions: For each $1 \le i \le n$, let P_i be the set of positive integers such that for each $\ell \in P_i$ there exists a maximal Abelian repetition whose period is ℓ and beginning position is $i-\ell+1$. For any $1 \le j \le |P_i|$, let ℓ_j denote the jth smallest element of P_i. We process ℓ_j in increasing order of $j = 1, \ldots, |P_i|$. Let h_j denote the left hand of the Abelian run that is computed from the maximal Abelian repetition whose period is ℓ_j and beginning position is $i-\ell_j+1$. For any $1 \le j < |P_i|$, assume that we have computed the length of the left hand h_{j-1} of the maximal Abelian repetition beginning at position $i-\ell_{j-1}+1$. We are now computing the left hand h_j of the next Abelian run. There are two cases to consider:

1. If $j = 1$ or $\ell_{j-1} + h_{j-1} \le \ell_j$, then we compute the left hand h_j of the maximal Abelian repetition beginning at position $i-\ell_j+1$, by comparing the Parikh vector $P_{w[i-\ell_j-k..i-\ell_j]}$ for increasing k from 0 up to h_j+1, with the Parikh vector $P_{w[i-\ell_j+1..i]}$. This takes $O(h_j)$ time. (See also Figure 5).
2. If $\ell_{j-1} + h_{j-1} > \ell_j$, then

$$P_{w[i-\ell_{j-1}-h_{j-1}+1..i-\ell_j]} \prec P_{w[i-\ell_{j-1}-h_{j-1}+1..i-\ell_{j-1}]} \prec P_{w[i-\ell_{j-1}+1..i]} \prec P_{w[i-\ell_j+1..i]}.$$

This implies that $h_j \ge \ell_{j-1} + h_{j-1} - \ell_j$. We can compute $P_{w[i-\ell_{j-1}-h_j..i-\ell_j]}$ from $P_{w[i-\ell_{j-1}-h_{j-1}+1..i-\ell_{j-1}]}$ in $O(\ell_j - \ell_{j-1})$ time. Then, we compute the left hand h_j

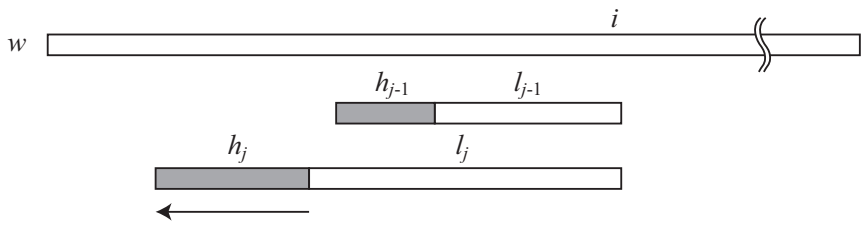


Figure 5. Illustration for Case 1 where $j = 1$ or $\ell_{j-1} + h_{j-1} \leq \ell_j$ of Theorem 10. We can compute the left hand h_j in $O(h_j)$ time by extending the substring to the left from position $i - \ell_j$.

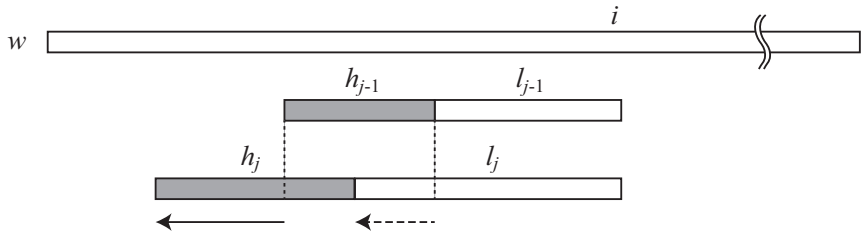


Figure 6. Illustration for Case 2 where $\ell_{j-1} + h_{j-1} > \ell_j$ of Theorem 10. In this case, we know that h_j is at least $\ell_{j-1} + h_{j-1} - \ell_j$. The Parikh vector of $w[i - \ell_{j-1} - h_j..i - \ell_j]$ can be computed in $O(\ell_j - \ell_{j-1})$ time by a scan of substring $w[i - \ell_j + 1..i - \ell_{j-1}]$ (dashed arrow). Then, we can compute the left hand h_j in a total of $O(h_j + \ell_j - h_{j-1} - \ell_{j-1})$ time by extending the substring to the left from position $i - \ell_{j-1} - h_{j-1}$ (solid arrow).

by comparing the Parikh vector $P_{w[i-\ell_{j-1}-h_{j-1}+1-k..i-\ell_j]}$ for increasing k from 0 up to $h_j + \ell_j - h_{j-1} - \ell_{j-1} + 1$. This takes $O(h_j + \ell_j - h_{j-1} - \ell_{j-1})$ time. (See also Figure 6).

Let J_i^1 and J_i^2 be the disjoint subsets of $[1, |P_i|]$ such that $j \in J_i^1$ if $\ell_j \in P_i$ corresponds to Case 1, and $j \in J_i^2$ if $\ell_j \in P_i$ corresponds to Case 2. Then, the summations $\sum_{j \in J_i^1}(h_j)$, $\sum_{j \in J_i^2}(\ell_j - \ell_{j-1})$, and $\sum_{j \in J_i^2}(h_j + \ell_j - \ell_{j-1} - h_{j-1})$ corresponding to the time costs for Cases 1 and 2 are all bounded by $O(n)$. Therefore, it takes a total of $O(n)$ time to compute the left hands of all Abelian runs that correspond to P_i, and the right hands can be computed similarly. Hence, it takes a total of $O(n^2)$ time to compute all Abelian runs in w. The working space of the algorithm is dominated by the two dimensional array L, which takes $O(n^2)$ space. □

4 Conclusions and future work

Abelian regularities on strings were initiated by Erdös [6] in the early 60's, and since then they have been extensively studied in Stringology. In this paper, we introduced new regularities on strings with respect to Abelian equivalence on strings, which we call *Abelian covers* and *Abelian runs*. Firstly, we showed an optimal $O(n)$-time $O(\sigma)$-space algorithm to determine whether or not a given string w of length n over an alphabet of size σ has an Abelian cover. As a consequence of this, we can compute the longest Abelian cover of w in $O(n)$-time. Secondly, we showed an $O(n^2)$-time algorithm to compute an $O(n^2)$-space representation of all (possibly exponentially many) Abelian covers of a string of length n. Thirdly, we presented an $O(n^2)$-time

algorithm to compute all Abelian runs in a string of length n. We also remarked that the maximum number of Abelian runs in a string of length n is $\Omega(n^2)$.

Our future work includes the following:

- The algorithm of Theorem 7 allows us to compute a shortest Abelian cover of a given string of length n in $O(n^2)$ time. Can we compute a *shortest* Abelian cover in $o(n^2)$ time?
- The algorithm of Theorem 10 requires $\Theta(n^2)$ time to compute all Abelian runs of a given string w of length n. This is due to the two dimensional array L of $\Theta(n^2)$ space. Can we compute all Abelian runs in w in optimal $O(n + r)$ time, where r is the number of Abelian runs in w?

References

1. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein: *On hardness of jumbled indexing*, in Proc. ICALP 2014 (to appear), 2014, Preprint is available at `http://arxiv.org/abs/1405.0189`.
2. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On table arrangements, scrabble freaks, and jumbled pattern matching*, in Proc. FUN 2010, 2010, pp. 89–101.
3. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, J. Pachocki, J. Radoszewski, W. Rytter, W. Tyczynski, and T. Walen: *A note on efficient computation of all Abelian periods in a string.* Inf. Process. Lett., 113(3) 2013, pp. 74–77.
4. L. J. Cummings and W. F. Smyth: *Weak repetitions in strings.* J. Combinatorial Mathematics and Combinatorial Computing, 24 1997, pp. 33–48.
5. R. C. Entringer and D. E. Jackson: *On nonrepetitive sequences.* J. Comb. Theory, Ser. A, 16(2) 1974, pp. 159–164.
6. P. Erdös: *Some unsolved problems.* Hungarian Academy of Sciences Mat. Kutató Intézet Közl, 6 1961, pp. 221–254.
7. G. Fici, T. Lecroq, A. Lefebvre, Élise Prieur-Gaston, and W. F. Smyth: *Quasi-linear time computation of the Abelian periods of a word*, in Proc. PSC 2012, 2012, pp. 103–110.
8. G. Fici, T. Lecroq, A. Lefebvre, and E. Prieur-Gaston: *Computing Abelian periods in words*, in Proc. PSC 2011, 2011, pp. 184–196.
9. T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann: *Binary jumbled pattern matching on trees and tree-like structures*, in Proc. ESA 2013, 2013, pp. 517–528.
10. D. Hermelin, G. M. Landau, Y. Rabinovich, and O. Weimann: *Binary jumbled pattern matching via all-pairs shortest paths.* CoRR, abs/1401.2065 2014.
11. V. Keränen: *Abelian squares are avoidable on 4 letters*, in Proc. ICALP 1992, 1992, pp. 41–52.
12. T. Kociumaka, J. Radoszewski, and W. Rytter: *Efficient indexes for jumbled pattern matching with constant-sized alphabet*, in Proc. ESA 2013, 2013, pp. 625–636.
13. T. Kociumaka, J. Radoszewski, and W. Rytter: *Fast algorithms for Abelian periods in words and greatest common divisor queries*, in Proc. STACS 2013, 2013, pp. 245–256.
14. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. FOCS 1999, 1999, pp. 596–604.
15. Y. Li and W. F. Smyth: *Computing the cover array in linear time.* Algorithmica, 32(1) 2002, pp. 95–106.
16. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Inf. Process. Lett., 110(18–19) 2010, pp. 795–798.
17. R. Parikh: *On context-free languages.* J. ACM, 13(4) 1966, pp. 570–581.
18. P. A. B. Pleasants: *Non-repetitive sequences.* Mathematical Proceedings of the Cambridge Philosophical Society, 68 9 1970, pp. 267–274.

Two Squares Canonical Factorization*

Haoyue Bai[1], Frantisek Franek[1], and William F. Smyth[1,2]

[1] Department of Computing and Software
McMaster University, Hamilton, Ontario, Canada
{baih3,franek,smyth}@mcmaster.ca
[2] School of Computer Science & Software Engineering
University of Western Australia

Abstract. We present a new combinatorial structure in a string: a canonical factorization for any two squares that occur at the same position and satisfy some size restrictions. We believe that this canonical factorization will have application to related problems such as the New Periodicity Lemma, Crochemore-Rytter Three Squares Lemma, and ultimately the maximum-number-of-runs conjecture.

Keywords: string, primitive string, square, double square, factorization

1 Introduction

In 1995 Crochemore and Rytter [2] considered three distinct squares, all prefixes of a given string $\boldsymbol{x}$, and proved the *Three Squares Lemma* stating that, subject to certain restrictions, the largest of the three was at least the length of the sum of the other two. In 2006 Fan *et al.* [4] considered a special case of such two squares prefixes of $\boldsymbol{x}$ with a third square possibly offset some distance to the right; they proved a *New Periodicity Lemma* describing conditions under which the third square could not exist. Since that time there has been considerable work done [1,5,6,8] in an effort to specify more precisely the combinatorial structure of the string in the neighbourhood of such two squares.

In this paper we present a unique *canonical factorization* into primitive strings of what we call *double squares* – i.e. two squares starting at the same position and satisfying some size restrictions. The notion of double squares and their unique factorization can be traced to Lam [7]. A version of the factorization for more specific double squares was presented in [3]. Here we present it in full generality. In conclusion we indicate how this result can be applied to the proof of New Periodicity Lemma.

2 Preliminaries

In this section we develop the basic combinatorial tools that will be used to determine a canonical factorization for a double square. Chief among these are the Synchronization Principle (see Lemma 2), and the Common Factor Lemma (see Lemma 3), that lead to the main result, the Two Squares Factorization Lemma (see Lemma 6).

A ***string*** $\boldsymbol{x}$ is a finite sequence of symbols, called ***letters***, drawn from a (finite or infinite) set Σ, called the ***alphabet***. The length of the sequence is called the ***length*** of $\boldsymbol{x}$, denoted $|\boldsymbol{x}|$. Sometimes for convenience we represent a string $\boldsymbol{x}$ of length n as an array $\boldsymbol{x}[1..n]$. The string of length zero is called the ***empty string***, denoted $\boldsymbol{\varepsilon}$. If a string $\boldsymbol{x} = \boldsymbol{uvw}$, where $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ are strings, then $\boldsymbol{u}$ (respectively, $\boldsymbol{v}$, $\boldsymbol{w}$) is said to

* This work was supported by the *Natural Sciences and Engineering Research Council of Canada*

Haoyue Bai, Frantisek Franek, William F. Smyth: Two Squares Canonical Factorization, pp. 52–58.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

be a ***prefix*** (respectively, ***substring***, ***suffix***) of $\boldsymbol{x}$; a ***proper prefix*** (respectively, ***proper substring***, ***proper suffix***) if $|\boldsymbol{u}| < |\boldsymbol{x}|$ (respectively, $|\boldsymbol{v}| < |\boldsymbol{x}|$, $|\boldsymbol{w}| < |\boldsymbol{x}|$). A substring is also called a ***factor***. Given strings $\boldsymbol{u}$ and $\boldsymbol{v}$, $\mathrm{lcp}(\boldsymbol{u}, \boldsymbol{v})$ (respectively, $\mathrm{lcs}(\boldsymbol{u}, \boldsymbol{v})$) is the ***longest common prefix*** (respectively, ***longest common suffix***) of $\boldsymbol{u}$ and $\boldsymbol{v}$.

If $\boldsymbol{x}$ is a concatenation of $k \geq 2$ copies of a nonempty string $\boldsymbol{u}$, we write $\boldsymbol{x} = \boldsymbol{u}^k$ and say that $\boldsymbol{x}$ is a ***repetition***; if $k = 2$, we say that $\boldsymbol{x} = \boldsymbol{u}^2$ is a ***square***; if there exist no such integer k and no such $\boldsymbol{u}$, we say that $\boldsymbol{x}$ is ***primitive***. If $\boldsymbol{x} = \boldsymbol{v}^2$ has a proper prefix $\boldsymbol{u}^2$, $|su| < |\boldsymbol{v}| < 2|\boldsymbol{u}|$, we say that $\boldsymbol{x}$ is a ***double square*** and write $\boldsymbol{x} = \mathrm{DS}(\boldsymbol{u}, \boldsymbol{v})$. A square $\boldsymbol{u}^2$ such that $\boldsymbol{u}$ has no square prefix is said to be ***regular***.

For $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $1 \leq i < j \leq j+k \leq n$, the string $\boldsymbol{x}[i+k..j+k]$ is a ***right cyclic shift*** by k positions of $\boldsymbol{x}[i..j]$ if $\boldsymbol{x}[i] = \boldsymbol{x}[j+1], \ldots, \boldsymbol{x}[i+k-1] = \boldsymbol{x}[j+k]$. Equivalently, we can say that $\boldsymbol{x}[i..j]$ is a ***left cyclic shift*** by k positions of $\boldsymbol{x}[i+k..j+k]$. When it is clear from the context, we may leave out the number of positions and just speak of a cyclic shift.

Strings $\boldsymbol{uv}$ and $\boldsymbol{vu}$ are ***conjugates***, written $\boldsymbol{uv} \sim \boldsymbol{vu}$. We also say that $\boldsymbol{vu}$ is the $|\boldsymbol{u}|^{\text{th}}$ ***rotation*** of $\boldsymbol{x}$, written $R_{|\boldsymbol{u}|}(\boldsymbol{x})$, or the $-|\boldsymbol{v}|^{\text{th}}$ ***rotation*** of $\boldsymbol{x}$, written $R_{-|\boldsymbol{v}|}(\boldsymbol{x})$, while $R_0(\boldsymbol{x}) = R_{-|\boldsymbol{x}|}(\boldsymbol{x}) = \boldsymbol{x}$ is a ***primitive rotation***. Similarly as for the cyclic shift, when it is clear from the context, we may leave out the number of rotations and just speak of a rotation. Note that all cyclic shifts are conjugates, but not the other way around.

In the following lemma, the symbol | denotes *divisibility*, i.e. $a \mid b$ means that a is divisible by b.

Lemma 1 [9, Lemma 1.4.2] *Let $\boldsymbol{x}$ be a string of length n and minimum period $\pi \leq n$, and let $j = 1, \ldots, n-1$ be an integer. Then $R_j(\boldsymbol{x}) = \boldsymbol{x}$ if and only if $\boldsymbol{x}$ is not primitive ($\pi < n$, $\pi \mid n$) and $j \mid \pi$.*

The following results (Lemmas 2–6) are based on the development given in [3]. Though Lemmas 2 and 3 are folklore, we include their proofs.

Lemma 2 (Synchronization Principle) *The primitive string $\boldsymbol{x}$ occurs exactly p times in $\boldsymbol{x_2 x^p x_1}$, where p is a nonnegative integer and $\boldsymbol{x_1}$ (respectively, $\boldsymbol{x_2}$) is a proper prefix (respectively, proper suffix) of $\boldsymbol{x}$.*

Proof. From Lemma 1 a rotation $R_j(\boldsymbol{x})$ of $\boldsymbol{x}$ can equal $\boldsymbol{x}$ only if $\boldsymbol{x}$ is not primitive. Since here $\boldsymbol{x}$ is primitive, the only occurrences of $\boldsymbol{x}$ are exactly those determined by $\boldsymbol{x}^p$. □

Lemma 3 (Common Factor Lemma) *Suppose that $\boldsymbol{x}$ and $\boldsymbol{y}$ are primitive strings, where $\boldsymbol{x_1}$ (respectively, $\boldsymbol{y_1}$) is a proper prefix and $\boldsymbol{x_2}$ (respectively, $\boldsymbol{y_2}$) a proper suffix of $\boldsymbol{x}$ (respectively, $\boldsymbol{y}$). If for nonnegative integers p and q, $\boldsymbol{x_2 x^p x_1}$ and $\boldsymbol{y_2 y^q y_1}$ have a common factor of length $|\boldsymbol{x}|+|\boldsymbol{y}|$, then $\boldsymbol{x} \sim \boldsymbol{y}$.*

Proof. First consider the special case $\boldsymbol{x_1} = \boldsymbol{x_2} = \boldsymbol{y_1} = \boldsymbol{y_2} = \boldsymbol{\varepsilon}$, where $\boldsymbol{x}^p$, $\boldsymbol{y}^q$ have a common *prefix* $\boldsymbol{f}$ of length $|\boldsymbol{x}|+|\boldsymbol{y}|$. We show that in this case $\boldsymbol{x} = \boldsymbol{y}$.

Observe that $\boldsymbol{f}$ has prefixes $\boldsymbol{x}$ and $\boldsymbol{y}$, so that if $|\boldsymbol{x}| = |\boldsymbol{y}|$, then $\boldsymbol{x} = \boldsymbol{y}$, as required. Therefore suppose WLOG that $|\boldsymbol{x}| < |\boldsymbol{y}|$. Note that $\boldsymbol{y} \neq \boldsymbol{x}^k$ for any integer $k \geq 2$, since otherwise $\boldsymbol{y}$ would not be primitive, contradicting the hypothesis of the lemma.

Hence there exists $k \geq 1$ such that $k|\boldsymbol{x}| < |\boldsymbol{y}|$ and $(k+1)|\boldsymbol{x}| > |\boldsymbol{y}|$. But since $\boldsymbol{f} = \boldsymbol{yx}$, it follows that

$$R_{|\boldsymbol{y}|-k|\boldsymbol{x}|}(\boldsymbol{x}) = \boldsymbol{x},$$

again by Lemma 1 contrary to the assumption that $\boldsymbol{x}$ is primitive. We conclude that $|\boldsymbol{x}| \not< |\boldsymbol{y}|$, hence that $|\boldsymbol{x}| = |\boldsymbol{y}|$ and $\boldsymbol{x} = \boldsymbol{y}$, as required.

Now consider the general case, where $\boldsymbol{f}$ of length $|\boldsymbol{x}|+|\boldsymbol{y}|$ is a common factor of $\boldsymbol{x_2}\boldsymbol{x}^p\boldsymbol{x_1}$ and $\boldsymbol{y_2}\boldsymbol{y}^q\boldsymbol{y_1}$. Then $\boldsymbol{x_2}\boldsymbol{x}^p\boldsymbol{x_1} = \boldsymbol{u}\boldsymbol{f}\boldsymbol{u'}$ for some $\boldsymbol{u}$ and $\boldsymbol{u'}$. If $|\boldsymbol{u}| \geq |\boldsymbol{x}|$, then $\boldsymbol{f}$ is a factor of $\boldsymbol{x_1}\boldsymbol{x}^{p-1}\boldsymbol{x_2}$, and so we can assume WLOG that $|\boldsymbol{u}| < |\boldsymbol{x}|$. Setting $\tilde{\boldsymbol{x}} = R_{|\boldsymbol{u}|}(\boldsymbol{x})$, we see that $\boldsymbol{f}$ is a prefix of $\tilde{\boldsymbol{x}}^p$.

Similarly, by setting $\boldsymbol{y_2}\boldsymbol{y}^q\boldsymbol{y_1} = \boldsymbol{v}\boldsymbol{f}\boldsymbol{v'}$, we can assume that $|\boldsymbol{v}| < |\boldsymbol{y}|$, hence that $\boldsymbol{f}$ is also a prefix of $\tilde{\boldsymbol{y}}^q$ for $\tilde{\boldsymbol{y}} = R_{|\boldsymbol{v}|}(\boldsymbol{y})$. But this is just the special case considered above, for which $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{y}}$. Since $\boldsymbol{x} \sim \tilde{\boldsymbol{x}}$ and $\boldsymbol{y} \sim \tilde{\boldsymbol{y}}$, the result follows. □

Note that Lemma 3 could be equivalently stated in a more general form:

Lemma 4 *Suppose that $\boldsymbol{x}$ and $\boldsymbol{y}$ are strings where $\boldsymbol{x_1}$ (respectively, $\boldsymbol{y_1}$) is a proper prefix and $\boldsymbol{x_2}$ (respectively, $\boldsymbol{y_2}$) a proper suffix of $\boldsymbol{x}$ (respectively, $\boldsymbol{y}$). If for nonnegative integers p and q, $\boldsymbol{x_2}\boldsymbol{x}^p\boldsymbol{x_1}$ and $\boldsymbol{y_2}\boldsymbol{y}^q\boldsymbol{y_1}$ have a common factor of length $|\boldsymbol{x}|+|\boldsymbol{y}|$, then the primitive root $\overline{\boldsymbol{x}}$ of $\boldsymbol{x}$ and the primitive root $\overline{\boldsymbol{y}}$ of $\boldsymbol{y}$ are conjugates.*

The Common Factor Lemma gives rise to the following useful corollary:

Lemma 5 *Suppose that $\boldsymbol{x}$ and $\boldsymbol{y}$ are primitive strings, and that p and q are positive integers.*

(a) *If $\boldsymbol{x}^p = \boldsymbol{y}^q$, then $\boldsymbol{x} = \boldsymbol{y}$ and $p = q$.*
(b) *If $\boldsymbol{x_1}$ (respectively, $\boldsymbol{y_1}$) is a proper prefix of $\boldsymbol{x}$ (respectively, $\boldsymbol{y}$) and $\boldsymbol{x}^p\boldsymbol{x_1} = \boldsymbol{y}^q\boldsymbol{y_1}$ for $p \geq 2,\ q \geq 2$, then $\boldsymbol{x} = \boldsymbol{y},\ \boldsymbol{x_1} = \boldsymbol{y_1}$ and $p = q$.*

Proof. For (a), first consider $p = 1$, thus $\boldsymbol{x} = \boldsymbol{y}^q$. Since $\boldsymbol{x}$ is primitive, therefore $q = 1$ and $\boldsymbol{x} = \boldsymbol{y}$, as required. Similarly for $q = 1$. Suppose then that $p, q \geq 2$. This means that $\boldsymbol{x}^p$ and $\boldsymbol{y}^q = \boldsymbol{x}^p$ have a common factor of length $p|\boldsymbol{x}| = q|\boldsymbol{y}| \geq |\boldsymbol{x}|+|\boldsymbol{y}|$, so that by Lemma 3 $\boldsymbol{x} \sim \boldsymbol{y}$. Hence $|\boldsymbol{x}| = |\boldsymbol{y}|$ and so $\boldsymbol{x} = \boldsymbol{y}$.

For (b), since again $p \geq 2,\ q \geq 2$, it follows as in (a) that $\boldsymbol{x}^p\boldsymbol{x_1} = \boldsymbol{y}^q\boldsymbol{y_1}$ has a common factor of length at least $|\boldsymbol{x}|+|\boldsymbol{y}|$, hence the result. □

Note that in Lemma 5(b) the requirement $p \geq 2,\ q \geq 2$ is essential. For instance, $\boldsymbol{x} = aabb,\ \boldsymbol{x_1} = aa$ and $p = 2$ yields $\boldsymbol{x}^p\boldsymbol{x_1} = aabbaabbaa$, identical to $\boldsymbol{y}^q\boldsymbol{y_1}$ produced by $\boldsymbol{y} = aabbaabba,\ \boldsymbol{y_1} = a$ and $q = 1$ — but of course $\boldsymbol{x} \neq \boldsymbol{y}$.

3 Main Result – Two Squares Factorization Lemma

The next lemma specifies the structure imposed by the occurrence of two squares at the same position in a string. This structure has been described before, see [3,4,5,6,7], but not as precisely and with more assumptions required; above all, Lemma 6 establishes the uniqueness of the breakdown.

Lemma 6 (Two Squares Factorization Lemma) *For a double square* $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v})$*, there exists a unique primitive string $\boldsymbol{u_1}$ such that $\boldsymbol{u} = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}$ and $\boldsymbol{v} = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}\boldsymbol{u_1}^{e_2}$, where $\boldsymbol{u_2}$ is a possibly empty proper prefix of $\boldsymbol{u_1}$ and $e_1,\ e_2$ are integers such that $e_1 \geq e_2 \geq 1$. Moreover,*

(a) *if* $|\boldsymbol{u_2}| = 0$*, then* $e_1 > e_2 \geq 1$*;*
(b) *if* $|\boldsymbol{u_2}| > 0$*, then* $\boldsymbol{v}$ *is primitive, and if in addition* $e_1 \geq 2$*, then* $\boldsymbol{u}$ *also is primitive.*

In both cases, the factorization is unique.

Proof. If we have $\boldsymbol{u}^k$, $k \geq 2$, we refer to the first copy of $\boldsymbol{u}$ as $\boldsymbol{u}_{[1]}$, to the second copy of $\boldsymbol{u}$ as $\boldsymbol{u}_{[2]}$ etc.

Let $\boldsymbol{z}$ be the nonempty proper prefix of $\boldsymbol{u}_{[2]}$ that is in addition a suffix $\boldsymbol{z}$ of $\boldsymbol{v}_{[1]}$. But then $\boldsymbol{z}$ is also a prefix of $\boldsymbol{v}_{[1]}$, hence of $\boldsymbol{v}_{[2]}$; thus if $|\boldsymbol{u}| \geq 2|\boldsymbol{z}|$, it follows that $\boldsymbol{z}^2$ is a prefix of $\boldsymbol{u}$. In general, there exists an integer $k = \lfloor |\boldsymbol{u}|/|\boldsymbol{z}| \rfloor \geq 1$ such that $\boldsymbol{u} = \boldsymbol{z}^k\boldsymbol{z'}$ for some proper suffix $\boldsymbol{z'}$ of $\boldsymbol{z}$. Let $\boldsymbol{u_1}$ be the primitive root of $\boldsymbol{z}$, so that $\boldsymbol{z} = \boldsymbol{u_1}^{e_2}$ for some integer $e_2 \geq 1$. Therefore, for some $e_1 \geq e_2 k$ and some prefix $\boldsymbol{u_2}$ of $\boldsymbol{u_1}$, $\boldsymbol{u} = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}$ and $\boldsymbol{v} = \boldsymbol{uz} = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}\boldsymbol{u_1}^{e_2}$, as required. To prove uniqueness we consider two cases:

(i) $|\boldsymbol{u_2}| = 0$
Here $\boldsymbol{u} = \boldsymbol{u_1}^{e_1}$ and $\boldsymbol{v} = \boldsymbol{u_1}^{e_1+e_2}$, so that $\boldsymbol{x} = \boldsymbol{u_1}^{2(e_1+e_2)}$. Since $|\boldsymbol{v}| < 2|\boldsymbol{u}|$ and $e_1 \geq e_2$, it follows that $e_1 > e_2$. The uniqueness of $\boldsymbol{u_1}$ is a consequence of Lemma 5(a).

(ii) $|\boldsymbol{u_2}| > 0$
Suppose the choice of $\boldsymbol{u_1}$ is not unique. Then there exists some primitive string $\boldsymbol{w_1}$ with proper prefix $\boldsymbol{w_2}$, together with integers $f_1 \geq f_2 \geq 1$, such that $\boldsymbol{u} = \boldsymbol{w_1}^{f_1}\boldsymbol{w_2}$ and $\boldsymbol{v} = \boldsymbol{w_1}^{f_1}\boldsymbol{w_2}\boldsymbol{w_1}^{f_2}$. If both $e_1 \geq 2$ and $f_1 \geq 2$, it follows from Lemma 5(b) that $\boldsymbol{u_1} = \boldsymbol{w_1}$ and $e_1 = f_1$. If $e_1 = f_1 = 1$, we observe that $\boldsymbol{v} = \boldsymbol{uu_1} = \boldsymbol{uw_1}$, so that again $\boldsymbol{u_1} = \boldsymbol{w_1}$. In the only remaining case, exactly one of e_1, f_1 equals 1: therefore suppose WLOG that $f_1 > e_1 = 1$. Then $\boldsymbol{u} = \boldsymbol{u_1}\boldsymbol{u_2} = \boldsymbol{w_1}^{f_1}\boldsymbol{w_2}$ and $\boldsymbol{v} = \boldsymbol{u_1}\boldsymbol{u_2}\boldsymbol{u_1} = \boldsymbol{w_1}^{f_1}\boldsymbol{w_2}\boldsymbol{w_1}^{f_2}$, so that $\boldsymbol{u_1} = \boldsymbol{w_1}^{f_2}$. But since $\boldsymbol{u_1}$ is primitive, this forces $f_2 = 1$ and $\boldsymbol{u_1} = \boldsymbol{w_1}$, which, since $\boldsymbol{u_1}\boldsymbol{u_2} = \boldsymbol{w_1}^{f_1}\boldsymbol{w_2} = \boldsymbol{u_1}^{f_1}\boldsymbol{w_2}$, implies that $f_1 = 1$, a contradiction. Thus all cases have been considered, and $\boldsymbol{u_1}$ is unique.

We now show that $\boldsymbol{v}$ is primitive. Suppose the contrary, so there exists some primitive $\boldsymbol{w}$ and an integer $k \geq 2$ such that $\boldsymbol{v} = \boldsymbol{w}^k$. It follows that $|\boldsymbol{w}| \leq |\boldsymbol{v}|/2 \leq |\boldsymbol{u_1}^{e_1}|+|\boldsymbol{u_2}|$. Note that

$$\boldsymbol{w}^{2k} = \boldsymbol{v}^2 = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}\boldsymbol{u_1}^{e_1+e_2}\boldsymbol{u_2}\boldsymbol{u_1}^{e_2}, \tag{1}$$

so that $\boldsymbol{w}^{2k}$ and $\boldsymbol{u_1}^{e_1+e_2}\boldsymbol{u_2}$ have a common factor $\boldsymbol{u_1}^{e_1+e_2}\boldsymbol{u_2}$ of length

$$(|\boldsymbol{u_1}^{e_1}|+|\boldsymbol{u_2}|)+|\boldsymbol{u_1}^{e_2}| \geq |\boldsymbol{w}|+|\boldsymbol{u_1}|.$$

Thus we can apply Common Factor Lemma 3 to conclude that $\boldsymbol{w} \sim \boldsymbol{u_1}$, thus by (1) that $\boldsymbol{w} = \boldsymbol{u_1}$. But (1) then requires that the primitive string $\boldsymbol{u_1} = \boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}}$ aligns with $\boldsymbol{u_2}\boldsymbol{u_1}$, and so $\overline{\boldsymbol{u}}_{\boldsymbol{2}}$ is a prefix of $\boldsymbol{u_1}$, in contradiction to Lemma 1. We conclude that $\boldsymbol{v}$ is primitive.

Now suppose in addition that $e_2 \geq 2$, but that $\boldsymbol{u}$ is not primitive. Then there exists some primitive $\boldsymbol{w}$ and some integer $k \geq 2$ such that $\boldsymbol{u} = \boldsymbol{w}^k$. Hence $|\boldsymbol{w}| \leq |\boldsymbol{u}|/2 = (|\boldsymbol{u_1}^{e_1}|+|\boldsymbol{u_2}|)/2 < |\boldsymbol{u_1}^{e_1-1}|+|\boldsymbol{u_2}|$, since $e_1 \geq 2$ and $|\boldsymbol{u_2}| > 0$. Therefore, since $\boldsymbol{u_1}^{e_1}\boldsymbol{u_2}$ is a prefix of $\boldsymbol{u}^2 = \boldsymbol{w}^{2k}$, and since $e_2 \geq 1$ by Lemma 6, $\boldsymbol{w}^{2k}$ and $\boldsymbol{u_1}^{e_1+e_2}$ have a common prefix $\boldsymbol{u_1}^{e_1}\boldsymbol{u_2}$. Note that $|\boldsymbol{u_1}^{e_1}\boldsymbol{u_2}| \geq |\boldsymbol{v}|+|\boldsymbol{u_1}|$, so that again applying Common Factor Lemma 3, we conclude that $\boldsymbol{u_1} = \boldsymbol{w}$. This in turn implies $\boldsymbol{u} = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2} = \boldsymbol{u_1}^k$, impossible since $0 < |\boldsymbol{u_2}| < |\boldsymbol{u_1}|$. Therefore $\boldsymbol{u}$ is primitive, as required.

Finally we remark that since $\boldsymbol{u_1}$ is a uniquely determined primitive string, therefore $\boldsymbol{u_2}$, e_1 and e_2 are also uniquely determined. □

The following examples show that the statement of the lemma is sharp:

(a) The second part of Lemma 6(b) requires that $e_1 \geq 2$. To see that this condition is not necessary, consider $\boldsymbol{v}^2 = abaababaab$, where $\boldsymbol{u} = (ab)a$, $\boldsymbol{v} = (ab)a(ab)$, so that $\boldsymbol{u_1} = ab$, $\boldsymbol{u_2} = a$, $e_1 = e_2 = 1$, but $\boldsymbol{u}$ is primitive.
(b) On the other hand, consider $\boldsymbol{v}^2 = abaabaabaababaabaabaab$, where $\boldsymbol{u} = (aba)^2 = (abaab)a$, $\boldsymbol{v}= (abaab)a(abaab)$, so that $\boldsymbol{u_1} = abaab$, $\boldsymbol{u_2} = a$, $e_1 = e_2 = 1$, where now $\boldsymbol{u_1}$ is *not* primitive.

Lemma 6 gives credence to the following definition of terminology and notation:

Definition 7 *For a double square* $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v})$ *we call the unique factorization* $\boldsymbol{v}^2 = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}\boldsymbol{u_1}^{e_1+e_2}\boldsymbol{u_2}\boldsymbol{u_1}^{e_2}$ *guaranteed by Lemma 6, the* ***canonical factorization*** *of* $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v})$ *and denote it by* $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u_1}, \boldsymbol{u_2}, e_1, e_2)$. *The symbol* $\overline{\boldsymbol{u}}_{\boldsymbol{2}}$ *denotes the suffix of* $\boldsymbol{u_1}$ *such that* $\boldsymbol{u_1} = \boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}}$.

Lemma 6 also gives rise to a number of important observations:

Observation 8 *In Lemma 6,* $|\boldsymbol{u_2}| > 0$ *if any one of the following conditions holds:*

(a) $\boldsymbol{v}$ *is primitive;*
(b) $\boldsymbol{u}$ *is primitive;*
(c) *there is no other occurrence of* $\boldsymbol{u}^2$ *farther to the right in* $\boldsymbol{v}^2$ *(*$\boldsymbol{u}^2$ *is rightmost);*
(d) $\boldsymbol{u}^2$ *is regular.*

Moreover:

(e) $|\boldsymbol{u_2}| > 0$ *if and only if* $\boldsymbol{v}$ *is primitive;*
(f) *If* $\boldsymbol{u}^2$ *is regular, then* $e_1 = e_2 = 1$ *and* $\boldsymbol{u_1}$ *is regular.*

Proof.

(a) $|\boldsymbol{u_2}| = 0$ implies $\boldsymbol{v}$ not primitive.
(b) $|\boldsymbol{u_2}| = 0$ implies $\boldsymbol{u}$ not primitive.
(c) $|\boldsymbol{u_2}| = 0$ implies $\boldsymbol{u}^2 = \boldsymbol{u_1}^{2e_1}$, which occurs twice in $\boldsymbol{v}^2 = \boldsymbol{u_1}^{2(e_1+e_2)}$, in particular as a suffix.
(d) Since $\boldsymbol{u}^2$ is regular, therefore $\boldsymbol{u}$ is primitive, so that by (b), $|\boldsymbol{u_2}| > 0$.
(e) By (a), primitive $\boldsymbol{v}$ implies $|\boldsymbol{u_2}| > 0$; by Lemma 6, $|\boldsymbol{u_2}| > 0$ implies that $\boldsymbol{v}$ is primitive.
(f) By (d), regular $\boldsymbol{u}^2$ implies $|\boldsymbol{u_2}| > 0$, so that $\boldsymbol{u} = \boldsymbol{u_1}^{e_1}\boldsymbol{u_2}$, which is regular only if $e_1 = e_2 = 1$ and $\boldsymbol{u_1}$ is regular. □

In the context of Observation 8(f), consider the double square $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v})$ where $\boldsymbol{u} = aabaa$, $\boldsymbol{v} = aabaaaab$. In this case, we find $\boldsymbol{u_1} = aab$, $\boldsymbol{u_2} = aa$, $e_1 = e_2 = 1$, but observe that $\boldsymbol{u}$ has prefix a^2, so $\boldsymbol{u}^2$ is not regular. Thus the condition $e_1 = 1$ is more general than the requirement that $\boldsymbol{u}^2$ be regular.

Now, following [3], consider the case $|\boldsymbol{u_2}| > 0$ of Lemma 6 and set $\boldsymbol{u_1} = \boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}}$. Thus $\boldsymbol{v}^2$ becomes

$$\begin{aligned} \boldsymbol{v}^2 \quad &= \quad (\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}})^{e_1}\boldsymbol{u_2}(\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}})^{e_1+e_2}\boldsymbol{u_2}(\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}})^{e_2} \\ &= (\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}})^{e_1-1}\boldsymbol{u_2}(\mathrm{IF})(\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}})^{e_1+e_2-2}\boldsymbol{u_2}(\mathrm{IF})(\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}})^{e_2-1} \end{aligned} \tag{2}$$

where $\mathrm{IF} = \overline{\boldsymbol{u}}_{\boldsymbol{2}}\boldsymbol{u_2}\boldsymbol{u_2}\overline{\boldsymbol{u}}_{\boldsymbol{2}} = R_{|\boldsymbol{u_2}|}(\boldsymbol{u_1})\boldsymbol{u_1}$ is called the ***inversion factor***.

Lemma 9 *Consider a double square* $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u_1}, \boldsymbol{u_2}, e_1, e_2)$ *with a non-empty* $\boldsymbol{u_2}$. *Then the inversion factor* IF *have exactly two occurrences in* $\boldsymbol{v}^2$ *exactly a distance of* $|\boldsymbol{v}|$ *apart as shown in* (2).

Proof. If IF occurs elsewhere in $\boldsymbol{v}^2$, by the Synchronization principle its subfactor $\boldsymbol{u_2\overline{u}_2}$ must align with an occurrence of $\boldsymbol{u_2\overline{u}_2}$ as it is primitive. Thus, its subfactor $\boldsymbol{\overline{u}_2u_2}$ must align with $\boldsymbol{u_2\overline{u}_2}$, contradicting the primitiveness of $\boldsymbol{u_2\overline{u}_2}$, see Lemma 1. □

The quantity $\mathrm{lcs}(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})$ gives the maximal number of positions the structures $(\boldsymbol{u_2\overline{u}_2})^{e_1+e_2}$ and $(\boldsymbol{u_2\overline{u}_2})^{e_2}$ can be cyclically shifted to the left in $\boldsymbol{v}^2$, while $\mathrm{lcp}(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})$ gives the maximal number of positions the structures $(\boldsymbol{u_2\overline{u}_2})^{e_1}$ and $(\boldsymbol{u_2\overline{u}_2})^{e_1+e_2}$ can be cyclically shifted to the right. In [3], the following lemma limiting the size of $\mathrm{lcs}(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})+\mathrm{lcp}(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})$ was given.

Lemma 10 ([3]) *Considering* $\boldsymbol{u_1}^{e_1}\boldsymbol{u_2u_1}^{e_1+e_2}\boldsymbol{u_2u_1}^{e_2}$, *where* $\boldsymbol{u_1}$ *is primitive and* $\boldsymbol{u_2}$ *is a non-empty proper prefix of* $\boldsymbol{u_1}$, $e_1 \geq e_2 \geq 1$, *and* $\boldsymbol{\overline{u}_2}$ *a suffix of* $\boldsymbol{u_1}$ *so that* $\boldsymbol{u_1} = \boldsymbol{u_2\overline{u}_2}$, *then* $lcs(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})+lcp(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2}) \leq |\boldsymbol{u_1}|-2$.

In fact, in [3] the inversion factor is defined more generally as any factor $\boldsymbol{\overline{w}ww\overline{w}}$ of $\boldsymbol{v}^2$ such that $|\boldsymbol{w}| = |\boldsymbol{u_2}|$ and $|\boldsymbol{\overline{w}}| = |\boldsymbol{\overline{u}_2}|$ and a stronger result is given (re-phrased in the terminology of this paper):

Lemma 11 ([3]) *Consider a double square* $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u_1}, \boldsymbol{u_2}, e_1, e_2)$ *with a non-empty* $\boldsymbol{u_2}$ *and let* $p = lcp(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})$ *and* $s = lcs(\boldsymbol{u_2\overline{u}_2}, \boldsymbol{\overline{u}_2u_2})$. *Then any inversion factor in* $\boldsymbol{v}^2$ *is either* $R_i(\mathrm{IF})$ *or* $R_{-j}(\mathrm{IF})$ *for some* $i \in 0, \ldots, p$ *or some* $j \in 0, \ldots, s$. *Moreover, every* $R_i(\mathrm{IF})$ *or* $R_{-j}(\mathrm{IF})$ *appear exactly twice in* $\boldsymbol{v}^2$ *exactly a distance* $|\boldsymbol{v}|$ *apart for every* $i \in 0, \ldots, p$ *and every* $j \in 0, \ldots, s$.

4 Possible application to New Periodicity Lemma

Some years ago a New Periodicity Lemma was published [4], showing that the occurrence of two special squares at a position i in a string, necessarily precludes the occurrence of other squares of specific period in a specific neighbourhood of i. The proof of this lemma was complex, breaking down into 14 subcases, and required a very strong condition that the shorter of the two squares be regular.

Lemma 12 ([4], New Periodicity Lemma) *Let* $\boldsymbol{x} = \mathrm{DS}(\boldsymbol{u}, \boldsymbol{v})$, *where we require that* $\boldsymbol{u}^2$ *be regular and that* $\boldsymbol{v}$ *be primitive. Then for all integers* k *and* w *such that* $0 \leq k < |\boldsymbol{v}|-|\boldsymbol{u}|$ *and* $|\boldsymbol{v}|-|\boldsymbol{u}| < w < |\boldsymbol{v}|$, $w \neq |\boldsymbol{u}|$, $\boldsymbol{x}[k{+}1..k{+}2w]$ *is not a square.*

First note that by Observation 8, the requirement that v be primitive is redundant; the fact that u^2 is regular necessarily forces the primitivness of v. Also note that the regularity of u^2 necessarily implies that in the canonical factorization of $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u_1}, \boldsymbol{u_2}, e_1, e_2)$, $e_1 = e_2 = 1$.

Consider $\mathrm{DS}(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u_1}, \boldsymbol{u_2}, 1, 1)$. Let $\boldsymbol{\overline{u}_2}$ be a suffix of $\boldsymbol{u_1}$ such that $\boldsymbol{u_1} = \boldsymbol{u_2\overline{u}_2}$. The canonical factorization thus has the form

$(\boldsymbol{u_2\overline{u}_2})\boldsymbol{u_2}(\boldsymbol{u_2\overline{u}_2})(\boldsymbol{u_2\overline{u}_2})\boldsymbol{u_2}(\boldsymbol{u_2\overline{u}_2})$.

Let us consider a square $\boldsymbol{w}^2$ such that $|\boldsymbol{u_1}| < |\boldsymbol{w}| < |\boldsymbol{v}|$ and $|\boldsymbol{w}| \neq |\boldsymbol{u}|$. We want to show that this is not possible.

If for instance $\boldsymbol{w}$ starts in the first $\boldsymbol{u_2}$ and ends in the fourth $\boldsymbol{u_2}$, then $\boldsymbol{w}$ contains fully the IF, so the second $\boldsymbol{w}$ has to as well, and so $|\boldsymbol{w}| \geq |\boldsymbol{v}|$, a contradiction.
If $\boldsymbol{w}$ ends in the second $\boldsymbol{\overline{u}_2}$ we cannot argue using IF, but still knowing that $\boldsymbol{u_2\overline{u}_2}$ is

primitive and also all its rotations are primitive, using the Synchronization principle can be applied to obtain a contradiction.

Almost all possible cases for $\boldsymbol{w}^2$ except two can be easily shown impossible using only the properties of the canonical factorization. Thus, we believe, and it is our immediate goal for future research, that the canonical factorization will not only provide us with a significantly simplified proof of New Periodicity Lemma, but will also allow us to significantly reduce the conditions on $\boldsymbol{u}^2$ from $\boldsymbol{u}$ being regular to just being primitive. We also believe that the canonical factorization in the same way will not only provide a simpler proof of Crochemore-Rytter Three Squares Lemma, but will extend the applicability of the lemma to three squares when any of the squares is primitive (the original lemma requires that the smallest square be primitive).

5 Conclusion and future work

We presented a unique factorization of a double square, i.e. a configuration of two squares $\boldsymbol{u}^2$ and $\boldsymbol{v}^2$ starting at the same position and satisfying $|u| < |v| < 2|u|$. We call this factorization the *canonical factorization*. It has very strong combinatorial properties as it is an almost periodic repetition of a primitive string. We indicated that we would like to use this new insight into the structure of double squares in improving the New Periodicity Lemma [4] and Crochemore-Rytter's Three Squares Lemma [2] and simplifying their proofs. As of preparing this final version of the Prague Stringology Conference 2014 proceedings, we are happy to report that the canonical factorization presented here indeed greatly simplified and generalized both. The follow-up work will focus on presenting of these results in a near future.

References

1. W. Bland and W. F. Smyth: *Overlapping squares: the general case characterized & applications*. submitted for publication, 2014.
2. M. Crochemore and W. Rytter: *Squares, cubes, and time-space efficient string searching*. Algorithmica, 13 1995, pp. 405–425.
3. A. Deza, F. Franek, and A. Thierry: *How many double squares can a string contain?* submitted for publication, 2013.
4. K. Fan, S. Puglisi, W. F. Smyth, and A. Turpin: *A new periodicity lemma*. SIAM Journal on Discrete Mathematics, 20 2006, pp. 656–668.
5. F. Franek, R. C. G. Fuller, J. Simpson, and W. F. Smyth: *More results on overlapping squares*. Journal of Discrete Algorithms, 17 2012, pp. 2–8.
6. E. Kopylova and W. F. Smyth: *The three squares lemma revisited*. Journal of Discrete Algorithms, 11 2012, pp. 3–14.
7. N. H. Lam: *On the number of squares in a string. AdvOL-Report 2013/2, McMaster University*, 2013.
8. J. Simpson: *Intersecting periodic words*. Theoretical Computer Science, 374 2007, pp. 58–65.
9. B. Smyth: *Computing Patterns in Strings*, Pearson Addison-Wesley, 2003.

Multiple Pattern Matching Revisited

Robert Susik[1], Szymon Grabowski[1], and Kimmo Fredriksson[2]

[1] Lodz University of Technology, Institute of Applied Computer Science
Al. Politechniki 11, 90–924 Łódź, Poland
{rsusik|sgrabow}@kis.p.lodz.pl
[2] School of Computing, University of Eastern Finland
P.O.B. 1627, FI-70211 Kuopio, Finland
kimmo.fredriksson@uef.fi

Abstract. We consider the classical exact multiple string matching problem. Our solution is based on q-grams combined with pattern superimposition, bit-parallelism and alphabet size reduction. We discuss the pros and cons of the various alternatives of how to achieve best combination. Our method is closely related to previous work by (Salmela et al., 2006). The experimental results show that our method performs well on different alphabet sizes and that they scale to large pattern sets.

Keywords: combinatorial problems, string algorithms, q-grams, word-level parallelism

1 Introduction

Multiple pattern matching is a classic problem, with about 40 years of history, with applications in intrusion detection, anti-virus software and bioinformatics, to name a few. The problem can be stated as follows: Given text T of length n and pattern set $\mathcal{P} = \{P_1, \ldots, P_r\}$, in which each pattern is of length m, and all considered sequences are over common alphabet Σ of size σ, find all pattern occurrences in T. The pattern equal length requirement may be removed. The multiple pattern matching problem is a straightforward generalization of single pattern matching and it is no surprise that many techniques worked out for a single pattern are borrowed in efficient algorithms for multiple patterns.

1.1 Related work

The classical algorithms for the present problem can be roughly divided into three different categories, (*i*) prefix searching, (*ii*) suffix searching and (*iii*) factor searching. Another way to classify the solutions is to say that they are based on character comparisons, hashing, or bit-parallelism. Yet another view is to say that they are based on filtering, aiming for good average case complexity, or on some kind of "direct search" with good worst case complexity guarantees. These different categorizations are of course not mutually exclusive, and many solutions are hybrids that borrow ideas from several techniques. For a good overview of the classical solutions we refer the reader e.g. to [21,16,9]. We briefly review some of them in the following.

Perhaps the most famous solution to the multiple pattern matching problem is the Aho–Corasick (AC) [1] algorithm, which works in linear time (prefix-based approach). It builds a pattern trie with extra (failure) links and actually generalizes the Knuth–Morris–Pratt algorithm [18] for a single pattern. More precisely, AC total time is $O(M + n + z)$, where M, the sum of pattern lengths, is the preprocessing cost, and z is the total number of pattern occurrences in T. Recently Fredriksson and

Robert Susik, Szymon Grabowski, Kimmo Fredriksson: Multiple Pattern Matching Revisited, pp. 59–70.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

Grabowski [15] showed an average-optimal filtering variant of the classic AC algorithm. They built the AC automaton over superimposed subpatterns, which allows to sample the text characters in regular distances, not to miss any match (i.e., any verification). This algorithm is based on the same ideas as the current work.

Another classic algorithm is Commentz–Walter [7], which generalizes the ideas of Boyer–Moore (BM) algorithm [4] for a single pattern to solve the multiple pattern matching problem (suffix-based approach). Set Horspool [12,21] may be considered its more practical simplification, exactly in the way that Boyer–Moore–Horspool (BMH) [17] is a simplification of the original BM. Set Horspool makes use of a generalized bad character function. The Horspool technique was used in a different way in an earlier algorithm by Wu and Manber [24]. These methods are based on backward matching over a sliding text window, which is shifted based on some rule, and the hope is that many text characters can be skipped altogether.

The first factor based algorithms were DAWG-match [8] and MultiBDM [10]. Like Commentz–Walter and Set Horspool they are based on backward matching. However, instead of recognizing the pattern suffixes, they recognize the factors, which effectively means that they work more per window, but in return they are able to make longer shifts of the sliding window, and in fact they obtain optimal average case complexity. At the same time they are linear in the worst case. The drawback is that these algorithms are reasonably complex and not very efficient in practice. More practical approach is the Set Backward Oracle Matching (SBOM) algorithm [2], which is based on the same idea as MultiBDM, but uses simpler data structures and is very efficient in practice. Yet another variant is the Succinct Backward DAWG Matching algorithm (SBDM) [14], which is practical for huge pattern sets due to replacing the suffix automaton with succinct index. The factor based algorithms usually lead to average optimal [19] complexity $O(n \log_\sigma(rm)/m)$.

Bit-parallelism can be used to replace the various automata in the previous methods to obtain very simple and very efficient variants of many classical algorithms. The classic method for a single pattern is Shift-Or [3]. The idea is to encode (nondeterministic) automaton as a bitvector, i.e. a small integer value, and simulate all the states in parallel using Boolean logic and arithmetic. The result is often the most practical method for the problem, but the drawback is that the scalability is limited by the number of bits in a computer word, although there exist ways to alleviate this problem somewhat, see [22,6]. Another way that is applicable to huge pattern sets is to combine bit-parallelism with q-grams; our method is also based on this, and we review the idea and related previous work in detail in the next section.

Some recent work also recognizes the neglected power of the SIMD instructions, which have been available on commodity computers well over a decade. For example, Faro and Külekci [11] make use of the Intel Streaming SIMD Extensions (SSE) technology, which gives wide registers and many special purpose instructions to work with. They develop (among other things) a wsfp (*word-size fingerprint instruction*) operation, based on hardware opcode for computing CRC32 checksums, which computes an α-bit fingerprint from a w-bit register handled as a block of α characters. Similar values are obtained for all α-sized factors of all the patterns in the preprocessing, and wsfp can therefore be used as a simple yet efficient hash-function to identify text blocks that may contain a matching pattern.

The paper is organized as follows. Section 2 describes and discusses the two key concepts underlying our work, q-grams and pattern superimposition. Section 3 presents the description of our algorithm, together with its complexity analysis. Sec-

tion 4 contains (preliminary) experimental results. The last section concludes and points some avenues for pursuing further research.

2 On q-grams and superimposition

A q-gram is (usually) a contiguous substring (factor) of q characters of a string, although non-contiguous q-grams have been considered [5]. In what follows, q can be considered a small constant, $2, \ldots, 6$ in practice, although we may analyze the optimal value for a given problem instance. We note that q-grams have been widely used in approximate (single and multiple) string matching, where they can be used to obtain fast filtering algorithms based on exact matching of a set of q-grams. Obviously these algorithms work for the exact case as well, as a special case, but they are not interesting in our point of view. Another use (which is not relevant in our case) is to speed up exact matching of a single pattern by treating the q-grams as a superalphabet, see [13].

In our case q-grams are interesting as combined with a technique called superimposition. Consider a set of patterns $\mathcal{P} = \{P_1, \ldots, P_r\}$. We form a single pattern P where each position $P[i]$ is no longer a single character, but a *set* of characters, i.e. $P[i] \subseteq \Sigma$. More precisely, $P[i] = \bigcup_j P_j[i]$. Now P can be used as a *filter*: we search candidate text substrings that might contain an occurrence of any of the patterns in $\mathcal{P}$. That is, if $T[i+j] \in P[j]$, for all $j \in 1, \ldots, m$, then $T[i \ldots i+m-1]$ may match with some pattern in $\mathcal{P}$.

For example, if $\mathcal{P} = \{abba, bbac\}$, the superimposed pattern will be $P = \{a,b\}\{b\}\{a,b\}\{a,c\}$, and there are a total of 8 different strings of length 4 that can match with P (and trigger verification). Therefore we immediately notice one of the problems with this approach, i.e. the probability that some text character t matches a pattern character p is no longer $1/\sigma$ (assuming uniform random distribution), it can be up to r/σ. This gets quickly out of hands when the number of patterns r grows.

To make the technique more useful, we first generate a new set of patterns, and then superimpose. The new patterns have the q-grams as the alphabet, which mean the new alphabet has size σ^q, and the probability of a false positive candidate will be considerably lower. There are two main approaches: overlapping and non-overlapping q-grams.

Consider first the overlapping q-grams. For each P_i we generate a new pattern such that $P'_i[j] = P_i[j \ldots j+q-1]$, for $j \in 1, \ldots, m-q+1$, that is, each q-gram $P_i[j \ldots j+q-1]$ is treated as a single "super character" in P'_i. Note also that the pattern lengths are decreased from m to $m-q+1$. Taking the previous example, if $\mathcal{P} = \{abba, bbac\}$ and now $q = 2$, the new pattern set is $\mathcal{P}' = \{[ab][bb][ba], [bb][ba][ac]\}$, where we use the brackets to denote the q-grams. The corresponding superimposed pattern is then $P' = \{[ab],[bb]\}\{[bb],[ba]\}\{[ba],[ac]\}$. To be able to search for P', the text must be factored in exactly the same way.

The other possibility is to use non-overlapping q-grams. In this case we have $P'_i[j] = P_i[(j-1)q+1 \ldots jq]$, for $j \in 1, \ldots, \lfloor m/q \rfloor$, and for our running example we get $P' = \{[ab],[bb]\}\{[ba],[ac]\}$. Again, the text must be factored similarly. But the problem now is that only every qth text position is considered, and to solve this problem we must consider all q possible shifts of the original patterns. That is, given a pattern P_i, we generate a set $\hat{P}_i = \{P_i[1 \ldots m], P_i[2 \ldots m], \ldots, P_i[q-1 \ldots m]\}$, and then generate $\hat{P}'_i$, and finally superimpose them.

The above two alternatives both have some benefits and drawbacks. For overlapping q-grams we have:

- pattern length is large $(m - q + 1)$, which means less verifications
- text length is practically unaffected $(n - q + 1)$

Non-overlapping:

- pattern length is short (m/q), which means potentially more verifications, but bit-parallelism works for bigger m
- text is shorter too (n/q)
- more patterns to superimpose (factor of q)

In the end, the benefits and drawbacks between the two approaches mostly cancel out each other, except bit-parallelism remains more applicable to non-overlapping q-grams.

To illustrate the power of this technique, let us have, for example, a random text over an alphabet of size $\sigma = 16$ and patterns generated according to the same probability distribution; q-grams are not used yet (i.e., we assume $q = 1$). If $r = 16$, then the expected size of a character class in the superimposed pattern is about 10.3, which means that a match probability for a single character position is about 64%. Even if high, this value may yet be feasible for long enough patterns, but if we increase r to 64, the character class expected size grows to over 15.7 and the corresponding probability to over 98%. This implies that match verifications are likely to be invoked for most positions of the text. Using q-grams has the effect of artificially growing the alphabet. In our example, if we use $q = 2$ and thus $\sigma' = 16^2 = 256$, the corresponding probabilities for $r = 16$ and $r = 64$ become about 6% and 22%, respectively, so they are significantly lower.

The main problem that remains is to decide between the two choices, properly choose a suitable q, and finally find a good algorithm to search the superimposed pattern. To this end, Salmela et al. [23] presented three algorithms combining the known mechanisms: Shift-Or, BNDM [20] and BMH, with overlapping q-grams; the former two of these algorithms are bit-parallel ones. The resulting algorithm were called SOG, BG and HG, respectively. In general larger q means better filtering, but on the other hand the size of the data structures (tables) that the algorithms use is $O(\sigma^q)$, which can be prohibitive. BGqus [25] tries to solve the problem by combining BG with hashing.

In general, not many classic algorithms can be generalized to handle superimposed patterns (character classes) efficiently, but bit-parallel methods generalize trivially. In the next section we describe our choice, FAOSO [15].

3 Our algorithm

In [15] a general technique of how to skip text characters, with any (linear time) string matching algorithm that can search for multiple patterns simultaneously was presented, alongside with several applications to known algorithms. In the following we review the idea, and for the moment assume that we already have done all factoring to q-grams, and that we have only a single pattern.

3.1 Average-optimal character skipping

The method takes a parameter k, and from the original pattern generates a set $\mathcal{K}$ of k new patterns $\mathcal{K} = \{P^0, \ldots, P^{k-1}\}$, each of length $m' = \lfloor m/k \rfloor$, as follows:

$$P^j[i] = P[j + ik], \quad j = 0, \ldots, k-1, \quad i = 0, \ldots, \lfloor m/k \rfloor - 1.$$

In other words, k different alignments of the original pattern P is generated, each alignment containing only every kth character. The total length of the patterns P^j is $k\lfloor m/k \rfloor \leq m$.

Assume now that P occurs at $T[i \ldots i+m-1]$. From the definition of P^j it directly follows that

$$P^j[h] = T[i + j + hk], \quad j = i \mod k, \quad h = 0, \ldots, m' - 1.$$

This means that the set $\mathcal{K}$ can be used as a filter for the pattern P, and that the filter needs only to scan every kth character of T. Fig. 1 serves as an illustration.

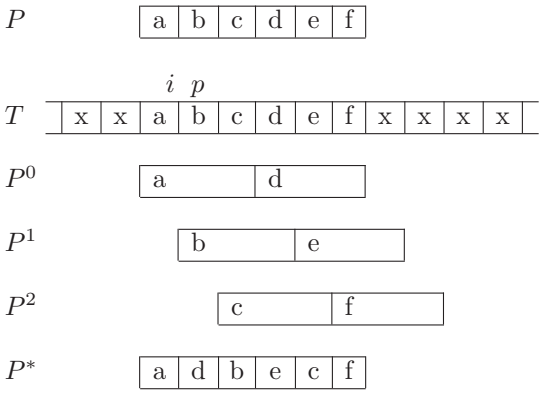


Figure 1. An example. Assume that $P =$ `abcdef` occurs at text position $T[i \ldots i+m-1]$, and that $k = 3$. The current text position is $p = 10$, and $T[p] =$ `b`. The next character the algorithm reads is $T[p+k] = T[13] =$ `e`. This triggers a match of $P^{p \mod k} = P^1$, and the text area $T[p-1 \ldots p-1+m-1] = T[i \ldots i+m-1]$ is verified.

The occurrences of the patterns in $\mathcal{K}$ can be searched for simultaneously using any multiple string matching algorithm. Assuming that the selected string matching algorithm runs generally in $O(n)$ time, then the filtering time becomes $O(n/k)$, as only every kth symbol of T is read. The filter searches for the exact matches of k patterns, each of length $\lfloor m/k \rfloor$. Assuming that each character occurs with probability $1/\sigma$, the probability that P^j occurs (triggering a verification) in a given text position is $(1/\sigma)^{\lfloor m/k \rfloor}$. A brute force verification cost is in the worst case $O(m)$. To keep the total time at most $O(n/k)$ on average, we select k so that $nm/\sigma^{m/k} = O(n/k)$. This is satisfied for $k = m/(2\log_\sigma(m))$, where the verification cost becomes $O(n/m)$ and filtering cost $O(n\log_\sigma(m)/m)$. The total average time is then dominated by the filtering time, i.e. $O(n\log_\sigma(m)/m)$, which is optimal [26].

3.2 Multiple matching with *q*-grams

To apply the previous idea to multiple matching, we just assume that the (single) input pattern (for the filter) is the non-overlapping q-gram factored and superimposed

pattern set. The verification phase just needs to be aware that there are possibly more than one pattern to verify. The analysis remains essentially the same: now the text length is n/q, pattern lengths are m/q, there are r patterns to verify, and the probability of a match is p instead of $1/\sigma$, where $p = O(1-(1-(1/\sigma^q))^{qr}) = O((qr)/\sigma^q)$. That is, the filtering time is $O(qn/(kq)) = O(n/k)$, verification cost is $O(rqm)$, and its probability is $O(p^{\lfloor m/(kq)\rfloor})$ for each of the n/q text positions. However, now we have two parameters to optimize, k and q, and the optimal value of one depends on the other.

In practice we want to choose q first, such that the verification probability is as low as possible. This means maximizing q, but the preprocessing cost (and space) grows as $O(\sigma^q)$, and we do not want this to exceed $O(rm)$ (or the filtering cost for that matter). So we select $q = \log_\sigma(rm)$, and then choose k as large as possible. Repeating the above analysis gives then

$$k = O\left(\frac{m}{\log_\sigma(rm)} \cdot \frac{\log_\sigma 1/\rho}{\log_\sigma(rm) + \log_\sigma 1/\rho}\right),$$

where $\rho = \log_\sigma(rm)/m$. We note that this is not average-optimal anymore, although we are still able to skip text characters.

To actually search the superimposed pattern, we use FAOSO [15], which is based on Shift-Or. The fact that the pattern consists of character classes is not a problem for bit-parallel algorithms, since it only affects the initial preprocessing of a single table. For details see [15]. The filter implemented with FAOSO runs in $O(n/k \cdot \lceil (m/q)/w \rceil)$ time in our case, where w is the number of bits in computer word (typically 64).

We note that Salmela et al. [23] have tried a similar approach, but dismissed it early because it did not look promising for short patterns in their tests.

Implementation. In the algorithms' point of view the q-gram, i.e. the super character, must have some suitable representation, and the convenient way is to compute a numerical value in the range $0, \ldots, \sigma^q - 1$, which is done as $\sum_{i=1}^{q} S[i] \cdot \sigma^{i-1}$ for a q-gram $S[1 \ldots q]$. This is computed using Horner's method to avoid the exponentiation. We have experimented with two different variants. The first encodes the whole text prior to starting the actual search algorithm, which is then more streamlined. This also means that the total complexity is $\Omega(n)$, the time to encode the text. We call the resulting algorithm SMAG (short of Simple Multi AOSO on q-Grams). The other alternative is to keep the text intact, and compute the numerical representation of the q-gram requested on the fly. This adds just constant overhead to the total complexity. We call this variant MAG (short of Multi AOSO on q-Grams). We have verified experimentally that MAG is generally better than SMAG.

3.3 Alphabet mapping

If the alphabet is large, then selecting a suitable q may become a problem. The reason is that some value q' may be too small to facilitate good filtering capability, yet, using $q = q' + 1$ can be problematic, as the preprocessing time and space grow with σ^q (note that q must be an integer). The other view of using length q strings as super characters, we may say that our characters have $q \log_2 \sigma$ bits, and we want to have more control of how many bits we use. One way to achieve this is to reduce the original alphabet size σ.

We note that in theory this method cannot achieve much, as reducing the alphabet size generally only worsens the filtering capability and therefore forces larger q, but in practice this allows better fine tuning of the parameters.

What we do is that we select some $\sigma' < \sigma$, compute a mapping $\mu : \Sigma \mapsto 0, \ldots, \sigma' - 1$, and use $\mu(c)$ whenever the (filtering) algorithm needs to access some character c from the text or the pattern set. Verifications still obviously use the original alphabet.

A simple method to achieve this is to compute the histogram of character distribution of the pattern set, and assign code 0 to the most frequent character, 1 to second most frequent, and so on, and put the $\sigma' - 1, \ldots, \sigma - 1$ most frequent characters to the last bin, i.e. giving them code $\sigma' - 1$. The text characters not appearing in the patterns also will have code $\sigma' - 1$.

A better strategy is to try to distribute the original characters into σ' bins so that each bin will have (approximately) equal weight, i.e. each $\mu(c)$, where $c \in 0, \ldots, \sigma' - 1$ will have (approximately) equal probability of appearance. This is NP-hard optimization problem, so we use a simple greedy heuristic.

Alphabet mapping on the *q*-grams. We note that the above method can be applied also on the q-gram alphabet. This allows a precise control of the table size, and combined with hashing, it can accommodate very large q as well. That is, we want to

1. Choose some (possibly very large) q;
2. compute the q-gram frequencies on the pattern set (using e.g. hashing to avoid possibly large tables);
3. choose some suitable σ', the size of the mapped q-gram alphabet;
4. use method of choice (e.g. bin-packing) to reduce the number of q-grams, i.e. map the q-grams to range $0, \ldots, \sigma' - 1$;
5. use hashing to store the mapping, along with the corresponding bitvectors needed by FAOSO.

Combined alphabet mapping and *q*-gram generation. Yet another method to reduce the alphabet is to combine the q-gram computations with some bit magic. The benefit is that the mapping tables need not to be preprocessed, and this allows further optimizations as we will see shortly. The drawback is that the quality of the mapping is worse than what is achieved with approaches like bin-packing.

Consider a (text sub-)string $S[1 \ldots q]$ over alphabet Σ of size σ. A simple way to reduce the alphabet is to consider only the ℓ low-order bits of each $S[i]$, where $\ell < \log_2 \sigma$. We can then compute $(q\ell)$-bit q-gram s simply as

$$s = (S[1] \,\&\, b) + (S[2] \,\&\, b) \ll \ell + (S[3] \,\&\, b) \ll 2\ell + \cdots + (S[q] \,\&\, b) \ll (q-1)\ell,$$

where $b = (1 \ll \ell) - 1$ and $\ll$ denotes the left shift and & the bitwise and.

The main benefit of this approach is that a sequence of shifts and adds can be often replaced by a multiplication (which can be seen as an algorithm performing just that). As an illustrative example, consider the case $\ell = 2$ and hence $b = 3$ (which coincides to DNA nicely). As an implementation detail, assume that the text is 8-bit ASCII text, and it is possible to address the text, a sequence of characters, as a sequence of 32-bit integers (which is easy e.g. in C). Then to compute a 8-bit 4-gram s we can simply do

$$s = (((x \gg 1) \,\&\, \texttt{0x03030303}) * \texttt{0x40100401}) \gg 24,$$

where x is the 32-bit integer containing the 4 chars $S[1 \ldots 4]$. Assuming 4 letter DNA alphabet, the right shift (by 1) and the (parallel) masking generate 2-bit unique (and case insensitive) codes for all 4 characters. If the alphabet is larger (some DNA sequences have rare additional symbols), those will be mapped in the same range, $0, \ldots, 3$. The multiplication then shifts and adds all those codes into an 8-bit quantity, and the final shift moves the 4-gram to the low order bits. Larger q-grams can be obtained by repeating the code.

We leave the implementation to future work.

4 Experimental results

In order to evaluate the performance of our approach, we run a few experiments, using the 200 MB versions of selected datasets (`dna`, `english` and `proteins`) from the widely used Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl/`).

We test the following algorithms:

- BNDM on q-grams (BG) [23],
- Shift-Or on q-grams (SOG) [23],
- BMH on q-grams (HG) [23],
- Rabin-Karp algorithm combined with binary search and two-level hashing (RK) [23],
- Multibom and Multibsom are variants of the Set Backward Oracle Matching algorithm [2],
- Succinct Backward DAWG Matching algorithm (SBDM) [14],
- Multi AOSO on q-Grams (MAG) (this work).

All codes were obtained from the original authors. Our MAG was implemented in C++ and compiled with `g++` version 4.8.1 with `-O3` optimization. The experiments were run on a desktop PC with an Intel i3-2100 CPU clocked at 3.1 GHz with 128 KB L1, 512 KB L2 and 3 MB L3 cache. The test machine was equiped with 4 GB of 1333 MHz DDR3 RAM and running Ubuntu 64-bit OS with kernel 3.11.0-17.

In Fig. 2 we show the results of all the listed algorithms on `english`, with a fixed pattern length m and growing number of patterns r. The used pattern lengths (one for each plot) are $\{8, 16, 32, 64\}$. Note that some algorithms (or rather their available implementations) cannot handle longer patterns ($m = 64$). Our algorithm, MAG, depends on several parameters: k, q and U. The first two were explained earlier, and U serves for an unrolling technique which reduces the number of executed conditionals in the search code (for more details, see [15]). We use two settings for MAG. In one of them we chose the best configurations of k, q and U, for each dataset and each value of r and m separately; this variant is presented as MAG-tuned. It dominates for longer patterns (32, 64) and its performance is mixed for $m = 8$ and $m = 16$. As expected, for all algorithms the search speed deteriorates with the number of patterns, and for $r = 10,000$ and relatively long patterns ($m = 32$) only MAG slightly exceeds 100 MB/s (the worst ones here, SOG and RK, are 10 times slower).

Although the "optimal" MAG settings may be found in the construction phase, assuming the patterns are randomly taken from the text, this approach is rather inelegant (and the tuning phase may be time-consuming). Therefore, we ran another test in which the parameters U and k (yet not q) are set for a particular dataset, m and r according to the following simple rules found experimentally: if the "best" value of q is greater than 5, we set $U = 8$ and $k = 1$, otherwise we set $U = 4$ and $k = 2$. The case of `english` and $m = 8$ is an exception, where $U = 8$ and $k = 1$ was

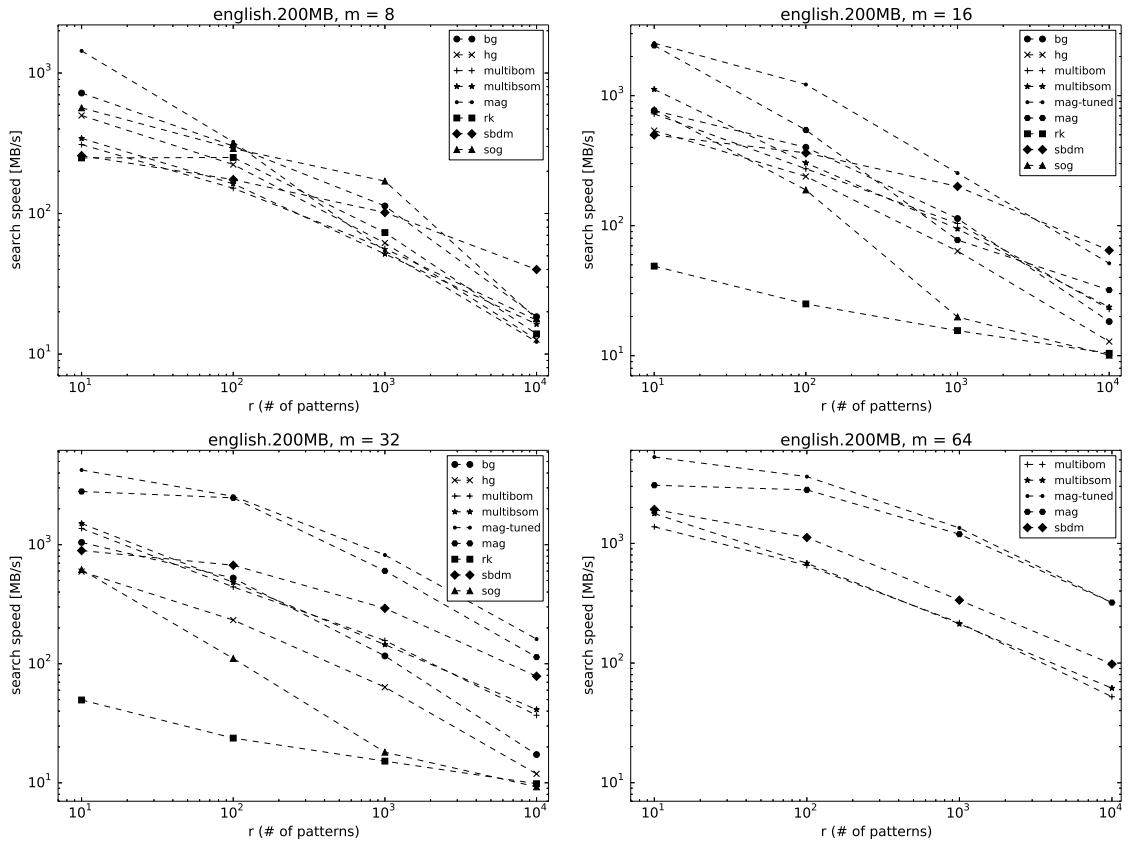


Figure 2. `english`, search speeds (MB/s) for varying number of patterns r. MAG is the same as MAG-tuned for $m = 8$, hence the "mag-tuned" points for this case are not presented.

set no matter the value of q. These results are presented on the plots as MAG. As expected, MAG is slower than MAG-tuned, but the differences are not huge.

In Fig. 3 the number of patterns r is fixed (1000), but m grows. MAG usually wins on `english` and `proteins` (except for the shortest patterns), yet is dominated by a few algorithms on `dna`. Overall, in the experiments the toughest competitor to MAG was SBDM, but in some cases the winner was SOG.

We also show how MAG performance changes with growing q (Fig. 4). As expected, larger q makes sense for large r, but a too large value of it slows down the search, presumably to many cache misses. The used MAG variant is MAG-tuned, the alphabet is quantized for all datasets. The new alphabet size, σ', was found (separately for each case) from the set $\{4, 5, 13, 14, 22\}$. Note that due to the quantization the original alphabet size does not (significantly) affect the choice of q.

5 Conclusions and future work

Multiple string matching is one of the most exploited problems in stringology. It is hard to find really novel ideas for this idea, and our work can also be seen as a new and quite successful combination of known building bricks. The presented algorithm, MAG, usually wins with its competitors on the three test datasets (`english` and `proteins`, `dna`). One of the key successful ideas was alphabet quantization (binning), which is performed in a greedy manner, after sorting the original alphabet by frequency. In the future, we are going to try other quantization techniques, also for

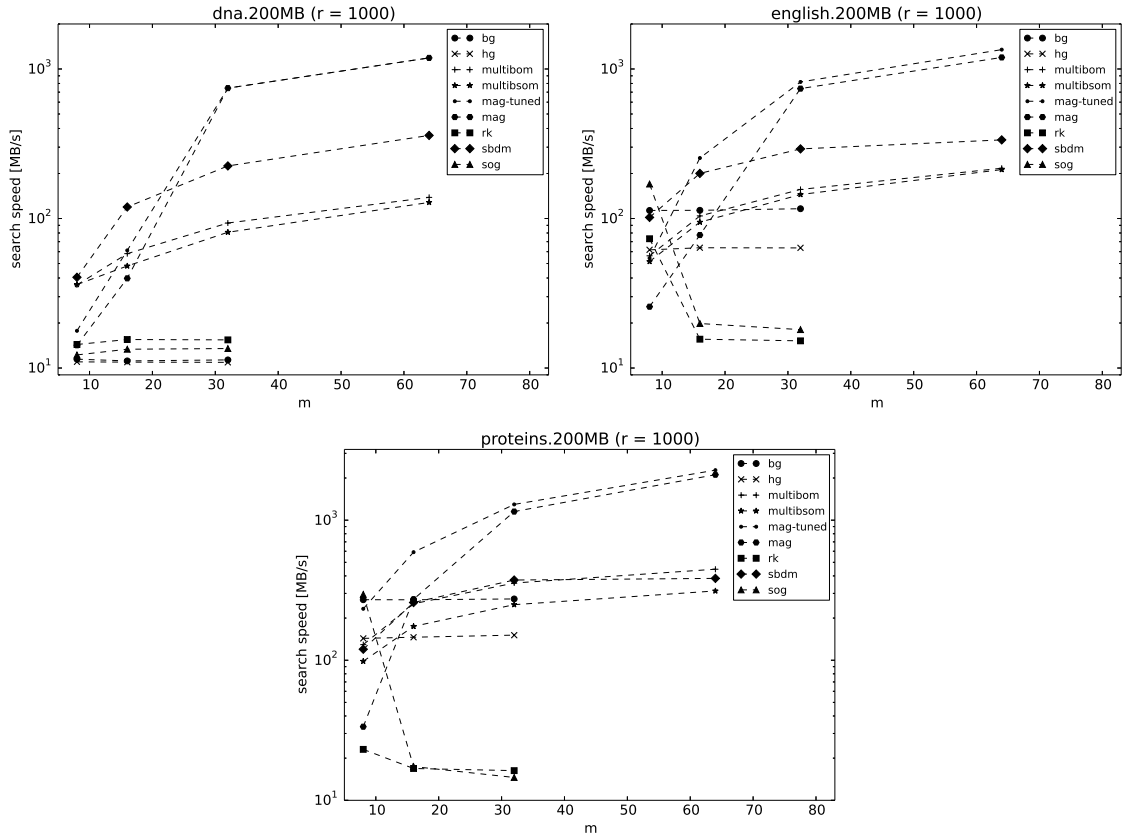


Figure 3. Search speeds for the number of patterns $r = 1000$ and varying pattern length m.

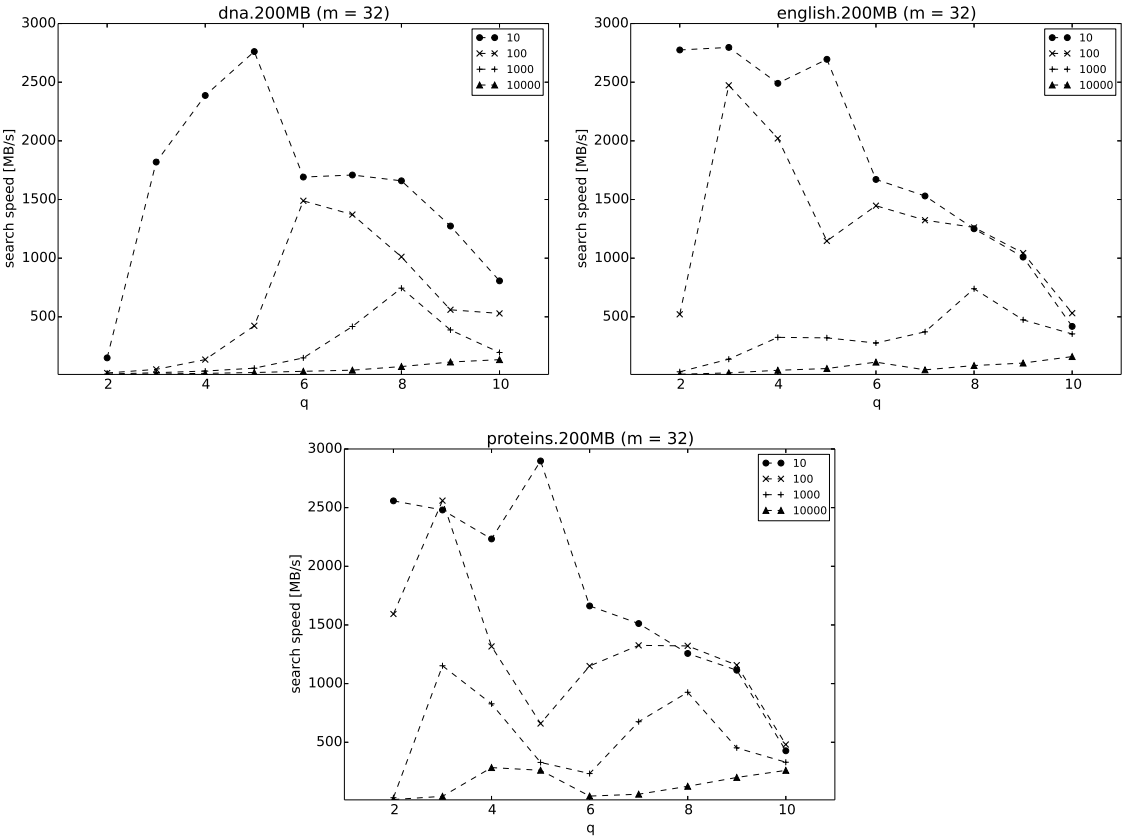


Figure 4. Search speeds for the pattern length $m = 32$ and varying q.

quantization of the alphabet built on q-grams. This could give further improvement in the algorithm performance and savings in memory consumption.

Apart from the mentioned issue, there are a number of interesting questions that we can pose here. We analytically showed that the presented approach is sublinear on average, yet not average optimal. Therefore, is it possible to choose the algorithm's parameters in order to reach average optimality (for $m = O(w)$)?

Real computers nowadays have a hierarchy of caches in their CPU-related architecture and it could be interesting to apply the I/O model (or cache-obvious model) for the multiple pattern matching problem. The cache efficiency issue may be crucial for very large pattern sets.

The underexplored power of the SIMD instructions also seems to offer great opportunities, especially for bit-parallel algorithms.

It was reported that dense codes (e.g., ETDC) for words or q-grams not only serve for compressing data (texts), but also enable faster pattern searches. Multiple pattern searching over such compressed data seems unexplored yet and it is interesting to apply our algorithm for this scenario (our preliminary results are rather promising).

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. C. Allauzen and M. Raffinot: *Factor oracle of a set of words*, Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.
3. R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
4. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
5. S. Burkhardt and J. Kärkkäinen: *Better filtering with gapped q-grams.* Fundam. Inform., 56(1-2) 2003, pp. 51–70.
6. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.
7. B. Commentz-Walter: *A string matching algorithm fast on the average*, in Proceedings of the 6th International Colloquium on Automata, Languages and Programming, H. A. Maurer, ed., no. 71 in Lecture Notes in Computer Science, Graz, Austria, 1979, Springer, pp. 118–132.
8. M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter: *Fast practical multi-pattern matching.* Inf. Process. Lett., 71(3-4) 1999, pp. 107–113.
9. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, New York, USA, 2007.
10. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
11. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming simd extensions technology*, in SPIRE, L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, eds., vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
12. M. Fisk and G. Varghese: *Fast content-based packet handling for intrusion detection*, tech. rep., DTIC Document, 2001.
13. K. Fredriksson: *Shift–or string matching with super-alphabets.* Information Processing Letters, 87(1) 2003, pp. 201–204.
14. K. Fredriksson: *Succinct backward-DAWG-matching.* J. Exp. Algorithmics, 13 2009, pp. 1.8–1.26.
15. K. Fredriksson and S. Grabowski: *Average-optimal string matching.* J. Discrete Algorithms, 7(4) 2009, pp. 579–594.
16. D. Gusfield: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
17. R. N. Horspool: *Practical fast searching in strings.* Softw., Pract. Exper., 10(6) 1980, pp. 501–506.

18. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(1) 1977, pp. 323–350.
19. G. Navarro and K. Fredriksson: *Average complexity of exact and approximate multiple string matching.* Theoretical Computer Science A, 321(2–3) 2004, pp. 283–290.
20. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM Journal of Experimental Algorithmics (JEA), 5 2000, p. article 4, 36 pages. `http://www.jea.acm.org/2000/NavarroString`.
21. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002, ISBN 0-521-81307-7. 280 pages.
22. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE2003), LNCS 2857, Springer–Verlag, 2003, pp. 80–94.
23. L. Salmela, J. Tarhio, and J. Kytöjoki: *Multipattern string matching with q-grams.* ACM Journal of Experimental Algorithmics, 11 2006.
24. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.
25. P. Yang, L. Liu, H. Fan, and Q. Huang: *Fast multi-pattern string matching algorithms based on q-grams bit-parallelism filter and hash*, in Proceedings of the 2012 International Conference on Information Technology and Software Engineering, vol. 211 of LNEE, Springer, 2013, pp. 487–495.
26. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM Journal on Computing, 8(3) 1979, pp. 368–387.

Improved Two-Way Bit-parallel Search*

Branislav Ďurian[1], Tamanna Chhabra[2], Sukhpal Singh Ghuman[2], Tommi Hirvola[2], Hannu Peltola[2], and Jorma Tarhio[2]

[1] S&T Slovakia s.r.o., Priemyselná 2, SK-010 01 Žilina, Slovakia
Branislav.Durian@snt.sk
[2] Department of Computer Science and Engineering, Aalto University
P.O.B. 15400, FI-00076 Aalto, Finland
{Tamanna.Chhabra, Suhkpal.Ghuman, Tommi.Hirvola}@aalto.fi,
hpeltola@cs.hut.fi, Jorma.Tarhio@aalto.fi

Abstract. New bit-parallel algorithms for exact and approximate string matching are introduced. TSO is a two-way Shift-Or algorithm, TSA is a two-way Shift-And algorithm, and TSAdd is a two-way Shift-Add algorithm. Tuned Shift-Add is a minimalist improvement to the original Shift-Add algorithm. TSO and TSA are for exact string matching, while TSAdd and tuned Shift-Add are for approximate string matching with k mismatches. TSO and TSA are shown to be linear in the worst case and sublinear in the average case. Practical experiments show that the new algorithms are competitive with earlier algorithms.

1 Introduction

String matching can be classified broadly as exact string matching and approximate string matching. In this paper, we consider both types. Let $T = t_1 t_2 \cdots t_n$ and $P = p_1 p_2 \cdots p_m$ be text and pattern respectively, over a finite alphabet Σ of size σ. The task of exact string matching is to find all occurrences of the pattern P in the text T, i.e. all positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_1 p_2 \cdots p_m$. Approximate string matching [14] has several variations. In this paper, we consider only the k mismatches variation, where the task is to find all the occurrences of P with at most k mismatches, where $0 \leq k < m$ holds.

We will present new sublinear variations of the widely known Shift-Or, Shift-And, and Shift-Add algorithms [3,19] which apply bit-parallelism. The key idea of the most of these algorithms is a two-way loop of j where text characters t_{i-j} and t_{i+j} are handled together. Our algorithms are linear in the worst case. Practical experiments show that the new algorithms with q-grams, loop unrolling, or with a greedy skip loop are competitive with earlier algorithms of same type.

All our algorithms utilize bit manipulation heavily. We use the following notations of the C programming language: '&', '|', '<<', and '>>'. These represent bitwise operations AND, OR, left shift, and right shift, respectively. Parenthesis and extra space has been used to clarify the correct evaluation order in pseudocodes. Let w be the register width (or word size informally speaking) of a processor, typically 32 or 64.

2 Previous algorithms

This section describes the previous solutions for exact and approximate string matching. First, we illustrate previous algorithms for exact matching which include Shift-Or and its variants like BNDM (Backward Nondeterministic DAWG Matching),

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

Branislav Ďurian, Tamanna Chhabra, Sukhpal Singh Ghuman, Tommi Hirvola, Hannu Peltola, Jorma Tarhio: Improved Two-Way Bit-parallel Search, pp. 71–83.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

TNDM (Two-way Nondeterministic DAWG Matching), LNDM (Linear Nondeterministic DAWG Matching), FSO (Fast Shift-Or) and FAOSO (Fast Average Optimal Shift-Or. Then the algorithms for approximate string matching are presented which cover Shift-Add and AOSA (Average Optimal Shift-Add).

2.1 Shift-Or and its variations

The Shift-Or algorithm [3] was the first string matching algorithm applying bit-parallelism. Processing of the algorithm can be interpreted as simulation of an automaton. The update operations to all states are identical. Operands in the algorithm are bit-vectors and the essential bit-vector containing the state of the automaton is called the state vector. The state vector is updated with the bit-shift and OR operations. FSO (Fast Shift-Or) [7] is a fast variation of the Shift-Or algorithm, and FAOSO (Fast Average Optimal Shift-Or) [7] is a sublinear variation of that algorithm.

BNDM [15] (Backward Nondeterministic DAWG Matching) is the bit-parallel simulation of an earlier algorithm called BDM (Backward DAWG Matching). BDM scans the alignment window from right to left and skips characters using a suffix automaton, which is made deterministic during preprocessing. BNDM, instead, simulates the nondeterministic automaton using bit-parallelism. BNDM applies the Shift-And method [19], which utilizes the bit-shift and AND operations.

TNDM (Two-way Nondeterministic DAWG Matching) [17] is a variation of BNDM applying two-way scanning. Our new algorithms are related to the Wide-Window algorithm [11] and its bit-parallel variations [4,11,10]. The LNDM (Linear Nondeterministic DAWG Matching) algorithm [10] is a two-way Shift-And algorithm with sequential symmetric scanning. The pseudocode of the LNDM is given as Alg. 1. The precomputed occurrence vector table B associates each character of the alphabet with a bit mask expressing occurrences of that character in the pattern P. We use table B for this purpose in all algorithms presented in this paper. In LNDM, the alignment window is shifted with fixed steps of m. Starting from the mth character of window the text characters are examined moving leftwards. The bitvector L becomes zero, when a mismatch is detected or (m shifts has been made while) m characters have

Algorithm 1 LNDM($P = p_1p_2 \cdots p_m$, $T = t_1t_2 \cdots t_n$)

Require: $m \leq w$

```
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0
 2: for i ← 1 to m do
 3:     B[p_i] ← B[p_i] | 1 << (i − 1)                          /* | 0^{m−i} 1 0^{i−1} */
    /* Searching */
 4: for i ← m step m while i ≤ n do
 5:     l ← 0; r ← 0; L ← (∼0) >> (w − m); R ← 0                /* L ← 1^m; R ← 0^m */
 6:     while L ≠ 0 do
 7:         L ← L & B[t_{i−l}]
 8:         l ← l + 1
 9:         (LR) ← (LR) >> 1
10:     R ← R >> (m − l)
11:     while R ≠ 0 do
12:         r ← r + 1
13:         if R & (1 << (m − 1)) ≠ 0 then report occurrence
14:         R ← (R << 1) & B[t_{i+r}]
```

been examined. The notation (LR) means the bitvector which is concatenated from two m bits long bitvectors L and R[1]. Next examining continues rightwards from the $m+1$ character of window. Simultaneously it is easy to notice the matches. In our two-way algorithms, these two scans are combined (into one scan). The characteristic feature in two-way algorithms is that the first characters bring plenty information to the state vector, but the last ones quite little.

2.2 Algorithms for the *k*-mismatches problem

Shift-Add [3, Fig. 8] is a bit-parallel algorithm for the k-mismatches problem. A vector of m states is used to represent the state of the search. A *field* of L bits is used for presenting each of the m states. The minimum value of L is $\lceil \log_2(k+1) \rceil + 1$. In the original Shift-Add the state i denotes the state of the search between the positions $1, \ldots, i$ of the pattern and positions $j-i+1, \ldots, j$ of the text, where j is the current position in the text.

A slightly more efficient variation of Shift-Add is (in the average case only) AOSA (Average Optimal Shift-Add) [7].

Galil and Giancarlo [9] presented a method for solving the k mismatches string matching problem in $\mathcal{O}(nk)$ time with constant time longest common extension (LCE) queries between P and T. Abrahamson [1] improved this for the case $\sqrt{(m \log m)} < k$ by giving an $\mathcal{O}(n\sqrt{m \log m})$ time algorithm based on convolutions. The asymptotically fastest algorithm known to date is given by Amir et al. [2], which achieves the worst-case time complexity of $\mathcal{O}(n\sqrt{k \log k})$. These algorithms are interesting in a theoretical sense, but in practice they perform worse than the trivial algorithm for reasonable values of m and k due to the heavy LCE and convolution operations. Hence we have the need for developing fast practical algorithms for string matching with k mismatches.

3 TSO and TSA

3.1 TSO

At first we introduce a new Two-way Shift-Or algorithm, TSO for short. The pseudocode of TSO is given as Alg. 2. TSO uses the same occurrence vectors B for characters as the original Shift-Or. The outer loop traverses the text with a fixed step of m characters. At each step i, an alignment window $t_{i-m+1}, \ldots, t_{i+m-1}$ is inspected. The positions $t_i, \ldots, t_{i+m-1}$ correspond to the end positions of possible matches and at the same time, to the positions of the state vector D. Inspection starts at the character t_i, and it proceeds with a pair of characters t_{i-j} and t_{i+j} until corresponding bits in D become 1^m or $j = m$ holds. Note that the two consecutive loops of LNDM are combined in TSO into one loop (lines 8–10 of Alg. 2). When the actually used bits in bit-vectors are seated to the highest order bits, in TSO the testing of the state vector D is slightly faster than in elsewhere.

Moreover one bit in D stays zero for each occurrence of the pattern in the inner loop on lines 8–10. The zero bits are switched to set bits on line 12. The count of set

[1] So in the right shift the lowest bit of L becomes the highest bit of R. Note that generally this is different from how e.g. gcc compiler handles this way variables of uint64_t type in x86 architecture in 32-bit mode. See also [12, p. 35].

Algorithm 2 TSO = Two-way Shift-Or($P = p_1p_2 \cdots p_m$, $T = t_1t_2 \cdots t_n$)

```
Require: m ≤ w
    /* Preprocessing */
 1: mask ← ∼0 << (w − m)                                   /* = 1^m 0^{w−m} */
 2: for all c ∈ Σ do B[c] ← mask
 3: for i ← 1 to m do /* Lowest bits remain 0 */
 4:       B[p_i] ← B[p_i] & ∼(1 << (w − m + i − 1))        /* & 1^{m−i} 0 1^{w−m+i−2} */
    /* Searching */
 5: matches ← 0
 6: for i ← m step m while i ≤ n do
 7:       D ← B[t_i]; j ← 1
 8:       while D < mask and j < m do
 9:            D ← D | (B[t_{i−j}] << j) | (B[t_{i+j}] >> j)      /* no need for additional masking */
10:            j ← j + 1
11:       if D < mask then /* Garbage is in the lowest bits */
12:            E ← (∼D) & mask
13:            matches ← matches + popcount(E)
```

bits is then calculated with the popcount[2] function [12] on line 13. An easy realization of popcount is the following:

$$\textbf{while } E > 0 \textbf{ do } matches \leftarrow matches + 1;\ E \leftarrow (E-1)\&E$$

This requires $\mathcal{O}(s)$ time in total where s is the number of occurrences. If the locations of occurrences need to be printed out, $\mathcal{O}(m)$ time is needed for every alignment window holding at least one match.

Alg. 2 works correctly when $n \bmod m = m - 1$ holds. If access to $t_{n+1}, \ldots$ is allowed and some character—e.g. 255—does not appear in P, assignment of stopper $t_{n+1} \leftarrow 255$ makes the algorithm work also for other values of n. Another easy way of handling the end of the text is to use Shift-Or algorithm, because same occurrence vectors are disposable.

In Figure 1, there is an example of the execution of TSO for $P =$ `abcab` and $T = \cdots$`xabcabcabx`$\cdots$.

3.2 TSA

Shift-And is a dual method of Shift-Or. Therefore it is fairly straight-forward to modify TSO to a Two-way Shift-And algorithm, TSA for short. The pseudocode of TSA is given as Alg. 3.

In TSA, $B[t_{i-j}]$ and $B[t_{i+j}]$ are brought to state vector on line 8. For example, let $B[t_{i-2}]$ and $B[t_{i+2}]$ be 1010 and 1011, respectively. (In this example and in the subsequent examples all numbers are binary numbers.) Then the corresponding padded bit strings are $((1010+1) << 2) - 1 = 101011$ and $(1011 >> 2) \,|\, 1111 << 2 = 111110$.

Original Shift-Or/Shift-And examines every text character once. Therefore its practical performance is extremely insensitive to the input data. Two-way algorithms check text in alignment windows of m consecutive text positions. A mismatch can be detected immediately based on the first examined text character. In the best case the performance can be $\Theta(n/m)$. On the other hand, if a match is in any position in

[2] Population count, popcount, counts the number of 1-bits in a register or word. On many computers it is a machine instruction; e.g. in Sparc, and in x86_64 processors in AMDs SSE4a extensions and in Intel's SSE4.2 instruction set extension.

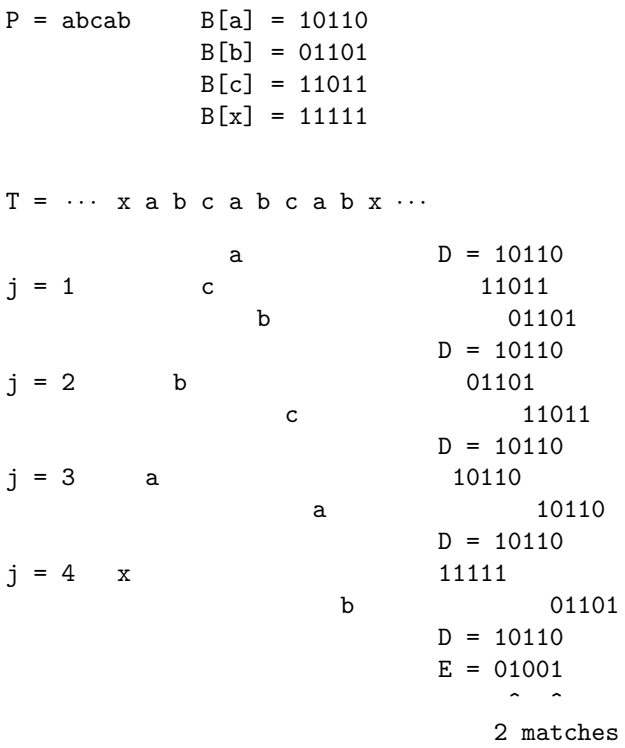


Figure 1. Example of work made in the inner loop of TSO.

Algorithm 3 TSA = Two-way Shift-And($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $m \leq w$

/* Preprocessing */

1: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow 0$

2: **for** $i \leftarrow 1$ **to** m **do**

3: $B[p_i] \leftarrow B[p_i] \mid 1 << (m-i)$ /* $\mid 0^{i-1}10^{m-i}$ */

/* Searching */

4: $matches \leftarrow 0$

5: **for** $i \leftarrow m$ **step** m **while** $i \leq n$ **do**

6: $D \leftarrow B[t_i]$; $j \leftarrow 1$

7: **while** $(D > 0)$ **and** $(j < m)$ **do** /* alternatively $D \neq 0$ */

8: $D \leftarrow D$ & $(((B[t_{i-j}] + 1) << j) - 1)$ & $((B[t_{i+j}] >> j) \mid (((\sim 0) >> (w-m)) << (m-j)))$ /* $(1^m << (m-j))$ */

9: $j \leftarrow j + 1$

10: **if** $D > 0$ **then** /* alternatively $D \neq 0$ */

11: $matches \leftarrow matches + popcount(D)$

the window, or if the mismatch is detected based on two last examined characters, then $2m - 1$ characters need to be examined. So in the worst case all text characters except the last characters in each alignment window are examined twice.

3.3 Practical optimizations

In modern processors, loop unrolling often improves the speed of bit-parallel searching algorithms [5]. In the case of TSO and TSA, it means that 3, 5, 7, or 9 characters are read in the beginning of the inner loop instead of a single character. We denote these versions by TSOx and TSAx, where x is the number of characters read in the beginning; x is odd. Line 7 of TSO3 is the following:

7: $D \leftarrow (B[t_{i-1}] \ll 1) \mid B[t_i] \mid (B[t_{i+1}] \gg 1)$; $j \leftarrow 2$

Moreover, the shifted values $B[a] \ll 1$ and $B[a] \gg 1$ can be stored to precomputed arrays in order to speed up access.

Many string searching algorithms apply a so called skip loop, which is used for fast scanning before entering the matching phase. The skip loop can be called greedy, if it handles two alignment windows at the same time [18]. Let us denote

$$(B[t_{i-1}] \ll 1) \mid B[t_i] \mid (B[t_{i+1}] \gg 1)$$

in TSO3 by $f(3, i)$. If the programming language has the short-circuit AND command, then we can use the following greedy skip loop enabling steps of $2m$ in TSO3:

$$\textbf{while } f(3, i) = mask \;\&\&\; f(3, i + m) = mask \textbf{ do } i \leftarrow i + 2 \cdot m$$

Because && is the short-circuit AND, the second condition is evaluated only if the first condition holds. The resulting version of TSO3 is denoted by GTSO3. (Initial G comes from greedy. GTSA3 is formed in a corresponding way.)

3.4 Analysis

We will show that TSO is linear in the worst case and sublinear in the average case. For simplicity we assume in the analysis that $m \leq w$ holds and w is divisible by m.

The outer loop of TSO is executed n/m times. In each round, the inner loop is executed at most $m-1$ times. The most trivial implementation of popcount requires $\mathcal{O}(m)$ time. So the total time in the worst case is $\mathcal{O}(nm/m) = \mathcal{O}(n)$.

When analyzing the average case complexity of TSO, we assume that the characters in P and T are statistically independent of each other and the distribution of characters is discrete uniform. We consider the time complexity as the number of read characters.

In each window, TSO reads $1 + 2k$ characters, $0 \leq k \leq m - 1$, where k depends on the window. Let us consider algorithms TSOr, $r = 1, 2, 3, \ldots$, such that TSOr reads an r-gram in the window before entering the inner loop. For odd r, TSOr was described in the previous section. For even r, TSOr is modified from TSO$(r-1)$ by reading $t_{i-r/2}$ before entering the inner loop. It is clear that TSOr_2 reads at least as many characters as TSOr_1, if $r_2 > r_1$ holds. Let us consider TSOr as a filtering algorithm. The reading of an r-gram and computing D for it belong to filtration and the rest of the computation is considered as verification. The verification probability is $(m - r + 1)/\sigma^r$. The verification cost is in the worst case $\mathcal{O}(m)$, but only $\mathcal{O}(1)$ on average. The total number of read characters is rn/m in filtration. When we select r to be $\log_\sigma m$, TSOr is sublinear. Because TSOr never reads less characters than TSO1 = TSO, we conclude that also TSO is sublinear.

In other words, the time complexity of TSO is optimal $\mathcal{O}(n \log_\sigma m/m)$ with a proper choice of r for $m = \mathcal{O}(w)$ and $\mathcal{O}(n \log_\sigma m/w)$ for larger m.

The time complexity of preprocessing of TSO is $\mathcal{O}(m+\sigma)$. Because of the similarity of TSO and TSA, TSA has the same time complexities as TSO. The space requirement of both algorithms is $\mathcal{O}(\sigma)$.

4 Variations of Shift-Add

4.1 Two-way Shift-Add

The basic idea in Shift-Add algorithm is to simultaneously evaluate the number of mismatches in each inside field using L bits. The highest bit in each field is an

overflow bit, which is used in preventing the error count rolling to the next field. The original Shift-Add algorithm actually used two state vectors, *State* and *Overflow* which were shifted L bits forward. Opposite this, two-way approach in exact matching is successful due to simple (one statement) analogy to the one-way algorithm (Shift-Or, Shift-And). Such an improved (one statement) Shift-Add is introduced in the next section.

The core problem is addition; there can be up to m mismatches. When in some position k errors is reached, we should stop addition into it. In the occurrence vector array, $B[\,]$, only the lowest bit in each field may be set. The key trick is to use the overflow bits in the state vector D. We take the logical AND operation between the applied occurrence vector and the $L-1$ right shifted complemented state vector D. Then the complemented overflow bits and the possibly set bits in the occurrence vector are aligned, and addition happens only when there is no overflow.

This idea is applied in the Two-way Shift-Add$q$ The limitation of Two-way Shift-Add on error level $k=0$ is that each field needs 2 bits.

When bit-vectors are aligned to the lowest order bits, the unessential bits in the right shifted occurrence vector fall off immediately, and in the right shifted ones they do not disturb because bit-vectors are unsigned.

On line 12 the shown form is required with character classes [16, p. 78]; otherwise also substraction works. The form of line 14 depends on q as before. Notice that there can happen larger overflows, but as long as $k \le q$ it does not matter; otherwise we need a larger value for L. Then the minimum value of L is $\lceil \log_2(q+1) \rceil + 1$.

Algorithm 4 Two-way Shift-Add$q(P = p_1p_2\cdots p_m,\ T = t_1t_2\cdots t_n,\ k)$

Require: $m \cdot L \le w$ and $L \ge \max\{2, \lceil \log_2(\max\{k,q\}+1) \rceil + 1\}$ and $m > (q+1)$ `div` 2

```
    /* Preprocessing */
 1: mask ← 0
 2: for i ← 1 to m do
 3:       mask ← (mask << L) | ((1 << (L − 1)) − k)
 4: for all c ∈ Σ do BW[c] ← mask
 5: mask ← 0
 6: for i ← 1 to m do
 7:       mask ← (mask << L) | 1
 8: for all c ∈ Σ do B[c] ← mask                          /* mask = (0^{L−1} 1_2)^{m−1} */
 9: mask ← mask << (L − 1)                                /* mask = (1 0_2^{L−1})^{m−1} */
10: for i ← 1 to m do
11:       BW[p_i] ← BW[p_i] − (1 << L · (i − 1))
12:       B[p_i] ← B[p_i] & ∼(1 << L · (i − 1))     /* − (1 << L · (i − 1)) also works normally */
    /* Searching */
13: for i ← m step m while i ≤ n do
14:       D ← BW[t_i] + (B[t_{i−1}] << L) + (B[t_{i+1}] >> L)        /* this one is for q = 3 */
15:       j ← (q+1) div 2                          /* integer division – values of q are odd */
16:       while j < m and (∼D) & mask do
17:             D ← D + (∼D >> (L − 1)) & B[t_{i−j}] << (L · j)
                        + (∼D >> (L − 1)) & B[t_{i+j}] >> (L · j)
18:             j ← j + 1
19:       E ← (∼D) & mask
20:       while E do
21:             report an occurrence                  /* shifting of E is not needed */
22:             E ← E & (E − 1)                       /* turning off rightmost 1-bit */
```

Figure 2 shows an example how Two-way Shift-Add finds a match. Unrelevant bits are not shown; they are all zeros. On each field (of L bits) in D the highest bit is an overflow bit, which indicates that there is no match on the corresponding text position. Vertical lines limit the computing area having interesting bit fields.

T	=	a	b	a	d	a	c	a	d	c	$\cdots$
P	=	b	a	c	a	c					
k	=	1									
L	=	3									One bit unnecessarily large
B[a]	=	001	000	001	000	001					Shown order of bit fields corresponds to P backwards
B[b]	=	001	001	001	001	000					Occurrences = 000
B[c]	=	000	001	000	001	001					
B[d]	=	001	001	001	001	001					As all other characters that do not appear in P
BW[a]	=	011	010	011	010	011					Again P backwards
BW[b]	=	011	011	011	011	010					011 minus number of errors still allowed
BW[c]	=	010	011	010	011	011					
BW[d]	=	011	011	011	011	011					
$BW[t_5] = BW$[a]	=	011	010	011	010	011					Starting to check next m positions
$+B[t_4] = B$[d] $<<$ 3	= 001	001	001	001	001						
$+B[t_6] = B$[c] $>>$ 3	=		000	001	000	001	001				Starting with $q = 3$ characters
$= D$	=	100	011	101	011	100					Note that overflow depends on q
$+B[t_3] = B$[a] $<<$ 6	= 000	001	000	001							Only lowest bits in fields may be set
$\& \sim D >> (L-1)$	=	0	1	0	1	0					So only the overflow bit is relevant on each field
$+B[t_7] = B$[a] $>>$ 6	=			001	000	001	000	$\cdots$			
$\& \sim D >> (L-1)$	=	0	1	0	1	0					
$= D$	=	100	011	101	011	100					Second and fourth position look promising
$+B[t_2] = B$[b] $<<$ 9	= 001	001	000								
$\& \sim D >> (L-1)$	=	0	1	0	1	0					
$+B[t_8] = B$[d] $>>$ 9	=				001	001	001	$\cdots$			
$\& \sim D >> (L-1)$	=	0	1	0	1	0					
$= D$	=	100	011	101	100	100					Overflow also in fourth position
$+B[t_1] = B$[b] $<<$ 12	= 001	000									
$\& \sim D >> (L-1)$	=	0	1	0	0	0					
$+B[t_9] = B$[c] $>>$ 12	=					000	001	$\cdots$			Last characters give only little information
$\& \sim D >> (L-1)$	=	0	1	0	0	0					
$= D$	=	100	011	101	100	100					
E	=	0	1	0	0	0					Match in second position

Figure 2. Example of checking m positions in Two-way Shift-Add.

4.2 Analysis

The worst case analysis is similar to the analysis of TSO/TSA given in subsection 3.4. For simplicity we assume in the analysis that $m \leq w$ holds and w is divisible by m. The outer loop of TSAdd$q$ is executed n/m times, and in each iteration $\mathcal{O}(m)$ text characters are read and $\mathcal{O}(m)$ occurrences are reported. Thus, the total time complexity is $\mathcal{O}(n/m) \cdot \mathcal{O}(m+m) = \mathcal{O}(n)$ for the worst case.

On the average case TSAdd$q$ is sublinear. It can been seen from the test results where the search time decreases when m gets larger.

4.3 Tuned Shift-Add

Algorithm 5 is Tuned Shift-Add. It is a minimalist version of Shift-Add algorithm. If bitvectors fit into computer register, the worst- and average-case complexity of the original Shift-Add algorithm $\mathcal{O}(n)$ [3, p. 75]; also Tuned Shift-Add is linear. The original Shift-Add algorithm is using an *overflow* vector in addition to the state vector (here D). The essential difference between the original Shift-Add algorithm and the Tuned Shift-Add is the state update. Using the same variable naming as in the Tuned Shift-Add the line 11 in Tuned Shift-Add was in original Shift-Add as follows. (Overflow bits are in the *ovmask*; only the highest bit in each bit field is set.)

$D \leftarrow \big((D << L) \ + \ BW[t_i]\big) \ \& \ mask2$
$overflow \leftarrow \big((overflow << L) \mid (D \ \& \ ovmask)\big) \& \ mask2$
$D \leftarrow D \ \& \ \sim ovmask$ /∗ clears overflow bits ∗/

Algorithm 5 Tuned Shift-Add($P = p_1p_2 \cdots p_m$, $T = t_1t_2 \cdots t_n$, k)

Require: $m \cdot L \leq w$ and $L \geq \max\{2, \lceil \log_2(k+1) \rceil + 1\}$
/* Preprocessing */
1: $mask \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** m **do**
3: $mask \leftarrow (mask << L) \mid 1$
4: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow mask$
5: **for** $i \leftarrow 1$ **to** m **do**
6: $B[p_i] \leftarrow B[p_i] \ \& \ \sim\big(1 << L \cdot (i-1)\big)$ /* $-\big(1 << L \cdot (i-1)\big)$ also works normally */
7: $mask \leftarrow 1 << (L \cdot m - 1)$
8: $Xmask \leftarrow \big(1 << (L-1)\big) - (k+1)$
/* Searching */
9: $D \leftarrow \sim 0$ /* $= 1_2^w$ */
10: **for** $i \leftarrow 1$ **to** n **do**
11: $D \leftarrow \big((D << L) \mid Xmask\big) + \big(B[t_i] \ \& \ (\sim(D << 1))\big)$
12: **if** $(D \& mask) = 0$ **then**
13: report an occurrence ending at i

5 Experiments

The tests were run on Intel Core i7-860 2.8GHz, 4 cores, with 16GiB memory; L2 cache is 256KiB / core and L3 cache: 8MiB. The computer is running Ubuntu 12.04 LTS, and has gcc 4.6.3 C compiler. Programs were written in the *C* programming language and compiled with gcc compiler using `-O3` optimization level. All the algorithms were implemented and tested in the testing framework[3] of Hume and Sunday [13]. New

[3] Hume and Sunday test framework allows directly and precisely measure preprocessing times. Test pattern can be selected as considered appropriate. This kind testing method where each algorithm is coded and separately ensures that the tested algorithms can not affect to each other by placement of data structures in memory and data cache. We have tested the search speed of e.g. Sunday's algorithm and various Boyer–Moore variations with implementations made by others. Thus we believe that implementations enclosed in Hume and Sunday test framework are very efficient. This kind of comparison makes it also possible to learn coding of efficient implementations. We encourage everybody to make comparisons with different implementations of same and similar algorithms.

algorithms were compared with the following earlier algorithms: Shift-Or[4] (SO) [3], FSO [7], FAOSO [7], BNDM [15], and LNDM [10]. The given run times of FAOSO are based on the best possible parameter combination for each text and pattern length. We have only 32 bit version of FAOSO, but all other tested algorithms were using 64-bit bit-vectors. For longer patterns than roughly 20 characters there are algorithms [6] which are faster than ones used in here. The results for pattern lengths are shown to demonstrate the behavior of the new algorithms.

We did not test the variations [4] of the Wide-Window algorithm [11], because according to the original experiments [4], these algorithms are only slightly better than BNDM. In addition, they require $m \leq w/2$.

In the test runs we used three texts: binary, DNA, and English, the size of each is 2 MB. The English text is the prefix of the KJV Bible. The binary text is a random text in the alphabet of two characters. The DNA text is from the genome of fruitfly (*Drosophila melanogaster*). Sets of patterns of various lengths were randomly taken from each text. Each set contains 200 patterns.

Data	Algorithm	2	4	8	12	16	20	30	40	50	60
Binary	SO	465	465	465	465	465	465	466	469	466	465
	FSO	1406	707	268	241	234	234	235	236	235	—
	FAOSO	3522	1728	859	745	695	469	372	239	263	239
	BNDM	1892	1579	1059	723	554	452	316	246	201	171
	LNDM	2814	2291	1573	1166	925	767	544	421	346	294
	TSA	1999	1501	927	641	491	399	276	215	177	152
	TSO	1565	1129	673	455	344	279	188	142	114	96.4
	TSO3	1429	1158	718	502	385	316	219	172	142	122
	TSO5	1704	911	632	462	359	297	207	161	135	116
	TSO9	1881	771	473	342	272	229	172	141	121	109
	GTSO3	1409	1165	719	499	381	313	217	169	139	121
	GTSA3	1529	1281	819	571	441	362	252	195	163	141

Table 1. Search time of algorithms (in milliseconds) for binary data

Tables 1–3 show the search times in milliseconds for these data sets. Before measuring the CPU time usage, the text and the pattern set were loaded to the main memory, and so the execution times do not contain I/O time. The results were obtained as an average of 100 runs. During repeated tests, the variation in timings was about 1 percent. The best execution times have been put in boxes. Overall, TSO9, TSO5 and GTSO3 appears to be the fastest for binary, DNA and English data respectively.

Table 1 presents run times for binary data. SO is the winner for $m \leq 4$, FSO for $8 \leq m \leq 16$, TSO9 for $20 \leq m \leq 50$, and TSO9 for $m \geq 60$. Table 2 presents run times for DNA data. FAOSO is the winner for $m = 2$, FSO for $m = 4$, TSO5 for $8 \leq m \leq 40$, and TSO9 for $m \geq 50$. Table 3 presents run times for English data. FSO is the winner for $m \leq 4$, GTSO3 for $8 \leq m \leq 16$, GTSA3 for $m = 20$, and TSO5 for $m \geq 30$.

[4] The performance of the Shift-Or algorithm is insensitive to the pattern length (when $m \leq w$) and also to the input data as long as the number of the matches is relative moderate. The relative speed of some algorithm compared to the speed of Shift-Or on given data and pattern length is suitable for comparing tests with similar data. This relative speed is useful for comparing roughly performance of exact string matching algorithms with different text lengths and processors even in different papers.

Data	Algorithm	2	4	8	12	16	20	30	40	50	60
Dna	SO	464	465	465	464	465	465	465	469	465	465
	FSO	709	272	235	235	234	235	235	234	235	—
	FAOSO	372	639	524	331	311	212	185	216	217	213
	BNDM	1496	984	548	385	302	248	175	134	111	93.4
	LNDM	2255	1438	843	609	481	398	281	219	181	154
	TSA	1481	869	498	355	285	241	179	143	119	103
	TSO	1353	757	364	243	192	161	117	90.9	74.6	62.7
	TSO3	758	491	295	225	189	164	128	106	89.3	79.8
	TSO5	992	401	215	153	121	102	78.1	65.6	60.9	56.1
	TSO9	1217	465	242	168	131	109	81.1	66.4	57.6	52.3
	GTSO3	753	474	289	223	191	167	132	107	92.4	74.7
	GTSA3	747	486	296	228	193	169	135	111	96.9	85.3

Table 2. Search time of algorithms (in milliseconds) for DNA data

Data	Algorithm	2	4	8	12	16	20	30	40	50	60
English	SO	465	465	465	465	464	465	464	464	465	465
	FSO	328	246	235	234	234	234	234	234	232	—
	FAOSO	1165	307	167	156	142	141	198	199	195	198
	BNDM	651	505	342	252	198	164	115	93.3	78.0	68.3
	LNDM	1398	903	561	412	326	272	194	154	126	109
	TSA	1243	652	348	245	195	168	121	99.7	87.1	78.4
	TSO	701	518	328	231	176	141	92.3	69.5	56.6	48.9
	TSO3	485	274	159	121	104	89.1	72.9	64.8	59.1	56.1
	TSO5	701	341	184	132	105	88.9	67.1	58.9	54.6	49.6
	TSO9	924	448	235	165	128	107	79.3	64.6	57.7	52.4
	GTSO3	449	249	149	115	96.5	86.6	71.8	63.4	57.6	52.8
	GTSA3	441	252	151	116	97.4	86.4	72.1	65.1	58.2	53.7

Table 3. Search time of algorithms (in milliseconds) for English data

5.1 Experiments for *k*-mismatches problem

For the k-mismatch problem we tested the following algorithms: Shift-Add (SAdd), Two-way Shift-Add with q-values 1, 3, and 5 (TSAdd-1, TSAdd-3, TSAdd-5), Tuned Shift-Add (TuSAdd), Average Optimal Shift-Add (AOSA), and CMFN. CMFN is a sublinear multi-pattern algorithm by Fredriksson and Navarro [8], and it is also suitable for approximate circular pattern matching problem.

The text files are same as before. The binary pattern set for $m = 5$ contains only 32 patterns, all different. To make the results comparable with other pattern sets containing 200 patterns, the timings have been multiplied with 200/32. The results were obtained as an average of 300 runs.

Programs were written in the *C* programming language and compiled with gcc compiler using `-O2` optimization level. During preliminary tests we noticed performance decrease in AOSA, which seems to be related to the optimization level in gcc compiler. For example on error level $k = 1$ and optimization `-O2` the search speed was 22%–52% faster than with here used `-O3`.

Tables 4–6 represent the results for the k-mismatches problem.

In our tests the Tuned Shift-Add was faster than the original Shift-Add. Both seem to suffer from relatively large number of occurrences. On $k = 1$ TSAdd-3 showed best performance on all other data set except on 5 nucleotide long DNA patterns. (This

	m	TSAdd-1	TSAdd-3	TuSAdd	SAdd	AOSA	CMFN
English	5	177	137	149	231	229	880
	10	98	77	145	228	115	270
	20	53	43	145	228	51	113
	30	37	30	145	228	38	93
DNA	5	226	225	165	246	267	2770
	10	136	114	145	228	164	1420
	20	69	58	145	228	92	1810
	30	47	39	145	227	62	3083
Bin	5	333	167	625	937	937	1062
	10	167	77	603	966	966	440
	20	83	39	600	947	467	240
	30	57	30	593	943	317	140

Table 4. Search times of algorithms (in milliseconds) for $k = 1$.

	m	TSAdd-1	TSAdd-3	TSAdd-5	TuSAdd	SAdd	AOSA	CMFN
English	5	238	201	186	161	245	253	2807
	10	124	107	101	145	230	137	533
	20	65	56	51	147	216	73	223
DNA	5	322	280	268	255	339	497	4203
	10	176	158	151	147	239	225	3183
	20	88	79	69	146	214	113	3563
Bin	5	354	146	270	625	958	937	5688
	10	167	73	127	642	962	947	800
	20	82	46	67	611	941	470	350

Table 5. Search times of algorithms (in milliseconds) for $k = 2$.

	m	TSAdd-1	TSAdd-3	TSAdd-5	TuSAdd	SAdd	AOSA	CMFN
English	5	299	259	247	209	291	377	3936
	10	155	137	133	145	236	297	1128
	20	78	70	67	145	217	107	292
DNA	5	357	316	310	447	536	1073	4290
	10	215	196	194	151	241	238	4900
	20	108	99	98	148	215	128	5293
Bin	5	333	146	250	604	937	917	5524
	10	160	77	120	580	910	893	808
	20	83	42	61	580	917	450	300

Table 6. Search times of algorithms (in milliseconds) for $k = 3$.

test was rerun, but results remained about the same.) TSAdd-3 was best on all tests using binary text. On English and DNA texts for $k = 2$ and $k = 3$ TSAdd and TuSAdd were the best.

To our surprise CMFN was not competitive in these tests. The macro `bitvector` was defined `unsigned long long`, but we suspect that some other compilation parameter was unoptimal.

6 Concluding remarks

We have presented two new bit-parallel algorithms based on Shift-Or/Shift-And and Shift-Add techniques for exact string matching. The compact form of these algorithms is an outcome of a long series of experimentation on bit-parallelism. The new algorithms and their tuned versions are efficient both in theory and practice. They run in linear time in the worst case and in sublinear time in the average case. Our experiments show that the best ones of the new algorithms are in most cases faster than the previous algorithms of the same type.

References

1. K. Abrahamson: *Generalized string matching.* SIAM Journal on Computing, 16(6) 1987, pp. 1039–1051.
2. A. Amir, M. Lewenstein, and E. Porat: *Faster algorithms for string matching with k mismatches.* Journal of Algorithms, 50(2) 2004, pp. 257–275.
3. R. Baeza-Yates and G. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
4. D. Cantone, S. Faro, and E. Giaquinta: *Bit-(parallelism)^2: Getting to the next level of parallelism*, in Fun with Algorithms, 5th International Conference, FUN 2010, June 2-4, 2010. Proceedings, P. Boldi and L. Gargano, eds., vol. 6099 of LNCS, Springer, 2010, pp. 166–177.
5. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Information Processing Letters, 110(4) 2010, pp. 148–152.
6. B. Durian, H. Peltola, L. Salmela, and J. Tarhio: *Bit-parallel search algorithms for long patterns*, in Experimental Algorithms, 9th International Symposium, SEA 2010, May 20-22, 2010. Proceedings, P. Festa, ed., vol. 6049 of LNCS, Springer, 2010, pp. 129–140.
7. K. Fredriksson and S. Grabowski: *Practical and optimal string matching*, in International Symposium on String Processing and Information Retrieval, SPIRE, LNCS, vol. 12, 2005.
8. K. Fredriksson and G. Navarro: *Average-optimal single and multiple approximate string matching.* ACM Journal of Experimental Algorithmics, 9 2004.
9. Z. Galil and R. Giancarlo: *Improved string matching with k mismatches.* SIGACT NEWS 62, 17(4) 1986, pp. 52–54.
10. L. He and B. Fang: *Linear nondeterministic dawg string matching algorithm*, in String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, October 5-8, 2004, Proceedings, A. Apostolico and M. Melucci, eds., vol. 3246 of LNCS, Springer, 2004, pp. 70–71.
11. L. He, B. Fang, and J. Sui: *The wide window string matching algorithm.* Theoretical Computer Science, 332 2005.
12. J. Henry and S. Warren: *Hacker's Delight*, Addison-Wesley, 2003.
13. A. Hume and D. Sunday: *Fast string searching.* Software—Practice and Experience, 21(11) 1991, pp. 1221–1248.
14. G. Navarro: *A guided tour to approximate string matching.* ACM Computing Surveys, 33(1) 2001, pp. 31–88.
15. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM Journal of Experimental Algorithmics, 5(4) 2000.
16. G. Navarro and M. Raffinot: *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
17. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in International Symposium on String Processing and Information Retrieval, SPIRE, LNCS, vol. 10, 2003, pp. 80–93.
18. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163(3) 2014, pp. 352–360.
19. S. Wu and U. Manber: *Fast text searching allowing errors.* Communications of the ACM, 35(10) 1992, p. 83.

Using Correctness-by-Construction to Derive Dead-zone Algorithms

Bruce W. Watson[1], Loek Cleophas[2], and Derrick G. Kourie[1]

[1] FASTAR Research Group, Department of Information Science, Stellenbosch University,
Private Bag X1, 7602 Matieland, Republic of South Africa
[2] Department of Computer Science, University of Pretoria,
Private Bag X20, 0028 Hatfield, Pretoria, Republic of South Africa
{bruce,loek,derrick}@fastar.org

Abstract. We give a derivation, in the form of a stepwise (refinement-oriented) presentation, of a *family* of algorithms for single keyword pattern matching, all based on the so-called *dead-zone* algorithm-style, in which input text parts are tracked as either unprocessed ('live'), or processed ('dead'). Such algorithms allow for Boyer-Moore-style shifting in the input in two directions (left and right) instead of one, and have shown promising results in practice. The algorithms are the more interesting because of their potential for concurrency (multithreading). The focus of our algorithm family presentation is on correctness-arguments (proofs) accompanying each step, and on the resulting elegance and efficiency. Several new algorithms are described as part of this algorithm family, including ones amenable to using concurrency.

Keywords: correctness-by-construction, algorithm derivation, keyword pattern matching, Boyer-Moore, concurrency

1 Introduction

In this paper, we give a stepwise derivation of a *family* of algorithms for single keyword pattern matching. The focus of the derivation is on clarity and confidence in the correctness of each step, which lead to efficiency and elegance. Because of the correctness arguments associated with each step, the presentation forms the essence of a *derivation* of the various algorithms. As such, the presentation forms a case study for the *Correctness-by-Construction* (CbC) approach to software or algorithm construction: we start with an abstract problem specification, in the form of pre- and postcondition, and iteratively refine these to obtain more refined (concrete) specifications [15]. In its strict form, the use of CbC requires that in each refinement step, rules are applied that guarantee and prove correctness of the resulting refined specification. For presentation clarity, and since many of the proof obligations involved are trivial, we will leave out many formal details in the present context. A CbC approach, and the derivations resulting from it, give a clear understanding of the algorithms and concepts involved, and give confidence in correctness, even if applied somewhat informally. Furthermore, new algorithms may arise from the process.

The main result of our presentation is the *derivation* of a substantial and partially new algorithm family, of which significant parts have not been explicitly presented before. In some sense, the algorithms in the family are reminiscent of the Boyer-Moore style of algorithms, in which *shift functions* are used to potentially skip reading parts of the subject string by moving to the right in the input text by *more than one* position. This has earned such algorithms the term *sublinear*.

The algorithm family we consider uses quite different algorithm skeletons from the Boyer-Moore style ones, although the latter form degenerate cases within our

Bruce W. Watson, Loek Cleophas, Derrick G. Kourie: Using Correctness-by-Construction to Derive Dead-zone Algorithms, pp. 84–95.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

algorithm family. It is based on characterizing positions in the text as either *(a)live* and part of a *live-zone* or *dead* and part of a *dead-zone*. A position or range of positions in the text is dead i.e. in a dead-zone, if (based on information obtained by an algorithm so far) it has been deemed to either not match the pattern/keyword or has been deemed to match the pattern and such a match has been reported or registered. Otherwise, it is live i.e. in a live-zone. Initially, the entire input text is live, and upon termination, the entire input text is dead. So-called dead-zone style algorithms such as those in the algorithm family presented here, can do better than Boyer-Moore style algorithms can: based on a single alignment of text to pattern, shifting (i.e. "killing" positions) can occur both to the right *and* to the left, i.e. yielding *two* remaining live text parts (and hence next alignments) to consider. Furthermore, these two parts can be processed independently, offering ample opportunity for concurrency and fundamentally distinguishes the dead-zone algorithm family from classical Boyer-Moore and most more recent single keyword pattern matching algorithms, which typically use a single sliding alignment window on the text. Dead-zone style algorithms can be used with any of the various Boyer-Moore style shift functions known from the literature, and they can even use left and right shift functions that are completely different (above and beyond the obvious mirroring required).

This algorithm family can be seen in the context of earlier work by the authors and their collaborators, based on the use of live-zones and dead-zones for keyword pattern matching. At the Second Prague Stringology Club Workshop (1997), Watson and Watson presented a paper on a new family of string pattern matching algorithms [19], with a later version appearing in 2003 in the South African Computer Journal [20]. In those papers, the focus is on representing liveness and deadness on a per-position basis. The use of ranges to represent liveness and deadness is only mentioned in passing, leading to a *recursive* range-based algorithm. Furthermore, this early work is not explicit about the use of Boyer-Moore style shift functions in the algorithms. At the International Workshop on Combinatorial Algorithms (IWOCA) 2012, Watson, Kourie and Strauss presented a slightly different recursive dead-zone-based algorithm with explicit use of Boyer-Moore style shift functions, as well as a C++ implementation and benchmarking [18]. This work was further extended by Mauch et al. [13], who present the recursive algorithm with some variations, including iterative ones obtained by (tail) recursion elimination. The focus there is on implementing and benchmarking these variants (in C and C++).

In this paper, we present and derive a family of iterative, range-based dead-zone algorithms, including different representation choices for the live-zones in the algorithms. As we will see, the family includes an alternative derivation of the recursive implicit-stack based algorithm used in [18,13]. Furthermore, various representation choices are considered, and various dead-zone-style algorithms using concurrency are presented and sketched, some of them new. The presentation is in the spirit of the CbC style pioneered by Dijkstra, Gries, and others over four decades ago [6,8,14,12]. For this reason, we assume some awareness of this style and we use the well-known Guarded Command Language (GCL; designed by Dijkstra) which was designed to support reasoning about correctness. The presentation starts out from a single abstract algorithm whose correctness is easy to argue based on the formulation of pre- and post-conditions and invariant. Using the CbC approach, this algorithm is then iteratively refined, leading to the derivation and presentation of the family of algorithms.

2 Problem Statement and Initial Solution Sketch

The single keyword pattern matching problem can be stated as follows:

> Given two strings $x, y \in \Sigma^*$ over an alphabet Σ—respectively called the *keyword* or *pattern*, and *(input or search) text*—find all occurrences of x as a contiguous substring of y.

We register any such occurrences or matches by keeping track of the indices in the text at which they occur, i.e. at which they *start*, in a *match set* variable, called *MS* (a set of integers)[1]. To make this more precise, we define a *helper predicate*

$$Match(x, y, j) \equiv (x = y_{[j,j+|x|)}).$$

Note that here and throughout this derivation, we use $[a, b)$ style ranges (inclusive at the left end, exclusive at the right end) to avoid numerous $+1$ and -1 terms in indexing a string.

Using predicate *Match*, the postcondition to be established by any algorithm solving the single keyword pattern matching problem therefore is:

$$MS = \bigcup_{j \in [0,|y|): Match(x,y,j)} \{j\}$$

Note that indexing in y starts from 0, in keeping with many programming languages as well as typical use in correctness-by-construction styles.

Also note that the above does not place any restrictions on x, y; in particular, y may be shorter than x (in which cases there are no matches), and one or both could be of length 0 (and in case x does, it trivially matches at every position of y). Many algorithm presentations in the literature needlessly give such restrictions as part of the problem statement—thereby cluttering the resulting algorithm(s).

As in the earlier derivations and presentations of dead-zone style algorithms mentioned in the introduction, our algorithms will keep track of all indices in y (i.e. members of the set $[0, |y|)$, categorizing each such index k into one of a number of disjoint sets. Here, we will use three (disjoint) sets, although one of them will not be represented explicitly:

1. *MS*, the set of indices where a match has already been found.
2. *Live_Todo*, the set of indices about which we know nothing (i.e. there may or may not be a match at such an index), i.e. that are still *live*.
3. $\neg(MS \cup Live\_Todo)$, the set of indices at which we *know* no match occurs.

The indices in category 2 are called *live* indices, while those in category 1 and 3 are called *dead*. In earlier dead-zone-style algorithms as presented in [19,20], two disjoint sets were used, representing the live (or to do) and dead (or done) indices respectively, with dead indices at which a match occurs being reported as soon as they are found.

The aim of every algorithm in this paper's algorithm family is to start with $Live\_Todo = [0, |y|)$ and reduce this to $Live\_Todo = \emptyset$, meaning that all indices have been checked and are in category 1 or 3; in other words, each such algorithm starts with the entire input text y being *live*, and ends with the entire input being *dead*. This is expressed by the predicates and algorithm skeleton given below.

[1] Of course, in practical implementations, we may print a message to the screen or otherwise report a match instead of accumulating the variable *MS*.

Algorithm 1 (Abstract Live and Dead Indices Matcher)
Live_Todo :=$[0, |y|)$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge(\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg\boldsymbol{Match}(x, y, j))$ }
{ **variant:** $|\boldsymbol{Live\_Todo}|$ }
S
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

Clearly, given an appropriate statement S satisfying the above pre- and postconditions for S, when $|Live\_Todo| = 0$, $Live\_Todo = \emptyset$ and hence the previously mentioned postcondition holds, i.e. $MS = \bigcup_{j \in [0,|y|): Match(x,y,j)} \{j\}$.

3 From Position-based to Range-based Iterative Dead-zone

We can immediately make a practical improvement to Algorithm 1 by representing *Live_Todo* as a *set of (live) ranges* $[l, h)$ (with l, h integers). At first glance this may seem less efficient, but we will benefit from this shortly. As a further conjunct to the original invariant, we also insist that the ranges in *Live_Todo* are pairwise disjoint, i.e. none of the ranges overlap with each other[2]. Furthermore, the ranges are assumed to be maximal, to keep *Live_Todo* small (although ranges may be empty). With this minor change of representation, we need to be clear about what $|Live\_Todo|$ in the variant means: we still use it to refer to the total number of *indices* represented in *Live_Todo*, not the number of ranges in it. The resulting new algorithm skeleton is given below.

Algorithm 2 (Abstract Live and Dead Ranges Matcher)
Live_Todo := $\{[0, |y|)\}$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\ (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg\boldsymbol{Match}(x, y, j))$ }
{ **variant:** $|\boldsymbol{Live\_Todo}|$ }
do *Live_Todo* $\neq \emptyset \rightarrow$
  Extract some $[l, h)$ *from Live_Todo*;
  S_0
od
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

The question now becomes what needs to be done in S_0 to re-establish the invariant. Clearly some or all of the indices in the range $[l, h)$ need to be checked to gain information about matches found or found to be impossible. Rather than check all indices in the range, we check for a match at the midpoint of the range, i.e. at

[2] Not doing so would lead to inefficiency by considering a live position more than once, or extra booking-keeping to eliminate positions which have already been considered

$m = \lfloor \frac{l+h}{2} \rfloor$ and split the range in two, inserting the remaining portions into *Live_Todo*, i.e. adding $[l, m)$ and $[m+1, h)$ to *Live_Todo*. Note that as indicated above, this does not make variant $|Live\_Todo|$ increase. It should also be noted that other choices than the midpoint of the range could be made, including choices that have the algorithm degenerate to e.g. Boyer-Moore [19], but we will not explore these here.

Either or both of $[l, m)$ and $[m+1, h)$ may be empty ranges. We can detect this before insertion and not insert the empty range(s) into *Live_Todo*. Here, we elect to do such range checking upon extraction of a range from *Live_Todo*, cutting the amount of pseudo-code. However, whenever an empty range is processed in the loop, the earlier variant $|Live\_Todo|$ does not strictly decrease but stays the same. Because of this, we change the variant to the pair $\langle |Live\_Todo|, E \rangle$ with E the number of empty ranges in *Live_Todo*. Using the standard lexicographical ordering on such pairs, this is once more a correct variant: it decreases not only when $|Live\_Todo|$ decreases, but also when $|Live\_Todo|$ stays the same yet an empty range is extracted from *Live_Todo* (since in that case E decreases).

Finally, we note that the last $|x|-1$ indices of input text y cannot contain a match, and we change the initialization of y accordingly. (Note that this could already have been done before, i.e. in Algorithms 1 and 2.)

The above refinements give us Algorithm 3.

Algorithm 3 (Live and Dead Ranges Matcher)

$Live\_Todo := \{[0, |y| - |x|)\}$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $Live\_Todo \neq \emptyset \rightarrow$
    Extract $[l, h)$ *from Live_Todo*;
    if $l \geq h \rightarrow$ { *empty range* } **skip**
    [] $l < h \rightarrow$
        $m := \lfloor \frac{l+h}{2} \rfloor$;
        if $Match(x, y, m) \rightarrow MS := MS \cup \{m\}$
        [] $\neg Match(x, y, m) \rightarrow$ **skip**
        fi;
        $Live\_Todo := Live\_Todo \cup [l, m) \cup [m+1, h)$
    fi
od
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

---

4 Improvements to Range-based Iterative Dead-zone

The preceding algorithm repeatedly performs match attempts, i.e. tests predicate *Match*. Testing this predicate boils down to a loop testing the symbols of x one-for-one against $y_{[m,m+|x|)}$. Such match attempts can be done letter-by-letter from right to left or vice versa, but also in parallel, or using a different order. The particular order used has typically been called the *match order*, with extensive discussions of different orders by Hume & Sunday [11] and elsewhere [16,3].

As with all Boyer-Moore style algorithms, we can eliminate more than one index at a time, i.e. not just index m but consecutive indices next to it as well, by cleverly using information gathered during the match attempt. Shifting more than one index at a time after this match attempt is usually done using a precomputed shift function. We do not consider the details of such "shifters" here. They are extensively covered in the literature [2,9,11], including with correctness arguments in [21,5]. Given that any such shift function is usable with the above algorithm skeleton, the resulting skeleton (not explicitly given here) in fact represents an extensive family of algorithms, even without considering the preceding abstract algorithms or the ones that follow in Section 5.

Since match attempts are made near the middle of a selected range however, our algorithm skeleton, in contrast to classical Boyer-Moore and all its variations, can use two shifters at the same time: one to shift to the right (as per Boyer-Moore and variants), increasing the left bound of range $[m+1, h)$; and a dual one to the left, decreasing the upper bound of range $[l, m)$. Although such left shifters have not been described in detail in the literature, they are in fact straightforwardly computed "duals" of the right shifters, as was pointed out in [18]. It is important to note that it is not even necessary to use a shifter and its dual shifter in the other direction, but that a shifter and the dual of a completely different Boyer-Moore style shifter can be used—thus, the algorithm family is even more extensive than it may seem at first glance. For example, a family member could use Horspool's shifter for its right shifts, and the dual of Sunday's shifter for its left shift, etc.

If we assume two such shift functions, returning values *shl* and *shr* for shift left and shift right, the update of *Live_Todo* near the bottom of Algorithm 3 can be replaced by

$$
\begin{array}{l}
l', h' := m - shl, m + shr; \\
Live\_Todo := Live\_Todo \cup [l, l') \cup [h', h);
\end{array}
$$

We can do a relatively simple running-time analysis, which relies on prior knowledge about the shift functions: they are both bounded above by $|x|$ in most cases[3] meaning we could make as few as $\left\lceil \frac{|y|}{2|x|} \right\rceil$ match attempts in the best case. This contrasts to $\left\lceil \frac{|y|}{|x|} \right\rceil$ match attempts for Boyer-Moore and variants. Note that this does not violate any information-theoretical bounds; in each match attempt, character comparisons from both the left and the right end of the current alignment are made, and following each match attempt, our family of algorithms simply shifts in two directions instead of one. In the best case, just like for the Boyer-Moore algorithm and variants, we have $\left\lceil \frac{|y|}{|x|} \right\rceil$ character comparisons. The worst case for this family, as with all keyword pattern matching variants, is $|y|$ match attempts, and $|y| * |x|$ character comparisons.

[3] The exception is with shift functions as used in e.g. the Berry-Ravindran algorithm [1], which uses characters next to x's current alignment in y as well. In such cases, the shift function is bounded by $|x| + c$ for some constant c.

5 Different Live-zone Representations

Until now, *Live_Todo* has been a set, meaning there is a measure of nondeterminism in the algorithms presented. This can lead to very poor performance in practice, in cases where the algorithm needs read access intermittently all across y.

Alternatively to a set representation, *Live_Todo* can easily be represented using a queue or using a stack. Predictably, these lead to breadth- respectively depth-wise traversals of the ranges in y.

As noted before, the ranges in *Live_Todo* are pairwise disjoint. This also means we can deal with them entirely independently and in parallel. Indeed, each iteration could give rise to new threads, leading to various concurrent versions of the algorithms presented. Two recent papers, [10] and [7], discuss approaches that have some similarities to this; they also process multiple text segments at the same time, but this is limited to two segments in the former, and a fixed number of segments in the latter. In [7], the fixed number of segments is strongly reflected in the structure of the source code, with some coupling between the segments' processing, and no obvious way of making the processing multi-threaded.

In the worst case, a queue can grow to $|y|$ and in the best case to $\left\lceil \frac{|y|}{|x|} \right\rceil$ elements (ranges). Because of this worst case behaviour, we do not further consider this representation choice in the current paper.

A stack representation is much more efficient than a queue one, giving a maximum stack size of $log_2|y|$. The resulting algorithm is a relatively straightforward refinement of (the bidirectional shifting-based refinement) of Algorithm 3, and is given below.

Algorithm 4 (Live and Dead Ranges Matcher using Stack)
$Live\_Todo := \langle [0, |y| - |x|) \rangle$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $Live\_Todo \neq \emptyset \rightarrow$
  Pop $[l, h)$ *from Live_Todo*;
  if $l \geq h \rightarrow$ { *empty range* } **skip**
  [] $l < h \rightarrow$
    $m := \left\lfloor \frac{l+h}{2} \right\rfloor$;
    if $Match(x, y, m) \rightarrow MS := MS \cup \{m\}$
    [] $\neg Match(x, y, m) \rightarrow$ **skip**
    fi;
    $l', h' := m - shl, m + shr$;
    Push $[h', h)$ *to Live_Todo*;
    Push $[l, l')$ *to Live_Todo*
  fi
od
{ **invariant** $\wedge |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

This algorithm in fact is an encoding with an explicit stack of a recursive dead-zone-style algorithm, similar to the ones presented in [18,13].

Algorithm 4 can be further refined to yield a version with information sharing; that is, given that zones may overlap, and since the algorithm considers zones from left to right, we can track a *known dead-zone* $[0, z)$ with z such that $(\forall\, i : 0 \le i < z : i \notin \mathit{Live\_Todo})$, and keep z maximal given what is presently known about the input text based on the algorithm's preceding processing.

Given such a value z, whenever we pop $[l, h)$ from the stack, it can be changed to $[l \max z, h)$, followed by updating z to $l \max z$. These changes are correct due to additional invariants added below, as well as the order in which ranges are pushed onto the stack, which ensures that whenever $[l, h)$ is popped from the stack, all indices to the left of l are already dead.

Algorithm 5 (Live and Dead Ranges Matcher using Stack and Sharing)

$\mathit{Live\_Todo} := \langle [0, |y| - |x|) \rangle$;
$MS := \emptyset$;
$z := 0$;
{ **invariant:** $(\forall\, j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\, (\forall\, j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\, (\forall\, i : 0 \le i < z : i \notin \boldsymbol{Live\_Todo})$ }
{ $\wedge\, (\boldsymbol{top}(\boldsymbol{Live\_Todo}) = [l, h)) \implies [0, l)$ **is dead** }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $\mathit{Live\_Todo} \neq \emptyset \to$
  Pop $[l, h)$ *from Live_Todo*;
  $l := l \max z$;
  $z := l$;
  if $l \ge h \to$ { *empty range* } **skip**
  $[\!]$ $l < h \to$
    $m := \lfloor \frac{l+h}{2} \rfloor$;
    if $\mathit{Match}(x, y, m) \to MS := MS \cup \{m\}$
    $[\!]$ $\neg \mathit{Match}(x, y, m) \to$ **skip**
    fi;
    $l', h' := m - \mathit{shl}, m + \mathit{shr}$;
    Push $[h', h)$ *to Live_Todo*;
    Push $[l, l')$ *to Live_Todo*
  fi
od
{ **invariant** $\wedge\, |\boldsymbol{Live\_Todo}| = 0$ }
{ **post** }

---

6 Using Concurrency for Match Attempts

The use of concurrency as suggested in the preceding section is but one possible use of concurrency in dead-zone-style algorithms. We consider a different, previously undescribed, use of concurrency below. The core observation that leads to this use of concurrency is that for certain shift functions, very little match attempt information is used to determine the shifts to be made (while often still providing competitive performance compared to other shift functions). This means that shifting and match attempt verification can be decoupled to some degree.

For example, Horspool's variant of Boyer-Moore uses shifts based only on the rightmost symbol of the input at the given keyword alignment (and in our context, dually uses the leftmost symbol for the left shifter). Similarly, Sunday's Quicksearch uses shifts solely based on the symbol to the right of that rightmost symbol; and in our context, dually based on the symbol to the left of the leftmost symbol of the alignment for the shift to the left. Thus, it is not necessary to attempt a complete match i.e. to fully evaluate predicate $Match(x, y, m)$ before such shifts can be calculated.

For a given alignment of the pattern to the input text, an alternative algorithm could read the two symbols needed for the respective right shift and left shift (one for the right, one for the left shift), and immediately compute the shifts and make appropriate updates to *Live_Todo*. The current index, at which a match attempt still has to be performed (i.e. for which predicate *Match* has to be evaluated), can be added to a separate set variable, say *Attempt*. The elements of this set can be processed after the main loop of the algorithm has terminated, yielding a different but still sequential dead-zone-style algorithm.

Instead of such a sequential algorithm, a version employing concurrency could be used: one or more separate "matcher" threads could repeatedly extract an element from *Attempt* and perform the match attempt for the alignment indicated by the element. Alternatively, each thread t can have its own thread-private set $Attempt_t$ from which to extract elements, with the main algorithm loop adding positions to the various threads's $Attempt_t$. We present the latter algorithm, including the procedure $Matcher_t$ to be executed by each of the matcher threads $t \in MThreads$. Note that the second conjunct of the invariant has been changed to cover the $Attempt_t$ variables, reflecting the fact that removal of an element from *Live_Todo* does not necessarily mean it represents a non-match, as long as the match attempt for the element still needs to be executed (i.e. the element is in one of the sets $Attempt_t$).

Algorithm 6 (Concurrent Live and Dead Ranges Matcher using Stack)

$Live\_Todo := \langle [0, |y| - |x|) \rangle$;
$MS := \emptyset$;
{ **invariant:** $(\forall\ j : j \in \boldsymbol{MS} : \boldsymbol{Match}(x, y, j))$ }
{ $\wedge\ (\forall\ j : j \notin (\boldsymbol{MS} \cup \boldsymbol{Live\_Todo} \cup \bigcup_{t \in \boldsymbol{MThreads}} \{\boldsymbol{Attempt}_t\}) : \neg \boldsymbol{Match}(x, y, j))$ }
{ **variant:** $\langle |\boldsymbol{Live\_Todo}|, E \rangle$ }
do $Live\_Todo \neq \emptyset \rightarrow$
  Pop $[l, h)$ *from Live_Todo*;
  if $l \geq h \rightarrow$ { *empty range* } **skip**
  [] $l < h \rightarrow$
    $m := \lfloor \frac{l+h}{2} \rfloor$;
    Add m *to queue* $Attempt_t$ *for some thread* t;
    $l', h' := m - shl, m + shr$;
    Push $[h', h)$ *to Live_Todo*;
    Push $[l, l')$ *to Live_Todo*
  fi
od
{ **invariant** $\wedge\ |\boldsymbol{Live\_Todo}| = 0 \wedge (\forall\ t : t \in \boldsymbol{MThreads} : \boldsymbol{Attempt}_t = \emptyset)$ }
{ **post** }

proc $Matcher_t$

```
    do Attempt_t ≠ ∅ →
        if Match(x, y, m) → MS := MS ∪ {m}
        [] ¬Match(x, y, m) → skip
        fi
    od
corp
```

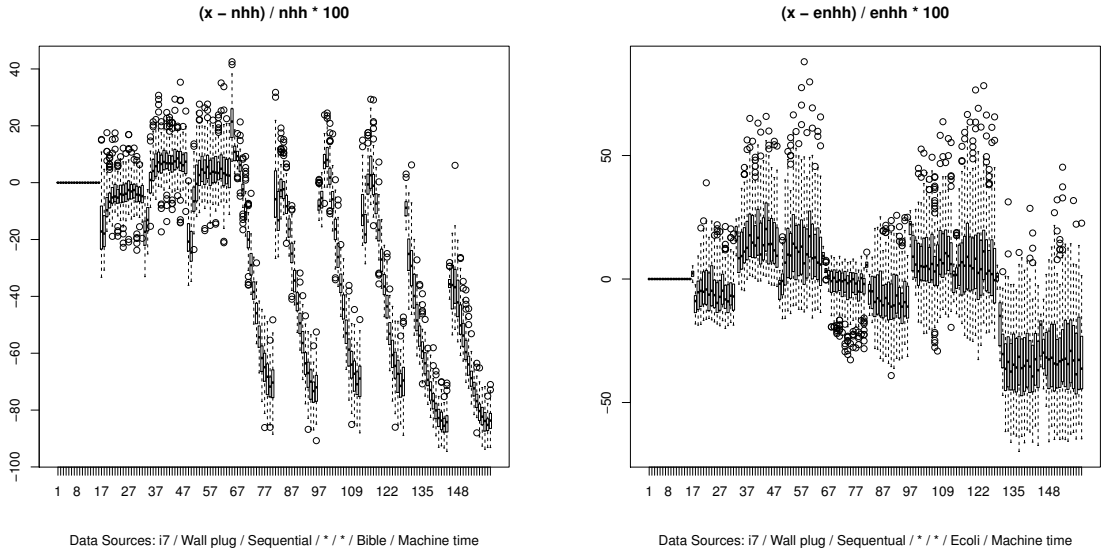
This algorithm assumes each matcher thread has write access to the shared match set variable *MS*. In practice, it is likely that an implementation would instead have a thread-local match set variable, and these would be gathered in shared memory after thread termination.

7 Concluding Remarks

We have given a derivation, in the form of a stepwise presentation, of an extensive family of single keyword pattern matching algorithms. Our exposition serves as a case study for the Correctness-by-Construction style of algorithm development, emphasizing correctness and clarity, which in turn lead to elegant and presumably efficient algorithms. Our presentation included several new algorithm variants, a result typically seen when using the correctness-by-construction style of algorithm derivation [17,4,15].

The algorithms in the family are all based on tracking input text parts as being dead-zones or live-zones, and using shifts to both the left and the right after an alignment and match attempt in the middle of a remaining text part to be considered. This approach offers ample opportunity for concurrency and fundamentally distinguishes the dead-zone algorithm family from classical Boyer-Moore and most more recent single keyword pattern matching algorithms, which typically use a single sliding alignment window on the text.

Previous publications [13,18] and ongoing benchmarking have already shown promising results for some of the members of the algorithm family as well as closely related dead-zone-based algorithms. Figure 1 shows some recent results. The details are not of particular interest here, but Figures 1a and 1b show the differences between each of 10 algorithms—8 dead-zone variants, followed by plain non-dead-zone Horspool and QuickSearch at the right of each figure—when run on Bible and Ecoli data respectively. The box plots reflect results over keyword lengths 2^i for $i \in [1, 16]$ with 100 pseudo-randomly generated keywords of each length—thus, each figure has $10 * 16 = 160$ box plots over 100 values. The results are relative (in percentage terms) to the first (leftmost) algorithm, an iterative dead-zone algorithm using Horspool's shifter (and dual). The results show current dead-zone implementations to already be close to competitive against plain non-dead-zone Horspool and QuickSearch, depending on the data sets pattern length under consideration. Since the new dead-zone algorithms derived and discussed in this paper include ones amenable to using concurrency (multithreading) in various ways, they are all the more interesting to explore further as future work.



(a) On bible data (b) On Ecoli data

Figure 1: Comparisons of relative performance of various dead-zone algorithms and plain Horspool and QuickSearch.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop'99, J. Holub and M. Simánek, eds., Czech Technical University, Prague, Czech Republic, 1999, pp. 16–26, Collaborative Report DC-99-05.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 62–72.
3. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2013, pp. 92–106.
4. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Eindhoven University of Technology, the Netherlands, Apr. 2008.
5. L. Cleophas, B. W. Watson, and G. Zwaan: *A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms.* Science of Computer Programming, 75 2010, pp. 1095–1112.
6. E. W. Dijkstra: *A Discipline of Programming*, Prentice Hall, 1976.
7. S. Faro and T. Lecroq: *A multiple sliding windows approach to speed up string matching algorithms*, in SEA, R. Klasing, ed., vol. 7276 of Lecture Notes in Computer Science, Springer, 2012, pp. 172–183.
8. D. Gries: *The Science of Computer Programming*, Springer-Verlag, second ed., 1980.
9. R. N. Horspool: *Practical fast searching in strings.* Software — Practice & Experience, 10(6) 1980, pp. 501–506.
10. A. Hudaib, R. Al-Khalid, D. Suleiman, M. Itriq, and A. Al-Anani: *A fast pattern matching algorithm with two sliding windows (TSW).* Journal of Computer Science, 4(5) 2008, pp. 393–401.
11. A. Hume and D. Sunday: *Fast string searching.* Software — Practice & Experience, 21(11) 1991, pp. 1221–1248.
12. D. G. Kourie and B. W. Watson: *The Correctness-by-Construction Approach to Programming*, Springer Verlag, 2012.
13. M. Mauch, D. G. Kourie, B. W. Watson, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in SAICSIT Conf., J. H. Kroeze and R. de Villiers, eds., ACM, 2012, pp. 59–68.
14. C. C. Morgan: *Programming from specifications, 2nd Edition*, Prentice Hall International series in computer science, Prentice Hall, 1994.

15. B. Watson, D. Kourie, and L. Cleophas: *Experience with correctness-by-construction.* Science of Computer Programming, 2013, in press.
16. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, Sept. 1995.
17. B. W. Watson: *Algorithms for Constructing Minimal Acyclic Deterministic Finite Automata*, PhD thesis, Department of Computer Science, University of Pretoria, South Africa, 2010.
18. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in IWOCA, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.
19. B. W. Watson and R. E. Watson: *A new family of string pattern matching algorithms*, in Proceedings of the Second Prague Stringologic Workshop, J. Holub, ed., Prague, Czech Republic, July 1997, Czech Technical University, pp. 12–23.
20. B. W. Watson and R. E. Watson: *A new family of string pattern matching algorithms.* South African Computer Journal, 30 June 2003, pp. 34–41.
21. B. W. Watson and G. Zwaan: *A taxonomy of sublinear multiple keyword pattern matching algorithms.* Science of Computer Programming, 27(2) 1996, pp. 85–118.

Random Access to Fibonacci Codes

Shmuel T. Klein[1] and Dana Shapira[2,3]

[1] Computer Science Department, Bar Ilan University, Israel
[2] Computer Science Department, Ashkelon Academic College, Israel
[3] Department of Computer Science and Mathematics, Ariel University, Israel
tomi@cs.biu.ac.il, shapird@gmail.com

Abstract. A Wavelet tree allows direct access to the underlying file, resulting in the fact that the compressed file is not needed any more. We adapt, in this paper, the Wavelet tree to Fibonacci Codes, so that in addition to supporting direct access to the Fibonacci encoded file, we also increase the compression savings when compared to the original Fibonacci compressed file.

1 Introduction and previous work

Variable length codes, such as Huffman and Fibonacci codes, were suggested long ago as alternatives to fixed length codes, since they might improve the compression performance. However, random access to the ith codeword of a file encoded by a variable length code is no longer trivial since the beginning position of the ith element depends on the lengths of all the preceding ones.

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

Another line of investigation applies efficiently implemented *rank* and *select* operations on bit-vectors [23,20] to develop a data structure called a *Wavelet Tree*, suggested by Grossi et al. [11], which allows direct access to any codeword, and in fact recodes the compressed file into an alternative form. The root of the Wavelet Tree holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in the compressed text. The left and right children of the root hold, respectively, the bitmaps obtained by concatenating, again in the given order, the *second* bit of each of the codewords starting with 0, respectively with 1. This process is repeated recursively with the children.

In this paper, we study the properties of Wavelet trees when applied to Fibonacci codes, and show how to improve the compression beyond the savings achieved by Wavelet trees for general prefix free codes. It should be noted that a Wavelet tree for general prefix free codes requires a small amount of additional memory storage as compared to the memory usage of the compressed file itself. However, since it enables efficient direct access, it is a price we are willing to pay. Wavelet trees, which are different implementations of compressed suffix arrays, yield a tradeoff between search time and memory storage. Given a string T of length n and an alphabet Σ, one of the implementations requires space $nH_h + O(\frac{n \log\log n}{\log_{|\Sigma|} n})$ bits, where H_h denotes the hth-order empirical entropy of the text, which is bounded by $\log |\Sigma|$, and processing time just $O(m \log |\Sigma| + \text{polylog}(n))$ for searching any pattern sequence of length m.

Shmuel T. Klein, Dana Shapira: Random Access to Fibonacci Codes, pp. 96–109.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

Grossi and Ottaviano introduce the Wavelet *trie*, which is a compressed indexed sequence of strings in which the shape of the tree is induced from the structure of the Patricia trie [19]. This enables efficient prefix computations (e.g. count the number of strings up to a given index having a given prefix) and supports dynamic changes to the alphabet.

Brisaboa et al. [4] use a variant of a Wavelet tree on Byte-Codes, which encodes the sequence and provides direct access. The root of the Wavelet tree contains the first *byte*, rather than the first bit, of all the codewords, in the same order as they appear in the original text. The root has as many children as the number of different bytes (e.g., 128 for ETDC). The second level nodes store the second byte of those codewords whose first byte corresponds to that child (in the same order as they appear in the text), and so on. The reordering of the compressed text bits becomes an implicit index representation of the text, which is empirically shown to be better than explicit main memory inverted indexes, built on the same collection of words, when using the same amount of space. We use, in this paper, a binary Wavelet tree rather than a 256-ary one for byte-codes, using less space.

In another work, Brisaboa et al. [6] introduced directly accessible codes (DACs) by integrating rank dictionaries into byte aligned codes. Their method is based on Vbyte coding [25], in which the codewords represent integers. The Vbyte code splits the $\lfloor \log x_i \rfloor + 1$ bits needed to represent an integer x_i in its binary form into blocks of b bits and prepends each block with a flag-bit as follows. The highest bit is 0 in the extended block holding the most significant bits of x_i. and 1 in the others. Thus, the 0 bits acts as a comma between codewords. For example, if $b = 3$, and $x_i = 25$, the binary representation of x_i, 11001, is split into two blocks, and after adding the flags to each block, the codeword is **0**011 **1**001. In the worst case, the Vbyte code loses one bit per b bits of x_i plus b bits for an almost empty leading block, which is worse than δ-Elias encoding. DACs can be regarded as a reorganization of the bits of Vbyte, plus extra space for the rank structures, that enables direct access to it. First, all the least significant blocks of all codewords are concatenated, then the second least significant blocks of all codewords having at least two blocks, and so on. Then the rank data structure is applied on the comma bits for attaining $\frac{\log(M)}{b}$ processing time, where M is the maximum integer to be encoded. In the current work, not only do we use the Fibonacci encoding which is better than δ- Elias encoding in terms of memory space, we even eliminate some of the bits of the original Fibonacci encoding, while still allowing direct access with better processing time.

Recently, Külekci [18] suggested the usage of Wavelet trees and the rank and select data structures for *Elias* and *Rice* variable length codes. This method is based on handling separately the unary and binary parts of the codeword in different strings so that random access is supported in constant time by two select queries. As an alternative, the usage of a Wavelet tree over the lengths of the unary section of each Elias or Rice codeword is proposed, while storing their binary section, allowing direct access in time $\log r$, where r is the number of distinct unary lengths in the file.

It should also be noted that better compression can obviously be obtained by the optimal Huffman codes. The application field of the current work is thus restricted to those instances in which static codes are preferred, for various reasons, to Huffman codes. These static codes include, among others, the different Elias codes, dense codes like ETDC and SCDC [5], and Fibonacci codes.

The rest of the paper is organized as follows. Section 2 brings some technical details on rank and select, as well as on Fibonacci codes. Section 3 deals with random

access to Fibonacci encoded files, first suggesting the use of an auxiliary index, then showing how to apply Wavelet trees especially adapted to Fibonacci compressed files. Section 4 further improves the self-indexing data structure by pruning the Wavelet tree, and Section 5 brings experimental results.

2 Preliminaries

We bring here some technical details on the rank and select operations, as well as on Fibonacci codes, which will be useful for the understanding of the ideas below.

2.1 Rank and Select

Given a bit vector B and a bit $b \in \{0,1\}$, $\mathsf{rank}_b(B,i)$ returns the number of occurrences of b up to and including position i; and $\mathsf{select}_b(B,i)$ returns the position of the ith occurrence of b in B. Note that $rank_{1-b}(B,i) = i - rank_b(B,i)$, thus, only one of the two, say, $rank_0(B,i)$ needs to be computed. Jacobson [14] showed that rank, on a bit-vector of length n, can be computed in $O(1)$ time using $n+O(\frac{n \log\log n}{\log n}) = n+o(n)$ bits. His solution is based on storing rank answers every $\log^2 n$ bits of B, using $\log n$ bits per sample, and then storing rank answers relative to the last sample every $\frac{\log n}{2}$ bits (requiring $\log(\log^2 n) = 2\log\log n$ bits per sub-sample, and using a universal table to complete the answer to a rank query within a subtable.

Raman et al. [23] partition the input bitmap B into blocks of length $t = \lceil \frac{\log n}{2} \rceil$. These are assigned to classes: a block with k 1s belongs to class k. Class k contains $\binom{t}{k}$ elements, so $\lceil \log \binom{t}{k} \rceil$ bits are used to refer to an element of it. A block is identified with a pair (k,r), where k is its class ($0 \le k \le t$) using $\lceil \log(t+1) \rceil$ bits, and r is the index of the block within the class using $\lceil \log \binom{t}{k} \rceil$ bits. A global universal table for each class k translates in $O(1)$ time any index r into the corresponding t bits. The sizes of these tables add up to $t2^t$ bits. The sequences of $\lceil \frac{n}{t} \rceil$ class identifiers k is stored in one array, K, and that of the indexes r is stored in another, R. The blocks are grouped into superblocks of length $s = \lfloor \log n \rfloor$. Each superblock stores the rank up to its beginning, and a pointer to R where its indexes start. The size of R is upper bounded by $nH_0(B)$, and the main space overhead comes from K which uses $n\lceil \log t + 1 \rceil$ bits.

To solve $\mathsf{rank}_b(B,i)$, the superblock of i is first computed, and its rank value up to its beginning is obtained. Second, the classes from the start of the superblock are scanned, and their k values are accumulated. The pointer to R is obtained in parallel by attaining the pointer value from the start of the superblock and adding $\lceil \log \binom{t}{k} \rceil$ bits for each class k which is processed. This scanning continues up to the block i belongs to, whose index is extracted from R, and its bits are recovered from the universal table.

The $\mathsf{select}_b(B,i)$ operation can be done by applying binary search in B on the index j so that $\mathsf{rank}_b(B,j) = i$ and $\mathsf{rank}_b(B,j-1) = i-1$. Using the $O(1)$ data structure of Jacobson or that of Raman et al. for rank implies an $O(\log n)$ time solution for select. As for the constant time solution for select, the bitmap B is partitioned into blocks, similar to the solution for the rank operation. For simplicity, let us assume that $b = 1$. The case in which $b = 0$ is dealt with symmetrically. In more details, B is partitioned into blocks of two kinds, each containing exactly $\log^2 n$ 1s. The first kind are the blocks that are long enough to store all their 1-positions within sublinear

space. These positions are stored explicitly using an array, in which the answer is read from the desired entry i. The second kind of blocks are the "short" blocks, of size $O(\log^c n)$, where c is a constant. Recording the 1-positions inside them requires only $O(\log \log n)$ bits by repartitioning these blocks, and storing their relative position. The remaining blocks are short enough to be handled in constant time using a universal table.

González et al. [10] give a practical solution for rank and select data structures. Okanohara and Sadakane [21] introduce four practical rank and select data structures, with different tradeoffs between memory storage and processing time. The difference between the methods is based on the treatment of sparse sets and dense sets. Although their methods do not always guarantee constant time, experimental results show that these data structures support fast query results and their sizes are close to the zero order entropy. Barbay et al. [1] propose a data structure that supports rank in time $O(\log \log |\Sigma|)$ and select in constant time using $nH_0(T) + o(n)(H_0(T) + 1)$ bits.

Navarro and Providel [20] present two data structures for rank and select that improve the space overheads of previous work. One using the bitmap in plain form and the other using the compressed form. In particular, they concentrate on improving the select operation since it is less trivial than rank and requires the computation of select_0 and select_1, unlike the symmetrical nature of rank. The memory storage improvement is achieved by replacing the universal tables of [23]'s implementation by on-the-fly generation of their cells. In addition, they combine the rank and select samplings instead of solving each operation separately, so that each operation uses its own sampling, possibly using also that of the other operation.

2.2 Fibonacci Codes

The Fibonacci code is a universal variable length encoding of the integers based on the Fibonacci sequence rather than on powers of 2. A finite prefix of this infinite sequence can be used as a fixed alternative to Huffman codes, giving obviously less compression, but adding simplicity (there is no need to generate a new code every time), robustness and speed [8,16]. The particular property of the binary Fibonacci encoding is that it contains no adjacent 1's, so that the string 11 can act like a *comma* between codewords. More precisely, the codeword set consists of all the binary strings for which the substring 11 appears exactly once, at the left end of the string.

The connection to the Fibonacci sequence can be seen as follows: just as any integer k has a standard binary representation, that is, can be uniquely represented as a sum of powers of 2, $k = \sum_{i \geq 0} b_i 2^i$, with $b_i \in \{0, 1\}$, there is another possible binary representation based on Fibonacci numbers, $k = \sum_{i \geq 2} f_i F(i)$, with $f_i \in \{0, 1\}$, where it is convenient to define the Fibonacci sequence here by

$$F(0) = 0, F(1) = 1 \qquad \text{and} \qquad F(i) = F(i-1) + F(i-2) \quad \text{for } i \geq 2. \tag{1}$$

This Fibonacci representation will be unique if, when encoding an integer, one repeatedly tries to fit in the largest possible Fibonacci number.

For example, the largest Fibonacci number fitting into 19 is 13, for the remainder 6 one can use the Fibonacci number 5, and the remainder 1 is a Fibonacci number itself. So one would represent 19 as $19 = 13 + 5 + 1$, yielding the binary string 101001. Note that the bit positions correspond to $F(i)$ for increasing values of i from right to left, just as for the standard binary representation, in which $19 = 16 + 2 + 1$ would be represented by 10011. Each such Fibonacci representation has a leading 1,

so by preceding it with an additional 1, one gets a sequence of uniquely decipherable codewords.

Decoding, however, would not be instantaneous, because the set lacks the prefix property. For example, a first attempt to start the parsing of the encoded string 1101111111110 by 110 11 11 11 11 would fail, because the remaining suffix 10 is not the prefix of any codeword. So only after having read 5 codewords in this case (and the example can obviously be extended) would one know that the correct parsing is 1101 11 11 11 110. To overcome this problem, the Fibonacci code defined in [8] simply reverses each of the codewords. The adjacent 1s are then at the right instead of at the left end of each codeword, yielding the prefix code {11, 011, 0011, 1011, 00011, 10011, 01011, 000011, 100011, 010011, 001011, 101011, 0000011, ...}.

Since the set of Fibonacci codewords is fixed in advance, and the codewords are assigned by non-increasing frequency of the elements, but otherwise independently from the exact probabilities, the compression performance of the code depends on how close the given probability distribution is to one for which the Fibonacci codeword lengths would be optimal. The lengths are 2, 3, 4, 4, 5, 5, 5, 6, ..., so the optimal (infinite) probability distribution would be $(\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{64}, \ldots)$. For any finite probability distribution, the compression by a prefix of the Fibonacci code will always be inferior to what can be achieved by a Huffman code. For a typical distribution of English characters, the excess of Fibonacci versus Huffman encoding is about 17% [8], and may be less, around 9%, on much larger alphabets [16]. On the other hand, Fibonacci coding may be significantly better than other static codes such as Elias coding, End-tagged dense codes (ETDC) and (s,c)-dense codes (SCDC) [16].

3 Random Access to Fibonacci Encoded Files

3.1 Using an Auxiliary Index

A trivial solution for gaining random access is to use an additional auxiliary index constructed in the following way:

1. Compress the input file, T, using a Fibonacci Code, resulting in the file $\mathcal{E}(T)$ of size u.
2. Generate a bitmap B of size u so that $B[i] = 1$ if and only if $\mathcal{E}(T)[i]$ is the first bit of a codeword.
3. Construct a rank and select succinct data structure for B.

Recall that $u = |\mathcal{E}(T)|$ is the size of the uncompressed text, and Σ denote the alphabet. In the suggested solution, the space used to accomplish constant time rank and select operations (excluding the encoded file) is

$$u + \mathcal{B}(u, |\Sigma| + u) + o(u) + O(\log\log|\Sigma|),$$

where $\mathcal{B}(x, y) = \lceil \log \binom{y}{x} \rceil$ (the information theoretic lower bound in bits for storing a set of x elements from a universe of size y) using Raman et al.'s implementation [23]. A better approach would be to omit the bitmap B of the first implementation and rather embed the index into the Fibonacci encoded file. This can be accomplished by treating two consecutive 1 bits in $\mathcal{E}(T[i])$ as a single 1-bit in B, and other bits in $\mathcal{E}(T[i])$ as a 0 in B. The memory storage is therefore reduced to $\mathcal{B}(u, |\Sigma| + u) + o(u) + O(\log\log|\Sigma|)$. Even better solutions are presented below.

3.2 Using Wavelet Trees

We adjust the Wavelet tree to Fibonacci codes in the following way. Given is an alphabet Σ and a text $T = t_1 t_2 \cdots t_n$ of size n, where $t_i \in \Sigma$. Let $\mathcal{E}_{fib}(T) = f(t_1) \cdots f(t_n)$ be the encoding of T using the first $|\Sigma|$ codewords of the Fibonacci code. The Wavelet tree is in fact a set of annotations to the nodes of the binary tree corresponding to the given prefix code. These annotations are bitmaps, which together form the encoded text, though the bits are reorganized in a different way to enable the random access. The exact definition of the stored bitmaps has been given above in the introduction.

Recall that the binary tree T_C corresponding to a prefix code C is defined as follows: we imagine that every edge pointing to a left child is labeled 0 and every edge pointing to a right child is labeled 1; each node v is associated with the bitstring obtained by concatenating the labels on the edges on the path from the root to v; finally, T_C is defined as the binary tree for which the set of bitstrings associated with its leaves is the code C. Figure 1 is the tree corresponding to the first 7 elements of the Fibonacci code. Since the bitmaps used by the Wavelet tree algorithms use the tree T_C as underlying structure, we shall refer to this tree as the Wavelet tree, for the ease of discourse.

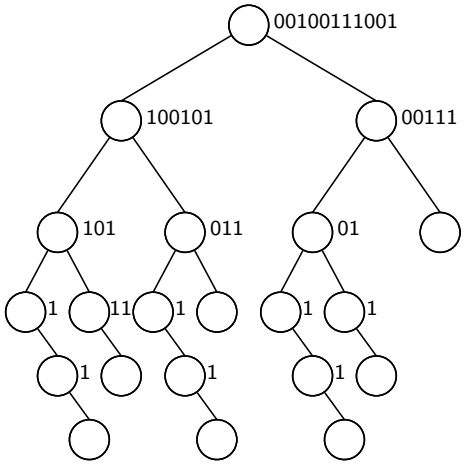


Figure 1. Fibonacci Wavelet Tree for the text $T =$`COMPRESSORS`.

The bitmaps in the nodes of the Wavelet tree can be stored as a single bit stream by concatenating them in order of any predetermined top-down tree traversal, such as depth-first or breadth-first. No delimiters between the individual bitmaps are required, since we can restore the tree topology along with the bitmaps lengths at each node once the size u of the text is given in the header of the file. We shall henceforth refer to the Wavelet tree built for a Fibonacci code as the *Fibonacci Wavelet tree* (FWT). They are related, but not quite identical, to the trees defined by Knuth [17] as Fibonacci trees.

Consider, for example, the text $T =$ `COMPRESSORS` over an alphabet {`C, M, P, E, O, R, S`} of size 7, whose elements appear {1, 1, 1, 1, 2, 2, 3} times, respectively. The Fibonacci encoded file of 39 bits is the following binary string, in which spaces have been added for clarity.

$$\mathcal{E}_{fib}(T) = 01011 \quad 0011 \quad 10011 \quad 00011 \quad 011 \quad 1011 \quad 11 \quad 11 \quad 0011 \quad 011 \quad 11$$

The corresponding FWT, including the annotating bitmaps, is given in Figure 1.

The Wavelet tree for $\mathcal{E}_{fib}(T)$ is a succinct data structure for T as it takes space asymptotically equal to the Fibonacci encoding of T, and it enables accessing any symbol t_i in time $O(|f(t_i)|)$, where $f(x)$ is the Fibonacci encoding of x. The algorithm for extracting t_i from an FWT rooted by v_{root} is given in Figure 2 using the function call $\mathsf{extract}(v_{root},i)$. B_v denotes the bit vector belonging to vertex v of the Wavelet tree, and $\cdot$ denotes concatenation. Computing the new index in the following bit vector is done by the rank operation, given in lines 3.3 and 4.3. As the Fibonacci code is a universal one, the decoding of *code* in line 5 is done by a fixed lookup table.

```
extract(v_root, i)
1   code ⟵ ε
2   while v is not a leaf
3       if B_v[i] = 0
3.1         v ⟵ left(v)
3.2         code ⟵ 0 · code
3.3         i ⟵ rank_0(B_v, i)
4       else
4.1         v ⟵ right(v)
4.2         code ⟵ 1 · code
4.3         i ⟵ rank_1(B_v, i)
5   return D(code)
```

Figure 2. Extracting t_i from a Fibonacci Wavelet Tree rooted at v_{root}.

We extend the definition of $\mathsf{select}_b(B,i)$, which was defined on bitmaps, to be defined on the text T for general alphabets, in a symmetric way. More precisely, we use the notation $\mathsf{select}_x(T,i)$ for returning the position of the ith occurrence of x in T.

Computing $\mathsf{select}_x(T,i)$ is done in the opposite way. We start from the leaf, ℓ, representing the Fibonacci codeword $f(x)$ of x, and work our way up to the root. The formal algorithm is given in Figure 3. The running time for $\mathsf{select}_x(T,i)$ is, again, $O(|f(x)|)$.

```
select_x(T, i)
1   ℓ ⟵ leaf corresponding to f(x)
2   v ⟵ father of ℓ
3   while v ≠ v_root
3.1     if ℓ is a left child of v
3.1.1           i ⟵ index of the ith 0 in B_v
3.2     else // ℓ is a right child of v
3.2.1           i ⟵ index of the ith 1 in B_v
3.3     v ⟵ father of v
4   return i
```

Figure 3. select the ith occurrence of x in T.

4 Enhanced Wavelet Trees for Fibonacci codes

In this section, we suggest to prune the Wavelet Tree, so that the attained pruned Wavelet Tree still achieves efficient rank and select operations, and even improves the processing time. The proposed compressed data structure not only provides efficient random access capability, but also improves the compression performance as compared to the original Wavelet Tree.

4.1 Pruning the tree

The idea is based on the property of the Fibonacci code that all codewords, except the first one 11, terminate with the suffix 011. The binary tree corresponding to the Fibonacci code is therefore not complete, as can be seen, e.g., in the example in Figure 1, and the nodes corresponding to this suffix, at least for the lowest levels of the tree, are redundant. We can therefore eliminate all nodes which are single children of their parent nodes. The bitmaps corresponding to the remaining *internal* nodes of the pruned tree are the only information needed in order to achieve constant random access. A similar idea to this collapsing strategy is applied on suffix or position trees in order to attain an efficient *compacted* suffix trie. They have also been applied on Huffman trees [15] producing a compact tree for efficient use, such as compressed pattern matching [24]. Applying this strategy on the FWT of Figure 1 results in the pruned Fibonacci Wavelet Tree given in Figure 4.

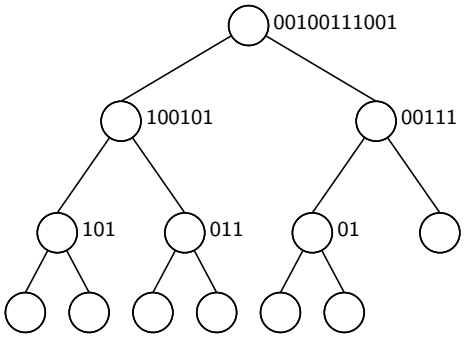


Figure 4. Pruned Fibonacci Wavelet Tree for the text $T =$ COMPRESSORS.

The $\mathsf{select}_x(T, i)$ algorithm for selecting the ith occurrence of x in T is the same as in Figure 3, gaining faster processing time since the lengths of the longer codewords were shortened. However, the algorithm for extracting t_i from a pruned FWT requires minor adjustments for concatenating the pruned parts. The following lines should be added instead of line 5 in the algorithm of Figure 2.

```
5    if suffix of code = 0
5.1      code ⟵ code · 11
6    if suffix of code ≠ 11
6.1      code ⟵ code · 1
7    return D(code)
```

Figure 5. Extracting t_i from the pruned Fibonacci Wavelet Tree.

The FWT of an alphabet of finite size is well defined and fixed. Therefore, only the size of the alphabet is needed for recovering the topological structure of the tree, as opposed to Huffman Wavelet Trees. Recall that the Wavelet tree for general prefix free codes is a reorganization of the bits of the underlying encoded file. The suggested pruned Fibonacci Wavelet tree only uses a partial set of the bits of the encoded file. The main savings of pruned FWTs as compared to the original FWTs of Section 3 stems from the fact that the bitmaps corresponding to the nodes are not all necessary for gaining the ability of direct access. These non-pruned nodes, therefore, are in a one-to-one correspondence with the bits of the encoded Fibonacci file. The bold bits of Figure 6 correspond to those bits that should be encoded; the others can be removed when we use the pruned FWT.

T	C					O					M					P					R			E					S		S			O					R			S		
$\mathcal{E}_{fib}(T)$	**0**	**1**	**0**	1	1	**0**	**0**	**1**	1	**1**	**0**	**0**	1	1	**0**	**0**	**0**	1	1	**0**	**1**	**1**	**1**	**0**	**1**	1	**1**	**1**	**1**	**1**	**0**	**0**	**1**	1	**0**	**1**	**1**	**1**	**1**				

Figure 6. The Bitmap Encoding

4.2 Analysis

We now turn to evaluate the number of nodes in the original and pruned FWTs, from which the compression savings can be derived. Two parameters have to be considered: the number of nodes in the trees, which relate to the storage overhead of applying the Wavelet trees, and the cumulative size of the bitmaps stored in them, which is the size of the compressed file. A certain codeword may appear several times in the compressed file, but will be recorded only once in the WFT.

Since we are interested in asymptotic values, we shall restrict our discussion here to prefixes of the Fibonacci code corresponding to full levels, that is, since the number of codewords of length $h+1$ is a Fibonacci number F_h [16], we assume that if the given tree is of depth $h+1$, then all the F_h codewords of length $h+1$ are in the alphabet. This restricts the size n of the alphabet to belong to the sequence 1, 2, 4, 7, 12, 20, 33, etc., or generally $n \in \{F_h - 1 | h \geq 3\}$. We defer the more involved calculations for general n to the full paper.

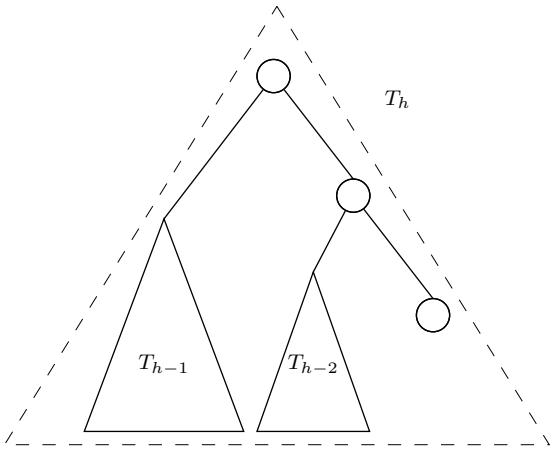


Figure 7. Recursive definition of a Fibonacci Wavelet Tree of depth h.

There are two ways to obtain the FWT of height $h+1$ from that of height h. The first is to consider the defining inductive process, as given in Figure 7. The left subtree of the root is the FWT of height h, while the right subtree of the root consists itself of a root, with a left subtree being the FWT of height $h-1$, and the right subtree being a single node. Denote by N_h the number of nodes in the FWT of height h, we then have

$$N_{h+1} = N_h + N_{h-1} + 3. \tag{2}$$

The second way is by adding the paths corresponding to the F_h longest codewords (of length $h+1$) to the tree for height h. This is done by referring to the nodes on level $h-2$ which have a single child, and there are again exactly F_h such nodes. The single child of these nodes corresponds to the bit 1, and their parent nodes are extended by adding trailing outgoing paths corresponding to the terminating string 011, turning each of them into a node with two children. For example, the grey nodes in Figure 8 are the FWT of height $h = 4$. The three darker nodes are those on level 2 which are internal nodes with only one child. In the passage to the FWT of height $h + 1 = 5$, the bold edges and nodes (representing the suffix 011) are appended to these nodes. This yields the recursion

$$N_{h+1} = N_h + 3F_h. \tag{3}$$

Applying eq. (3) repeatedly gives

$$N_{h+1} = N_{h-1} + 3(F_{h-1} + F_h) = N_{h-2} + 3(F_{h-2} + F_{h-1} + F_h),$$

and in general after k stages we get that

$$N_{h+1} = N_{h-k} + 3(\sum_{i=h-k}^{h} F_i).$$

When substituting $h-k$ by 2 we get that

$$N_{h+1} = N_2 + 3(\sum_{i=2}^{h} F_i).$$

By induction it is easy to show that

$$\sum_{i=2}^{h} F_i = F_{h+2} - 2.$$

Since $N_2 = 3$, we get that

$$N_{h+1} = 3 + 3(F_{h+2} - 2) = 3F_{h+2} - 3. \tag{4}$$

This is also consistent with our first derivation, since the basis of the induction is obviously the same, and assuming the truth of eq. (2) for values up to h, we get by inserting eq. (4) for N_h and N_{h-1} that

$$N_{h+1} = (3F_{h+1} - 3) + (3F_h - 3) + 3 = 3F_{h+2} - 3.$$

The pruned FWT corresponding to the FWT of height $h+1$ is of height $h-1$ and obtained by pruning all single child nodes of the FWT: for each of the F_h leaves of the lowest level $h+1$, two nodes are saved, and for each of the F_{h-1} leaves on level h,

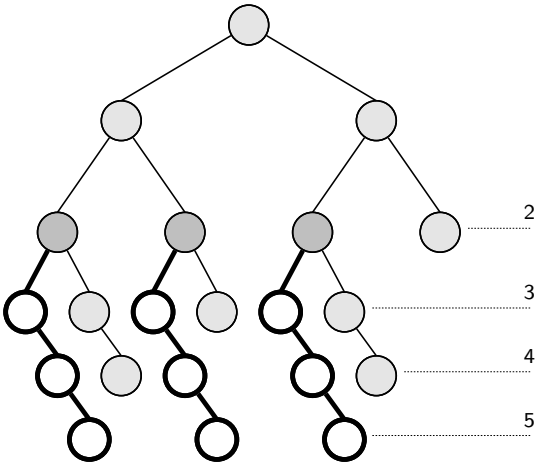


Figure 8. Extending a Fibonacci Wavelet Tree

only a single node is erased. Denoting by S_h the number of nodes in a pruned FWT of height h, we get

$$S_{h-1} = N_{h+1} - 2F_h - F_{h-1}. \tag{5}$$

But

$$2F_h + F_{h-1} = F_{h+1} + F_h = F_{h+2},$$

so substituting the value for N_{h+1} from eq. (4), we get

$$S_{h-1} = 3F_{h+2} - 3 - F_{h+2} = 2F_{h+2} - 3.$$

The ratio of the sizes of the pruned to the original FWTs is therefore

$$\frac{S_{h-1}}{N_{h+1}} = \frac{2F_{h+2} - 3}{3F_{h+2} - 3} \quad \underset{h\to\infty}{\longrightarrow} \quad \frac{2}{3},$$

when the size of the tree grows to infinity, so that about one third of the nodes will be saved. Figure 9 plots the number of nodes in both original and pruned FWTs as a function of the tree's heights.

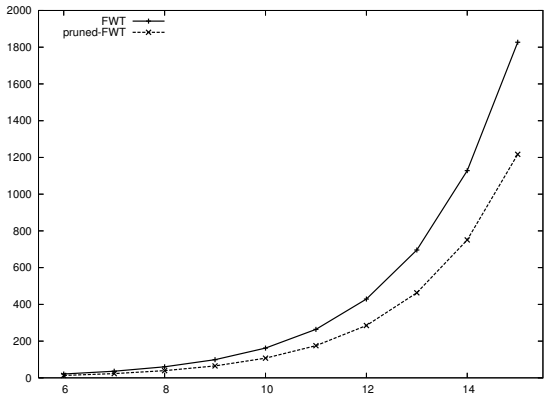


Figure 9. Number of nodes in original and pruned FWT as function of height.

5 Experimental Results

While the number of nodes saved in the pruning process could be analytically derived in the previous section, the number of bits to be saved in the compressed file will depend on the distribution of the different encoded elements. It might be hard to define a "typical" distribution of probabilities, so we decided to calculate the savings for the distribution of characters in several real-life languages.

File	n	height	FWT	pruned	Huffman
English	26	8	4.90	4.43	4.19
Finnish	29	8	4.76	4.44	4.04
French	26	8	4.53	4.14	4.00
German	30	8	4.70	4.37	4.15
Hebrew	30	8	4.82	4.42	4.29
Italian	26	8	4.70	4.32	4.00
Portuguese	26	8	4.67	4.28	4.01
Spanish	26	8	4.71	4.30	4.05
Russian	32	8	5.13	4.76	4.47
English-2	378	14	8.78	8.56	7.44
Hebrew-2	743	15	9.13	8.97	8.04

Table 1. Compression Performance

The distribution of the 26 letters and the 371 letter pairs of English was taken from Heaps [12]; the distribution of the 29 letters of Finnish is from Pesonen [22]; the distribution for French (26 letters) has been computed from the database of the *Trésor de la Langue Française* (TLF) of about 112 million words (for details on TLF, see [3]); for German, the distribution of 30 letters (including blank and *Umlaute*) is given in Bauer & Goos [2]; for Hebrew (30 letters including two kinds of apostrophes and blank, and 735 bigrams), the distribution has been computed using the database of The Responsa Retrieval Project (RRP) [7] of about 40 million Hebrew and Aramaic words; the distribution for Italian, Portuguese and Spanish (26 letters each) can be found in Gaines [9], and for Russian (32 letters) in Herdan [13].

Note that the input of our tests consists of published probability distributions, not of actual texts. There are therefore no available texts that could be compressed. We can only calculate the average codeword lengths, from which the expected size of the compressed form of some typical natural language text can be extrapolated. To still get some idea on the compression performance, we add as comparison the average codeword length of an optimal Huffman code.

The results are summarized in Table 1, the two last lines corresponding to the bigrams. The second column shows the size n of the alphabet. The column entitled height is the height of the original FWT tree for the given distribution, FWT shows the average codeword length for the original FWT, and pruned the corresponding value for the pruned FWT. As can be seen, there is a 7–10% gain for the smaller alphabets, and 2–3% for the larger ones. The reduced savings can be explained by the fact that though a third of the nodes has been eliminated, they correspond to the leaves with lowest probabilities, so the expected savings are lower. The last column, entitled Huffman, gived the average codeword length of an optimal Huffman code. We see that the increase, relative to the Huffman encoded files, of the size of the

FWT compressed files can roughly be reduced to half by the pruning technique. For example, for English, the FWT compressed file is 17 % than the Huffman compressed one, but the pruned FWT reduces this excess to 6 %. All the numbers have been calculated for the given sizes n of the alphabets, and not been approximated by trees with full levels.

References

1. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich: *Alphabet partitioning for compressed rank/select and applications.* Algorithms and Computation, Lecture Notes in Computer Science, 6507 2010, pp. 315–326.
2. F. Bauer and G. Goos: *Informatik, Eine einführende Übersicht, Erster Teil*, Springer Verlag, Berlin, 1973.
3. A. Bookstein, S. T. Klein, and D. A. Ziff: *A systematic approach to compressing a full text retrieval system.* Information Processing & Management, 28 1992, pp. 795–806.
4. N. R. Brisaboa, A. Fariña, G. Ladra, and G. Navarro: *Reorganizing compressed text*, in Proc. of the 31th Annual Internetional ACM SIGIR Conference on Research and Developing in Information Retrieval (SIGIR), 2008, pp. 139–146.
5. 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, LNCS, vol. 2857, Springer Verlag, 2003, pp. 122–136.
6. N. R. Brisaboa, S. Ladra, and G. Navarro: *DACs: Bringing direct access to variable length codes.* Information Processing and Management, 49(1) 2013, pp. 392–404.
7. A. S. Fraenkel: *All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, expanded summary.* Jurimetrics J., 16 1976, pp. 149–156.
8. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
9. H. F. Gaines: *Cryptanalysis, a study of ciphers and their solution.* Dover Publ. Inc. New York, 1956.
10. R. González, S. Grabowski, V. Mäkinen, and G. Navarro: *Practical implementation of rank and select queries*, in Poster Proceedings of 4th Workshop on Efficient and Experimental Algorithms (WEA05), Greece (2005), 2005, pp. 27–38.
11. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.
12. H. S. Heaps: *Information Retrieval Computational and Theoretical Aspects*, Academic Press, New York, 1978.
13. G. Herdan: *The Advanced Theory of Language as Choice and Chance*, Springer-Verlag, New York, 1966.
14. G. Jacobson: *Space efficient static trees and graphs*, in Proceedings of FOCS, 1989, pp. 549–554.
15. S. T. Klein: *Skeleton trees for the efficient decoding of Huffman encoded texts.* in the Special issue on Compression and Efficiency in Information Retrieval of the Kluwer Journal of Information Retrieval, 3 2000, pp. 7–23.
16. S. T. Klein and M. Kopel Ben-Nissan: *On the usefulness of Fibonacci compression codes.* The Computer Journal, 53 2010, pp. 701–716.
17. D. Knuth: *The Art of Computer Programming, Sorting and Searching*, vol. III, Addison-Wesley, Reading, MA, 1973.
18. M. Külekci: *Enhanced variable-length codes: Improved compression with efficient random access*, in Proc. Data Compression Conference DCC–2014, Snowbird, Utah, 2014, pp. 362–371.
19. D. Morrison: *Patricia - practical algorithm to retrieve information coded in alphanumeric.* Journal of the ACM, 15(4) 1968, pp. 514–534.
20. G. Navarro and E. Providel: *Fast, small, simple rank/select on bitmaps.* Experimental Algorithms, LNCS, 7276 2012, pp. 295–306.

21. D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proc. ALENEX, SIAM, 2007.
22. J. Pesonen: *Word inflexions and their letter and syllable structure in finnish newspaper text.* Research Rep. 6, Dept. of Special Education, University of Jyräskylä, Finland (in Finnish, with English summary), 1971.
23. R. Raman, V. Raman, and S. Rao Satti: *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets.* Transactions on Algorithms (TALG), 2007, pp. 233–242.
24. D. Shapira and A. Daptardar: *Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts.* Information Processing and Management, IP & M, 42(2) 2006, pp. 429–439.
25. H. Williams and J. Zobel: *Compressing integers for fast file access.* The Computer Journal, 42(30) 1999, pp. 192–201.

Speeding up Compressed Matching with SBNDM2

Kerttu Pollari-Malmi, Jussi Rautio, and Jorma Tarhio*

Department of Computer Science and Engineering
Aalto University, P.O. Box 15400, FI-00076 Aalto, Finland
{kerttu.pollari-malmi, jorma.tarhio}@aalto.fi, kipuna@gmail.com

Abstract. We consider a compression scheme for natural-language texts and genetic data. The method encodes characters with variable-length codewords of k-bit base symbols. We present a new search algorithm, based on the SBNDM2 algorithm, for this encoding. The results of practical experiments show that the method supersedes the previous comparable methods in search speed.

Keywords: compressed matching

1 Introduction

With the amount of information available constantly growing, fast information retrieval is a key concept in many on-line applications. The *string matching problem* is defined as follows: given a pattern $P = p_1 \cdots p_m$ and a text $T = t_1 \cdots t_n$ in an alphabet Σ, find all the occurrences of P in T. Various good solutions [7] have been presented for this problem. The most efficient solutions in practice are based on the Boyer-Moore algorithm [3] or on the BNDM algorithm [21].

The *compressed matching problem* [1] has gained much attention. In this problem, string matching is done in a compressed text without decompressing it. Researchers have proposed several methods [2,19,20,22] based on Huffman coding [15] or the Ziv-Lempel family [28,29]. Alternatively, *indexing methods* [11,26] can be used to speed up string matching. However, in this paper we concentrate on traditional compressed matching.

One presented idea is to encode whole words by using end-tagged dense code or (s, c)-dense code [4]. In both the approaches, codewords consist of one or more bytes. In end-tagged dense code, the first bit of each byte specifies whether the byte is the last byte of the codeword or not. Thus, there are 128 possible byte values which end the codeword, called *stoppers*, and 128 possible values for *continuers*, which are not the last byte of the codeword. In (s, c)-dense code, the proportion between stoppers and continuers can be chosen more freely to minimize the size of the compressed text. The bytes whose value is less than c are continuers whereas the bytes whose value is at least c but less than $s + c$ (the number of possible byte values) are stoppers. String matching in the compressed text is performed by compressing the pattern and searching for it by using the Boyer-Moore-Horspool algorithm [14]. Brisaboa et al. [5,6] also present dynamic versions of end-tagged dense code and (s, c)-dense code allowing fast searching. All these methods use word-based encoding.

Culpepper and Moffat [8,9] present another word-based compression approach where the value of the first byte of a codeword always specifies the length of the

* Work supported by the Academy of Finland (grant 134287).

Kerttu Pollari-Malmi, Jussi Rautio, Jorma Tarhio: Speeding up Compressed Matching with SBNDM2, pp. 110–123.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

codeword. The rest of bytes in the codeword can obtain any of the possible values. The compressed search can be performed in several different ways, for example using a Knuth-Morris-Pratt [17] or Boyer-Moore-Horspool type algorithm. However, the search algorithm requires that a shift never places the byte-level alignment between codeword boundaries.

The word-based compression methods are poor in the case of highly inflectional languages like Finnish. In this paper, we will concentrate on character-based compression. Shibata et al. [25] present a method called BM-BPE which finds text patterns faster in a compressed text than Agrep [27] finds the same patterns in an uncompressed text. Their search method is based on the Boyer-Moore algorithm and they employ a restricted version of byte pair encoding (BPE) [13], which replaces common pairs of characters with unused characters.

Maruyama et al. [18] present a compression method related to BPE using recursive pairing. However, in their method they select a digram A_iB_i (A_i and B_i are either symbols from alphabet Σ or variables presented earlier in the compression process) that occurs most frequently after $a_i \in \Sigma$ and replace the string $a_iA_iB_i$ with the string a_iX where X is a new variable. This is performed for every $a_i \in \Sigma$. Because the symbol preceding a digram is taken into account, the variable X can be used to replace different digrams after different symbols. They also present an efficient pattern matching algorithm for compressed text. The search algorithm is Knuth-Morris-Pratt [17]. The automaton is modified so that it memorizes the symbol read previously and its *Jump* and *Output* functions are also defined for variables occurring in the compressed text.

We present a different character-based method which is faster than BM-BPE. In our method, characters are encoded as variable-length *codewords*, which consist of *base symbols* containing a fixed number of bits. Our encoding approach is a generalization of that of de Moura et al. [19], where bytes are used as base symbols for coding words. We present several variants of our encoding scheme and give two methods for string matching in the compressed text.

Earlier we have presented a search algorithm [23,24] based on Tuned Boyer-Moore [16], which is a variation of the Boyer-Moore-Horspool algorithm [14]. The shift function is based on several base symbols in order to enable longer jumps than the ordinary occurrence heuristic. In this paper, we present a new search algorithm based on SBNDM2 [10] which is a variation of BNDM, the Backward Nondeterministic DAWG Matching algorithm [21]. SBNDM2 is a fast bit-parallel algorithm, which recognizes factors of the pattern by simulating a nondeterministic automaton of the reversed pattern. Fredriksson and Nikitin [12] have earlier used BNDM to search for patterns in the text that has been compressed by their own algorithm. However, in their tests BNDM on compressed text was slower than the Boyer-Moore-Horspool algorithm on uncompressed text, while in our experiments our SBNDM based search algorithm was faster.

We present test results of four variations of our algorithms together with two reference algorithms for compressed texts as well as two other reference algorithms for uncompressed texts. We are interested only in search speed. The tests were run on DNA and on English and Finnish texts. Our SBNDM2 based search method was the fastest among all the tested algorithms for English patterns of at least 9 characters.

The paper has been organized as follows. Section 2 provides an enhanced presentation of our coding scheme [24]. Our two search algorithms are described in Section 3.

Then Section 4 reports the results of our practical experiments before the conclusions in Section 5.

2 Stopper encoding

2.1 Stoppers and continuers

We apply a semi-static encoding scheme called *stopper encoding* for characters, where the codes are based on frequencies of characters in the text to be compressed. The frequencies of characters are gathered in the first pass of the text before the encoding in the second pass. Alternatively, fixed frequencies based on the language and the type of the text may be used.

A codeword is a variable-length sequence of *base symbols* which are represented as k bits, where k is a parameter of our scheme. The different variants of our scheme are denoted as SE_k, where SE stands for stopper encoding.

Because the length of a code varies, we need a mechanism to recognize where a new one starts. A simple solution is to reserve some of the base symbols as *stoppers* which can only be used as the last base symbol of a code. All other base symbols are *continuers* which can be used anywhere but at the end of a code. If $u_1 \cdots u_j$ is a code, then $u_1, \ldots, u_{j-1}$ are continuers and u_j is a stopper.

De Moura et al. [19] use a scheme related to our approach. They apply 8-bit base symbols to encode words where one bit is used to describe whether the base symbol is a stopper or a continuer. Thus they have 128 stoppers and 128 continuers. Brisaboa et al. [4] use the optimal number of stoppers.

2.2 Number of stoppers

It is an optimization problem to choose the number of stoppers to achieve the best compression ratio (the size of the compressed file divided by that of the original file). The optimal number of stoppers depends on the number of different characters and the frequencies of the characters. If there are s stoppers of k bits, there are $c = 2^k - s$ continuers of k bits. A codeword of l base symbols consists of $l - 1$ continuers and one stopper. Thus, there are sc^{l-1} valid codewords of length l.

Let $\Sigma = \{a_1, a_2, \ldots, a_q\}$ be the alphabet, and let $f(a_i)$ be the frequency of a_i. For simplicity, we assume that the alphabet is ordered according to the frequency, i.e. $f(a_i) \geq f(a_j)$ for $i < j$.

Now $a_1, \ldots, a_s$ are encoded with one base symbol, $a_{s+1}, \ldots, a_{s+cs}$ are encoded with two base symbols, $a_{s+cs+1}, \ldots, a_{s+c^2s}$ are encoded with three base symbols, and so on. So the number of characters that can be encoded with l or less base symbols is

$$\sum_{j=1}^{l} c^{j-1}s = \frac{s(c^l - 1)}{c - 1}.$$

From this we can calculate the number of k-bit base symbols $l(a_i, k, s)$ needed to encode the character a_i in the case of s stoppers. It must be possible to encode at least i first characters by using $l(a_i, k, s)$ or less base symbols and thus

$$i \leq \frac{s(c^{l(a_i,k,s)} - 1)}{c - 1}.$$

By solving the inequality, we obtain

$$l(a_i, k, s) \geq \log_c(i(c-1)/s + 1).$$

Because $l(a_i, k, s)$ must be an integer and we want to choose the smallest possible integer,

$$l(a_i, k, s) = \lceil \log_c(i(c-1)/s + 1) \rceil.$$

Then the compression ratio C is

$$C = \sum_{i=1}^{q} \frac{f(a_i) l(a_i, k, s) k}{8}. \quad (1)$$

Let us consider 4-bit base symbols as an example. Let us assume that all the characters are equally frequent. Table 1 shows the optimal numbers of stoppers for different sizes of the alphabet.

# of characters	*Stoppers*	*Base symb./char at end*
1–31	15	1.55
31–43	14	1.70
43–53	13	1.77
53–61	12	1.82
61–67	11	1.85
67–71	10	1.87
71–73	9	1.89
73–587	8	2.87

Table 1. Optimal stopper selection.

To find the optimal number of stoppers, the frequencies of characters need to be calculated first. After that, formula (1) can be used for all reasonable numbers of stoppers ($8 - 16$ for 4-bit base symbols), and the number producing the lowest compression ratio can be picked. Another possibility is presented by Brisaboa et al. in [4], where binary search is used to find a minimum compression ratio calculated by using formula (1). They consider the curve which presents the compression ratio as a function of the number of stoppers. At each step, it is checked whether the current point is in the decreasing or increasing part of the curve and the search moves towards the decreasing direction. Using this strategy demands that there exists a unique minimum of the curve, but in practice natural language texts usually have that property.

2.3 Building the encoding table

After the number of stoppers (and with it, the compression ratio) has been decided, an encoding table can be created. The average search time is smaller if the distribution of base symbols is as uniform as possible. We use a heuristic algorithm, which produces comparable results with an optimal solution. With this algorithm, the encoding can be decided directly from the order of the frequencies of characters.

Base symbols of two and four bits are the easiest cases. In this encoding with 2^k different base symbols and s stoppers, the base symbols $0, 1, \ldots, s-1$ will act as stoppers and the base symbols $s, \ldots, 2^k - 1$ as continuers. At first, the s most common characters are assigned their own stopper base symbol, in the inverse order

of frequency (the most common character receiving the base symbol $s-1$ and so on). After that, starting from the $s+1$:th common character, each character is assigned a two-symbol codeword in numerical order (first the ones with the smallest possible continuer, in the order starting from the smallest stopper). Exactly the same ordering is used with codewords of three or more base symbols. We call this technique *folding*, since the order of the most frequent characters is reversed until a certain folding point. The aim of folding is to equalize the distribution of base symbols in order to speed up searching. The resulting encoding of the characters in the KJV Bible is presented in Table 2.

ch	code	ch	code	ch	code	ch	code
␣	d	w	e1	D	e9	N	fe1
e	c	y	f1	T	f9	P	ff1
t	b	c	e2	R	ea	C	ee2
h	a	g	f2	G	fa	x	ef2
a	9	b	e3	J	eb	q	fe2
o	8	p	f3	S	fb	Z	ff2
n	7	↩	e4	B	ec	Y	ee3
s	6	v	f4	?	fc	K	ef3
i	5	.	e5	H	ed	!	fe3
r	4	k	f5	M	fd	U	ff3
d	3	A	e6	E	ee0	(	ee4
l	2	I	f6	j	ef0	)	ef4
u	1	:	e7	W	fe0	V	fe4
f	0	;	f7	F	ff0	-	ff4
m	e0	L	e8	’	ee1	Q	ee5
,	f0	O	f8	z	ef1		

Table 2. Encoding table for the King James Bible, 4-bit version, 14 stoppers. The characters have been sorted by the order of their relative frequencies.

bsym	freq	bsym	freq	bsym	freq	bsym	freq
0	4.85%	4	4.58%	8	5.11%	c	8.48%
1	4.31%	5	4.62%	9	5.56%	d	16.21%
2	4.62%	6	4.38%	a	5.96%	e	8.03%
3	4.80%	7	4.99%	b	6.51%	f	7.01

Table 3. Relative frequencies of base symbols in the encoded text (average: 6.25%).

Table 3 shows the relative frequencies of the base symbols in our example. The frequencies of the first stoppers depend on the frequencies of the most common characters. E.g., the frequency of `d` depends on the frequency of the space symbol.

2.4 Variant: code splitting

Consider a normal, uncompressed text file, composed of natural-language text. In the file, most of the information (*entropy*) in every character is located in the 5–6 lowest bits. A search algorithm could exploit this by having the fast loop ignore the highest bits, and take the lower bits of several characters at once. The idea is not usable as such, because the time required for `shift` and `and` operations is too much for a

fast loop to perform efficiently. However, it is the basis for a technique called *code splitting*.

Applying code splitting to SE_6 means that each 6-bit base symbol is split into two parts: the 4-bit *low part* and the 2-bit *high part*, which are stored separately. The search algorithm first works with low parts only, trying to find a match in them. Only after such a match is found, the high parts are checked. Depending on the alphabet, it may also be applicable to split the codes into 4-bit high part and 2-bit low part and to start the search with the high part. We denote a code-split variant as $SE_{6,4}$ or $SE_{6,2}$, where the smaller number is the number of low bits.

Code splitting can also be applied to SE_8. $SE_{8,4}$ is a special case. The characters of the original text are split as such into two 4-bit arrays. This speeds up search but does not involve compression. Code splitting is useful in evenly-distributed alphabets where no compression could be gained, or as an end-coding method for some compression method, including de Moura [19].

3 String matching

We start with a description of our old searching algorithm [24]. This will help to understand the details of the new algorithm, which is more complicated.

3.1 Boyer-Moore based algorithm

Let us consider a text with less than 16 different characters. With the SE_4 schema, any character of this text can be represented with a codeword consisting of a single four-bit base symbol. For effective use, two consecutive base symbols are stored into each byte of the encoded text, producing an exact compression ratio of 50%.

Instead of searching directly the 4-bit array, which would require expensive `shift` and `and` operations, we use the 8-bit bytes. There are two 8-bit *alignments* of each 4-bit pattern: one starting from the beginning of a 8-bit character and the other one from the middle of it, as presented in Table 4. For example, consider the encoded pattern (in hexadecimal) `618e05`. Occurrences of both `61-8e-05` and `*6-18-e0-5*`[1] clearly need to be reported as matches.

The final algorithm works exactly in the same way: by finding an occurrence of the encoded pattern consisting of longer codewords in the text. One additional constraint exists: the base symbol directly preceding the presumed match must be a stopper symbol. Otherwise, the meaning of the first base symbol is altered. It is not the beginning of a codeword, but it belongs to another codeword which begins before the presumed match. Thus, the match is not complete.

It is also noteworthy that because positions in the original text are not mapped one-to-one into positions in the encoded text, an occurrence in the encoded text cannot be converted to a position in the original text. However, we get the *context* of the encoded pattern, and with fast on-line decoding, it can be expanded.

The search algorithm, based on Tuned Boyer-Moore (TBM) [16], is called *Boyer-Moore for Stopper Encoding*, or BM-SE. It is a direct extension of TBM allowing multiple patterns and classes of characters. The algorithm contains a fast loop designed to quickly skip over most of the candidate positions, and a slow loop to verify possible matches produced by the fast loop.

[1] The asterisk `*` denotes a wild card, i.e. any base symbol.

...	6	1	8	e	0	5 ...
...	...	6	1	8	e	0 5

Table 4. An example of BM-SE, pattern `618e05`.

The last byte of both alignments not containing wild cards, the *pointer byte*, has a skip length of 0. The previous bytes have their skip lengths relative to their distances from the pointer byte. The skip lengths for the example pattern `618e05` are presented in Table 5. The fast loop operates by reading a character from text, and obtaining its *skip length*. If the skip length is 0, the position is examined with a slow loop, otherwise the algorithm moves forward a number of positions equal to the skip length.

A half-byte (one containing a wild-card) at the end of the pattern is ignored for the fast loop, whereas one in the beginning sets all 16 variations to have the desired jump length. This works according to the bad-character, or occurrence, heuristic of Boyer and Moore [3].

encoded ch	skip
05, e0	0
8e, 18	1
61, *6	2
**	3

Table 5. Skip lengths. The symbol * is a wild card.

String matching with 2-bit base symbols works basically in the same way as with 4-bit ones. There are 4 parallel alignments instead of 2, and the number of characters enabled by wild-cards can vary from 4 to 64. Compression and string matching with 2-bit base symbols and its various extension possibilities are presented in [23].

3.2 SBNDM2 matching

Our new search solution, called SBNDM2-SE, uses the SBNDM2 algorithm [10] for string matching in SE-coded texts. SBNDM2 is a simplified version of BNDM [21]. The idea of SBNDM2 is to maintain a state vector D, which has a one in each position where a factor of the pattern starts such that the factor is a suffix of the processed text string. For example, if the processed text string is "abc" and the pattern is "pabcabdabc", the value of D is 0100000100. To make maintaining D possible, table B which associates each character a with a bit mask expressing its locations in the pattern is precomputed before searching.

When a new alignment is checked, at first two characters of the text are read before testing the state vector D. We use a precomputed table F such that for each pair of characters, $F[c_i c_j] = B[c_i] \& (B[c_j] \ll 1)$.

At each alignment i we check whether $D = F[t_i, t_{i+1}]$ is zero. If it is, it means that the characters t_i and t_{i+1} do not appear successively in the pattern and we can fast proceed to the next alignment $m - 1$ positions forward. If D is not zero, we continue by calculating $D \leftarrow B[t_{i-1}] \& (D \ll 1)$ and decrementing the value of i by 1 until D becomes zero. This happens when either the pattern has been found or the processed substring does not belong to the pattern. The former case can been distinguished from the latter by the number of characters processed before D became zero.

When we apply the SBNDM2 algorithm to an SE-encoded text, we start by encoding the pattern with the same algorithm as the text was encoded. If SE_4 is used, the length of the base symbol is 4 bits. This means that the pattern may start either from the beginning of a 8-bit character or from the middle of it. To find the occurrence in both cases, we search for two patterns simultaneously: one which starts from the beginning of the encoded pattern and the other which starts from the second half-byte of the encoded pattern. In SBNDM2, it is possible to search for two equal length patterns by searching for the pattern which has been constructed by concatenating the two patterns. For example, if the encoded pattern (in hexadecimal) is `618e0`, we search for both pattern `61-8e` and pattern `18-e0` simultaneously by searching the pattern `61-8e-18-e0`. An extra bit vector (denoted by f in the pseudocode of the algorithm) is used to prevent from finding matches which consist of the end of the first half and the beginning of the second half like `8e-18` in the example case. If SBNDM2 finds the first or the second half of the longer pattern, it is checked whether this is the real occurrence of the pattern by matching the bytes of the encoded pattern and the bytes of the text one by one. It is also checked that the first or last half-byte is found in the text.

If the length of the encoded pattern is even, for example in the case of the pattern `618e05`, the long pattern is constructed by concatenating strings where the first is formed by dropping the last byte of the encoded string and the latter by dropping the first and last half-byte of the encoded string. In the case of the example string, the pattern `61-8e-18-e0` would be searched for by SBNDM2. Again, if the first or the second half of the longer pattern is found, the match is checked by comparing the bytes of the text and the pattern one by one. The pseudocode is shown as Algorithm 1.

If $SE_{8,4}$ encoding is used, SBNDM2 algorithm is applied only to the file containing the low part (the 4 lowest bits) of each encoded byte. The search is performed similarly to the search in the case of SE_4. If an occurrence is found, the high bits are checked one by one.

4 Experiments

We tested the stopper encoding schemes against other similar algorithms and each other. As a reference method for uncompressed texts, we use TBM (`uf_fwd_md2` from Hume and Sunday's widely known test bench [16]), and SBNDM2 [10]. As a reference method for compressed texts, we use BM-BPE [25] and Knuth-Morris-Pratt search algorithm on recursive-pairing compressed text developed by Maruyama et al. [18]. Their compression algorithm is denoted by BPX and their search algorithm for compressed text by KMP-BPX. The implementation is by courtesy of Maruyama. For BM-BPE, we used the recommended version with a maximum of 3 original characters per encoded character. The implementation is by courtesy of Takeda.

As to our own variants, we tested TBM and SBBDM2 based algorithms for SE_4 and $SE_{8,4}$ encoded tests (denoted by BM-SE_4 and BM-$SE_{8,4}$, SBNDM2-SE_4, and SBNDM2-$SE_{8,4}$, respectively). SE_4 is a representative of our compression scheme, and $SE_{8,4}$ represents code splitting. Note that $SE_{8,4}$ offers no compression at all.

The most important properties of these algorithms are search speed and compression ratio, in that order. In this experiment, only these properties are examined. Encoding and decoding speeds were not considered. In search, we focus on pattern lengths of 5, 10, 20, and 30. In the case of English and Finnish, the pattern length of 30 was not used, because some of the encoded patterns were too long for the 32-bit

Algorithm 1 SBNDM2 for SE_4-encoded text. ($P^2 = p_1p_2 \cdots p_{2m}$ is a string containing both the alignments of the encoded pattern, $T = t_1t_2 \cdots t_n$ is the encoded text.)

```
    /* Preprocessing */
 1: for all c ∈ Σ' do B[c] ← 0  /* c is a byte in the compressed text. */
 2: for all c_i, c_j ∈ Σ' do F[c_i c_j] ← 0
 3: for j ← 1 to 2m do
 4:     B[p_j] ← B[p_j] | (1 << (2m − j))
 5: for i ← 1 to 2m do
 6:     for j ← 1 to 2m do
 7:         F[p_i p_j] ← B[p_i] & (B[p_j] << 1)
 8: f ← a bit vector containing 0s in positions 0 (the least significant bit) and m and otherwise 1s.
    /* Searching */
 9: j ← m
10: while (true) do
11:     while (D ← F[t_{j−1} j t_j]) = 0 do
12:         j ← j + m − 1
13:     pos ← j
14:     while (D ← (D << 1) & f & B[t_{j−2}]) ≠ 0 do
15:         j ← j − 1
16:     j ← j + m − 2
17:     if j ≤ pos then
18:         if j > n then
19:             End of text reached, exit.
20:         A possible occurrence ending at pos found, check
21:         by matching the bytes one by one.
22:         j ← pos +1
```

bitvector used by SBNDM2-SE_4. (Because SBNDM2-SE_4 searches for two patterns at the same time, one which starts from the beginning of the encoded pattern and the other starting from the second half-byte of the encoded pattern, the maximum length of the encoded pattern is 34 half-bytes when the 32-bit bitvector is used.) After the preliminary experiments, we added tests for pattern lenghts of 6, 7, 8 and 9 for English texts and for pattern lengths of 11, 12, 13, 14 and 15 for Finnish texts to find out where SBNDM2-SE_4 becomes faster than SBNDM2.

As a DNA text, we used a 5 MB long part of a DNA of the fruitfly. As an English text, we used the KJV Bible such that the beginning of the text was concatenated to the end to lengthen the text to 5 MB. As a Finnish text, we used the Finnish Bible (1938). Because the original size of the Finnish text was about 4 MB, we concatenated the beginning of the text to the end to lengthen the text to 5 MB.

The compression ratios for various encoding algorithms are presented in Table 6. The ratios are given for the original (not extended) text files. For the compression of natural-language texts, the BPX algorithm is better and BPE slightly better than SE_4. This is true also for DNA text, because SE_4 always uses at least four bits for each symbol.

All experiments were run on a 2.40 GHz Lenovo ThinkPad t400s laptop with Debian Linux and the Intel® Core™2 Duo CPU P9400 2.40 GHz processor. The computer had 3851 megabytes of main memory. There were no other users using the test computer at the same time. The Linux function `sched_setaffinity` was used to bind the process to only one core. All the programs were compiled using `gcc` [version 4.6.3] with the optimization flag `-O3` and the flag `-m64` to produce 64 bit code. The

	KJV Bible	Finnish Bible	DNA
BPX	28.0%	32.6%	27.8%
BPE	51.0%	52.1%	34.0%
$SE_{8,4}$	100.0%	100.0%	100.0%
SE_4	58.8%	58.2%	50.0%

Table 6. Compression ratios.

programs were modified to measure their own execution time by using the `times()` function similarly to Hume and Sunday's test bench. The clocked time includes everything except program argument processing and reading the file containing the encoded text from disk.

In each test, the set of 200 patterns was searched for. The patterns were randomly picked from the corresponding text. The searches were performed sequentially one pattern at a time. The preprocessing was repeated 100 times and the searching 500 times for each pattern. The average times for the whole set are presented in Tables 7, 8, and 9, and graphically in Figures 1, 2, and 3.

For short DNA patterns, KMP-BPX and SBNDM2-SE_4 were the fastest. For longer DNA patterns, all our algorithms were clearly faster than KMP-BPX. SBNDM2-SE_4 was the fastest for longer patterns.

For short English patterns, SBNDM2 was clearly faster than our algorithms, which were comparable to KMP-BPX. For longer patterns, our algorithms outperformed KMP-BPX and BM-BPE, but mostly they were comparable to SBNDM2. SBNDM2-SE_4 was faster than SBNDM2 for longer patterns. To be more exact, SBNDM2-SE_4 was still slower than SBNDM2 for the pattern length 8, but the fastest among the tested methods for the pattern length 9.

The test results for Finnish patterns were rather similar to the results for English patterns, but the search times of SBNDM2-SE_4 and SBNDM2-$SE_{8,4}$ were nearer to each other. For some pattern lengths, SBNDM2-SE_4 was slightly faster than SBNDM2-$SE_{8,4}$, for some other pattern lengths it was the opposite. The smallest pattern length was 11 where SBNDM2-SE_4 and SBNDM2-$SE_{8,4}$ were faster than SBNDM2 and the fastest among the tested methods. Interestingly, most algorithmss were a bit faster with Finnish data than with English data.

pattern length → algorithm ↓	5	10	20	30
TBM	2336	1918	1779	1733
SBNDM2	1609	1014	582	407
BM-BPE	1387	728	424	336
KMP-BPX	969	951	952	957
BM-SE_4	1548	751	491	416
BM-$SE_{8,4}$	1562	766	501	425
SBNDM2-SE_4	974	397	210	169
SBNDM2-$SE_{8,4}$	1067	475	277	235

Table 7. Search times of DNA patterns in milliseconds, text size 5 MB. The pattern set contained 200 patterns.

pattern length → algorithm ↓	5	6	7	8	9	10	20
TBM	1072	939	831	756	706	668	465
SBNDM2	542	507	448	399	397	370	295
BM-BPE	1275	1250	923	871	860	701	431
KMP-BPX	960	959	960	959	960	962	966
BM-SE_4	925	749	626	543	489	440	239
BM-$SE_{8,4}$	1040	719	705	522	532	428	240
SBNDM2-SE_4	951	726	501	418	354	308	156
SBNDM2-$SE_{8,4}$	912	949	505	489	362	350	169

Table 8. Search times of English patterns in milliseconds, text size 5 MB. The pattern set contained 200 patterns.

pattern length → algorithm ↓	5	10	11	12	13	14	15	20
TBM	1007	630	593	567	540	527	504	439
SBNDM2	473	308	295	286	277	272	260	238
BM-BPE	1240	670	657	645	549	539	534	406
KMP-BPX	1118	1123	1126	1125	1127	1127	1126	1129
BM-SE_4	914	428	389	360	335	314	289	226
BM-$SE_{8,4}$	1000	403	414	345	348	298	301	222
SBNDM2-SE_4	969	311	266	240	221	205	189	143
SBNDM2-$SE_{8,4}$	907	327	254	254	210	212	182	148

Table 9. Search times of Finnish patterns in milliseconds, text size 5 MB. The pattern set contained 200 patterns.

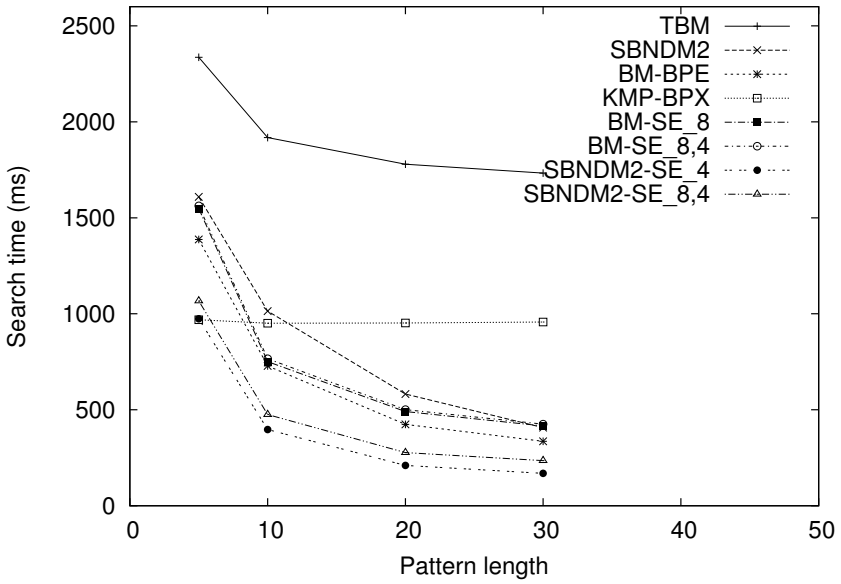


Figure 1. Search times of DNA patterns in milliseconds.

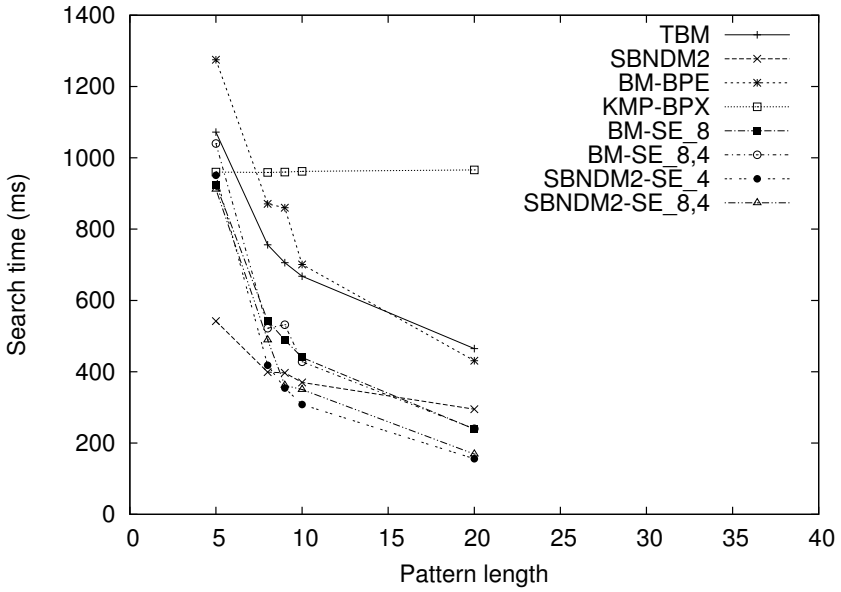


Figure 2. Search times of English patterns in milliseconds.

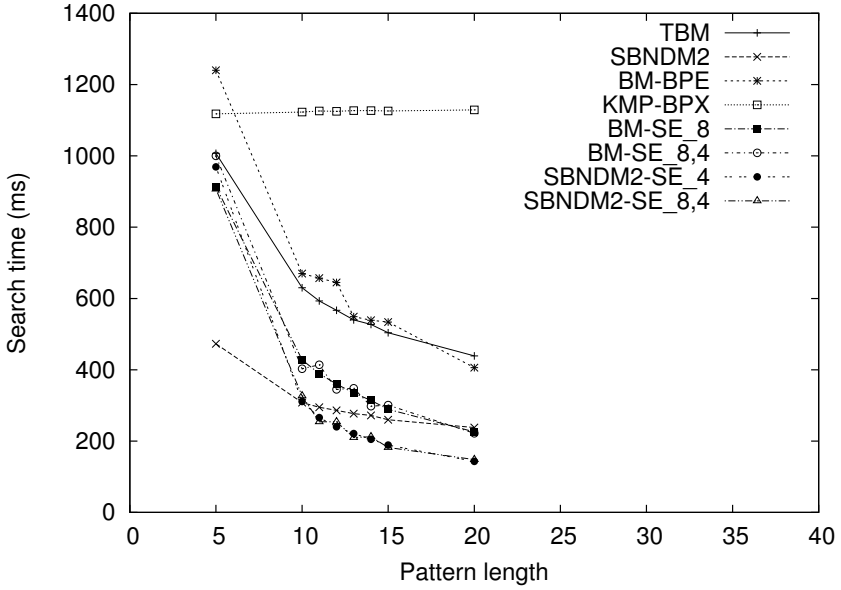


Figure 3. Search times of Finnish patterns in milliseconds.

5 Concluding remarks

Stopper encoding is a semi-static character-based compression method enabling fast searches. It is useful in applications where on-line updates and on-line decoding are needed. Stopper encoding resembles the semi-static Huffman encoding [15] and whole-word de Moura encoding [19]. Every character of the original text is encoded with variable-length codewords, which consist of fixed-length base symbols. The base symbols could have 2, 4, 6, or 8 bits. Base symbols of 6 or 8 bits allow code splitting to be applied, dividing the bits of each base symbol into two parts, which are stored, respectively, into two arrays in order to speed up searching.

We presented a new search algorithm for stopper encoding, based on the SBNDM2 algorithm [10]. Stopper encoding is not restricted to exact matching nor to this search algorithm, but it can be applied to string matching problems of other types as well.

We tested our algorithm experimentally. The running time of the search algorithm was compared with two reference algorithms for compressed texts as well as two other reference algorithms for uncompressed texts. Our SBNDM2 based search method SBNDM2-SE_4 was the fastest among all the tested algorithms for English patterns of *at least 9* characters and either SBNDM2-SE_4 or SBNDM2-$SE_{8,4}$ for Finnish patterns of *at least 11* characters. Especially, SBNDM2-SE_4 was faster than the standard search algorithms in uncompressed texts for these patterns.

References

1. A. Amir and G. Benson: *Efficient two-dimensional compressed matching*, in Proc. of DCC, IEEE, 1992, pp. 279–288.
2. A. Amir, G. Benson, and M. Farach: *Let sleeping files lie: Pattern matching in z-compressed files.* J. Comput. Syst. Sci., 52(2) 1996, pp. 299–307.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
4. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá: *Lightweight natural language text compression.* Inf. Retr., 10(1) 2007, pp. 1–33.
5. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá: *New adaptive compressors for natural language text.* Softw: Pract. Exper., 38(13) 2008, pp. 1429–1450.
6. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá: *Dynamic lightweight text compression.* ACM Trans. Inf. Syst., 28(3) 2010.
7. M. Crochemore and W. Rytter: *Jewels of stringology*, World Scientific, 2002.
8. J. S. Culpepper and A. Moffat: *Enhanced byte codes with restricted prefix properties*, in Proc. SPIRE, vol. 3772 of LNCS, Springer, 2005, pp. 1–12.
9. J. S. Culpepper and A. Moffat: *Phrase-based pattern matching in compressed text*, in Proc. SPIRE, vol. 4209 of LNCS, Springer, 2006, pp. 337–345.
10. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
11. P. Ferragina and G. Manzini: *An experimental study of an opportunistic index*, in Proc. SODA, ACM/SIAM, 2001, pp. 269–278.
12. K. Fredriksson and F. Nikitin: *Simple compression code supporting random access and fast string matching*, in Proc. WEA, vol. 4525 of LNCS, Springer, 2007, pp. 203–216.
13. P. Gage: *A new algorithm for data compression.* C Users J., 12(2) Feb. 1994, pp. 23–38.
14. R. N. Horspool: *Practical fast searching in strings.* Softw: Pract. Exper., 10(6) 1980, pp. 501–506.
15. D. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) Sept 1952, pp. 1098–1101.
16. A. Hume and D. Sunday: *Fast string searching.* Softw: Pract. Exper., 21(11) 1991, pp. 1221–1248.
17. 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.
18. S. Maruyama, Y. Tanaka, H. Sakamoto, and M. Takeda: *Context-sensitive grammar transform: Compression and pattern matching*, in Proc. SPIRE, vol. 5280 of LNCS, Springer, 2008, pp. 27–38.
19. E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Trans. Inf. Syst., 18(2) 2000, pp. 113–139.
20. G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa: *Faster approximate string matching over compressed text*, in DCC, IEEE, 2001, pp. 459–468.
21. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM Journal of Experimental Algorithmics, 5 2000, p. 4.

22. G. Navarro and J. Tarhio: *Boyer-Moore string matching over Ziv-Lempel compressed text*, in Proc. CPM, vol. 1848 of LNCS, Springer, 2000, pp. 166–180.
23. J. Rautio: *Context-dependent stopper encoding*, in Proc. Stringology, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005, pp. 143–152.
24. J. Rautio, J. Tanninen, and J. Tarhio: *String matching with stopper encoding and code splitting*, in Proc. CPM, vol. 2373 of LNCS, Springer, 2002, pp. 42–52.
25. Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa: *A Boyer-Moore type algorithm for compressed pattern matching*, in Proc. CPM, vol. 1848 of LNCS, Springer, 2000, pp. 181–194.
26. I. H. Witten, A. Moffat, and T. C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann, 1999.
27. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proc. USENIX Technical Conference, Winter 1992, pp. 153–162.
28. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.
29. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.

Threshold Approximate Matching in Grammar-Compressed Strings

Alexander Tiskin*

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
tiskin@dcs.warwick.ac.uk

Abstract. A grammar-compressed (GC) string is a string generated by a context-free grammar. This compression model captures many practical applications, and includes LZ78 and LZW compression as a special case. We give an efficient algorithm for threshold approximate matching on a GC-text against a plain pattern. Our algorithm improves on existing algorithms whenever the pattern is sufficiently long. The algorithm employs the technique of fast unit-Monge matrix distance multiplication, as well as a new technique for implicit unit-Monge matrix searching, which we believe to be of independent interest.

1 Introduction

String compression is a standard approach to dealing with massive data sets. From an algorithmic viewpoint, it is natural to ask whether compressed strings can be processed efficiently without decompression. For a recent survey on the topic, see Lohrey [14]. Efficient algorithms for compressed strings can also be applied to achieve speedup over ordinary string processing algorithms for plain strings that are highly compressible.

We consider the following general model of compression.

Definition 1. *Let t be a string of length n (typically large). String t will be called a* grammar-compressed string (GC-string)*, if it is generated by a context-free grammar, also called a* straight-line program (SLP)*. An SLP of length $\bar{n}$, $\bar{n} \leq n$, is a sequence of $\bar{n}$* statements*. A statement numbered k, $1 \leq k \leq \bar{n}$, has one of the following forms: $t_k = \alpha$, where α is an alphabet character, or $t_k = t_i t_j$, where $1 \leq i, j < k$.*

We identify every symbol t_r with the string it represents; in particular, we have $t = t_{\bar{n}}$. In general, the plain string length n can be exponential in the GC-string length $\bar{n}$. Grammar compression includes as a special case the classical LZ78 and LZW compression schemes by Ziv, Lempel and Welch [24,22].

Approximate pattern matching is a natural generalisation of both the ordinary (exact) pattern matching, and of the alignment score and the edit distance problems. Given a text string t of length n and a pattern string p of length $m \leq n$, the *approximate pattern matching problem* asks to find the substrings of the text that are locally closest to the pattern, i.e. that have the locally highest alignment score (or, equivalently, lowest edit distance) against the pattern. The precise definition of "locally" may vary in different versions of the problem.

Definition 2. *The* threshold approximate matching problem *(often called simply "approximate matching") assumes an alignment score with arbitrary weights, and,*

* Research supported by the Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, EPSRC award EP/D063191/1.

Alexander Tiskin: Threshold Approximate Matching in Grammar-Compressed Strings, pp. 124–138.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

given a threshold score h, asks for all substrings of text t that have alignment score $\geq h$ against pattern p.

The substrings asked for by Definition 2 will be called *matching substrings*. The precise definition of "alignment score with weights" will be given in Subsection 2.5.

An important special case arises when the unweighted LCS score is chosen as the alignment score, and the pattern length m is chosen as the threshold h. This special case is known as the *local subsequence recognition problem*.

Approximate pattern matching on compressed text has been studied by Kärkkäinen et al. [11]. For a GC-text of length $\bar{n}$, an uncompressed pattern of length m, and an edit distance threshold k, the (suitably generalised) algorithm of [11] solves the threshold approximate matching problem in time $O(m\bar{n}k^2 + output)$. In the special case of LZ78 or LZW compression, the running time is reduced to $O(m\bar{n}k + output)$. Bille et al. [5] gave an efficient general scheme for adapting an arbitrary threshold approximate matching algorithm to work on a GC-text. In particular, when the algorithms by Landau and Vishkin [12] and by Cole and Hariharan [8] are each plugged into their scheme, the resulting algorithm runs in time $O(\bar{n} \cdot \min(mk, k^4 + m) + \bar{n}^2 + output)$. In the special case of LZ78 or LZW compression, Bille et al. [4] show that the running time can be reduced to $O(\bar{n} \cdot \min(mk, k^4 + m) + output)$.

For the special case of the local subsequence recognition problem on a GC-text, Cégielski et al. [6] gave an algorithm running in time $O(m^2 \log m \cdot \bar{n} + output)$. In [21], we improved the running time to $O(m \log m\bar{n} + output)$. Recently, Yamamoto et al. [23] improved the running time still further to $O(m\bar{n} + output)$.

In this paper, we consider the threshold approximate matching problem on a GC-text against a plain pattern. We give an algorithm running in time $O(m \log m \cdot \bar{n} + output)$, which improves on existing algorithms whenever the pattern is sufficiently long. The algorithm employs the technique of fast unit-Monge matrix distance multiplication [16], as well as a new technique for implicit unit-Monge matrix searching (Lemma 12).

This paper is a sequel to [21]; we encourage the reader to refer there for a warm-up to the result and the techniques presented in the current paper. Due to space constraints, some proofs and examples are omitted. They can be found in [17].

2 General techniques

We recall the framework developed in [18,19,20,16,21], and fully presented in [17].

2.1 Preliminaries

For indices, we will use either integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, or half-integers $\{\ldots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots\}$. For ease of reading, half-integer variables will be indicated by hats (e.g. $\hat{\imath}$, $\hat{\jmath}$). Ordinary variable names (e.g. i, j, with possible subscripts or superscripts), will normally denote integer variables, but can sometimes denote a variable that may be either integer, or half-integer.

It will be convenient to denote $i^- = i - \frac{1}{2}$, $i^+ = i + \frac{1}{2}$ for any integer or half-integer i. The set of all half-integers can now be written as $\{\ldots, (-3)^+, (-2)^+, (-1)^+, 0^+, 1^+, 2^+, \ldots\}$. We denote integer and half-integer *intervals* by $[i : j] = \{i, i+1, \ldots, j-1, j\}$, $\langle i : j \rangle = \{i^+, i + \frac{3}{2}, \ldots, j - \frac{3}{2}, j^-\}$. In both cases, the interval is defined by integer endpoints.

Given two index ranges I, J, it will be convenient to denote their Cartesian product by $(I \mid J)$. We extend this notation to Cartesian products of intervals:

$$[i_0 : i_1 \mid j_0 : j_1] = ([i_0 : i_1] \mid [j_0 : j_1]) \qquad \langle i_0 : i_1 \mid j_0 : j_1 \rangle = (\langle i_0 : i_1 \rangle \mid \langle j_0 : j_1 \rangle)$$

Given index ranges I, J, a *vector over* I is indexed by $i \in I$, and a *matrix over* $(I \mid J)$ is indexed by $i \in I$, $j \in J$.

We will use the parenthesis notation for indexing matrices, e.g. $A(i, j)$. We will also use straightforward notation for selecting subvectors and submatrices: for example, given a matrix A over $[0 : n \mid 0 : n]$, we denote by $A[i_0 : i_1 \mid j_0 : j_1]$ the submatrix defined by the given sub-intervals. A star $*$ will indicate that for a particular index, its whole range is being used, e.g. $A[* \mid j_0 : j_1] = A[0 : n \mid j_0 : j_1]$. In particular, $A(*, j)$ and $A(i, *)$ will denote a full matrix column and row, respectively.

We recall the following definitions from [21,17]. We define two natural strict partial orders on points, called $\ll$- and $\gtrless$-*dominance*:

$$(i_0, j_0) \ll (i_1, j_1) \quad \text{if } i_0 < i_1 \text{ and } j_0 < j_1 \qquad (i_0, j_0) \gtrless (i_1, j_1) \quad \text{if } i_0 > i_1 \text{ and } j_0 < j_1$$

When visualising points, we will deviate from the standard Cartesian visualisation of the coordinate axes, and will use instead the matrix indexing convention: the first coordinate in a pair increases downwards, and the second coordinate rightwards. Hence, $\ll$- and $\gtrless$-dominance correspond respectively to the "above-left" and "below-left" partial orders. The latter order corresponds to the standard visual convention for dominance in computational geometry.

Definition 3. *Let D be a matrix over $\langle i_0 : i_1 \mid j_0 : j_1 \rangle$. Its* distribution matrix D^Σ *over $[i_0 : i_1 \mid j_0 : j_1]$ is defined by $D^\Sigma(i, j) = \sum_{\hat{\imath} \in \langle i : i_1 \rangle, \hat{\jmath} \in \langle j_0 : j \rangle} D(\hat{\imath}, \hat{\jmath})$ for all $i \in [i_0 : i_1]$, $j \in [j_0 : j_1]$.*

Definition 4. *Let A be a matrix over $[i_0 : i_1 \mid j_0 : j_1]$. Its* density matrix $A^\square$ *over $\langle i_0 : i_1 \mid j_0 : j_1 \rangle$ is defined by $A^\square(\hat{\imath}, \hat{\jmath}) = A(\hat{\imath}^+, \hat{\jmath}^-) - A(\hat{\imath}^-, \hat{\jmath}^-) - A(\hat{\imath}^+, \hat{\jmath}^+) + A(\hat{\imath}^-, \hat{\jmath}^+)$ for all $\hat{\imath} \in \langle i_0 : i_1 \rangle$, $\hat{\jmath} \in \langle j_0 : j_1 \rangle$.*

Definition 5. *Matrix A over $[i_0 : i_1 \mid j_0 : j_1]$ will be called* simple*, if $A(i_1, j) = A(i, j_0) = 0$ for all i, j. Equivalently, A is simple if $A^{\square\Sigma} = A$.*

Definition 6. *Matrix A is called* totally monotone*, if $A(i, j) > A(i, j') \Rightarrow A(i', j) > A(i', j')$ for all $i \le i'$, $j \le j'$.*

Definition 7. *Matrix A is called* a Monge matrix*, if $A(i, j) + A(i', j') \le A(i, j') + A(i', j)$ for all $i \le i'$, $j \le j'$. Equivalently, matrix A is a Monge matrix, if $A^\square$ is nonnegative. Matrix A is called* an anti-Monge matrix*, if $-A$ is Monge.*

Definition 8. *A* permutation *(respectively,* subpermutation*)* matrix *is a zero-one matrix containing exactly one (respectively, at most one) nonzero in every row and every column.*

Definition 9. *0 Matrix A is called* a unit-Monge *(respectively,* subunit-Monge*)* matrix*, if $A^\square$ is a permutation (respectively, subpermutation) matrix. Matrix A is called* a unit-anti-Monge *(respectively,* subunit-anti-Monge*)* matrix*, if $-A$ is unit-Monge (respectively, subunit-Monge).*

A permutation matrix P of size n can be regarded as an implicit representation of the simple unit-Monge matrix P^{Σ}. Geometrically, a value $P^{\Sigma}(i,j)$ is the number of (half-integer) nonzeros in matrix P that are $\lessgtr$-dominated by the (integer) point (i,j). Matrix P can be preprocessed to allow efficient element queries on $P^{\Sigma}(i,j)$. Here, we consider *incremental queries*, which are given an element of an implicit simple (sub)unit-Monge matrix, and return the value of an adjacent element. This kind of query can be answered directly from the (sub)permutation matrix, without any preprocessing.

Theorem 10. *Given a (sub)permutation matrix P of size n, and the value $P^{\Sigma}(i,j)$, $i,j \in [0:n]$, the values $P^{\Sigma}(i \pm 1, j)$, $P^{\Sigma}(i, j \pm 1)$, where they exist, can be queried in time $O(1)$.*

Proof. Straightforward; see [17].

2.2 Implicit matrix searching

We recall a classical row minima searching algorithm by Aggarwal et al. [1], often nicknamed the "SMAWK algorithm".

Lemma 11 ([1]). *Let A be an $n_1 \times n_2$ implicit totally monotone matrix, where each element can be queried in time q. The problem of finding the (say, leftmost) minimum element in every row of A can be solved in time $O(qn)$, where $n = \max(n_1, n_2)$.*

Proof (Lemma 11). Without loss of generality, let A be over $[0:n \mid 0:n]$. Let B be an implicit $\frac{n}{2} \times n$ matrix over $\left[0:\frac{n}{2} \mid 0:n\right]$, obtained by taking every other row of A. Clearly, at most $\frac{n}{2}$ columns of B contain a leftmost row minimum. The key idea of the algorithm is to eliminate $\frac{n}{2}$ of the remaining columns in an efficient process, based on the total monotonicity property.

We call a matrix element *marked* (for elimination), if its column has not (yet) been eliminated, but the element is already known not to be a leftmost row minimum. A column gets eliminated when all its elements become marked.

Initially, both the set of eliminated columns and the set of marked elements are empty. In the process of column elimination, marked elements may only be contained in the i leftmost uneliminated columns; the value of i is initially equal to 1, and gets either incremented or decremented in every step of the algorithm. The marked elements form a *staircase*: in the first, second, ..., i-th uneliminated column, respectively zero, one, ..., $i-1$ topmost elements are marked. In every iteration of the algorithm, two outcomes are possible: either the staircase gets extended to the right to the $i+1$-st uneliminated column, or the whole i-th uneliminated column gets eliminated, and therefore deleted from the staircase.

Let j, j' denote respectively the indices of the i-th and $i+1$-st uneliminated column in the original matrix (across both uneliminated and eliminated columns). The outcome of the current iteration depends on the comparison of element $B(i,j)$, which is the topmost unmarked element in the i-th uneliminated column, against element $B(i,j')$, which is the next uneliminated (and unmarked) element immediately to its right. The outcomes of this comparison and the rest of the elimination procedure are given in Table 1. By storing indices of uneliminated columns in an appropriate dynamic data structure, such as a doubly-linked list, a single iteration of this procedure can be implemented to run in time $O(q)$. The whole procedure runs in time $O(qn)$, and eliminates $\frac{n}{2}$ columns.

$i \leftarrow 0$; $j \leftarrow 0$; $j' \leftarrow 1$
while $j' \leq n$:
  case $B(i,j) \leq B(i,j')$:
    case $i < \frac{n}{2}$: $i \leftarrow i+1$; $j \leftarrow j'$
    case $i = \frac{n}{2}$: eliminate column j'
    $j' \leftarrow j'+1$
  case $B(i,j) > B(i,j')$:
    eliminate column j
    case $i = 0$: $j \leftarrow j'$; $j' \leftarrow j'+1$
    case $i > 0$: $i \leftarrow i-1$; $j \leftarrow \max\{k : k \text{ uneliminated and } < j\}$

Table 1. Elimination procedure of Lemmas 11 and 12.

Let A' be the $\frac{n}{2} \times \frac{n}{2}$ matrix obtained from B by deleting the $\frac{n}{2}$ eliminated columns. We call the algorithm recursively on A'. This recursive call returns the leftmost row minima of A', and therefore also of B. It is now straightforward to fill in the leftmost minima in the remaining rows of A in time $O(qn)$. Thus, the top level of recursion runs in time $O(qn)$. The amount of work gets halved with every recursion level, therefore the overall running time is $O(qn)$.

Let us now restrict our attention to implicit unit-Monge matrices. An element of such a matrix (represented by an appropriate data structure, see [17]) can be queried in time $q = O(\log^2 n)$. A more careful analysis of the elimination procedure of Lemma 11 shows that the required matrix elements can be obtained, instead of standalone element queries, by the more efficient incremental queries of Theorem 10.

Lemma 12. *Let A be an implicit (sub)unit-Monge matrix over $[0 : n_1 \mid 0 : n_2]$, represented by the (sub)permutation matrix $P = A^{\square}$ and vectors $b = A(n_1, *)$, $c = A(*, 0)$, such that $A(i,j) = P^{\Sigma}(i,j) + b(j) + c(i) - b(0)$ for all i, j. The problem of finding the (say, leftmost) minimum element in every row of A can be solved in time $O(n \log\log n)$, where $n = \max(n_1, n_2)$.*

Proof (Lemma 12). First, observe that vector c has no effect on the positions (as opposed to the values) of any row minima. Therefore, we assume without loss of generality that $c(i) = 0$ for all i (and, in particular, $b(0) = c(n_1) = 0$). Further, suppose that some column $P(*, \hat{\jmath})$ is identically zero; then, depending on whether $b(\hat{\jmath}^-) \leq b(\hat{\jmath}^+)$ or $b(\hat{\jmath}^-) > b(\hat{\jmath}^+)$, we may delete respectively column $A(*, \hat{\jmath}^+)$ or $A(*, \hat{\jmath}^-)$ as it does not contain any leftmost row minima. Also, suppose that some row $P(\hat{\imath}, *)$ is identically zero; then the minimum value in row $A(\hat{\imath}^-, *)$ lies in the same column as the minimum value in row $A(\hat{\imath}^-, *)$, hence we can delete one of these rows. Therefore, we assume without loss of generality that A is an implicit unit-Monge matrix over $[0 : n \mid 0 : n]$, and hence P is a permutation matrix.

To find all the leftmost row minima, we adopt the column elimination procedure of Lemma 11 (see Table 1), with some modifications outlined below.

Let B be an implicit $n^{1/2} \times n$ matrix, obtained by taking a subset of $n^{1/2}$ rows of A at regular intervals of $n^{1/2}$. Clearly, at most $n^{1/2}$ columns of B contain a leftmost row minimum. We need to eliminate $n - n^{1/2}$ of the remaining columns.

Let B be over $\left[0 : \frac{n}{2} \mid 0 : n\right]$. Throughout the elimination procedure, we maintain a vector $d(i)$, $i \in [0 : n^{1/2} - 1]$, initialised by zero values. In every iteration, given a current value of the index j', each value $d(i)$ gives the count of nonzeros $P(s,t) = 1$ within the rectangle $s \in \left\langle n^{1/2} i : n^{1/2}(i+1) \right\rangle$, $t \in \langle 0 : j' \rangle$.

Consider an iteration of the column elimination procedure of Lemma 11 with given values i, j, j', operating on matrix elements $B(i,j)$, $B(i,j')$. For the iteration that follows the current one, the following matrix elements may be required:

- $B(i-1,j')$, $B(i+1,j')$. These values can be obtained respectively as $B(i,j)+d(i-1)$ and $B(i,j)-d(i)$.
- $B(i,j'+1)$, $B(i+1,j'+1)$. These values can be obtained respectively from $B(i,j')$, $B(i+1,j')$ by a rowwise incremental query of matrix P^{Σ} via Theorem 10, plus a single access to vector b.
- $B(i-1,\{k : k \text{ uneliminated and } < j\})$. This element was already queried in the iteration at which its column was first added to the staircase. There is at most one such element per column, therefore each of them can be stored and subsequently queried in constant time.

At the end of the current iteration, index j' may be incremented (i.e. the staircase may grow by one column). In this case, we also need to update vector d for the next iteration. Let $s \in \langle 0 : n \rangle$ be such that $P(s, j' - \frac{1}{2}) = 1$. Let $i = \lfloor s/n^{1/2} \rfloor$; we have $s \in \langle n^{1/2}i : n^{1/2}(i+1) \rangle$. The update consists in incrementing $d(i)$ by 1.

The total number of iterations in the elimination procedure is at most $2n$. This is because in total, at most n columns are added to the staircase, and at most n (in fact, exactly $n - n^{1/2}$) columns are eliminated. Therefore, the elimination procedure runs in time $O(n)$.

Let A' be the $n^{1/2} \times n^{1/2}$ matrix obtained from B by deleting the $n - n^{1/2}$ eliminated columns. Using incremental queries to matrix P, it is straightforward to obtain matrix A' explicitly in random-access memory in time $O(n)$. We now call the algorithm of Lemma 11 to compute the row minima of A', and therefore also of B, in time $O(n)$.

We now need to fill in the remaining row minima of matrix A. The row minima of matrix A' define a chain of $n^{1/2}$ submatrices in A at which these remaining row minima may be located. More specifically, given two successive row minima of A', all the $n^{1/2}$ row minima that are located between the two corresponding rows in A must also be located between the two corresponding columns. Each of the resulting submatrices has $n^{1/2}$ rows; the number of columns may vary from submatrix to submatrix. It is straightforward to eliminate from each submatrix all columns not containing any nonzero of matrix P; therefore, without loss of generality, we may assume that every submatrix is of size $n^{1/2} \times n^{1/2}$.

We now call the algorithm recursively on each submatrix to fill in the remaining leftmost row minima. The amount of work remains $O(n)$ in every recursion level. There are $\log \log n$ recursion levels, therefore the overall running time of the algorithm is $O(n \log \log n)$.

An even faster algorithm, running in optimal time $O(n)$ on the RAM model, has been recently suggested by Gawrychowski [9]. The algorithm of Lemma 12, which is thus suboptimal but has weaker model requirements, will be sufficient for the purposes of this paper.

2.3 Semi-local LCS

We will consider strings of characters taken from an alphabet. Two alphabet characters α, β *match*, if $\alpha = \beta$, and *mismatch* otherwise. In addition to alphabet characters, we introduce two special extra characters: the *guard character* '$', which only matches

itself and no other characters, and the *wildcard character* '?', which matches itself and all other characters.

It will be convenient to index strings by half-integer, rather than integer indices, e.g. string $a = \alpha_{0^+}\alpha_{1^+}\cdots\alpha_{m^-}$. We will index strings as vectors, writing e.g. $a(\hat{\imath}) = \alpha_{\hat{\imath}}$, $a\langle i : j\rangle = \alpha_{i^+}\cdots\alpha_{j^-}$. Given strings a over $\langle i : j\rangle$ and b over $\langle i' : j'\rangle$, we will distinguish between string *right concatenation* $\underline{a}b$, which is over $\langle i : j + j' - i'\rangle$ and preserves the indexing within a, and *left concatenation* $a\underline{b}$, which is over $\langle i' - j + i : j'\rangle$ and preserves the indexing within b. We extend this notation to concatenation of more than two strings, e.g. $a\underline{b}c$ is a concatenation of three strings, where the indexing of the second string is preserved. If no string is marked in the concatenation, then right concatenation is assumed by default.

Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. Special cases of a substring are *a prefix* and *a suffix* of a string. Unless indicated otherwise, an algorithm's input is a string a of length m, and a string b of length n.

We recall the following definitions from [21,17].

Definition 13. *Given strings a, b, the* longest common subsequence (LCS) problem *asks for the length of the longest string that is a subsequence of both a and b. We will call this length the* LCS score *of strings a, b.*

Definition 14. *Given strings a, b, the* semi-local LCS problem *asks for the LCS scores as follows: a against every substring of b (the* string-substring LCS scores*); every prefix of a against every suffix of b (the* prefix-suffix LCS scores*); symmetrically, the* substring-string LCS scores *and the* suffix-prefix LCS scores*, defined as above but with the roles of a and b exchanged. The first three (respectively, the last three) components, taken together, will also be called the* extended string-substring *(respectively,* substring-string*)* LCS problem*.*

Definition 15. *A* grid-diagonal dag *is a weighted dag, defined on the set of nodes $v_{l,i}$, $l \in [0 : m]$, $i \in [0 : n]$. The edge and path weights are called* scores*. For all $l \in [0 : m]$, $\hat{l} \in \langle 0 : m\rangle$, $i \in [0 : n]$, $\hat{\imath} \in \langle 0 : n\rangle$, the grid-diagonal dag contains:*

- *the horizontal edge $v_{l,\hat{\imath}^-} \to v_{l,\hat{\imath}^+}$ and the vertical edge $v_{\hat{l}^-,i} \to v_{\hat{l}^+,i}$, both with score 0;*
- *the diagonal edge $v_{\hat{l}^-,\hat{\imath}^-} \to v_{\hat{l}^+,\hat{\imath}^+}$ with score either 0 or 1.*

Definition 16. *An instance of the semi-local LCS problem on strings a, b corresponds to an $m \times n$ grid-diagonal dag $\mathrm{G}_{a,b}$, called the* alignment dag *of a and b. A cell indexed by $\hat{l} \in \langle 0 : m\rangle$, $\hat{\imath} \in \langle 0 : n\rangle$ is called a* match cell*, if $a(\hat{l})$ matches $b(\hat{\imath})$, and a* mismatch cell *otherwise (recall that the strings may contain wildcard characters). The diagonal edges in match cells have score 1, and in mismatch cells score 0.*

Definition 17. *Given strings a, b, the corresponding* semi-local score matrix *is a matrix over $[-m : n \mid 0 : m+n]$, defined by $\mathrm{H}_{a,b}(i,j) = \max score(v_{0,i} \rightsquigarrow v_{m,j})$, where $i \in [-m : n]$, $j \in [0 : m+n]$, and the maximum is taken across all paths between the given endpoints $v_{0,i}$, $v_{m,j}$ in the $m \times (2m+n)$ padded alignment dag $\mathrm{G}_{a,?^m\underline{b}?^m}$. If $i = j$, we have $\mathrm{H}_{a,b}(i,j) = 0$. By convention, if $j < i$, then we let $\mathrm{H}_{a,b}(i,j) = j - i < 0$.*

Theorem 18. *Given strings a, b, the corresponding semi-local score matrix $\mathrm{H}_{a,b}$ is unit-anti-Monge. More precisely, we have $\mathrm{H}_{a,b}(i,j) = j-i-\mathrm{P}^{\Sigma}_{a,b}(i,j) = m-\mathrm{P}^{T\Sigma T}_{a,b}(i,j)$, where $\mathrm{P}_{a,b}$ is a permutation matrix over $\langle -m : n \mid 0 : m+n\rangle$. In particular, string a is a subsequence of substring $b\langle i : j\rangle$ for some $i, j \in [0 : n]$, if and only if $\mathrm{P}^{T\Sigma T}_{a,b}(i,j) = 0$.*

Definition 19. *Given strings a, b, the* semi-local seaweed matrix *is a permutation matrix $\mathrm{P}_{a,b}$ over $\langle -m : n \mid 0 : m+n \rangle$, defined by Theorem 18.*

When talking about semi-local score and seaweed matrices, we will sometimes omit the qualifier "semi-local", as long as it is clear from the context.

2.4 Seaweed submatrix notation

The four individual components of the semi-local LCS problem correspond to a partitioning of both the score matrix $\mathrm{H}_{a,b}$ and the seaweed matrix $\mathrm{P}_{a,b}$ into submatrices. It will be convenient to introduce a special notation for the resulting subranges of their respective index ranges $[-m : n \mid 0 : m+n]$ and $\langle -m : n \mid 0 : m+n \rangle$. This notation will be used as matrix superscripts, e.g. $\mathrm{H}^{\blacksquare\square}_{a,b} = \mathrm{H}_{a,b}[0 : n \mid 0 : n]$ denotes the matrix of all string-substring LCS scores for strings a, b. The notation for other subranges is as follows.

	$0 : n$	$n : m+n$
$-m : 0$	$\mathrm{H}^{\blacksquare\blacksquare}_{a,b}$	$\mathrm{H}^{\square\blacksquare}_{a,b}$
$0 : n$	$\mathrm{H}^{\blacksquare\square}_{a,b}$	$\mathrm{H}^{\square\square}_{a,b}$

and analogously for $\mathrm{P}_{a,b}$. Note that the four defined half-integer subranges of matrix $\mathrm{P}_{a,b}$ are disjoint, but the corresponding four integer subranges of matrix $\mathrm{H}_{a,b}$ overlap by one row/column at the boundaries.

Definition 20. *Given strings a, b, the corresponding* suffix-prefix, substring-string, string-substring and prefix-suffix score *(respectively,* seaweed*)* matrices *are the submatrices $\mathrm{H}^{\blacksquare\blacksquare}_{a,b}$, $\mathrm{H}^{\square\blacksquare}_{a,b}$, $\mathrm{H}^{\blacksquare\square}_{a,b}$, $\mathrm{H}^{\square\square}_{a,b}$ (respectively, $\mathrm{P}^{\blacksquare\blacksquare}_{a,b}$, $\mathrm{P}^{\square\blacksquare}_{a,b}$, $\mathrm{P}^{\blacksquare\square}_{a,b}$, $\mathrm{P}^{\square\square}_{a,b}$). The defined seaweed submatrices are all disjoint; the defined score submatrices overlap by one row/column at the boundaries. In particular, the global LCS score $\mathrm{H}_{a,b}(0,n)$ belongs to all four score submatrices.*

The nonzeros of each seaweed submatrix introduced in Definition 20 can be regarded as an implicit solution to the corresponding component of the semi-local LCS problem. Similarly, by considering only three out of the four submatrices, we can define an implicit solution to the extended string-substring (respectively, substring-string) LCS problem.

Definition 21. *Given strings a, b, we define the* extended string-substring *(respectively,* substring-string*)* seaweed matrix *over $\langle -m : n \mid 0 : m+n \rangle$ as*

$$\mathrm{P}^{\blacksquare\blacksquare}_{a,b} = \begin{bmatrix} \mathrm{P}^{\blacksquare\square}_{a,b} & \cdot \\ \mathrm{P}^{\square\blacksquare}_{a,b} & \mathrm{P}^{\square\square}_{a,b} \end{bmatrix} \qquad \mathrm{P}^{\square\blacksquare}_{a,b} = \begin{bmatrix} \mathrm{P}^{\blacksquare\square}_{a,b} & \mathrm{P}^{\square\square}_{a,b} \\ \cdot & \mathrm{P}^{\square\square}_{a,b} \end{bmatrix}$$

The extended string-substring seaweed matrix $\mathrm{P}^{\blacksquare\blacksquare}_{a,b}$ contains at least n and at most $\min(m+n, 2n)$ nonzeros. Note that for $m \geq n$, the number of nonzeros in $\mathrm{P}^{\blacksquare\blacksquare}_{a,b}$ is at most $2n$, which is convenient when m is large. Analogously, for $m \leq n$, the number of nonzeros in the extended substring-string matrix $\mathrm{P}^{\square\blacksquare}_{a,b}$ is at most $2m$, which is convenient when n is large.

Let string a of length m be a concatenation of two fixed strings: $a = a'a''$, where a', a'' are nonempty strings of length m', m'' respectively, and $m = m' + m''$. A substring of the form $a\langle i' : i'' \rangle$ with $i' \in [0 : m'-1]$, $i'' \in [m'+1 : m]$ will be called a *cross-substring*. In other words, a cross-substring of a consists of a nonempty suffix

of a' and a nonempty prefix of a''. A cross-substring that is a prefix or a suffix of a will be called a *cross-prefix* and a *cross-suffix*, respectively. Given string b of length n that is a concatenation of two fixed strings, $b = b'b''$, cross-substrings of b are defined analogously.

Definition 22. *Given strings $a = a'a''$ and b, the corresponding* cross-semi-local score matrix *is the submatrix $\mathrm{H}_{a',a'';b} = \mathrm{H}_{a,b}[-m' : n \mid 0 : m''+n]$. Symmetrically, given strings a and $b = b'b''$, the corresponding* cross-semi-local score matrix *is the submatrix $\mathrm{H}_{a;b',b''} = \mathrm{H}_{a,b}[-m : n' \mid n' : m+n]$. The* cross-semi-local seaweed matrices *are defined analogously: $\mathrm{P}_{a',a'';b} = \mathrm{P}_{a,b}\langle -m' : n \mid 0 : m''+n\rangle$, $\mathrm{P}_{a;b',b''} = \mathrm{P}_{a,b}\langle -m : n' \mid n' : m+n\rangle$.*

A cross-semi-local score matrix represents the solution of a restricted version of the semi-local LCS problem. In this version, instead of all substrings (prefixes, suffixes) of string a (respectively, b), we only consider cross-substrings (cross-prefixes, cross-suffixes). At the submatrix boundaries $\mathrm{H}_{a',a'';b}(*, m''+n)$ and $\mathrm{H}_{a',a'';b}(-m', *)$, cross-substrings of string a degenerate to suffixes of a' and prefixes of a''; in particular, cross-prefixes and cross-suffixes of a degenerate respectively to the whole a' and a''. The submatrix boundaries $\mathrm{H}_{a;b',b''}(*, n')$ and $\mathrm{H}_{a;b',b''}(n', *)$ correspond to similar degenerate cross-substrings of string b.

As before, the cross-semi-local seaweed matrix $\mathrm{P}_{a',a'';b}$ (respectively, $\mathrm{P}_{a;b',b''}$) gives an implicit representation for the corresponding score matrix $\mathrm{H}_{a',a'';b}$ (respectively, $\mathrm{H}_{a;b',b''}$).

Occasionally, we will use cross-semi-local score and seaweed matrices in combination with the superscript subrange notation, introduced earlier in this section. In such cases, the range of the resulting matrix will be determined by the intersection of the ranges implied by the superscript and the subscript. For example, matrix $\mathrm{H}^{\square}_{a;b',b''} = \mathrm{H}_{a,b}[0 : n' \mid n' : n]$ is the matrix of all LCS scores between string a and all cross-substrings of string $b = b'b''$.

2.5 Weighted scores and edit distances

The concept of LCS score is generalised by that of *(weighted) alignment score*. An *alignment* of strings a, b is obtained by putting a subsequence of a into one-to-one correspondence with a (not necessarily identical) subsequence of b, character by character and respecting the index order. The corresponding pair of characters, one from a and the other from b, are said to be *aligned*. A character that is not aligned against a character of another string is said to be aligned against a *gap* in that string. Each of the resulting character alignments is given a real *weight*:

- a pair of aligned matching characters has weight $w_{\mathsf{m}} \geq 0$;
- a pair of aligned mismatching characters has weight $w_{\mathsf{x}} < w_{\mathsf{m}}$;
- a gap-character or character-gap pair has weight $w_{\mathsf{g}} \leq \frac{1}{2}w_{\mathsf{x}}$; it is normally assumed that $w_{\mathsf{g}} \leq 0$ (i.e. this weight is in fact a penalty).

The intuition behind the weight inequalities is as follows: aligning a matching pair of characters is always better than aligning a mismatching pair of characters, which in its turn is never worse than leaving both characters unaligned (aligned against a gap).

Definition 23. *The* (weighted) alignment score *for strings a, b is the maximum total weight of character pairs in an alignment of a against b.*

We define the *semi-local (weighted) alignment score problem* and its component (string-substring, etc.) subproblems by straightforward extension of Definition 14. The concepts of alignment dag and score matrix can be naturally generalised to the weighted case. To distinguish between the weighted and unweighted cases, we will use a script font in the corresponding notation.

The weighted alignment of strings a, b corresponds to a *weighted alignment dag* $\mathcal{G}_{a,b}$, where diagonal match edges, diagonal mismatch edges, and horizontal/vertical edges have weight w_{m}, w_{x}, w_{g}, respectively. A semi-local alignment score corresponds to a boundary-to-boundary highest-scoring path in $\mathcal{G}_{a,b}$. The complete output of the semi-local alignment score problem is a *semi-local (weighted) score matrix* $\mathcal{H}_{a,b}$. This matrix is anti-Monge; however, in contrast with the unweighted case, it is not necessarily unit-anti-Monge.

Given an arbitrary set of alignment weights, it is often convenient to normalise them so that $0 = w_{\mathsf{g}} \le w_{\mathsf{x}} < w_{\mathsf{m}} = 1$. To obtain such a normalisation, first observe that, given a pair of strings a, b, and arbitrary weights $w_{\mathsf{m}} \ge 0$, $w_{\mathsf{x}} < w_{\mathsf{m}}$, $w_{\mathsf{g}} \le \frac{1}{2} w_{\mathsf{x}}$, we can replace the weights respectively by $w_{\mathsf{m}} + 2x$, $w_{\mathsf{x}} + 2x$, $w_{\mathsf{g}} + x$, for any real x. This weight transformation increases the score of every global alignment (top-left to bottom-right) path in $\mathrm{G}_{a,b}$ by $(m+n)x$. Therefore, the relative scores of different global alignment paths do not change. In particular, the maximum global alignment score is attained by the same path as before the transformation. By taking $x = -w_{\mathsf{g}}$, and dividing the resulting weights by $w_{\mathsf{m}} - 2w_{\mathsf{g}} > 0$, we achieve the desired normalisation. (A similar method is used e.g. by Rice et al. [15]).

Definition 24. *Given original weights w_{m}, w_{x}, w_{g}, the corresponding* normalised weights *are $w^*_{\mathsf{m}} = 1$, $w^*_{\mathsf{x}} = \frac{w_{\mathsf{x}} - 2w_{\mathsf{g}}}{w_{\mathsf{m}} - 2w_{\mathsf{g}}}$, $w^*_{\mathsf{g}} = 0$. We call the corresponding alignment score the* normalised score. *The original alignment score h can be restored from the normalised score h^* by reversing the normalisation: $h = h^* \cdot (w_{\mathsf{m}} - 2w_{\mathsf{g}}) + (m+n) \cdot w_{\mathsf{g}}$.*

Thus, for fixed string lengths m and n, maximising the normalised global alignment score h^* is equivalent to maximising the original score h. However, more care is needed when maximising the alignment score across variable strings of different lengths, e.g. in the context of semi-local alignment. In such cases, an explicit conversion from normalised weights to original weights will be necessary prior to the maximisation.

Definition 25. *A set of character alignment weights will be called* rational*, if all the weights are rational numbers.*

Given a rational set of normalised weights, the semi-local alignment score problem on strings a, b can be reduced to the semi-local LCS problem by the following *blow-up* procedure. Let $w_{\mathsf{x}} = \frac{\mu}{\nu} < 1$, where μ, ν are positive natural numbers. We transform input strings a, b of lengths m, n into new *blown-up* strings $\tilde{a}$, $\tilde{b}$ of lengths $\tilde{m} = \nu m$, $\tilde{n} = \nu n$. The transformation consists in replacing every character γ in each of the strings by a substring $\$^{\mu}\gamma^{\nu-\mu}$ of length ν (recall that $\$$ is a special guard character, not present in the original strings). We have

$$\mathcal{H}_{a,b}(i,j) = \tfrac{1}{\nu} \cdot \mathrm{H}_{\tilde{a},\tilde{b}}(\nu i, \nu j)$$

for all $i \in [-m : n]$, $j \in [0 : m+n]$, where the matrix $\mathcal{H}_{a,b}$ is defined by the normalised weights on the original strings a, b, and the matrix $\mathrm{H}_{\tilde{a},\tilde{b}}$ by the LCS weights on the blown-up strings $\tilde{a}$, $\tilde{b}$. Therefore, all the techniques of the previous chapters apply

to the rational-weighted semi-local alignment score problem, assuming that ν is a constant.

An important special case of weighted string alignment is the *edit distance problem*. Here, the characters are assumed to match "by default": $w_{\mathsf{m}} = 0$. The mismatches and gaps are penalised: $2w_{\mathsf{g}} \leq w_{\mathsf{x}} < 0$. The resulting score is always nonpositive. Equivalently, we regard string a as being transformed into string b by a sequence of weighted *character edits*:

- character insertion or deletion (*indel*) has weight $-w_{\mathsf{g}} > 0$;
- character *substitution* has weight $-w_{\mathsf{x}} > 0$.

Definition 26. *The* (weighted) edit distance *between strings a, b is the minimum total weight of a sequence of character edits transforming a into b. Equivalently, it is the (nonnegative) absolute value of the corresponding (nonpositive) alignment score.*

In the rest of this work, the edit distance problem will be treated as a special case of the weighted alignment problem. In particular, all the techniques of the previous sections apply to the semi-local edit distance problem, as long as the character edit weights are rational.

3 Threshold approximate matching in compressed strings

Given a matrix A and a threshold h, it will be convenient to denote the subset of entries above the threshold by $\tau_h(A) = \big\{(i,j), \text{ such that } A(i,j) \geq h\big\}$. The threshold approximate matching problem (Definition 2) corresponds to all points in the set $\tau_h(\mathcal{H}^{\blacksquare}_{p,t})$.

Using the techniques of the previous sections, we now show how the threshold approximate matching problem on a GC-text can be solved more efficiently, assuming a sufficiently high value of the edit distance threshold k. The algorithm extends Algorithms 1 and 2 of [21], and assumes an arbitrary rational-weighted alignment score. As in Algorithm 2 of [21], we assume for simplicity the constant-time index arithmetic, keeping the index remapping implicit. The details of index remapping used to lift this assumption can be found in Algorithm 1 of [21].

Algorithm 1 (Threshold approximate matching).
Parameters: character alignment weights w_{m}, w_{x}, w_{g}, assumed to be constant rationals.
Input: plain pattern string p of length m; SLP of length $\bar{n}$, generating text string t of length n; score threshold h.
Output: locations (or count) of matching substrings in t.
Description.
First phase. Recursion on the input SLP generating t.

To reduce the problem to an unweighted LCS score, we apply the blow-up technique described in Subsection 2.5. Consider the normalised weights (Definition 24), and define the corresponding blown-up strings $\tilde{p}$, $\tilde{t}$ of length $\tilde{m} = \nu m$, $\tilde{n} = \nu n$, respectively.
Recursion base: $n = \bar{n} = 1$, $\tilde{n} = \nu$. The extended substring-string seaweed matrix $\mathrm{P}^{\blacksquare}_{\tilde{p},\tilde{t}}$ can be computed by the seaweed algorithm [20,17] in ν linear sweeps of string $\tilde{p}$. This matrix can be used to query the LCS score $\mathrm{H}_{\tilde{p},\tilde{t}}(0,\nu)$ between $\tilde{p}$ and $\tilde{t}$. String t is matching, if and only if the corresponding weighted alignment score $\mathcal{H}_{p,t}(0,1)$ is at least h.

Recursive step: $n \geq \bar{n} > 1$, $\tilde{n} = \nu n$. Let $t = t't''$ be the SLP statement defining string t. We have $\tilde{t} = \tilde{t}'\tilde{t}''$ for the corresponding blown-up strings.

As in Algorithm 2 of [21], we obtain recursively the extended substring-string seaweed matrix $\mathrm{P}^{\blacksquare}_{\tilde{p},\tilde{t}}$ and the cross-semi-local seaweed matrix $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$ by an application of the fast algorithm for distance multiplication of unit-Monge matrices [16]. These two subpermutation matrices are typically very sparse: $\mathrm{P}^{\blacksquare}_{\tilde{p},\tilde{t}}$ contains at most $2\tilde{m} = 2\nu m$ nonzeros, and $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$ exactly $\tilde{m} = \nu m$ nonzeros.

In contrast to Algorithm 2 of [21], it is no longer sufficient to consider just the $\gtrdot$-maximal nonzeros of $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$; we now have to consider all its $\tilde{m}$ nonzeros. Let us denote the indices of these nonzeros, in increasing order independently for each dimension, by

$$\hat{\imath}_{0^+} < \hat{\imath}_{1^+} < \cdots < \hat{\imath}_{\tilde{m}^-} \qquad \hat{\jmath}_{0^+} < \hat{\jmath}_{1^+} < \cdots < \hat{\jmath}_{\tilde{m}^-} \tag{1}$$

These two index sequences define an $\tilde{m} \times \tilde{m}$ non-contiguous permutation submatrix of $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$:

$$P(\hat{s}, \hat{t}) = \mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}(\hat{\imath}_{\hat{s}}, \hat{\jmath}_{\hat{t}}) \tag{2}$$

for all $\hat{s}, \hat{t} \in \langle 0 : \tilde{m} \rangle$.

Index sequence $\hat{\imath}_{\hat{s}}$ (respectively, $\hat{\jmath}_{\hat{t}}$) partitions the range $[-\tilde{m} : \tilde{n}']$ (respectively, $[\tilde{n}' : \tilde{m} + \tilde{n}]$) into $\tilde{m}+1$ disjoint non-empty intervals of varying lengths. Therefore, we have a partitioning of the cross-semi-local score matrix $\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}$ into $(\tilde{m}+1)^2$ disjoint non-empty rectangular *H-blocks* of varying dimensions. Consider an arbitrary H-block

$$\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}\big[\hat{\imath}^+_{u^-} : \hat{\imath}^-_{u^+} \mid \hat{\jmath}^+_{v^-} : \hat{\jmath}^-_{v^+}\big] \tag{3}$$

where $u, v \in [0 : \tilde{m}]$. For the boundary H-blocks, some of the above bounds are not defined. In these cases, we let $\hat{\imath}_{0^-} = -\tilde{m}$, $\hat{\imath}_{\tilde{m}^+} = \tilde{n}'$, $\hat{\jmath}_{0^-} = \tilde{n}'$, $\hat{\jmath}_{\tilde{m}^+} = \tilde{m} + \tilde{n}$.

Since the boundaries of an H-block (3) are given by adjacent pairs of indices from (1), we have $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}(\hat{\imath}, \hat{\jmath}) = 0$ for all $\hat{\imath} \in \big\langle \hat{\imath}^+_{u^-} : \hat{\imath}^-_{u^+} \big\rangle$, $u \in [0 : \tilde{m}]$ and arbitrary $\hat{\jmath}$, as well as for arbitrary $\hat{\imath}$ and all $\hat{\jmath} \in \big\langle \hat{\jmath}^+_{v^-} : \hat{\jmath}^-_{v^+} \big\rangle$, $v \in [0 : \tilde{m}]$. Therefore, given a fixed H-block, all its points are $\gtrdot$-dominated by some fixed set of nonzeros in $\mathrm{P}_{\tilde{p};\tilde{t}',\tilde{t}''}$ (and hence also in $\mathrm{P}_{\tilde{p},\tilde{t}}$). The number of nonzeros in this set is

$$d = \mathrm{P}^{T\Sigma T}_{\tilde{p};\tilde{t}',\tilde{t}''}(\hat{\imath}^-_{u^+}, \hat{\jmath}^+_{v^-}) = \mathrm{P}^{T\Sigma T}_{\tilde{p},\tilde{t}}(\hat{\imath}^-_{u^+}, \hat{\jmath}^+_{v^-}) \tag{4}$$

where the H-block's bottom-left ($\gtrdot$-minimal) point $(\hat{\imath}^-_{u^+}, \hat{\jmath}^+_{v^-})$ is chosen arbitrarily to be its reference point. Since the value of d is constant across the H-block, all its entries have identical value: we have

$$\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}(i, j) = \tilde{m} - d \tag{5}$$

for all $i \in [\hat{\imath}^+_{u^-} : \hat{\imath}^-_{u^+}]$, $j \in [\hat{\jmath}^+_{v^-} : \hat{\jmath}^-_{v^+}]$.

We now switch our focus from the blown-up strings $\tilde{p}$, $\tilde{t}'$, $\tilde{t}''$ back to the original strings p, t', t''. The partitioning of the LCS score matrix $\mathrm{H}_{\tilde{p};\tilde{t}',\tilde{t}''}$ into H-blocks induces a partitioning of the alignment score matrix $\mathcal{H}_{p;t',t''}$ into $(\tilde{m}+1)^2$ disjoint rectangular *$\mathcal{H}$-blocks* of varying dimensions. The $\mathcal{H}$-block corresponding to H-block (3) is

$$\mathcal{H}_{p;t',t''}\big[\hat{\imath}^{(+)}_{u^-} : \hat{\imath}^{(-)}_{u^+} \mid \hat{\jmath}^{(+)}_{v^-} : \hat{\jmath}^{(-)}_{v^+}\big] \tag{6}$$

where we denote $\hat{\imath}^{(-)} = \left\lfloor \frac{1}{\nu}\hat{\imath} \right\rfloor$, $\hat{\imath}^{(+)} = \left\lceil \frac{1}{\nu}\hat{\imath} \right\rceil$, for any $\hat{\imath}$. Note that, although an H-block (3) is by definition non-empty, the corresponding $\mathcal{H}$-block (6) may be empty: we have $\hat{\imath}^{(+)}_{u^-} > \hat{\imath}^{(-)}_{u^+}$ (respectively, $\hat{\jmath}^{(+)}_{v^-} > \hat{\jmath}^{(-)}_{v^+}$), if the interval $\left[\hat{\imath}^{+}_{u^-} : \hat{\imath}^{-}_{u^+}\right]$ (respectively, $\left[\hat{\jmath}^{+}_{v^-} : \hat{\jmath}^{-}_{v^+}\right]$) contains no multiples of ν.

Although all entries within an H-block (3) are constant, the entries within the corresponding $\mathcal{H}$-block (6) will typically vary. By (5) and Definition 24, we have

$$\mathcal{H}_{p;t',t''}(i,j) = \tfrac{\tilde{m}-d}{\nu} \cdot (w_{\mathsf{m}} - 2w_{\mathsf{g}}) + (m + j - i) \cdot w_{\mathsf{g}} \tag{7}$$

where $i \in \left[\hat{\imath}^{(+)}_{u^-} : \hat{\imath}^{(-)}_{u^+}\right]$, $j \in \left[\hat{\jmath}^{(+)}_{v^-} : \hat{\jmath}^{(-)}_{v^+}\right]$. Recall that $w_{\mathsf{g}} \leq 0$. Therefore, the score within an $\mathcal{H}$-block is maximised when $j - i$ is minimised, so the maximum score is attained by the block's bottom-left (i.e. $\gtrless$-minimal) entry $\mathcal{H}_{p;t',t''}\big(\hat{\imath}^{(-)}_{u^+}, \hat{\jmath}^{(+)}_{v^-}\big)$. If $w_{\mathsf{g}} < 0$, then this maximum is strict; otherwise, we have $w_{\mathsf{g}} = 0$, and all the entries in the $\mathcal{H}$-block have an identical value $\frac{\tilde{m}-d}{\nu} \cdot w_{\mathsf{m}}$.

Without loss of generality, let us now assume that all the $\mathcal{H}$-blocks (6) are non-empty. We are interested in the bottom-left entries attaining block maxima, taken across all the $\mathcal{H}$-blocks. The leftmost column and the bottom row of these entries (respectively $\mathcal{H}_{p;t',t''}\big(\hat{\imath}^{(-)}_{u^+}, n'\big)$ and $\mathcal{H}_{p;t',t''}\big(n', \hat{\jmath}^{(+)}_{v^-}\big)$ for all u, v) lie on the boundary of matrix $\mathcal{H}_{p;t',t''}$, and correspond to comparing p against respectively suffixes of t' and prefixes of t'', rather than cross-substrings of t. After excluding such boundary entries, the remaining block maxima form an $\tilde{m} \times \tilde{m}$ non-contiguous submatrix $H(u,v) = \mathcal{H}_{p;t',t''}\big(\hat{\imath}^{(-)}_{u^+}, \hat{\jmath}^{(+)}_{v^-}\big)$, where $u \in [0 : \tilde{m} - 1]$, $v \in [1 : \tilde{m}]$.

Consider the string-substring submatrix of H (i.e. the submatrix of entries that correspond to the comparison of a string against a cross-substring, as opposed to a prefix against a cross-suffix, or a suffix against a cross-prefix): $H^{\blacksquare} = \big(H(u,v) : \big(\hat{\imath}^{(-)}_{u^+}, \hat{\jmath}^{(+)}_{v^-}\big) \in [0 : n' \mid n' : n]\big)$. Since matrix $\mathcal{H}_{p;t',t''}$ is anti-Monge, its submatrices H and $H^{\blacksquare}$ are also anti-Monge.

We now need to obtain the row maxima of matrix $H^{\blacksquare}$. Let

$$N(u,v) = \frac{\nu}{2w_{\mathsf{g}} - w_{\mathsf{m}}} H^{\blacksquare}(u,v) = P^{T\Sigma T}(u,v) - \tilde{m} + \frac{\nu\big(m + \hat{\jmath}^{(+)}_{v^-} - \hat{\imath}^{(-)}_{u^+}\big)w_{\mathsf{g}}}{2w_{\mathsf{g}} - w_{\mathsf{m}}}$$

(By (7), (4), (2))

Since $2w_{\mathsf{g}} - w_{\mathsf{m}} < 0$, the problem of finding row maxima of $H^{\blacksquare}$ is equivalent to finding row minima of matrix N, or, equivalently, column minima of the transpose matrix N^T. This matrix (and therefore N itself) is subunit-Monge: we have $N^T(v,u) = N(u,v) = P^{T\Sigma}(v,u) + b(u) + c(v)$, where $b(u) = -\frac{\nu\hat{\imath}^{(-)}_{u^+} \cdot w_{\mathsf{g}}}{2w_{\mathsf{g}} - w_{\mathsf{m}}}$, $c(v) = -\tilde{m} + \frac{\nu\big(m + \hat{\jmath}^{(+)}_{v^-}\big)w_{\mathsf{g}}}{2w_{\mathsf{g}} - w_{\mathsf{m}}}$. Therefore, the column minima of N^T can be found by Lemma 12 (replacing row minima with column minima by symmetry).

The set $\tau_h(H^{\blacksquare})$ of all entries in $H^{\blacksquare}$ scoring above the threshold h, and therefore the set of all matching cross-substrings of a, can now be obtained by a local search in the neighbourhoods of the row maxima. (End of recursive step)

Second phase. For every SLP symbol, we now have the relative locations of its minimally matching cross-substrings. It is now straightforward to obtain their absolute locations and/or their count (as in Algorithm 2 of [21], substituting "matching" for "minimally matching").

Cost analysis.

First phase. Each seaweed matrix multiplication runs in time $O(\tilde{m} \log \tilde{m}) = O(m \log m)$. The algorithm of Lemma 12 runs in time $O(\tilde{m} \log\log \tilde{m}) = O(m \log\log m)$.

Hence, the running time of a recursive step is $O(m \log m)$. There are $\bar{n}$ recursive steps in total, therefore the whole recursion runs in time $O(m \log m \cdot \bar{n})$.
Second phase. For every SLP symbol, there are at most $m - 1$ minimally matching cross-substrings. Given the output of the first phase, the absolute locations of all minimally matching substrings in t can be reported in time $O(m\bar{n} + output)$.
Total. The overall running time is $O(m \log m \cdot \bar{n} + output)$.

Algorithm 1 improves on the algorithm of [11], as long as $k = \omega\big((\log m)^{1/2}\big)$ in the case of general GC-compression, and $k = \omega(\log m)$ in the case of LZ78 or LZW compression. Algorithm 1 also improves on the algorithms of [4,5], as long as $k = \omega\big((m \log m)^{1/4}\big)$, in the case of both general GC-compression and LZ78 or LZW compression.

4 Conclusions

We have obtained a new efficient algorithm for threshold approximate matching between a GC-text and a plain pattern. Our algorithm is of interest not only in its own right, but also as a natural application of fast unit-Monge matrix multiplication, developed in our previous works. We have also demonstrated a new technique for incremental searching in an implicit totally monotone matrix, which which we believe to be of independent interest.

References

1. A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber: *Geometric applications of a matrix-searching algorithm.* Algorithmica, 2(1) 1987, pp. 195–208.
2. A. Apostolico and C. Guerra: *The longest common subsequence problem revisited.* Algorithmica, 2(1) 1987, pp. 315–336.
3. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in Proceedings of the 7th SPIRE, 2000, pp. 39–48.
4. P. Bille, R. Fagerberg, and I. L. Gørtz: *Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts.* ACM Transactions on Algorithms, 6(1) 2009, pp. 3:1–3:14.
5. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann: *Random access to grammar-compressed strings*, in Proceedings of the 22nd ACM-SIAM SODA, 2011, pp. 373–389.
6. P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich: *Window subsequence problems for compressed texts*, in Proceedings of CSR, vol. 3967 of Lecture Notes in Computer Science, 2006, pp. 127–136.
7. K.-M. Chao and L. Zhang: *Sequence Comparison: Theory and Methods*, vol. 7 of Computational Biology Series, Springer, 2009.
8. R. Cole and R. Hariharan: *Approximate string matching: A simpler faster algorithm.* SIAM Journal on Computing, 31(6) 2002, pp. 1761–1782.
9. P. Gawrychowski: *Faster algorithm for computing the edit distance between SLP-compressed strings*, in Proceedings of SPIRE, vol. 7608 of Lecture Notes in Computer Science, 2012, pp. 229–236.
10. D. Hermelin, G. M. Landau, S. Landau, and O. Weimann: *A unified algorithm for accelerating edit-distance computation via text-compression*, in Proceedings of the 26th STACS, 2009, pp. 529–540.
11. J. Kärkkäinen, G. Navarro, and E. Ukkonen: *Approximate string matching on Ziv–Lempel compressed text.* Journal of Discrete Algorithms, 1 2003, pp. 313–338.
12. G. M. Landau and U. Vishkin: *Fast parallel and serial approximate string matching.* Journal of Algorithms, 10(2) 1989, pp. 157–169.

13. V. Levenshtein: *Binary codes capable of correcting spurious insertions and deletions of ones.* Problems of Information Transmission, 1 1965, pp. 8–17.
14. M. Lohrey: *Algorithmics on SLP-compressed strings: a survey.* Groups Complexity Cryptology, 4(2) 2012, pp. 241–299.
15. S. V. Rice, H. Bunke, and T. A. Nartker: *Classes of cost functions for string edit distance.* Algorithmica, 18 1997, pp. 271–280.
16. A. Tiskin: *Fast distance multiplication of unit-Monge matrices.* Algorithmica, To appear.
17. A. Tiskin: *Semi-local string comparison: Algorithmic techniques and applications*, Tech. Rep. 0707.3619, arXiv.
18. A. Tiskin: *Semi-local longest common subsequences in subquadratic time.* Journal of Discrete Algorithms, 6(4) 2008, pp. 570–581.
19. A. Tiskin: *Semi-local string comparison: Algorithmic techniques and applications.* Mathematics in Computer Science, 1(4) 2008, pp. 571–603.
20. A. Tiskin: *Periodic string comparison*, in Proceedings of CPM, vol. 5577 of Lecture Notes in Computer Science, 2009, pp. 193–206.
21. A. Tiskin: *Towards approximate matching in compressed strings: Local subsequence recognition*, in Proceedings of CSR, vol. 6651 of Lecture Notes in Computer Science, 2011, pp. 401–414.
22. T. A. Welch: *A technique for high-performance data compression.* Computer, 17(6) 1984, pp. 8–19.
23. T. Yamamoto, H. Bannai, S. Inenaga, and M. Takeda: *Faster subsequence and don't-care pattern matching on compressed texts*, in Proceedings of CPM, vol. 6661 of Lecture Notes in Computer Science, 2011, pp. 309–322.
24. G. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.

Metric Preserving Dense SIFT Compression

Shmuel T. Klein[1] and Dana Shapira[2,3]

[1] Computer Science Department, Bar Ilan University, Israel
[2] Computer Science Department, Ashkelon Academic College, Israel
[3] Department of Computer Science and Mathematics, Ariel University, Israel
tomi@cs.biu.ac.il, shapird@gmail.com

Abstract. The problem of compressing a large collection of feature vectors so that object identification can further be processed on the compressed form of the features is investigated. The idea is to perform matching against a query image in the compressed form of the feature descriptor vectors retaining the metric. Given two SIFT feature vectors, in previous work we suggested to compress them using a lossless encoding for which the pairwise matching can be done directly on the compressed files, by means of a Fibonacci code. In this paper we extend our work to Dense SIFT and in particular to PHOW features, that contain, for each image, about 300 times as many vectors as the original SIFT.

1 Introduction

The tremendous storage requirements and ever increasing resolutions of digital images, necessitate automated analysis and compression tools for information processing and extraction. There are several methods for transforming an image into a set of feature vectors, such as SIFT (Scale Invariant Feature Transform) by Lowe [13], GLOH (Gradient Location and Orientation Histogram) [14], and SURF (Speed-up-Robust Features) [1], to mention only a few. Ideally, such descriptors are invariant to scaling, rotation, illumination changes and local geometric distortion.

The main idea is to carefully choose a subset of the features so that this reduced set will be representative of the original image and will be processed instead. Obviously, there are applications in which working on a dense set of features, rather than the sparse subsets mentioned above, is much better, since a larger set of local image descriptors provides more information than the corresponding descriptors evaluated only at selected interest points. In this paper we suggest that instead of choosing a representative set of interest points, possibly reducing the object detection accuracy, one can apply metric preserving compression methods so that a larger number of key points can be processed using the same amount of memory storage.

SIFT vectors are computed for the extracted key points of objects from a set of reference images, which are then stored in a database. An object in a new image is identified after matching its features against this database using the Euclidean L_2 distance. In the case of object category or scene classification, experimental evaluations show that better classification results are often obtained by computing the so-called Dense SIFT descriptors (or DSIFT for short) as opposed to SIFT on a sparse set of interest points [3]. The dense sets may contain about 300 times more vectors than the sparse sets.

Query feature compression can contribute to faster retrieval, for example, when the query data is transmitted over a network, as in the case when mobile visual applications are used for identifying products in comparison shopping. Moreover,

Shmuel T. Klein, Dana Shapira: Metric Preserving Dense SIFT Compression, pp. 139–147.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

since the memory space on the mobile device is restricted, working directly on the compressed form of the data is sometimes required.

A feature descriptor encoder is presented in Chandrasekhar et al. [7]. They transfer the compressed features over the network and *decompress* them once data is received for further pairwise image matching. Chen et al. [8] perform tree-based retrieval, using a scalable vocabulary tree. Since the tree histogram suffices for accurate classification, the histogram is transmitted instead of individual feature descriptors. Also Chandrasekhar et al. [5] encode a set of feature descriptors jointly and use tree-based retrieval when the order in which data is transmitted does not matter, as in our case. Several other SIFT feature vector compressors were proposed, and we refer the reader to [4] for a comprehensive survey.

We propose a special encoding which is not only compact in its representation, but can also be processed directly *without* any decompression. That is, unlike traditional feature vectors compression which decompresses before applying pairwise matching, the current suggestion omits the decompression stage, and performs pairwise matching directly on the compressed data. Similar work, using quantization, has been suggested by Chandrasekhar et al. [6]. We do not apply quantization, but rather use a lossless encoding.

Working on a shorter representation and saving the decompression process may save processing time, as well as memory storage. By using a lossless compression and applying the same norm for performing the pairwise matching we make sure not to hurt the true positives and false negatives probabilities. Moreover, representing the same set of feature descriptors in less space can allow us to keep a larger set of representatives, which can result in a higher probability for object identification by reducing the number of mismatches.

The main idea is to perform the matching against the query image in the compressed form of the feature descriptor vectors so that the metric is retained, i.e., vectors are close in the original distance (e.g., Euclidean distance based on nearest neighbors according to the Best-Bin-First-Search algorithm in SIFT [2]) if and only if they are close in their compressed counterparts. This can be done either by using the same metric but requiring that the compression does not affect the metric, or by changing the distance so that the number of false matches and true mismatches does not increase under this new distance. In the present work, we stick to the first alternative and do not change the L_2 metric used in SIFT.

For the formal description of the general case, let $\{\boldsymbol{f_1}, \boldsymbol{f_2}, \ldots, \boldsymbol{f_n}\}$ be a set of feature descriptor vectors generated using some feature based object detector, and let $\| \quad \|_M$ be a metric associated with the pairwise matching of this object detector. The *Compressed Feature Based Matching Problem* (CFBM) is to find a compression encoding of the vectors, denoted $\mathcal{E}(\boldsymbol{f_i})$, and an equivalent metric m so that for every $\varepsilon > 0$ there exists a $\delta > 0$ such that $\forall i, j \in \{1, \ldots, n\}$

$$\|\boldsymbol{f_i} - \boldsymbol{f_j}\|_M < \varepsilon \iff \|\mathcal{E}(\boldsymbol{f_i}) - \mathcal{E}(\boldsymbol{f_j})\|_m < \delta.$$

The rest of the paper is organized as follows. Section 2 gives a description of our lossless encoding for DSIFT feature vectors; Section 3 suggests improving the compression for PHOW (Pyramid Histogram Of visual Words) vectors [3]; Section 4 presents the algorithm used for compressed pairwise matching the feature vectors without decompression; Section 5 presents results on the compression performance of our lossless encoding for DSIFT descriptors and PHOW features, and the last section concludes.

2 Lossless Encoding for DSIFT

Given two SIFT feature vectors, we suggest in [12] achieving our goal to compress them using a lossless encoding so that the pairwise matching can be done directly on the compressed form of the file, by means of a *Fibonacci Code*. It turns out that the Fibonacci Code is also suitable for DSIFT and PHOW feature vectors.

The Fibonacci code is a universal variable length encoding of the integers based on the Fibonacci sequence rather than on powers of 2. A subset of these encodings can be used as a fixed alternative to Huffman codes, giving obviously less compression, but adding simplicity (there is no need to generate a new code every time), robustness and speed [9,10]. The particular property of the binary Fibonacci encoding is that there are no adjacent 1's, so that the string 11 can act like a *comma* between codewords. More precisely, the codeword set consists of all the binary strings for which the substring 11 appears exactly once, at the left end of the string.

The connection to the Fibonacci sequence can be seen as follows: just as any integer k has a standard binary representation, that is, it can be uniquely represented as a sum of powers of 2, $k = \sum_{i\geq 0} b_i 2^i$, with $b_i \in \{0,1\}$, there is another possible binary representation based on Fibonacci numbers, $k = \sum_{i\geq 0} f_i F(i)$, with $f_i \in \{0,1\}$, where it is convenient to define the Fibonacci sequence here by

$$F(0) = 1, F(1) = 2;\ F(i) = F(i-1) + F(i-2) \text{ for } i \geq 2.$$

This Fibonacci representation will be unique if, when encoding an integer, one repeatedly tries to fit in the largest possible Fibonacci number. For example, the largest Fibonacci number fitting into 19 is 13, for the remainder 6 one can use the Fibonacci number 5, and the remainder 1 is a Fibonacci number itself. So one would represent 19 as $19 = 13 + 5 + 1$, yielding the binary string 101001. Note that the bit positions correspond to $F(i)$ for increasing values of i from right to left, just as for the standard binary representation, in which $19 = 16 + 2 + 1$ would be represented by 10011. Each such Fibonacci representation starts with a 1, so by preceding it with an additional 1, one gets a sequence of uniquely decipherable codewords.

Decoding, however, would not be instantaneous, because the set lacks the prefix property. For example, a first attempt to start the parsing of the encoded string 1101111111110 by 110 11 11 11 11 would fail, because the remaining suffix 10 is not the prefix of any codeword. So only after having read 5 codewords in this case (and the example can obviously be extended) would one know that the correct parsing is 1101 11 11 11 110. To overcome this problem, the Fibonacci code defined in [9] simply reverses each of the codewords. The adjacent 1s are then at the right instead of at the left end of each codeword, thus yielding the prefix code {11, 011, 0011, 1011, 00011, 10011, 01011, 000011, 100011, 010011, 001011, 101011, 0000011, ...}.

A disadvantage of this reversing process is that the order preserving characteristic of the previous representation is lost, e.g., the codewords corresponding to 17 and 19 are 1010011 and 1001011, but if we compare them as if they were standard binary representations of integers, the first, with value 83, is larger than the second, with value 75. At first sight, this seems to be critical, because we want to compare numbers in order to subtract the smaller from the larger. But in fact, since we calculate the L_2 norm, the *square* of the differences of the coordinates is needed. It therefore does not matter if we calculate $x - y$ or $y - x$, and there is no problem dealing with negative numbers. The reversed representation can therefore be kept.

We wish to encode DSIFT and PHOW feature vectors, each consisting of exactly 128 coordinates. Thus, in addition to the ability of parsing an encoded feature vector into its constituting coordinates, separating adjacent vectors could simply be done by counting the number of codewords, which is easily done with a prefix code.

Empirically, DSIFT and PHOW vectors are characterized by having smaller integers appear with higher probability. To illustrate this, we considered the Lenna image (an almost standard compression benchmark) and applied *vlfeat Matlab*'s [1] DSIFT and PHOW applications on it, generating 253,009 and 237,182 feature vectors, respectively, of 128 coordinates each. The numbers (thousands of occurrences for values from 2 to 255) are plotted in Figure 1.

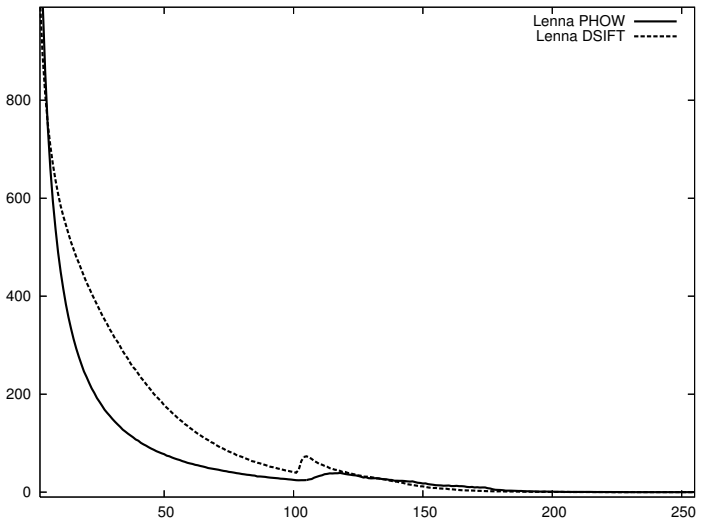


Figure 1. Value distribution in feature vectors.

The usual approach for using a universal code, such as the Fibonacci code, is first sorting the probabilities of the source alphabet symbols in decreasing order and then assigning the universal codewords by increasing codeword lengths, so that high probability symbols are given the shorter codewords. In our case, in order to be able to perform compressed pairwise matching, we omit sorting the probabilities, as already suggested in [11] for Huffman coding. Figure 1 shows that the order is not strictly monotonic, but that the fluctuations are small. Indeed, experimental results show that encoding the numbers themselves instead of their indices has hardly any influence (less than 0.4% on our test images).

3 Further compression of PHOW

DSIFT feature vectors contain repeated zero-runs, as could be expected by the high number of zeros. However, we noticed a difference between the zero-runs of DSIFT and PHOW feature vectors, leading to the idea of representing a *pair* of adjacent 0s by a single codeword. That is, the pair 00 is assigned the first Fibonacci codeword 11, a single 0 is encoded by the second codeword 011, and generally, the integer k is represented by the Fibonacci codeword corresponding to the integer $k+2$, for $k \geq 0$.

[1] `http://www.vlfeat.org/`

This special codeword assignment was empirically shown to be useful for PHOW but not for general DSIFT.

As example, consider the 25th PHOW feature vector of Lenna's Image, consisting of 128 coordinates, in which the first 20 are

$$8, 19, 3, 1, 5, 7, 0, 0, 0, 0, 1, 1, 32, 60, 0, 0, 0, 0, 0, 0, \ldots$$

Instead of encoding this vector as

$$100011 \quad 0101011 \quad 1011 \quad 011 \quad 10011 \quad 000011 \quad 11 \quad 11 \quad 11 \quad 11$$
$$011 \quad 011 \quad 00101011 \quad 1001000011 \quad 11 \quad 11 \quad 11 \quad 11 \quad 11 \quad 11$$

as we would do for general DSIFT, we rather encode it as

$$010011 \quad 00000011 \quad 00011 \quad 0011 \quad 01011 \quad 100011 \quad 11 \quad 11$$
$$0011 \quad 0011 \quad 000000011 \quad 0101000011 \quad 11 \quad 11 \quad 11$$

reducing 75 bits to 71, rather than 160 bits for the first 20 elements of the original uncompressed PHOW vector using one byte per integer.

Note that since all numbers are simply shifted by 1 for the case of DSIFT and by 2 for the case of PHOW, the difference between two Fibonacci encodings is preserved, which is an essential property for computing their distance in the compressed form.

4 Compressed pairwise matching

The algorithm for computing the subtraction of two Fibonacci encoded coordinates was presented in [12] and is given here for the sake of completeness. We start with a general algorithm, Sub( ), for subtraction which is used in both DSIFT and PHOW L_2 norm computations. Given two encoded descriptors, one needs to compute their L_2 norm. Each component is first subtracted from the corresponding component, then the squares of these differences are summed. The algorithm for computing the subtraction of two corresponding Fibonacci encoded coordinates A and B is given in Figure 2. We start by stripping the trailing 1s from both, and pad, if necessary, the shorter codeword with zeros at its right end so that both representations are of equal length. Note that the term first, second and next refer to the order from right to left.

At the end of the while loop, there are 2 unread bits left in B, which can be 00, 10 or 01, with values 0, 1 or 2 in the Fibonacci representation, but when read as standard binary numbers, the values are 0, 2 and 1. This is corrected in the commands after the while loop of the algorithm. The evaluation relies on the fact that a 1 in position i of the Fibonacci representation is equivalent to, and can thus be replaced by, 1s in positions $i + 1$ and $i + 2$. This allows us to iteratively process the subtraction, independently of the Fibonacci number corresponding to the leading bits of the given numbers. Processing is, therefore, done in time proportional to the size of the compressed file, without any decoding.

To calculate the L_2 norm, the two Fibonacci encoded input vectors have to be scanned in parallel from left to right. In each iteration, the first codeword (identified as the shortest prefix ending in 11) is removed from each of the two input vectors, and each pair of coordinates is processed according to the procedure Sub(A, B) above. As opposed to the computation of the DSIFT L_2 norm, which simply sums the squares

```
Sub(A, B)
scan the bits of A and B from right to left
a_1 ⟵ first bit of A
a_2 ⟵ second bit of A
while next bit of A exists {
    a_3 ⟵ next bit of A
    b_1 ⟵ next bit of B
    a_1 ⟵ a_1 − b_1
    a_2 ⟵ a_1 + a_2
    a_3 ⟵ a_1 + a_3
    a_1 ⟵ a_2
    a_2 ⟵ a_3
}
b ⟵ value of last 2 bits of B
if b ≠ 0 then b ⟵ 2 − b
return 2 ∗ a_1 + a_2 − b
```

Figure 2. Compressed differencing of DSIFT and PHOW encoded coordinates.

of the differences of two corresponding Fibonacci encoded coordinates, the PHOW encoding works as follows. The codeword 11, representing two consecutive zeros, needs a special treatment only if the other codeword, say B, is not 11. In this case, 11 should be replaced by two codewords 011, each representing a single zero. We thus perform Sub(B, 011), and then concatenate the second 011 in front of the remaining input vector, to be processed in the following iteration. The details appear in Figure 3, where $\|$ denotes concatenation and the variable SSQ, accumulating the sum of the squares of the differences, is initialized to 0. At each iteration, the result, S, of the subtraction of the two given Fibonacci encoded numbers is computed by the function Sub( ); it is then squared and added to the accumulated value SSQ. By definition, the L_2 norm is the square root of the sum of the squares.

5 Compression performance

We considered three images for our experiments : *Lenna*, *Peppers* and *House*, which were taken from the SIPI (Signal and Image Processing Institute) Image Data Base[2]. We applied DSIFT and PHOW on all images getting for each 253,009 and 237,182 feature vectors, respectively. For comparison, the number of interest points on which the original SIFT features were generated for the three test images was 737, 832 and 991, respectively. Table 1 presents the compression performance of our Fibonacci encoding applied on the coordinates of the DSIFT vectors suitable for compressed matching as compared to other compressors. The second column shows the original sizes in MB. The other figures are compression ratios, defined as the original size divided by the compressed size, so that larger numbers indicate better compression. The third column presents the compression performance when each number is represented by its Fibonacci encoding. To evaluate the compression loss due to omitting the sorting of

[2] `http://sipi.usc.edu/database/`

```
PHOWL2Norm(V1, V2)
SSQ ⟵ 0
while V1 and V2 are not empty {
    remove first codeword from V1
            and assign it to A
    remove first codeword from V2
            and assign it to B
    if A ≠ B then
        if A = 11 then
            S ⟵ Sub(B, 011)
            V1 ⟵ 011 || V1
        else if B = 11 then
            S ⟵ Sub(A, 011)
            V2 ⟵ 011 || V2
        else S ⟵ Sub(A, B)
        SSQ ⟵ SSQ + S^2
}
return √SSQ
```

Figure 3. PHOW compressed L_2 norm computation.

the frequencies, we considered the compression where each symbol is encoded using the Fibonacci codeword assigned according to its position in the list of decreasing order of frequencies. These values appear in the fourth column headed ordered. For comparison, the compression achieved by a Huffman code is also included as an upper bound.

Image	Original size	Fibonacci	Ordered	Huffman
Lenna	83.31	3.164	3.164	3.481
Peppers	83.88	3.113	3.114	3.455
House	82.80	3.184	3.185	3.486

Table 1. Compression efficiency of the proposed encodings for DSIFT Features.

Table 2 presents the corresponding results for the PHOW vectors. The third column presents the compression performance in which each number is represented by its Fibonacci encoding using the first codeword for encoding 00. The fourth column is ordered Fibonacci as defined in Table 1. Huffman encoding is given in the fifth column. This time it was applied on the alphabet including the symbol 00.

As can be seen, encoding the numbers themselves instead of their indices induces a negligible compression loss. The high probability for small integers also reduces the gap between the performances of Fibonacci and Huffman codes.

6 Conclusion

We have dealt with the problem of compressing a set of Dense SIFT feature vectors, and in particular on DSIFT and PHOW features, under the constraint of allowing

Image	Original size	Fibonacci	Ordered Huffman	
Lenna	70.664	3.854	3.859	4.046
Peppers	70.561	3.874	3.883	4.065
House	73.017	3.640	3.646	3.917

Table 2. Compression efficiency of the proposed encodings for PHOW Features.

processing the data directly in its compressed form. Such an approach is advantageous not only to save storage space, but also to the manipulation speed, and in fact improves the whole data handling from transmission to processing.

Our solution is based on encoding the vector elements by means of a Fibonacci code, which is generally inferior to Huffman coding from the compression point of view, but has several advantages, turning it into the preferred choice in our case: (a) simplicity – the code is fixed and need not be generated anew for different distributions; (b) the possibility to identify each individual codeword – avoiding the necessity of adding separators, and not requiring a sequential scan; (c) allowing to perform subtractions using the compressed form – and thereby calculating the L_2 norm, whereas a Huffman code would have to use some translation table.

The experiments suggest that there is only a negligible loss in compression efficiency, of 1% and even less, relative to the ordered Fibonacci code, and only a small increase, of 4–10%, in the size of the compressed file relative to the size achieved by the optimal Huffman codes, which might be worth a price to pay for the improved processing.

References

1. H. Bay, T. Tuytelaars, and L. Gool: *SURF: Speeded Up Robust Features*, in European Conference on Computer Vision (ECCV), 2006, pp. 404–417.
2. J. Beis and D. G. Lowe: *Shape indexing using approximate nearest-neighbour search in high-dimensional spaces*, in Conference on Computer Vision and Pattern Recognition, 1997, pp. 1000–1006.
3. A. Bosch, A. Zisserman, and X. Munoz: *Image classification using random forests and ferns*, in Proc. 11th International Conference on Computer Vision (ICCV'07), Rio de Janeiro, Brazil, 2007, pp. 1–8.
4. V. Chandrasekhar, M. Makar, G. Takacs, D. M. Chen, S. S. Tsai, N. M. Cheung, R. Grzeszczuk, Y. A. Reznik, and B. Girod: *Survey of SIFT compression schemes*, in Proc. Int. Workshop Mobile Multimedia Processing, 2010.
5. V. Chandrasekhar, Y. A. Reznik, G. Takacs, D. M. Chen, S. S. Tsai, R. Grzeszczuk, and B. Girod: *Compressing feature sets with digital search trees*, in ICCV Workshops, 2011, pp. 32–39.
6. V. Chandrasekhar, G. Takacs, D. M. Chen, S. S. Tsai, Y. A. Reznik, R. Grzeszczuk, and B. Girod: *Compressed histogram of gradients: A low-bitrate descriptor.* International Journal of Computer Vision, 96(3) 2012, pp. 384–399.
7. V. Chandrasekhar, G. Takacs, D. M. Chen, S. S. Tsai, J. Singh, and B. Girod: *Transform coding of image feature descriptors*, in SPIE Visual Communications and Image Processing (VCIP), 2009.
8. D. M. Chen, S. S. Tsai, V. Chandrasekhar, G. Takacs, J. P. Singh, and B. Girod: *Tree histogram coding for mobile image matching*, in Data Compression Conference, DCC–09, 2009, pp. 143–152.
9. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.

10. S. T. Klein and M. Kopel Ben-Nissan: *On the usefulness of Fibonacci compression codes.* The Computer Journal, 53 2010, pp. 701–716.
11. S. T. Klein and D. Shapira: *Huffman coding with non-sorted frequencies.* Mathematics in Computer Science, 5(2) 2011, pp. 171–178.
12. S. T. Klein and D. Shapira: *Compressed SIFT feature based matching.* To appear in Proc. The Fourth International Conference on Advances in Information Mining and Management, IMMM–14, 2014.
13. D. G. Lowe: *Distinctive image features from scale-invariant keypoints.* International Journal of Computer Vision, 60 (2) 2004, pp. 91–110.
14. K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool: *A comparison of affine region detectors*, in International Journal of Computer Vision, vol. 65 (1-2), 2005, pp. 43–72.

Approximation of Greedy Algorithms for Max-ATSP, Maximal Compression, Maximal Cycle Cover, and Shortest Cyclic Cover of Strings

Bastien Cazaux and Eric Rivals*

L.I.R.M.M. Université Montpellier II, CNRS U.M.R. 5506
161 rue Ada, F-34392 Montpellier Cedex 5, France
{cazaux, rivals}@lirmm.fr

Abstract. Covering a directed graph by a Hamiltonian path or a set of words by a superstring belong to well studied optimisation problems that prove difficult to approximate. Indeed, the *Maximum Asymmetric Travelling Salesman Problem* (Max-ATSP), which asks for a Hamiltonian path of maximum weight covering a digraph, and the *Shortest Superstring Problem* (*SSP*), which, for a finite language $P := \{s_1, \ldots, s_p\}$, searches for a string of minimal length having each input word as a substring, are both Max-SNP hard. Finding a short superstring requires to choose a permutation of words and the associated overlaps to minimise the superstring length or to maximise the compression of P. Hence, a strong relation exists between Max-ATSP and *SSP* since solving Max-ATSP on the Overlap Graph for P gives a shortest superstring. Numerous works have designed algorithms that improve the approximation ratio but are increasingly complex. Often, these rely on solving the pendant problems where the cover is made of cycles instead of single path (Max-CC and SCCS). Finally, the greedy algorithm remains an attractive solution for its simplicity and ease of implementation. Its approximation ratios have been obtained by different approaches. In a seminal but complex proof, Tarhio and Ukkonen showed that it achieves 1/2 compression ratio for Max-CC. Here, using the full power of subset systems, we provide a unified approach for proving simply the approximation ratio of a greedy algorithm for these four problems. Especially, our proof for Maximal Compression shows that the Monge property suffices to derive the 1/2 tight bound.

1 Introduction

Given a set of words $P = \{s_1, \ldots, s_p\}$ over a finite alphabet, the *Shortest Superstring Problem* (*SSP*) or *Maximal Compression* (MC) problems ask for a shortest string u that contains each of the given words as a substring. It is a key problem in data compression and in bioinformatics, where it models the question of sequence assembly. Indeed, sequencing machines yield only short reads that need to be aggregated according to their overlaps to obtain the whole sequence of the target molecule [4]. Recent progress in sequencing technologies have permitted an exponential increase in throughput, making acute the need for simple and efficient assembly algorithms. Two measures can be optimised for *SSP*: either the length of the superstring is minimised, or the compression is maximised (*i.e.*, $\|S\| - |u| := \sum_{s_i \in S} |s_i| - |u|$). Unfortunately, even for a binary alphabet, *SSP* is NP-hard [3] and MAX-SNP-hard relative to both measures [2]. Among many approximation algorithms, the best known fixed ratios are $2\frac{11}{23}$ for the superstring [10] and 3/4 for the compression [11]. A famous conjecture

* This work is supported by ANR Colib'read (ANR-12-BS02-0008) and Défi MASTODONS SePhHaDe from CNRS.

Bastien Cazaux and Eric Rivals: Approximation of Greedy Algorithms for Max-ATSP, Maximal Compression, Maximal Cycle Cover, and Shortest Cyclic Cover of Strings, pp. 148–161.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

states that a simple, greedy agglomeration algorithm achieves a ratio 2 for the superstring measure, while it is known to approximate tightly MC with ratio 1/2, but the later proofs are quite complex involving many cases of overlaps [13,14]. Figure 2 and Example 7 on page 153 illustrate the difference between two optimisation measures. The best approximation algorithms use the *Shortest Cyclic Cover of Strings* (SCCS) as a procedure, which asks for a set of cyclic strings of total minimum length that collectively contain the input words as substrings. The SCCS problem can be solved in polynomial time in $\|S\|$ [12,2].

These problems on strings can be viewed as problems on the Overlap Graph, in which the input words are the nodes, and an arc represents the asymmetric maximum overlap between two words. Figure 1 on p.150 displays an example of overlap graph. Covering the Overlap Graph with either a maximum weight Hamiltonian path or a maximum weight cyclic cover gives a solution for the problems of *Maximal Compression* or of *Shortest Cyclic Cover of Strings*, respectively. This expresses the relation between the *Maximum Asymmetric Travelling Salesman Problem* (Max-ATSP) and *Maximal Compression* on one hand, as well as between *Maximum Cyclic Cover* (Max-CC) and *Shortest Cyclic Cover of Strings* on the other. Both Max-ATSP and Max-CC have been extensively studied as essential computer science problems. Table 1 presents all these problems and their greedy approximation ratios.

Type of cover	Input: directed graph			Input: set of strings		
	name	ratio	ref.	name	ratio	ref.
Hamiltonian path	*Maximum Asymmetric Travelling Salesman*	1/3 y	[5,14]	*Maximal Compression*	1/2 y	[13,14]
				Shortest Superstring	7/2	[7]
Set of cycles	*Maximum Cyclic Cover*	Poly	[12]	*Shortest Cyclic Cover of Strings*	Poly	1 [4]
		1/2 y	here			

Table 1: The approximation performance of the greedy algorithm on the five optimisation problems considered here. The input is either a directed graph or a set of strings (in columns), while the type of cover can be a Hamiltonian path or a set of cycles (in lines). For each problem, the best greedy approximation ratio, its tightness, and the bibliographic reference are shown. Highlighted in blue: the approximation bounds for which we provide a proof relying on subset systems. The bound for *Maximum Cyclic Cover* was open. "Poly" means that the problem is solvable in polynomial time. A "y" after the bound means that it is tight.

Our contributions: Subset systems were introduced recently to investigate the approximation performances of greedy algorithms in a unified framework [8]. As mentioned earlier, the ratio of greedy for the five problems considered (except Max-CC) have been shown with different proofs and using distinct combinatorial properties. With subset systems, we investigate the approximation achieved by greedy algorithms on four of these problems in a unified manner, and provide new and simple proofs the results mentioned in Table 1. After introducing the required notation and concepts, we study the case of the Max-ATSP and Max-CC problems in Section 2, then we focus on the *Maximal Compression* problem in Section 3, and state the results regarding *Shortest Cyclic Cover of Strings* in Section 3.1, before concluding.

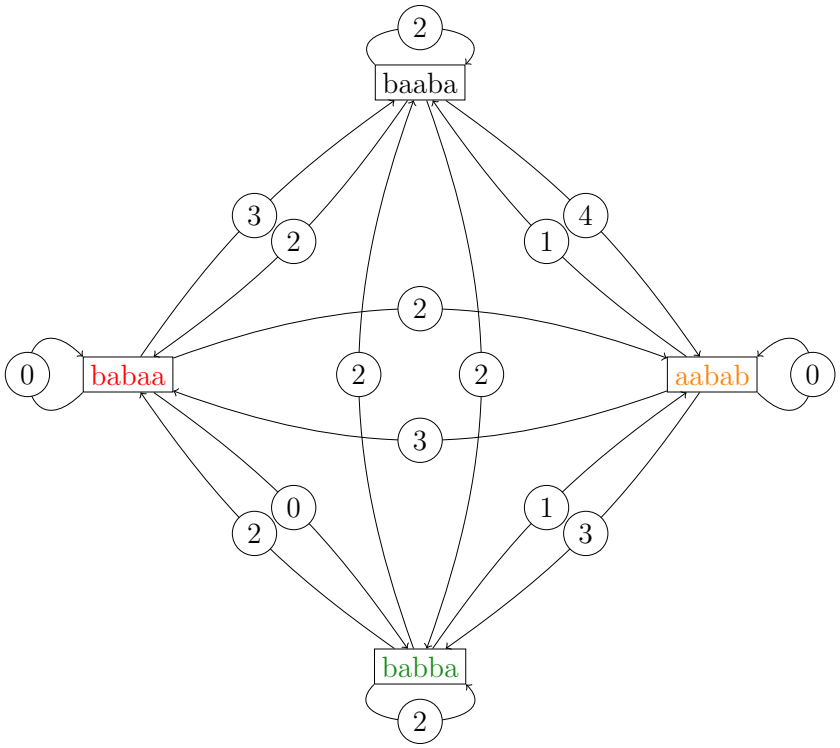


Figure 1: Example of an Overlap Graph for the input words $P := \{\text{baaba}, \text{babaa}, \text{aabab}, \text{babba}\}$.

1.1 Sets, strings, and overlaps.

We denote by $\#(\Lambda)$ the cardinality of any finite set Λ.

An *alphabet* Σ is a finite set of *letters*. A linear *word* or *string* over Σ is a finite sequence of elements of Σ. The set of all finite words over Σ is denoted by $\Sigma^\star$, and ϵ denotes the empty word. For a word x, $|x|$ denotes the *length* of x. Given two words x and y, we denote by xy the *concatenation* of x and y. For every $1 \leq i \leq j \leq |x|$, $x[i]$ denotes the i-th letter of x, and $x[i\,;\,j]$ denotes the *substring* $x[i]x[i+1]\cdots x[j]$.

A *cyclic string* or *necklace* is a finite string in which the last symbol precedes the first one. It can be viewed as a linear string written on a torus with both ends joined.

Overlaps and agglomeration Let s, t, u be three strings of $\Sigma^\star$. We denote by $ov(s,t)$ the maximum overlap from s over t; let $pr(s,t)$ be the prefix of s such that $s = pr(s,t)\,.\,ov(s,t)$, then we denote the *agglomeration of s over t* by $s \oplus t := pr(s,t)t$. Note that neither the overlap nor the agglomeration are symmetrical. Clearly, one has $(s \oplus t) \oplus (t \oplus u) = (s \oplus t) \oplus u$.

Example 1. Let $P := \{abbaa, baabb, aabba\}$. One has $ov(abbaa, baabb) = baa$ and $abbaa \oplus baabb = abbaabb$. Considering possible agglomerations of these words, we get $w_1 = abbaa \oplus baabb \oplus aabba = abbaabb \oplus aabba = abbaabba$, $w_2 = aabba \oplus abbaa \oplus baabb = aabbaa \oplus baabb = aabbaabb$ and $w_3 = baabb \oplus abbaa \oplus aabba = baabbaa \oplus aabba = baabbaabba$. Thus, $|w_1| = |pr(abbaa, baabb)| + |pr(baabb, aabba)| + |aabba| = |ab| + |b| + |aabba| = 2 + 1 + 5 = 8$, $\|P\| - |w_1| = 15 - 8 = 7$ and $|ov(abbaa, baabb)| + |ov(baabb, aabba)| = |baa| + |aabb| = 3 + 4 = 7$

1.2 Notation on graphs

We consider directed graphs with weighted arcs. A *directed graph* G is a pair (V_G, E_G) comprising a set of nodes V_G, and a set E_G of directed edges called *arcs*. An *arc* is an ordered pair of nodes.

Let w be a mapping from E_G onto the set of non negative integers (denoted $\mathbb{N}$). The *weighted directed graph* $G := (V_G, E_G, w)$ is a directed graph with the weights on its arcs given by w.

A *route* of G is an oriented path of G, that is a subset of V_G forming a chain between two nodes at its extremities. A *cycle* of G is a route of G where the same node is at both extremities. The weight of a route r equals the sum of the weights of its arcs. For simplicity, we extend the mapping w and let $w(r)$ denote the weight of r.

We investigate the performances of greedy algorithms for different types of covers of a graph, either by a route or by a set of cycles. Let X be a subset of arcs of V_G. X *covers* G if and only if each vertex v of G is the extremity of an arc of X.

1.3 Subset systems, extension, and greedy algorithms

A greedy algorithm builds a solution set by adding selected elements from a finite universe to maximise a given measure. In other words, the solution is iteratively extended. *Subset systems* are useful concepts to investigate how greedy algorithms can iteratively extend a current solution to a problem. A *subset system* is a pair $(E, \mathcal{L})$ comprising a finite set of elements E, and $\mathcal{L}$ a family of subsets of E satisfying two conditions:

(HS1) $\mathcal{L} \neq \emptyset$,
(HS2) If $A' \subseteq A$ and $A \in \mathcal{L}$, then $A' \in \mathcal{L}$. *i.e.*, $\mathcal{L}$ is close by taking a subset.

Let $A, B \in \mathcal{L}$. One says that B is an *extension of* A if $A \subseteq B$ and $B \in \mathcal{L}$. A subset system $(E, \mathcal{L})$ is said to be *k-extendible* if for all $C \in \mathcal{L}$ and $x \notin C$ such that $C \cup \{x\} \in \mathcal{L}$, and for any extension D of C, there exists a subset $Y \subseteq D \setminus C$ with $\#(Y) \leq k$ satisfying $D \setminus Y \cup \{x\} \in \mathcal{L}$.

The greedy algorithm associated with $(E, \mathcal{L})$ and a weight function w is presented in Algorithm 1. Checking whether $F \cup \{e_i\} \in \mathcal{L}$ consists in verifying the system's conditions. In the sequel of this paper, we will simply use "the greedy algorithm" to mean the greedy algorithm associated to a subset system, if the system is clear from the context. Mestre has shown that a matroid is a 1-extendible subset system, thereby demonstrating that a subset system is a generalisation of a matroid [8, Theorem 1]. In addition, a theorem from Mestre links k-extendibility and the approximation ratio of the associated greedy algorithm.

Theorem 2 (Mestre [8]). *Let $(E, \mathcal{L})$ be a k-extendible subset system. The associated greedy algorithm defined for the problem $(E, \mathcal{L})$ with weights w gives a $\frac{1}{k}$ approximation ratio.*

1.4 Definitions of problems and related work

Graph covers Let $G := (V_G, E_G, w)$ be a weighted directed graph.

The well known *Hamiltonian path* problem on G requires that the cover is a single path, while the *Cyclic Cover* problem searches for a cover made of cycles. We consider

Algorithm 1: The greedy algorithm associated with the subset system $(E, \mathcal{L})$ and weight function w.

```
Input : (E, L)
The elements e_i of E sorted by increasing weight: w(e_1) ≤ w(e_2) ≤ ... ≤ w(e_n)
F ← ∅
for i = 1 to n do
    if F ∪ {e_i} ∈ L then F ← F ∪ {e_i};
    ;
return F
```

the weighted versions of these two problems, where the solution must maximise the weight of the path or the sum of the weights of the cycles, respectively. In a general case, the graph is not symmetrical, and the weight function does not satisfy the Triangle inequality. When a Hamiltonian path is searched for, the problem is known as the *Maximum Asymmetric Travelling Salesman Problem* or Max-ATSP for short.

Definition 3 (Max-ATSP). *Let G be a weighted directed graph.* Max-ATSP *searches for a maximum weight Hamiltonian path on G.*

Max-ATSP is an important and well studied problem. It is known to be NP-hard and hard to approximate (precisely, Max-SNP hard). The best known approximation ratio of 2/3 is achieved by using a rounding technique on a Linear Programming relaxation of the problem [6]. However, the approximation ratio obtained by a simple greedy algorithm remains an interesting question, especially since other approximation algorithms are usually less efficient than a greedy one. In fact, Turner has shown a 1/3 approximation ratio for Max-ATSP [14, Thm 2.4]. As later explained, Max-ATSP is strongly related to the *Shortest Superstring Problem* and the *Maximal Compression* problems on strings.

If a set of cycles is needed as a cover the graph, the problem is called *Maximum Cyclic Cover*. In the general setup, cycles made of singletons are allowed in a solution.

Definition 4 (Max Cyclic Cover). *Let G be a weighted directed graph.* Maximum Cyclic Cover *searches for a set of cycles of maximum weight that collectively cover G.*

To our knowledge, the performance of a greedy algorithm for *Maximum Cyclic Cover* (Max-CC) has not yet been established, although variants of Max-CC with binary weights or with cycles of predefined lengths have been studied [1].

Superstring and Maximal Compression

Definition 5 (Superstring). *Let $P = \{s_1, s_2, \ldots, s_p\}$ be a set of p strings of $\Sigma^\star$. A* superstring *of P is a string s' such that s_i is a substring of s' for any i in $[1, p]$.*

Let us denote the sum of the lengths of the input strings by $\|S\| := \sum_{s_i \in S} |s_i|$. For any superstring s', there exists a set $\{i_1, \ldots, i_p\} = \{1, \ldots, p\}$ such that $s' = s_{i_1} \oplus s_{i_2} \oplus \cdots \oplus s_{i_p}$, and then $\|S\| - |s'| = \sum_{j=1}^{p-1} |ov(s_{i_j}, s_{i_{j+1}})|$.

Definition 6 (Shortest Superstring Problem (*SSP*)). *Let p be a positive integer and $P := \{s_1, s_2, \ldots, s_p\}$ be a set of p strings over Σ. Find s' a superstring of P of minimal length.*

Two approximation measures can be optimised:

- the length of the obtained superstring, that is $|s'|$, or
- the compression of the input strings achieved by the superstring: $\|P\| - |s'|$.

The corresponding approximation problems are termed *Shortest Superstring Problem* in the first case, or *Maximal Compression* in the second.

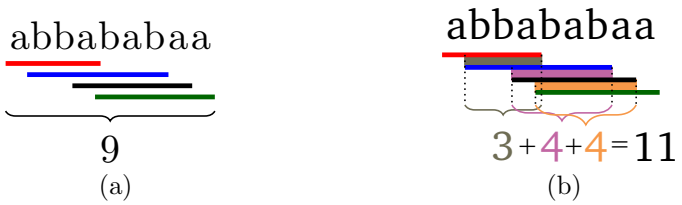


Figure 2: Consider the $P := \{abba, bbabab, ababa, babaa\}$. (a) The string *abbababaa* is a superstring of P of length 9, the figure shows the order of the word of P in the superstring. (b): the sum of the overlaps between adjacent words in *abbababaa* equals $\|P\| - |abbababaa| = 11$.

Figure 2 shows an input for Max-CC or *SSP*, with a superstring of length 9, which achieves a compression of 11.

Example 7. Let $P := \{a^k b, b^{k+1}, bc^k\}$ be a set of words; $w_g = a^k bc^k b^{k+1}$ is a superstring found by the greedy algorithm and $w_{opt} = a^k b^{k+1} c^k$ is an optimal superstring. Thus, the ratio of approximation is $\frac{|w_g|}{|w_{opt}|} = \frac{3k+2}{3k+1} \xrightarrow[k\to\infty]{} 1$ and the ratio of compression is 1/2. In other words, a greedy superstring may be almost optimal in length, but its compression is only 1/2.

Both *Maximal Compression* and *Shortest Superstring Problem* are NP-hard [3] and Max-SNP hard [2]. Numerous, complex algorithms have been designed for them, or their variants. Many are quite similar and use a procedure to find a *Maximum Cyclic Cover* of the input strings . The best known approximation ratio for the *Shortest Superstring Problem* was obtained in 2012 and equals $2\frac{11}{13}$ [10], although an optimal ratio of 2 has been conjectured in the 80's [13,2].

For the *Maximal Compression* problem, a recent algorithm gives a ratio of 3/4 [11]. A seminal work gave a proof of an approximation ratio of 1/2 by an algorithm that iteratively updates the input set by agglomerating two maximally overlapping strings until one string is left [13]. This algorithm was termed `greedy` but does not correspond to a greedy algorithm for it modifies the original input set. We demonstrate in Appendix that this algorithm yields the same result than a greedy algorithm defined for an appropriate subset system. Another proof of this ratio was given in [14]. Both proofs are quite intricate and include many subcases [13]. Thanks to subset systems, we provide a much simpler proof of this approximation ratio for *Maximal Compression* by a greedy algorithm, as well as an optimal and polynomial time greedy algorithm for the problem of Max Cyclic Covers on Strings.

Definition 8 (Shortest Cyclic Cover of Strings (SCCS)). *Let $p \in \mathbb{N}$ and let P be a set of p linear strings over Σ: $P := \{s_1, s_2, \ldots, s_p\}$. Find a set of cyclic strings of minimal cumulated length such that any input s_i, with $1 \leq i \leq p$, is a substring of at least one cyclic string.*

Several approximation algorithms for the *Shortest Superstring Problem* problem uses a procedure to solve SCCS on the instance, which is based on a modification of a polynomial time algorithm for the *assignment problem* [12,2,4]. This further indicates the importance of SCCS.

Both the *Maximal Compression* and the *Shortest Cyclic Cover of Strings* problems can be expressed as a cover of the *Overlap Graph*. In the Overlap Graph, the vertices represent the input strings, and an arc links s_i to s_j with weight $|ov(s_i, s_j)|$. Hence, the overlap graph is a complete graph with null or positive weights. A Hamiltonian path of this graph provides a permutation of the input strings; by agglomerating these strings in the order given by the permutation one obtains a superstring of P. Hence, the maximum weight Hamiltonian path induces a superstring that accumulates an optimal set of maximal overlaps, in other words a superstring that achieves maximal compression on P. Thus, a ρ approximation for Max-ATSP gives the same ratio for *Maximal Compression*. The same relation exists between the *Shortest Cyclic Cover of Strings* and *Maximum Cyclic Cover* on graphs. Indeed, SCCS optimises $\|P\| - \sum_j |c_j|$, where each c_j is a cyclic string in the solution, and Max-CC optimises the cumulated weight of the cycles of G. With the Overlap Graph, a minimal cyclic string is associated to each graph cycle by agglomerating the input strings in this cycle. Thus, the cumulated weight of a set of graph cycles corresponds to compression achieved by the set of induced cyclic strings. In other words, *Shortest Cyclic Cover of Strings* could also be called the Maximal Compression Cyclic Cover of Strings problem (and seen as a maximisation problem). The performance of a greedy algorithm for the *Shortest Cyclic Cover of Strings* problem is declared to be open in [15], while a claim saying that greedy is an exact algorithm for this problem appears in [4].

2 Maximum Asymmetric Travelling Salesman and Maximum Cyclic Cover Problems

Let w be a mapping from E_G onto the set of non negative integers and let $G := (V_G, E_G, w)$ be a directed graph with the weights on its arcs given by w. We first define a subset system for Max-ATSP and its accompanying greedy algorithm.

Definition 9. *Let $\mathcal{L}_S$ be the powerset of E_G. We define the pair $(E_G, \mathcal{L}_S)$ such that any F in $\mathcal{L}_S$ satisfies*

(L1) $\forall x, y$ *and* $z \in V_G$, (x, z) *and* $(y, z) \in F$ *implies* $x = y$,
(L2) $\forall x, y$ *and* $z \in V_G$, (z, x) *and* $(z, y) \in F$ *implies* $x = y$,
(L3) for any $r \in N^\star$, *there does not exist any cycle* $((x_1, x_2), \ldots, (x_{r-1}, x_r), (x_r, x_1))$ *in* F, *where* $\forall k \in \{1, \ldots, r\}, x_k \in V_G$.

Remark 10.

- In other words, for a subset F of E_G, Condition (L1) (resp. (L2)) allows only one ingoing (resp. outgoing) arc for each vertex of G.
- For all $F \in \mathcal{L}_S$ and for any $v \in V_G$, the arc (v, v) cannot belong to F, by Condition (L3) for $r = 1$.
- If in condition (L_3), one changes the set of forbidden values for r, the subset system addresses a different problem. As the proofs in this section do not depend of r, all results remain valid for these problems as well. For instance, with $r \in \{1\}$, only cycles of length one are forbidden; the solution is either a maximal path or

cyclic cover with cycles of length larger than one. The 1/3 approximation ratio obtained in Theorem 13 remains valid. We will consider later the case where all cycles are allowed (*i.e.*, $r \in \emptyset$).

Proposition 11. $(E_G, \mathcal{L}_S)$ *is a subset system.*

Proof. For (HS1), it suffices to note that $\emptyset \in \mathcal{L}_S$.For (HS2), we must show that each subset of an element of $\mathcal{L}_S$ is an element of $\mathcal{L}_S$. This is true since Conditions (L1), (L2), and (L3) are inherited by any subset of an element of $\mathcal{L}_S$.

Proposition 12 shows that the defined subset system is 3-extendible.

Proposition 12. $(E_G, \mathcal{L}_S)$ *is* 3*-extendible.*

Proof. Let $C \in \mathcal{L}_S$ and $e \notin C$ such that $C \cup \{e\} \in \mathcal{L}_S$. Let D be an extension of C. One must show that there exists a subset $Y \subseteq D \setminus C$ with $\#(Y) \leq 3$ such that $D \setminus Y \cup \{e\}$ belongs to $\mathcal{L}_S$.

As $e \in E_G$, there exists x and y such that $e = (x, y)$. Let Y be the set of elements of $D \setminus C$ of the form (x, z), (z, y), and (z, x) for any $z \in V_G$ where (z, x) belongs to a cycle in $D \cup \{x\}$. As D is an extension of C, D belongs to $\mathcal{L}_S$ and satisfies conditions (L1) and (L2). Hence, $\#(Y) \leq 3$.

It remains to show that $D \setminus Y \cup \{e\} \in \mathcal{L}_S$. As $C \cup \{e\} \in \mathcal{L}_S$, $C \cup \{e\}$ satisfies conditions (L1) and (L2), we know that for each $z \in V_G \setminus \{x, y\}$, the arcs (x, z) and (z, y) are not in C.

By the definition of Y, for each $z \in V_G$, we have that (x, z) and $(z, y) \notin D \setminus C$. Therefore, for all $z \in V_G$, (x, z) and $(z, y) \notin D \setminus Y$. Hence, $D \setminus Y \cup \{e\}$ satisfies conditions (L1) and (L2).

Now assume that $D \setminus Y \cup \{e\}$ violates Condition (L3). As $D \in \mathcal{L}_S$, D satisfies condition (L3) and $D \setminus Y$ too. The only element who can generate a cycle is e. As $C \cup \{e\} \in \mathcal{L}_S$, e does not generate a cycle in $C \cup \{e\}$, which implies that it generates a cycle in $D \setminus (C \cup Y)$. Hence, there exists $z \in V_G$ such that $(z, x) \in D \setminus (C \cup Y)$, which contradicts the definition of Y.

Now we derive the approximation ratio of the greedy algorithm for Max-ATSP. Another proof for this result originally published by [5] is given in [8, Theorem 6].

Theorem 13. *The greedy algorithm of* $(E_G, \mathcal{L}_S)$ *yields a* 1/3 *approximation ratio for Max-*ATSP*.*

Proof. By Proposition 12, $(E_G, \mathcal{L}_S)$ is 3-extendible. A direct application of Mestre's theorem (Theorem 2) yields the 1/3 approximation ratio for Max-ATSP.

Case of the Maximum Cyclic Cover problem If in condition (L_3) we ask that $r \in \emptyset$, (L_3) is not a constraint anymore and all cycles are allowed. This defines a new subset system, denoted by $(E_G, \mathcal{L}_\mathcal{C})$. As in the proof of Proposition 12, it suffices now to set $Y := \{(x, z), (z, y)\}$ (one does not need to remove an element of a cycle), and thus $\#(Y) \leq 2$. It follows that $(E_G, \mathcal{L}_\mathcal{C})$ is 2-extendible and that the greedy algorithm achieves a 1/2 approximation ratio for the *Maximum Cyclic Cover* problem.

3 Maximal Compression and Shortest Cyclic Cover of Strings

Blum and colleagues [2] have designed an algorithm called `greedy` that iteratively constructs a superstring for both the *Shortest Superstring Problem* and *Maximal Compression* problems. As mentioned in introduction, this algorithm is not a greedy algorithm per se. Below, we define a subset system corresponding to that of Max ATSP for the Overlap Graph, and study the approximation of the associated greedy algorithm. Before being able to conclude on the approximation ratio of the `greedy` algorithm of [2], we need to prove that `greedy` computes exactly the same superstring as the greedy algorithm of the subset system of Definition 14. This proof is given in Appendix. Knowing that these two algorithms are equivalent in terms of output, the approximation ratio of Theorem 18 is valid for both of them.

From now on, let $P := \{s_1, s_2, \ldots, s_p\}$ be a set of p strings of $\Sigma^\star$.

The subset system for *Maximal Compression* is similar to that of Max-ATSP. For any two strings s, t, $s \odot t$ represents the maximum overlap of s over t[1]. We set $E_P = \{s_i \odot s_j \mid s_i \text{ and } s_j \in P\}$. Hence, E_P is the set of maximum overlaps between any two words of S.

Definition 14 (Subset system for *Maximal Compression*). *Let $\mathcal{L}_\mathcal{P}$ as the set of $F \subseteq E_P$ such that:*

(L1) $\forall s_i,\ s_j$ *and* $s_k \in S$, $s_i \odot s_k$ *and* $s_j \odot s_k \in F \Rightarrow i = j$, i.e. *for each string, there is only one overlap to the left*

(L2) $\forall s_i,\ s_j$ *and* $s_k \in S$, $s_k \odot s_i$ *and* $s_k \odot s_i \in F \Rightarrow i = j$, *and only one overlap to the right*

(L3) for any $r \in N^\star$, *there exists no cycle* $(s_{i_1} \odot s_{i_2},\ \ldots,\ s_{i_{r-1}} \odot s_{i_r},\ s_{i_r} \odot s_{i_1})$ *in* F, *such that* $\forall k \in \{1, \ldots, r\}, s_{i_k} \in S$.

For each set $F := \{s_{i_1} \odot s_{i_2}, \ldots, s_{i_{p-1}} \odot s_{i_p}\}$ that is an inclusion-wise maximal element of $\mathcal{L}_\mathcal{P}$, we denote by $l(F)$ the superstring of S obtained by agglomerating the input strings of P according to the order induced by F:

$$l(F) := s_{i_1} \oplus s_{i_2} \oplus \cdots \oplus s_{i_p}.$$

First, knowing that *Maximal Compression* is equivalent to Max-ATSP on the Overlap Graph (see Section 1.4), we get a 1/3 approximation ratio for *Maximal Compression* as a corollary of Theorem 13. Another way to obtain this ratio is to show that the subset system is 3-extendible (the proof is identical to that of Proposition 12) and then use Theorem 2. However, the following example shows that the system $(E_P, \mathcal{L}_\mathcal{P})$ is not 2-extendible.

Example 15. The subset system $(E_P, \mathcal{L}_\mathcal{P})$ is not 2-extendible. Let $P := \{s_1, s_2, s_3, s_4, s_5\}$, $C := \emptyset$, $x := s_1 \odot s_2$. Then clearly $C \cup \{x\}$ belongs to $\mathcal{L}_\mathcal{P}$ and the set $D := \{s_1 \odot s_3,\ s_4 \odot s_2,\ s_5 \odot s_1,\ s_2 \odot s_5\}$ is an extension of C. However, when searching for a set Y such that Y included in $D \setminus C = D$ and such that $(D \setminus Y) \cup \{x\} \in \mathcal{L}_\mathcal{P}$ then $s_1 \odot s_3,\ s_4 \odot s_2$ must be removed to avoid violating $(L1)$ or $(L2)$, and at least one among $s_5 \odot s_1,\ s_2 \odot s_5$ must be removed to avoid violating $(L3)$. It follows that $\#(Y) \geq 3$.

[1] The notation $s \odot t$ represents the fact that s can be aggregated with t according to their maximal overlap. $ov(s, t)$ is a word representing a maximum overlap between s and t. Hence, $s \odot t$ differs $ov(s, t)$.

To prove a better approximation ratio for the greedy algorithm, we will need the Monge inequality [9] adapted to word overlaps.

Lemma 16. *Let s_1, s_2, s_3 and s_4 be four different words satisfying $|ov(s_1, s_2)| \geq |ov(s_1, s_4)|$ and $|ov(s_1, s_2)| \geq |ov(s_3, s_2)|$. So we have :*

$$|ov(s_1, s_2)| + |ov(s_3, s_4)| \geq |ov(s_1, s_4)| + |ov(s_3, s_2)|.$$

When for three sets A, B, C, we write $A \cup B \setminus C$, it means $(A \cup B) \setminus C$. Let $A \in \mathcal{L}_\mathcal{P}$ and let $\textsc{opt}(A)$ denote an extension of A of maximum weight. Thus, $\textsc{opt}(\emptyset)$ is an element of $\mathcal{L}_\mathcal{P}$ of maximum weight. The next lemma follows from this definition.

Lemma 17. *Let be $F \in \mathcal{L}_\mathcal{P}$ and $x \in E_P$, $w(\textsc{opt}(F \cup \{x\})) \leq w(\textsc{opt}(F))$.*

Now we can prove a better approximation ratio.

Theorem 18. *The approximation ratio of the greedy algorithm for the* Maximal Compression *problem is* $1/2$.

Proof. To prove this ratio, we revisit the proof of Theorem 2 in [8].

Let $x_1, x_2, \ldots, x_l$ denote the elements in the order in which the greedy algorithm includes them in its solution F, and let $F_0 := \emptyset, \ldots, F_l$ denote the successive values of the set F during the algorithm, in other words $F_i := F_{i-1} \cup \{x_i\}$ (see Algorithm 1 on p. 152). The structure of the proof is first to show for any element x_i incorporated by the greedy algorithm, the inequality $w(\textsc{opt}(F_{i-1})) \leq w(\textsc{opt}(F_i)) + w(x_i)$, and second, to reason by induction on the sets F_i starting with F_0.

One knows that $\textsc{opt}(F_{i-1})$ is an extension of F_{i-1}. By the greedy algorithm and by the definitions of F_{i-1} and x_i, one gets $F_{i-1} \cup \{x_i\} \in \mathcal{L}_\mathcal{P}$. As $x_i \in E_P$, there exist s_p and s_o such that $x_i = s_p \odot s_o$. Like in the proof of Proposition 12, let Y_i denote the subset of elements of $\textsc{opt}(F_{i-1}) \setminus F_{i-1}$ of the form $s_p \odot s_k$, $s_k \odot s_o$, or $s_k \odot s_p$, where $s_k \odot s_p$ belongs to a cycle in $\textsc{opt}(F_{i-1}) \cup \{x_i\}$. Thus, $\textsc{opt}(F_{i-1}) \setminus Y_i \cup \{x_i\} \in \mathcal{L}_\mathcal{P}$, and

$$\begin{aligned} w(\textsc{opt}(F_{i-1})) &= w(\textsc{opt}(F_{i-1}) \setminus Y_i \cup \{x_i\}) + w(Y_i) - w(x_i) \\ &\leq w(\textsc{opt}(F_i)) + w(Y_i) - w(x_i). \end{aligned}$$

Indeed, $w(\textsc{opt}(F_{i-1}) \setminus Y_i \cup \{x_i\}) \leq w(\textsc{opt}(F_i))$ because $\textsc{opt}(F_{i-1}) \setminus Y_i \cup \{x_i\}$ is an extension of $F_{i-1} \cup \{x_i\}$ and because $\textsc{opt}(F_i)$ is an extension of maximum weight of $F_{i-1} \cup \{x_i\}$.

Now let us show by contraposition that for any element $y \in Y_i$, $w(y) \leq w(x_i)$. Assume that there exists $y \in Y_i$ such that $w(y) > w(x_i)$. As $y \notin F_{i-1}$, y has already been considered by the greedy algorithm and not incorporated in the F. Hence, there exists $j \leq i$ such that $F_j \cup \{y\} \notin \mathcal{L}_\mathcal{P}$, but $F_j \cup \{y\} \subseteq \textsc{opt}(F_{i-1}) \in \mathcal{L}_\mathcal{P}$, which is a contradiction. Thus, we obtain $w(y) \leq w(x_i)$ for any $y \in Y_i$.

Now we know that $\#(Y_i) \leq 3$. Let us inspect two subcases.

Case 1 : $\#(Y_i) \leq 2$.
We have $w(Y) \geq 2w(x_i)$, hence $w(\textsc{opt}(F_{i-1})) \leq w(\textsc{opt}(F_i)) + w(x_i)$.

Case $2 : \#(Y_i) = 3$.
There exists s_k and $s_{k'}$ such that $s_p \odot s_{k'}$ and $s_k \odot s_o$ are in Y_i. By Lemma 16, we have $w(x_i) + w(s_k \odot s_{k'}) \geq w(s_p \odot s_{k'}) + w(s_k \odot s_o)$. As $s_p \odot s_{k'}$ and $s_k \odot s_o$ belong to $\text{OPT}(F_{i-1})$, one deduces $s_k \odot s_{k'} \notin \text{OPT}(F_{i-1})$.

We get $\text{OPT}(F_{i-1}) \setminus Y_i \cup \{x_i, s_k \odot s_{k'}\} \in \mathcal{L}_\mathcal{P}$. Indeed, as $Y_i \subseteq \text{OPT}(F_{i-1})$, neither a right overlap of s_k, nor a left overlap of $s_{k'}$ can belong to $\text{OPT}(F_{i-1})$. Furthermore, adding $s_k \odot s_{k'}$ to $\text{OPT}(F_{i-1}) \setminus Y_i \cup \{x_i\}$ cannot create a cycle, since otherwise a cycle would have already existed in $\text{OPT}(F_{i-1})$. This situation is illustrated in Figure 3.

We have $w(\text{OPT}(F_{i-1}) \setminus Y_i \cup \{x_i,\ s_k \odot s_{k'}\}) \leq w(\text{OPT}(F_{i-1} \cup \{x_i,\ s_k \odot s_{k'}\}))$, because $\text{OPT}(F_{i-1}) \setminus Y_i \cup \{x_i,\ s_k \odot s_{k'}\}$ is an extension of $F_{i-1} \cup \{x_i,\ s_k \odot s_{k'}\}$ and $\text{OPT}(F_{i-1} \cup \{x_i,\ s_k \odot s_{k'}\})$ is a maximum weight extension of $F_{i-1} \cup \{x_i,\ s_k \odot s_{k'}\}$. As $w(\text{OPT}(F_{i-1} \cup \{x_i,\ s_k \odot s_{k'}\})) \leq w(\text{OPT}(F_{i-1} \cup \{x_i\}))$, by Lemma 17 one gets:

$$\begin{aligned} w(\text{OPT}(F_{i-1})) &= w(\text{OPT}(F_{i-1}) \setminus Y_i \cup \{x_i,\ s_k \odot s_{k'}\}) + w(Y_i) - w(x_i) - w(s_k \odot s_{k'}) \\ &\leq w(\text{OPT}(F_{i-1} \cup \{x_i,\ s_k \odot s_{k'}\})) + w(Y_i) - w(x_i) - w(s_k \odot s_{k'}) \\ &\leq w(\text{OPT}(F_i)) + w(Y_i) - w(x_i) - w(s_k \odot s_{k'}). \end{aligned}$$

As $Y_i = \{s_p \odot s_{k'}, s_k \odot s_o, s_{k''} \odot s_p\}$, one obtains

$$\begin{aligned} w(\text{OPT}(F_{i-1})) &\leq w(\text{OPT}(F_i)) - w(s_k \odot s_{k'}) + w(Y_i) - w(x_i) \\ &\leq w(\text{OPT}(F_i)) - w(s_k \odot s_{k'}) + w(s_p \odot s'_k) + w(s_k \odot s_o) + w(s_{k''} \odot s_p) - w(x_i) \\ &\leq w(\text{OPT}(F_i)) + w(s_{k''} \odot s_p) \\ &\leq w(\text{OPT}(F_i)) + w(x_i). \end{aligned}$$

Remembering that $\text{OPT}(\emptyset)$ is an optimum solution, by induction one gets

$$\begin{aligned} w(\text{OPT}(F_0)) &\leq w(\text{OPT}(F_l)) + \sum_{i=1}^{l} w(x_i) \\ &\leq w(F_l) + w(F_l) \\ &\leq 2w(F_l). \end{aligned}$$

We can substitute $w(\text{OPT}(F_l))$ by $w(F_l)$ since F_l has a maximal weight by definition. Let s_{opt} be an optimal solution for *Maximal Compression*, $\|P\| - |s_{opt}| = w(\text{OPT}(\emptyset))$. As F_l is maximum, $l(F_l)$ is the superstring of P output by the greedy algorithm and thus, $\|P\| - |l(F_l)| = w(F_l)$. Therefore,

$$\frac{1}{2}(\|P\| - |s_{opt}|) \leq \|P\| - |l(F_l)|\,.$$

Finally, we obtain the desired ratio: the greedy algorithm of the subset system achieves an approximation ratio of 1/2 for the *Maximal Compression* problem.

3.1 Shortest Cyclic Cover of Strings

A solution for MC must avoid overlaps forming cycles in the constructed superstring. However, for the *Shortest Cyclic Cover of Strings* problem, cycles of any positive length are allowed. As in Definition 14, we can define a subset system for SCCS as the pair $(E_P, \mathcal{L}_\mathcal{C})$, where $\mathcal{L}_\mathcal{C}$ is now the set of $F \subseteq E_P$ satisfying only condition $(L1)$ and $(L2)$. A solution for this system with the weights defined as the length of maximal

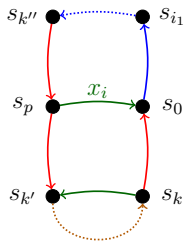


Figure 3: Impossibility to create a cycle by adding $s_k \odot s_{k'}$ to $\text{OPT}(F_{i-1}) \setminus Y_i \cup \{x_i\}$, without having an already existing cycle in $\text{OPT}(F_{i-1})$. Since we are adding x_i to $\text{OPT}(F_{i-1})$, we need to remove three elements in red: $s_{k''} \odot s_p, s_p \odot s_{k'}, s_k \odot s_o$.

overlaps is a set of cyclic strings containing the input words of P as substrings. One can see that the proof of Theorem 18 giving the 1/2 ratio for MC can be simplified to show that the greedy algorithm associated with the subset system $(E_P, \mathcal{L}_\mathcal{C})$ achieves a 1/1 approximation ratio, in other words exactly solves SCCS.

Theorem 19. *The greedy algorithm of* $(E_P, \mathcal{L}_\mathcal{C})$ *exactly solves* Shortest Cyclic Cover of Strings *problem in polynomial time.*

4 Conclusion

Greedy algorithms are algorithmically simpler, and usually easier to implement than more complex approximation algorithms [7,2,6,10,11]. In this work, we investigated the approximation ratio of greedy algorithms on several well known problems using the power of subset systems. Our major result is to prove these ratios with a unified and simple line of proof. Moreover, this approach can likely be reused for variants of these problems [1]. For the cover of graphs with maximum weight Hamiltonian path or set of cycles, the subset system and its associated greedy algorithm, provides an approximation ratio for a variety of problems, since distinct kinds of cycles can be forbidden in the third condition of the subset system (see Def. 9 on p. 154). For the general *Maximum Asymmetric Travelling Salesman Problem* problem, it achieves a 1/3 ratio, and a 1/2 ratio for the *Maximum Cyclic Cover* problem.

Today, the upper and lower bounds of approximation are still being refined for the *Shortest Superstring Problem* and *Maximal Compression* problems. It is important to know how good greedy algorithms are. Here, we have shown that the greedy algorithm solves the *Shortest Cyclic Cover of Strings* problem exactly, and gave an alternative proof of the 1/2 approximation ratio for *Maximal Compression*(Theorem 18). The latter is important for it shows that, beside the 3-extendibility, one only needs to consider the Monge property, to achieve this bound. It also illustrates how a combinatorial property that is problem specific can help to extend the approach of Mestre, while still using the theory of subset systems [8].

References

1. M. Bläser and B. Manthey: *Approximating maximum weight cycle covers in directed graphs with weights zero and one.* Algorithmica, 42(2) 2005, pp. 121–139.
2. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis: *Linear approximation of shortest superstrings*, in ACM Symposium on the Theory of Computing, 1991, pp. 328–336.

3. J. Gallant, D. Maier, and J. A. Storer: *On finding minimal length superstrings.* Journal of Computer and System Sciences, 20 1980, pp. 50–58.
4. D. Gusfield: *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, 1997.
5. T. A. Jenkyns: *The greedy travelling salesman's problem.* Networks, 9(4) 1979, pp. 363–373.
6. H. Kaplan, M. Lewenstein, N. Shafrir, and M. Sviridenko: *Approximation algorithms for asymmetric tsp by decomposing directed regular multigraphs.* J. of Association for Computing Machinery, 52(4) July 2005, pp. 602–626.
7. H. Kaplan and N. Shafrir: *The greedy algorithm for shortest superstrings.* Information Processing Letters, 93(1) 2005, pp. 13–17.
8. J. Mestre: *Greedy in Approximation Algorithms*, in Proceedings of 14th Annual European Symposium on Algorithms (ESA), vol. 4168 of Lecture Notes in Computer Science, Springer, 2006, pp. 528–539.
9. G. Monge: *Mémoire sur la théorie des déblais et des remblais*, in Mémoires de l'Académie Royale des Sciences, 1781, pp. 666–704.
10. M. Mucha: *Lyndon words and short superstrings*, in Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 2013, pp. 958–972.
11. K. E. Paluch: *Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring.* CoRR, abs/1401.3670 2014.
12. C. H. Papadimitriou and K. Steiglitz: *Combinatorial optimization : algorithms and complexity*, Dover Publications, Inc., 2nd ed., 1998, 496 p.
13. J. Tarhio and E. Ukkonen: *A greedy approximation algorithm for constructing shortest common superstrings.* Theoretical Computer Sciences, 57 1988, pp. 131–145.
14. J. S. Turner: *Approximation algorithms for the shortest common superstring problem.* Information and Computation, 83(1) Oct. 1989, pp. 1–20.
15. M. Weinard and G. Schnitger: *On the greedy superstring conjecture.* SIAM Journal on Discrete Mathematics, 20(2) 2006, pp. 502–522.

Appendix

Here, we prove that the algorithm `greedy` defined by Tarhio and Ukkonen [13] and studied by Blum and colleagues [2] for the *Maximal Compression* problem, computes exactly the same superstring as the greedy algorithm of the subset system $(E_P, \mathcal{L}_\mathcal{P})$ (see Definition 14 on p. 156). This is to show that these two algorithms are equivalent in terms of output and that the approximation ratio of 1/2 of Theorem 18 is valid for both of them. Remind that the input, $P := \{s_1, s_2, \ldots, s_p\}$, is a set of p strings of $\Sigma^\star$.

Proposition 20. *Let F be an maximal element for inclusion of $\mathcal{L}_\mathcal{P}$. Thus, there exists a permutation of the input strings, that is a set $\{i_1, \ldots, i_p\} = \{1, \ldots, p\}$ such that*

$$F = \{s_{i_1} \odot s_{i_2}, s_{i_2} \odot s_{i_3}, \ldots, s_{i_{p-1}} \odot s_{i_p}\}.$$

Proof. By the condition (L3), cycles are forbidden in F. Hence there exist $s_{d_1}, s_x \in S$ such that $s_{d_1} \odot s_x \in F$, and for all $s_y \in S$, $s_y \odot s_{d_1} \notin F$.

Thus, let $(i_j)_{j\in I}$ be the sequence of elements of P such that $i_1 = d_1$, for all $j \in I$ such that $j+1 \in I$, $s_{i_j} \odot s_{i_{j+1}} \in F$, and the size of I is maximum. As F has no cycle (condition L3), I is finite; then let us denote by t_1 its largest element. We have for all $s_y \in P$, $s_{t_1} \odot s_y \notin F$. Hence, $\cup_{j\in I} i_j$ is the interval comprised between s_{d_1} and s_{t_1}.

Assume that $F \setminus \{\cup_{j\in I} i_j\} \neq \emptyset$. We iterate the reasoning by taking the interval between s_{d_2} and s_{t_2} and so on until F is exhausted. We obtain that F is the set of intervals between s_{d_i} and s_{t_i}. By the condition (L1) and (L2), s_{t_1} (resp. s_{d_2}) is in the interval between s_{d_j} and $s_{t_j} \Rightarrow j = 1$ (resp. $j = 2$). As $s_{t_1} \odot s_{d_2} \in E$, and $F \cup \{s_{t_1} \odot s_{d_2}\} \in \mathcal{L}_\mathcal{P}$, F is not maximum, which contradicts our hypothesis.

We obtain that $F \setminus \{\cup_{j\in I} i_j\} = \emptyset$, hence the result.

For each set $F := \{s_{i_1} \odot s_{i_2}, \ldots, s_{i_{p-1}} \odot s_{i_p}\}$ that is a maximal element of $\mathcal{L}_\mathcal{P}$ for inclusion, remind that $l(F)$ denotes the superstring of S obtained by agglomerating the input strings of P according to the order induced by F:

$$l(F) := s_{i_1} \oplus s_{i_2} \oplus \cdots \oplus s_{i_p}.$$

The algorithm `greedy` takes from set P two words u and v having the largest maximum overlap, replaces u and v with $a \oplus b$ in P, and iterates until P is a singleton.

Proposition 21. *Let F be the output of the greedy algorithm of subset system $(E_P, \mathcal{L}_\mathcal{P})$, and S the output of Algorithm* `Greedy` *for the input P. Then $S = \{l(F)\}$.*

Proof. First, see that for any i between 1 and p, there exists s_j and s_k such that $e_i = s_j \odot s_k$. If $F \cup \{e_i\} \in \mathcal{L}_\mathcal{P}$, then by Conditions (L1) and (L2), one forbids any other left overlap of s_k or any other right overlap of s_j are prohibited in the following. As cycles are forbidden by condition (L3), one will finally obtain the same superstring by exchanging the pair s_j and s_k with $s_j \oplus s_k$ in E.

The algorithm `greedy` from [13] can be seen as the greedy algorithm of the subset system $(E_P, \mathcal{L}_\mathcal{P})$. By the definition of the weight w, the later also answers to the *Maximal Compression* problem. Both algorithms are thus equivalent.

Closed Factorization*

Golnaz Badkobeh[1], Hideo Bannai[2], Keisuke Goto[2], Tomohiro I[2],
Costas S. Iliopoulos[3], Shunsuke Inenaga[2], Simon J. Puglisi[4], and Shiho Sugimoto[2]

[1] Department of Computer Science,
University of Sheffield
United Kingdom
g.badkobeh@sheffield.ac.uk

[2] Department of Informatics,
Kyushu University
Japan
{bannai,keisuke.gotou,tomohiro.i,inenaga,shiho.sugimoto}@inf.kyushu-u.ac.jp

[3] Department of Informatics,
King's College London
United Kingdom
c.iliopoulos@kcl.ac.uk

[4] Department of Computer Science,
University of Helsinki
Finland
puglisi@cs.helsinki.fi

Abstract. A *closed string* is a string with a proper substring that occurs in the string as a prefix *and* a suffix, but not elsewhere. Closed strings were introduced by Fici (Proc. WORDS, 2011) as objects of combinatorial interest in the study of Trapezoidal and Sturmian words. In this paper we consider algorithms for computing closed factors (substrings) in strings, and in particular for greedily factorizing a string into a sequence of longest closed factors. We describe an algorithm for this problem that uses linear time and space. We then consider the related problem of computing, for every position in the string, the longest closed factor starting at that position. We describe a simple algorithm for the problem that runs in $O(n \log n / \log\log n)$ time.

1 Introduction

A *closed string* is a string with a proper substring that occurs as a prefix and a suffix but does not have internal occurrences. Closed strings were introduced by Fici [3] as objects of combinatorial interest in the study of Trapezoidal and Sturmian words. Since then, Badkobeh, Fici, and Liptak [1] have proved a tight lowerbound for the number of closed factors (substrings) in strings of given length and alphabet.

In this paper we initiate the study of algorithms for computing closed factors. In particular we consider two algorithmic problems. The first, which we call the *closed factorization problem*, is to greedily factorize a given string into a sequence of longest closed factors (we give a formal definition of the problem below, in Section 2). We describe an algorithm for this problem that uses $O(n)$ time and space, where n is the length of the given string.

The second problem we consider is the *closed factor array problem*, which requires us to compute the length of the longest closed factor starting at each position in the

* This research is partially supported by the Academy of Finland through grants 258308 and 250345 (CoECGR), and by the Japan Society for the Promotion of Science.

Golnaz Badkobeh, Hideo Bannai, Keisuke Goto, Tomohiro I, Costas S. Iliopoulos, Shunsuke Inenaga, Simon J. Puglisi, Shiho Sugimoto: Closed Factorization, pp. 162–168.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

input string. We show that this problem can be solved in $O(n\frac{\log n}{\log\log n})$ time, using techniques from computational geometry.

This paper proceeds as follows. In the next section we set notation, define the problems more formally, and outline basic data structures and concepts. Section 3 describes an efficient solution to the closed factorization problem and Section 4 then considers the closed factor array. Reflections and outlook are offered in Section 5.

2 Preliminaries

2.1 Strings and Closed Factorization

Let Σ denote a fixed integer alphabet. An element of Σ^* is called a string. For any strings $\mathsf{W}, \mathsf{X}, \mathsf{Y}, \mathsf{Z}$ such that $\mathsf{W} = \mathsf{XYZ}$, the strings $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are respectively called a prefix, substring, and suffix of W. The length of a string X will be denoted by $|\mathsf{X}|$. Let ε denote the empty string of length 0, i.e., $|\varepsilon| = 0$. For any non-negative integer n, $\mathsf{X}[1, n]$ denotes a string X of length n. A prefix X of a string W with $|\mathsf{X}| < |\mathsf{W}|$ is called a proper prefix of W. Similarly, a suffix X of W with $|\mathsf{Z}| < |\mathsf{W}|$ is called a proper suffix of W. For any string X and integer $1 \leq i \leq |\mathsf{X}|$, let $\mathsf{X}[i]$ denote the ith character of X, and for any integers $1 \leq i \leq j \leq |\mathsf{X}|$, let $\mathsf{X}[i..j]$ denote the substring of X that starts at position i and ends at position j. For convenience, let $\mathsf{X}[i..j]$ be the empty string if $j < i$. For any strings X and Y, if $\mathsf{Y} = \mathsf{X}[i..j]$, then we say that i is an occurrence of Y in X.

If a non-empty string X is both a proper prefix and suffix of string W, then, X is called a *border* of W. A string W is said to be *closed*, if there exists a border X of W that occurs exactly twice in W, i.e., $\mathsf{X} = \mathsf{W}[1..|\mathsf{X}|] = \mathsf{W}[|\mathsf{W}| - |\mathsf{X}| + 1..|\mathsf{W}|]$ and $\mathsf{X} \neq \mathsf{W}[i..i + |\mathsf{X}| - 1]$ for any $2 \leq i \leq |\mathsf{W}| - |\mathsf{X}|$. We suppose that any single character $\mathsf{C} \in \Sigma$ is closed, assuming that the empty string ε occurs exactly twice in C. A string X is a *closed factor* of W, if X is closed and is a substring of W. Throughout we consider a string $\mathsf{X}[1, n]$ on Σ. We define the *closed factorization* of string $\mathsf{X}[1, n]$ as follows.

Definition 1 (Closed Factorization). *The closed factorization of string* $\mathsf{X}[1, n]$*, denoted* $\mathsf{CF}(\mathsf{X})$*, is a sequence* $(\mathsf{G}_0, \mathsf{G}_1, \ldots, \mathsf{G}_k)$ *of strings such that* $\mathsf{G}_0 = \varepsilon$*,* $\mathsf{X}[1, n] = \mathsf{G}_1 \cdots \mathsf{G}_k$ *and, for each* $1 \leq j \leq k$*,* G_j *is the longest prefix of* $\mathsf{X}[|\mathsf{G}_1 \cdots \mathsf{G}_{j-1}| + 1..n]$ *that is closed.*

Example 2. For string $\mathsf{X} = \mathtt{ababaacbbbcbcc\$}$, $\mathsf{CF}(\mathsf{X}) = (\varepsilon, \mathtt{ababa}, \mathtt{a}, \mathtt{cbbbcb}, \mathtt{cc}, \mathtt{\$})$.

We remark that a closed factor G_j is a single character if and only if $|\mathsf{G}_1 \cdots \mathsf{G}_{j-1}|+1$ is the rightmost (last) occurrence of character $\mathsf{X}[|\mathsf{G}_1 \cdots \mathsf{G}_{j-1}| + 1]$ in X.

We also define the *longest closed factor array* of string $\mathsf{X}[1, n]$.

Definition 3 (Longest Closed Factor Array). *The longest closed factor array of* $\mathsf{X}[1, n]$ *is an array* $\mathsf{A}[1, n]$ *of integers such that for any* $1 \leq i \leq n$*,* $\mathsf{A}[i] = \ell$ *if and only if* ℓ *is the length of the longest prefix of* $\mathsf{X}[i..n]$ *that is closed.*

Example 4. For string $\mathsf{X} = \mathtt{ababaacbbbcbcc\$}$, $\mathsf{A} = [5, 4, 3, 5, 2, 1, 6, 3, 2, 4, 3, 1, 2, 1, 1]$.

Clearly, given the longest closed factor array $\mathsf{A}[1, n]$ of string X, $\mathsf{CF}(\mathsf{X})$ can be computed in $O(n)$ time. However, the algorithm we describe in Section 4 to compute $\mathsf{A}[1, n]$ requires $O(n\frac{\log n}{\log\log n})$ time, and so using it to compute $\mathsf{CF}(\mathsf{X})$ would also take $O(n\frac{\log n}{\log\log n})$ time overall. In Section 3 we present an optimal $O(n)$-time algorithm to compute $\mathsf{CF}(\mathsf{X})$ that does not require $\mathsf{A}[1, n]$.

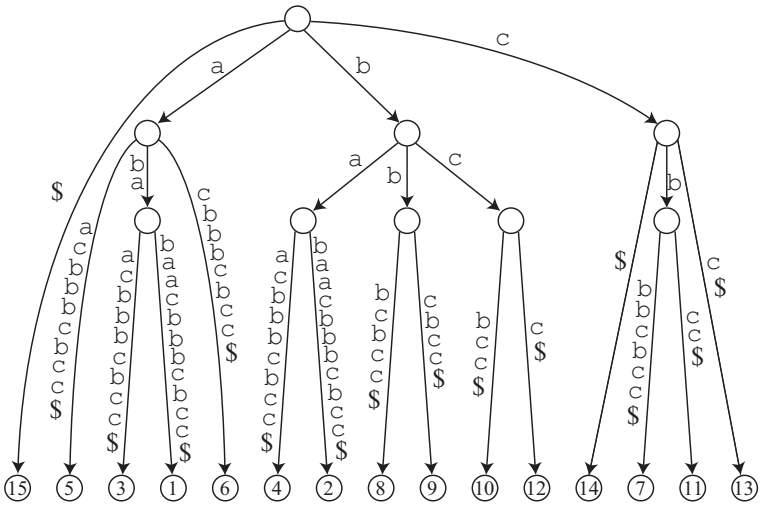


Figure 1. The suffix tree of string X = ababaacbbbcbcc\$, where each leaf stores the beginning position of the corresponding suffix. All the branches from an internal node are sorted in ascending lexicographical order, assuming \$ is the lexicographically smallest. The suffix array SA of X is $[15, 5, 3, 1, 6, 4, 2, 8, 9, 10, 12, 14, 7, 11, 13]$, which corresponds to the sequence of leaves from left to right, and thus can be computed in linear time by a depth first traversal on the suffix tree.

2.2 Tools

The suffix array [5] SA_X (we drop subscripts when they are clear from the context) of a string X is an array $\mathsf{SA}[1..n]$ which contains a permutation of the integers $[1..n]$ such that $\mathsf{X}[\mathsf{SA}[1]..n] < \mathsf{X}[\mathsf{SA}[2]..n] < \cdots < \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n]$ is the j^{th} suffix of X in ascending lexicographical order.

The suffix tree [8] of a string $\mathsf{X}[1, n]$ is a compacted trie consisting of all suffixes of X. Suffix trees can be represented in linear space, and can be constructed in linear time for integer alphabets [2]. Figure 1 illustrates the suffix tree for an example string.

For a node w in the suffix tree let $\mathsf{pathlabel}(w)$ be the string spelt out by the letters on the edges on the path from the root to w. If there is a branch from node u to node v and u is an ancestor of v then we say $u = \mathsf{parent}(v)$. Assuming that every string X terminates with a special character \$ which occurs nowhere else in X, there is a one-to-one correspondence between the suffixes of X and the leaves of the suffix tree of X. We assume the branches from a node u to each child v of u are stored in ascending lexicographical order of $\mathsf{pathlabel}(v)$. When this is the case, SA is simply the leaves of the suffix tree when read during a depth-first traversal. At each internal node v in the suffix tree we store two additional values $v.s$ and $v.e$ such that $\mathsf{SA}[v.s..v.e]$ contains the beginning positions of all the suffixes in the subtree rooted at v.

3 Greedy Longest Closed Factorization in Linear Time

In this section, we present how to compute the closed factorization $\mathsf{CF}(\mathsf{X})$ of a given string $\mathsf{X}[1, n]$. Our high level strategy is to build a data structure that helps us to efficiently compute, for a given position i in X, the longest closed factor starting at i. The core of this data structure is the suffix tree for X, which we decorate in various ways.

Let S be the set of the beginning positions of the longest closed factors in $\mathsf{CF}(\mathsf{X})$. For any $i \in S$, let $\mathsf{G} = \mathsf{X}[i..i + |\mathsf{G}| - 1]$ be the longest closed factor of X starting at position i in X.

Let G' be the unique border of the longest closed factor G starting at position i of X, and b_i be its length, i.e., $\mathsf{G}' = \mathsf{G}[1..b_i] = \mathsf{X}[i..i + b_i - 1]$ (if G is a single character, then $\mathsf{G}' = \varepsilon$ and $b_i = 0$). The following lemma shows that we can efficiently compute $\mathsf{CF}(\mathsf{X})$ if we know b_i for all $i \in S$.

Lemma 5. *Given b_i for all $i \in S$, we can compute $\mathsf{CF}(\mathsf{X})$ in a total of $\mathrm{O}(n)$ time and space independently of the alphabet size.*

Proof. If $b_i = 0$, then $\mathsf{G} = \mathsf{X}[i]$. Hence, in this case it clearly takes $\mathrm{O}(1)$ time and space to compute G.

If $b_i > 1$, then we can compute G in $\mathrm{O}(|\mathsf{G}|)$ time and $\mathrm{O}(b_i)$ space, as follows. We preprocess the border G' of G using the Knuth-Morris-Pratt (KMP) string matching algorithm [4]. This preprocessing takes $\mathrm{O}(b_i)$ time and space. We then search for the first occurrence of G' in $\mathsf{X}[i + 1..n]$ (i.e. the next occurrence of the longest border of $\mathsf{G}[1, m]$ to the right of the occurrence $\mathsf{X}[i..i + b_i - 1]$). The location of the next occurrence tells us where the end of the closed factor is, and so it also tells us $\mathsf{G} = \mathsf{X}[i..i + |\mathsf{G}| - 1]$. The search takes $\mathrm{O}(|\mathsf{G}|)$ time — i.e. time proportional to the length of the closed factor. Because the sum of the lengths of the closed factors is n, over the whole factorization we take $\mathrm{O}(n)$ time and space. The running time and space usage of the algorithm are clearly independent of the alphabet size. □

What remains is to be able to efficiently compute b_i for a given $i \in S$. The following lemma gives an efficient solution to this subproblem:

Lemma 6. *We can preprocess the suffix tree of string $\mathsf{X}[1, n]$ in $\mathrm{O}(n)$ time and space, so that b_i for each $i \in S$ can be computed in $\mathrm{O}(1)$ time.*

Proof. In each leaf of the suffix tree, we store the beginning position of the suffix corresponding to the leaf. For any internal node v of the suffix tree of X, let $\max(v)$ denote the maximum leaf value in the subtree rooted at v, i.e.,

$$\max(v) = \max\{i \mid \mathsf{X}[i..i + \mathsf{pathlabel}(v) - 1], 1 \leq i \leq n - \mathsf{pathlabel}(v) - 1\}.$$

We can compute $\max(v)$ for every v in a total of $\mathrm{O}(n)$ time total via a depth first traversal. Next, let $\mathsf{P}[1, n]$ be an array of pointers to suffix tree nodes (to be computed next). Initially every $\mathsf{P}[i]$ is set to null. We traverse the suffix tree in pre-order, and for each node v we encounter we set $\mathsf{P}[\max(v)] = v$ if $\mathsf{P}[\max(v)]$ is null. At the end of the traversal $\mathsf{P}[i]$ will contain a pointer to the highest node w in the tree for which i is the maximum leaf value (i.e., i is the rightmost occurrence of $\mathsf{pathlabel}(w)$).

We are now able to compute b_i, the length of the unique border of the longest closed factor starting at any given i, as follows. First we retrieve node $v = \mathsf{P}[i]$. Observe that, because of the definition of $\mathsf{P}[i]$, there are no occurrences of substring $\mathsf{X}[i..i + |\mathsf{pathlabel}(v)|]$ to the right of i. Let $u = \mathsf{parent}(v)$. There are two cases to consider:

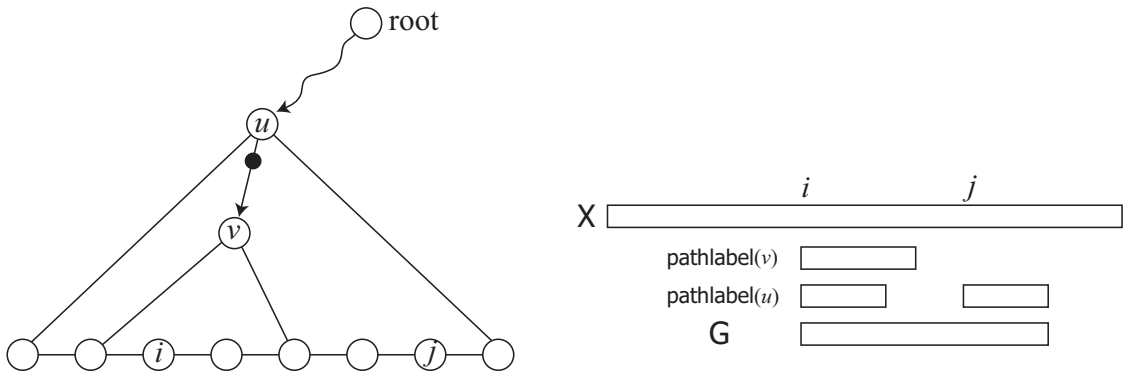


Figure 2. Illustration for Lemma 6. We consider the longest closed factor G starting at position i of string X. We retrieve node $\mathsf{P}[i] = v$, which implies $\max(v) = i$. Let u be the parent of v. The black circle represents a (possibly implicit) node which represents $\mathsf{X}[i..i + \mathsf{pathlabel}(u)]$, which has the same set of occurrences as $\mathsf{pathlabel}(v)$. Hence $b_i = |\mathsf{pathlabel}(u)|$, and therefore $\mathsf{G} = \mathsf{X}[i..j + \mathsf{pathlabel}(u) - 1]$, where j is the leftmost occurrence of $\mathsf{pathlabel}(u)$ with $j > i$.

- If u is not the root, then observe that there always exists an occurrence of substring $\mathsf{pathlabel}(u)$ to the right of position i (otherwise i would be the rightmost occurrence of $\mathsf{pathlabel}(u)$, but this cannot be the case since u is higher than v, and we defined $\mathsf{P}[i]$ to be the highest node w with $\max(w) = i$). Let j be the the leftmost occurrence of $\mathsf{pathlabel}(u)$ to the right of i. Then, the longest closed factor starting at position i is $\mathsf{X}[i..j + |\mathsf{pathlabel}(u)| - 1]$ (this position j is found by the KMP algorithm as in Lemma 5).
- If u is the root, then it turns out that i is the rightmost occurrence of character $\mathsf{X}[i]$ in X. Hence, the longest closed factor starting at position i is $\mathsf{X}[i]$.

The thing we have not shown is that $|\mathsf{pathlabel}(u)| = b_i$. This is indeed the case, since the set of occurrences of $\mathsf{X}[i..j + |\mathsf{pathlabel}(u)|]$ (i.e., leaves in the subtree corresponding to the string) is equivalent to that of $\mathsf{pathlabel}(v)$, any substring starting at i that is longer than $|\mathsf{pathlabel}(u)|$ does not occur to the right of i and thus b_i cannot be any longer. Hence $|\mathsf{pathlabel}(u)| = b_i$. (See also Figure 2).

Clearly $v = \mathsf{P}[i]$ can be retrieved in $\mathrm{O}(1)$ time for a given i, and then $u = \mathsf{parent}(v)$ can be obtained in $\mathrm{O}(1)$ time from v. This completes the proof. □

The main result of this section follows:

Theorem 7. *Given a string* $\mathsf{X}[1, n]$ *over an integer alphabet, the* closed factorization $\mathsf{CF}(\mathsf{X}) = (\mathsf{G}_1, \ldots, \mathsf{G}_k)$ *of* X *can be computed in* $\mathrm{O}(n)$ *time and space.*

Proof. $\mathsf{G}_0 = \varepsilon$ by definition and so does not need to be computed. We compute the other G_j in ascending order of $j = 1, \ldots, k$. Let s_i be the beginning position of G_i in X, i.e., $s_1 = 1$ and $s_i = |\mathsf{G}_1 \cdots \mathsf{G}_{i-1}| + 1$ for $1 < i \leq k$. We compute G_1 in $\mathrm{O}(|\mathsf{G}_1|)$ time and space from b_{s_1} using Lemma 5 and Lemma 6. Assume we have computed the first $j - 1$ factors $\mathsf{G}_1, \ldots, \mathsf{G}_{j-1}$ for any $1 \leq j < k - 1$. We then compute G_j in $\mathrm{O}(|\mathsf{G}_j|)$ time and space from b_{s_j}, again using Lemmas 5 and 6. Since $\sum_{j=1}^{k} |\mathsf{G}_j| = n$, the proof completes. □

The following is an example of how the algorithm presented in this section computes $\mathsf{CF}(\mathsf{X})$ for a given string X.

Example 8. Consider the running example string $\mathsf{X} = \texttt{ababaacbbbcbcc\$}$, and see Figure 1, which shows the suffix tree of X.

1. We begin with node $\mathsf{P}[1]$ representing $\texttt{ababaacbbbcbcc\$}$, whose parent represents $\texttt{aba}$. Hence we get $b_1 = |\texttt{aba}| = 3$. We run the KMP algorithm with pattern $\texttt{aba}$ and find the first factor $\mathsf{G}_1 = \texttt{ababa}$.
2. We then check node $\mathsf{P}[6]$ representing $\texttt{a}$. Since its parent is the root, we get $b_2 = 0$ and therefore the second factor is $\mathsf{G}_2 = \texttt{a}$.
3. We then check node $\mathsf{P}[7]$ representing $\texttt{cbbbcbcc\$}$, whose parent represents $\texttt{cb}$. Hence we get $b_3 = |\texttt{cb}| = 2$. We run the KMP algorithm with pattern $\texttt{cb}$ and find the third factor $\mathsf{G}_3 = \texttt{cbbbcb}$.
4. We then check node $\mathsf{P}[13]$ representing $\texttt{cc\$}$, whose parent represents $\texttt{c}$. Hence we get $b_4 = |\texttt{c}| = 1$. We run the KMP algorithm with pattern $\texttt{c}$ and find the fourth factor $\mathsf{G}_4 = \texttt{cc}$.
5. We finally check node $\mathsf{P}[15]$ representing \$. Since its parent is the root, we get $b_5 = 0$ and therefore the fifth factor is $\mathsf{G}_5 = \$$.

Consequently, we obtain $\mathsf{CF}(\mathsf{X}) = (\texttt{ababa}, \texttt{a}, \texttt{cbbbcb}, \texttt{cc}, \$)$, which coincides with Example 2.

4 Longest Closed Factor Array

A natural extension of the problem in the previous section is to compute the longest closed factor starting at *every* position in X in linear time — not just those involved in the factorization. Formally, we would like to compute the longest closed factor array of X, i.e., an array $\mathsf{A}[1, n]$ of integers such that $\mathsf{A}[i] = \ell$ if and only if ℓ is the length of the longest closed factor starting at position i in X.

Our algorithm for closed factorization computes the longest closed factor starting at a given position in time proportional to the factor's length, and so does not immediately provide a linear time algorithm for computing A; indeed, applying the algorithm naïvely at each position would take $\mathrm{O}(n^2)$ time to compute A. In what follows, we present a more efficient solution:

Theorem 9. *Given a string* $\mathsf{X}[1, n]$ *over an integer alphabet, the* closed factor array *of* X *can be computed in* $\mathrm{O}(n\frac{\log n}{\log\log n})$ *time and* $\mathrm{O}(n)$ *space.*

Proof. We extend the data structure of the last section to allow A to be computed in $\mathrm{O}(n\frac{\log n}{\log\log n})$ time and $\mathrm{O}(n)$ space. The main change is to replace the KMP algorithm scanning in the first algorithm with a data structure that allows us to find the end of the closed factor in time independent of its length.

We first preprocess the suffix array SA for *range successor* queries, building the data structure of Yu, Hon and Wang [9]. A range successor query $\mathsf{rsq}_{\mathsf{SA}}(s, e, k)$ returns, given a range $[s, e] \subseteq [1, n]$, the smallest value $x \in \mathsf{SA}[s..e]$ such that $x > k$, or null if there is no value larger than k in $\mathsf{SA}[s..e]$. Yu et al.'s data structure allows range successor queries to be answered in $\mathrm{O}(\frac{\log n}{\log\log n})$ time each, takes $\mathrm{O}(n)$ space, and $\mathrm{O}(n\frac{\log n}{\log\log n})$ time to construct.

Now, to compute the longest closed factor starting at a given position i in X (i.e. to compute $\mathsf{A}[i]$) we do the following. First we compute b_i, the length of the border of the longest closed factor starting at i, in $\mathrm{O}(1)$ time using Lemma 6. Recall that in the process of computing b_i we determine the node u having $\mathsf{pathlabel}(u) = \mathsf{X}[i..i+b_i-1]$.

To determine the end of the closed factor we must find the smallest $j > i$ such that $\mathsf{X}[j..j + b_i - 1] = \mathsf{X}[i..i + b_i - 1]$. Observe that j, if it exists, is precisely the answer to $\mathsf{rsq}_{\mathsf{SA}}(u.s, u.e, i)$. (See also the left diagram of Figure 2. Assuming that the leaves in the subtree rooted at u are sorted in the lexicographical order, the leftmost and rightmost leaves in the subtree correspond to the $u.s$-th and $u.e$-th entries of SA, respectively. Hence, $j = \mathsf{rsq}_{\mathsf{SA}}(u.s, u.e, i)$). For each $\mathsf{A}[i]$ we spend $\mathrm{O}(\frac{\log n}{\log \log n})$ time and so overall the algorithm takes $\mathrm{O}(n\frac{\log n}{\log \log n})$ time. The space requirement is clearly $\mathrm{O}(n)$. □

We note that recently Navarro and Neckrich [7] described range successor data structures with faster $\mathrm{O}(\sqrt{\log n})$-time queries, but straightforward construction takes $\mathrm{O}(n \log n)$ time [6], so overall this does not improve the runtime of our algorithm.

5 Concluding Remarks

We have considered but two problems on closed factors here, and many others remain. For example, how efficiently can one compute all the closed factors in a string (or, say, the closed factors that occur at least k times)? Relatedly, how many closed factors does a string contain in the worst case and on average?

One also wonders if the closed factor array can be computed in linear time, by somehow avoiding range successor queries.

References

1. G. Badkobeh, G. Fici, and Zs. Lipták: *A note on words with the smallest number of closed factors*. `http://arxiv.org/abs/1305.6395`, 2013.
2. M. Farach: *Optimal suffix tree construction with large alphabets*, in Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 1997, pp. 137–143.
3. G. Fici: *A classification of trapezoidal words*, in Proc. 8th International Conference Words 2011 (WORDS 2011), Electronic Proceedings in Theoretical Computer Science 63, 2011, pp. 129–137, See also `http://arxiv.org/abs/1108.3629v1`.
4. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
5. U. Manber and G. W. Myers: *Suffix arrays: a new method for on-line string searches*. SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
6. G. Navarro and Y. Neckrich: Personal Communication.
7. G. Navarro and Y. Neckrich: *Sorted range reporting*, in Proc. Scandinavian Workshop on Algorithm Theory, LNCS 7357, Springer, 2012, pp. 271–282.
8. P. Weiner: *Linear pattern matching*, in IEEE 14th Annual Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.
9. C.-C. Yu, W.-K. Hon, and B.-F. Wang: *Improved data structures for the orthogonal range successor problem*. Computational Geometry, 44(3) 2011, pp. 148–159.

Alternative Algorithms for Lyndon Factorization*

Sukhpal Singh Ghuman[1], Emanuele Giaquinta[2], and Jorma Tarhio[1]

[1] Department of Computer Science and Engineering, Aalto University
P.O.B. 15400, FI-00076 Aalto, Finland
{Sukhpal.Ghuman,Jorma.Tarhio}@aalto.fi
[2] Department of Computer Science, P.O.B. 68, FI-00014 University of Helsinki, Finland
Emanuele.Giaquinta@cs.helsinki.fi

Abstract. We present two variations of Duval's algorithm for computing the Lyndon factorization of a word. The first algorithm is designed for the case of small alphabets and is able to skip a significant portion of the characters of the string, for strings containing runs of the smallest character in the alphabet. Experimental results show that it is faster than Duval's original algorithm, more than ten times in the case of long DNA strings. The second algorithm computes, given a run-length encoded string R of length ρ, the Lyndon factorization of R in $O(\rho)$ time and constant space.

1 Introduction

Given two strings w and w', w' is a rotation of w if $w = uv$ and $w' = vu$, for some strings u and v. A string is a Lyndon word if it is lexicographically smaller than all its proper rotations. Every string has a unique factorization in Lyndon words such that the corresponding sequence of factors is nonincreasing with respect to lexicographical order. This factorization was introduced by Chen, Fox and Lyndon [2]. Duval's classical algorithm [3] computes the factorization in linear time and constant space. The Lyndon factorization is a key ingredient in a recent method for sorting the suffixes of a text [8], which is a fundamental step in the construction of the Burrows-Wheeler transform and of the suffix array, as well as in the bijective variant of the Burrows-Wheeler transform [4] [6]. The Burrows-Wheeler transform is an invertible transformation of a string, based on the sorting of its rotations, while the suffix array is a lexicographically sorted array of the suffixes of a string. They are the basis for important data compression methods and text indexes. Although Duval's algorithm runs in linear time and is thus efficient, it can still be useful to further improve the time for the computation of the Lyndon factorization in the cases where the string is either huge or compressible and given in a compressed form.

Various alternative algorithms for the Lyndon factorization have been proposed in the last twenty years. Apostolico and Crochemore presented a parallel algorithm [1], while Roh *et al.* described an external memory algorithm [10]. Recently, I *et al.* showed how to compute the Lyndon factorization of a string given in grammar-compressed form and in Lempel-Ziv 78 encoding [5].

In this paper, we present two variations of Duval's algorithm. The first variation is designed for the case of small alphabets like the DNA alphabet {a, c, g, t}. If the string contains runs of the smallest character, the algorithm is able to skip a significant portion of the characters of the string. In our experiments, the new algorithm is more than ten times faster than the original one for long DNA strings.

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

Sukhpal Singh Ghuman, Emanuele Giaquinta, Jorma Tarhio: Alternative Algorithms for Lyndon Factorization, pp. 169–178.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

The second variation is for strings compressed with run-length encoding. The run-length encoding of a string is a simple encoding where each maximal consecutive sequence of the same symbol is encoded as a pair consisting of the symbol plus the length of the sequence. Given a run-length encoded string R of length ρ, our algorithm computes the Lyndon factorization of R in $O(\rho)$ time and uses constant space. It is thus preferable to Duval's algorithm in the cases in which the strings are stored or maintained in run-length encoding.

2 Basic definitions

Let Σ be a finite ordered alphabet of symbols and let Σ^* be the set of words (strings) over Σ ordered by lexicographic order. The empty word ε is a word of length 0. Let also Σ^+ be equal to $\Sigma^* \setminus \{\varepsilon\}$. Given a word w, we denote with $|w|$ the length of w and with $w[i]$ the i-th symbol of w, for $0 \leq i < |w|$. The concatenation of two words u and v is denoted by uv. Given two words u and v, v is a substring of u if there are indices $0 \leq i, j < |u|$ such that $v = u[i] \cdots u[j]$. If $i = 0$ ($j = |u| - 1$) then v is a prefix (suffix) of u. We denote by $u[i..j]$ the substring of u starting at position i and ending at position j. For $i > j$ $u[i..j] = \varepsilon$. We denote by u^k the concatenation of k u's, for $u \in \Sigma^+$ and $k \geq 1$. The longest border of a word w, denoted with $\beta(w)$, is the longest proper prefix of w which is also a suffix of w. Let $lcp(w, w')$ denote the length of the longest common prefix of words w and w'. We write $w < w'$ if either $lcp(w, w') = |w| < |w'|$, i.e., if w is a proper prefix of w', or if $w[lcp(w, w')] < w'[lcp(w, w')]$. For any $0 \leq i < |w|$, $\textsc{rot}(w, i) = w[i..|w| - 1]w[0..i - 1]$ is a rotation of w. A Lyndon word is a word w such that $w < \textsc{rot}(w, i)$, for $1 \leq i < |w|$. Given a Lyndon word w, the following properties hold:

1. $|\beta(w)| = 0$;
2. either $|w| = 1$ or $w[0] < w[|w| - 1]$.

Both properties imply that no word a^k, for $a \in \Sigma$, $k \geq 2$, is a Lyndon word. The following result is due to Chen, Fox and Lyndon [7]:

Theorem 1. *Any word w admits a unique factorization $CFL(w) = w_1, w_2, \ldots, w_m$, such that w_i is a Lyndon word, for $1 \leq i \leq m$, and $w_1 \geq w_2 \geq \cdots \geq w_m$.*

The run-length encoding (RLE) of a word w, denoted by $\textsc{rle}(w)$, is a sequence of pairs (runs) $\langle (c_1, l_1), (c_2, l_2,), \ldots, (c_\rho, l_\rho) \rangle$ such that $c_i \in \Sigma$, $l_i \geq 1$, $c_i \neq c_{i+1}$ for $1 \leq i < r$, and $w = c_1^{l_1} c_2^{l_2} \cdots c_\rho^{l_\rho}$. The interval of positions in w of the factor w_i in the Lyndon factorization of w is $[a_i, b_i]$, where $a_i = \sum_{j=1}^{i-1} |w_j|$, $b_i = \sum_{j=1}^{i} |w_j| - 1$. Similarly, the interval of positions in w of the run (c_i, l_i) is $[a_i^{rle}, b_i^{rle}]$ where $a_i^{rle} = \sum_{j=1}^{i-1} l_j$, $b_i^{rle} = \sum_{j=1}^{i} l_j - 1$.

3 Duval's algorithm

In this section we briefly describe Duval's algorithm for the computation of the Lyndon factorization of a word. Let L be the set of Lyndon words and let

$$P = \{w \mid w \in \Sigma^+ \text{ and } w\Sigma^* \cap L \neq \emptyset\},$$

be the set of nonempty prefixes of Lyndon words. Let also $P' = P \cup \{c^k \mid k \geq 2\}$, where c is the maximum symbol in Σ. Duval's algorithm is based on the following Lemmas, proved in [3]:

```
LF-DUVAL(w)
 1. k ← 0
 2. while k < |w| do
 3.         i ← k + 1
 4.         j ← k + 2
 5.         while TRUE do
 6.             if j = |w| + 1 or w[j − 1] < w[i − 1] then
 7.                 while k < i do
 8.                     output(w[k..k + j − i])
 9.                     k ← k + j − i
10.                 break
11.             else
12.                 if w[j − 1] > w[i − 1] then
13.                     i ← k + 1
14.                 else
15.                     i ← i + 1
16.                 j ← j + 1
```

Figure 1. Duval's algorithm to compute the Lyndon factorization of a string.

Lemma 2. *Let $w \in \Sigma^+$ and w_1 be the longest prefix of $w = w_1 w'$ which is in L. We have $CFL(w) = w_1 CFL(w')$.*

Lemma 3. *$P' = \{(uv)^k u \mid u \in \Sigma^*, v \in \Sigma^+, k \geq 1$ and $uv \in L\}$.*

Lemma 4. *Let $w = (uav')^k u$, with $u, v' \in \Sigma^*$, $a \in \Sigma$, $k \geq 1$ and $uav' \in L$. The following propositions hold:*

1. *For $a' \in \Sigma$ and $a > a'$, $wa' \notin P'$;*
2. *For $a' \in \Sigma$ and $a < a'$, $wa' \in L$;*
3. *For $a' = a$, $wa' \in P' \setminus L$.*

Lemma 2 states that the computation of the Lyndon factorization of a word w can be carried out by computing the longest prefix w_1 of $w = w_1 w'$ which is a Lyndon word and then recursively restarting the process from w'. Lemma 3 states that the nonempty prefixes of Lyndon words are all of the form $(uv)^k u$, where $u \in \Sigma^*, v \in \Sigma^+, k \geq 1$ and $uv \in L$. By the first property of Lyndon words, the longest prefix of $(uv)^k u$ which is in L is uv. Hence, if we know that $w = (uv)^k uav'$, $(uv)^k u \in P'$ but $(uv)^k ua \notin P'$, then by Lemma 2 and by induction we have $CFL(w) = w_1 w_2 \cdots w_k \, CFL(uav')$, where $w_1 = w_2 = \cdots = w_k = uv$. Finally, Lemma 4 explains how to compute, given a word $w \in P'$ and a symbol $a \in \Sigma$, whether $wa \in P'$, and thus makes it possible to compute the factorization using a left to right parsing. Note that, given a word $w \in P'$ with $|\beta(w)| = i$, we have $w[0..|w| - i - 1] \in L$ and $w = (w[0..|w| - i - 1])^q w[0..r - 1]$ with $q = \lfloor \frac{|w|}{|w|-i} \rfloor$ and $r = |w| \mod (|w| - i)$. For example, if $w = abbabbab$, we have $|w| = 8$, $|\beta(w)| = 5$, $q = 2$, $r = 2$ and $w = (abb)^2 ab$. The code of Duval's algorithm is shown in Figure 1.

The following is an alternative formulation of Duval's algorithm by I *et al.* [5]:

Lemma 5. *Let $j > 0$ be any position of a string w such that $w < w[i..|w| - 1]$ for any $0 < i \leq j$ and $lcp(w, w[j..|w| - 1]) \geq 1$. Then, $w < w[k..|w| - 1]$ also holds for any $j < k \leq j + lcp(w, w[j..|w| - 1])$.*

```
LF-SKIP(w)
 1. e ← |w| − 1
 2. while e ≥ 0 and w[e] = c̄ do
 3.       e ← e − 1
 4. l ← |w| − 1 − e
 5. w ← w[0..e]
 6. s ← min Occ_{c̄c̄}(w) ∪ {|w|}
 7. LF-DUVAL(w[0..s − 1])
 8. r ← 0
 9. while s < |w| do
10.       w ← w[s..|w| − 1]
11.       while w[r] = c̄ do
12.             r ← r + 1
13.       s ← |w|
14.       k ← 1
15.       P ← {c̄^r c | c ≤ w[r]}
16.       j ← 0
17.       for i ∈ Occ_P(w) : i > j do
18.             h ← lcp(w, w[i..|w| − 1])
19.             if h = |w| − i or w[i + h] < w[h] then
20.                   s ← i
21.                   k ← 1 + ⌊h/s⌋
22.                   break
23.             j ← i + h
24.       for i ← 1 to k do
25.             output(w[0..s − 1])
26.       s ← s × k
27. for i ← 1 to l do
28.       output(c̄)
```

Figure 2. The algorithm to compute the Lyndon factorization that can potentially skip symbols.

Lemma 6. *Let w be a string with $CFL(w) = w_1, w_2, \ldots, w_m$. It holds that $|w_1| = \min\{j \mid w[j..|w|-1] < w\}$ and $w_1 = w_2 = \cdots = w_k = w[0..|w_1|-1]$, where $k = 1 + \lfloor lcp(w, w[|w_1|..|w|-1])/|w_1| \rfloor$.*

Based on these Lemmas, Duval's algorithm can be implemented by initializing $j \leftarrow 1$ and executing the following steps until w becomes ε: 1) compute $h \leftarrow lcp(w, w[j..|w|-1])$. 2) if $j+h < |w|$ and $w[h] < w[j+h]$ set $j \leftarrow j+h+1$; otherwise output $w[0..j-1]$ k times and set $w \leftarrow w[jk..|w|-1]$, where $k = 1 + \lfloor h/j \rfloor$, and set $j \leftarrow 1$.

4 Improved algorithm for small alphabets

Let w be a word over an alphabet Σ with $CFL(w) = w_1, w_2, \ldots, w_m$ and let $\bar{c}$ be the smallest symbol in Σ. Suppose that there exists $k \geq 2, i \geq 1$ such that $\bar{c}^k$ is a prefix of w_i. If the last symbol of w is not $\bar{c}$, then by Theorem 1 and by the properties of Lyndon words, $\bar{c}^k$ is a prefix of each of $w_{i+1}, w_{i+1}, \ldots, w_m$. This property can be exploited to devise an algorithm for Lyndon factorization that can potentially skip symbols. Our algorithm is based on the alternative formulation of Duval's algorithm by I *et al.*. Given a set of strings $\mathcal{P}$, let $Occ_{\mathcal{P}}(w)$ be the set of all (starting) positions in w corresponding to occurrences of the strings in $\mathcal{P}$. We start with the following Lemmas:

Lemma 7. *Let w be a word and let $s = \max\{i \mid w[i] > \bar{c}\} \cup \{-1\}$. Then, $CFL(w) = CFL(w[0..s])CFL(\bar{c}^{(|w|-1-s)})$.*

Proof. If $s = -1$ or $s = |w| - 1$ the Lemma plainly holds. Otherwise, Let w_i be the factor in $CFL(w)$ such that $s \in [a_i, b_i]$. To prove the claim we have to show that $b_i = s$. Suppose by contradiction that $s < b_i$, which implies $|w_i| \geq 2$. Then, $w_i[|w_i| - 1] = \bar{c}$, which contradicts the second property of Lyndon words. □

Lemma 8. *Let w be a word such that $\bar{c}\bar{c}$ occurs in it and let $s = \min Occ_{\{\bar{c}\bar{c}\}}(w)$. Then, we have $CFL(w) = CFL(w[0..s-1])CFL(w[s..|w|-1])$.*

Proof. Let w_i be the factor in $CFL(w)$ such that $s \in [a_i, b_i]$. To prove the claim we have to show that $a_i = s$. Suppose by contradiction that $s > a_i$, which implies $|w_i| \geq 2$. If $s = b_i$ then $w_i[|w_i|-1] = \bar{c}$, which contradicts the second property of Lyndon words. Otherwise, since w_i is a Lyndon word it must hold that $w_i < \textsc{rot}(w_i, s - a_i)$. This implies at least that $w_i[0] = w_i[1] = \bar{c}$, which contradicts the hypothesis that s is the smallest element in $Occ_{\{\bar{c}\bar{c}\}}(w)$. □

Lemma 9. *Let w be a word such that $w[0] = w[1] = \bar{c}$ and $w[|w| - 1] \neq \bar{c}$. Let r be the smallest position in w such that $w[r] \neq \bar{c}$. Note that $w[0..r-1] = \bar{c}^r$. Let also $\mathcal{P} = \{\bar{c}^r c \mid c \leq w[r]\}$. Then we have*

$$b_1 = \min\{s \in Occ_{\mathcal{P}}(w) \mid w[s..|w| - 1] < w\} \cup \{|w|\} - 1\,,$$

where b_1 is the ending position of factor w_1.

Proof. By Lemma 6 we have that $b_1 = \min\{s \mid w[s..|w| - 1] < w\} - 1$. Since $w[0..r - 1] = \bar{c}^r$ and $|w| \geq r + 1$, for any string v such that $v < w$ we must have that either $v[0..r] \in \mathcal{P}$, if $|v| \geq r+1$, or $v = \bar{c}^{|v|}$ otherwise. Since $w[|w| - 1] \neq \bar{c}$, the only position s that satisfies $w[s..|w| - 1] = \bar{c}^{|w|-s}$ is $|w|$, corresponding to the empty word. Hence,

$$\{s \mid w[s..|w| - 1] < w\} = \{s \in Occ_{\mathcal{P}}(w) \mid w[s..|w| - 1] < w\} \cup \{|w|\}$$

□

Based on these Lemmas, we can devise a faster factorization algorithm for words containing runs of $\bar{c}$. The key idea is that, using Lemma 9, it is possible to skip symbols in the computation of b_1, if a suitable string matching algorithm is used to compute $Occ_{\mathcal{P}}(w)$. W.l.o.g. we assume that the last symbol of w is different from $\bar{c}$. In the general case, by Lemma 7, we can reduce the factorization of w to the one of its longest prefix with last symbol different from $\bar{c}$, as the remaining suffix is a concatenation of $\bar{c}$ symbols, whose factorization is a sequence of factors equal to $\bar{c}$. Suppose that $\bar{c}\bar{c}$ occurs in w. By Lemma 8 we can split the factorization of w in $CFL(u)$ and $CFL(v)$ where $uv = w$ and $|u| = \min Occ_{\{\bar{c}\bar{c}\}}(w)$. The factorization of $CFL(u)$ can be computed using Duval's original algorithm.

Concerning v, let $r = \min\{i \mid v[i] \neq \bar{c}\}$. By definition $v[0] = v[1] = \bar{c}$ and $v[|v| - 1] \neq \bar{c}$, and we can apply Lemma 9 on v to find the ending position s of the first factor in $CFL(v)$. To this end, we have to find the position $\min\{i \in Occ_{\mathcal{P}}(v) \mid v[i..|v|-1] < v\}$, where $\mathcal{P} = \{\bar{c}^r c \mid c \leq v[r]\}$. For this purpose, we can use any algorithm for multiple string matching to iteratively compute $Occ_{\mathcal{P}}(v)$ until either a position i is found that satisfies $v[i..|v|-1] < v$ or we reach the end of the string. Let $h = lcp(v, v[i..|v|-1])$, for a given $i \in Occ_{\mathcal{P}}(v)$. Observe that $h \geq r$ and, if $v < v[i..|v|-1]$, then, by Lemma 5,

```
LF-RLE(R)
1. k ← 0
2. while k < |R| do
3.     (m, q) ← LF-RLE-NEXT(R, k)
4.     for i ← 1 to q do
5.         output (k, k + m − 1)
6.         k ← k + m
```

```
LF-RLE-NEXT(R = ⟨(c_1, l_1), ..., (c_ρ, l_ρ)⟩, k)
 1. i ← k
 2. j ← k + 1
 3. while TRUE do
 4.     if i > k and l_{j−1} < l_{i−1} then
 5.         z ← 1
 6.     else z ← 0
 7.     s ← i − z
 8.     if j = |R| or c_j < c_s or
 9.         (c_j = c_s and l_j > l_s and c_j < c_{s+1}) then
10.         return (j − i, ⌊(j − k − z)/(j − i)⌋)
11.     else
12.         if c_j > c_s or l_j > l_s then
13.             i ← k
14.         else
15.             i ← i + 1
16.         j ← j + 1
```

Figure 3. The algorithm to compute the Lyndon factorization of a run-length encoded string.

we do not need to verify the positions $i' \in Occ_{\mathcal{P}}(v)$ such that $i' \leq i + h$. Given that all the patterns in $\mathcal{P}$ differ in the last symbol only, we can express $\mathcal{P}$ more succinctly using a character class for the last symbol and match this pattern using a string matching algorithm that supports character classes, such as the algorithms based on bit-parallelism. In this respect, SBNDM2 [11], a variation of the BNDM algorithm [9] is an ideal choice, as it is sublinear on average. Instead of $\mathcal{P}$, it is naturally possible to search for $\bar{c}^r$, but that solution is slower in practice for small alphabets. Note that the same algorithm can also be used to compute $\min Occ_{\bar{c}\bar{c}}(w)$ in the first phase.

Let $h = lcp(v, v[s..|v| - 1])$ and $k = 1 + \lfloor h/s \rfloor$. Based on Lemma 6, the algorithm then outputs $v[0..s - 1]$ k times and iteratively applies the above method on $v' = v[sk..|v| - 1]$. It is not hard to verify that, if $v' \neq \varepsilon$, then $|v'| \geq r + 1$, $v'[0..r - 1] = \bar{c}$ and $v'[|v'| - 1] \neq \bar{c}$, and so Lemma 9 can be used on v'. The code of the algorithm is shown in Figure 2. The computation of the value $r' = \min\{i \mid v'[i] \neq \bar{c}\}$ for v' takes advantage of the fact that $v'[0..r - 1] = \bar{c}$, so as to avoid useless comparisons. If the the total time spent for the iteration over the sets $Occ_{\mathcal{P}}(v)$ is $O(|w|)$, the full algorithm has also linear time complexity in the worst case. To see why, it is enough to observe that the positions i for which the algorithm verifies if $v[i..|v| - 1] < v$ are a subset of the positions verified by the original algorithm.

5 Computing the Lyndon factorization of a run-length encoded string

In this section we present an algorithm to compute the Lyndon factorization of a string given in RLE form. The algorithm is based on Duval's original algorithm and on a combinatorial property between the Lyndon factorization of a string and its RLE, and has $O(\rho)$-time and $O(1)$-space complexity, where ρ is the length of the RLE. We start with the following Lemma:

Lemma 10. *Let w be a word over Σ and let $w_1, w_2, \ldots, w_m$ be its Lyndon factorization. For any $1 \leq i \leq |\text{RLE}(w)|$, let $1 \leq j, k \leq m$, $j \leq k$, such that $a_i^{rle} \in [a_j, b_j]$ and $b_i^{rle} \in [a_k, b_k]$. Then, either $j = k$ or $|w_j| = |w_k| = 1$.*

Proof. Suppose by contradiction that $j < k$ and either $|w_j| > 1$ or $|w_k| > 1$. By definition of j, k, we have $w_j \geq w_k$. Moreover, since both $[a_j, b_j]$ and $[a_k, b_k]$ overlap with $[a_i^{rle}, b_i^{rle}]$, we also have $w_j[|w_j| - 1] = w_k[0]$. If $|w_j| > 1$, then, by definition of w_j, we have $w_j[0] < w_j[|w_j| - 1] = w_k[0]$. Instead, if $|w_k| > 1$ and $|w_j| = 1$, we have that w_j is a prefix of w_k. Hence, in both cases we obtain $w_j < w_k$, which is a contradiction. □

The consequence of this Lemma is that a run of length l in the RLE is either contained in *one* factor of the Lyndon factorization, or it corresponds to l unit-length factors. Formally:

Corollary 11. *Let w be a word over Σ and let $w_1, w_2, \ldots, w_m$ be its Lyndon factorization. Then, for any $1 \leq i \leq |\text{RLE}(w)|$, either there exists w_j such that $[a_i^{rle}, b_i^{rle}]$ is contained in $[a_j, b_j]$ or there exist l_i factors $w_j, w_{j+1}, \ldots, w_{j+l_i-1}$ such that $|w_{j+k}| = 1$ and $a_{j+k} \in [a_i^{rle}, b_i^{rle}]$, for $0 \leq k < l_i$.*

This property can be exploited to obtain an algorithm for the Lyndon factorization that runs in $O(\rho)$ time. First, we introduce the following definition:

Definition 12. *A word w is a LR word if it is either a Lyndon word or it is equal to a^k, for some $a \in \Sigma$, $k \geq 2$. The LR factorization of a word w is the factorization in LR words obtained from the Lyndon factorization of w by merging in a single factor the maximal sequences of unit-length factors with the same symbol.*

For example, the LR factorization of *cctgccaa* is $\langle cctg, cc, aa\rangle$. Observe that this factorization is a (reversible) encoding of the Lyndon factorization. Moreover, in this encoding it holds that each run in the RLE is contained in one factor and thus the size of the LR factorization is $O(\rho)$. Let L' be the set of LR words. We now present the algorithm LF-RLE-NEXT(R, k) which computes, given an RLE sequence R and an integer k, the longest LR word in R starting at position k. Analogously to Duval's algorithm, it reads the RLE sequence from left to right maintaining two integers, j and ℓ, which satisfy the following invariant:

$$\begin{aligned} &c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} \in P'; \\ &\ell = \begin{cases} |\text{RLE}(\beta(c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}))| & \text{if } j - k > 1, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

The integer j, initialized to $k + 1$, is the index of the next run to read and is incremented at each iteration until either $j = |R|$ or $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} \notin P'$. The integer ℓ, initialized to 0, is the length in runs of the longest border of $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$, if $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$ spans at least two runs, and equal to 0 otherwise. For example, in the case of the word ab^2ab^2ab we have $\beta(ab^2ab^2ab) = ab^2ab$ and $\ell = 4$. Let $i = k + \ell$. In general, if $\ell > 0$, we have

$$\begin{aligned} &l_{j-1} \leq l_{i-1}, l_k \leq l_{j-\ell}, \\ &\beta(c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}) = c_k^{l_k} c_{k+1}^{l_{k+1}} \cdots c_{i-2}^{l_{i-2}} c_{i-1}^{l_{j-1}} = c_{j-\ell}^{l_k} c_{j-\ell+1}^{l_{j-\ell+1}} \cdots c_{j-2}^{l_{j-2}} c_{j-1}^{l_{j-1}}. \end{aligned}$$

Note that the longest border may not fully cover the last (first) run of the corresponding prefix (suffix). Such the case is for example for the word ab^2a^2b. However, since $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} \in P'$ it must hold that $l_{j-\ell} = l_k$, i.e., the first run of the suffix is fully covered. Let

$$z = \begin{cases} 1 & \text{if } \ell > 0 \wedge l_{j-1} < l_{i-1}, \\ 0 & \textit{otherwise.} \end{cases}$$

Informally, the integer z is equal to 1 if the longest border of $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$ does not fully cover the run (c_{i-1}, l_{i-1}). By 1 we have that $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}}$ can be written as $(uv)^q u$, where

$$q = \lfloor \tfrac{j-k-z}{j-i} \rfloor, r = z + (j-k-z) \mod (j-i),$$
$$u = c_{j-r}^{l_{j-r}} \cdots c_{j-1}^{l_{j-1}}, uv = c_k^{l_k} \cdots c_{j-\ell-1}^{l_{j-\ell-1}} = c_{i-r}^{l_{i-r}} \cdots c_{j-r-1}^{l_{j-r-1}},$$
$$uv \in L'$$

For example, in the case of the word ab^2ab^2ab, for $k = 0$, we have $j = 6, i = 4, q = 2, r = 2$. The algorithm is based on the following Lemma:

Lemma 13. *Let j, ℓ be such that invariant 1 holds and let $s = i - z$. Then, we have the following cases:*

1. *If $c_j < c_s$ then $c_k^{l_k} \cdots c_j^{l_j} \notin P'$;*
2. *If $c_j > c_s$ then $c_k^{l_k} \cdots c_j^{l_j} \in L'$ and 1 holds for $j+1$, $\ell' = 0$;*

Moreover, if $z = 0$, we also have:

3. *If $c_j = c_i$ and $l_j \leq l_i$, then $c_k^{l_k} \cdots c_j^{l_j} \in P'$ and 1 holds for $j+1$, $\ell' = \ell + 1$;*
4. *If $c_j = c_i$ and $l_j > l_i$, either $c_j < c_{i+1}$ and $c_k^{l_k} \cdots c_j^{l_j} \notin P'$ or $c_j > c_{i+1}$, $c_k^{l_k} \cdots c_j^{l_j} \in L'$ and 1 holds for $j+1$, $\ell' = 0$.*

Proof. The idea is the following: we apply Lemma 4 with the word $(uv)^q u$ as defined above and symbol c_j. Observe that c_j is compared with symbol $v[0]$, which is equal to $c_{k+r-1} = c_{i-1}$ if $z = 1$ and to $c_{k+r} = c_i$ otherwise.

First note that, if $z = 1$, $c_j \neq c_{i-1}$, since otherwise we would have $c_{j-1} = c_{i-1} = c_j$. In the first three cases, we obtain the first, second and third proposition of Lemma 4, respectively, for the word $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} c_j$. Independently of the derived proposition, it is easy to verify that the same proposition also holds for $c_k^{l_k} \cdots c_{j-1}^{l_{j-1}} c_j^m$, $m \leq l_j$. Consider now the fourth case. By a similar reasoning, we have that the third proposition of Lemma 4 holds for $c_k^{l_k} \cdots c_j^{l_i}$. If we then apply Lemma 4 to $c_k^{l_k} \cdots c_j^{l_i}$ and c_j, c_j is compared to c_{i+1} and we must have $c_j \neq c_{i+1}$ as otherwise $c_i = c_j = c_{i+1}$. Hence, either the first (if $c_j < c_{i+1}$) or the second (if $c_j > c_{i+1}$) proposition of Lemma 4 must hold for the word $c_k^{l_k} \cdots c_j^{l_i+1}$. □

We prove by induction that invariant 1 is maintained. At the beginning, the variables j and ℓ are initialized to $k+1$ and 0, respectively, so the base case trivially holds. Suppose that the invariant holds for j, ℓ. Then, by Lemma 13, either $c_k^{l_k} \cdots c_j^{l_j} \notin P'$ or it follows that the invariant also holds for $j+1, \ell'$, where ℓ' is equal to $\ell + 1$, if $z = 0$, $c_j = c_i$ and $l_j \leq l_i$, and to 0 otherwise. When $c_k^{l_k} \cdots c_j^{l_j} \notin P'$ the algorithm returns the pair $(j - i, q)$, i.e., the length of uv and the corresponding exponent. Based on

Lemma 2, the factorization of R can then be computed by iteratively calling LF-RLE-NEXT. When a given call to LF-RLE-NEXT returns, the factorization algorithm outputs the q factors uv starting at positions k, $k+(j-i)$, $\ldots$, $k+(q-1)(j-i)$ and restarts the factorization at position $k+q(j-i)$. The code of the algorithm is shown in Figure 3. We now prove that the algorithm runs in $O(\rho)$ time. First, observe that, by definition of LR factorization, the for loop at line 4 is executed $O(\rho)$ times. Suppose that the number of iterations of the while loop at line 2 is n and let $k_1, k_2, \ldots, k_{n+1}$ be the corresponding values of k, with $k_1 = 0$ and $k_{n+1} = |R|$. We now show that the s-th call to LF-RLE-NEXT performs less than $2(k_{s+1} - k_s)$ iterations, which will yield $O(\rho)$ number of iterations in total. This analysis is analogous to the one used by Duval. Suppose that i', j' and z' are the values of i, j and z at the end of the s-th call to LF-RLE-NEXT. The number of iterations performed during this call is equal to $j' - k_s$. We have $k_{s+1} = k_s + q(j' - i')$, where $q = \lfloor \frac{j'-k_s-z}{j-i'} \rfloor$, which implies $j' - k_s < 2(k_{s+1} - k_s) + 1$, since, for any positive integers x, y, $x < 2\lfloor x/y \rfloor y$ holds.

6 Experiments with LF-Skip

The experiments were run on MacBook Pro with the 2.4 GHz Intel Core 2 Duo processor and 2 GB memory. Programs were written in the C programming language and compiled with the gcc compiler (4.8.2) using the -O3 optimization level.

We tested the LF-skip algorithm and Duval's algorithm with various texts. With the protein sequence of the Saccharomyces cerevisiae genome (3 MB), LF-skip gave a speed-up of 3.5 times over Duval's algorithm. Table 1 shows the speed-ups for random texts of 5 MB with various alphabets sizes. With longer texts, speed-ups were larger. For example, the speed-up for the 50 MB DNA text (without newlines) from the Pizza&Chili Corpus[1] was 14.6 times.

$\|\Sigma\|$	Speed-up
2	9.0
3	7.7
4	7.2
5	6.1
6	4.8
8	4.3
10	3.5
12	3.4
15	2.4
20	2.5
25	2.2
30	1.9

Table 1. Speed-up of LF-skip with various alphabet sizes in a random text.

We made also some tests with texts of natural language. Because runs are very short in natural language, the benefit of LF-skip is marginal. We even tried alphabet transformations in order to vary the smallest character of the text, but that did not help.

[1] http://pizzachili.dcc.uchile.cl/

7 Conclusions

In this paper we have presented two variations of Duval's algorithm for computing the Lyndon factorization of a string. The first algorithm was designed for the case of small alphabets and is able to skip a significant portion of the characters of the string for strings containing runs of the smallest character in the alphabet. Experimental results show that the algorithm is considerably faster than Duval's original algorithm. The second algorithm is for strings compressed with run-length encoding and computes the Lyndon factorization of a run-length encoded string of length ρ in $O(\rho)$ time and constant space.

References

1. A. Apostolico and M. Crochemore: *Fast parallel Lyndon factorization with applications.* Mathematical Systems Theory, 28(2) 1995, pp. 89–108.
2. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. IV. The quotient groups of the lower central series.* Annals of Mathematics, 68(1) 1958, pp. 81–95.
3. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
4. J. Y. Gil and D. A. Scott: *A bijective string sorting transform.* CoRR, abs/1201.3077 2012.
5. T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Faster lyndon factorization algorithms for SLP and LZ78 compressed text*, in SPIRE, O. Kurland, M. Lewenstein, and E. Porat, eds., vol. 8214 of Lecture Notes in Computer Science, Springer, 2013, pp. 174–185.
6. M. Kufleitner: *On bijective variants of the Burrows-Wheeler transform*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., 2009, pp. 65–79.
7. M. Lothaire: *Combinatorics on Words*, Cambridge Mathematical Library, Cambridge University Press, 1997.
8. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *Sorting suffixes of a text via its Lyndon factorization*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žďárek, eds., 2013, pp. 119–127.
9. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM Journal of Experimental Algorithmics, 5 2000.
10. K. Roh, M. Crochemore, C. S. Iliopoulos, and K. Park: *External memory algorithms for string problems.* Fundam. Inform., 84(1) 2008, pp. 17–32.
11. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.

Two Simple Full-Text Indexes Based on the Suffix Array

Szymon Grabowski and Marcin Raniszewski

Lodz University of Technology, Institute of Applied Computer Science,
Al. Politechniki 11, 90–924 Łódź, Poland
{sgrabow|mranisz}@kis.p.lodz.pl

Abstract. We propose two suffix array inspired full-text indexes. One, called SA-hash, augments the suffix array with a hash table to speed up pattern searches due to significantly narrowed search interval before the binary search phase. The other, called FBCSA, is a compact data structure, similar to Mäkinen's compact suffix array, but working on fixed sized blocks, which allows to arrange the data in multiples of 32 bits, beneficial for CPU access. Experimental results on the Pizza & Chili 200 MB datasets show that SA-hash is about 2.5–3 times faster in pattern searches (counts) than the standard suffix array, for the price of requiring $0.3n - 2.0n$ bytes of extra space, where n is the text length, and setting a minimum pattern length. The latter limitation can be removed for the price of even more extra space. FBCSA is relatively fast in single cell accesses (a few times faster than related indexes at about the same or better compression), but not competitive if many consecutive cells are to be extracted. Still, for the task of extracting e.g. 10 successive cells its time-space relation remains attractive.

Keywords: suffix array, compressed indexes, compact indexes, hashing

1 Introduction

The field of text-oriented data structures continues to bloom. Curiously, in many cases several years after ingenious theoretical solutions their more practical (which means: faster and/or simpler) counterparts are presented, to mention only recent advances in rank/select implementations [11] or the FM-index reaching the compression ratio bounded by k-th order entropy with very simple means [17].

Despite the great interest in compact or compressed[1] full-text indexes in recent years [22], we believe that in some applications search speed is more important than memory savings, thus different space-time tradeoffs are worth being explored. The classic suffix array (SA) [21], combining speed, simplicity and often reasonable memory use, may be a good starting point for such research.

In this paper we present two SA-based full-text indexes, combining effectiveness and simplicity. One augments the standard SA with a hash table to speed up searches, for a moderate overhead in the memory use, the other is a byte-aligned variant of Mäkinen's compact suffix array [19,20].

[1] By the latter we mean indexes with space use bounded by $O(nH_0)$ or even $O(nH_k)$ bits, where n is the text length, σ the alphabet size, and H_0 and H_k respectively the order-0 and the order-k entropy. The former term, compact full-text indexes, is less definite, and, roughly speaking, may fit any structure with less than $n \log_2 n$ bits of space, at least for "typical" texts.

Szymon Grabowski, Marcin Raniszewski: Two Simple Full-Text Indexes Based on the Suffix Array, pp. 179–191.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

2 Preliminaries

We use 0-based sequence notation, that is, a sequence S of length n is written as $S[0 \ldots n-1]$, or equivalently as $s_0 s_1 \cdots s_{n-1}$.

One may define a *full-text index* over text T of length n as a data structure supporting at least two basic types of queries, both with respect to a pattern P of length m, both T and P over a common finite integer alphabet of size σ. One query type is *count*: return the number $occ \geq 0$ of occurrences of P in T. The other query type is *locate*: for each pattern occurrence report its position in T, that is, such j that $P[0 \ldots m-1] = T[j \ldots j+m-1]$.

The *suffix array* $SA[0 \ldots n-1]$ for text T is a permutation of the indexes $\{0, 1, \ldots, n-1\}$ such that $T[SA[i] \ldots n-1] \prec T[SA[i+1] \ldots n-1]$ for all $0 \leq i < n-1$, where the "$\prec$" relation is the lexicographical order. The inverse suffix array SA^{-1} is the inverse permutation of SA, that is, $SA^{-1}[j] = i \Leftrightarrow SA[i] = j$.

If not stated otherwise, all logarithms throughout the paper are in base 2.

3 Related work

The full-text indexing history starts with the *suffix tree* (ST) [25], a trie whose string collection is the set of all the suffixes of a given text, with an additional requirement that all non-branching paths of edges are converted into single edges. The structure can be built in linear time [25,3]. Assuming constant-time access to any child of a given node, the search in the ST takes only $O(m + occ)$ time in the worst case. In practice, this is cumbersome for a large alphabet, as it requires using perfect hashing, which also makes the construction time linear only in expectation. A small alphabet is easier to handle, which goes in line with the wide use of the suffix tree in bioinformatics.

The main problem with the suffix tree is its large space requirement. Even in the most economical version [18] the ST space use reaches almost $9n$ bytes on average and $16n$ in the worst case, plus the text, for $\sigma \leq 256$, and even more for large alphabets. Most implementations need $20n$ bytes or more.

An important alternative to the suffix tree is the *suffix array* (SA) [21]. It is an array of n pointers to text suffixes arranged in the order of lexicographic ordering of the sequences (i.e., the suffixes) the pointers store references to. The SA needs $n \log n$ bits for its n suffix pointers (indexes), plus $n \log \sigma$ bits for the text, which typically translates to $5n$ bytes in total. The pattern search time is $O(m \log n)$ in the worst case and $O(m \log_\sigma n + \log n)$ on average, which can be improved to $O(m + \log n)$ in the worst case using the longest common prefix (lcp) table. Alternatively, the $O(m+\log n)$ time can be reached even without the lcp, in a more theoretical solution with a specific suffix permutation [8]. Yet Manber and Myers in their seminal paper [21] presented a nice trick saving several first steps in the binary search: if we know the SA intervals for all the possible first k symbols of the pattern, we can immediately start the binary search in a corresponding interval. We can set k close to $\log_\sigma n$, with $O(n \log n)$ extra bits of space, but constant expected size of the interval, which leads to $O(m)$ average search time and only $O(\lceil m/|cache\_line| \rceil)$ cache misses on average, where $|cache\_line|$ is the cache line length expressed in symbols, typically 64 symbols / bytes in a modern CPU. Unfortunately, real texts are far from random, hence in practice, if text symbols are bytes, we can use k up to 3, which offers a limited (yet, non-negligible) benefit. This idea, later denoted as using a lookup table (LUT), is fairly well known, see e.g. its impact in the search over a suffix array on words [4].

A number of suffix tree or suffix array inspired indexes have been proposed as well, including the suffix cactus [16] and the enhanced suffix array (ESA) [1], with space use usually between SA and ST, but according to our knowledge they generally are not faster than their famous predecessors in the count or locate queries.

On a theoretical front, the suffix tray by Cole et al. [2] allows to achieve $O(m + \log \sigma)$ search time (with $O(n)$ worst-case time construction and $O(n \log n)$ bits of space), which was recently improved by Fischer and Gawrychowski [7] to $O(m + \log \log \sigma)$ deterministic time, with preserved construction cost complexities.

The common wisdom about the practical performance of ST and SA is that they are comparable, but Grimsmo in his interesting experimental work [14] showed that a careful ST implementation may be up to about 50% faster than SA if the number of matches is very small (in particular, one hit), but if the number of hits grows, the SA becomes more competitive, sometimes being even about an order of magnitude faster. Another conclusion from Grimsmo's experiments is that the ESA may also be moderately faster than SA if the alphabet is small (say, up to 8) but SA easily wins for a large alphabet.

Since around 2000 we can witness a great interest in succinct data structures, in particular, text indexes. Two main ideas that deserve being mentioned are the compressed suffix array (CSA) [15,24] and the FM-index [6]; the reader is referred to the survey [22] for an extensive coverage of the area.

It was noticed in extensive experimental comparisons [5,11] that compressed indexes are not much slower, and sometimes comparable, to the suffix array in count queries, but locate is 2–3 orders of magnitude slower if the number of matches is large. This instigated researchers to follow one of two paths in order to mitigate the locate cost for succinct indexes. One, pioneered by Mäkinen [19,20] and addressed in a different way by González et al. [12,13], exploits repetitions in the suffix array (the idea is explained in Section 5). The other approach is to build semi-external data structures (see [9,10] and references therein).

4 Suffix array with deep buckets

The mentioned idea of Manber and Myers with precomputed interval (bucket) boundaries for k starting symbols tends to bring more gain with growing k, but also precomputing costs grow exponentially. Obviously, σ^k integers are needed to be kept in the lookup table. Our proposal is to apply hashing on relatively long strings, with an extra trick to reduce the number of unnecessary references to the text.

We start with building the hash table HT (Fig. 1). The hash function is calculated for the *distinct* k-symbol ($k \geq 2$) prefixes of suffixes from the (previously built) suffix array. That is, we process the suffixes in their SA order and if the current suffix shares its k-long prefix with its predecessor, it is skipped (line 08). The value written to HT (line 11) is a pair: (the position in the SA of the first suffix with the given prefix, the position in the SA of the last suffix with the given prefix). Linear probing is used as the collision resolution method. As for the hash function, we used sdbm (`http://www.cse.yorku.ca/~oz/hash.html`).

Fig. 2 presents the pattern search (locate) procedure. It is assumed that the pattern length m is not less than k. First the range of rows in the suffix array corresponding to the first two symbols of the pattern is found in a "standard" lookup table (line 1); an empty range immediately terminates the search with no matches returned (line 2). Then, the hash function over the pattern prefix is calculated and a scan over

HT_build($T[0 \dots n-1]$, $SA[0 \dots n-1]$, k, z, $h(.)$)
Precondition: $k \geq 2$

```
(01)  allocate HT[0 ... z − 1]
(02)  for j ← 0 to z − 1 do HT[j] ← NIL
(03)  prevStr ← ε
(04)  j ← NIL
(05)  left ← NIL; right ← NIL
(06)  for i ← 0 to n − 1 do
(07)     if SA[i] ≥ n − k then continue
(08)     if T[SA[i] ... SA[i] + k − 1] ≠ prevStr then
(09)           if j ≠ NIL then
(10)                right ← i − 1
(11)                HT[j] ← (left, right)
(12)           j ← h(T[SA[i] ... SA[i] + k − 1])
(13)           prevStr ← T[SA[i] ... SA[i] + k − 1]
(14)           repeat
(15)                if HT[j] = NIL then
(16)                     left ← i
(17)                     break
(18)                else j ← (j + 1) % z
(19)           until false
(20)  HT[j] ← (right + 1, n − 1)
(21)  return HT
```

Figure 1. Building the hash table of a given size z

the hash table performed until no extra collisions (line 5; return no matches) or found a match over the pattern prefix, which give us information about the range of suffixes starting with the current prefix (line 6). In this case, the binary search strategy is applied to narrow down the SA interval to contain exactly the suffixes starting with the whole pattern. (As an implementation note: the binary search could be modified to ignore the first k symbols in the comparisons, but it did not help in our experiments, due to specifics of the used A_strcmp function from the asmlib library[2]).

Pattern_search($T[0 \dots n-1]$, $SA[0 \dots n-1]$, $HT[0 \dots z-1]$, k, $h(.)$, $P[0 \dots m-1]$)
Precondition: $m \geq k \geq 2$

```
(1)  beg, end ← LUT2[p0, p1]
(2)  if end < beg then report no matches; return
(3)  j ← h(P[0 ... k − 1])
(4)  repeat
(5)     if HT[j] = NIL then report no matches; return
(6)     if (beg ≤ HT[j].left ≤ end) and (T[SA[HT[j].left] ... SA[HT[j].left] + k − 1] = P[0 ... k − 1])
(7)          then binSearch(P[0 ... m − 1], HT[j].left, HT[j].right); return
(8)     j ← (j + 1) % z
(9)  until false
```

Figure 2. Pattern search

[2] `http://www.agner.org/optimize/asmlib.zip`, v2.34, by Agner Fog.

5 Fixed Block based Compact Suffix Array

We propose a variant of Mäkinen's compact suffix array [19,20], whose key feature is finding repeating suffix areas of fixed size, e.g., 32 bytes. This allows to maintain a byte aligned data layout, beneficial for speed and simplicity. Even more, by setting a natural restriction on one of the key parameters we force the structure's building bricks to be multiples of 32 bits, which prevents misaligned access to data.

Mäkinen's index was the first *opportunistic* scheme for compressing a suffix array, that is such that uses less space on compressible texts. The key idea was to exploit runs in the SA, that is, maximal segments $SA[i \dots i+\ell-1]$ for which there exists another segment $SA[j \dots j+\ell-1]$, such that $SA[j+s] = SA[i+s]+1$ for all $0 \le s < \ell$. This structure still allows for binary search, only the accesses to SA cells require local decompression.

FBCSA_build($SA[0 \dots n-1]$, T^{BWT}, bs, ss)

```
/* assume n is a multiple of bs */
(01)  arr1 ← [ ]; arr2 ← [ ]
(02)  j ← 0
(03)  repeat
        /* current block of the suffix array is SA[j ... j + bs − 1] */
(04)    find 3 most frequent symbols in T^BWT[j ... j + bs − 1] and store them in MFS[0 ... 2]
            /* if there are less than 3 distinct symbols in T^BWT[j ... j + bs − 1],
                 the trailing cells of MFS[0 ... 2] are set to NIL) */
(05)    for i ← 0 to bs − 1 do
(06)        if T^BWT[j + i] = MFS[0] then arr1.append(00)
(07)        else if T^BWT[j + i] = MFS[1] then arr1.append(01)
(08)            else if T^BWT[j + i] = MFS[2] then arr1.append(10)
(09)                else arr1.append(11)
(10)    pos0 = T^BWT[j ... j + bs − 1].pos(MFS[0])
(11)    pos1 = T^BWT[j ... j + bs − 1].pos(MFS[1]) /* set NIL if MFS[1] = NIL */
(12)    pos2 = T^BWT[j ... j + bs − 1].pos(MFS[2]) /* set NIL if MFS[2] = NIL */
(13)    a2s = |arr2|
(14)    arr2.append(SA^−1[SA[j + pos0] − 1])
(15)    arr2.append(SA^−1[SA[j + pos1] − 1]) /* append −1 if pos1 = NIL */
(16)    arr2.append(SA^−1[SA[j + pos2] − 1]) /* append −1 if pos2 = NIL */
(17)    for i ← 0 to bs − 1 do
(18)        if (T^BWT[j + i] ∉ {MFS[0], MFS[1], MFS[2]}) or (SA[j + i] % ss = 0) then
(19)            arr1.append(1); arr2.append(SA[j + i])
(20)        else arr1.append(0)
(21)    arr1.append(a2s)
(22)    j ← j + bs
(23)    if j = n then break
(24)  until false
(25)  return (arr1, arr2)
```

Figure 3. Building the fixed block based compact suffix array (FBCSA)

We resign from *maximal* segments in our proposal. The construction algorithm for our structure, called *fixed block based compact suffix array* (FBCSA), is presented in Fig. 3. As a result, we obtain two arrays, arr_1 and arr_2, which are empty at the beginning, and their elements are always appended at the end during the construction.

The elements appended to arr_1 are single bits or pairs of bits while arr_2 stores suffix array indexes (32-bit integers).

The construction makes use of the suffix array SA of text T, the inverse suffix array SA^{-1} and T^{BWT} (which can be obtained from T and SA, that is, $T^{BWT}[i] = T[(SA[i] - 1) \mod n]$).

Additionally, there are two construction-time parameters: block size bs and sampling step ss. The block size tells how many successive SA indexes are encoded together and is assumed to be a multiple of 32, for int32-alignment of the structure layout. The parameter ss means that every ss-th SA index will be represented verbatim. This sampling parameter is a time-space tradeoff; using larger ss reduces the overall space but decoding a particular SA index typically involves more recursive invocations.

Let us describe the encoding procedure for one block, $SA[j \dots j + bs - 1]$, where j is a multiple of bs.

First we find the three most frequent symbols in $T^{BWT}[j \dots j + bs - 1]$ and store them (in arbitrary order) in a small helper array $MFS[0 \dots 2]$ (line 04). If the current block of T^{BWT} does not contain three different symbols, the NIL value will be written in the last one or two cell(s) of MFS. Then we write information about the symbols from MFS in the current block of T^{BWT} into arr_1: we append 2-bit combination (00, 01 or 10) if a given symbol is from MFS and the remaining combination (11) otherwise (lines 05–09). We also store the positions of the first occurrences of the symbols from MFS in the current block of T^{BWT}, using the variables pos_0, pos_1, pos_2 (lines 10–12); again NIL values are used if needed. These positions allow to use links to runs of suffixes preceding subsets of the current ones marked by the respective symbols from MFS.

We believe that a small example will be useful here. Let $bs = 8$ and the current block be $SA[400 \dots 407]$ (note this is a toy example and in the real implementation bs must be a multiple of 32). The SA block contains the indexes: 1000, 522, 801, 303, 906, 477, 52, 610. Let their preceding symbols (from T^{BWT}) be: a, b, a, c, d, d, b, b. The three most frequent symbols, written to MFS, are thus: b, a, d. The first occurrences of these symbols are at positions: 401, 400 and 404, respectively (that is, $400 + pos_0 = 401$, etc.). The SA offsets: 521 ($= 522 - 1$), 999 ($= 1000 - 1$) and 905 ($= 906 - 1$) will be linked to the current block. We conclude that the preceding groups of suffix offsets are: $[521, 522, 523]$ (as there are three symbols b in the current block of T^{BWT}), $[999, 1000]$ and $[905, 906]$.

We come back to the pseudocode. The described (up to three) links are obtained thanks to SA^{-1} (lines 14–16) and are written to arr_2. Finally, the offsets of the suffixes preceded with a symbol not from MFS (if any) have to be written to arr_2 explicitly. Additionally, the sampled suffixes (i.e., those whose offset modulo ss is 0) are handled in the same way (line 18). To distinguish between referentially encoded and explicitly written suffix offsets, we spent a bit per suffix and append them to arr_1 (lines 19–20). To allow for easy synchronization between the portions of data in arr_1 and arr_2, the size of arr_2 (in bytes) as it was before processing the current block is written to arr_1 (line 21).

6 Experimental results

All experiments were run on a laptop computer with an Intel i3 2.1 GHz CPU, equipped with 8 GB of DDR3 RAM and running Windows 7 Home Premium

SP1 64-bit. All codes were written in C++ and compiled with Microsoft Visual Studio 2010. The source codes for the FBCSA algorithm can be downloaded from `http://ranisz.iis.p.lodz.pl/indexes/fbcsa/`.

The test datasets were taken from the popular Pizza & Chili site (`http://pizzachili.dcc.uchile.cl/`). We used the 200-megabyte versions of the files `dna`, `english`, `proteins`, `sources` and `xml`. In order to test the search algorithms, we generated 500 thousand patterns for each used pattern length; the patterns were extracted randomly from the corresponding datasets (i.e., each pattern returns at least one match).

In the first experiment we compared pattern search (count) speed using the following indexes:

- plain suffix array (SA),
- suffix array with a lookup table over the first 2 symbols (SA-LUT2),
- suffix array with a lookup table over the first 3 symbols (SA-LUT3),
- the proposed suffix array with deep buckets, with hashing the prefixes of length $k = 8$ (only for `dna` $k = 12$ and for `proteins` $k = 5$ is used); the load factor α in the hash table was set to 50% (SA-hash),
- the proposed fixed block based compact suffix array with parameters $bs = 32$ and $ss = 5$ (FBCSA),
- FBCSA (parameters as before) with a lookup table over the first 2 symbols (FBCSA-LUT2),
- FBCSA (parameters as before) with a lookup table over the first 3 symbols (FBCSA-LUT3),
- FBCSA (parameters as before) with a hashes of prefixes of length $k = 8$ (only for `dna` $k = 12$ and for `proteins` $k = 5$ is used); the load factor in the hash table was set to 50% (FBCSA-hash).

The results are presented in Fig. 4. As expected, SA-hash is the fastest index among the tested ones. The reader may also look at Table 1 with a rundown of the achieved speedups, where the plain suffix array is the baseline index and its speed is denoted with 1.00.

	`dna`	`english`	`proteins`	`sources`	`xml`
$m = 16$					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.13	1.34	1.36	1.43	1.35
SA-LUT3	1.17	1.49	1.61	1.65	1.47
SA-hash	3.75	2.88	2.70	2.90	2.03
$m = 64$					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.12	1.33	1.34	1.42	1.34
SA-LUT3	1.17	1.49	1.58	1.64	1.44
SA-hash	3.81	2.87	2.62	2.75	1.79

Table 1. Speedups with regard to the search speed of the plain suffix array, for the five datasets and pattern lengths $m = 16$ and $m = 64$

The SA-hash index has two drawbacks: it requires significantly more space than the standard SA and we assume (at construction time) a minimal pattern length m_{min}. The latter issue may be eliminated, but for the price of even more space use; namely,

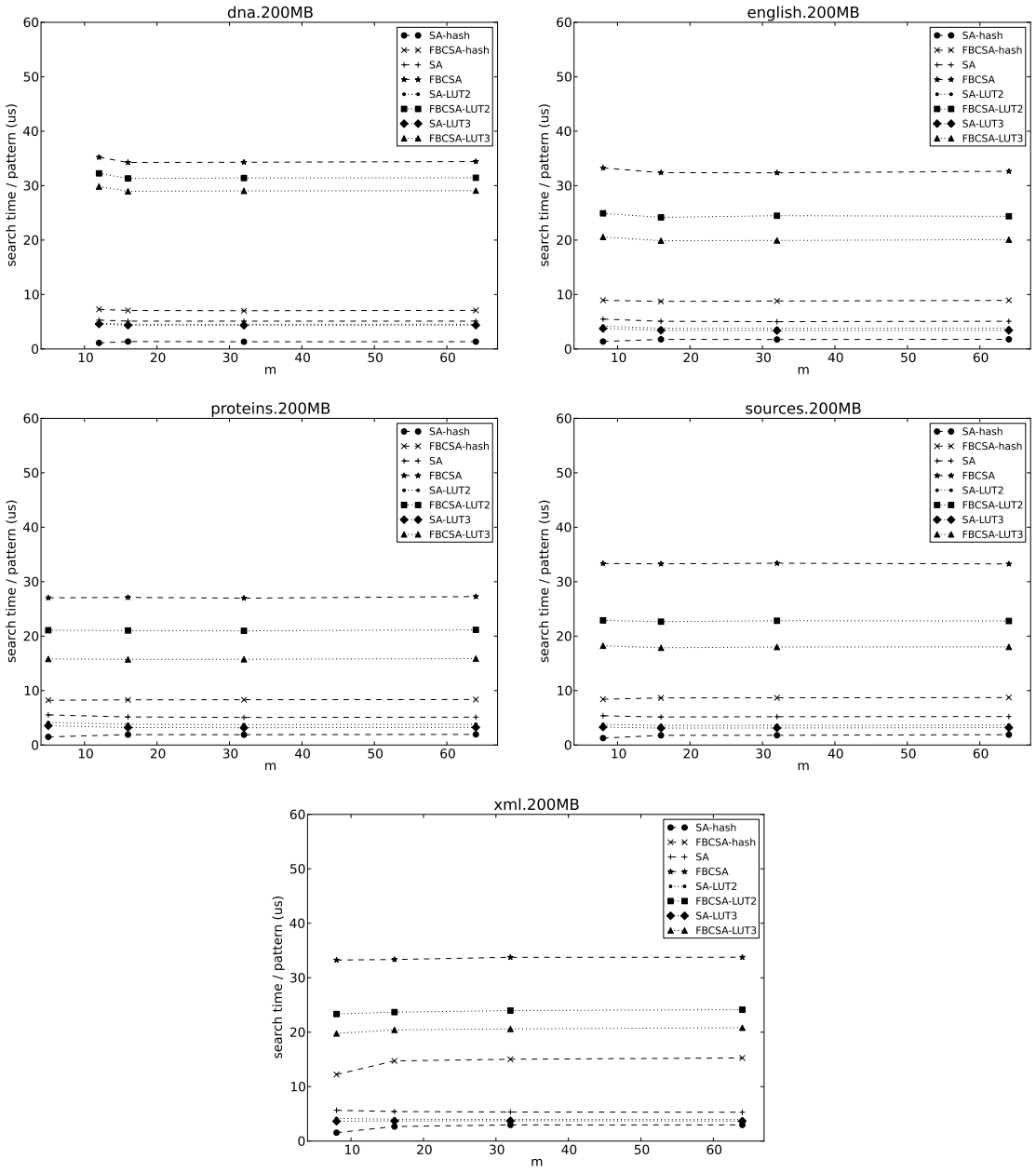


Figure 4. Pattern search time (count query). All times are averages over 500K random patterns of the same length $m = \{m_{min}, 16, 32, 64\}$, where m_{min} is 8 for most datasets except for `dna` (12) and `proteins` (5). The patterns were extracted from the respective texts.

we can build one hash table for each pattern length from 1 to m_{min} (counting queries for those short patterns do not ever need to perform binary search over the suffix array). For the shortest lengths ($\{1, 2\}$ or $\{1, 2, 3\}$) lookup tables may be alternatively used.

We have not implemented this "all-HT" variant, but it is easy to estimate the memory use for each dataset. To this end, one needs to know the number of distinct q-grams for $q \leq m_{min}$ (Table 2).

q	dna	english	proteins	sources	xml
1	16	225	25	230	96
2	152	10,829	607	9,525	7,054
3	683	102,666	11,607	253,831	141,783
4	2,222	589,230	224,132	1,719,387	908,131
5	5,892	2,150,525	3,623,281	5,252,826	2,716,438
6	12,804	5,566,993	36,525,895	10,669,627	5,555,190
7	28,473	11,599,445	94,488,651	17,826,241	8,957,209
8	80,397	20,782,043	112,880,347	26,325,724	12,534,152
9	279,680	33,143,032	117,199,335	35,666,486	16,212,609
10	1,065,613	48,061,001	119,518,691	45,354,280	20,018,262

Table 2. The number of distinct q-grams ($1 \ldots 10$) in the datasets. The number of distinct 12-grams for dna is 13,752,341.

The number of bytes for one hash table with z entries and $0 < \alpha \leq 1$ load factor is, in our implementation, $z \times 8 \times (1/\alpha)$, since each entry contains two 4-byte integers. For example, in our experiments the hash table for english needed 20,782,043 $\times 16 =$ 332,512,688 bytes, i.e., 158.6% of the size of the text itself.

An obvious idea to reduce the HT space, in an open addressing scheme, is increasing its load factor α. The search times then are, however, likely to grow. We checked several values of α on two datasets (Table 3) to conclude that using $\alpha = 80\%$ may be a reasonable alternative to $\alpha = 50\%$, as the pattern search times grow by only about 10%.

	HT load factor (%)					
	25	50	60	70	80	90
dna, $m = 12$	1.088	1.111	1.122	1.172	1.214	1.390
dna, $m = 16$	1.359	1.362	1.389	1.421	1.491	1.668
dna, $m = 32$	1.320	1.347	1.360	1.391	1.463	1.662
dna, $m = 64$	1.345	1.394	1.409	1.428	1.491	1.672
english, $m = 8$	1.292	1.386	1.402	1.487	1.524	1.617
english, $m = 16$	1.670	1.761	1.781	1.846	1.892	1.998
english, $m = 32$	1.665	1.762	1.813	1.858	1.931	2.015
english, $m = 64$	1.714	1.794	1.829	1.869	1.967	2.039

Table 3. Average pattern search times (in μs) in function of the HT load factor α for the SA-hash algorithm

Finally, in Table 4 we present the overall space use for the four non-compact SA variants: plain SA, SA-LUT2, SA-LUT3 and SA-hash, plus SA-allHT, which is a (not implemented) structure comprising a suffix array, a LUT2 and one hash table for each $k \in \{3, 4, \ldots, m_{min}\}$. The space is expressed as a multiple of the text length n (including the text), which is for example 5.000 for the plain suffix array. We note that

the lookup table structures become a relatively smaller fraction when larger texts are indexed. For the variants with hash tables we take two load factors: 50% and 80%.

	dna	english	proteins	sources	xml
SA	5.000	5.000	5.000	5.000	5.000
SA-LUT2	5.001	5.001	5.001	5.001	5.001
SA-LUT3	5.321	5.321	5.321	5.321	5.321
SA-hash-50	6.050	6.587	5.278	7.010	5.958
SA-hash-80	5.657	5.992	5.174	6.257	5.600
SA-allHT-50	6.472	8.114	5.296	9.736	7.353
SA-allHT-80	5.920	6.947	5.185	7.960	6.471

Table 4. Space use for the non-compact data structures as a multiple of the indexed text size (including the text), with the assumption that text symbols are represented in 1 byte each and SA offsets are represented in 4 bytes. The value of m_{min} for SA-hash-50 and SA-hash-80, used in the construction of these structures and affecting their size, is like in the experiments from Fig. 4. The index SA-allHT-* contains one hash table for each $k \in \{3, 4, \ldots, m_{min}\}$, when m_{min} depends on the current dataset, as explained. The -50 and -80 suffixes in the structure names denote the hash load factors (in percent).

In the next set of experiments we evaluated the FBCSA index. Its properties of interest, for various block size (bs) and sampling step (ss) parameters, are: the space use, pattern search times, times to access (extract) one random SA cell, times to access (extract) multiple consecutive SA cells. For bs we set the values 32 and 64. The ss was tested in a wider range (3, 5, 8, 16, 32). Using $bs = 64$ results in better compression but decoding a cell is also slightly slower (see Fig. 5).

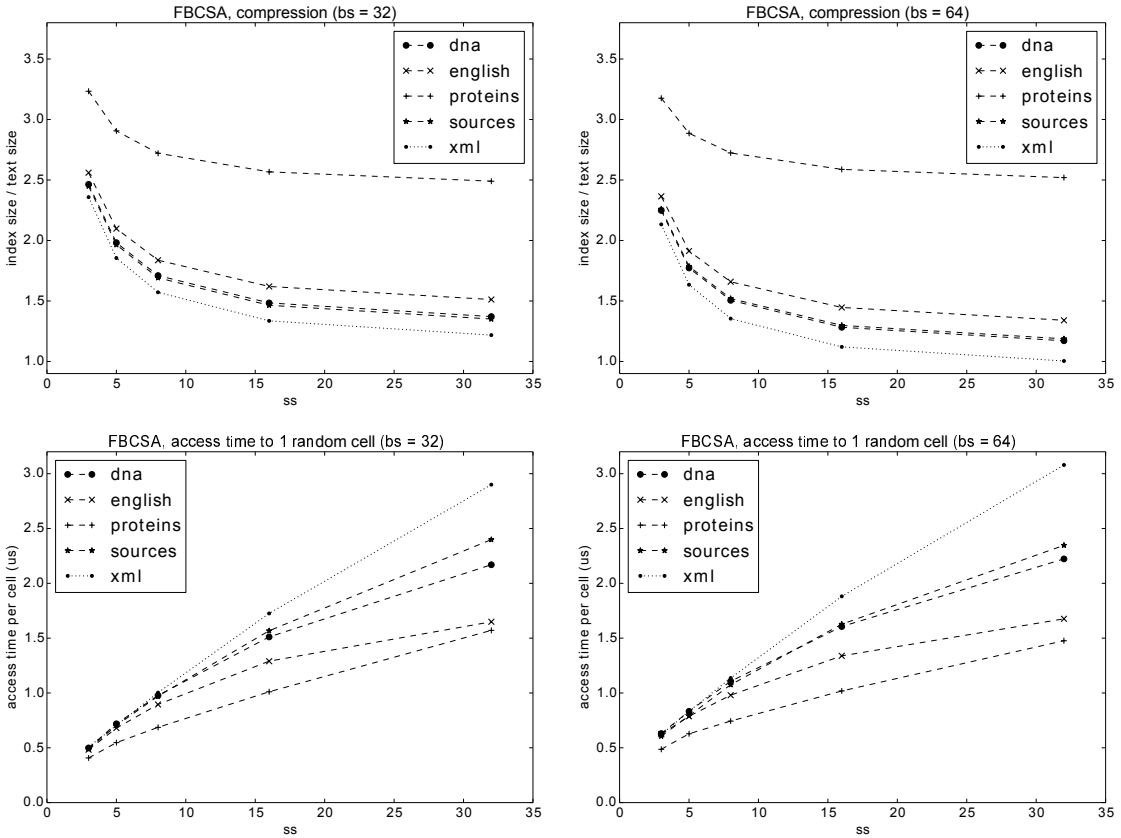


Figure 5. FBCSA index sizes and cell access times with varying ss parameter (3, 5, 8, 16, 32). The parameter bs was set to 32 (left figures) or 64 (right figures). The times are averages over 10M random cell accesses.

Unfortunately, our tests were run under Windows and it was not easy for us to adapt other competitive compact indexes to run on our platform, yet from the comparison with the results presented in [13, Sect. 4] we conclude that FBCSA is a few times faster in single cell access than the other related algorithms, MakCSA [20] (augmented with a compressed bitmap from [23] to extract arbitrary ranges of the suffix array) and LCSA / LCSA-Psi [13], at similar or better compression. Extracting c consecutive cells is not however an efficient operation for FBCSA (as opposed to MakCSA and LCSA / LCSA-Psi, see Figs 5–7 in [13]), yet for small ss the time growth is slower than linear, due to a few sampled (and thus written explicitly) SA offsets in a typical block (Fig. 6). Therefore, in extracting only 5 or 10 successive cells our index is still competitive.

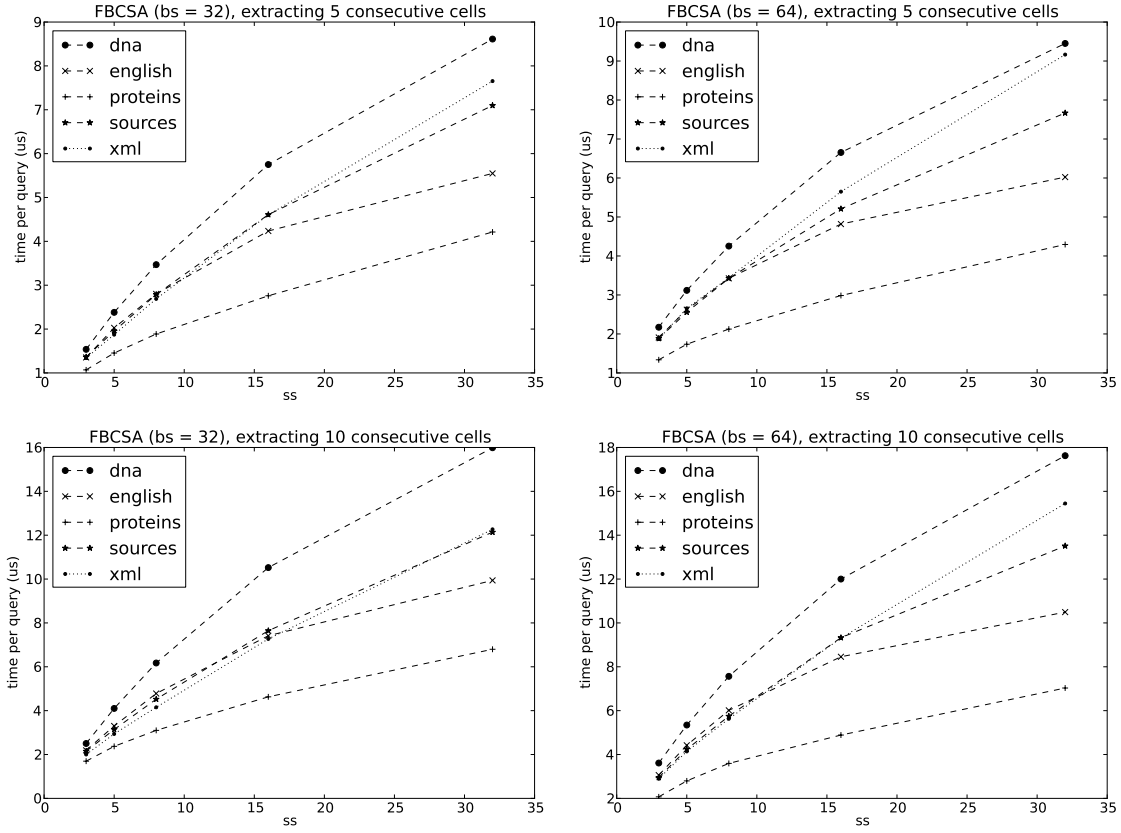


Figure 6. FBCSA, extraction time for $c = 5$ (top figures) and $c = 10$ (bottom figures) consecutive cells, with varying ss parameter (3, 5, 8, 16, 32). The parameter bs was set to 32 (left figures) or 64 (right figures). The times are averages over 1M random cell run extractions.

7 Conclusions

We presented two simple full-text indexes. One, called SA-hash, speeds up standard suffix array searches with reducing significantly the initial search range, thanks to a hash table storing range boundaries of all intervals sharing a prefix of a specified length. Despite its simplicity, we are not aware of such use of hashing in exact pattern matching, and the approximately 3-fold speedups compared to a standard SA may be worth the extra space in many applications.

The other presented data structure is a compact variant of the suffix array, related to Mäkinen's compact SA [20]. Our solution works on blocks of fixed size, which provides int32 alignment of the layout. This index is rather fast in single cell access, but not competitive if many (e.g., 100) consecutive cells are to be extracted.

Several aspects of the presented indexes requires further study. In the SA-hash scheme collisions in the HT may be eliminated with using perfect hashing or cuckoo hashing. This should also reduce the overall space use. In case of plain text, the standard suffix array component may be replaced with a suffix array on words [4], with possibly new interesting space-time tradeoffs. The idea of deep buckets may be incorporated into some compressed indexes, e.g., to save on the several first LF-mapping steps in the FM-index.

Acknowledgement

The work was supported by the National Science Centre under the project DEC-2013/09/B/ST6/03117 (both authors).

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *The enhanced suffix array and its applications to genome analysis*, in Algorithms in Bioinformatics, Springer, 2002, pp. 449–463.
2. R. Cole, T. Kopelowitz, and M. Lewenstein: *Suffix trays and suffix trists: structures for faster text indexing*, in Automata, Languages and Programming, Springer, 2006, pp. 358–369.
3. M. Farach: *Optimal suffix tree construction with large alphabets*, in Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science, 1997, pp. 137–143.
4. P. Ferragina and J. Fischer: *Suffix arrays on words*, in CPM, vol. 4580 of Lecture Notes in Computer Science, Springer, 2007, pp. 328–339.
5. P. Ferragina, R. González, G. Navarro, and R. Venturini: *Compressed text indexes: From theory to practice.* Journal of Experimental Algorithmics, 13(article 12) 2009, 30 pages.
6. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in Proceedings of the 41st IEEE Annual Symposium on Foundations of Computer Science, 2000, pp. 390–398.
7. J. Fischer and P. Gawrychowski: *Alphabet-dependent string searching with wexponential search trees.* arXiv preprint arXiv:1302.3347, 2013.
8. G. Franceschini and R. Grossi: *No sorting? Better searching!* ACM Transactions on Algorithms, 4(1) 2008.
9. S. Gog and A. Moffat: *Adding compression and blended search to a compact two-level suffix array*, in SPIRE, vol. 8214 of Lecture Notes in Computer Science, Springer, 2013, pp. 141–152.
10. S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth: *Large-scale pattern search using reduced-space on-disk suffix arrays.* IEEE Trans. Knowledge and Data Engineering (to appear), 2013, http://arxiv.org/abs/1303.6481.
11. S. Gog and M. Petri: *Optimized succinct data structures for massive data.* Software–Practice and Experience, 2013, DOI: 10.1002/spe.2198.
12. R. González and G. Navarro: *Compressed text indexes with fast locate*, in CPM, vol. 4580 of Lecture Notes in Computer Science, Springer, 2007, pp. 216–227.
13. R. González, G. Navarro, and H. Ferrada: *Locally compressed suffix arrays.* ACM Journal of Experimental Algorithmics, 2014, to appear.
14. N. Grimsmo: *On performance and cache effects in substring indexes*, Tech. Rep. IDI-TR-2007-04, NTNU, Department of Computer and Information Science, Sem Salands vei 7-9, NO-7491 Trondheim, Norway, 2007.
15. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, in Proceedings of the 32nd ACM Symposium on the Theory of Computing, ACM Press, 2000, pp. 397–406.

16. J. Kärkkäinen: *Suffix cactus: A cross between suffix tree and suffix array*, in CPM, vol. 937 of Lecture Notes in Computer Science, Springer, 1995, pp. 191–204.
17. J. Kärkkäinen and S. J. Puglisi: *Fixed block compression boosting in FM-indexes*, in SPIRE, R. Grossi, F. Sebastiani, and F. Silvestri, eds., vol. 7024 of Lecture Notes in Computer Science, Springer, 2011, pp. 174–184.
18. S. Kurtz and B. Balkenhol: *Space efficient linear time computation of the Burrows and Wheeler transformation*, in Numbers, Information and Complexity, Kluwer Academic Publishers, 2000, pp. 375–383.
19. V. Mäkinen: *Compact suffix array*, in CPM, R. Giancarlo and D. Sankoff, eds., vol. 1848 of Lecture Notes in Computer Science, Springer, 2000, pp. 305–319.
20. V. Mäkinen: *Compact suffix array – a space-efficient full-text index*. Fundam. Inform., 56(1-2) 2003, pp. 191–210.
21. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches*, in Proceedings of the 1st ACM-SIAM Annual Symposium on Discrete Algorithms, SIAM, 1990, pp. 319–327.
22. G. Navarro and V. Mäkinen: *Compressed full-text indexes*. ACM Computing Surveys, 39(1) 2007, p. article 2.
23. R. Raman, V. Raman, and S. S. Rao: *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets*, in SODA, 2002, pp. 233–242.
24. K. Sadakane: *Succinct representations of lcp information and improvements in the compressed suffix arrays*, in Proceedings of the 13th ACM-SIAM Annual Symposium on Discrete Algorithms, SIAM, 2002, pp. 225–232.
25. P. Weiner: *Linear pattern matching algorithm*, in Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, Washington, DC, 1973, pp. 1–11.

Reducing Squares in Suffix Arrays

Peter Leupold

Institut für Informatik,
Universität Leipzig,
Leipzig, Germany
Peter.Leupold@web.de

Abstract. In contrast to other mutations, duplication leaves an easily detectable trace: a repetition. Therefore it is a convenient starting point for computing a phylogenetic network. Basically, all squares must be detected to compute all possible direct predecessors. To find all the possible, not necessarily direct predecessors, this process must be iterated for all the resulting strings. We show how to reuse the work for one string in this process for the detection of squares in all the strings that are derived from it. For the detection of squares we propose suffix arrays as data structure and show how they can be updated after the reduction of a square.

Keywords: duplication, suffix array

1 Introduction

The duplication of parts of a sequence is a very frequent gene mutation in DNA [5]. The result of such a duplication is called a tandem repeat. As duplications occur frequently in genomes, they can offer a way of reconstructing the way in which different populations of a given species have developed. For example, Wapinski et al. [8] wanted to determine the relations between seventeen different populations of a species of fungi. They did this by looking at the tandem repeats in the genes. From these they induced possible relationships.

In a very simplified way, the approach works like this: suppose at the same location in the genome of different individuals (or entire populations) of the same species we have the following sequences: uv, uuv, $uvuv$, and $uvuvvuv$. In this case, it is possible and even probable that the latter sequences have evolved from the first one via duplication. The second and third sequences are direct derivations from the first. To reach the last sequence, several duplications are necessary. So the question is, whether the corresponding individual can be a descendant of one or both of the others. Further duplications in uuv cannot change the fact that there are two successive letters u in the string. Since $uvuvvuv$ does not contain uu, it cannot be a descendant. On the other hand, it is obtained from $uvuv$ by duplicating the last three letters. So in this case the structure of repeats suggests a relationship like $uuv \Leftarrow uv \Rightarrow uvuv \Rightarrow uvuvvuv$. It is the only relationship that is possible using only duplication.

In general, the representation of evolutionary relationships between different nucleotide sequences, genes, chromosomes, genomes, or species is called a *phylogenetic network* [1]. Duplication is a mutation that is relatively easy to detect. It results in a repeat and thus leaves a visible and detectable trace unlike a deletion or a substitution. Thus possible predecessors can be computed by only looking at the sequence under question instead of comparing it to candidates for predecessors. In order to find possible ancestors via duplication, in principle the entire possible duplication history of a string must be reconstructed. Figure 1 graphically displays such a history. The main task here is finding repetitions in and thus possible ancestors of the given string.

Peter Leupold: Reducing Squares in Suffix Arrays, pp. 192–201.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

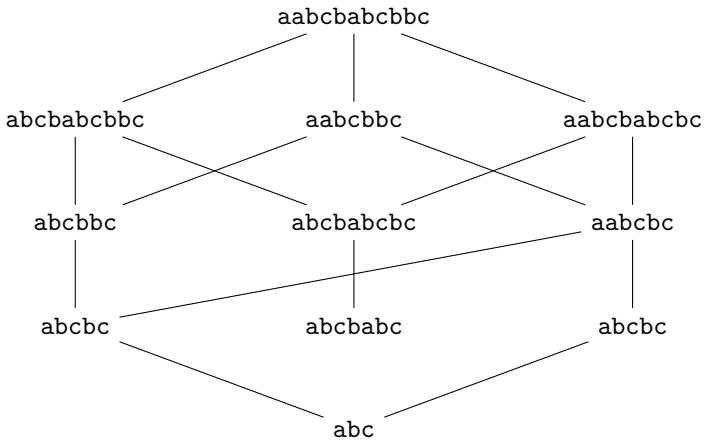


Figure 1. The duplication history for the string `aabcbabcbbc`. The direction of reductions is top to bottom.

There exist efficient algorithms for the detection of tandem repeats like the ones implemented in the `mreps` web interface [2]. These take into account biochemical conditions and do not necessarily search for perfect repetitions. For strings, the detection of perfect repetitions is frequently based on suffix arrays or similar data structures [6]. There are also good algorithms for the construction of these. However, if we want to search for more repetitions in the predecessors, we have to compute their suffix arrays from scratch. Since the change in the string is only local, the new suffix array will be very similar to the old one. We aim to characterize this similarity and to show how the new suffix array can be computed by modifying the old one rather than computing the new one from scratch. Here we address this problem for strings and perfect repetitions without regarding possible biochemical restrictions of tandem repeats in real DNA.

It is worth noting that such a computation might prove infeasible already due to the mere size of the solution. Note that at the end of each complete sequence of reduction there is a square-free string, i.e. one without any repetition. Obviously no further repetition can be reduced, if there is none by definition. When we consider the elimination of repetitions as a string-rewriting process, these square-free strings are the original string's normal forms. Even the number of these normal forms, which form only the end points in the paths of the duplication histories, can be exponential in the length of the string.

Theorem 1 ([4]). *For every positive integer n there are words of length n whose number N of normal forms under eliminating squares is bounded by:*

$$\frac{1}{30}110^{\frac{n}{42}} \leq N \leq 2^n.$$

Thus there can be no hope of finding an efficient algorithm for computing the entire duplication history for all possible strings. However, many strings will not reach this worst case. For them, we intend to find good methods for computing their duplication histories. The efficient detection of repetitions is a first step in such a computation.

2 Preliminaries on Strings

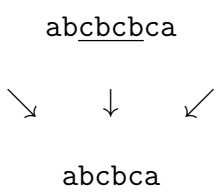
We recall some basic concepts on strings and fix notations for them. For a string w we denote by $w[i]$ the symbol at position i, where we start counting at 0. $w[i \dots j]$ denotes the substring starting at position i and ending at position j. A string w has a positive integer k as a *period*, if for all i, j such that $i \equiv j (\mathrm{mod}\ k)$ we have $w[i] = w[j]$, if both $w[i]$ and $w[j]$ are defined. The length of w is always a trivial period of w. If a string has a non-trivial period then it is called *periodic*.

A string u is a *prefix* of w if there exists an $i \leq |w|$ such that $u = w[0 \dots i]$; if $i < |w|$, then the prefix is called *proper*. Suffixes are the corresponding concept reading from the back of the word to the front. The *longest common prefix* of two strings u and v is the prefix $lcp(u, v) = u[0 \dots i] = v[0 \dots i]$ such that either $u[i+1] \neq v[i+1]$ or at least one of the strings has length only $i + 1$. So the function *lcp* takes two strings as its arguments and returns the length of their longest common prefix.

The *lexicographical order* is defined as follows for strings u and v over an ordered alphabet: $u \leq v$ if u is a prefix of v, or if the two strings can be factorized as $u = wau'$ and $v = wbv'$ for some w and letters a and b such that $a < b$.

3 Runs, not Squares

Before we start to reduce squares, let us take a look at the effect that this operation has in periodic factors. In the following example, we see that reduction of either of the three squares in the periodic factor *bcbcbc* leads to the same result:



Thus it would not be efficient to do all the three reductions and produce three times the same string. A maximal periodic factor like this is called a *run*. Maximal here means that if we choose a longer factor that includes the current one, then this longer factor does not have the same period any more. In the string above, *bcbcbc* is a run. It has period two, but its extensions *abcbcbc* to the left and *bcbcbca* to the right do not have period two any more.

it is rather straight-forward to see how the example from above generalizes to arbitrary periodic factors. For the sake of completeness we give a formal proof of this fact.

Lemma 2. *Let w be a string with period k. Then any deletion of a factor of length k will result in the same string.*

Proof. Because the string w has period k, the deleted factor starts with a suffix and ends with a prefix of $w[0 \dots k-1]$. Thus it is of the form

$$w[i+1 \dots k-1]w[0 \dots i]$$

for some $i < k$. If it starts at position $\ell + 1$, then the string

$$w[0 \ldots \ell]w[i + 1 \ldots k - 1]w[0 \ldots i]w[\ell + k + 1 \ldots |w| - 1]$$

is converted to

$$w[0 \ldots \ell]w[\ell + k + 1 \ldots |w| - 1].$$

For a deletion at position ℓ, which is one step more to the left, we obtain

$$w[0 \ldots \ell - 1]w[i]w[\ell + k + 1 \ldots |w| - 1].$$

Since $w[i] = w[\ell + k]$ this letter is equal to $w[\ell]$ due to the period k, and thus

$$w[0 \ldots \ell] = w[0 \ldots \ell - 1]w[i],$$

and the two results are the same. In an analogous way the deleted factor can be moved to the right. The result is the same, also for several consecutive movements. □

So rather than looking for squares, we should actually look for runs and reduce only one square within each of them. Then all the resulting strings will be different from each other.

As stated above, the most common algorithms for detecting runs are based on suffix arrays and related data structures [6]. Using these methods, we would employ a strategy along the lines of Algorithm 1. Then this method would again be applied

Algorithm 1: Computing all the strings reachable from w by reduction of squares.

```
   Input: string: w;
   Data: stringlist: S (contains w);
 1 while (S nonempty) do
 2 |   x := POP(S);
 3 |   Construct the suffix array of x;
 4 |   if (there are runs in x) then
 5 |   |   foreach run r do
 6 |   |   |   Reduce one square in r;
 7 |   |   |   Add new string to S;
 8 |   |   end
 9 |   end
10 |   else output x;
11 |   ;
12 end
```

to all the resulting strings which are not square-free. Our main aim is to improve line 3 by modifying the antecedent suffix array instead of constructing the new one from scratch. For this we first recall what a suffix array is.

4 Suffix Arrays

In string algorithms suffix arrays are a very common data structure, because they allow fast search for patterns. A suffix array of a string w consists of the two tables depicted on the left-hand side of Figure 2: SA is the lexicographically ordered list of all the suffixes of w; typically their starting position is saved rather than the entire

suffix. LCP is the list of the longest common prefixes between these suffixes. Here we only provide the values for direct neighbors. Depending on the application, they may be saved for all pairs. SA and LCP are also called an *extended* suffix array in contrast to SA alone.

SA	LCP	
7	1	`a`
0	0	`abcbbcba`
6	1	`ba`
3	1	`bbcba`
4	3	`bcba`
1	0	`bcbbcba`
5	2	`cba`
2		`cbbcba`

$\Rightarrow$

SA	LCP	
$7-3=4$	1	`a`
0	0	`abcba` (new)
$6-3=3$	1	`ba`
		—
$5-3=2$	0	`bcba`
		—
$4-3=1$		`cba`
		—

Figure 2. Modification of the suffix array by deletion of `bcb` in `abcbbcba`.

Now let there be a run with period k that contains at least one square uu starting in position i in a string; this means the run has at least length $2k$. Then the positions i and $i+k$ have an LCP of at least k and are very close to each other in the suffix array, because both suffixes start with u. This is why suffix arrays can be used to detect runs without looking at the actual string again once the array is computed.

On the right-hand side of Figure 2 we see how the deletion of `bcb` changes the suffix array. There is no change in the relative order nor in the LCP values for all the suffixes that start to the right of the deletion site; here it is more convenient to consider the first half of the square as the deleted one, because then we see immediately that also for the positions in the remaining right half nothing changes, see also Figure 3.

The only new suffix is `abcba`. It starts with the same letter as `abcbbcba`, the one it comes from; also the following *bcb* is the same as before, because the deleted factor is replaced by another copy of itself — only after that there can be some change. Thus the new suffix will not be very far from the old one in lexicographic order. Formulating these observations in a more general and exact way will be the objective of the next section.

5 Updating the Suffix Array

The problem we treat here is the following: Given a string w with a square of length n starting at position k and given the suffix array of w, compute the suffix array of $w[0 \ldots k-1]w[k+n \ldots |w|-1]$. So $w[k-1 \ldots k+n-1]$ is deleted from the original string, not $w[k+n \ldots k+2n-1]$. The result is, of course the same; however, it is convenient for our considerations to suppose that it is the first half of the square that is deleted. In this way, it is a little bit easier to see, which suffixes of the original string are also suffixes of the new one.

Figure 3 illustrates the simple fact that the positions to the right of a deleted square remain in the same order and that this is also true for the positions in the second half of the square, which is not deleted.

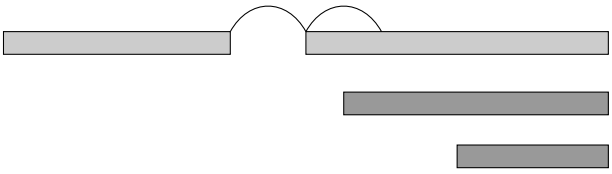


Figure 3. The order and LCP of suffixes that start right of the deletion remain unchanged.

For updating a suffix array, this means that only suffixes starting in positions to the left of the deleted site change and thus might also change their position in the suffix array. Figure 4 shows that no such change occurs if the LCP value is not greater than the sum of the length of the deleted factor (because this factor is replaced by another copy of itself from the right) and the distance from the start of the suffix to the start of the square (because it remains a prefix).

Only if the LCP is greater than this sum the suffix changes some of its first LCP many symbols as depicted in Figure 5. In this case we have to check if the position of the suffix in the suffix array changes. Lemma 3 formally proves these conditions.

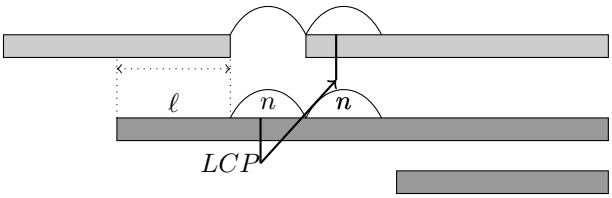


Figure 4. If the LCP is not greater than $\ell + n$, then the order of the new suffix relative to the old ones remains unchanged, because its $\ell + n$ are the same as before the deletion.

Lemma 3. *Let the LCP of two strings z and uvw be k and let $z < uvw$. Then z and $uvvw$ have the same LCP and $z < uvvw$ unless $LCP(z, uvw) \geq |uv|$; in the latter case also $LCP(z, uvvw) \geq |uv|$.*

Proof. If $LCP(z, uvw) < |uv|$ then the first position from the left where z and uvw differ is within uv. As uv is also a prefix of $uvvw$, z and $uvvw$ have their first difference in the same position. Thus LCP and the lexicographic order remain the same.

If $LCP(z, uvw) \geq |uv|$, then uv is a common prefix of z and $uvvw$. Thus also $LCP(z, uvvw) \geq |uv|$.

□

So we know that we only have to process suffixes starting to the left of the deleted factor. But also here not necessarily all suffixes have to be checked. To be more exact, as soon as starting from the right one suffix does not fulfill the conditions of Lemma 3, all the suffixes starting to the left of it will not fulfill these conditions either.

Lemma 4. *Let $LCP[j] = k$ in the suffix array of a string w of length $n + 1$. Then for $i < j$ we always have $LCP[i] \leq k + j - i$.*

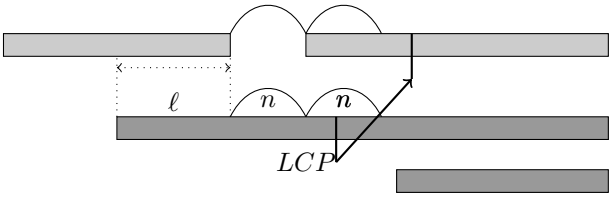


Figure 5. Only if the LCP is greater than $\ell + n$ some letter within the prefix of length LCP might change.

Proof. Let us suppose the contrary of the statement, i.e. $LCP[i] > k + j - i$. Further let the suffix of w that is lexicographically following $w[i \dots n]$ start at position m. This suffix shares a prefix of length at least $k + j - i + 1$ with $w[i \dots n]$, i.e.

$$w[m \dots m + k + 1] = w[i \dots i + k + 1].$$

Disregarding the first $j - i$ letters we obtain

$$w[m + (j - i) \dots m + (j - i) + k + 1] = w[i + (j - i) \dots i + (j - i) + k + 1]$$

and this gives us

$$w[m + j - i \dots m + j - i + k + 1] = w[j \dots j + k + 1].$$

So $w[i \dots n]$ shares $k + 1$ letters with the suffix $w[m + j - i \dots m + j - i + k + 1]$ of w. Further, since $w[m \dots n]$ is lexicographically greater than $w[i \dots n]$ also $w[m + j - i \dots n]$ is greater than $w[i \dots n]$. Therefore $LCP[i]$ must be at least $k+1$ contradicting our assumption.

□

Thus the best strategy seems to be to start at the first position left of the deleted factor and check the condition of Lemma 3. If it indicates that the position and LCP change, these changes are done and we move one position left. As soon as the condition of Lemma 3 says that no change can occur for the present position we can stop, because there will not be changes for the positions further to the left either.

Algorithm 2 implements this strategy avoiding unnecessary work according to the observations of this section. first we treat all the suffixes that start behind the deletion. They are simply decreased by n, the length of the deleted factor. The positions k to $k + n - 1$ are thus deleted. For the other suffixes we need to find out whether they must be moved to a different position. The test in line 5 checks exactly the condition of Lemma 3. Lemma 4 shows that after $LCP[i] > n+k-i$ we do not need to continue. The positions of the remaining suffixes can again be copied.

There is the variable m that appears in line 10 that is not bound. The reason is the following: If in line 6 a suffix is moved down in the array, then it is best to insert it directly in $SAnew$. If, on the other hand, it is moved up, then this part of $SAnew$ is not yet computed or copied. Therefore we should insert it into its position in the old SA and copy it in the final for-loop. So m should be the number of suffixes that are moved up in line 6 and it should be logged every time line 6 is executed.

It remains to implement the reordering and the computation of the new LCP value.

Algorithm 2: Computing the new suffix array.

```
   Input: string: w; arrays: SA, LCP;
   length and pos of square: n,k;
 1 for j = n + k to |w| − 1 do
 2 |  SAnew[j]:=SA[j]-n;
 3 end
 4 i := k − 1;
 5 while (LCP[i] > n + k − i AND i ≥ 0) do
 6 |  compute SAnew of w[i ... k − 1]w[k + n ... |w| − 1];
 7 |  compute new LCP[i];
 8 |  i := i − 1;
 9 end
10 for j = 0 to i + m do
11 |  SAnew[j]:=SA[j];
12 end
```

6 Computing the Changes

There are two tasks whose details are left open in Algorithm 2. For those suffixes whose position changes, we need to find the new position. Then also all of the affected LCP-values must be updated.

6.1 Computing the New Position

Intuitively, the position of a new suffix is not very far from the position of the old suffix it is derived from. If a suffix $wuuv$ is converted to wuv, then the prefix wu remains the same. Both w and u are non-empty in our case, because only suffixes that start left of the deletion can change their position as Figure 3 showed. Therefore $|wu| \geq 2$ and consequently $lcp(wuuv, wuv) \geq 2$. Further, it is clear that also all the suffixes that are between the positions of $wuuv$ and wuv must start with wu and thus have a LCP with both of them that is greater or equal to $|wu|$. Therefore we can restrict our search for the position of wuv to the part of the suffix array around $wuuv$, where the LCP-values are not smaller than $|wu|$. Within this range we use the standard method for inserting a string in a suffix array.

6.2 Updating the LCP

At this point, where a suffix is moved to another position, the LCP-table must be updated to contain the new LCP between the predecessor and the successor. This is computed via the following, well-known equality: for three consecutive suffixes u, v, and w in the suffix array, always $lcp(u, w) = min(lcp(u, v), lcp(v, w))$ holds, see also the work of Salson et al. that treats deletions in general [7].

When a new suffix is inserted into the suffix array or moved to a new position, we actually need to compute two LCP values: the ones with either of the neighboring positions. Here we distinguish two cases.

If we have inserted at the position just after $i+\ell$ from the original position i, then we know that the LCP of wuv with the suffix at position $j + 1$ is at least $|wu|$ as explained in Section 6.1. Therefore we only need to test starting from the first letter after wu whether there are further matching letters.

The LCP with the following position might be as low as zero. In this case, however, also the original $LCP[i+\ell]$ was zero, because the two suffixes start with the same letter as they share the prefix $|wu|$. More generally, if $LCP[i+\ell]$ was less than $|wu|$, then the inserted suffix has the same LCP with its successor. So we only need to compare more letters if the old $LCP[i+\ell]$ is greater than $|wu|$. Otherwise we can inherit the old value.

For the case that the new position in the suffix array is higher, we proceed symmetrically. The LCP with the successor is at least $|wu|$, and the LCP with the predecessor can be taken from the original list unless this value was greater than $|wu|$.

7 Conclusion and Perspectives

In the best case our method will immediately detect that essentially no change in the suffix array need to be done. For long squares this is even probable, because the longer the square the smaller the probability that the LCP-value will be even bigger. Then the test in line 5 of Algorithm 2 immediately fails, and we essentially copy the relevant parts of the old suffix array.

On the other hand, even in the worst case we should save time compared to constructing the new suffix array from scratch. Everything behind the reduced square can be copied. And for the new suffixes that change position we can use some of the old information for making the finding of the new position and the computation of LCP easier. The real question is in how far we can do better than the general updating after a deletion that Salson et al. designed [7]. This can probably only be answered by testing actual implementations on data set in analyses like the ones carried out by Léonard et al. [3].

We have only looked at how to update a suffix array efficiently. But for actually computing a duplication history several more problems must be handled in an efficient way. As one word can produce many descendants, many suffix arrays must be derived from the same one. Then all of these must be stored at the same time. Again, they are all similar to each other. The question is whether there are ways in which this similarity can be used to store them more compactly.

Further, a typical duplication history contains many paths to a given word. For example for a word $w_1u^2w_2v^2w_3x^2w_4$ that contains three squares that do not overlap, there is one normal form $w_1uw_2vw_3xw_4$ and there are six paths leading to this string. Every intermediate word is on two of these paths. The ones with one square left are reached from two different words, the normal form is reached from the three strings with only one square. How do we avoid computing a string more than once? Is there a way of knowing that the result was already obtained in an earlier reduction?

Depending in the goal of the computation, we can possibly do something about the length of the squares that are reduced. Squares of lengths one can be reduced first, if we do not want the entire reduction graph, but only the normal forms. For detecting and reducing these squares, it is faster to just run a window of size two over the string in low linear time without building the suffix array. After this, the value $n+k-i$ from line 5 of the algorithm would always be at least two. Squares of length two can already overlap with others in a way that reduction of one square makes reduction of the other impossible like in the string `abcbabcbc`; here reduction of the final `bcbc` leads to a square-free string, and the other normal form `abc` cannot be reached anymore.

References

1. D. H. Huson, R. Rupp, and C. Scornavacca: *Phylogenetic Networks*, Cambridge University Press, Cambridge, 2010.
2. R. Kolpakov, G. Bana, and G. Kucherov: *mreps: efficient and flexible detection of tandem repeats in DNA*. Nucleic Acids Research, 31(13) 2003, pp. 3672–3678.
3. M. Léonard, L. Mouchard, and M. Salson: *On the number of elements to reorder when updating a suffix array*. Journal of Discrete Algorithms, 11 2012, pp. 87–99.
4. P. Leupold: *Reducing repetitions*, in Prague Stringology Conference, J. Holub and J. Žďárek, eds., Prague Stringology Club Publications, Prague, 2009, pp. 225–236.
5. A. Meyer: *Molecular evolution: Duplication, duplication*. Nature, 421 2003, pp. 31–32.
6. S. J. Puglisi, W. F. Smyth, and M. Yusufu: *Fast, practical algorithms for computing all the repeats in a string*. Mathematics in Computer Science, 3(4) 2010, pp. 373–389.
7. M. Salson, T. Lecroq, M. Léonard, and L. Mouchard: *Dynamic extended suffix arrays*. Journal of Discrete Algorithms, 8(2) 2010, pp. 241–257.
8. I. Wapinski, A. Pfeffer, N. Friedman, and A. Regev: *Natural history and evolutionary principles of gene duplication in fungi*. Nature, 449 2007, pp. 54–61.

New Tabulation and Sparse Dynamic Programming Based Techniques for Sequence Similarity Problems

Szymon Grabowski

Lodz University of Technology, Institute of Applied Computer Science,
Al. Politechniki 11, 90–924 Łódź, Poland
sgrabow@kis.p.lodz.pl

Abstract. Calculating the length of a longest common subsequence (LCS) of two strings, A of length n and B of length m, is a classic research topic, with many worst-case oriented results known. We present two algorithms for LCS length calculation with respectively $O(mn \log\log n / \log^2 n)$ and $O(mn/\log^2 n + r)$ time complexity, the latter working for $r = o(mn/(\log n \log\log n))$, where r is the number of matches in the dynamic programming matrix. We also describe conditions for a given problem sufficient to apply our techniques, with several concrete examples presented, namely the edit distance, LCTS and MerLCS problems.

Keywords: sequence similarity, longest common subsequence, sparse dynamic programming, tabulation

1 Introduction

Measuring the similarity of sequences is an old research topic and many actual measures are known in the string matching literature. One classic example concerns the computation of a longest common subsequence (LCS) in which a subsequence that is common to all sequences and has the maximal possible length is looked for. A simple dynamic programming (DP) solution works in $O(mn)$ time for two sequences of length n and m, respectively, but faster algorithms are known. The LCS problem has many applications in diverse areas, like version control systems, comparison of DNA strings, structural alignment of RNA sequences. Other related problems comprise calculating the edit (Levenshtein) distance between two sequences, the longest common transposition-invariant subsequence, or LCS with constraints in which the longest common subsequence of two sequences must contain, or exclude, some other sequence.

Let us focus first on the LCS problem, for two sequences A and B. It is defined as follows. Given two sequences, $A = a_1 \cdots a_n$ and $B = b_1 \cdots b_m$, over an alphabet Σ of size σ, find a longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_\ell} \rangle$ of A such that $a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \ldots, a_{i_\ell} = b_{j_\ell}$, where $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ and $1 \le j_1 < j_2 < \cdots < j_\ell \le m$. The found sequence may not be unique. W.l.o.g. we assume $n \ge m$. To avoid uninteresting complications, we also assume that $m = \Omega(\log^2 n)$. Additionally, we assume that $\sigma = O(m)$. The case of a general alphabet, however, can be handled with standard means, i.e., we can initially map the sequences A and B onto an alphabet of size $\sigma' = O(m)$, in $O(n \log \sigma')$ time, using a balanced binary search tree. We do not comprise this tentative preprocessing step in further complexity considerations.

Often, a simplified version of the LCS problem is considered, when one is interested in telling only the length of a longest common subsequence (LLCS).

Szymon Grabowski: New Tabulation and Sparse Dynamic Programming Based Techniques for Sequence Similarity Problems, pp. 202–211.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

In this paper we present two techniques for finding the LCS length, one (Section 3) based on tabulation and improving the result of Bille and Farach-Colton [3] by factor $\log\log n$, the other (Section 4) combining tabulation and sparse dynamic programming and being slightly faster if the number of matches is appropriately limited. In Section 5 we show the conditions necessary to apply these algorithmic techniques. Some other, LCS-related, problems fulfill these conditions, so we immediately obtain new results for these problems as well.

Throughout the paper, we assume the word-RAM model of computation with machine word size $w \geq \log n$. All used logarithms are base 2.

2 Related work

A standard solution to the LCS problem is based on dynamic programming, and it is to fill a matrix M of size $(n+1) \times (m+1)$, where each cell value depends on a pair of compared symbols from A and B (that is, only if they match or not), and its (at most) three already computed neighbor cells. Each computed $M[i,j]$ cell, $1 \leq i \leq n, 1 \leq j \leq m$, stores the value of $LLCS(A[1 \ldots i], B[1 \ldots j])$. A well-known property describes adjacent cells: $M(i,j) - M(i-1,j) \in \{0,1\}$ and $M(i,j) - M(i,j-1) \in \{0,1\}$ for all valid i, j.

Despite almost 40 years of research, surprisingly little can be said about the worst-case time complexity of LCS. It is known that in the very restrictive model of unconstrained alphabet and comparisons with equal/unequal answers only, the lower bound is $\Omega(mn)$ [19], which is reached by a trivial DP algorithm. If the input alphabet is of constant size, the known lower bound is simply $\Omega(n)$, but if total order between alphabet symbols exists and $\leq$-comparisons are allowed, then the lower bound grows to $\Omega(n \log n)$ [10]. In other words, the gap between the proven lower bounds and the best worst-case algorithm is huge.

A simple idea proposed in 1977 by Hunt and Szymanski [12] has become a milestone in LCS reseach, and the departure point for theoretically better algorithms (e.g., [8]). The Hunt–Szymanski (HS) algorithm is essentially based on dynamic programming, but it visits only the matching cells of the matrix, typically a small fraction of the entire set of cells. This kind of selective scan over the DP matrix is called *sparse dynamic programming* (SDP). We note that the number of all matches in M, denoted with the symbol r, can be found in $O(n)$ time, and after this (negligible) preprocessing we can decide if the HS approach is promising to given data. More precisely, the HS algorithm works in $O(n + r \log m)$ or even $O(n + r \log\log m)$ time. Note that in the worst case, i.e., for $r = \Theta(mn)$, this complexity is however superquadratic.

The Hunt–Szymanski concept was an inspiration for a number of subsequent algorithms for LCS calculation, and the best of them, the algorithm of Eppstein et al. [8], achieves $O(D \log\log(\min(D, mn/D)))$ worst-case time (plus $O(n\sigma)$ preprocessing), where $D \leq r$ is the number of so-called dominant matches in M (a match (i,j) is called dominant iff $M[i,j] = M[i-1,j] + 1 = M[i,j-1] + 1$). Note that this complexity is $O(mn)$ for any value of D. A more recent algorithm, by Sakai [18], is an improvement if the alphabet is very small (in particular, constant), as its time complexity is $O(m\sigma + \min(D\sigma, p(m-q)) + n)$, where $p = LLCS(A,B)$ and $q = LLCS(A[1 \ldots m], B)$.

A different approach is to divide the dynamic matrix into small blocks, such that the number of essentially different blocks is small enough to be precomputed before the main processing phase. In this way, the block may be processed in constant time

each, making use of a built lookup table (LUT). This "Four Russians" technique was first used to the LCS problem by Masek and Paterson [15], for a constant alphabet, and refined by Bille and Farach-Colton [3] to work with an arbitrary alphabet. The obtained time compexities were $O(mn/\log^2 n)$ and $O(mn(\log\log n)^2/\log^2 n)$, respectively, with linear space.

A related, but different approach, is to use bit-parallelism to compute several cells of the dynamic programming matrix at a time. There are a few such variants (see [13] and references therein), all of them working in $O(\lceil m/w\rceil n)$ worst-case time, after $O(\sigma\lceil m/w\rceil + m)$-time and $O(\sigma m)$-space preprocessing, where w is the machine word size.

Yet another line of research considers the input sequences in compressed form. There exist such LCS algorithms for RLE-, LZ- and grammar-compressed inputs. We briefly mention two results. Crochemore et al. [4] exploited the LZ78-factorization of the input sequences over a constant alphabet, to achieve $O(hmn/\log n)$ time, where $h \leq 1$ is the entropy of the inputs. Gawrychowski [9] considered the case of two strings described by SLPs (straight line programs) of total size k, to show a solution computing their edit distance in $O(kn\sqrt{\log(n/k)})$ time, where n is the sum of their (non-compressed) length.

Some other LCS-related results can be found in the surveys [1,2].

3 LCS in $O(mn \log\log n/\log^2 n)$ time

In this section we modify the technique of Bille and Farach-Colton (BFC) [3, Sect. 4], improving its worst-case time by factor $\log\log n$, to achieve $O(mn\log\log n/\log^2 n)$ time complexity, with linear space. First we present the original BFC idea, and then signal how our algorithm diverts from it. In the presentation, some unimportant details of the BFC solution are changed, to make the description more compatible with our variant.

The dynamic programming matrix $M[0\ldots n, 0\ldots m]$ is divided into rectangular blocks with shared borders, of size $(x_1+1)\times(x_2+1)$, and the matrix is processed in horizontal stripes of x_2 rows. By "shared borders" we mean that e.g. the bottom row of some block being part of its output is also part of the input of the block below. Values inside each block depend on:

(*i*) x_1 corresponding symbols from sequence A,
(*ii*) x_2 corresponding symbols from sequence B,
(*iii*) the top row of the block, which can be encoded differentially in x_1 bits,
(*iv*) the leftmost column of the block, which can be encoded differentially in x_2 bits.

The output of each block will be found via a lookup table built in a preprocessing stage. The key idea of the BFC technique is alphabet remapping in superblocks of size $y\times y$. W.l.o.g. we assume that x_1 divides y and x_2 divides y. Consider one superblock of the matrix, corresponding to the two substrings: $A[i'y+1\ldots(i'+1)y]$ and $B[j'y+1\ldots(j'+1)y]$, for some i' and j'. For the substring $B[j'y+1\ldots(j'+1)y]$ its symbols are sorted and $q \leq y$ unique symbols are found. Then, the y symbols are remapped to $\Sigma_{B_{j'}} = \{0\ldots q-1\}$, using a balanced BST. Next, for each symbol from the snippet $A[i'y+1\ldots(i'+1)y]$ we find its encoding in $\Sigma_{B_{j'}}$, or assign q to it if it wasn't found there. This takes $O(\log y)$ time per symbol, thus the substrings of A and B associated with the superblock are remapped in $O(y\log y)$ time. The overall alphabet remapping time for the whole matrix is thus $O((mn\log y)/y)$.

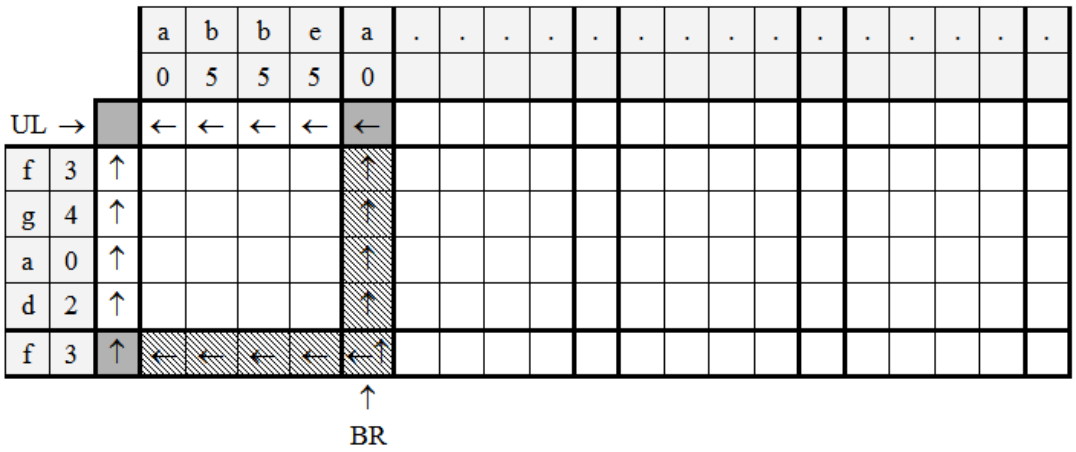


Figure 1. One horizontal stripe of the DP matrix, with 4 blocks of size 5×5 ($x_1 = x_2 = 4$). The corresponding snippets from sequence A and B are `abbea` and `fgadf`, respectively. These snippets are translated to a new alphabet (the procedure for creating the new alphabet is not shown here) of size 6, where the characters from A are mapped onto the alphabet $\{0, 1, \ldots, 4\}$ and value 5 is used for the characters from B not used in the encoding of the symbols from A belonging to the current superblock (the superblock is not shown here). The LCS values are stored explictly in the dark shaded cells. The white and dark shaded cells with arrows are part of the input, and their LCS values are encoded differentially, with regard to their left or upper neighbor. The diagonally shaded cells are the output cells, also encoded differentially. The bottom right corner (BR) is stored in three forms: as the difference to its left neighbor (0 or 1), as the difference to its upper neighbor (0 or 1) and the value of UL (upper-left corner) plus the difference between BR and UL. The difference between BR and UL is part of the LUT output for the current block.

This remapping technique allows to represent the symbols from the input components (i) and (ii) on $O(\log \min(y+1, \sigma))$ bits each, rather than $\Theta(\log \sigma)$ bits. It works because not the actual symbols from A and B are important for LCS computations, but only equality relations between them. To simplify notation, let us assume a large enough alphabet so that $\min(y + 1, \sigma) = y + 1$.

In this way, the input per block, comprising the components (i)–(iv) listed above, takes $x_1 \log(y + 1) + x_2 \log(y + 1) + x_1 + x_2$ bits, which cannot sum to $\omega(\log n)$ bits, otherwise the preprocessing time and space for building the LUT handling all possible blocks would be superpolynomial in n. Setting $y = x_1^2$ and $x_1 = x_2 = \log n/(6 \log \log n)$, we obtain $O(mn(\log \log n)^2/\log^2 n)$ overall time, with sublinear LUT space.

Now, we present our idea. Again, the matrix is processed in horizontal stripes of x_2 rows and the alphabet remapping in superblocks of size $y \times y$ is used. The difference concerns the lookup table; instead of one, we build many of them. More precisely, for each (remapped) substring of length x_2 from sequence B we build a lookup table for fast handling of the blocks in one horizontal stripe. Once a stripe is processed, its LUT is discarded to save space. This requires to compute the answers for all possible inputs in components (i), (iii) and (iv) (the component (ii) is fixed for a given stripe). The input thus takes $x_1 \log(y+1) + x_1 + x_2 = x_1 \log(2(y+1)) + x_2$ bits.

The return value associated with each LUT key are the bottom and the right border of a block, in differential form (the lowest cell in the right border and the rightmost cell in the bottom border are the same cell, which is represented twice;

once as a difference (0 or 1) to its left neighbor in the bottom border and once as a difference (0 or 1) to its upper neighbor in the right border) and the difference between the values of the bottom right and the top left corner (to know the explicit value of M in the bottom right corner), requiring $x_1 + x_2 + \log(\min(x_1, x_2) + 1)$ bits in total. Fig. 1 illustrates.

As long as the input and the output of an LUT fits a machine word, i.e., does not exceed w bits, we will process one block in constant time. Still, as in the original BFC algorithm, the LUT building costs also impose a limitation. More precisely, we are going to minimize the total time of remapping the alphabet in all the superblocks, building all $O(m/x_2)$ LUTs and finally processing all the blocks, which is described by the formula:

$$O(m \log y + (mn \log y)/y + (m/x_2)2^{x_1 \log(2(y+1))+x_2} x_1 x_2 + mn/(x_1 x_2)),$$

where $2^{x_1 \log(2(y+1))+x_2}$ is the number of all possible LUT inputs and the $x_1 x_2$ multiplier corresponds to the computation time per one LUT cell. Let us set $y = \log^2 n/2$, $x_1 = \log n/(4 \log \log n)$ and $x_2 = \log n/4$. In total we obtain $O(mn \log \log n/\log^2 n)$ time with $o(n)$ extra space (for the lookup tables, used one at a time, and alphabet remapping), which improves the Bille and Farach-Colton result by factor $\log \log n$. The improvement is achieved thanks to using multiple lookup tables (one per horizontal stripe). Formally, we obtain the following theorem.

Theorem 1. *The length of the longest common subsequence (LCS) between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \log^2 n$, both over an integer alphabet, can be computed in $O(mn \log \log n/\log^2 n)$ worst-case time. The algorithm needs $o(n)$ words of space, apart for the two sequences themselves.*

4 LCS in $O(mn/\log^2 n + r)$ time (for some r)

In this algorithm we also work in blocks, of size $(b+1) \times (b+1)$, but divide them into two groups: sparse blocks are those which contain at most K matches and dense blocks are those which contain more than K matches. Obviously, we do not count possible matches on the input boundaries of a block.

We observe that knowing the left and top boundary of a block plus the location of all the matches in the block is enough to compute the remaining (right and bottom) boundaries. This is a nice property as it eliminates the need to (explicitly) access the corresponding substrings of A and B.

The sparse block input will be encoded as:

(*i*) the top row of the block, represented differentially in b bits,
(*ii*) the leftmost column of the block, represented differentially in b bits,
(*iii*) the match locations inside the block, each in $\log(b^2)$ bits, totalling $O(K \log b)$ bits.

Each sparse block will be computed in constant time, thanks to an LUT. Dense blocks, on the other hand, will be partitioned into smaller blocks, which in turn will be handled with our algorithm from Section 3. Clearly, we have $b = O(\log n)$ (otherwise the LUT build costs would be dominating) and $b = \omega(\log n/\sqrt{\log \log n})$ (otherwise this algorithm would never be better than the one from Section 3), which implies that $K = \Theta(\log n/\log \log n)$, with an appropriate constant.

As this algorithm's worst-case time is $\Omega(mn/\log^2 n)$, it is easy to notice that the preprocessing costs for building required LUTs and alphabet mapping will not dominate. Each dense block is divided into smaller blocks of size $\Theta(\log n/\log\log n) \times \Theta(b)$. Let the fraction of dense blocks in the matrix be denoted as f_d (for example, if half of the $(b+1) \times (b+1)$ blocks in the matrix are dense, $f_d = 0.5$). The total time complexity (without preprocessing) is then

$$O((1 - f_d)mn/b^2 + f_d(mn \log\log n/(b \log n))).$$

The fraction f_d must be $o(1)$, otherwise this algorithm is not better in complexity than the previous one. This also means that $1 - f_d$ may be replaced with 1 in further complexity considerations.

Recall that r is the number of matches in the dynamic programming matrix. We have $f_d = O((r/K)/(mn/b^2)) = O(rb^2 \log\log n/(mn \log n))$. From the $f_d = o(1)$ condition we also obtain that $rb^2 = o(mn \log n/\log\log n)$. If $r = o(mn/(\log n \log\log n))$, then we can safely use the maximum possible value of b, i.e., $b = \Theta(\log n)$ and obtain the time of $O(mn/\log^2 n)$.

Unfortunately, in the preprocessing we have to find and encode all matches in all sparse blocks, which requires $O(n + r)$ time. Overall, this leads to the following theorem.

Theorem 2. *The length of the longest common subsequence (LCS) between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \log^2 n$, both over an integer alphabet, can be computed in $O(mn/\log^2 n + r)$ worst-case time, assuming $r = o(mn/(\log n \log\log n))$, where r is the number of matching pairs of symbols between A and B.*

Considering to the presented restriction on r, the achieved complexity is better than the result from the previous section.

On the other hand, it is essential to compare the obtained time complexity with the one from Eppstein et al. algorithm [8]. All we can say about the number of dominant matches D is the $D \leq r$ inequality[1], so we replace D with r in their complexity formula to obtain $O(r \log\log(\min(r, mn/r)))$ in the worst case. Our result is better if $r = \omega(mn/(\log^2 n \log\log\log n))$ and $r = o(mn)$. Overall, it gives the niche of $r = \omega(mn/(\log^2 n \log\log\log n))$ and $r = o(mn \log\log n/\log^2 n)$ in which the algorithm presented in this section is competitive.

The alphabet size is yet another constraint. From the comparison to Sakai's algorithm [18] we conclude that our algorithm needs $\sigma = \omega(\log\log\log n)$ to dominate for the case of $r = \omega(mn/(\log^2 n \log\log\log n))$.

5 Algorithmic applications

The techniques presented in the two previous sections may be applied to any sequence similarity problem fulfilling certain properties. The conditions are specified in the following lemma.

Lemma 3. *Let Q be a sequence similarity problem returning the length of a desired subsequence, involving two sequences, A of length n and B of length m, both over a*

[1] A slightly more precise bound on D is $\min(r, m^2)$, but it may matter, in complexity terms, only if $m = o(n)$ (cf. also [18, Th. 1]), which is a less interesting case.

common integer alphabet Σ of size $\sigma = O(m)$. We assume that $1 \leq m \leq n$. Let Q admit a dynamic programming solution in which $M(i,j) - M(i-1,j) \in \{-1,0,1\}$, $M(i,j) - M(i,j-1) \in \{-1,0,1\}$ for all valid i and j, and $M(i,j)$ depends only on the values of its (at most) three neighbors $M(i-1,j)$, $M(i,j-1)$, $M(i-1,j-1)$, and whether $A_i = B_j$.

There exists a solution to problem Q with $O(mn \log\log n / \log^2 n)$ worst-case time. There also exists a solution to Q with $O(mn/\log^2 n + r)$ worst-case time, for $r = o(mn/(\log n \log\log n))$, where r is the number of symbols pairs A_i, B_j such that $A_i = B_j$. The space use in both solutions is $O(n)$ words.

Proof. We straightforwardly apply the ideas presented in the previous two sections. The only modification is to allow a broader range of differences ($\{-1,0,1\}$) between adjacent cells in the dynamic programming matrix. This only affects a constant factor in parameter setting. □

Lemma 3 immediately serves to calculate the edit (Levenshtein) distance between two sequences (in fact, the BFC technique was presented in [3] in terms of the edit distance). We therefore obtain the following theorem.

Theorem 4. *The edit distance between two sequences, A, of length n, and B, of length m, where $n \geq m \geq \log^2 n$, both over an integer alphabet, can be computed in $O(mn \log\log n / \log^2 n)$ worst-case time. Alternatively, the distance can be found in $O(mn/\log^2 n + r)$ worst-case time, for $r = o(mn/(\log n \log\log n))$, where r is the number of symbols pairs A_i, B_j such that $A_i = B_j$. The space use in both solutions is $O(n)$ words.*

Another feasible problem is the longest common transposition-invariant subsequence (LCTS) [14,5], in which we look for a longest subsequence of the form $(s_1 + t)(s_2 + t)\cdots(s_\ell + t)$ such that all s_i belong to A (in increasing order), all corresponding values $s_i + t$ belong to B (in increasing order), and $t \in \{-\sigma + 1, \ldots, \sigma - 1\}$ is some integer, called a transposition. This problem is motivated by music information retrieval. The best known results for LCTS are $O(mn \log\log\sigma)$ [16,5] and $O(mn\sigma(\log\log n)^2/\log^2 n)$ if the BFC technique is applied for all transpositions (which is $O(mn)$ if $\sigma = O(\log^2 n/(\log\log n)^2)$). Applying the former result from Lemma 3, for all possible transpositions, gives immediately $O(mn\sigma \log\log n/\log^2 n)$ time complexity (if $\sigma = O(n^{1-\varepsilon})$, for any $\varepsilon > 0$, otherwise the LUT build costs would dominate). Applying the latter result requires more care. First we notice that the number of matches over all the transpositions sum up to mn, so $\Theta(mn)$ is the total preprocessing cost. Let us divide the transpositions into dense ones and sparse ones, where the dense ones are those that have at least $mn \log\log n/\sigma$ matches. The number of dense transpositions is thus limited to $O(\sigma/\log\log n)$. We handle dense transpositions with the technique from Section 3 and sparse ones with the technique from Section 4. This gives us $O(mn + mn(\sigma/\log\log n)\log\log n/\log^2 n + mn\sigma/\log^2 n) = O(mn(1 + \sigma/\log^2 n))$ total time, assuming that $\sigma = \omega(\log n(\log\log n)^2)$, as this condition on σ implies the number of matches in each sparse transposition limited to $o(mn/(\log n \log\log n))$, as required. We note that $\sigma = \omega(\log^2 n/(\log\log n)^2)$ and $\sigma = O(\log^2 n)$ is the niche in which our algorithm is the first one to achieve $O(mn)$ total time.

Theorem 5. *The length of the longest common transposition-invariant subsequence (LCTS) between two sequences, A, of length n, and B, of length m, where $n \geq m \geq$*

$\log^2 n$, *both over an integer alphabet of size* σ, *can be computed in* $O(mn(1+\sigma/\log^2 n))$ *worst-case time, assuming that* $\sigma = \omega(\log n(\log\log n)^2)$.

A natural extension of Lemma 3 is to involve more than two (yet a constant number of) sequences. In particular, problems on three sequences have practical importance.

Lemma 6. *Let* Q *be a sequence similarity problem returning the length of a desired subsequence, involving three sequences,* A *of length* n, B *of length* m *and* P *of length* u, *all over a common integer alphabet* Σ *of size* $\sigma = O(m)$. *We assume that* $1 \leq m \leq n$ *and* $u = \Omega(n^c)$, *for some constant* $c > 0$. *Let* Q *admit a dynamic programming solution in which* $M(i,j,k) - M(i-1,j,k) \in \{-1,0,1\}$, $M(i,j,k) - M(i,j-1,k) \in \{-1,0,1\}$ *and* $M(i,j,k) - M(i,j,k-1) \in \{-1,0,1\}$, *for all valid* i, j *and* k, *and* $M(i,j,k)$ *depends only on the values of its (at most) seven neighbors:* $M(i-1,j,k)$, $M(i,j-1,k)$, $M(i-1,j-1,k)$, $M(i,j,k-1)$, $M(i-1,j,k-1)$, $M(i,j-1,k-1)$ *and* $M(i-1,j-1,k-1)$, *and whether* $A_i = B_j$, $A_i = P_k$ *and* $B_j = P_k$.

There exists a solution to Q *with* $O(mnu/\log^{3/2} n)$ *worst-case time. The space use is* $O(n)$ *words.*

Proof. The solution works on cubes of size $b \times b \times b$, setting $b = \Theta(\sqrt{\log n})$ with an appropriate constant. Instead of horizontal stripes, 3D "columns" of size $b \times b \times u$ are now used. The LUT input consists of b symbols from sequence P, encoded with respect to a supercube in $O(\log\log n)$ bits each, and three walls, of size $b \times b$ each, in differential representation. The output are the three opposite walls of a cube. The restriction $u = \Omega(n^c)$ implies that the overall time formula *without the LUT build times* is $\Omega(mn^{1+c}/\log^{3/2} n)$, which is $\Omega(mn^{1+c'})$, for some constant c', $c \geq c' > 0$, e.g., for $c' = c/2$. The build time for all LUTs can be made $O(mn^{1+c''})$, for any constant $c'' > 0$, if the constant associated with b is chosen appropriately. We now set $c'' = c'$ to show the build time for the LUTs is not dominating. □

As an application of Lemma 6 we present the merged longest common subsequence (MerLCS) problem [11], which involves three sequences, A, B and P, and its returned value is a longest sequence T that is a subsequence of P and can be split into two subsequences T' and T'' such that T' is a subsequence of A and T'' is a subsequence of B. Deorowicz and Danek [6] showed that in the DP formula for this problem $M(i,j,k)$ is equal to or larger by 1 than any of the neighbors: $M(i-1,j,k)$, $M(i,j-1,k)$ and $M(i,j,k-1)$. They also gave an algorithm working in $O(\lceil u/w \rceil mn \log w)$ time. Peng et al. [17] gave an algorithm with $O(\ell mn)$ time complexity, where $\ell \leq n$ is the length of the result. Motivations for the MerLCS problem, from bioinformatics and signal processing, can be found e.g. in [6].

Based on the cited DP formula property [6] we can apply Lemma 6 to obtain $O(mnu/\log^{3/2} n)$ time for MerLCS (if $u = \Omega(n^c)$ for some $c > 0$), which may be competitive with existing solutions.

Theorem 7. *The length of the merged longest common subsequence (MerLCS) involving three sequences,* A, B *and* P, *of length respectively* n, m *and* u, *where* $m \leq n$ *and* $u = \Omega(n^c)$, *for some constant* $c > 0$, *all over an integer alphabet of size* σ, *can be computed in* $O(mnu/\log^{3/2} n)$ *worst-case time.*

6 Conclusions

On the example of the longest common subsequence problem we presented two algorithmic techniques, making use of tabulation and sparse dynamic programming paradigms, which allow to obtain competitive time complexities. Then we generalize the ideas by specifying conditions on DP dependencies whose fulfilments lead to immediate applications of these techniques. The actual problems considered here as applications comprise the edit distance, LCTS and MerLCS.

As a future work, we are going to relax the DP dependencies, which may for example improve the SEQ-EC-LCS result from [7]. Another research option is to try to improve the tabulation based result on compressible sequences.

Acknowledgments

The author wishes to thank Sebastian Deorowicz and an anonymous reviewer for helpful comments on a preliminary version of the manuscript.

References

1. A. Apostolico: *String editing and longest common subsequences*, in Handbook of Formal Languages, vol. 2 Linear Modeling: Background and Application, Springer, 1997, ch. 8, pp. 361–398.
2. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in SPIRE, IEEE Computer Society, 2000, pp. 39–48.
3. P. Bille and M. Farach-Colton: *Fast and compact regular expression matching.* Theoret. Comput. Sci., 409(3) 2008, pp. 486–496.
4. M. Crochemore, G. M. Landau, and M. Ziv-Ukelson: *A subquadratic sequence alignment algorithm for unrestricted scoring matrices.* SIAM J. Comput., 32(6) 2003, pp. 1654–1673.
5. S. Deorowicz: *Speeding up transposition-invariant string matching.* Inform. Process. Lett., 100(1) 2006, pp. 14–20.
6. S. Deorowicz and A. Danek: *Bit-parallel algorithms for the merged longest common subsequence problem.* Internat. J. Foundations Comput. Sci., 24(08) 2013, pp. 1281–1298.
7. S. Deorowicz and S. Grabowski: *Subcubic algorithms for the sequence excluded LCS problem*, in Man-Machine Interactions 3, 2014, pp. 503–510.
8. D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano: *Sparse dynamic programming I: Linear cost functions.* J. ACM, 39(3) 1992, pp. 519–545.
9. P. Gawrychowski: *Faster algorithm for computing the edit distance between SLP-compressed strings*, in SPIRE, 2012, pp. 229–236.
10. D. S. Hirschberg: *An information-theoretic lower bound for the longest common subsequence problem.* Inform. Process. Lett., 7(1) 1978, pp. 40–41.
11. K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng: *Efficient algorithm for finding interleaving relationship between sequences.* Inform. Process. Lett., 105(5) 2008, pp. 188–193.
12. J. W. Hunt and T. G. Szymanski: *A fast algorithm for computing longest common subsequences.* Comm. ACM, 20(5) 1977, pp. 350–353.
13. H. Hyyrö: *Bit-parallel LCS-length computation revisited*, in AWOCA, Australia, 2004, University of Sydney, pp. 16–27.
14. V. Mäkinen, G. Navarro, and E. Ukkonen: *Transposition invariant string matching.* Journal of Algorithms, 56(2) 2005, pp. 124–153.
15. W. Masek and M. Paterson: *A faster algorithm computing string edit distances.* J. Comput. Syst. Sci., 20(1) 1980, pp. 18–31.
16. G. Navarro, S. Grabowski, V. Mäkinen, and S. Deorowicz: *Improved time and space complexities for transposition invariant string matching*, Technical Report TR/DCC-2005-4, Department of Computer Science, University of Chile, 2005.

17. Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor: *Efficient sparse dynamic programming for the merged LCS problem with block constraints*. International Journal of Innovative Computing, Information and Control, 6(4) 2010, pp. 1935–1947.
18. Y. Sakai: *A fast on-line algorithm for the longest common subsequence problem with constant alphabet*. IEICE Transactions, 95-A(1) 2012, pp. 354–361.
19. C. K. Wong and A. K. Chandra: *Bounds for the string editing problem*. J. ACM, 23(1) 1976, pp. 13–16.

Author Index

Proceedings of the Prague Stringology Conference 2014
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, Praha 6, 160 00, Czech Republic.

ISBN 978-80-01-05547-2

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