Proceedings of the Prague Stringology Conference 2021

Edited by Jan Holub and Jan Žďárek

August 2021

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

ISBN 978-80-01-06869-4

Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2021 (PSC 2021) held on August 30–31, 2021 at the Czech Technical University in Prague, which organizes the event. The conference focused on stringology, i.e., a discipline concerned with algorithmic processing of strings and sequences, and related topics.

The submitted papers were reviewed by the program committee subject to originality and quality. The eight papers in this proceedings made the cut and were selected for regular presentation at the conference.

The PSC 2021 was organized in both present and remote form like PSC 2020. Each participant could decide based on COVID-19 travel restrictions in her or his country whether to arrive to Prague or to participate remotely.

The Prague Stringology Conference has a long tradition. PSC 2021 is the twenty-fifth PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2020 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions have been regularly published in special issues of journals such as: 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.

The PSC 2021 pays homage to Bořivoj Melichar, the founder of Stringology in Prague. We was also co-founder of Prague Stringology Conference.

We would like to thank all those who had submitted papers for PSC 2021 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 2021. Last, but not least, our thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2021

Jan Holub and Amihood Amir

Conference Organisation

Program Committee

Amihood Amir, *Co-chair*	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Simone Faro	(Università di Catania, Italy)
František Franěk	(McMaster University, Canada)
Jan Holub, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Shunsuke Inenaga	(Kyushu University, Japan)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Thierry Lecroq	(Université de Rouen, France)
Marie-France Sagot	(INRIA Rhône-Alpes, France)
William F. Smyth	(McMaster University, Canada, and Murdoch University, Australia)
Bruce W. Watson	(FASTAR Group/Stellenbosch University, South Africa)
Jan Žďárek	(Czech Technical University in Prague, Czech Republic)

Organising Committee

Ondřej Guth	Tomáš Pecka	Jan Trávníček, *Co-chair*
Jan Holub, *Co-chair*	Eliška Šestáková	Jan Žďárek
Radomír Polách		

External Referees

Hideo Bannai	Neerja Mhaskar
M. Oguzhan Kulekci	Yuto Nakashima

Table of Contents

Contributed Talks

Automata Approach to Inexact Tree Pattern Matching Using 1-degree Edit Distance *by Eliška Šestáková, Ondřej Guth, and Jan Janoušek* 1

Pitfalls of Algorithm Comparison *by Waltteri Pakalén, Hannu Peltola, Jorma Tarhio, and Bruce W. Watson* 16

Refined Upper Bounds on the Size of the Condensed Neighbourhood of Sequences *by Cedric Chauve, Marni Mishna, and France Paquet-Nadeau* 30

Computational Substantiation of the d-step Conjecture for Distinct Squares Revisited *by Frantisek Franek and Michael Liut* 41

Counting Lyndon Subsequences *by Ryo Hirakawa, Yuto Nakashima, Shunsuke Inenaga, and Masayuki Takeda* 53

The n-ary Initial Literal and Literal Shuffle *by Stefan Hoffmann* 61

Towards an Efficient Text Sampling Approach for Exact and Approximate Matching *by Simone Faro, Francesco Pio Marino, Arianna Pavone, and Antonio Scardace* 75

Searching with Extended Guard and Pivot Loop *by Waltteri Pakalén, Jorma Tarhio, and Bruce W. Watson* 90

Author Index 103

Automata Approach to Inexact Tree Pattern Matching Using 1-degree Edit Distance [⋆]

Eliška Šestáková, Ondřej Guth, and Jan Janoušek

Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Praha 6
Czech Republic
{sestaeli,guthondr,janousej}@fit.cvut.cz

Abstract. We compare labeled ordered trees based on unit cost 1-degree edit distance that uses operations vertex relabeling, leaf insertion, and leaf deletion. Given an input tree T and a tree pattern P, we find all subtrees in T that match P with up to k errors. We show that this problem can be solved by finite automaton when T and P are represented in linear, prefix bar, notation. First, we solve this problem by a pushdown automaton. Then, we show that it can be transformed into a nondeterministic finite automaton due to its restricted use of the pushdown store. We also show a simulation of the nondeterministic finite automaton by dynamic programming.

Keywords: inexact tree pattern matching, approximate tree pattern matching, finite automaton, pushdown automaton, subtree matching, dynamic programming, linear tree notation, prefix bar notation, Selkow distance, 1-degree edit distance, ordered trees

1 Introduction

The problem of inexact (or approximate) tree pattern matching is for a given input tree T and tree pattern P, find all subtrees in T that match P with up to k errors. This type of tree pattern matching can be helpful if one of the trees (or both) can be subjects of deformation or corruption; in such circumstances, the tree pattern matching needs to be more tolerant when comparing two trees.

The problem of measuring similarities between two trees is called the tree edit distance problem (or tree-to-tree correction problem). This problem is a generalization of the well-known string edit distance problem, and it is defined as the minimum cost sequence of vertex edit operations that transform one tree into the other [1].

In this paper, we consider labeled ordered trees in which each vertex is associated with a label, and sibling order matters. For labeled ordered trees, Tai [10] introduced the set of operations that included vertex relabeling, vertex insertion, and vertex deletion. A different cost may accompany the operations. Given two labeled ordered trees with m and n vertices, where $n \geq m$, the tree edit distance between those trees can be solved in cubic $\mathcal{O}(n^3)$ time [3]. According to a recent result [2], it is unlikely that a truly subcubic algorithm for the ordered tree edit distance problem exists.

For unordered trees, Zhang et al. proved that the tree edit distance problem is NP-complete, even for binary trees having an alphabet containing just two labels [12].

[⋆] *The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16 019/0000765 "Research Center for Informatics". This work was also supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/208/OHK3/3T/18.*

Several authors have also proposed restricted forms and variations of the tree edit distance problem. For example, Selkow [8] restricted the vertex insertion and deletion to leaves of a tree only. These operations may be used recursively to allow insertion or deletion of a subtree of arbitrary size. This distance is in the literature often referred to as 1-degree edit distance. Selkow also gave an algorithm running in $\mathcal{O}(nm)$ time and space, where n and m are the numbers of vertices of the two labeled ordered input trees. His algorithm uses a dynamic programming approach in which the input trees are recursively decomposed into smaller subproblems. A similar approach is used in most state-of-the-art algorithms for the tree edit distance problem.

The dynamic programming approach was also successfully used to solve the string edit distance problem and the inexact (approximate) string pattern matching problem [7]. Besides dynamic programming, however, finite automata can also be used to solve the inexact string pattern matching problem [6,7].

Inspired by techniques from string matching, we aim to show that the automata approach can also be used to solve the inexact tree pattern matching problem. We consider labeled ordered (unranked) trees and 1-degree edit distance. For simplicity, we use unit cost, where each operation costs one. However, the extension of our approach to non-unit cost distance is also discussed.

First, we solve the problem by a pushdown automaton. Then, we show that it can be transformed into a finite automaton due to its restricted use of the pushdown store. The deterministic version of the finite automaton finds all occurrences of the tree pattern in time linear to the size of the input tree. We also present an algorithm based on dynamic programming, which is a simulation of the nondeterministic finite automaton. The space complexity of this approach is $\mathcal{O}(mk)$ and the time complexity is $\mathcal{O}(kmn)$, where m is the number of vertices of the tree pattern, n is the number of vertices of the input tree, and $k \leq m$ represents the number of errors allowed in the pattern.

Our approach extends the previous result by Šestáková, Melichar, and Janoušek [9]. In their paper, they used a finite automaton to solve the inexact tree pattern matching problem with a more restricted 1-degree edit distance; the distance uses the same set of operations defined by Selkow, but these operations cannot be used recursively to allow insertion or deletion of a subtree of arbitrary size. Therefore, it may not always be possible to transform one tree into the other.

To be able to process trees using (string) automata, we represent trees using a linear notation called the prefix bar notation [5]. This notation is similar to the bracketed notation in which each subtree is enclosed in brackets. The prefix bar notation uses just the closing bracket (denoted by bar "|" symbol) due to the simple observation that the left bracket is redundant; there is always the root of a subtree just behind the left bracket. We note that this notation corresponds, for example, to the notation used in XML; each end-tag can be mapped to the bar symbol. Similarly straightforward is the transformation of JSON.

This paper is organized as follows. In Section 2, we give notational and mathematical preliminaries together with the formal definition of the problem statement. In Section 3, we present an algorithm for the computation of an auxiliary data structure, the subtree jump table. In Section 4, we present our automata approach. In Section 5, we show a dynamic programming algorithm that simulates the nondeterministic finite automaton. In Section 6, we conclude the paper and discuss the future work.

2 Preliminaries

An *alphabet*, denoted by Σ, is a finite nonempty set whose elements are called *symbols*. A *string* over Σ is a finite sequence of elements of Σ. The empty sequence is called the *empty string* and is denoted by ε. *The set of all strings* over Σ is denoted by Σ^*. The *length* of string $\boldsymbol{x}$ is the length of the sequence associated with $\boldsymbol{x}$ and is denoted by $|\boldsymbol{x}|$. By $\boldsymbol{x}[i]$, where $i \in \{1, \ldots, |\boldsymbol{x}|\}$, we denote the symbol at index i of $\boldsymbol{x}$. The substring of $\boldsymbol{x}$ that starts at index i and ends at index j is denoted by $\boldsymbol{x}[i \ldots j]$; i.e., $\boldsymbol{x}[i \ldots j] = \boldsymbol{x}[i]\boldsymbol{x}[i+1] \ldots \boldsymbol{x}[j]$. A *language* over an alphabet Σ is a set of strings over Σ.

2.1 Trees

A *tree* is a graph $T = (V, E)$, where V represent the nonempty set of vertices and E the set of edges, and where one of its vertices is called the *root* of the tree; the remaining vertices are called *descendants* of the root and can be partitioned into $s \geq 0$ disjoint sets $T_1, \ldots, T_s$, and each of these sets, in turn, is a tree. The trees $T_1, \ldots, T_s$ are called the *subtrees* of the root. There is an edge from the root to the root of each subtree. An *ordered* tree is a tree where the relative order of the subtrees $T_1, \ldots, T_s$ is important.

If a tree is equipped with a vertex labeling function $V \to \Sigma$, we call it *a labeled tree* over Σ. The set of all labeled ordered trees over alphabet Σ is denoted by $\mathrm{T}\textsc{r}(\Sigma)$. All trees we consider in the paper are labeled ordered trees. Therefore, we will omit the words “labeled ordered” when referencing labeled ordered trees.

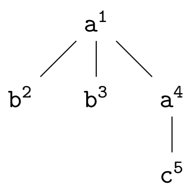
By T_v, where $v \in V$, we denote the *subtree of T rooted at vertex v*; i.e., T_v is a subgraph of tree T induced by vertex subset V' that contains vertex v (the root of tree T_v), and all its descendants. If a vertex does not have any descendants, it is called a *leaf*.

The prefix bar notation [5] of tree T, denoted by $\textsc{pref-bar}(T)$, is defined as follows: If T contains only the root vertex with no subtrees, then $\textsc{pref-bar}(T) = a|$, where a is the label of the root vertex. Otherwise,

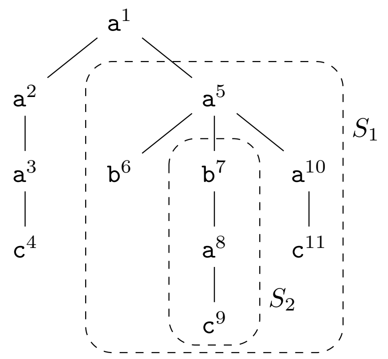
$$\textsc{pref-bar}(T) = a\,\textsc{pref-bar}(T_1)\,\textsc{pref-bar}(T_2) \cdots \textsc{pref-bar}(T_s)\,|,$$

where a is the label of the root of T and $T_1, \ldots, T_s$ are its subtrees. For a tree T with n vertices, the prefix bar notation is always of length $2n$. For every label a in the prefix bar notation, there is the corresponding bar symbol “|” indicating the end of the subtree T_a; we call such pair of a label and its corresponding bar symbol a *label-bar pair*.

The *subtree jump table* for prefix bar notation is a linear auxiliary structure introduced by Trávníček [11] that contains the start and the end position of each subtree for trees represented in the prefix bar notation. Formally, given a tree T with n vertices and its prefix bar notation $\textsc{pref-bar}(T)$ with length $2n$, the subtree jump table $\boldsymbol{S}_T$ for T is a mapping from a set of integers $\{1, \ldots, 2n\}$ into a set of integers $\{0, \ldots, 2n+1\}$. If the substring $\boldsymbol{x}[i \ldots j]$, where $1 \leq i < j$ is the prefix bar representation of a subtree of T, then $\boldsymbol{S}_T[i] = j+1$ and $\boldsymbol{S}_T[j] = i-1$. In the prefix bar notation, it holds that every subtree of tree T is a substring of $\textsc{pref-bar}(T)$. It also holds that every such substring ends with the bar symbol. When discussing the time complexity of our algorithms, we will assume that the subtree jump table is implemented as an array, and therefore each position $i \in \{1, \ldots, 2n\}$ can be accessed in $\mathcal{O}(1)$ time.



(a) Example of tree pattern P.



(b) Example of input tree T and its subtrees S_1, S_2.

Figure 1: Example ordered labeled trees.

Example 1 (Prefix bar notation and subtree jump table). Let P be a tree illustrated in Figure 1a. Then, PREF-BAR$(P) = \boldsymbol{p} =$ ab|b|ac|||. The subtree jump table $\boldsymbol{S}_P$ for P is as follows:

$\boldsymbol{p}$	a	b	|	b	|	a	c	|	|	|
j	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{S}_P[j]$	11	4	1	6	3	10	9	6	5	0

2.2 Pushdown and finite automaton

An *(extended) pushdown automaton* (PDA) is a 7-tuple $\mathcal{M}_{\text{PDA}} = (Q, \Sigma, G, \delta, q_0, \mathtt{Z}, F)$ where Q is a finite set of states, Σ is an input alphabet, G is a pushdown store alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times G^* \to \mathcal{P}(Q \times G^*)$ is a transition function (not necessarily total), where $\mathcal{P}(Q \times G^*)$ contains only finite subsets of $Q \times G^*$, $q_0 \in Q$ is the initial state, $\mathtt{Z} \in G$ is the initial pushdown symbol, $F \subseteq Q$ is the set of final states. By $L(\mathcal{M}_{\text{PDA}})$ we denote the language accepted by $\mathcal{M}_{\text{PDA}}$ by a final state.

A *nondeterministic finite automaton* (NFA) is a 5-tuple $\mathcal{M}_{\text{NFA}} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is a state transition function (not necessarily total), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states. A finite automaton is *deterministic* (DFA) if $\forall a \in \Sigma$ and $q \in Q : |\delta(q, a)| \leq 1$. By $L(\mathcal{M}_{\text{FA}})$ we denote the language accepted by $\mathcal{M}_{\text{FA}}$.

2.3 Problem statement

In the Introduction, we have defined the problem of inexact tree pattern matching as finding the subtrees in an input tree that match a tree pattern with up to k errors. We now give a more formal definition for which we consider the following primitive operations applied to a tree $T = (V, E)$:

vertex relabel change the label of a vertex v,
leaf insert insert a vertex v as a leaf of an existing vertex u in V, and
leaf delete delete a non-root leaf v from T.

The operations may be used recursively to allow insertion or deletion of a subtree of arbitrary size. This set of operations was originally introduced by Selkow [8], and we will refer to it as to the set of *1-degree edit operations.*

The *unit cost 1-degree edit distance* is a function $d : \text{Tr}(\Sigma) \times \text{Tr}(\Sigma) \to \mathbb{N}_0$. Given two trees T_1 and T_2, the number $d(T_1, T_2)$ corresponds to the minimal number of 1-degree edit operations that transform T_1 into T_2.

Example 2 (1-degree edit distance). Let P be the tree illustrated in Figure 1a and S_1 be the tree illustrated in Figure 1b. Then, $d(P, S_1) = 2$ since we need to insert a leaf labeled by `a` as a child of the vertex with identifier 3 into P. Then, we add the leaf with label `c` as the child of the node `a` we inserted in the previous step.

Definition 3 (Inexact 1-degree tree pattern matching problem). *Let Σ be an alphabet. Let $T = (V_T, E_T)$ be an input tree with n vertices over Σ. Let $P = (V_P, E_P)$ be a comparatively smaller tree pattern over Σ with m vertices. Let k be a non-negative integer representing the maximum number of errors allowed. Let d be the unit cost 1-degree edit distance function. Given T, P, k, and d, the inexact 1-degree tree pattern matching problem is to return a set*

$$\Big\{v : v \in V_T \wedge d(T_v, P) \leq k\Big\}.$$

In other words, the problem is to return the set of all vertices such that each vertex v represents the root of a subtree of T which distance from the tree pattern P is at most k.

Example 4 (Inexact 1-degree tree pattern matching problem). Let P be the tree pattern illustrated in Figure 1a, T be the input tree shown in Figure 1b, and $k = 2$. The solution of the 1-degree inexact tree pattern matching problem is $\{2, 5\}$; i.e., with respect to the maximal number of allowed errors, P occurs in T in the subtrees rooted at nodes 2 and 5.

In the rest of the text, we will use the following naming conventions: T and P will represent an input tree and a tree pattern, respectively. We use n, m, k to represent the number of vertices in T, the number of vertices in P, and the maximum number of errors allowed. For brevity, we will use $\boldsymbol{t}$ and $\boldsymbol{p}$ as a shorthand for PREF-BAR(T) and PREF-BAR(P), respectively.

3 Subtree jump table computation for prefix unranked bar notation

A linear-time algorithm for computation of the subtree jump table was given by Trávníček [11, Section 5.2.2]. However, Trávníček's algorithm works for prefix ranked bar notation, which combines the prefix notation and the bar notation. Therefore, we give an alternative algorithm for computation of the subtree jump table that works directly with prefix (unranked) bar notation of trees (see Algorithm 1).

The central idea of our algorithm is the use of a pushdown store for recording the positions of the labels. When the bar symbol is found in the prefix bar notation, the position of the corresponding label is popped from the pushdown store.

Theorem 5 (Correctness of the subtree jump table computation). *Let $\boldsymbol{p}$ be a string, such that $\boldsymbol{p} =$ PREF-BAR(P) for some tree P. Algorithm 1 correctly computes the subtree jump table for $\boldsymbol{p}$.*

Proof. In the first **for**-loop (line 3), we use the pushdown store to save all indexes (line 7) that correspond to the positions of all vertex labels in the prefix bar notation. When the bar symbol is encountered, the position of the corresponding subtree root label is at the top of the pushdown store; we retrieve it and subtract one (line 5) since

Algorithm 1 Computation of the subtree jump table.

```
Input String $\boldsymbol{p}$, such that $\boldsymbol{p} = \textsc{pref-bar}(P)$ for some tree $P$.
Output The subtree jump table $\boldsymbol{S}_P$ for $\boldsymbol{p}$.
 1 $Y$: empty pushdown store
 2 $\boldsymbol{S}_P$: empty array of size $|\boldsymbol{p}|$
 3 for each position $j$ of $\boldsymbol{p}$:
 4     if $\boldsymbol{p}[j] = |$
 5         $\boldsymbol{S}_P[j] \leftarrow \textsc{pop}(Y) - 1$
 6     else
 7         $\textsc{push}(Y, j)$
 8         $\boldsymbol{S}_P[j] \leftarrow \textsc{null}$
 9 for each position $j$ of $\boldsymbol{p}$:
10     if $\boldsymbol{S}_P[j] \neq \textsc{null}$
11         $\boldsymbol{S}_P[\boldsymbol{S}_P[j] + 1] \leftarrow j + 1$
12 return $\boldsymbol{S}_P$
```

the subtree jump table contains the index of the previous element, not the index of the subtree root label itself. In the second **for**-loop (line 9), we define the remaining positions in the subtree jump table. The $\boldsymbol{S}_P[\boldsymbol{S}_P[j] + 1]$ expression (line 11) computes the position of the subtree root label corresponding to the bar symbol at position $\boldsymbol{p}[j]$ and saves there the position $j + 1$ that is the index of the element following the current bar symbol at index j.

Theorem 6 (Time complexity of the subtree jump table computation). *Let $\boldsymbol{p}$ be a string, such that $\boldsymbol{p} = \textsc{pref-bar}(P)$ for some tree P. The subtree jump table for $\boldsymbol{p}$ can be computed in $\mathcal{O}(|\boldsymbol{p}|)$ time using Algorithm 1.*

4 Automata approach

To be able to solve the inexact 1-degree tree pattern matching problem defined in Section 2.3 using (string) automata, we represent trees as strings using the prefix bar notation. Therefore, given a string $\boldsymbol{x} = x_1 x_2 \cdots x_r$ of length $r \geq 2$ over alphabet $\Sigma \cup \{|\}$ that represents the prefix bar notation of a tree, the 1-degree (tree) edit operations corresponds to the following string operations:

- the operation relabeling $\mathrm{R}(i, b)$ that for $i \in \{1, \ldots, r-1\}$, $b \in \Sigma$, and $\boldsymbol{x}[i] \in (\Sigma \setminus \{b\})$, change the symbol $\boldsymbol{x}[i]$ into symbol b;
- the operation insertion $\mathrm{I}(i, a)$ that for $i \in \{2, \ldots, r-1\}$ and $a \in \Sigma$ inserts the substring (leaf) "$a|$" at position i; and
- the operation deletion $\mathrm{D}(i)$ that for $i \in \{2, \ldots r-2\}$, $\boldsymbol{x}[i] \in \Sigma$, and $\boldsymbol{x}[i+1] = |$, deletes the substring (leaf) $\boldsymbol{x}[i]\boldsymbol{x}[i+1]$.

Example 7 (Application of 1-degree edit operations to strings). Let P be the tree illustrated in Figure 1a and S_1, S_2 be the trees illustrated in Figure 1b; $\textsc{pref-bar}(P) =$ `ab|b|ac|||`, $\textsc{pref-bar}(S_1) =$ `ab|bac|||ac|||`, and $\textsc{pref-bar}(S_2) =$ `bac|||`. Then, the distance $d(P, S_1) = 2$ and $d(P, S_2) = 3$ since

$$\mathtt{ab|b|ac|||} \xrightarrow{I(5,\mathrm{a})} \mathtt{ab|ba||ac|||} \xrightarrow{I(6,\mathrm{c})} \mathtt{ab|bac|||ac|||} \text{ and}$$

$$\mathtt{ab|b|ac|||} \xrightarrow{R(1,\mathrm{b})} \mathtt{bb|b|ac|||} \xrightarrow{D(2)} \mathtt{bb|ac|||} \xrightarrow{D(2)} \mathtt{bac|||}.$$

Given two strings $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ that both correspond to the prefix bar notation of trees, using the 1-degree (string) edit operations, we can define the unit cost 1-degree (string) edit distance as a function $d_s : (\Sigma \cup \{|\})^* \times (\Sigma \cup \{|\})^* \to \mathbb{N}_0$ such that $d_s(\boldsymbol{t_1}, \boldsymbol{t_2}) = d(T_1, T_2)$, where $\boldsymbol{t_1} = \text{PREF-BAR}(T_1)$ and $\boldsymbol{t_2} = \text{PREF-BAR}(T_2)$. Since the functions d and d_s differ only in argument types, we will use the notation d for both trees and string arguments.

Using the prefix bar representation of trees, we can specify the problem of inexact 1-degree tree pattern matching as finding all positions $i \in \{1, \dots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i]+1 \dots i]) \leq k$. Recall that $\boldsymbol{S}_T$ represent the subtree jump table for input tree T and $\boldsymbol{S}_T[i] + 1$ returns the position in $\boldsymbol{t}$ that contains the subtree root label corresponding to the bar symbol at position $\boldsymbol{t}[i]$. In other words, our methods output end positions of the occurrences. The position of the corresponding root label can be computed in $\mathcal{O}(1)$ time for each end position using the subtree jump table for T.

Proposition 8. *Let $\mathcal{M}$ be either a pushdown or finite automaton accepting the language*

$$\{\boldsymbol{sp'} : \boldsymbol{s} \in (\Sigma \cup \{|\})^* \wedge d(\boldsymbol{p}, \boldsymbol{p'}) \leq k\}. \tag{1}$$

The automaton $\mathcal{M}$ is called 1-degree automaton and it solves the inexact 1-degree tree pattern matching problem.

The 1-degree automaton $\mathcal{M}$ accepts infinite language. It can read (not necessarily accept) any prefix bar notation of a tree (over alphabet Σ), i.e., it does not fail due to non-existing transition. Algorithm 2 illustrates how $\mathcal{M}$ can be used to solve the inexact 1-degree tree pattern matching problem. In the following sections, we will discuss the construction of the 1-degree automaton in detail.

Algorithm 2 Automata approach to inexact 1-degree tree pattern matching.

Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for tree pattern P over alphabet Σ, a string $\boldsymbol{t}$ of length $2n$ such that $\boldsymbol{t} = \text{PREF-BAR}(T)$ for input tree T over alphabet Σ, a non-negative integer $k \leq m$, a 1-degree automaton $\mathcal{M}$ for P and k.
Output All positions $i \in \{1, \dots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i] + 1 \dots i]) \leq k$.

```
1  read $\boldsymbol{t}$ using $\mathcal{M}$ symbol-by-symbol ($\boldsymbol{t}[i]$ is the currently read symbol):
2      if a final state is reached:
3          output $i$
```

4.1 1-degree pushdown automaton

In this section, we show that the 1-degree automaton can be constructed as a pushdown automaton. Our method is similar to the construction of approximate string pattern matching automaton [6]. Algorithm 3 describes the construction of the 1-degree PDA in detail. An example of $\mathcal{M}_{\text{PDA}}$ is illustrated in Figure 2.

Each state of the automaton has a label j^l, where $0 \leq j \leq 2m$ is a *depth* of the state (position in the pattern) and $l \in \{0, \dots, k\}$ is a *level* of the state (actual number of errors). The pushdown store is used to match label-bar pairs and, therefore, to simulate leaf insertion operation.

Vertex relabeling operation can be applied if there is a different vertex label in $\boldsymbol{p}$ and $\boldsymbol{t}$ at the current position. Each relabel operation increases the distance by 1.

Algorithm 3 Construction of 1-degree pushdown automaton.

Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for a tree pattern P over alphabet Σ, a non-negative integer $k \leq m$, the subtree jump table $\boldsymbol{S}_P$ for P.
Output 1-degree pushdown automaton $\mathcal{M}_{\text{PDA}}$ for P and k.

```
1  define states $Q = \{0^0\} \cup \{j^l : 1 \leq j \leq 2m \wedge 0 \leq l \leq k\}$
2  define final states $F = \{2m^l : 0 \leq l \leq k\}$
3  define pushdown alphabet $G = \{\mathtt{Z}, \mathtt{c}\}$
4  add initial loop for the bar symbol: $\delta(0^0, |, \mathtt{Z}) = \{(0^0, \mathtt{Z})\}$
5  add initial state transitions for labels: $\delta(0^0, a, \mathtt{Z}) = \begin{cases} \{(0^0, \mathtt{Z}), (1^0, \mathtt{Z})\} : a = \boldsymbol{p}[1] \\ \{(0^0, \mathtt{Z}), (1^1, \mathtt{Z})\} : a \in (\Sigma \setminus \{\boldsymbol{p}[1]\}) \end{cases}$
6  for every pattern position $j : 2 \leq j \leq 2m$:
7      for every allowed number of errors $l : 0 \leq l \leq k$:
8          $\delta((j-1)^l, \boldsymbol{p}[j], \mathtt{Z}) = \{(j^l, \mathtt{Z})\}$    (label or bar match)
9          $\delta((j-1)^l, a, \mathtt{Z}) = \{(j^{l+1}, \mathtt{Z}) : l < k\} : a \in \Sigma \setminus \{\boldsymbol{p}[j]\}$    (relabel)
10         $\delta((j-1)^l, a, \varepsilon) = \delta((j-1)^l, a, \varepsilon) \cup \{((j-1)^{l+1}, \mathtt{c}) : l < k\} : a \in \Sigma \setminus \{|\}$    (label insert)
11         $\delta((j-1)^l, |, \mathtt{c}) = \{((j-1)^l, \varepsilon) : l > 0\}$    (bar insert)
12         $\delta((j-1)^l, \varepsilon, \mathtt{Z}) = \{((\boldsymbol{S}_P[j]-1)^{l+(\boldsymbol{S}_P[j]-j)/2}, \mathtt{Z}) : l + \frac{\boldsymbol{S}_P[j]-j}{2} \leq k\} : \boldsymbol{S}_P[j] > j$    (delete)
13 return $\mathcal{M}_{\text{PDA}} = (Q, \Sigma \cup \{|\}, G, \delta, 0^0, \mathtt{Z}, F)$
```

These operations are represented by "diagonal" transitions labeled by the symbols of the alphabet Σ for which no "direct" transition to the next state exists.

Leaf deletion operations correspond to a situation in which a vertex label followed by the bar symbol is skipped in $\boldsymbol{p}$ while nothing is read in $\boldsymbol{t}$. The automaton performs such an operation by following one of its ε-transitions. Since 1-degree edit distance allows to delete a subtree of arbitrary size by picking its leaves one by one, the automaton needs to reflect this. That is why the target state of the ε-transitions is provided by the subtree jump table. The number of errors of such an operation is equal to the number of skipped vertex labels. Since a subtree of tree pattern P is a substring of $\boldsymbol{p}$ where label-bar pairs are balanced, the number of errors is equal to the substring length divided by 2.

Leaf insertion operations correspond to a situation in which a vertex label followed by the bar symbol is read in $\boldsymbol{t}$ while there is no advance in $\boldsymbol{p}$. To allow insertion of a subtree of arbitrary size into the tree pattern leaf by leaf, we use a special pushdown symbol $\mathtt{c}$. By pushing it when a label is read and popping it whenever the bar symbol is encountered, we ensure that the substring represents the correct prefix bar notation of a tree. The insertion operation is complete once the pushdown store contains only the initial pushdown store symbol; this is the only case in which the automaton can again start to advance in $\boldsymbol{p}$.

Note 9. Algorithm 3 can be modified to construct PDA that works with non-unit cost 1-degree edit distance. With unit cost operations, each transition in the PDA (corresponding to some edit operation) goes from a state with level l to a state with level $l+1$. With non-unit cost operations, transitions would go to states with a level that is increased accordingly by the cost of the operation.

4.2 1-degree finite automaton

Due to its restricted use of the pushdown store, we can transform the 1-degree PDA into an equivalent finite automaton. The PDA constructed by Algorithm 3 uses only symbol $\mathtt{c}$ for pushdown store operations (the initial pushdown symbol $\mathtt{Z}$ is never pushed). Moreover, pushdown store operations are only used for insertion operations.

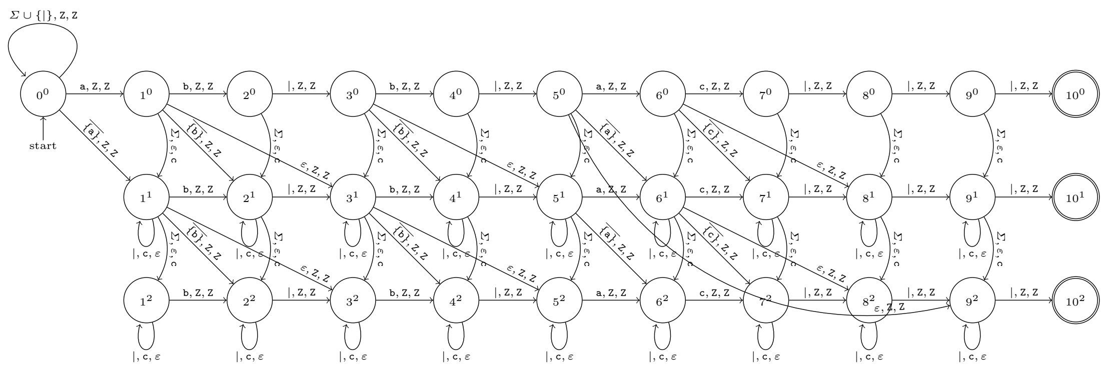


Figure 2: Transition diagram of the 1-degree pushdown automaton for tree pattern P from Figure 1a and $k = 2$. The double-circled nodes correspond to final states. The edge labeled x, y, z from state q_1 to state q_2 corresponds to transition $\delta(q_1, x, y) = \{(q_2, z)\}$. The complement $\overline{a}$ means $\Sigma \setminus \{a\}$.

Since the number of editing operations is limited by $k \leq m$, the length of the pushdown store is also bounded by k. In other words, the pushdown store serves as a bounded counter. Therefore, we can represent each possible content of the pushdown store by a state. The construction of the 1-degree nondeterministic finite automaton is described by Algorithm 4. It reuses the $\mathcal{M}_{\text{PDA}}$ structure and construction steps. The only difference is the use of states j^l_c $(c > 0)$ representing a situation where the pushdown store contained c symbols.

An example of $\mathcal{M}_{\text{NFA}}$ is depicted in Figure 3. The set of active states of this automaton while reading the input tree T illustrated in Figure 1b is shown in Table 1.

Algorithm 4 Construction of 1-degree nondeterministic finite automaton.

```
Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for the tree pattern $P$ over alphabet $\Sigma$, a non-negative integer $k \leq m$, the subtree jump table $\boldsymbol{S}_P$ for $\boldsymbol{p}$.
Output 1-degree nondeterministic finite automaton $\mathcal{M}_{\text{NFA}}$ for $P$ and $k$.
1  define states $Q = \{0^0_0\} \cup \{j^l_c : 1 \leq j \leq |\boldsymbol{p}| \wedge 0 \leq l \leq k \wedge 0 \leq c \leq k\}$
2  define final states $F = \{|\boldsymbol{p}|^l_0 : 0 \leq l \leq k\}$
3  add an initial loop for the bar symbol: $\delta(0^0_0, |) = \{0^0_0\}$
4  add initial state transitions for labels: $\delta(0^0_0, a) = \begin{cases} \{0^0_0, 1^0_0\} : a = \boldsymbol{p}[1] \\ \{0^0_0, 1^1_0\} : a \in \Sigma \setminus \{\boldsymbol{p}[1]\} \end{cases}$
5  for every pattern position $j : 2 \leq j \leq |\boldsymbol{p}|$:
6      for every allowed number of errors $l : 0 \leq l \leq k$:
7          $\delta((j-1)^l_0, \boldsymbol{p}[j]) = \{j^l_0\}$                                   (label or bar match)
8          $\delta((j-1)^l_0, a) = \{j^{l+1}_0 : l < k\} : a \in \Sigma \setminus \{\boldsymbol{p}[j]\}$                (relabel)
9          for each counter value $c : 0 \leq c < k$:
10             $\delta((j-1)^l_c, a) = \delta((j-1)^l_c, a) \cup \{(j-1)^{l+1}_{c+1} : l < k\} : a \in \Sigma \setminus \{|\}$   (label insert)
11             $\delta((j-1)^l_{c+1}, |) = \{(j-1)^l_c\}$                                  (bar insert)
12         $\delta((j-1)^l_0, \varepsilon) = \{(\boldsymbol{S}_P[j]-1)^{l+(\boldsymbol{S}_P[j]-j)/2}_0 : l + \frac{\boldsymbol{S}_P[j]-j}{2} \leq k\} : \boldsymbol{S}_P[j] > j$   (delete)
13 return $\mathcal{M}_{\text{NFA}} = (Q, \Sigma \cup \{|\}, \delta, 0^0_0, F)$
```

Because any NFA can be algorithmically transformed into a DFA, a deterministic finite automaton can be used for inexact 1-degree tree pattern matching. In such case, the set of all positions $i \in \{1, \ldots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i] + 1 \ldots i]) \leq k$ can be computed in $\mathcal{O}(n)$ time. However, the issue can be the size of the deterministic automaton, which can be exponential in the number of vertices of the tree pattern [4]. Therefore, in the next section, we also present how

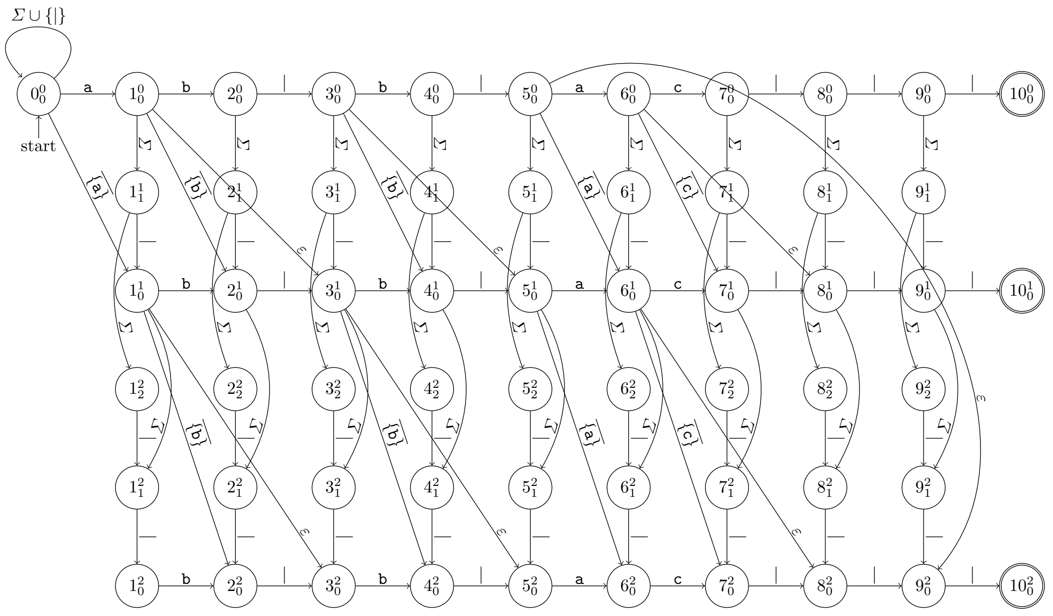


Figure 3: Transition diagram of the 1-degree NFA for tree pattern P illustrated in Figure 1a and $k = 2$. The double-circled nodes correspond to final states. The complement $\overline{a}$ means $\Sigma \setminus \{a\}$.

$\boldsymbol{t}$	a	a	a	c	|	|	|	a	b	|	b	a	c	|	|	|	a	c	|	|	|	|
0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0	0_0^0
	1_0^0	1_0^0	1_0^0	1_0^1	1_0^1	1_0^2	10_0^2	1_0^0	1_0^1	1_0^1	1_0^1	1_0^0	1_0^1	3_0^2	4_0^2	5_0^2	1_0^0	1_0^1	3_0^2	9_0^2	10_0^2	
		1_1^1	1_1^1	1_1^1	3_0^1	3_0^2			1_1^1	3_0^0	2_0^1	2_0^2	3_1^2	5_0^2			6_0^2	3_1^2	3_0^1			
		2_0^1	2_0^1	2_0^1	5_0^2	9_0^2			2_0^0	5_0^1	4_0^2	1_1^2	4_0^2	3_0^1				2_0^1	1_0^1			
		4_0^2	4_0^2	4_0^2	1_1^2				4_0^1	3_0^2	1_1^2	2_1^2	2_0^1	1_0^1				1_1^1	8_0^2			
		6_0^2	6_0^2	1_2^2	2_0^2				3_1^2		4_0^0	4_1^1	1_1^1	4_1^2				7_0^2				
		3_1^2	3_1^2	2_1^2	8_0^2						5_1^2		4_2^2									
			2_1^2	7_0^2	3_0^2						6_0^2											
			1_2^2	3_1^2																		

Table 1: Active states of $\mathcal{M}_{\text{NFA}}$ from Figure 3 for the input tree illustrated in Figure 1b.

dynamic programming can be used to simulate the nondeterministic finite automaton to achieve better space complexity.

5 Dynamic programming

An alternative approach to the use of $\mathcal{M}_{\text{DFA}}$ for inexact 1-degree tree pattern matching is a run simulation of $\mathcal{M}_{\text{NFA}}$ constructed by Algorithm 4. For such a simulation, an approach based on dynamic programming is presented in this section.

Algorithm 2 that uses $\mathcal{M}_{\text{NFA}}$ can be simulated by a three-dimensional array $\boldsymbol{D}$. Each field of $\boldsymbol{D}$ represents possibly active states of $\mathcal{M}_{\text{NFA}}$. More precisely, the first dimension $\boldsymbol{D}_i$ stands for the number of read symbols from $\boldsymbol{t}$; the second dimension $\boldsymbol{D}_{i,j}$ represents the portion of successfully matched pattern (i.e., when state j_c^l is active, $\boldsymbol{D}_{i,j}$ corresponds to j); finally, the third dimension $\boldsymbol{D}_{i,j,c}$ represents the (possibly) unbalanced symbol-bar pair (i.e., when state j_c^l is active, $\boldsymbol{D}_{i,j,c}$ corresponds to c). The value in $\boldsymbol{D}_{i,j,c}$ represents the distance—when state j^l_c is active, the value corresponds to l; the value ∞ represents the situation when no corresponding state of $\mathcal{M}_{\mathrm{NFA}}$ is active. Each field value is computed from other fields value based on the transition function δ of $\mathcal{M}_{\mathrm{NFA}}$.

The part of $\boldsymbol{D}$ recording computation before reading any symbol (i.e., $\boldsymbol{D}_{0,j,c}$) corresponds to the set of active states of $\mathcal{M}_{\mathrm{NFA}}$: the initial state 0^0_0 only. Due to the self-loop in state 0^0_0, the initial state remains active after reading any symbol from the input. This corresponds to value 0 in $\boldsymbol{D}_{i,0,0}$. The initialization of $\boldsymbol{D}$ is formally given in (2).

$$\begin{aligned}&\forall c,i : 0 \le c \le, 0 \le i \le 2n : \boldsymbol{D}_{i,0,c} = \begin{cases} 0 : c = 0, \\ \infty : c > 0 \end{cases} \\ &\forall c,j : 0 \le c \le k, 1 \le j \le 2m : \boldsymbol{D}_{0,j,c} = \infty\end{aligned} \tag{2}$$

When matching a symbol without an edit operation, i.e., reading the same symbol from both $\boldsymbol{p}$ and $\boldsymbol{t}$, the transition in $\mathcal{M}_{\mathrm{NFA}}$ goes from state $(j-1)^l_0$ to state j^l_0. Reading from both $\boldsymbol{p}$ and $\boldsymbol{t}$ means increasing both i and j dimensions in $\boldsymbol{D}$. Matching symbols without an edit operation in $\boldsymbol{D}$ is formally given in (3).

$$\boldsymbol{D}_{i,j,0} = \boldsymbol{D}_{i-1,j-1,0} : \boldsymbol{t}[i] = \boldsymbol{p}[j] \wedge 1 \le i \le 2n \wedge 1 \le j \le 2m \tag{3}$$

Representation of vertex relabeling operation in $\mathcal{M}_{\mathrm{NFA}}$ is similar to matching symbols without an edit operation. Relabeling vertices in $\boldsymbol{D}$ is formally given in (4).

$$\boldsymbol{D}_{i,j,0} = \boldsymbol{D}_{i-1,j-1,0} + 1 : \boldsymbol{t}[i], \boldsymbol{p}[j] \in \Sigma \wedge 1 \le i \le 2n \wedge 1 \le j \le 2m \tag{4}$$

In $\mathcal{M}_{\mathrm{NFA}}$, leaf deletion operation is represented by an ε-transition: skipping part of $\boldsymbol{p}$ (the length of the skip is given by the subtree jump table $\boldsymbol{S}_P$) while reading nothing from $\boldsymbol{t}$. These ε-transitions can be (using standard algorithm) replaced by symbol transitions. More precisely, the ε-transition from state $q^{l_1}_0$ to state $r^{l_2}_0$ can be interpreted as transition from state $q^{l_1}_0$ using symbol $\boldsymbol{p}[q+1]$ to state $(r+1)^{l_2}_0$. Also, considering the sequence of operations delete and relabel, the ε-transition can be interpreted as transitions from state $q^{l_1}_0$ using symbols $\Sigma \setminus \{\boldsymbol{p}[q+1]\}$ to state $(r+1)^{l_2+1}_0$. By contrast, the sequence of transitions for operations delete and insert is not considered in the simulation, as it cannot find more matches than single operation relabel. While $\mathcal{M}_{\mathrm{NFA}}$ skips a leaf "forward", during the computation of a value in $\boldsymbol{D}$, we look "backward". Note that in $\mathcal{M}_{\mathrm{NFA}}$, there can be chains of ε-transitions that correspond to deleting multiple leaves (siblings). This is done in $\boldsymbol{D}$ by multiple evaluation of $\boldsymbol{S}_P$. Deleting from the pattern in $\boldsymbol{D}$ is formally given in (5) and (6).

$$\begin{aligned}\boldsymbol{D}_{i,j,0} =& \boldsymbol{D}_{i-1,\boldsymbol{S}_P[h],0} + \frac{j - \boldsymbol{S}_P[h] + 1}{2} : \boldsymbol{t}[i] = \boldsymbol{p}[j] \wedge \boldsymbol{p}[j-1] = | \wedge \\ &\wedge 1 \le i \le 2n \wedge 2 \le j \le 2m \wedge 1 \le h \le 2m \wedge \boldsymbol{S}_P[h] < j\end{aligned} \tag{5}$$

$$\begin{aligned}\boldsymbol{D}_{i,j,0} =& \boldsymbol{D}_{i-1,\boldsymbol{S}_P[h],0} + \frac{j - \boldsymbol{S}_P[h] + 2}{2} : \boldsymbol{t}[i], \boldsymbol{p}[j] \in \Sigma \wedge \boldsymbol{t}[i] \ne \boldsymbol{p}[j] \wedge \\ &\wedge \boldsymbol{p}[j-1] = | \wedge 1 \le i \le 2n \wedge 2 \le j \le 2m \wedge 1 \le h \le 2m \wedge \boldsymbol{S}_P[h] < j\end{aligned} \tag{6}$$

In $\mathcal{M}_{\mathrm{NFA}}$, leaf insertion operation is represented by a pair of transitions and states: from state j^l_c to state j^{l+1}_{c+1} (read a vertex label from $\boldsymbol{t}$ and record an unbalanced symbol) and from state j_c^l to state j_{c-1}^l (read the bar from $\boldsymbol{t}$ and record a balanced symbol-bar pair). It is not possible to use any transition besides insert until the inserted labels and bars are balanced. To track the balance between inserted labels and bars, the third dimension of $\boldsymbol{D}$ is used. Inserting into the pattern in $\boldsymbol{D}$ is formally given in (7) and (8).

$$\boldsymbol{D}_{i,j,c} = \boldsymbol{D}_{i-1,j,c-1} + 1 : \boldsymbol{t}[i] \in \Sigma \wedge 1 \leq i \leq 2n \wedge 1 \leq j \leq 2m \wedge 1 \leq c \leq k \tag{7}$$

$$\boldsymbol{D}_{i,j,c} = \boldsymbol{D}_{i-1,j,c+1} : \boldsymbol{t}[i] = | \wedge 1 \leq i \leq 2n \wedge 1 \leq j \leq 2m \wedge 0 \leq c < k \tag{8}$$

The previous expressions do not limit the values stored in the cells in $\boldsymbol{D}$. However, only values between 0 and k are useful. This is summarized in the following proposition.

Proposition 10 (Distance value representation in $\boldsymbol{D}$-table). *In $\mathcal{M}_{\text{NFA}}$, there exists no state j_c^l with $l > k$. Therefore, the field values in the $\boldsymbol{D}$-table greater than k can be represented by ∞.*

Among the active states in $\mathcal{M}_{\text{NFA}}$, there can be those of the same depth but different level; for example, states 5_0^0 and 5_0^2. (See Example 11 that shows such a situation.) However, to solve the inexact 1-degree tree pattern matching problem, we do not need multiple integers to represent multiple possibly active states j_0^l and $j_0^{l'}$ in $\boldsymbol{D}_{i,j,0}$. This is summarized in Lemma 12.

Example 11. Let $\mathcal{M}_{\text{NFA}}$ be the NFA depicted in Figure 3. After reading string `ab|b|`, the set of active states of $\mathcal{M}_{\text{NFA}}$ is $\{0_0^0, 1_0^2, 3_0^1, 5_0^0, 5_0^2\}$.

Lemma 12. *Storing only single integer in every field $\boldsymbol{D}_{i,j,c}$ is sufficient for correct solution of the problem from Definition 3.*

Proof. Although $\mathcal{M}_{\text{NFA}}$ can have multiple active states for the same c and j, only states with the smallest l are interesting for solving the problem from Definition 3. If the state $j_c^{l'}$ where $l' > l$ is not considered active, no occurrence of the pattern can be missed, as due to regular structure of $\mathcal{M}_{\text{NFA}}$, there is no additional path from such state $j_c^{l'}$ to a final state compared to state j_c^l. Storing only the minimum integer in $\boldsymbol{D}$ corresponds to considering only the state with the minimum l active.

The simulation of $\mathcal{M}_{\text{NFA}}$ for inexact 1-degree tree pattern matching is summarized in Algorithm 5. See an example of the computation in Table 2.

		1	2	3	4	5	6	7	8	9	10	11	
		a	a	a	c	|	|	|	a	b	|	b	...
	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	$0, \infty, \infty$	
a	∞, ∞, ∞	$0, \infty, \infty$	$0, 1, \infty$	$0, 1, 2$	$1, 1, 2$	$1, 2, \infty$	$2, \infty, \infty$	∞, ∞, ∞	$0, \infty, \infty$	$1, 1, \infty$	$1, \infty, \infty$	$1, 2, \infty$	
b	∞, ∞, ∞	∞, ∞, ∞	$1, \infty, \infty$	$1, 2, \infty$	$1, 2, \infty$	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$0, \infty, \infty$	∞, ∞, ∞	$1, \infty, \infty$	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$1, \infty, \infty$	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$0, \infty, \infty$	∞, ∞, ∞	
b	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	$2, \infty, \infty$	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$1, \infty, \infty$	∞, ∞, ∞	$0, \infty, \infty$	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$1, \infty, \infty$	∞, ∞, ∞	
a	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	
c	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	
|	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	$2, \infty, \infty$	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞	

Table 2: Example of dynamic programming computation for $k = 2$, the tree pattern from Figure 1a, and a part of the input tree from Figure 1b. An occurrence is found at position $i = 7$.

Algorithm 5 Simulation of 1-degree nondeterministic finite automaton.

Input A string $\boldsymbol{p}$ of length $2m$ such that $\boldsymbol{p} = \text{PREF-BAR}(P)$ for tree pattern P over alphabet Σ, a string $\boldsymbol{t}$ of length $2n$ such that $\boldsymbol{t} = \text{PREF-BAR}(T)$ for tree T over alphabet Σ, a non-negative integer k such that $k \leq m$.

Output All positions $i \in \{1, \dots, 2n\}$ in $\boldsymbol{t}$ such that $\boldsymbol{t}[i] = |$ and $d(\boldsymbol{p}, \boldsymbol{t}[\boldsymbol{S}_T[i] + 1 \dots i]) \leq k$.

```
compute the subtree jump table $\boldsymbol{S}_P$ for $P$ using Algorithm 1
initialize $\boldsymbol{D}$ according to (2)
for each position index $i$ of $\boldsymbol{t}$:
    for each position index $j$ of $\boldsymbol{p}$:
        compute cell $(i, j, c)$ of $\boldsymbol{D}$ as the minimum from (applicable only)
            match according to (3) ($c = 0$ only)
            relabel according to (4) ($c = 0$ only)
            for each counter value $c : 0 \leq c \leq k$:
                insert into the pattern (the bar is pending) according to (7)
                insert into the pattern (match the bar) according to (8)
            delete subtree(s) from the pattern ($c = 0$ only):
                $h = j - 1$
                while $\boldsymbol{S}_P[h] < j$:
                    consider value according to (5) and (6)
                    $h \leftarrow \boldsymbol{S}_P[h]$
    if $\boldsymbol{D}_{i,|\boldsymbol{p}|,0} \leq k$:
        output $i$
```

Theorem 13 (Space complexity). *The problem from Definition 3 can be solved using $\mathcal{O}(km)$ space by Algorithm 5.*

Proof. During the computation of the value of $\boldsymbol{D}_{i,j,c}$, only two columns (i-th and $(i-1)$-th) of $\boldsymbol{D}$ are needed in the memory. Each column contains $2m + 1$ rows (each for one position in $\boldsymbol{t}$ plus the 0-th row). Each row stores $k + 1$ integers (each for one distinct c value), while their possible and useful values are between 0 and k (plus one additional for all the values greater than k, according to Proposition 10), thus each of these integers may be represented by $\lfloor \log_2(k + 1) \rfloor + 1$ bits. Therefore, the entire $\boldsymbol{D}$-table requires $2(2m + 1)(k + 1)(\lfloor \log_2(k + 1) \rfloor + 1)$ bits. Also, $\boldsymbol{S}_P$ and $\boldsymbol{p}$ need to be stored. Array $\boldsymbol{S}_P$ contains $2m$ integers of values between 0 and $2m + 1$, thus requires $2m(\lfloor \log_2(2m + 1) \rfloor + 1)$ bits. String $\boldsymbol{p}$ contains $2m$ characters that are either the bar or from Σ, thus requires $2m(\lfloor \log_2(|\Sigma| + 1) \rfloor + 1)$ bits. In total, it is $2(2km+2m+k+1)(\lfloor \log_2(k+1) \rfloor+1)+4m(\lfloor \log_2(2m+1) \rfloor+\lfloor \log_2(|\Sigma|+1) \rfloor+2)$ bits, i.e., $\mathcal{O}(km \log k + \log |\Sigma|)$. When considering integer and symbol encoding independent of the tree size and the alphabet, we get $\mathcal{O}(km)$.

Theorem 14 (Time complexity). *The problem from Definition 3 can be solved in $\mathcal{O}(kmn)$ time by Algorithm 5.*

Proof. The subtree jump table is computed in $\mathcal{O}(m)$ time. There are $\mathcal{O}(mn)$ match and relabel computations, each needs $\mathcal{O}(1)$ time. There are $\mathcal{O}(kmn)$ insert computations, each in $\mathcal{O}(1)$ time. The number of delete computations depends, besides mn, on number of subtree skips, which is $\mathcal{O}(m)$. Effectively, the number of subtree skips is limited by k, as there is no point to skip subtree(s) with more than k vertices. Therefore, there are $\mathcal{O}(kmn)$ delete computations, each takes $\mathcal{O}(1)$ time.

Recall that the bar position i in $\boldsymbol{t}$ returned by Algorithm 5 corresponds to the vertex v in T where v is the root of the found subtree (therefore, it is a correct solution of the problem from Definition 3). Additionally, it is possible to obtain v from i in $\mathcal{O}(1)$ time while still having the $\mathcal{O}(kmn)$ time complexity of Algorithm 5, e.g., by adding pointers to vertices of T into $\textsc{pref-bar}(T)$. This could be done at the cost of adding $2n$ to space complexity.

Note 15. Algorithm 5 can be extended for non-unit cost operations in a straightforward way. When computing the value of a field of $\boldsymbol{D}$, instead of adding one (for an edit operation), we add the value corresponding to the cost of the used edit operation.

6 Conclusions

Inspired by techniques from string matching, we showed that the automata approach can also be used to solve the inexact tree pattern matching problem. To process trees using (string) automata, we represented trees as strings using the prefix bar notation. We considered labeled ordered (unranked) trees and 1-degree edit distance where tree operations are restricted to vertex relabeling, leaf insertion, and leaf deletion. For simplicity, we used the unit cost for all operations. However, the extension of our approach to non-unit cost distance was also discussed.

Given a tree pattern P with m vertices, an input tree T with n vertices, and $k \leq m$ representing the maximal number of errors, we first proposed a pushdown automaton that can find all subtrees in T that match P with up to k errors. Then, we discussed that the pushdown automaton can be transformed into a finite automaton due to its restricted use of the pushdown store. The deterministic version of the finite automaton finds all occurrences of the tree pattern in time linear to the size of the input tree.

We also presented an algorithm based on dynamic programming, which was a simulation of the nondeterministic finite automaton. The space complexity of this approach is $\mathcal{O}(mk)$ and the time complexity is $\mathcal{O}(kmn)$, where m is the number of vertices of the tree pattern, n is the number of vertices of the input tree, and $k \leq m$ represents the number of errors allowed in the pattern. In the paper, we also presented the algorithm for subtree jump table construction for a tree in prefix bar notation where the arity (rank) of each vertex is not known in advance.

In future work, we aim to study the space complexity of the DFA and the time complexity of its direct construction in detail. We also want to experimentally evaluate our algorithms. Bit parallelism can also be explored as way of simulating the NFA for tree pattern matching.

References

1. P. Bille: *Pattern Matching in Trees and Strings*, PhD thesis, University of Copenhagen, 2007.
2. K. Bringmann, P. Gawrychowski, S. Mozes, and O. Weimann: *Tree edit distance cannot be computed in strongly subcubic time (unless APSP can).* ACM Trans. Algorithms, 16(4) 2020.
3. E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann: *An optimal decomposition algorithm for tree edit distance.* ACM Trans. Algorithms, 6(1) Dec. 2010, pp. 1–19.
4. J. Hopcroft, R. Motwani, and J. Ullman: *Introduction to automata theory, languages, and computation*, Pearson Education, Harlow, Essex, 3 ed., 2014.
5. J. Janoušek: *Arbology: Algorithms on trees and pushdown automata*, Habilitation thesis, Brno University of Technology, 2010.
6. B. Melichar: *Approximate string matching by finite automata*, in Computer Analysis of Images and Patterns, Springer Berlin Heidelberg, 1995, pp. 342–349.
7. G. Navarro: *A guided tour to approximate string matching.* ACM Comput. Surv., 33(1) 2001, pp. 31–88.

8. S. M. Selkow: *The tree-to-tree editing problem.* Inf. Process. Lett., 6(6) 1977, pp. 184–186.
9. E. Šestáková, B. Melichar, and J. Janoušek: *Constrained approximate subtree matching by finite automata*, in Proceedings of the Prague Stringology Conference 2018, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2018, pp. 79–90.
10. K.-C. Tai: *The Tree-to-Tree correction problem.* J. ACM, 26(3) 1979, pp. 422–433.
11. J. Trávníček: *(Nonlinear) Tree Pattern Indexing and Backward Matching*, PhD thesis, Faculty of Information Technology, Czech Technical University in Prague, 2018.
12. K. Zhang, R. Statman, and D. Shasha: *On the editing distance between unordered labeled trees.* Inf. Process. Lett., 42(3) May 1992, pp. 133–139.

Pitfalls of Algorithm Comparison

Waltteri Pakalén[1], Hannu Peltola[1], Jorma Tarhio[1], and Bruce W. Watson[2]

[1] Department of Computer Science
Aalto University, Finland
[2] Information Science, Centre for AI Research
School for Data-Science & Computational Thinking
Stellenbosch University, South Africa

Abstract. Why is Algorithm A faster than Algorithm B in one comparison, and vice versa in another? In this paper, we review some reasons for such differences in experimental comparisons of exact string matching algorithms. We address issues related to timing, memory management, compilers, tuning/tune-up, validation, and technology development. In addition, we consider limitations of the widely used testing environments, Hume & Sunday and SMART. A part of our observations likely apply to comparisons of other types of algorithms.

Keywords: exact string matching, experimental comparison of algorithms

1 Introduction

Developing new algorithms is a common research objective in Computer Science, and new solutions are often experimentally compared with older ones. This is especially the case for string matching algorithms. We will consider aspects which may lead to incorrect conclusions while comparing string matching algorithms. We concentrate on running times of exact matching of a single pattern, but many of our considerations likely apply to other variations of string matching or even to other types of algorithms.

Formally, the *exact string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an alphabet Σ, find all the occurrences (including overlapping ones) of P in T. So far, dozens of algorithms have been developed for this problem — see e.g. Faro and Lecroq [6].

Many experimental comparisons of string matching algorithms employ one of two testing environments, Hume & Sunday [11] (HS for short) and SMART [7], for measuring running times. The HS environment consists of a main program and shell scripts. Each algorithm is compiled to a separate executable. Most comparisons applying the HS environment actually use some variation of it, see e.g. Hirvola [9].

SMART [5] is an environment developed by Faro et al. [7], which includes implementations for more than a hundred algorithms for exact string matching. SMART tries to make comparisons easy for a user by offering a user interface and tools for various subtasks of a comparison. The SMART application controls all the runs of algorithms. SMART offers options to present the results in various forms.

The rest of the paper is organized as follows: Section 2 presents general aspects of algorithm comparison; Section 3 reviews issues related to the testing environments HS and SMART; Section 4 studies how running time should be measured; Sections 5 and 6 analyze how cache and shared memory affect running times; Section 7 lists miscellaneous observations; and the discussion of Section 8 concludes the article.

2 General

Comparing the running times of algorithms may seem easy: store the clock time in the beginning, run the algorithm, store the clock time in the end, and calculate the difference. However, the results are valid only for the combination of implementation, input, compiler and hardware used in the same workload.

Comparison of algorithms should be done according to *good measurement practice.* In the case of exact string matching algorithms, it means several things. One should verify that algorithms work properly: whether all the matches are found, and whether the search always stops properly at the end of the text; additionally, it can happen that a match in the beginning or in the end of the text is not correctly recognized. When measuring is focused on performance, it is standard to use at least some level of optimization in the compilation. The measurement should not disturb the work of algorithms. For example, printing of matches during time measurement is questionable because printing produces also additional overhead, which is partly unsynchronized. More generally, one should investigate all possibly measurement disturbances and rule them out, if possible. We think that the preprocessing of a pattern should not be included in the search time because the speed of an algorithm may otherwise depend on the length of the text.

Use of averages easily hides some details. Averages in general or calculations on speed-ups may lead to biased conclusions. One should be especially careful with arithmetic mean [8]. Median would be a better measure than arithmetic mean for many purposes because the effect of outliers is smaller, but computing median requires storing all the individual numbers.

The choice of an implementation language for algorithms usually limits available features: the number of different data types and the exactness of requirements given to them varies. The programming language Java is defined precisely, but it lacks the unsigned integer data type, which is useful for implementing bit-vectors. String type should not be used for serious string matching comparisons. On the other hand, the Java virtual machine adds an additional layer on the top of hardware. The programming language C is flexible, but its standard states quite loose requirements for the precision of integers. With assembly language, it would be possible to produce the most efficient machine code, while losing portability to different hardware. Nevertheless, the programming language C is currently the de facto language for implementing efficient exact string matching algorithms.

The C language standard from 1999 introduced the header file `stdint.h`, which is included via the header `inttypes.h`. The fastest minimum-width integer types designate integer types that are usually fastest to operate with among all the types that have at least the specified width. However, footnote 216 in the standard states 'The designated type is not guaranteed to be fastest for all purposes, if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements'. For example, the choice of certain data types may cause implicit type conversions that may ruin the otherwise fast operation.

The exact-width integer types are ideal for use as bit-vectors. The typedef name `uint`N`_t` designates an unsigned integer type with a width of N bits. These types are optional. However, if the implementation (of a C compiler) provides integer types with widths 8, 16, 32, and 64 bits, it shall define the corresponding typedef names.

Therefore, the exact-width integer types should be available in all the C compilers conforming to the C99 standard.

The change of the running process from one core to another empties cache memories with various degree. Often caches are shared by several cores, slowing down reads from memory and induce annoying variation to the timing of test runs. To avoid it we recommend to use the Linux function `sched_setaffinity` to bind the process to only one processor or core. The use of this function reduced substantially variation in time measurements in our experiments.

3 Aspects on Testing Environments

SMART includes a wide selection of corpora with 15 different types of texts. In addition, it contains implementations over 100 string matching algorithms. Both texts and algorithm implementations serve as valuable reference material for any comparison of exact string matching, regardless of the testing approach used.

Both HS and SMART allow comparison of algorithms compiled with different parameters, like optimization level and buffer size, or with different compilers; unfortunately, this is occasionally also a source of incorrect results.

An advantage of SMART is that it verifies the correctness of a new algorithm by checking the output of the algorithm (the number of matches), but not that the implementation actually matches the target algorithm. In other words, the implementation might be incorrect despite producing correct output. For example, the SMART implementation[1] of the bndm algorithm [15] finds correct matches but it shifts incorrectly to the last found factor of the pattern instead of the last found prefix.

Moreover, unit tests, like the verification aspect in SMART, often fail to capture all erroneous cases. In cases where the verification fails, SMART does not directly support debugging code, therefore one may need at least a separate main program for debugging.

Another means of verifying correctness is to manually inspect the count of pattern occurrences, which is employed in HS. SMART reports the occurrences as average occurrences over a pattern set. These averages are based on integer division which may hide edge problems common in developing string matching algorithms.

The generating of pattern files is delegated to the user in HS. SMART dynamically generates patterns for each experiment, and this has some drawbacks. The pattern sets vary from run to run, which causes unpredictable variation between otherwise identical runs, as well as making debugging cumbersome.

Lastly, SMART cannot be used in the Unix-like subsystems of Microsoft Windows because they do not support shared memory.

4 Measuring Running Time

Background. The C standard library offers only the `clock` function for watching the CPU time usage of processes. To determine the time in seconds, the value returned by the `clock` function should be divided by the value of the macro `CLOCKS_PER_SEC`. Additionally the POSIX standard[2] declares that `CLOCKS_PER_SEC` is defined to be one million in `<time.h>`, and also that ‘the resolution on any particular system need not

[1] Release 13.04
[2] IEEE Std 1003.1-2008

be to the microsecond accuracy'. Essentially, time is an internal counter in Linux that is incremented periodically. A periodic interrupt invokes an interrupt handler that increments the counter by one. At a common 100 Hz frequency, the counter has a granularity of 10 ms as it is incremented every 10 ms.

The POSIX standard offers the `times` function for getting process (and waited-for child process) times. The number of clock ticks per second can be obtained by a call `sysconf(_SC_CLK_TCK)`.

The POSIX function `clock_gettime` returns the current value *tp* for the specified clock, *clock_id*. The struct *tp* is given as a parameter. If `_POSIX_CPUTIME` is defined, implementations shall support the special *clock_id* value `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process. The resolution of the given clock at the `clock_gettime` function is provided with the `clock_getres` function.

The POSIX function `gettimeofday` returns the current time, expressed as seconds and microseconds since the Epoch. Applications should use the `clock_gettime()` function instead of the obsolescent `gettimeofday()` function.

Algorithm timings in HS and SMART. The algorithm timings are affected in at least two ways. First, the functions to read time directly determine the timings. Second, the testing environments alter the internal state of the computer system which introduces interference on the algorithm execution and timing.

HS and SMART use two different functions, `times` and `clock_gettime` respectively, to read time. Their implementations are platform specific, but on x86/Linux they work somewhat similarly. The running times of algorithms in HS are given as user time (`tms_utime`) fetched with the `times` function. So the system time is excluded. Occasionally we have checked that in HS there is not any hidden use of the system time, but have never noticed such use. The function used in SMART, `clock_gettime` additionally improves the granularity to nanoseconds through other means such as interpolating a time stamp counter (TSC) that counts core cycles [2]. Furthermore, SMART includes time spent in user space and kernel space, whereas HS includes only user space time.

The rest of this section deals with time measuring in HS.

Precision of individual search. When the digital clock moves evenly, it is safe to assume that its value is incremented at regular fixed intervals. These time intervals are called *clock ticks*, and they are typically so long that during individual tick several instructions are executed. It this section, the term *clock tick* refers to the precision of time measurements, and it is assumed that the processor time increases one tick at a time. When the measuring of a time interval starts, we fetch the last updated value of the clock, but a part of the current clock tick may be already spent. This time follows the continuous uniform distribution $[0, 1]$. So its mean is 0.5 and variance 1/12. Respectively when the measuring of a time interval ends, possibly a part of the current clock tick may be unspent. This slice follows the continuous uniform distribution $[-1, 0]$.

Thus the time measurement with clock ticks has an inaccuracy which is the sum of two error terms following the above mentioned distributions. When the length of the measured interval is at least one clock tick, the probability density function of the sum is

$$f(x) = \begin{cases} 1 + x \text{ if } -1 \le x \le 0, \\ 1 - x \text{ if } 0 \le x \le 1, \\ 0 \qquad \text{otherwise.} \end{cases}$$

The mean of the inaccuracy caused by clock ticks is 0 and the variance 1/6.

The variance of the inaccuracy caused by clock ticks becomes relatively smaller, when the count of clock ticks increases. An easy way to achieve this is to use a longer text, which is at the same time more representative (statistically). However, it is not advisable to use concatenated multiples of a short text of a few kilobytes, because it is probable that the shifts of patterns start to follow similar sequences in such a case. This happens surely, if the pattern matches the text, assuming that the pattern is moved from left to right, and the shifting logic does not have any random behavior. Therefore the text produced with concatenation will with a high probability show the same statistical peculiarities as the original text element.

There are also other causes for inaccuracy. Generally, all context switches of a process produce some delay, which is very difficult to minimize on a single CPU system. We have noticed that on modern multicore processors it is possible to get a more accurate measurement of used CPU time than with singlecore processors spending similar number of clock ticks. The variance caused by other processes becomes relatively smaller, when the measured time intervals get longer. Then the results are more accurate.

Precision of search with a pattern set. The search with a pattern set brings yet another source of variance to the time measurements. Search for some patterns is more laborious, while others are highly efficient with algorithms tuned for them. This joint impact of the patterns and the algorithms can be seen as samples of the all possible cases between the worst and the best cases of a given algorithm. This kind of variance is minimal with the well known shift-or algorithm [1] where, in practice, a large number of occurrences cause small variation. One may also argue that if certain algorithms have a similar search time, the algorithm with smallest variance can be regarded the best.

When several successive time measurements are done within a relatively short period, it is possible that the unused time slice (before the next clock tick) is utilized in the next time measurement. Thus the time measurements may not be completely independent. In HS and SMART, preprocessing and search are alternating. If the measured time intervals are at least a few clock ticks, there is always sufficient variance that it is unlike that these surplus times cumulate more to either preprocessing or search.

Let us consider the variance of the mean. If the measurements are (statistically) independent, then the variance of the sum of times is the sum of the variances of individual time measurement. If the measurements have the same variance, and if they are independent of each other, the variance V of the mean of r measurements is

$$V\left(\frac{X_1 + X_2 + \cdots + X_r}{r}\right) = \frac{1}{r^2} V(X_1 + X_2 + \cdots + X_r) = \frac{r \cdot V(X_1)}{r^2} = \frac{V(X_1)}{r}$$

If the measured time is zero within the given accuracy of measurements, this could cause a bias in other measurements.

5 Cache Effects

CPU Cache memory is typically divided into three levels: L1, L2, and L3, which are accessed in this order. Caches that distinguish instruction data from other data are identified with suffixes i and d, respectively (e.g. L1i and L1d).

The relationship between subsequent runs and their memory accesses is clear in HS: all the runs of an algorithm are performed in a relatively tight loop. There is little other execution, beside proceeding from one loop iteration to another, between `-p` runs of `prep`, `-e` runs of `exec`[3], and each pattern in the patterns.

In the case of SMART, the relationship between subsequent runs and their memory accesses is more muddied. All the runs of `alg`[4] are executed in their own process. Quantifying execution in between the runs is not straightforward; a lot of it is performed by the operating system when creating and running processes. Moreover, SMART performs its own bookkeeping, such as reading the `search` times and storing them, in between the runs. It is quite possible that some of the residual data is replaced in the cache.

Faro et al. [7] claim that SMART is free of such residual data altogether. However, SMART takes no measures to prevent it and there is no reason why the data could not be accessed in the cache between different runs. In principle, this could be solved if the caches were logically addressed, but caches are often physically addressed. The physical addresses of the shared memory segments remain unchanged throughout the runs. That is, all the runs of a given algorithm reference the same physical addresses over and over.

One case where `alg` is less likely to access residual data in the cache is if all the caches are private, as SMART lets its processes run on any core without pinning them. As a result, if a run and a subsequent run are executed on different cores, the runs fail to look-up the data from their respective caches. That said, shared last level caches are common.

The lack of thread pinning in SMART comes with additional nondeterminism. Any of the processes might be migrated from one core to another during execution, which disrupts multiple aspects of the execution including the caches, branch prediction, and prefetching. Moreover, cache line states may affect cache latencies. A cache line in a shared state may have a different access latency than the cache line in a different state [14], and cache line states are currently unpredictable in SMART.

Lastly, caches also cache instructions. Similar reasoning (as above) applies to reading instructions from the cache between runs. However, string matching algorithms often compile to only a few hundred instructions. Thus, the space they occupy is small. Whether they load from the cache or main memory on the first accesses likely has little overall effect. The same few hundred instructions are referenced continuously, which should always hit the cache after the first accesses. The first accesses are few compared to the overall accesses during string matching.

Experiments. We performed extensive experiments with HS and SMART in order to find out how cache memory affects running times of algorithms. The tests were run in two core Intel Core i7-6500U CPU (Skylake microarchitecture) with 16 GiB

[3] `prep` is the subprogram for preprocessing and `exec` is the subprogram for searching. Their repeats are given with options `-p` and `-e`.

[4] `alg` is the subprogram inside which a particular algorithm is embedded.

DDR3-1600 SDRAM memory. Each core has 32 KiB L1d cache, and 256 KiB exclusive L2 cache, the 4 MiB inclusive L3 cache is shared. The operation system was Ubuntu 16.04 LTS. We used a widely used interface, PAPI [3], for accessing the hardware performance counters and to interpret phenomena during runs. All the running times shown were obtained without PAPI, although the overhead of PAPI was minimal.

To interpret the resulting cache metrics during string matching, the values must be compared to some reference value. String matching algorithms behave very predictably in that the pattern is continuously shifted from left to right over the text without ever backtracking. At each alignment window, i.e. an alignment of the pattern in the text, at least one text character is inspected before possibly shifting the pattern again. Accessing the text character unavoidably fills the corresponding cache line in the highest level cache (L1d). Now, under the assumption that the text does not reside in the cache, the access results in a cache miss across the whole cache hierarchy all the way to main memory. Thus, given maximal shifts of m, the lower bound for cache misses is

$$\lfloor n/max(m, \text{cache line size}) \rfloor \tag{1}$$

where n is the text size. Clearly, multiple accesses to the same cache line in short succession are not going to fill the cache line into the cache over and over again. Hence, a pattern incapable of shifting past whole cache lines only fills a cache line once despite possibly referencing it on multiple alignment windows.

In general, with any shift length, it is fair to assume that a cache line is never filled into the cache more than once. A text character remains in alignment windows spanning at most $2m - 1$ characters. After the alignment windows have passed the text character, it is never referenced again. Modern caches are several times the size of even the larger patterns. Only a bad cache line replacement policy or a bad hardware prefetcher would evict the corresponding cache line during processing the $2m - 1$ window.

The above lower bound ignores memory accesses to the pattern and its preprocessed data structures. However, these realistically cause very few cache misses. As stated above, patterns are relatively small compared to cache sizes. A very rarely occurring pattern might drop cache lines towards its one end but the dropped data is refilled as rarely as the pattern occurs. All in all, the lower bound is the expected number of cache misses during string matching.

The following cache metrics results and other measurements have been collected over a static set of 500 random patterns. All of the experiments used concatenated multiples of 1 MiB prefix of King James Version (KJV) as the text to keep searches over different text sizes as comparable as possible.

In the experiment with HS, each pattern is searched `-e` times such that the effective text size is roughly 100 MiB. A pattern search is repeated until roughly 100 MiB of text has been covered. E.g., `-e` is 100 for 1 MiB text, 50 for 2 MiB text, etc. Such a sliding scale is necessary because short running times involve a large margin of error from the low granularity of the `times` function, while constant `-e` repetitions suitable for small texts cause too long an experiment for larger texts. Ideally, the effective text size would be always 100 MiB but 100 is not divisible by all text sizes (e.g. 6 MiB). So the point is to minimize error introduced by `times`, which we deemed to be minimal at around 100 MiB of text. In the case of SMART, a static pattern set of 500 patterns was multiplied[5] to effectively search 100 MiB of text. For HS, we ported the SMART implementations of algorithms. The algorithms were compiled with `-O3` optimization level. The reported results excluded preprocessing and they are arithmetic means over all of the executions.

Table 1 shows results from running the brute force algorithm ($m = 16$) against KJV on both testing environments. The table also includes the expected cache misses explained above. If the cache misses fall below the expected value on any of the caches, the cache contained residual data.

Table 1. Measured average cache metrics during string matching with the brute force algorithm ($m = 16$).

	Text size (MiB)	1	2	4	8	16	32
	Expected misses	16 384	32 768	65 536	131 072	262 144	524 288
HS	L3 requests	34 303	68 695	139 158	280 662	564 696	1 131 615
	L3 misses	452	3 922	28 459	164 434	389 601	809 080
	L2 requests	37 713	75 630	151 011	301 570	603 800	1 206 983
	L2 misses	21 718	43 453	86 954	174 312	348 535	696 914
	L1d misses	16 434	32 829	65 622	131 502	262 926	525 782
SMART	L3 requests	36 085	72 196	143 657	287 142	574 129	1 146 851
	L3 misses	26 542	53 456	106 510	213 023	426 182	852 296
	L2 requests	37 735	75 478	150 871	301 502	602 771	1 204 804
	L2 misses	21 728	43 616	87 153	174 071	347 920	695 519
	L1d misses	16 559	32 933	65 828	131 599	263 075	525 832

The L1d misses are approximately 16k for every 1 MiB increase in text size, which matches the expected value. That is, the L1d contains no residual data between multiple runs. Additionally, the cache misses strongly suggest that a cache line is only ever loaded into the cache once throughout string matching, as reasoned above. Hypothetically, the L2 prefetchers could thrash L2 and L3, but this seems unrealistic[6].

The most important metric, L3 misses, clearly indicates the existence of residual data in HS for short texts. The L3 misses are too few for the smaller text sizes. That is, many requests to L3 hit instead of miss. The trend is also such that the misses grow at a changing rate. The expectation is a constant increase of the L3 misses since the caches should behave similarly from one text size to another. At minimum, a doubling of the text size causes a doubling of the L3 misses. This only happens towards the largest text sizes, but not the smaller ones.

Figure 1 shows relative increases in running times for the brute force algorithm (bf), Horspool's algorithm [10] (hor), and the sbndmq2 algorithm [4] in HS as a function of text size. The y values are relative changes to the respective running times for the text of 1 MiB (thus y is zero at $x = 1$). In other words, the inverses of search speeds (s/MiB) are compared.

For HS in Figure 1 there is a modest increase between -1% and 14% in the relative running times as the text size grows until it is roughly 1.5 times the size of the last level cache L3. Longer patterns exhibit larger increase because the same amount of cache misses divide over shorter running times. Moreover, the hardware prefetchers have less time to perform their function, which possibly leads to a bigger overlap between demand and prefetch request. Similarly, bf exhibits very little increase because the

[5] This required small changes to the code of SMART.

[6] Note that prefetching to cache may go to a different cache level in some other processors.

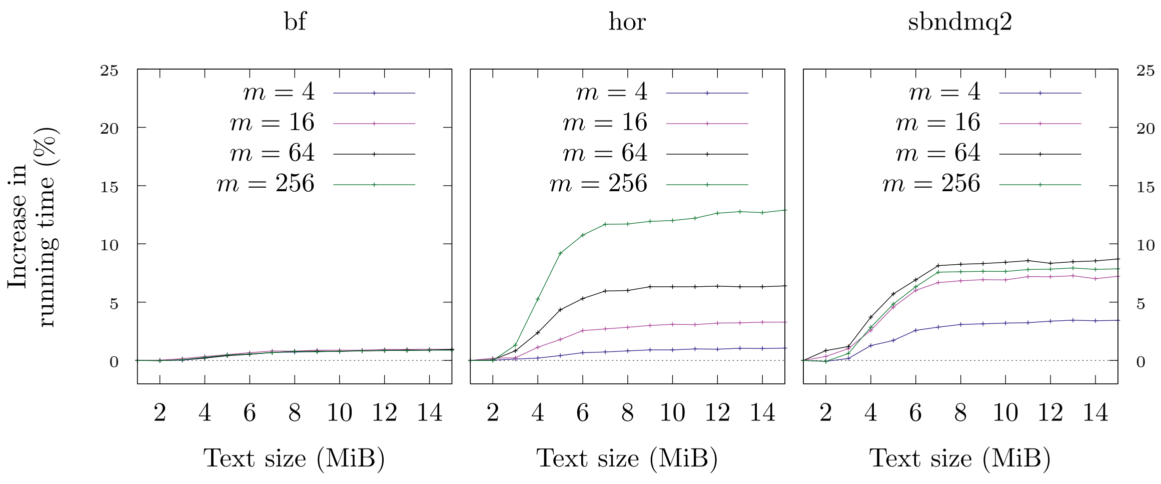


Figure 1. Relative increases in running times during string matching in HS when adjusting text size for three algorithms and four pattern lengths.

running times are already large from its many memory accesses and instructions executed with little variation between pattern sizes.

As Figure 1 shows, the increase of running time depends on the length of the text and the algorithm for a fixed pattern size. So the text should be long enough, 1.5 times the size of L3, that the experiment would be close to a steady state.

We ran a similar experiment to Figure 1 in SMART and repeated it on another computer. The results were incoherent — there was no consistent decrease of speed when the text grows. More investigation would be necessary to understand how cache affects running times of algorithms in SMART.

6 Effects of Shared Memory

The SMART environment [7] uses shared memory for storing the text. The obvious reason for that is to make it easier to execute multiple tests in one run. We noticed inconsistent differences in timing results of certain algorithms in HS and SMART (see Table 3). When we investigated those findings carefully, we found out that the use of shared memory was the reason.

While running the data cache experiments, SMART exhibited unexplained behavior. Invalidating cache lines with `clflush` resulted in significantly faster running times of algorithms. A similar effect was observable whenever the shared memory was touched in any way in `alg` before string matching. The reason turns out to be minor page faults that occur on every first page access which is explored next.

To inspect this, let us count the minor page faults that occur during a string matching. On our test computer, PAPI includes a native event `perf::MINOR-FAULT` to count minor page faults. Figure 2 plots the counts over multiple text sizes for both HS and SMART. The difference between the two is very apparent. HS incurs no page faults during string matching whereas the number of SMART grows linearly at roughly the rate of 250 page faults per 1 MiB of text. With a 4 KiB page size, 250 page faults equate 4 KiB * 250 = 1000 KiB $\approx$ 1 MiB of memory, which matches the text size. Basically, every page backing the shared memory faults once. These minor page faults are irrespective of any other parameter such as the algorithm used, pattern size, the test computer, etc.

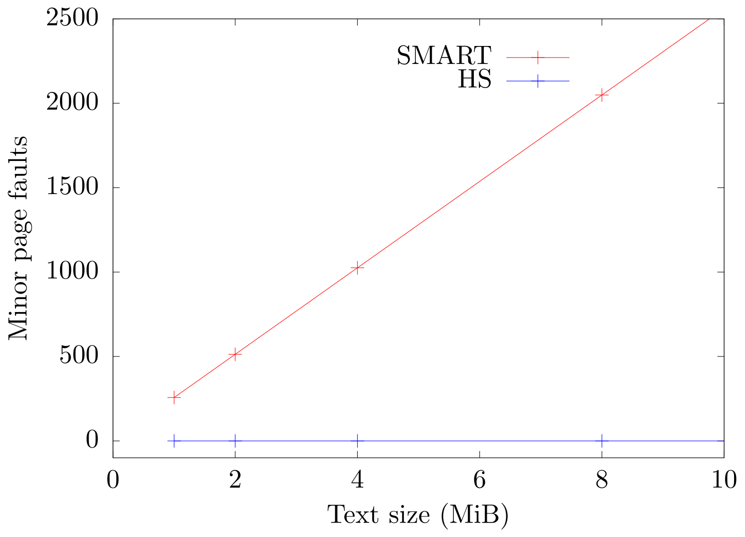


Figure 2. Minor page faults during each string matching.

The reason for the minor page faults is not entirely apparent, but it most likely traces back to the memory management techniques of Linux. Rusling [17] briefly describes System V shared memory in the book *The Linux Kernel*. According to him, attaching a process to shared memory merely modifies the virtual address space of the process, but does not back it up with physical memory. Thus, the first memory access of a process to any page of shared memory generates a page fault which supports the above observation. The actual reason is not further explored and other resources on the topic seem to be scarce. However, Linux generally employs lazy allocation of main memory with techniques such as demand paging, copy-on-write, and memory overcommitment. Perhaps the page faults to shared memory line up with this philosophy.

Whatever the cause, the underlying implications are problematic for running times. First, the continuous interrupts disrupt normal execution of algorithms. The interrupt handler requires context switching, modifying page tables, etc. Second, SMART uses wall-clock time to measure running times. The timings comprise both user space and kernel space execution, which includes time spent on resolving the page faults. These add up given the little time spent on one page.

Figure 3 illustrates these effects. It shows the difference in running times when pages are prefaulted compared to faulting during string matching. The figure shows the differences for multiple pattern lengths for all the implemented algorithms in SMART. Prefaulting can be achieved by explicitly accessing each page or by locking the memory (e.g. with `mlockall`[7]). The former is effectively what invalidating the cache lines did. The latter locks the virtual address space of a process into main memory to ensure it never faults. We used memory locking to measure the prefaulted running times, which additionally required reconfiguring system-wide maximum locked memory size. The change to `alg` is a simple addition given in Figure 4.

Figure 3 reveals insights on the effects of page faults. For $m > 4$, the page faults result in a fairly steady ~1.2 ms average increase in running times. The increase has little variation across all the algorithms, but there still exists a dozen outliers consistently over the different pattern sizes. For $m \leq 4$, the differences seem more erratic

[7] Defined in POSIX Realtime Extensions 1003.1b-1993 and 1003.1i-1995

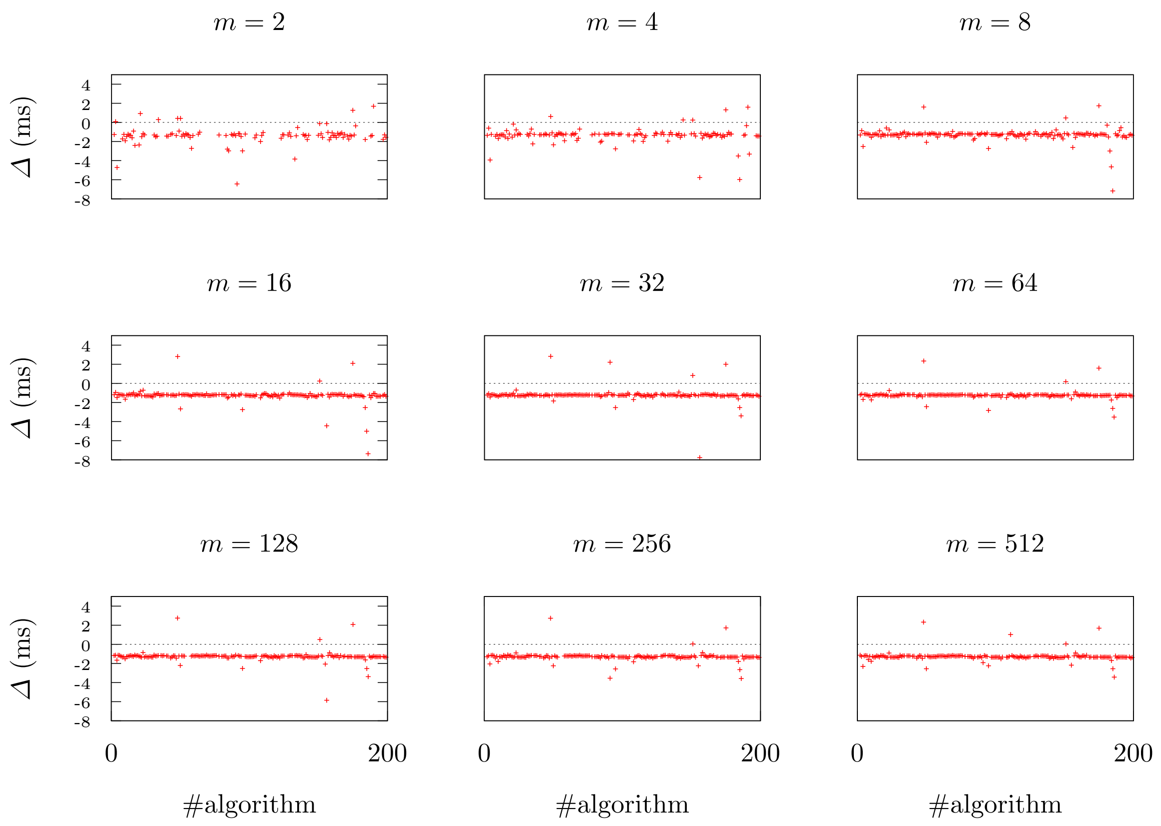


Figure 3. Change in running times (new running time − old running time) for each algorithm in SMART when prefaulting pages on the Skylake computer given 4 MiB KJV.

```
#include <sys/mman.h>
...
if (mlockall(MCL_CURRENT) == -1) {
  perror("mlockall");
  return 1;
}
int count = search(p,m,t,n);
...
```

Figure 4. An addition to `alg` to lock memory.

and, admittedly, a few algorithms deviate from the average increase that otherwise do not. However, less algorithms work for these pattern sizes which contributes to the impression of more deviation.

Figure 3 also gives a view to the accuracy of the time measurement method of SMART. The cause of the farthest outliers needs further study.

Overall, the changes in running times skew algorithm comparisons. Increasing the running time of (almost) every algorithm dilutes relative differences. For example, a closer inspection on the fastest algorithms according to the original running times is presented in Table 2. The largest increase at $m = 512$ is almost 90%. Comparing the original time 1.47 ms to another algorithm with an original time 1.57 ms yields a difference of only ∼7%. Comparing the prefaulted time 0.16 ms to the prefaulted time 0.26 ms of the other algorithm yields a difference of ∼63%, which is almost an order of magnitude different. This effect is more pronounced on large patterns as they tend to be faster. Moreover, the outliers are evaluated unfairly as they might overtake or fall behind other algorithms after prefaulting.

Table 2. Original and prefaulted running times (ms) of the fastest algorithms according to the original running times.

m	2	4	8	16	32	64	128	256	512
Original	1.73	1.86	2.04	1.84	1.57	1.56	1.57	1.52	1.47
Prefaulted	0.54	0.66	0.83	0.62	0.37	0.36	0.25	0.19	0.16

Lastly, the diluted relative differences can be further demonstrated by comparing running times in HS, SMART, and SMART with prefaulting. Table 3 shows such running times for bf, hor, and sbndmq2 introduced in Section 5. The HS and the prefaulted SMART running times are quite close to one another, while the running times of actual SMART are larger. For instance, sbndmq2 is 15% and 16% of the running time of bf in HS and prefaulted SMART, respectively, while it is 27% in SMART.

Table 3. Average per pattern running times (ms) for 8 MiB KJV and $m = 16$.

	bf	hor	sbndmq2
HS	16.31	5.09	2.44
SMART	18.75	7.75	5.04
prefaulted SMART	16.28	5.31	2.61

It is unfortunate that the use of shared memory disturbs running times, though the original aim was obviously to achieve more dependable results. Our correction eliminates the disturbance. However, the correction is rude and it is not yet suitable for production use.

7 Other Issues

The space character is typically the most frequent character in a text of natural language. Therefore, the result of an experimental comparison may depend on whether the patterns contain spaces or not [11,4]. Especially, the space as the last character of a pattern slows down many algorithms of the Boyer–Moore type if the pattern contains another space.

One problem in comparing algorithms is the tuning/tune-up level. Should one compare original versions or versions at the same tune-up level? Skip loop, sentinel, guard, and multicharacter read are all tune-ups which may greatly affect the running time. Even the implementations in the SMART repository are not fully comparable in this respect. For example, in the past it was a well-known fact that the memcmp function is slower than an ordinary match loop. So the SMART implementation[8] of Horspool's algorithm uses a match loop instead of memcmp applied in the original algorithm [10]. However, memcmp is now faster than a match loop on many new processors.

The results of a comparison may depend on the technology used. Thus, results of old comparisons may not hold any more. We demonstrate this by an example. We ran an experiment of two algorithms sbndm4 [4] with 16-bit reads and ufast-rev-md2 [11]

[8] Release 13.04

on two processors of different age. These processors (Intel Pentium and Intel Core i7-4578U) were introduced in 1993 and 2014, respectively. The text was KJV and m was 10. Sbndm4 was considerably faster on i7 — its running time was only 23% of the running time of ufast-rev-md2, but the situation was the opposite on Pentium: the running time of ufast-rev-md2 was 32% of the running time of sbndm4. Potential sources for the great difference are changes in relative memory speed, cache size, and penalty for misaligned memory accesses.

Likewise, two compilers may produce dissimilar results — see, for example, the running time of NSN in [12]. Sometimes, an old algorithm using much memory can become relatively faster in a newer computer — the algorithm by Kim and Shawe-Taylor [13] is such an example. This reflects another downside of technology development: you may find an old "inefficient" and rejected algorithm idea of yours has recently become viable, and is then published by someone else.

Which then could be a more universal measure than execution time to compare algorithms? Some researchers count character comparisons. When the first string matching algorithms were introduced, the number of comparisons was an important measure to reflect the work load of an algorithm. Because many of newer algorithms, like bndm [15], do not use comparisons, researchers started to use the number of read text characters. When the technology advances, even the number of read text characters is no longer a good estimate for speed, as the brute force algorithm with a q-gram guard [16] shows.

One problem of the area of string matching is that the developers are enthusiastic about too small improvements. Differences less than 5% are not significant in practice. Small changes in the code, like reordering of variables and arrays, or switching the computer may contribute a similar difference. We think that 20% is a fair threshold for a significant improvement.

8 Conclusions

Mostly we reviewed good testing practices but there are issues which may lead incorrect conclusions in algorithm comparisons. Experimental algorithm rankings are never absolute because evolving technology affects them. The ranking order of algorithms may even change when the comparison is repeated on another processor of the same age or generation. Therefore, conclusions based on a difference of less than 5% in running times are not acceptable. When selecting data for experiments, the length of text should be at least 1.5 times the cache size in order to avoid cache interference with running times. The most remarkable finding of this paper is how the use of shared memory may disturb running times.

References

1. R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
2. J. Boháč: *Reliable TSC-based timekeeping for platforms with unstable TSC*, Master's thesis, Charles University, Prague, Czech Republic, 2008.
3. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci: *A portable programming interface for performance evaluation on modern processors.* International Journal of High Performance Computing Applications, 14(3) 2000, pp. 189–204.
4. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.

5. S. Faro: *SMART.* `https://github.com/smart-tool/smart`, 2016, Commit cd7464526d41396e11912c6a681eddb965e17f58. Accessed 12.6.2020.
6. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Computing Surveys, 45(2) 2013.
7. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111.
8. P. J. Fleming and J. J. Wallace: *How not to lie with statistics: The correct way to summarize benchmark results.* Commun. ACM, 29(3) 1986, pp. 218–221.
9. T. Hirvola: *Bit-parallel approximate matching of circular strings with k mismatches.* `https://github.com/hirvola/bsa`, 2017, Commit 1f5264c481ea4152c68c47cdfe2c76657448ba7c. Accessed 20.11.2020.
10. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
11. A. Hume and D. Sunday: *Fast string searching.* Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
12. M. A. Khan: *A transformation for optimizing string-matching algorithms for long patterns.* The Computer Journal, 59(12) 2016, pp. 1749–1759.
13. J. Y. Kim and J. Shawe-Taylor: *Fast string matching using an n-gram algorithm.* Software: Practice and Experience, 24(1) 1994, pp. 79–88.
14. D. Levinthal: *Performance analysis guide for Intel® Core i7 Processor and Intel® Xeon 5500 processors.* Intel, 2009.
15. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., Berlin, Heidelberg, 1998, Springer-Verlag, pp. 14–33.
16. W. Pakalén, J. Tarhio, and B. W. Watson: *Searching with extended guard and pivot loop*, in Proceedings of the Prague Stringology Conference 2021, Czech Technical University in Prague, Czech Republic, 2021.
17. D. A. Rusling: *The Linux Kernel*, New Riders Pub, 2000.

Refined Upper Bounds on the Size of the Condensed Neighbourhood of Sequences

Cedric Chauve[1,2], Marni Mishna[1], and France Paquet-Nadeau[1]

[1] Department of Mathematics, Simon Fraser University
8888 University Drive, Burnaby, BC, V5A 1S6, Canada
[2] LaBRI, Université de Bordeaux
351 Cours de la Libération
33405 Talence Cedex, France

Abstract. The d-neighbourhood of a sequence s is the set of sequences that are at distance at most d from s, for a given measure of distance and parameter d. The condensed d-neighbourhood is obtained by removing from the neighbourhood any sequence having a prefix also in the neighbourhood. Estimating the maximum size of the condensed neighbourhood over all DNA sequences of a given length k for the Levenshtein distance is a problem related, among others, to the analysis of the BLAST (Basic Local Alignment Search Tool, Altschul et al., 1990). In this work, we analyse recurrences for computing an upper bound to the maximum size of the condensed d-neighbourhood for sequences of length k and provide a simpler asymptotic expression that we conjecture results in a dramatically improved upper bound.

Keywords: sequence neighbourhood, algorithms analysis, analytic combinatorics

1 Introduction

The search for all of the approximate occurrences of a query sequence within a text, known as approximate pattern matching, is a central problem in biological sequence analysis [5]. Algorithms based on the seed-and-extend approach, such as BLAST [1], proceed in two phases, the first one identifying *seeds* that are exact short patterns present both in the query and the text, that are later *extended* through dynamic programming. BLAST performs the first phase, that detects seeds, in two steps, *neighbourhood generation* and *filtration*. The neighbourhood generation step consists in computing the neighbourhood of all or a set of k-mers present in the pattern – actually the *condensed neighbourhood*, a subset of the neighbourhood – which is then filtered to keep only the the neighbourhood sequences that also appear in the text. It follows that the maximum size of the (condensed) neighbourhood plays an important role in the analysis of such algorithms, a topic that has been studied in several papers [5,6].

In this note we determine an asymptotics expression for the upper bound formula for the size of the (k, d)-condensed neighbourhood for the Levenshtein distance and provide experimental evidence this expression is an actual upper bound. In Section 2 we define formally the concepts of neighbourhood and condensed neighbourhood for the Levenshtein distance, and describe previous results on upper bounds for the maximum size of the neighbourhood. Then in Section 3, we describe our improved upper bound and apply it to the asymptotic analysis of the approximate pattern matching algorithm introduced in [5], that serves as a basis for BLAST.

2 Background

2.1 Neighbourhood and condensed neighbourhood

We remind that the Levenshtein distance between two sequences is the minimum number of edit operations (insertion, deletion, substitution of a single character) needed to transform one sequence into the other. We denote by $d_{Lev}(v,w)$ the Levenshtein distance between two sequences v and w.

Formally, given a sequence w of length k on an alphabet Σ (with $|\Sigma| = s$), the d-neighbourhood of w, denoted by $N(d,w)$, is the set of all sequences on Σ at Levenshtein distance of w at most d:

$$N(d,w) := \{v \mid d_{Lev}(v,w) \leq d\}.$$

The condensed neighbourhood of w, denoted by $CN(d,w)$, is the subset of this neighbourhood comprising sequences that have none of their prefixes in the neighbourhood:

$$CN(d,w) := \{v \mid v \in N(d,w) \text{ and there is no } u \in N(d,w) \text{ that is a prefix of } v\}.$$

The time complexity analysis of approximate pattern matching algorithms requires an estimate of the maximum size of a condensed d-neighbourhood over all sequences w of length k on an alphabet of size s. We denote this maximum size $CN(s,k,d)$:

$$CN(s,k,d) := \max_{w \in \Sigma^k} |CN(d,w)|.$$

In the course of Myer's discussion of the BLAST algorithm in [6], he provides such an upper bound for $CN(s,k,d)$ and asks for a tighter bound. In this work, we do exactly this and provide a better upper bound, by combining the recurrence equations given by Myers in [6] and techniques from basic analytic combinatorics.

2.2 Recurrences and known upper bound

In [6], Myers describes recurrences suitable for both generation and counting of edit scripts of distance at most d, over k symbols taken from an alphabet of size s. Furthermore he showed how these recurrences could be used to bound $CN(s,k,d)$.

Lemma 1 (Myers, [6]). *Let $S(s,k,d)$ be defined by the following trivariate recurrence. If $k \leq d$ or $d = 0$ then*

$$S(s,k,d) := 1,$$

otherwise

$$S(s,k,d) := \begin{cases} S(s,k-1,d) + (s-1)S(s,k-1,d-1) \\ +(s-1)\displaystyle\sum_{j=0}^{d-1} s^j S(s,k-2,d-1-j) \\ +(s-1)^2 \displaystyle\sum_{j=0}^{d-2} s^j S(s,k-2,d-2-j) \\ +\displaystyle\sum_{j=0}^{d-1} S(s,k-2-j,d-1-j) \end{cases}$$

Let

$$T(s,k,d) := S(s,k,d) + \sum_{j=1}^{d} s^j S(s,k-1,d-j).$$

Then

$$CN(s,k,d) \le T(s,k,d).$$

The term $S(s,k,d)$ counts edit scripts between two sequences over an alphabet of size s (a text and a query, assumed to be of length k) and with a Levenshtein score (edit distance implied by the edit script) at most d. From an edit script, one can generate a word of the neighbourhood by discarding gaps from the text. However, not all edit scripts are considered by the recurrences as some sets of edit scripts are *redundant* and generate the same sequence of the neighbourhood. Nevertheless, the recurrences of Lemma 1 generate redundant sequences from the neighbourhood and can then only be used to provide an upper bound to the maximum size of the condensed neighbourhood. For example, considering an alphabet composed of the single symbol $\{a\}$, $k=5$ and $d=2$, the sequence *aaa* belongs to the condensed neighbourhood of *aaaaa* and is generated 10 times by the recurrences of Lemma 1. We refer to [6,7] for a detailed explanation of the recurrences and an analysis of the redundancy.

From these recurrences, Myers managed to prove that there exists a function $B(s,k,d,c)$

$$B(s,k,d,c) := \left(\frac{c+1}{c-1}\right)^k c^d s^d$$

such that for any $c \ge 1$

$$S(s,k,d) \le B(s,k,d,c)$$

$$CN(s,k,d) \le \frac{c}{c-1} B(s,k,d,c).$$

Moreover, it is not difficult to show that $B(s,k,d,c)$ is minimized when

$$c = c^\star := \epsilon^{-1} + \sqrt{1+\epsilon^{-2}}$$

with $\epsilon = d/k$. This leads to

$$\frac{c^\star}{c^\star - 1} = \frac{1+\sqrt{2}}{\sqrt{2}},$$

and we can deduce from there upper bound on both $S(s,k,d)$ and $CN(s,k,d)$, given in Theorem 1 below.

Theorem 1. *Let*

$$M(s,k,d) := \frac{1+\sqrt{2}}{\sqrt{2}} B(s,k,d,c^\star).$$

Then

$$S(s,k,d) \le B(s,k,d,c^\star) \text{ and } CN(s,k,d) \le M(s,k,d).$$

An important remark is that this upper bound involves a term with exponent d and a term with exponent k, while simple experiments suggest that, for given s and d, $CN(s,k,d)$ as a function of k fits to a polynomial of degree d [7].

3 Results

In this section, we use basic singularity analysis techniques from analytic combinatorics to illustrate how to approximate the bounds in Lemma 1, with a quantity we conjecture is an actual upper bound, which we show experimentally for $s = 4$ (the DNA alphabet size), $k \leq 50$ and $d \leq 4$. For more detail on the discussion, please see [7]. Then we apply this approximation to the expected time analysis of the approximate pattern matching algorithm introduced in [5].

3.1 An improved upper bound

Theorem 2. *Let $s, k, d \in \mathbb{N}$. Then asymptotically as $k \to \infty$*

$$T(s,k,d) \simeq \frac{(2s-1)^d k^d}{d!}.$$

To simplify the following discussion, we set

$$A(s,k,d) := \frac{(2s-1)^d k^d}{d!}.$$

We will first show the following limit, valid for positive s, d:

$$\lim_{k\to\infty} \frac{S(s,k,d)}{A(s,k,d)} = 1.$$

This limit can be shown by considering the generating function for the sequence $(S(s,k,d))_{k\geq 0}$. This is the formal power series

$$\mathbf{S}_{s,d}(z) := \sum_{k=1}^{\infty} S(s,k,d) z^k.$$

Analytic combinatorics (as described in [3]) connects the singularities of the series (viewed as function) to the behaviour of its coefficients. In this case, the generating functions $\mathbf{S}_{s,d}(z)$ are Taylor series of rational functions, and hence this is a straightforward.

The recurrences given in Lemma 1 lead immediately to a system of functional equations satisfied by $\mathbf{S}_{s,1}(z), \ldots, \mathbf{S}_{s,d}(z)$, for fixed but arbitrary s and d. The system is easily solvable and determines closed forms for the generating functions as rational functions. We provide an illustration below for $d = 1$, followed by a formal proof.

The recurrences of Lemma 1 translate into $S(s,k,1) = S(s,k-1,1) + (s-1)S(s,k-1,0) + (s-1)S(s,k-2,0) + S(s,k-2,0)$ which simplifies to

$$S(s,k,1) = S(s,k-1,1) + 2s - 1.$$

We convert the coefficient recurrence into a functional equation for $\mathbf{S}_{s,d}(z)$ by multiplying each side by z^k and summing from $k = 1$ to infinity. This gives:

$$\begin{aligned}
\mathbf{S}_{s,1}(z) - 1 = \sum_{k=1}^{\infty} S(s,k,1) z^k &= \sum_{k=1}^{\infty} S(s,k-1,1) z^k + (2s-1)\sum_{k=1}^{\infty} z^k \\
&= z\sum_{k=1}^{\infty} S(s,k-1,1) z^{k-1} + (2s-1)\sum_{k=1}^{\infty} z^k \\
&= z\mathbf{S}_{s,1}(z) + \frac{(2s-1)z}{1-z}.
\end{aligned}$$

From this we compute

$$\mathbf{S}_{s,1}(z) = \frac{(2s-2)z+1}{(1-z)^2}.$$

For any fixed d we can determine a recurrence, and solve it to determine a closed form for the generating function $\mathbf{S}_{s,d}(z)$. Indeed, this can be automated using a system of computer algebra such as Maple [4]. The next values are given in the following table.

d	$\mathbf{S}_{s,d}(z)$
2	$(z^3 - z^2(4s^2 - 4s + 3) + 2z - 1)(z-1)^{-3}$
3	$(z^5(2s^3 - s^2 - 3s + 3) - z^4(4s^3 - 6s^2 + 2s + 3) + z^3(10s^3 - 17s^2 + 11s - 2) + 3z^2 - 3z + 1)(z-1)^{-4}$

Given a rational function, it is straightforward to determine the asymptotic growth of its Taylor series coefficients [3], which, in our case, leads to the following formula for the dominant term of the asymptotic growth of the coefficients $S(s,k,d)$:

$$\lim_{k\to\infty} \frac{S(s,k,d)}{A(s,k,d)} = 1$$

We can extend this approach to the analysis of the size of the condensed neighbourhood, using the relation

$$CN(s,k,d) \leq S(s,k,d) + \sum_{j=1}^{d} s^j S(s,k-1,d-j).$$

given in Lemma 1. Let us denote by $\mathbf{T}_{s,d}(z)$ the ordinary generating function defined by

$$\mathbf{T}_{s,d}(z) := \sum_{k=1}^{\infty} T(s,k,d) z^k.$$

Applying the same technique than previously yields the following

$$\begin{aligned}\mathbf{T}_{s,d}(z) &= \sum_{k=1}^{\infty} S(s,k,d) z^k + \sum_{k=1}^{\infty}\sum_{j=1}^{d} s^j S(s,k-1,d-j) z^k \\ &= \mathbf{S}_{s,d}(z) + z\sum_{k=1}^{\infty}\sum_{j=1}^{d-1} s^j S(s,k-1,d-j) z^k + s^d \sum_{k=1}^{\infty} z^k\end{aligned}$$

which leads immediately to

$$\mathbf{T}_{s,d}(z) = \mathbf{S}_{s,d}(z) + z\left(\sum_{j=1}^{d-1} s^j \left(\mathbf{S}_{s,d-j}(z) - 1\right)\right) + \frac{s^d}{1-z}.$$

Asymptotic analysis of the generating function $\mathbf{T}_{s,d}(z)$ shows that asymptotically, its coefficients are equivalent to the function $A(s,k,d)$ defined above.

We provide at `https://github.com/cchauve/CondensedNeighbourhoods` a Maple session that illustrates this process and shows the bounds claimed in Theorem 2.

We now provide a formal proof. We denote by $[z^k]\mathbf{F}(z)$ the coefficient of z^k in a generating function $F(z)$.

Lemma 2. *Let d be a strictly positive integer. Suppose $P(z)$ is a polynomial such that $P(1) \neq 0$. Then asymptotically, when k becomes large,*

$$[z^k]\frac{P(z)}{(1-z)^{d+1}} \sim \frac{P(1)k^d}{d!}.$$

Proof. This follows from basic coefficient asymptotics of rational functions. The only (and hence dominant) singularity of $\frac{P(z)}{(1-z)^d}$ is at $z = 1$, and it is a pole of order $d+1$. The coefficient asymptotics are a direct consequence of the transfer theorem [2].

Lemma 3.
$$\mathbf{S}_{s,d}(z) = \frac{P_{s,d}(z)}{(1-z)^{d+1}}$$
where $P_{s,d}(z)$ is a polynomial that satisfies $P_{s,d}(1) = (2s-1)^d$.

Proof. We prove the result by induction on d. From the recurrences in Lemma 1, we can write

$$\begin{aligned}\mathbf{S}_{s,d}(z) &= z\mathbf{S}_{s,d}(z) + z(s-1)\mathbf{S}_{s,d-1}(z)\\ &\quad + (s-1)z^2\mathbf{S}_{s,d-1}(z) + z\mathbf{S}_{s,d-1}(z) + \frac{X(z)}{(1-z)^{d-1}}\end{aligned}$$

where $X(z)$ is a linear combination of the $P_{s,d'}$ for $d' < d-1$. We rearrange to obtain

$$\mathbf{S}_{s,d}(z)(1-z) = \mathbf{S}_{s,d-1}(z)\left((s-1)z + (s-1)z^2 + z\right) + X(z).$$

By induction we have

$$\begin{aligned}\mathbf{S}_{s,d}(z) &= \frac{1}{(1-z)}\left(\frac{P_{s,d-1}(z)}{(1-z)^d}\left((s-1)z + (s-1)z^2 + z\right) + \frac{X(z)}{(1-z)^{d-1}}\right)\\ &= \frac{P_{s,d-1}(z)\left((s-1)z + (s-1)z^2 + z\right) + (1-z)X(z)}{(1-z)^{d+1}}\end{aligned}$$

The numerator, when evaluated at $z = 1$ is equal to $P_{s,d-1}(1)(2s-1) = (2s-1)^d$, upon applying the inductive hypothesis. This proves the claimed result.

Proof (Theorem 2). We know that

$$\mathbf{T}_{s,d}(z) = \mathbf{S}_{s,d}(z) + z\left(\sum_{j=1}^{d-1} s^j\left(\mathbf{S}_{s,d-j}(z) - 1\right)\right) + \frac{s^d}{1-z},$$

hence has a pole at $z = 1$. We know from Lemma 3 that it is a pole of order $d+1$, since the dominant singularity for $\mathbf{T}_{s,d}(z)$ is from $\mathbf{S}_{s,d}(z)$. Consequently, as was the case with $\mathbf{S}_{s,d}(z)$, we can show that

$$\lim_{k\to\infty}\frac{[z^k]\mathbf{T}_{s,d}(z)}{A(s,k,d)} = 1.$$

We illustrate in Fig. 1 the behaviour of the three expressions introduced so far to bound up $CN(s,k,d)$, $T(s,k,d)$, $M(s,k,d)$ and $A(s,k,d)$, which shows for $s=4$, $d \leq 4$ and $k \leq 50$ our asymptotics estimate is an actual upper bound that improves dramatically over the previous known upper bound[1].

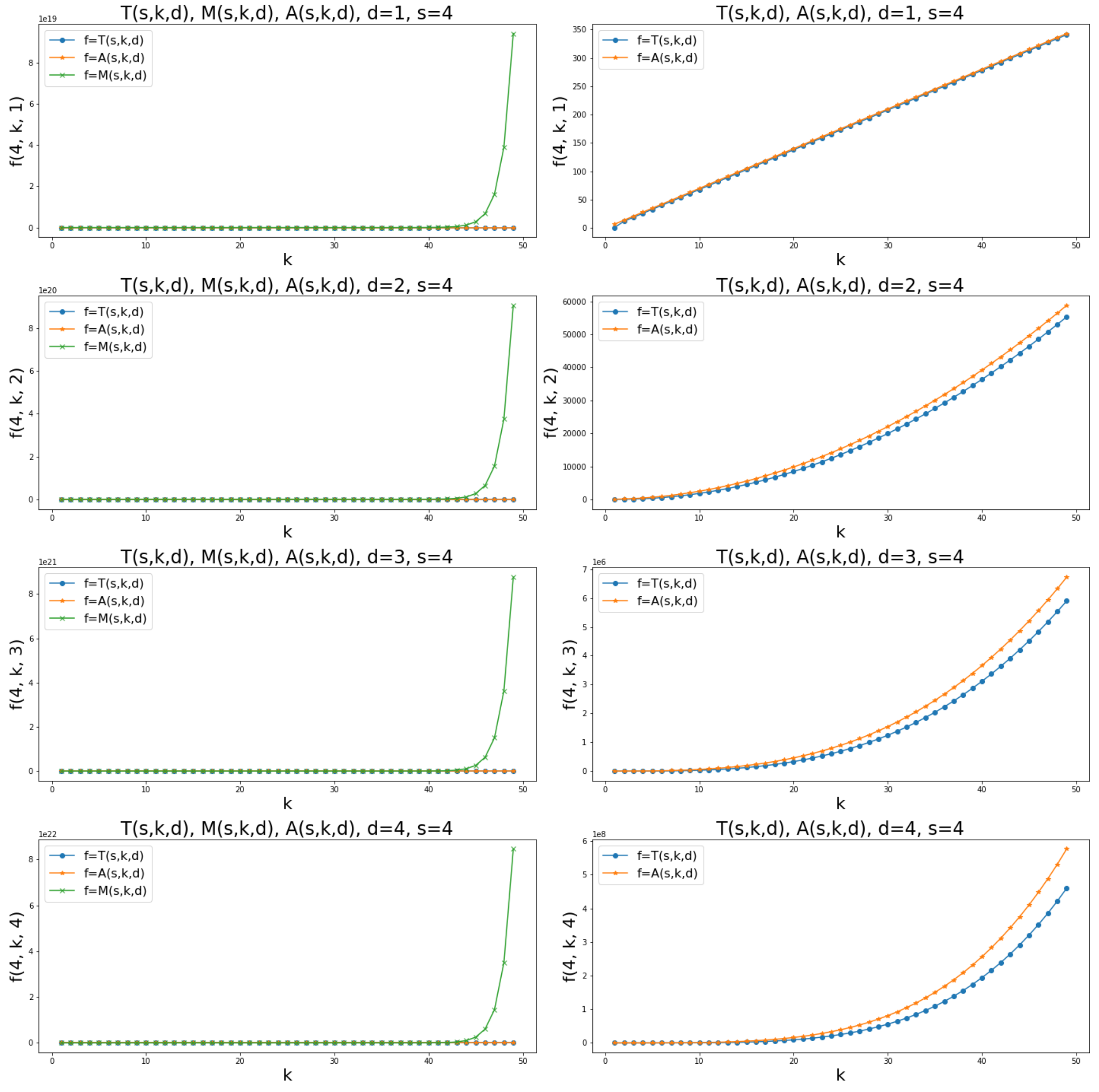


Figure 1. Illustration of the behaviour of $T(s,k,d)$, $M(s,k,d)$ and $A(s,k,d)$ for $s=4$, $d=1,2,3,4$ and $k \geq 30$. (Left): the three functions are shown. (Right): the functions $T(s,k,d)$ and $A(s,k,d)$ are shown.

Our experimental results lead to the proposition and conjecture below.

Proposition 1. *Let* $s \in \{1,\ldots,4\}$, $k \in \{1,\ldots,50\}$, $d \in \{1,\ldots,4\}$. *Then*

$$CN(s,k,d) \leq \frac{(2s-1)^d k^d}{d!}.$$

Conjecture 1 *Let* $s,k,d \in \mathbb{N}$. *Then*

$$CN(s,k,d) \leq \frac{(2s-1)^d k^d}{d!}.$$

[1] The python code used to generate the figures is available in the github repository `https://github.com/cchauve/CondensedNeighbourhoods`.

3.2 Approximate pattern matching expected-time complexity

Bounding the size of condensed neighbourhoods is a key element in the analysis of the time complexity of the approximate pattern matching algorithm described in [5]. The approximate pattern matching problem can be stated as follows: given a (long) text of length n, a (short) pattern of length p, and an integer $e < p$, how can we find in the text all the occurrences of sequences that are at distance at most e from the pattern (e-approximate pattern occurrences).

Myers describes an algorithm that, for a given value k to be discussed later, splits the pattern into p/k non-overlapping substrings of length k (k-mers), then computes for each such k-mer its condensed neighbourhood, searches (through a pre-built index) occurrences of the sequences in these neighbourhoods in the text, and for any such occurrence, tries to extend it into an approximate pattern occurrence. The algorithm is more complex than the high level overview above, but we are interested here in its expected time complexity, under the assumption Conjecture 1 is true.

Expected-time complexity analysis from [6]. Let $\epsilon = e/p$ and so $d = \lceil k\epsilon \rceil$. Assume there exists a function $\alpha(\epsilon)$ such that

$$\frac{1}{\alpha(\epsilon)^k}$$

is an upper bound to the maximum probability, taken over all possible k-mers w, that a random position in a random Bernoulli text is the start of a d-approximate occurrence of w (i.e. belongs to its condensed neighbourhood). Denote this probability by $Pr(k, d)$.

Then, if h is the expected number of e-approximate occurrences of the pattern in the text, the expected-time complexity of the algorithm described in [5] is

$$O\left(e \cdot CN(s, k, d) + n \cdot e \cdot k \cdot Pr(k, d) + h \cdot e \cdot p\right)$$

which gives

$$O\left(e \cdot CN(s, k, d) + \frac{n \cdot e \cdot k}{\alpha(\epsilon)^k} + h \cdot e \cdot p\right)$$

with an optimal value of k being $k = \log_s(n)$.

It can be shown that $Pr(k, d)$ is bounded by $CN(s, k, d)/s^k$. From this, Myers deduced that $Pr(k, d)$ is bounded above by

$$\left(\frac{c^*}{c^* - 1}\right) \frac{B(s, k, d, c^*)}{s^k}$$

to define α by

$$\alpha(\epsilon) := \left(\frac{c^* - 1}{c^* + 1}\right) (c^*)^{-\epsilon} s^{1-\epsilon}.$$

Moreover, if we define

$$pow(\epsilon) := \log_s \left(\frac{c^* + 1}{c^* - 1}\right) + \epsilon \log_s (c^*) + \epsilon,$$

then

$$CN(s, k, d) = O\left(\left(s^{pow(\epsilon)}\right)^k\right) \text{ and } \alpha(\epsilon) = O\left(s^{1-pow(\epsilon)}\right)$$

which implies that the expected time complexity of the algorithm is

$$O\left(e\left(s^k\right)^{pow(\epsilon)}\left(1+k\frac{n}{s^k}\right)+h\cdot e\cdot p\right)$$

which gives, for $k=\log_s(n)$,

$$O\left(e\cdot n^{pow(\epsilon)}\cdot\log_s(n)+h\cdot e\cdot p\right).$$

This is sub-linear in n if $pow(\epsilon)<1$, which Myers showed is true if $\epsilon\leq 1/3$ for $s=4$ (DNA alphabet), i.e. if we are looking for approximate pattern occurrences with roughly a 33% difference rate.

Improved expected-time complexity analysis. The motivation for obtaining a better bound on $CN(s,k,d)$ is to improve the exponential factor (parameterized by ϵ), currently $pow(\epsilon)$. Of particular interest is the question of a bounding function that would intersect the line $y=1$ further than $\epsilon=1/3$ and would thus increase the window of sublinearity of the approximate pattern matching algorithm described in [5].

The key assumption is that, for fixed s and d, $CN(s,k,d)/s^k$ provides an upper bound to $Pr(k,d)$. This leads to the following expected time complexity for the full algorithm, using our improved upper bound $A(s,k,d)$:

$$O\left(e\cdot A(s,k,d)+n\cdot e\cdot k\cdot\frac{A(s,k,d)}{s^k}+h\cdot e\cdot p\right)$$

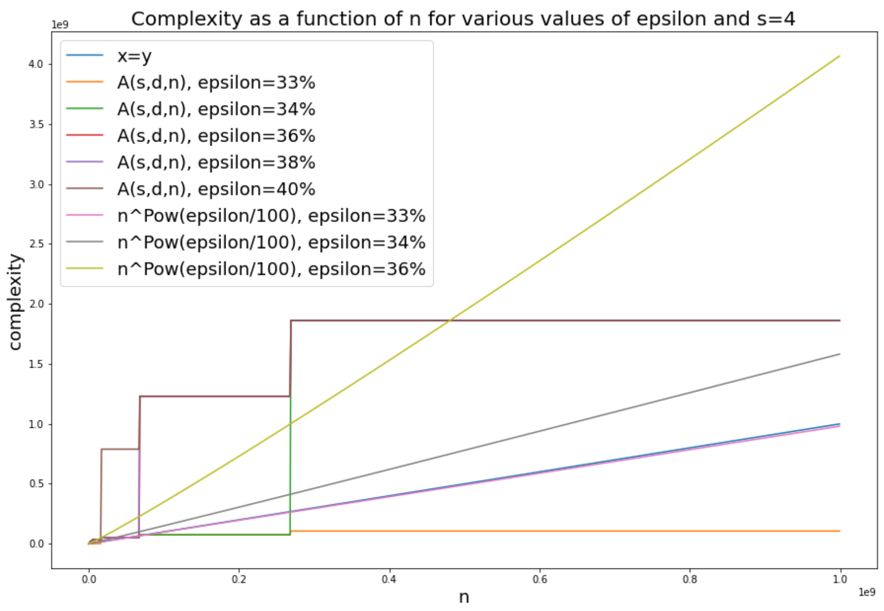
or equivalently

$$O\left(e\cdot A(s,k,d)\left(1+k\frac{n}{s^k}\right)+h\cdot e\cdot p\right)$$

and we are left with the task to see if there exists a function $f(\epsilon)$ such that

- $A(s,k,d)$ can be expressed as or bounded upon by $s^{k\cdot f(\epsilon)}$, or, if we assume $k=\log_s(n)$, $n^{f(\epsilon)}$, and
- it intersects $y=1$ further than $pow(\epsilon)$.

To evaluate this experimentally, we computed the value of $A(s,k,d)$ and $n^{pow(\epsilon)}$ for $s=4$ and $k=\lceil\log_S(n)\rceil$ for values of n going up to 10^9 (roughly the size of a human genome) and various values of ϵ. We also compared $pow(\epsilon)$ with $\log_n(A(s,k,d))$, taken as a function of ϵ, for $n=10^9$. The results of both computations are shown in Fig. 2.



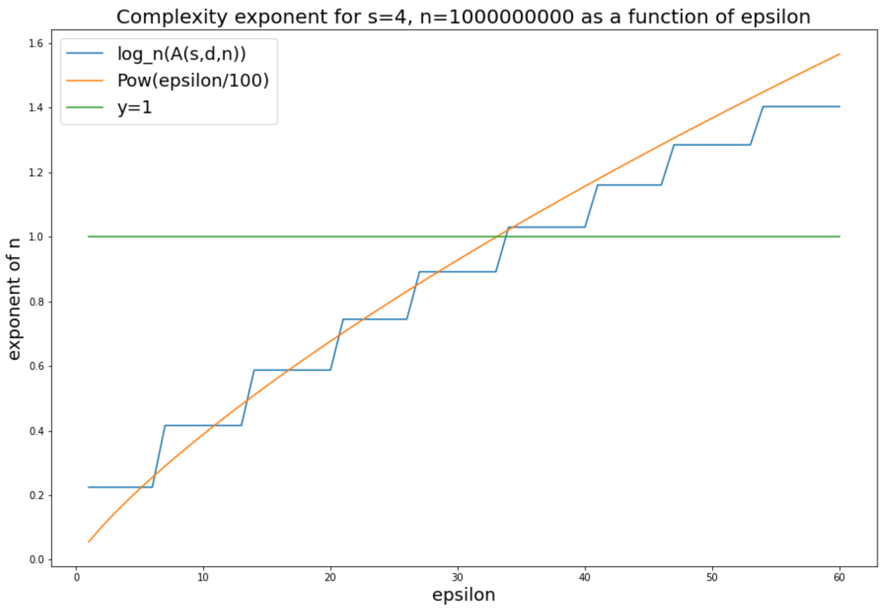


Figure 2. (Top) Illustration of the behaviour of $A(s,k,d)$ and $n^{pow(\epsilon)}$ compared to $f(n) = n$. (Bottom) Illustration of the behaviour of $pow(\epsilon)$ and $\log_n(A(s,k,d))$ for $n = 10^9$ as a function of ϵ.

4 Conclusion

This work contains two main parts. The first one (Theorem 2 and Conjecture 1) suggests a novel upper bound for the size of the condensed neighbourhood. The experimental results illustrated in Fig. 1 suggest strongly that this upper bound is much better than the one provided in [6], although we do not have a formal proof of this claim at the time. The code we provide allows to actually test if our estimate is an actual upper-bound for any given setting defined by s, k, d. Nevertheless, we can observe that our asymtotics expression is very close the actual expression $T(s, k, d)$, although the gap widens as d increases.

The second part addresses the expected-time complexity of the approximate pattern matching algorithm introduced in [5], with the goal to extend the window of sublinearity of the algorithm in terms of the parameter ϵ. We can observe on Fig. 2 that despite being tighter, our upper bound on the size of the condensed neighbourhood does not seem to lead to a much wider window of sublinearity, for a value of n close to the size of a human genome. Indeed, we can see that $\log_n(A(s, k, d))$ intersects with $y = 1$ just right of $pow(\epsilon)$. So we do not seem to obtain a significant improvement, although we obtain a slight one. Note however that the behaviour of the function shown in Fig. 2 based on $A(s, k, d)$ depends significantly on the fact that we impose that k and d are integers and thus take $k = \lceil \log_s(n) \rceil$ and $d = \lceil k\epsilon \rceil$. The graph of the function where any of these expressions is taken as a floating number intersects $y = 1$ much further to 1/3.

Our results might indicate that extending the window of sublinear behaviour of [5] might require to improve the recurrences of Lemma 1 more than to obtain a tighter bound of $S(s, k, d)$. Indeed, the recurrences of Lemma 1 result in edit scripts that lead to some redundant words, and there might thus be room to obtain better recurrences, on which similar analytic combinatorics techniques could then be applied to obtain tight bounds.

References

1. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215 1990, pp. 403–410.
2. P. Flajolet and A. M. Odlyzko: *Singularity analysis of generating functions.* SIAM J. Discret. Math., 3(2) 1990, pp. 216–240.
3. P. Flajolet and R. Sedgewick: *Analytic Combinatorics*, Cambridge University Press, New York, NY, USA, 1 ed., 2009.
4. Maplesoft, a division of Waterloo Maple Inc..: *Maple.*
5. E. W. Myers: *A sublinear algorithm for approximate keyword searching.* Algorithmica, 12(4/5) 1994, pp. 345–374.
6. G. Myers: *What's Behind Blast*, in Models and Algorithms for Genome Evolution, Springer, 2013, pp. 3–15, Video presentation at `https://www.youtube.com/watch?v=pVFX3V0Q2Rg`.
7. F. Paquet-Nadeau: *On the maximum size of condensed sequences neighbourhoods under the Levenshtein distance.* MSc Project Report, Department of Mathematics, Simon Fraser University, Burnaby, BC, Canada, URL: `https://github.com/cchauve/CondensedNeighbourhoods`, 2017.

Computational Substantiation of the *d*-step Conjecture for Distinct Squares Revisited

Frantisek Franek[1] and Michael Liut[2]

[1] Department of Computing and Software
McMaster University, Hamilton, Canada
`franek@mcmaster.ca`

[2] Department of Mathematical and Computational Sciences
University of Toronto Mississauga, Canada
`michael.liut@utoronto.ca`

Abstract. The maximum number of distinct squares problem was introduced in 1998 by Fraenkel and Simspon. They provided a bound of $2n$ for a string of length n and conjectured that the bound should be at most n. Though there have been several improvements since, the conjecture is still unresolved. In 2011, Deza et al. introduced the d-step conjecture for the maximum number of distinct primitively-rooted squares: for a string of length n with d distinct symbols, the number of distinct squares is bounded by the value of $n-d$. In 2016, Deza et al. presented a framework for computer search for strings exhibiting the maximum number of distinct primitively-rooted squares. The framework was based on the d-step method and the main tool, the S-cover, allowed them to approximately double the length of strings that can be analyzed in comparison to the brute force. For instance, they were able to compute the values for binary strings up to length 70. We present a framework for computer search for counterexamples to the d-step conjecture. This change of focus, combined with additional novel analysis, allow us to constrain the search space to a larger degree, thus enabling a verification of the d-step conjecture to higher lengths. For instance, we have fully verified the d-step conjecture for all combinations of n and d such that $n–d \leq 24$ and for binary strings up to length 90. The computational efforts are still continuing. Since neither the maximum number of distinct squares conjecture nor the d-step conjecture can be resolved using a computer, the usefulness of our effort is twofold. Firstly, the primary aspiration is that with the identification of sufficient constraints, the non-existence of counterexamples can be established analytically. Secondly, the verification of the conjectures for higher lengths acts indirectly as evidence of the validity of the conjectures, which indicates that effort should be directed towards proving the conjectures rather than disproving them.

Keywords: string, square, root of square, primitively-rooted square, number of distinct squares, maximum number of distinct squares conjecture, d-step method, d-step conjecture

1 Introduction

Counting distinct squares means counting each type of square only once, not the occurrences – a problem introduced by Fraenkel and Simpson, [9,10]. They provided a bound of $2n$ for strings of length n and conjectured that it should be bounded by n. Their main idea was to count only the rightmost occurrences of squares, or what we call in this paper, the *rightmost squares*. That simplified the combinatorics of many squares starting at the same position to at most two, allowing them to bound the number of rightmost squares by $2n$. Their conjecture, known as the *maximum number of distinct squares conjecture* remains unresolved. In 2007, Ilie presented an asymptotic bound of $2n - \Theta(\log\ n)$, [11]. In 2015, Deza et al. in [8] presented a result on the bound for the maximum number of FS-double squares (i.e., two rightmost squares starting at the same position) $\frac{5}{6}n$, which in turn gives $\frac{11}{6}n \approx 1.83n$ bound for distinct squares. The work is a careful analysis of the combinatorial relationships of FS-double squares based on the pioneering work of Lam, [13]. In 2020, Thierry posted a preprint in arXiv in which he claims a bound of $1.5n$ based on the same techniques used in [8], however, as far as we know, the preprint has not been submitted for peer review, and there are some aspects of the proof that are not clear to us, see [15].

In 2011, Deza et al. introduced the d-step conjecture for the maximum number of distinct primitively-rooted squares: for a string of length n with d distinct symbols, the number of distinct primitively-rooted squares is bounded by the value of $n - d$, see [5] and an overview in [3]. The d-step conjecture also remains unresolved. In 2014, Janoska et al., [12], proposed a slightly stronger conjecture for binary strings – namely that the number of distinct squares in a binary string of length n is bounded by $\frac{2k-1}{2k+2}n$, where k is the number of occurrences of the symbol occurring the least number of times in the string. Since $k \leq \lfloor \frac{n}{2} \rfloor$, it follows that $\frac{2k-1}{2k+2}n \leq n - 2$ when $n \geq 4$, and thus it is a slight strengthening of the d-step conjecture. They show several classes of binary strings for which their conjecture holds true. In 2015, Manea and Seki, [14], introduced the notion of square-density of a string as a ratio of the number of distinct squares and the length of the string. They showed that binary strings exhibit the largest square densities in the form that for any string over a ternary or higher alphabet, there is a binary string with a higher square-density. To that end, they presented a homomorphism that transforms a given string to a binary string with a higher square-density of significantly longer length. Since the lengths of the original string and its homomorphic transformation are quite different, this result cannot be used for direct comparison of strings of the same length over different alphabets that is need to resolve the d-step conjecture. Moreover, the consequence of the work [14] is that if the maximum number of distinct squares conjecture is shown to hold for strings over a particular non-unary fixed alphabet, then it holds true for strings over any other no-unary fixed alphabet. In 2015, Blanchet-Sadri et al. established upper bounds for the number of primitively-rooted squares in partial words with holes, see [1]. Many researchers intuitively believe that a "heavy" concentration of double squares at the beginning of a string prolongs the string and is compensated by a "light" concentration of squares at the tail. In 2017, Blanchet-Sadri et al. investigated the density of distinct primitively-rooted squares in a string concluding a that a string starting with m consecutive FS-double squares must be at least $7m + 3$ long, thus quantifying a part of the intuitive belief, see [2].

In 2016, Deza et al., [7], presented a framework for computer search for strings exhibiting a maximum number of distinct primitively-rooted squares. The main tool, the S-cover, allowed them to approximately double the length of strings that can be analyzed in comparison to the brute force approach. For instance, they were able to compute the values for binary strings up to length 70 and thus verify the d-step conjecture for binary strings up to length 70.

A naive approach to computing the maximum number of distinct squares for strings of length n is to generate all strings and compute the number of distinct squares in each while recording the maximum. The major cost of this approach is generating d^n many strings, however, with a bit of ingenuity you only need to generate a bit less than $d^{(n-d)}$. This is because the first d distinct symbols can be fixed as the number of distinct squares in a string does not change when the symbols in the string are permuted. Thus, it is important to constrain the generation of the string as early as possible, so the tail of the string does not need to be generated. Deza et al. achieved this by only generating the S-covered strings, reducing the number of strings generated to approximately $d^{\frac{n}{2}}$. A major focus of their paper was devoted to showing why the strings that have no S-cover do not need to be considered, and why many of the strings that may have an S-cover do not need to be considered. The emphasis of the computation was to identify the square-maximal strings and thus care was applied in making sure that only strings that cannot achieve a maximum were eliminated. For smaller lengths, all square-maximal strings were computed, while for longer strings in situations when the number of distinct squares was constrained, only a single square-maximal string was produced, see [4]. Since the strings produced cannot be independently verified to be square-maximal, the verification of the result is the computer code – if the code is correct, the strings are truly square-maximal.

We took the effort one step further, we are no longer searching for square-maximal candidates, instead we are searching for strings of length n with d distinct symbols that violate the d-step conjecture, i.e., the strings that have strictly more than $n-d$ distinct squares. This allows us to constrain the search space even more and thus verify the d-step conjecture to higher values. For instance, we have fully verified the d-step conjecture for all combinations of n and d such that $n-d \leq 24$ and for binary strings up to length 90. The computational efforts are still continuing. Since the result of the computation is an empty set of counterexamples, the result cannot be independently verified; as with [7], the verification of the result is the computer program – if the program is correct, the set of counterexamples is truly empty. It is clear that the d-step conjecture cannot be validated by computer search. So, the question "why bother?" comes to mind. Well, the ultimate goal of our approach is to analytically establish that the set of counterexample strings is empty. The constraints for generation may be strengthened to the point that it would be viable to show analytically that such strings do not exist – for example, in Lemma 5 it is shown that a counterexample string cannot be a square. Meanwhile, it strengthens the empirical evidence that the conjecture is plausible and more effort should be directed towards proving it rather than disproving it.

2 Terminology and Notation

An integer range is denoted by $..$, i.e., $a\,..\,b = \{a, a{+}1, a{+}2, ..., b{-}2, b{-}1, b\}$. A bold font is reserved for strings; $\boldsymbol{x}$ identifies a string named $\boldsymbol{x}$, while x can be used for any other mathematical entity such as integer, or length, etc. For a string $\boldsymbol{x}$ of length n we use the array notation indexing from 1: thus $\boldsymbol{x}[1\,..\,n] = \boldsymbol{x}[1]\boldsymbol{x}[2]...\boldsymbol{x}[n{-}1]\boldsymbol{x}[n]$. The term (d, n)*-string* is used for a string of length n with d distinct symbols. For instance, *ababb* is a $(2, 5)$-string. A *square* is a tandem repeat (concatenation) of the same string, e.g., $\boldsymbol{uu}$. A power of a string $\boldsymbol{u}$ means more than one concatenation of $\boldsymbol{u}$, e.g., $\boldsymbol{u}^2$ means $\boldsymbol{uu}$, $\boldsymbol{u}^3$ means $\boldsymbol{uuu}$, etc. If a string is not a power, it is said to be *primitive*. A *root* of a square $\boldsymbol{uu}$ is the string $\boldsymbol{u}$. A *primitively-rooted* square $\boldsymbol{uu}$ means that the root $\boldsymbol{u}$ is primitive. We identify a square in a string $\boldsymbol{x}[1\,..\,n]$ with a pair (a, b) of positions $1 \leq a < b \leq n$, where a is the starting position and b the ending position of the square, i.e., $\boldsymbol{x}[a\,..\,b]$ is the square. A square (a, b) is *rightmost*, if it is the rightmost occurrence of the square in $\boldsymbol{x}$. For a string $\boldsymbol{x}$, $\mathcal{A}_{\boldsymbol{x}}$ is the string's *alphabet*, i.e., the set of all symbols occurring in $\boldsymbol{x}$. If a symbol occurs only at one position in the string, it is referred to as a *singleton*. A string $\boldsymbol{x}$ is *singleton-free* if every symbol of its alphabet occurs in $\boldsymbol{x}$ at least twice. Symbol $\overleftarrow{\boldsymbol{x}}$ indicates the reversed string $\boldsymbol{x}$, i.e., $\overleftarrow{\boldsymbol{x}}[1\,..\,n] = \boldsymbol{x}[n]\boldsymbol{x}[n{-}1]...\boldsymbol{x}[2]\boldsymbol{x}[1]$. $|\boldsymbol{x}|$ indicate the *size* (*length*) of the string $\boldsymbol{x}$. If $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{u}$ is a *prefix*, $\boldsymbol{w}$ is a *suffix*, and $\boldsymbol{v}$ is a *substring* or *factor* of $\boldsymbol{x}$. If $|\boldsymbol{u}| < |\boldsymbol{x}|$ we speak of a *proper prefix*, if $|\boldsymbol{w}| < |\boldsymbol{x}|$ we speak of a *proper suffix*, and if $|\boldsymbol{v}| < |\boldsymbol{v}|$ we speak of a *proper substring* or *proper factor*. The symbol $s(\boldsymbol{x})$ is used for the number of rightmost squares of the string $\boldsymbol{x}$. The symbol $\sigma_d(n)$ is used for the maximum number of rightmost primitively-rooted squares over all strings of length n with d distinct symbols, i.e., $\sigma_d(n) = \max\{s(\boldsymbol{x}) \mid \boldsymbol{x}\ \textit{is a}\ (d,n)\textit{-string}\}$. For a string $\boldsymbol{x}$ of length n, $B_{\boldsymbol{x}}(i,j)$, $1 \le i \le j \le n$, is defined as the number of rightmost primitively-rooted squares that start in the interval $i\,..\,j$, while $E_{\boldsymbol{x}}(i,j)$, $1 \le i \le j \le n$, is defined as the number of rightmost primitively-rooted squares that end in the interval $i\,..\,j$. Note that $B_{\boldsymbol{x}}(1,i) - E_{\boldsymbol{x}}(1,k)$ is the number of rightmost primitively-rooted squares that start in $1\,..\,i$ and end in $k{+}1\,..\,n$. If it is clear from the context, we drop the subscript $\boldsymbol{x}$ for $B_{\boldsymbol{x}}$ and $E_{\boldsymbol{x}}$.

3 Basic Facts

The following lemma indicates that it makes sense to try using induction over $n{-}d$ rather than over n. The columns in the $(n, d{-}n)$-table, see [4], are completely filled in up to $n{-}d = 19$. In other words, we have a base case for induction over $n{-}d$.

Lemma 1. *Let $\boldsymbol{x}$ be a singleton-free (d,n)-string, $1 \le d < n$ and let d_1 be the number of distinct symbols in a non-empty proper prefix (resp. suffix) of $\boldsymbol{x}$ of length n_1. Then, $n_1{-}d_1 < n{-}d$.*

Proof. Consider a non-empty proper prefix $\boldsymbol{x}[1\,..\,n_1]$ for some $1 \le n_1 < n$. First we consider the extreme case $n_1 = 1$: then $d_1 = 1$ and so $n_1{-}d_1 = 1{-}1 = 0 < n{-}d$. Thus, we can assume that $1 < n_1 < n$.
Let $n_2 = n{-}(n_1{+}1) + 1 = n{-}n_1$, i.e., the length of $\boldsymbol{x}[n_1{+}1\,..\,n]$, and let d_2 be the number of distinct symbols in $\boldsymbol{x}[n_1{+}1\,..\,n]$. If $d_1 = d$, then $n_1{-}d_1 = n_1{-}d < n - d$. So we can assume that $1 \le d_1 < d$.
Let $r = d{-}d_1$. Then $r > 0$ and $1 \le d_1, d_2 \le d$ and $d_1{+}d_2 \ge d$. Since r is the number of distinct symbols from $\boldsymbol{x}$ that are missing in $\boldsymbol{x}[1\,..\,n_1]$, they must occur in $\boldsymbol{x}[n_1{+}1\,..\,n]$ and hence $d_2 \ge r$.
First let us show that $n_2 \ge r$: if $n_2 < r$, then $d_2 < r$ as $d_2 \le n_2$. Hence $d_1{+}d_2 < d_1{+}r = d$, a contradiction.
Second let us show that in fact $n_2 > r$: if $n_2 = r$, then $d_2 \le n_2 = r$ and also $d_2 \ge r$, so $d_2 = r = n_2$. Thus, the number of distinct symbols in $\boldsymbol{x}[n_1{+}1\,..\,n]$ equals the length, i.e., they are all singletons in $\boldsymbol{x}[n_1{+}1\,..\,n]$, and since $\boldsymbol{x}[n_1{+}1\,..\,n]$ must contain the r symbols missing in $\boldsymbol{x}[1\,..\,n_1]$, they must be singletons in $\boldsymbol{x}$ as well, a contradiction.
Thus, $n_2 > r$ and so $n_1 = n{-}n_2 < n{-}r$, giving $n_1{-}d_1 < n{-}r{-}d_1 = n{-}d$.
For a non-empty proper suffix $\boldsymbol{x}[j\,..\,n]$, $1 < j \le n$, let $n_1 = n{-}j{+}1$, i.e., the length of the suffix. Let d_1 be the number of distinct symbols in $\boldsymbol{x}[j\,..\,n]$. Consider the string $\overleftarrow{\boldsymbol{x}}$ which is the string $\boldsymbol{x}$ reversed. Then $\overleftarrow{\boldsymbol{x}}[1\,..\,n_1]$ is a non-empty proper prefix of $\overleftarrow{\boldsymbol{x}}$ of length n_1 with d_1 number of distinct symbols. Therefore, $n_1{-}d_1 < n{-}d$.

□

The next lemma shows, that if the starts of the rightmost squares are not tightly packed, the string cannot be a first counterexample to the d-step conjecture.

Lemma 2. *Assume that $\sigma_{d'}(n') \le n'-d'$ for any $n'-d' < n-d$. Let $\boldsymbol{x}$ be a singleton-free (d,n)-string, $2 \le d < n$. Let $1 \le k < n$ and let d_2 be the number of distinct symbols of $\boldsymbol{x}[k+1 \mathinner{..} n]$ and let e be the number of distinct symbols occurring in both $\boldsymbol{x}[1 \mathinner{..} k]$ and $\boldsymbol{x}[k+1 \mathinner{..} n]$.*
(i) If $B_{\boldsymbol{x}}(1,k) \le k-d+d_2$, then $s(\boldsymbol{x}) \le n-d$.
(ii) If $B_{\boldsymbol{x}}(1,k)-E_{\boldsymbol{x}}(1,k) \le e$, then $s(\boldsymbol{x}) \le n-d$.

Proof. Let $\boldsymbol{x}_1 = \boldsymbol{x}[1 \mathinner{..} k]$, $n_1 = k$, and d_1 be the number of distinct symbols of $\boldsymbol{x}_1$. Thus, $\boldsymbol{x}_1$ is a (d_1,n_1)-string. Let $\boldsymbol{x}_2 = \boldsymbol{x}[k+1 \mathinner{..} n]$, $n_2 = n-k$. Thus, $\boldsymbol{x}_2$ is a (d_2,n_2)-string. Let e_1 be the number of distinct symbols of $\boldsymbol{x}$ that occur only in $\boldsymbol{x}_1$, and let e_2 be the number of distinct symbols of $\boldsymbol{x}$ that occur only $\boldsymbol{x}_2$. Then $d_1 = e_1+e$, $d_2 = e_2+e$, and $d = e_1+e_2+e$, $d_1+d_2 = e_1+e_2+2e = d+e$.

(i) Assume $B_{\boldsymbol{x}}(1,k) \le k-d+d_2$.
- Case $d_2 = 1$.
  If $k = n-1$, then $s(\boldsymbol{x}) = B_{\boldsymbol{x}}(1,k) \le k-d+d_2 = k-d+1 = (n-1)-d+1 = n-d$. If $k \le n-2$, then $s(\boldsymbol{x}_2) = 1$ as $d_2 = 1$, and so $s(\boldsymbol{x}) = B_{\boldsymbol{x}}(1,k)+s(\boldsymbol{x}_2) = B_{\boldsymbol{x}}(1,k)+1 \le (k-d+1)+1 \le n-2-d+2 = (k-d)+2 \le (n-2-d)+2 = n-d$.
- Case $d_2 \ge 2$.
  Then $n_2 \ge d_2$ and so $k = n_1 \le n-d_2$. By Lemma 1, $n_2-d_2 < n-d$ and so by the assumption of this lemma, $s(\boldsymbol{x}_2) \le n_2-d_2$. Thus, $s(\boldsymbol{x}) = B_{\boldsymbol{x}}(1,k)+s(\boldsymbol{x}_2) \le (k-d+d_2)+(n_2-d_2) = k-d+n_2 \le k-d+(n-k) = n-d$.

(ii) Assume that $B_{\boldsymbol{x}}(1,k)-E_{\boldsymbol{x}}(1,k) \le e$.
Since $\boldsymbol{x}_1$ is a proper prefix of $\boldsymbol{x}$, and since $\boldsymbol{x}_2$ is a proper suffix of $\boldsymbol{x}$, by Lemma 1, $n_1-d_1, n+2-d_2 < n-d$, and by the assumption of this lemma, $s(\boldsymbol{x}_1) \le n_1-d_1$ and $s(\boldsymbol{x}_2) \le n_2-d_2$. Thus, $s(\boldsymbol{x}) = \big(B_{\boldsymbol{x}}(1,k)-E_{\boldsymbol{x}}(1,k)\big)+s(\boldsymbol{x}_1)+s(\boldsymbol{x}_2) \le \big(B_{\boldsymbol{x}}(1,k)-E_{\boldsymbol{x}}(1,k)\big)+(n_1-d_1)+(n_2-d_2) = \big(B_{\boldsymbol{x}}(1,k)-E_{\boldsymbol{x}}(1,k)\big)+n-d-e \le e+n-d-e = n-d$.

$\square$

Corollary 3. *Assume that $\sigma_{d'}(n') \le n'-d'$ for any $n'-d' < n-2$. Let $\boldsymbol{x}$ be a singleton-free binary string of length n, $n \ge 3$. Let $1 \le k < n-2$.*
(i) If $B_{\boldsymbol{x}}(1,k) \le k$, then $s(\boldsymbol{x}) \le n-2$.
(ii) If $B_{\boldsymbol{x}}(1,k)-E_{\boldsymbol{x}}(1,k) \le 2$, then $s(\boldsymbol{x}) \le n-2$.

Proof. Let $\boldsymbol{x}_1 = \boldsymbol{x}[1 \mathinner{..} k]$, d_1 be the number of distinct symbols of $\boldsymbol{x}_1$, and $n_1 = k$ be its length. Let $\boldsymbol{x}_2 = \boldsymbol{x}[k+1 \mathinner{..} n]$, d_2 be the number of distinct symbols of $\boldsymbol{x}_2$, and $n_2 = n-k$ be its length. Let e be the number of distinct symbols common to both $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$.
For (i):

- If $d_2 = 2$, it follows directly from Lemma 2.
- If $d_2 = 1$, then $s(\boldsymbol{x}_2) \le 1$. So $s(\boldsymbol{x}) = B_{\boldsymbol{x}}(1,k)+s(\boldsymbol{x}_2) \le k+1 \le n-3+1 = n-2$.

For (ii):

- If $d_1 = d_2 = 2$, it follows directly from Lemma 2 as $e = 2$.
- If $d_1 = 1$ and $d_2 = 2$, then $s(\boldsymbol{x}_1) \le 1$, and so $s(\boldsymbol{x}) \le 1+s(\boldsymbol{x}_2) \le 1+n_2-2 = n_2-1 \le (n-1)-1 = n-2$.
- If $d_1 = 2$ and $d_2 = 1$, we proceed as in the previous case.

- If $d_1 = d_2 = 1$, it follows that $e = 0$ and $B_{\boldsymbol{x}}(1,k) - E_{\boldsymbol{x}}(1,k) = 0$. Then $s(\boldsymbol{x}) \le B_{\boldsymbol{x}}(1,k) + s(\boldsymbol{x}_1) + s(\boldsymbol{x}_2)$.
  If $n = 3$, if $k = 1$ then $s(\boldsymbol{x}_1) = 0$ and $s(\boldsymbol{x}_2) \le 1$, and if $k = 2$, then $s(\boldsymbol{x}_1) \le 1$ and $s(\boldsymbol{x}_2) = 0$, so $s(\boldsymbol{x}_1) + s(\boldsymbol{x}_2) \le 1$. Thus, $s(\boldsymbol{x}) \le 1 = 3-2 = n-2$.
  If $n \ge 4$, then $s(\boldsymbol{x}) = s(\boldsymbol{x}_1) + s(\boldsymbol{x}_2) \le 1+1 = 2 \le n-2$.

□

In the following, by the *first* counterexample we mean a (d,n)-string from the first column of the $(d, n–d)$ table (see [3,5,6,7]) that does not satisfy the d-step conjecture. Note that the assumption of Corollary 4 is that up to $n-d$, all columns of the table satisfy the d-step conjecture, so the column $n–d$ may harbor the first counterexample. When a square is both the rightmost and the leftmost occurrence in a string, we refer to such a square as *unique*.

Thus, the next lemma shows that a first counterexample to the d-step conjecture $\boldsymbol{x}[1\,..\,n]$ must start with two rightmost (and hence unique) squares, and a rightmost (and hence unique) square at the second position. Since $s(\overleftarrow{\boldsymbol{x}}) = s(\boldsymbol{x})$, $\overleftarrow{\boldsymbol{x}}$ must start with two unique squares, and a unique square at the second position, i.e., $\boldsymbol{x}$ must have two unique squares ending in $\boldsymbol{x}[n]$ and a unique square ending in $n-1$.

Corollary 4. *Assume that $\sigma_{d'}(n') \le n'–d'$ for any $n'–d' < n–d$. Let $\boldsymbol{x}$ be a singleton-free (d,n)-string, $2 \le d < n$.*
- *If $B_{\boldsymbol{x}}(1,1) \le 1$, then $s(\boldsymbol{x}) \le n-d$.*
- *If $B_{\boldsymbol{x}}(1,2) \le 2$, then $s(\boldsymbol{x}) \le n-d$.*

Proof. Since $\boldsymbol{x}$ is singleton-free, $\boldsymbol{x}[2\,..\,n]$ has d distinct symbols. If $B_{\boldsymbol{x}}(1,1) \le 1$, then $s(\boldsymbol{x}) \le 1 + s(\boldsymbol{x}[2\,..\,n]) \le 1 + (n-1) - d = n-d$. Thus, we can assume $B_{\boldsymbol{x}}(1,1) = 2$. It follows that $\boldsymbol{x}[3\,..\,n]$ has d distinct symbols. If $B_{\boldsymbol{x}}(1,2) \le 2$, then $s(\boldsymbol{x}) = B_{\boldsymbol{x}}(1,2) + s(\boldsymbol{x}[3\,..\,n]) \le 2 + (n-2) - d = n-d$.

□

Lemma 5 shows that a first counterexample to the d-step conjecture cannot be a square.

Lemma 5. *Assume that $\sigma_{d'}(n') \le n' - d'$ for any $n' - d' < n-d$. Let $2 \le d < n$. For any (d,n)-string $\boldsymbol{x}$ that is square, $s(\boldsymbol{x}) \le n-d$.*

Proof. The proof relies on the combinatorics of FS-double squares analyzed in [8]; the notion of FS-double squares and inversion factors are presented and discussed there, as well as the notions of α-mate, β-mate, γ-mate, δ-mate, and ϵ-mate.

For a deeper understanding of the proof, the reader must consider reading and understanding the work in [8]. Just to make the proof a little bit more self-contained, here are a few relevant facts from [8]:

- At any position, at most two rightmost squares can start.
- Two rightmost squares starting at the same positions form a so-called FS-double square; every FS-double square has a form $\boldsymbol{u}_1^p \boldsymbol{u}_2 \boldsymbol{u}_1^{p+q} \boldsymbol{u}_2 \boldsymbol{u}_1^q$ for some primitive $\boldsymbol{u}_1$, a non-empty proper prefix $\boldsymbol{u}_2$ of $\boldsymbol{u}_1$, and some integers $p \ge q \ge 1$; the longer square is $[\boldsymbol{u}_1^p \boldsymbol{u}_2 \boldsymbol{u}_1^q][\boldsymbol{u}_1^p \boldsymbol{u}_2 \boldsymbol{u}_1^q]$ and the shorter square is $[\boldsymbol{u}_1^p \boldsymbol{u}_2][\boldsymbol{u}_1^p \boldsymbol{u}_2]$.
- If $\boldsymbol{u}_1 = \boldsymbol{u}_2 \bar{\boldsymbol{u}}_2$, then the so-called inversion factor $\bar{\boldsymbol{u}}_2 \boldsymbol{u}_2 \boldsymbol{u}_2 \bar{\boldsymbol{u}}_2$ only occurs twice in the FS-double square $\boldsymbol{u}_1^p \boldsymbol{u}_2 \boldsymbol{u}_1^{p+q} \boldsymbol{u}_2 \boldsymbol{u}_1^q = (\boldsymbol{u}_2 \bar{\boldsymbol{u}}_2)^p \boldsymbol{u}_2 (\boldsymbol{u}_2 \bar{\boldsymbol{u}}_2)^{p+q} \boldsymbol{u}_2 (\boldsymbol{u}_2 \bar{\boldsymbol{u}}_2)^q$, which highly constrain occurrences of squares starting after the FS-double square.

- The maximum left cyclic shift of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ is determined by $lcs(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$, where *lcs* stands for *longest common suffix*, while the maximum right cyclic shift is determined by $\bar{\boldsymbol{u}}_2\boldsymbol{u}_2\boldsymbol{u}_2\bar{\boldsymbol{u}}_2$ is controlled by the value of $lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$, where *lcp* stands for *longest common prefix.*
- $0 \le lcs(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)+lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2) \le |\boldsymbol{u}_1|-2$, see Lemma 11 of [8].
- The starting point of the first occurrence of the inversion factor $\bar{\boldsymbol{u}}_2\boldsymbol{u}_2\boldsymbol{u}_2\bar{\boldsymbol{u}}_2$ is the position L_1, while the starting point of its maximum right cyclic shift is the position R_1, hence $R_1 \le L_1+|\boldsymbol{u}_1|-1$.
- An FS-double square $\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^{r+t}\boldsymbol{v}_2\boldsymbol{v}_1^t$ starting at a position i is an α-mate of an FS-double-square $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ starting at a position j if $j < i \le R_1$ of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ and it is a right cyclic shift of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$.
- An FS-double square $\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^{r+t}\boldsymbol{v}_2\boldsymbol{v}_1^t$ starting at a position i is a β-mate of an FS-double-square
  $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ starting at a position j if it starts at a position $j < i \le R_1$ of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ and it is a right cyclic shift of $\boldsymbol{u}_1^{p-k}\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^{q+k}$ for some k such that $p-k \ge q+k \ge 1$.
- An FS-double square $\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^{r+t}\boldsymbol{v}_2\boldsymbol{v}_1^t$ starting at a position i is a γ-mate of an FS-double-square
  $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ starting at a position j if $j < i \le R_1$ of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ and $\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^r\boldsymbol{v}_2$ is a right cyclic shift of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$.
- An FS-double square $\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^{r+t}\boldsymbol{v}_2\boldsymbol{v}_1^t$ starting at a position i is a δ-mate of an FS-double-square
  $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ starting at a position j if $j < i \le R_1$ of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ and $|\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^r\boldsymbol{v}_2| > |\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q|$.
- An FS-double square $\boldsymbol{v}_1^r\boldsymbol{v}_2\boldsymbol{v}_1^{r+t}\boldsymbol{v}_2\boldsymbol{v}_1^t$ starting at a position i is an ϵ-mate of an FS-double-square
  $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ starting at a position j if $i > R_1$.
- For any two FS-double squares, the one starting later is either an α-mate, or β-mate, or γ-mate, or δ-mate, or ϵ-mate. There are no other possibilities.

Because $\boldsymbol{x}$ is a square, it is singleton-free. By Corollary 4, it must start with two unique squares, otherwise $s(x) \le n-d$ and we are done. Hence $\boldsymbol{x}$ is an FS-double square $\boldsymbol{x} = \boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^{p+q}\boldsymbol{u}_2\boldsymbol{u}_1^q$ for some primitive $\boldsymbol{u}_1$, a non-empty proper prefix $\boldsymbol{u}_2$ of $\boldsymbol{u}_1$, and some $1 \le q \le p$. Consider an FS-double square starting at a position i, $1 < i < R_1$. By Lemma 19 of [8], it must be one of the 5 cases – either it is an α-mate of the starting double square, or a β-mate, or a γ-mate, or a δ-mate, or an ϵ-mate. It cannot be an α-mate as its longer square would have the same size as $\boldsymbol{x}$ and hence would not fit in $\boldsymbol{x}$, nor could it be a β-mate as again its longer square would not fit in $\boldsymbol{x}$, nor could it be a γ-mate as its shorter square would have the same size as $\boldsymbol{x}$ and hence would not fit, nor could it be a δ-mate as again its longer square would not fit in $\boldsymbol{x}$, nor could it be an ϵ-mate as it would have to start at the position R_1 or later. So, we must conclude that no FS-double square can start in the positions $2 \mathinner{..} R_1-1$, so at the most a single rightmost square can start at any of the positions $2 \mathinner{..} R_1-1$.

Consider a rightmost square $\boldsymbol{vv}$ starting at a some position of $2 \mathinner{..} R_1-1$. Since $|\boldsymbol{v}| < |\boldsymbol{x}|$, by Lemma 17 of [8], v must be a right cyclic shift of $\boldsymbol{u}_1^j\boldsymbol{u}_2$ for some $q < j \le p$. Since maximal right cyclic shift is at most $|\boldsymbol{u}_1|-2$, there is no square starting at the position $H = 2+lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2) < |\boldsymbol{u}_1|+1$: first there are the right cyclic shifts of $\boldsymbol{u}_1^p\boldsymbol{u}_2\boldsymbol{u}_1^p\boldsymbol{u}_2$, and there are at most $|\boldsymbol{u}_1|-2$ of them, and so the last one starts at the position $1{+}lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$. If $p = q$, it is all, and thus the position $H = 2{+}lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2) < |\boldsymbol{u}_1|{+}1$ has no square starting there. If $p > q$, then next square is the maximal left cyclic shift of $\boldsymbol{u}_1^{p-1}\boldsymbol{u}_2\boldsymbol{u}_2\boldsymbol{u}_1^{p-1}$ at the position $|\boldsymbol{u}_1|{+}1$, hence it starts at the position $|\boldsymbol{u}_1|{+}1{-}lcs(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$. Since $lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2){+}lcs(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2) \le |\boldsymbol{u}_1|{-}2$, so again the position $H = 2{+}lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$ has no square starting there.

So there is a double square at position 1, at most single squares at positions $2\,..\,H{-}1$, and no square at position H, so $B(1, H) \le H$. Since $\boldsymbol{x}[H{+}1\,..\,n]$ contains $\boldsymbol{u}$, then d_2, the number of distinct symbols of $\boldsymbol{x}[H{+}1\,..\,n]$, equals d. By Lemma 2, $s(\boldsymbol{x}) \le n{-}d$.

□

As indicated in Section 2, we denote a square in $\boldsymbol{x}$ that starts at the position a and ends at the position b as a pair (a, b). The notion of S-cover was introduced in [7].

Definition 6. *Consider a string* $\boldsymbol{x}[1\,..\,n]$*. The sequence of squares* $S = \{(a_i, b_i) \ :\ 1 \le i \le k\}$, $1 \le k$, *is a* partial S-cover *of* $\boldsymbol{x}$ *if*

(i) each (a_i, b_i) *is a rightmost primitively rooted square of* $\boldsymbol{x}$,
(ii) for every $i < k$, $a_i < a_{i+1} < b_i < b_{i+1}$,
(iii) for every rightmost primitively-rooted square (a, b) *of* $\boldsymbol{x}$ *where* $a \le a_k$, *there is* i, $1 \le i \le k$ *so that* $a_i \le a$ *and* $b \le b_i$.

S *is an* S-cover *of* $\boldsymbol{x}$ *if it is a partial* S*-cover of* $\boldsymbol{x}$ *and* $b_k = n$.

Note 7. If a string has an S-cover, it is necessarily singleton-free.

Lemma 8 ([7]). *Consider a* (d, n)*-string* $\boldsymbol{x}$, $2 \le d < n$. *Then either* $s(\boldsymbol{x}) \le n{-}d$ *or* $\boldsymbol{x}$ *has an* S*-cover.*

Lemma 9 ([7]). *Consider a* (d, n)*-string* $\boldsymbol{x}$, $2 \le d < n$. *If* $\boldsymbol{x}$ *admits an* S*-cover, the* S*-cover is unique.*

The following corollary gives a strong restriction for the S-cover of a first counterexample.

Corollary 10. *Assume that* $\sigma_{d'}(n') \le n' - d'$ *for any* $n'{-}d' < n{-}d$. *Let* $2 \le d < n$. *Consider a* (d, n)*-string* $\boldsymbol{x}$, $2 \le d < n$ *with an* S*-cover* $S = \{(a_i, b_i) \ :\ 1 \le i \le k\}$.

(i) If S *has size 1, then* $s(\boldsymbol{x}) \le n{-}d$.
(ii) Either $s(\boldsymbol{x}) \le n - d$, *or* $(a_1, b_1) = (1, b_1)$ *is an FS-double square* $(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^p\boldsymbol{u}_2(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^{p+q}\boldsymbol{u}_2(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^q$ *for some suitable* $\boldsymbol{u}_2, \bar{\boldsymbol{u}}_2, p$, *and* q, *and*
   (a) $\boldsymbol{x}[1] = \boldsymbol{x}[b_1{+}1]$ *and* $a_2 = 2$ *and* $b_1{+}1 \le b_2 \le n$, *or*
   (b) $\boldsymbol{x}[1] \ne \boldsymbol{x}[b_1{+}1]$ *and* $a_2 = 2$ *and* $b_1{+}1 < b_2 \le n$, *or*
   (c) $\boldsymbol{x}[1] \ne \boldsymbol{x}[b_1{+}1]$ *and* $2 < a_2 \le 2{+}lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$, *and there is a rightmost square of* $\boldsymbol{x}[1\,..\,b_1]$ *at the position 2.*

Proof. For (i): If $\boldsymbol{x}$ has an S-cover of size 1, then $\boldsymbol{x}$ is a square. By Lemma 5, $s(\boldsymbol{x}) \le n{-}d$.
For (ii): If $B_{\boldsymbol{x}}(1, 1) < 2$, then by Corollary 4, $s(\boldsymbol{x}) \le n{-}d$. Hence (a_1, b_1) must be an FS-double square $(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^p\boldsymbol{u}_2(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^{p+q}\boldsymbol{u}_2(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^q$ for some suitable $\boldsymbol{u}_2, \bar{\boldsymbol{u}}_2, p$, and q.

- If $\boldsymbol{x}[1] = \boldsymbol{x}[b_1{+}1]$ then (a_1, b_1) can be cyclically shifted to the right, and so a_2 is forced to be equal to 2. Either $(2, b_1 + 1)$ is the longest rightmost square of $\boldsymbol{x}$ starting at the position 2, and then $(a_2, b_2) = (2, b_1{+}1)$, or there is a longer rightmost square starting at the position 2. Since (a_2, b_2) is the longest rightmost square starting at the position 2, $b_2 > b_1{+}1$.

- If $\boldsymbol{x}[1] \neq \boldsymbol{x}[b_1+1]$ and there is no rightmost square of $\boldsymbol{x}[1\,..\,b_1]$ starting at the position 2, then by Corollary 4, there must be a rightmost square of $\boldsymbol{x}$ starting at the position 2 otherwise the number of rightmost squares of $\boldsymbol{x}$ would be bounded by $n{-}d$. Hence the rightmost square at the position 2 must be strictly longer than b_1, and so $b_1+1 < b_2$.
- Let $H = 2{+}lcp(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2, \bar{\boldsymbol{u}}_2\boldsymbol{u}_2)$. If $H < a_2$, then a rightmost square starting at a position $2\,..\,H$ must be completely covered by (a_1, b_1) due to property (iii) of S-cover, and so by Lemma 17 of [8], must be maximal left cyclic shift of $\boldsymbol{u}^j\boldsymbol{v}\boldsymbol{u}^j\boldsymbol{v}$ for some $q \leq j \leq p$. Similarly as in the proof of Lemma 5, we can show that there are at most one rightmost square starting at a position from $2\,..\,H$ (as $H < a_2$), and that there is not a square starting at the position H, so by Corollary 4, $s(\boldsymbol{x}) \leq n{-}d$. By Lemma 17 of [8], the size of (a_2, b_2) must be $\geq$ the size of (a_1, b_1), and so $b_1 \leq b_2{-}a_2{+}1$ giving $b_2 \geq b_1{+}a_2{-}1$.

□

4 The Computer Search Framework for a Counterexample to the *d*-step Conjecture

Due to the complexity of a detailed framework, we present a high-level logic outline of the algorithm with justification for each step. Given n and d, $2 \leq d < n$ and knowing that whenever $n'{-}d' < n{-}d$, then $\sigma_{d'}(n') < n'{-}d'$, we search for the first counterexample. This is done by recursively generating a suitable S-cover, rather than the string.

1. The recursion starts with generating all possible (a_1, b_1) in a loop. From Corollary 10, we must generate $(a_1, b_1) = (1, b_1)$ as a primitively-rooted FS-double square $(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^p\boldsymbol{u}_2(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^{p+q}\boldsymbol{u}_2(\boldsymbol{u}_2\bar{\boldsymbol{u}}_2)^q$ where $|\boldsymbol{u}_2|^{2(p+q+1)} + |\bar{\boldsymbol{u}}_2|^{2(p+q)} < n$. Each generated partial S-cover (a_1, b_1) is passed to the next recursive call. When the loop is over, we return to the caller.

2. Using a loop, the next recursive step generates all possible primitively-rooted squares (a_2, b_2) as indicated by Corollary 10. We extend the partial S-cover (a_1, b_1) by the generated (a_2, b_2) and compute all rightmost squares of $\boldsymbol{x}[1\,..\,b_2]$, $B(1, a_2{-}1)$ and $E(1, a_2{-}1)$. For each partial S-cover $(a_1, b_1), (a_2, b_2)$ we perform the following checks, and if they are all successful, the partial S-cover $(a_1, b_1), (a_2, b_2)$ is passed to the next recursive call. When the loop is over, we return to the caller.
   - If $d = 2$ and $B(1, j) \leq j$ for some $2 \leq j < a_2$, then by Corollary 4, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_2, b_2).
   - If $d = 2$ and $B(1, j){-}E(1, j) \leq 2$ for some $2 \leq j < a_2$, then by Corollary 4, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_2, b_2).
   - If $d > 2$, by Lemma 2, if $B(1, j) \leq j{-}d{+}2$ for some $2 \leq j < a_2$, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_2, b_2).
   - If $d > 2$ and $B(1, j){-}E(1, j) = 0$ for some $2 \leq j < a_2$, then by Lemma 2, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_2, b_2).

- If $a_2 > 2$ and there is no rightmost square at the position 2 (there was one in $\boldsymbol{x}[1\mathinner{..}b_1]$ but in the extension $\boldsymbol{x}[1\mathinner{..}b_2]$ there is another occurrence, so it is no longer rightmost), we abandon further extension and jump to the top of the loop to try another (a_2, b_2).
- If $b_2 = n$, we check if the string $\boldsymbol{x}[1\mathinner{..}n]$ has d distinct symbols. If not, we jump to the top of the loop to try another (a_2, b_2).
- If $b_2 = n$ and the number of the rightmost squares is $\leq n{-}d$ we jump to the top of the loop to try another (a_2, b_2). On the other hand, if it is $> n{-}d$, **we stop the execution and announce and display the found counterexample**.
- If $b_2 < n$ and (a_1, b_1) is no longer a rightmost square (there is a further occurrence of $\boldsymbol{x}[a_1\mathinner{..}b_1]$ in $\boldsymbol{x}[a_1\mathinner{..}b_2]$), we abandon the generation of the rest as we are no longer building a viable S-cover and jump to the top of the loop to try another (a_2, b_2).
- If there is a rightmost square that is not completely covered by (a_1, b_1) or (a_2, b_2), (property *(iii)* of Definition 6), again we are not building a viable S-cover and so further generation is abandoned and we jump to the top of the loop to try another (a_2, b_2).
- Otherwise, all checks were successful, and so we pass the partial S-cover (a_1, b_1), (a_2, b_2) to the next recursive call for further extension. When the call returns, we jump to the top of the loop to try another (a_2, b_2).

3. In the next recursive step, the partial S-cover $(a_1, b_1), \ldots, (a_k, b_k)$ for $2 \leq k$ is extended in a loop by all possible combinations of primitively-rooted (a_{k+1}, b_{k+1}). From the definition of S-cover, $a_{k+1} < a_k$ and $b_{k+1} > b_k$. We compute all rightmost squares of $\boldsymbol{x}[1\mathinner{..}b_{k+1}]$, $B(1, a_{k+1}-1)$ and $E(1, a_{k+1}-1)$. For each partial S-cover $(a_1, b_1), \ldots, (a_{k+1}, b_{k+1})$, we perform the following checks, and if they are all successful, the partial S-cover $(a_1, b_1), \ldots, (a_{k+1}, b_{k+1})$ is passed to the next recursive call. When the loop is over, we return to the caller.

- If $d = 2$ and $B(1, j) \leq j$ for some $2 \leq j < a_{k+1}$, then by Corollary 4, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- If $d = 2$ and $B(1, j){-}E(1, j) \leq 2$ for some $2 \leq j < a_{k+1}$, then by Corollary 4, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- If $d > 2$, by Lemma 2, if $B(1, j) \leq j{-}d{+}2$ fro some $2 \leq j < a_{k+1}$, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- If $d > 2$ and $B(1, j){-}E(1, j) = 0$ for some $2 \leq j < a_2$, then by Lemma 2, it cannot be a beginning of a counterexample, so we abandon it and jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- If $a_2 > 2$ and there is no rightmost square at the position 2 (there was one in $\boldsymbol{x}[1\mathinner{..}b_1]$ but in the extension $\boldsymbol{x}[1\mathinner{..}b_{k+1}]$ there is another occurrence, so it is no longer rightmost), we abandon further extension and jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- If $b_{k+1} = n$, we check if the string $\boldsymbol{x}[1\mathinner{..}n]$ has d distinct symbols. If not, we jump to the top of the loop to try another (a_{k+1}, b_{k+1}).

- If $b_2 = n$ and the number of the rightmost squares is $\leq n{-}d$ we jump to the top of the loop to try another (a_2, b_2). On the other hand, if it is $> n{-}d$, **we stop the execution and announce and display the found counterexample**.
- If $b_2 < n$ and (a_1, b_1) is no longer a rightmost square (there is a further occurrence of $\boldsymbol{x}[a_1 \mathinner{..} b_1]$ in $\boldsymbol{x}[a_1 \mathinner{..} b_2]$), we abandon the generation of the rest as we are no longer building a viable S-cover and jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- If there is a rightmost square that is not completely covered by $(a_1, b_1), ..., (a_{k+1}, b_{k+1})$, (property (*iii*) of Definition 6), again we are not building a viable S-cover and so further generation is abandoned and we jump to the top of the loop to try another (a_{k+1}, b_{k+1}).
- Otherwise all checks were successful, and so we pass the partial S-cover $(a_1, b_1), ..., (a_{k+1}, b_{k+1})$ to the next recursive call for further extension. When the call returns, we jump to the top of the loop to try another (a_{k+1}, b_{k+1}).

5 Conclusion and Future Work

In conclusion, we present a framework for computer search for counterexamples to the d-step conjecture, i.e., strings of length n with d distinct symbols that admit strictly more than $n - d$ rightmost squares. The significant restriction on a form of a first counterexample presented in a series of lemmas and corollaries, allows for an early abandonment of partially generated strings that could not possibly be counterexamples, thus, dramatically reducing the search space.

6 Acknowledgment

The research of the first author was supported by the National Sciences and Research Council of Canada (NSERC) grant RGPIN/5504-2018. The research of the second author was supported by the University of Toronto Mississauga's Office of the Vice-Principal, Research (OVPR) Fund.

References

1. F. Blanchet-Sadri, M. Bodnar, J. Nikkel, J. Quigley, and X. Zhang: *Squares and primitivity in partial words.* Discrete Applied Mathematics, 185 2015, pp. 26–37.
2. F. Blanchet-Sadri and S. Osborne: *Constructing words with high distinct square densities.* Electronic Proceedings in Theoretical Computer Science, 252 08 2017, pp. 71–85.
3. A. Deza and F. Franek: *A d-step approach to the maximum number of distinct squares and runs in strings.* Discrete Applied Mathematics, 163 2014, pp. 268–274.
4. A. Deza, F. Franek, and M. Jiang: *Square-maximal strings,* `http://optlab.mcmaster.ca/~jiangm5/research/square.html`.
5. A. Deza, F. Franek, and M. Jiang: *A d-step approach for distinct squares in strings*, in Proceedings of 22nd Annual Symposium on Combinatorial Pattern Matching - CPM 2011, 2011, pp. 11–89.
6. A. Deza, F. Franek, and M. Jiang: *A computational framework for determining square-maximal strings*, in Proceedings of Prague Stringology Conference 2012, J. Holub and J. Žd'árek, eds., Czech Technical University, Prague, Czech Republic, 2012, pp. 112–119.
7. A. Deza, F. Franek, and M. Jiang: *A computational substantiation of the d-step approach to the number of distinct squares problem.* Discrete Applied Mathematics, 2016, pp. 81–87.
8. A. Deza, F. Franek, and A. Thierry: *How many double squares can a string contain?* Discrete Applied Mathematics, 180 2015, pp. 52–69.

9. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* Journal of Combinatorial Theory Series A, 82 1998, pp. 112–120.
10. L. Ilie: *A simple proof that a word of length n has at most $2n$ distinct squares.* Journal of Combinatorial Theory Series A, 112 2005, pp. 163–164.
11. L. Ilie: *A note on the number of squares in a word.* Theoretical Computer Science, 380 2007, pp. 373–376.
12. N. Jonoska, , F. Manea, and S. Seki: *A Stronger Square Conjecture on Binary Words*, in SOFSEM 2014: Theory and Practice of Computer Science, Springer International Publishing, 2014, pp. 339–350.
13. N. Lam: *On the number of squares in a string.* AdvOL-Report 2013/2, McMaster University, 2013.
14. F. Manea and S. Seki: *Square-Density Increasing Mappings*, in Combinatorics on Words, Springer International Publishing, 2015, pp. 160–169.
15. A. Thierry: *A proof that a word of length n has less than $1.5n$ distinct squares. arXiv:2001.02996*, 2020.

Counting Lyndon Subsequences

Ryo Hirakawa[1], Yuto Nakashima[2], Shunsuke Inenaga[2,3], and Masayuki Takeda[2]

[1] Department of Information Science and Technology, Kyushu University, Fukuoka, Japan
`hirakawa.ryo.460@s.kyushu-u.ac.jp`
[2] Department of Informatics, Kyushu University, Fukuoka, Japan
`{yuto.nakashima, inenaga, takeda}@inf.kyushu-u.ac.jp`
[3] PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Abstract. Counting substrings/subsequences that preserve some property (e.g., palindromes, squares) is an important mathematical interest in stringology. Recently, Glen et al. studied the number of Lyndon factors in a string. A string $w = uv$ is called a Lyndon word if it is the lexicographically smallest among all of its conjugates vu. In this paper, we consider a more general problem "counting Lyndon subsequences". We show (1) the maximum total number of Lyndon subsequences in a string, (2) the expected total number of Lyndon subsequences in a string, (3) the expected number of distinct Lyndon subsequences in a string.

1 Introduction

A string $x = uv$ is said to be a *conjugate* of another string y if $y = vu$. A string w is called a *Lyndon word* if it is the lexicographically smallest among all of its conjugates. It is also known that w is a Lyndon word iff w is the lexicographically smallest suffix of itself (excluding the empty suffix).

A *factor* of a string w is a sequence of characters that appear contiguously in w. A factor f of a string w is called a *Lyndon factor* if f is a Lyndon word. Lyndon factors enjoy a rich class of algorithmic and stringology applications including: counting and finding the maximal repetitions (a.k.a. runs) in a string [2] and in a trie [8], constant-space pattern matching [3], comparison of the sizes of run-length Burrows-Wheeler Transform of a string and its reverse [4], substring minimal suffix queries [1], the shortest common superstring problem [7], and grammar-compressed self-index (Lyndon-SLP) [9].

Since Lyndon factors are important combinatorial objects, it is natural to wonder how many Lyndon factors can exist in a string. Regarding this question, the next four types of counting problems are interesting:

- $MTF(\sigma, n)$: the *maximum total* number of Lyndon factors in a string of length n over an alphabet of size σ.
- $MDF(\sigma, n)$: the *maximum* number of *distinct* Lyndon factors in a string of length n over an alphabet of size σ.
- $ETF(\sigma, n)$: the *expected total* number of Lyndon factors in a string of length n over an alphabet of size σ.
- $EDF(\sigma, n)$: the *expected* number of *distinct* Lyndon factors in a string of length n over an alphabet of size σ.

Glen et al. [5] were the first who tackled these problems, and they gave exact values for $MDF(\sigma, n)$, $ETF(\sigma, n)$, and $EDF(\sigma, n)$. Using the number $L(\sigma, n)$ of Lyndon words of length n over an alphabet of size σ, their results can be written as shown in Table 1.

Number of Lyndon Factors in a String	
Maximum Total $MTF(\sigma, n)$	$\binom{n+1}{2} - (\sigma - p)\binom{m+1}{2} - p\binom{m+2}{2} + n$ [this work]
Maximum Distinct $MDF(\sigma, n)$	$\binom{n+1}{2} - (\sigma - p)\binom{m+1}{2} - p\binom{m+2}{2} + \sigma$ [5]
Expected Total $ETF(\sigma, n)$	$\sum_{m=1}^{n} L(\sigma, m)(n - m + 1)\sigma^{-m}$ [5]
Expected Distinct $EDF(\sigma, n)$	$\sum_{m=1}^{n} L(\sigma, m) \sum_{s=1}^{\lfloor n/m \rfloor} (-1)^{s+1}\binom{n - sm + s}{s}\sigma^{-sm}$ [5]

Table 1. The numbers of Lyndon factors in a string of length n over an alphabet of size σ, where $n = m\sigma + p$ with $0 \le p < \sigma$ for $MTF(\sigma, n)$ and $MDF(\sigma, n)$.

The first contribution of this paper is filling the missing piece of Table 1, the exact value of $MTF(\sigma, n)$, thus closing this line of research for Lyndon factors (substrings).

We then extend the problems to subsequences. A subsequence of a string w is a sequence of characters that can be obtained by removing 0 or more characters from w. A subsequence s of a string w is said to be a *Lyndon subsequence* if s is a Lyndon word. As a counterpart of the case of Lyndon factors, it is interesting to consider the next four types of counting problems of Lyndon subsequences:

- $MTS(\sigma, n)$: the *maximum total* number of Lyndon subsequences in a string of length n over an alphabet of size σ.
- $MDS(\sigma, n)$: the *maximum* number of *distinct* Lyndon subsequences in a string of length n over an alphabet of size σ.
- $ETS(\sigma, n)$: the *expected total* number of Lyndon subsequences in a string of length n over an alphabet of size σ.
- $EDS(\sigma, n)$: the *expected* number of *distinct* Lyndon subsequences in a string of length n over an alphabet of size σ.

Among these, we present the exact values for $MTS(\sigma, n)$, $ETS(\sigma, n)$, and $EDS(\sigma, n)$. Our results are summarized in Table 2. Although the main ideas of our proofs are analogous to the results for substrings, there exist differences based on properties of substrings and subsequences.

Number of Lyndon Subsequences in a String	
Maximum Total $MTS(\sigma, n)$	$2^n - (p + \sigma)2^m + n + \sigma - 1$ [this work]
Maximum Distinct $MDS(\sigma, n)$	open
Expected Total $ETS(\sigma, n)$	$\sum_{m=1}^{n}\left[L(\sigma, m)\binom{n}{m}\sigma^{n-m}\right]\sigma^{-n}$ [this work]
Expected Distinct $EDS(\sigma, n)$	$\sum_{m=1}^{n}\left[L(\sigma, m)\sum_{k=m}^{n}\binom{n}{k}(\sigma - 1)^{n-k}\right]\sigma^{-n}$ [this work]

Table 2. The numbers of Lyndon subsequences in a string of length n over an alphabet of size σ, where $n = m\sigma + p$ with $0 \le p < \sigma$ for $MTS(\sigma, n)$.

In the future work, we hope to determine the exact value for $MDS(\sigma, n)$.

2 Preliminaries

2.1 Strings

Let $\Sigma = \{a_1, \ldots, a_\sigma\}$ be an ordered *alphabet* of size σ such that $a_1 < \ldots < a_\sigma$. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The i-th character of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$. A string x is said to be a subsequence of a string w if there exists a set of positions $\{i_1, \ldots, i_{|x|}\}$ $(1 \leq i_1 < \ldots < i_{|x|} \leq |w|)$ such that $x = w[i_1] \cdots w[i_{|x|}]$. We say that a subsequence x occurs at $\{i_1, \ldots, i_{|x|}\}$ $(1 \leq i_1 < \ldots < i_{|x|} \leq |w|)$ if $x = w[i_1] \cdots w[i_{|x|}]$.

2.2 Lyndon words

A string $x = uv$ is said to be a *conjugate* of another string y if $y = vu$. A string w is called a *Lyndon word* if it is the lexicographically smallest among all of its conjugates. Equivalently, a string w is said to be a Lyndon word, if w is lexicographically smaller than all of its non-empty proper suffixes.

Let μ be the *Möbius function* on the set of positive integers defined as follows.

$$\mu(n) = \begin{cases} 1 & (n = 1) \\ 0 & (\text{if } n \text{ is divisible by a square}) \\ (-1)^k & (\text{if } n \text{ is the product of } k \text{ distinct primes}) \end{cases}$$

It is known that the number $L(\sigma, n)$ of Lyndon words of length n over an alphabet of size σ can be represented as

$$L(\sigma, n) = \frac{1}{n} \sum_{d|n} \mu\left(\frac{n}{d}\right) \sigma^d,$$

where $d|n$ is the set of divisors d of n [6].

3 Maximum total number of Lyndon subsequences

Let $MTS(\sigma, n)$ be the maximum total number of Lyndon subsequences in a string of length n over an alphabet Σ of size σ. In this section, we determine $MTS(\sigma, n)$.

Theorem 1. *For any σ and n such that $\sigma < n$,*

$$MTS(\sigma, n) = 2^n - (p + \sigma)2^m + n + \sigma - 1$$

where $n = m\sigma + p$ $(0 \leq p < \sigma)$. Moreover, the number of strings that contain $MTS(\sigma, n)$ Lyndon subsequences is $\binom{\sigma}{p}$, and the following string w is one of such strings;

$$w = {a_1}^m \cdots {a_{\sigma-p}}^m {a_{\sigma-p+1}}^{m+1} \cdots {a_\sigma}^{m+1}.$$

Proof. Consider a string w of the form

$$w = {a_1}^{k_1}{a_2}^{k_2}\cdots{a_\sigma}^{k_\sigma}$$

where $\sum_{i=1}^{\sigma} k_i = n$ and $k_i \geq 0$ for any i. For any subsequence x of w, x is a Lyndon word if x is not a unary string of length at least 2. It is easy to see that this form is a necessary condition for the maximum number ($\because$ there exist several non-Lyndon subsequences if $w[i] > w[j]$ for some $i < j$). Hence, the number of Lyndon subsequences of w can be represented as

$$\begin{aligned}(2^n - 1) - \sum_{i=1}^{\sigma}(2^{k_i} - 1 - k_i) &= 2^n - 1 - \sum_{i=1}^{\sigma} 2^{k_i} + \sum_{i=1}^{\sigma} k_i + \sigma \\ &= 2^n - 1 - \sum_{i=1}^{\sigma} 2^{k_i} + n + \sigma.\end{aligned}$$

This formula is maximized when $\sum_{i=1}^{\sigma} 2^{k_i}$ is minimized. It is known that

$$2^a + 2^b > 2^{a-1} + 2^{b+1}$$

holds for any integer a, b such that $a \geq b + 2$. From this fact, $\sum_{i=1}^{\sigma} 2^{k_i}$ is minimized when the difference of k_i and k_j is less than or equal to 1 for any i, j. Thus, if we choose p k_i's as $m+1$, and set m for other $(\sigma - p)$ k_i's where $n = m\sigma + p$ $(0 \leq p < \sigma)$, then $\sum_{i=1}^{\sigma} 2^{k_i}$ is minimized. Hence,

$$\begin{aligned}\min(2^n - 1 - \sum_{i=1}^{\sigma} 2^{k_i} + n + \sigma) &= 2^n - 1 - p \cdot 2^{m+1} - (\sigma - p)2^m + n + \sigma \\ &= 2^n - (p + \sigma)2^m + n + \sigma - 1\end{aligned}$$

Moreover, one of such strings is

$${a_1}^m \cdots {a_{\sigma-p}}^m {a_{\sigma-p+1}}^{m+1} \cdots {a_\sigma}^{m+1}.$$

Therefore, this theorem holds. □

We can apply the above strategy to the version of substrings. Namely, we can also obtain the following result.

Corollary 2. *Let $MTF(\sigma, n)$ be the maximum total number of Lyndon substrings in a string of length n over an alphabet of size σ. For any σ and n such that $\sigma < n$,*

$$MTF(\sigma, n) = \binom{n}{2} - (\sigma - p)\binom{m+1}{2} - p\binom{m+2}{2} + n$$

where $n = m\sigma + p$ $(0 \leq p < \sigma)$. Moreover, the number of strings that contain $MTF(\sigma, n)$ Lyndon subsequences is $\binom{\sigma}{p}$, and the following string w is one of such strings;

$$w = {a_1}^m \cdots {a_{\sigma-p}}^m {a_{\sigma-p+1}}^{m+1} \cdots {a_\sigma}^{m+1}.$$

Proof. Consider a string w of the form

$$w = {a_1}^{k_1}{a_2}^{k_2}\cdots{a_\sigma}^{k_\sigma}$$

where $\sum_{i=1}^{\sigma} k_i = n$ and $k_i \geq 0$ for any i. In a similar way to the above discussion, the number of Lyndon substrings of w can be represented as

$$\binom{n+1}{2} - \sum_{i=1}^{\sigma}\left[\binom{k_i+1}{2} - k_i\right] = \binom{n+1}{2} - \sum_{i=1}^{\sigma}\binom{k_i+1}{2} + n.$$

We can use the following inequation that holds for any a, b such that $a \geq b+2$;

$$\binom{a}{2} + \binom{b}{2} > \binom{a-1}{2} + \binom{b+1}{2}.$$

Then,

$$\min\left[\binom{n+1}{2} - \sum_{i=1}^{\sigma}\binom{k_i+1}{2} + n\right] = \binom{n}{2} - (\sigma - p)\binom{m+1}{2} - p\binom{m+2}{2} + n$$

holds. □

Finally, we give exact values $MTS(\sigma, n)$ for several conditions in Table 3.

n	$MTS(2,n)$	$MTS(5,n)$	$MTS(10,n)$
1	1	1	1
2	3	3	3
3	6	7	7
4	13	15	15
5	26	31	31
6	55	62	63
7	122	125	127
8	233	252	255
9	474	507	511
10	971	1018	1023
11	1964	2039	2046
12	3981	4084	4093
13	8014	8177	8188
14	16143	16366	16379
15	32400	32747	32762

Table 3. Values $MTS(\sigma, n)$ for $\sigma = 2, 5, 10$, $n = 1, 2, \cdots, 15$.

4 Expected total number of Lyndon subsequences

Let $TS(\sigma, n)$ be the total number of Lyndon subsequences in all strings of length n over an alphabet Σ of size σ. In this section, we determine the expected total number $ETS(\sigma, n)$ of Lyndon subsequences in a string of length n over an alphabet Σ of size σ, namely, $ETS(\sigma, n) = TS(\sigma, n)/\sigma^n$.

Theorem 3. *For any σ and n such that $\sigma < n$,*

$$TS(\sigma, n) = \sum_{m=1}^{n}\left[L(\sigma, m)\binom{n}{m}\sigma^{n-m}\right].$$

Moreover, $ETS(\sigma, n) = TS(\sigma, n)/\sigma^n$.

Proof. Let $Occ(w,x)$ be the number of occurrences of subsequence x in w, and $L(\sigma,n)$ the set of Lyndon words of length less than or equal to n over an alphabet of size σ. By a simple observation, $TS(\sigma,n)$ can be written as

$$TS(\sigma,n) = \sum_{x\in\mathcal{L}(\sigma,n)} \sum_{w\in\Sigma^n} Occ(w,x).$$

Firstly, we consider $\sum_{w\in\Sigma^n} Occ(w,x)$ for a Lyndon word x of length m. Let $\{i_1,\ldots,i_m\}$ be a set of m positions in a string of length n where $1 \leq i_1 < \ldots < i_m \leq n$. The number of strings that contain x as a subsequence at $\{i_1,\ldots,i_m\}$ is σ^{n-m}. In addition, the number of combinations of m positions is $\binom{n}{m}$. Hence, $\sum_{w\in\Sigma^n} Occ(w,x) = \binom{n}{m}\sigma^{n-m}$. This implies that

$$TS(\sigma,n) = \sum_{m=1}^{n}\left[L(\sigma,m)\binom{n}{m}\sigma^{n-m}\right].$$

Finally, since the number of strings of length n over an alphabet of size σ is σ^n, $ETS(\sigma,n) = TS(\sigma,n)/\sigma^n$. Therefore, this theorem holds. □

Finally, we give exact values $TS(\sigma,n), ETS(\sigma,n)$ for several conditions in Table 4.

n	$TS(2,n)$	$ETS(2,n)$	$TS(5,n)$	$ETS(5,n)$
1	2	1.00	5	1.00
2	9	2.25	60	2.40
3	32	4.00	565	4.52
4	107	6.69	4950	7.92
5	356	11.13	42499	13.60
6	1205	18.83	365050	23.36
7	4176	32.63	3163435	40.49
8	14798	57.80	27731650	70.99
9	53396	104.29	245950375	125.93
10	195323	190.75	2204719998	225.76

Table 4. Values $TS(\sigma,n), ETS(\sigma,n)$ for $\sigma = 2,5, n = 1,2,\cdots,10$.

5 Expected number of distinct Lyndon subsequences

Let $TDS(\sigma,n)$ be the total number of distinct Lyndon subsequences in all strings of length n over an alphabet Σ of size σ. In this section, we determine the expected number $EDS(\sigma,n)$ of distinct Lyndon subsequences in a string of length n over an alphabet Σ of size σ, namely, $EDS(\sigma,n) = TDS(\sigma,n)/\sigma^n$.

Theorem 4. *For any σ and n such that $\sigma < n$,*

$$TDS(\sigma,n) = \sum_{m=1}^{n}\left[L(\sigma,m)\sum_{k=m}^{n}\binom{n}{k}(\sigma-1)^{n-k}\right].$$

Moreover, $EDS(\sigma,n) = TDS(\sigma,n)/\sigma^n$.

To prove this theorem, we introduce the following lemmas.

Lemma 5. *For any $x_1, x_2 \in \Sigma^m$ and m, n $(m \leq n)$, the number of strings in Σ^n which contain x_1 as a subsequence is equal to the number of strings in Σ^n which contain x_2 as a subsequence.*

Proof (of Lemma 5). Let $C(n, \Sigma, x)$ be the number of strings in Σ^n which contain a string x as a subsequence. We prove $C(n, \Sigma, x_1) = C(n, \Sigma, x_2)$ for any $x_1, x_2 \in \Sigma^m$ by induction on the length m.

Suppose that $m = 1$. It is clear that the set of strings which contain $x \in \Sigma$ is $\Sigma^n - (\Sigma - \{x\})^n$, and $C(n, \Sigma, x) = \sigma^n - (\sigma - 1)^n$. Thus, $C(n, \Sigma, x_1) = C(n, \Sigma, x_2)$ for any x_1, x_2 if $|x_1| = |x_2| = 1$.

Suppose that the statement holds for some $k \geq 1$. We prove $C(n, \Sigma, x_1) = C(n, \Sigma, x_2)$ for any $x_1, x_2 \in \Sigma^{k+1}$ by induction on n. If $n = k + 1$, then $C(n, \Sigma, x_1) = C(n, \Sigma, x_2) = 1$. Assume that the statement holds for some $\ell \geq k + 1$. Let $x = yc$ be a string of length $k + 1$ such that $y \in \Sigma^k, c \in \Sigma$. Each string w of length $\ell + 1$ which contains x as a subsequence satisfies either

- $w[1..\ell]$ contains x as a subsequence, or
- $w[1..\ell]$ does not contain x as a subsequence.

The number of strings w in the first case is $\sigma \cdot C(j, \Sigma, yc)$. On the other hand, the number of strings w in the second case is $C(\ell, \Sigma, y) - C(\ell, \Sigma, yc)$. Hence, $C(\ell + 1, \Sigma, x) = \sigma C(\ell, \Sigma, yc) + C(\ell, \Sigma, y) - C(\ell, \Sigma, yc)$. Let $x_1 = y_1c_1$ and $x_2 = y_2c_2$ be strings of length $k + 1$. By an induction hypothesis, $C(\ell, \Sigma, y_1c_1) = C(\ell, \Sigma, y_2c_2)$ and $C(\ell, \Sigma, y_1) = C(\ell, \Sigma, y_2)$ hold. Thus, $C(\ell + 1, \Sigma, x_1) = C(\ell + 1, \Sigma, x_2)$ also holds.

Therefore, this lemma holds. □

Lemma 6. *For any string x of length $m \leq n$,*

$$C(n, \Sigma, x) = \sum_{k=m}^{n} \binom{n}{k} (\sigma - 1)^{n-k}.$$

Proof (of Lemma 6). For any character c, it is clear that the number of strings that contain c exactly k times is $\binom{n}{k}(\sigma - 1)^{n-k}$. By Lemma 5,

$$C(n, \Sigma, x) = C(n, \Sigma, c^m) = \sum_{k=m}^{n} \binom{n}{k} (\sigma - 1)^{n-k}.$$

Hence, this lemma holds. □

Then, we can obtain Theorem 4 as follows.

Proof (of Theorem 4). Thanks to Lemma 6, the number of strings of length n which contain a Lyndon word of length m is also $\sum_{k=m}^{n} \binom{n}{k}(\sigma - 1)^{n-k}$. Since the number of Lyndon words of length m over an alphabet of size σ is $L(\sigma, m)$,

$$TDS(\sigma, n) = \sum_{m=1}^{n} \left[L(\sigma, m) \sum_{k=m}^{n} \binom{n}{k} (\sigma - 1)^{n-k} \right].$$

Finally, since the number of strings of length n over an alphabet of size σ is σ^n, $EDS(\sigma, n) = TDS(\sigma, n)/\sigma^n$. Therefore, Theorem 4 holds. □

We give exact values $EDS(\sigma, n)$ for several conditions in Table 5.

n	$EDS(2,n)$	$EDS(5,n)$
1	1.00	1.00
2	1.75	2.20
3	2.50	3.80
4	3.38	6.09
5	4.50	9.51
6	6.00	14.80
7	8.03	23.12
8	10.81	36.43
9	14.63	57.95
10	19.93	93.08
15	100.57	1121.29
20	559.42	15444.90

Table 5. Values $EDS(\sigma, n)$ for $\sigma = 2, 5, n = 1, \ldots, 10, 15, 20$.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP18K18002 (YN), JP21K17705 (YN), JP18H04098 (MT), JST ACT-X Grant Number JPMJAX200K (YN), and by JST PRESTO Grant Number JPMJPR1922 (SI).

References

1. M. A. Babenko, P. Gawrychowski, T. Kociumaka, I. I. Kolesnichenko, and T. Starikovskaya: *Computing minimal and maximal suffixes of a substring.* Theor. Comput. Sci., 638 2016, pp. 112–121.
2. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "runs" theorem.* SIAM J. Comput., 46(5) 2017, pp. 1501–1514.
3. M. Crochemore and D. Perrin: *Two-way string matching.* J. ACM, 38(3) 1991, pp. 651–675.
4. S. Giuliani, S. Inenaga, Z. Lipták, N. Prezza, M. Sciortino, and A. Toffanello: *Novel results on the number of runs of the Burrows-Wheeler-transform*, in SOFSEM 2021, vol. 12607 of Lecture Notes in Computer Science, Springer, 2021, pp. 249–262.
5. A. Glen, J. Simpson, and W. F. Smyth: *Counting Lyndon factors.* The Electronic Journal of Combinatorics, 24 2017, p. P3.28.
6. M. Lothaire: *Combinatorics on Words*, Addison-Wesley, 1983.
7. M. Mucha: *Lyndon words and short superstrings*, in Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013, SIAM, 2013, pp. 958–972.
8. R. Sugahara, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Computing runs on a trie*, in CPM 2019, vol. 128 of LIPIcs, 2019, pp. 23:1–23:11.
9. K. Tsuruta, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Grammar-compressed self-index with Lyndon words.* IPSJ Transactions on Mathematical Modeling and its Applications (TOM), 13(2) 2020, pp. 84–92.

The n-ary Initial Literal and Literal Shuffle

Stefan Hoffmann

Informatikwissenschaften, FB IV, Universität Trier
Universit atsring 15, 54296 Trier, Germany
hoffmanns@informatik.uni-trier.de

Abstract. The literal and the initial literal shuffle have been introduced to model the behavior of two synchronized processes. However, it is not possible to describe the synchronization of multiple processes. Furthermore, both restricted forms of shuffling are not associative. Here, we extend the literal shuffle and the initial literal shuffle to multiple arguments. We also introduce iterated versions, much different from the iterated ones previously introduced for the binary literal and initial literal shuffle. We investigate formal properties, and show that in terms of expressive power, in a full trio, they coincide with the general shuffle. Furthermore, we look at closure properties with respect to the regular, context-free, context-sensitive, recursive and recursively enumerable languages for all operations introduced. Then, we investigate various decision problems motivated by analogous problems for the (ordinary) shuffle operation. Most problems we look at are tractable, but we also identify one intractable decision problem.

Keywords: shuffle, literal shuffle, initial literal shuffle, formal language theory

1 Motivation and Contribution

In [2, 3], the *initial literal shuffle* and the *literal shuffle* were introduced, by giving the following natural motivation (taken from [2, 3]):

> [...] The shuffle operation naturally appears in several problems, like concurrency of processes [13, 21, 25] or multipoint communication, where all stations share a single bus [13]. That is one of the reasons of the large theoretical literature about this operation (see, for instance [1, 9, 13, 14, 15, 20]). In the latter example [of midpoint communication], the general shuffle operation models the asynchronous case, where each transmitter uses asynchronously the single communication channel. If the hypothesis of synchronism is made (step-lock transmission), the situation is modelled by what can be named 'literal' shuffle. Each transmitter emits, in turn, one elementary signal. The same remark holds for concurrency, where general shuffle corresponds to asynchronism and literal shuffle to synchronism. [...]

So, the shuffle operation corresponds to the parallel composition of words, which model instructions or event sequences of processes, i.e., *sequentialized execution histories of concurrent processes.*

In this framework, the initial literal shuffle is motivated by modelling the synchronous operation of two processes that start at the same point in time, whereas the literal shuffle could model the synchronous operation if started at different points in time. However, both restricted shuffle variants are only binary operations, which are not associative. Hence, actually only the case of *two* processes synchronized to each other is modelled, i.e., the (initial) literal shuffling applied multiple times, in any order, does not model adequately the synchronous operation of multiple processes. So, the iterative versions, as introduced in [2, 3], are not adequate to model multiple processes, and, because of the lack of associativity, the bracketing is essential, which is, from a mathematical point of view, somewhat unsatisfying.

Here, we built up on [2, 3] by extending both restricted shuffle variants to multiple arguments, which do not arise as the combination of binary operations. So, technically, for each n we have a different operation taking n arguments. With these operations, we derive iterated variants in a uniform manner. We introduce two variants:

(1) the *n-ary initial literal shuffle*, motivated by modelling n synchronous processes started at the same point in time;
(2) the *n-ary literal shuffle*, motivated by modelling n synchronous processes started at different points in time.

Additionally, in Section 7, we also introduce two additional variants for which the results are independent of the order of the arguments. Hence, our variants might be used when a more precise approach is necessary than the general shuffle can provide.

We study the above mentioned operations and their iterative variants, their relations to each other and their expressive power. We also study their closure properties with respect to the classical families of the Chomsky hierarchy [12] and the recursive languages. We also show that, when adding the full trio operations, the expressive power of each shuffle variant is as powerful as the general shuffle operation. In terms of computational complexity, most problem we consider, which are motivated from related decision problems for the (general) shuffle operation, are tractable. However, we also identify a decision problem for the second variant that is NP-complete.

The goal of the present work is to give an analysis of these operations from the point of view of *formal language theory.*

2 The Shuffle Operation in Formal Language Theory

Beside [2, 3], we briefly review other work related to the shuffle operation. We focus on research in formal language theory and computational complexity.

The shuffle and iterated shuffle have been introduced and studied to understand the semantics of parallel programs. This was undertaken, as it appears to be, independently by Campbell and Habermann [7], by Mazurkiewicz [23] and by Shaw [28]. They introduced *flow expressions*, which allow for sequential operators (catenation and iterated catenation) as well as for parallel operators (shuffle and iterated shuffle). See also [26, 27] for an approach using only the ordinary shuffle, but not the iterated shuffle. Starting from this, various subclasses of the flow expressions were investigated [4, 6, 15, 16, 17, 18, 19, 31].

Beside the literal shuffle [2, 3], and the variants introduced in this work, other variants and generalizations of the shuffle product were introduced. Maybe the most versatile is *shuffle by trajectories* [22], whereby the selection of the letters from two input words is controlled by a given *language of trajectories* that indicates, as a binary language, at which positions letters from the first or second word are allowed. This framework entails numerous existing operations, from concatenation up to the general shuffle, and in [22] the authors related algebraic properties and decision procedures of resulting operations to properties of the trajectory language. In [30], with a similar motivation as ours, different notions of *synchronized shuffles* were introduced. But in this approach, two words have to "link", or synchronize, at a specified subword drawn from a subalphabet (the letters, or actions, that should be synchronized), which the authors termed the *backbone*. Hence, their approach differs in that the synchronization appears letter-wise, whereas here we synchronize position-wise, i.e., at specific points in time the actions occur together in steps, and are not merged as in [30].

3 Preliminaries and Definitions

By $\mathbb{N}_0$ we denote the *natural numbers* including zero. The *symmetric group*, i.e., the set of all permutations with function composition as operation, is $\mathcal{S}_n = \{f : \{1,\ldots,n\} \to \{1,\ldots,n\} \mid f \text{ bijective}\}$.

By Σ we denote a finite set of symbols, called an *alphabet*. The set Σ^* denotes the set of all finite sequences, i.e., of all words with the concatenation operation. The finite sequence of length zero, or the *empty word*, is denoted by ε. Subsets of Σ^* are called *languages*. For a given word, we denote by $|w|$ its length, and for $a \in \Sigma$ by $|w|_a$ the number of occurrences of the symbol a in w. For a word $w = u_1 \cdots u_n$ with $u_i \in \Sigma$, $i \in \{1,\ldots,n\}$, we write $w^R = u_n \cdots u_1$ for the *mirror operation*. For $L \subseteq \Sigma^*$ we set $L^+ = \bigcup_{i=1}^{\infty} L^i$ and $L^* = L^+ \cup \{\varepsilon\}$, where we set $L^1 = L$ and $L^{i+1} = \{uv \mid u \in L^i, v \in L\}$ for $i \geq 1$.

A *finite deterministic and complete automaton* will be denoted by $\mathcal{A} = (\Sigma, S, \delta, s_0, F)$ with $\delta : S \times \Sigma \to S$ the state transition function, S a finite set of states, $s_0 \in S$ the start state and $F \subseteq S$ the set of final states. The properties of being deterministic and complete are implied by the definition of δ as a total function. The transition function $\delta : S \times \Sigma \to S$ could be extended to a transition function on words $\delta^* : S \times \Sigma^* \to S$ by setting $\delta^*(s, \varepsilon) := s$ and $\delta^*(s, wa) := \delta(\delta^*(s,w), a)$ for $s \in S$, $a \in \Sigma$ and $w \in \Sigma^*$. In the remainder we drop the distinction between both functions and also denote this extension by δ. The *language accepted* by an automaton $\mathcal{A} = (\Sigma, S, \delta, s_0, F)$ is $L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta(s_0, w) \in F\}$. A language $L \subseteq \Sigma^*$ is called *regular* if $L = L(\mathcal{A})$ for some finite automaton.

Definition 1. *The* shuffle operation*, denoted by* $\sqcup\!\sqcup$*, is defined by*

$$\begin{aligned} u \sqcup\!\sqcup v = \{w \in \Sigma^* \mid\ & w = x_1 y_1 x_2 y_2 \cdots x_n y_n \text{ for some words} \\ & x_1, \ldots, x_n, y_1, \ldots, y_n \in \Sigma^* \text{ such that } u = x_1 x_2 \cdots x_n \text{ and } v = y_1 y_2 \cdots y_n\}, \end{aligned}$$

for $u, v \in \Sigma^*$ *and* $L_1 \sqcup\!\sqcup L_2 := \bigcup_{x \in L_1, y \in L_2} (x \sqcup\!\sqcup y)$ *for* $L_1, L_2 \subseteq \Sigma^*$.

Example 2. $\{ab\} \sqcup\!\sqcup \{cd\} = \{abcd, acbd, acdb, cadb, cdab, cabd\}$

The shuffle operation is commutative, associative and distributive over union. We will use these properties without further mention. In writing formulas without brackets we suppose that the shuffle operation binds stronger than the set operations, and the concatenation operator has the strongest binding. For $L \subseteq \Sigma^*$ the *iterated shuffle* is $L^{\sqcup\!\sqcup,*} = \bigcup_{i=0}^{\infty} L^{\sqcup\!\sqcup,i}$ with $L^{\sqcup\!\sqcup,0} = \{\varepsilon\}$ and $L^{\sqcup\!\sqcup,i+1} = L \sqcup\!\sqcup L^{\sqcup\!\sqcup,i}$. The *positive iterated shuffle* is $L^{\sqcup\!\sqcup,+} = \bigcup_{i=1}^{\infty} L^{\sqcup\!\sqcup,i}$.

A *full trio* [10] is a family of languages closed under homomorphisms, inverse homomorphisms and intersections with regular sets. A full trio is closed under arbitrary intersection if and only if it is closed under shuffle [9]. Also, by a theorem of Nivat [24], a family of languages forms a full trio if and only if it is closed under *generalized sequential machine mappings (gsm mappings)*, also called *finite state transductions*. For the definition of gsm mappings, as well as of *context-sensitive* and *recursively enumerable languages*, we refer to the literature, for example [12]. For two arguments, the *interleaving operation* (or *perfect shuffle* [11]) was introduced in [2, 3]. Here, we give a straightforward generalization for multiple, equal-length, input words.

Definition 3 (n-ary interleaving operation). *Let $n \geq 1$, $k \geq 0$, $u_1, \ldots, u_n \in \Sigma^k$. If $k > 0$, write $u_i = x_1^{(i)} \cdots x_k^{(i)}$, $x_j^{(i)} \in \Sigma$ for $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, n\}$. Then we define $I : (\Sigma^k)^n \to \Sigma^{nk}$ by*

$$I(u_1, \ldots, u_n) = x_1^{(1)} \cdots x_1^{(n)} x_2^{(1)} \cdots x_2^{(n)} \cdot \ldots \cdot x_k^{(1)} \cdots x_k^{(n)}.$$

If $k = 0$, then $I(\varepsilon, \ldots, \varepsilon) = \varepsilon$.

Example 4. $I(aab, bbb, aaa) = abaababba$.

If we interleave all equal-length words in given regular languages, the resulting language is still regular.

Proposition 5. *Let $L_1, \ldots, L_n \subseteq \Sigma^*$ be regular. Then $\{I(u_1, \ldots, u_n) \mid \exists m \geq 0\ \forall i \in \{1, \ldots, n\} : u_i \in L_i \cap \Sigma^m\}$ is regular.*

In [2, 3], the *initial literal shuffle* and the *literal shuffle* were introduced.

Definition 6 ([2, 3]). *Let $U, V \subseteq \Sigma^*$. The* initial literal shuffle *of U and V is*

$$U \sqcup\!\sqcup_1 V = \{I(u, v)w \mid u, v, w \in \Sigma^*, |u| = |v|, (uw \in U, v \in V) \text{ or } (u \in U, vw \in V)\}.$$

and the literal shuffle *is*

$$\begin{aligned} U \sqcup\!\sqcup_2 V = \{w_1 I(u, v) w_2 \mid\ & w_1, u, v, w_2 \in \Sigma^*, |u| = |v|, \\ & (w_1 u w_2 \in U, v \in V) \text{ or } (u \in U, w_1 v w_2 \in V) \text{ or } \\ & (w_1 u \in U, v w_2 \in V) \text{ or } (u w_2 \in U, w_1 v \in V)\}. \end{aligned}$$

Example 7. $\{abc\} \sqcup\!\sqcup_1 \{de\} = \{adbec\}$, $\{abc\} \sqcup\!\sqcup_2 \{de\} = \{abcde, abdce, adbec, daebc, deabc\}$.

The following iterative variants were introduced in [2, 3].

Definition 8 ([2, 3]). *Let $L \subseteq \Sigma^*$. For $i \in \{1, 2\}$, set*

$$L^{\sqcup\!\sqcup_i^*} = \bigcup_{n \geq 0} L_n, \text{ where } L_0 = \{\varepsilon\} \text{ and } L_{n+1} = L_n \sqcup\!\sqcup_i L.$$

The next results are stated in [2, 3].

Proposition 9 ([2, 3]). *Let $\mathcal{L}$ be a full trio. The following are equivalent:*

1. *$\mathcal{L}$ is closed under shuffle.*
2. *$\mathcal{L}$ is closed under literal shuffle.*
3. *$\mathcal{L}$ is closed under initial literal shuffle.*

Proposition 10 ([2, 3]). *Let $F \subseteq \Sigma^*$ be finite. Then $F^{\sqcup\!\sqcup_1^*}$ is regular.*

4 The n-ary Initial Literal and Literal Shuffle

Here, for any number of arguments, we introduce both shuffle variants, define iterated versions and state basic properties.

Definition 11. *Let $u_1, \ldots, u_n \in \Sigma^*$ and $N = \max\{|u_i| \mid i \in \{1, \ldots, n\}\}$. Set*

1. $\shuffle_1^n(u_1, \ldots, u_n) = h(I(u_1\$^{N-|u_1|}, \ldots, u_n\$^{N-|u_n|}))$ *and*
2. $\shuffle_2^n(u_1, \ldots, u_n) = \{h(I(v_1, \ldots, v_n)) \mid v_i \in U_i, i \in \{1, \ldots, n\}\}$,

where $U_i = \{\$^k u_i \$^{r-k} \mid 0 \le k \le r$ with $r = n \cdot N - |u_i|\}$ for $i \in \{1, \ldots, n\}$ and $h : (\Sigma \cup \{\$\})^* \to \Sigma^*$ is the homomorphism given by $h(\$) = \varepsilon$ and $h(x) = x$ for $x \in \Sigma$.

Note that writing the number of arguments in the upper index should pose no problem or confusion with the power operator on functions, as the n-ary shuffle variants are only of interest for $n \ge 2$, and raising a function to a power only makes sense for function with a single argument.

For languages $L_1, \ldots, L_n \subseteq \Sigma^*$, we set

$$\shuffle_1^n(L_1, \ldots, L_n) = \bigcup_{u_1 \in L_1, \ldots u_n \in L_n} \{\shuffle_1^n(u_1, \ldots, u_n)\}$$
$$\shuffle_2^n(L_1, \ldots, L_n) = \bigcup_{u_1 \in L_1, \ldots u_n \in L_n} \shuffle_2^n(u_1, \ldots, u_n).$$

Example 12. Let $u = a, v = bb, w = c$. Then $\shuffle_1^3(u, v, w) = abcb$ and

$$\shuffle_2^3(u, v, w) = \{bbac, babc, abcb, acbb, abbc, bbca, bcba, bcab, cbab, cabb, cbba\}$$
$$\shuffle_2^3(v, u, w) = \{bbac, babc, bacb, abcb, abbc, acbb, cbab, cbba, cabb, bcba, bbca\}$$

We see $bacb \notin \shuffle_2^3(u, v, w)$, but $bacb \in \shuffle_2^3(v, u, w)$.

Example 13. Please see Figure 1 for a graphical depiction of the word

$$a_1^{(1)} a_2^{(1)} a_3^{(1)} a_4^{(1)} a_1^{(2)} a_5^{(1)} a_2^{(2)} a_6^{(1)} a_3^{(2)} a_1^{(3)} a_7^{(1)} a_4^{(2)} a_2^{(3)} a_5^{(2)} a_3^{(3)} a_6^{(2)} a_7^{(2)} a_8^{(2)} a_9^{(2)}$$

from $\shuffle_2^3(u, v, w)$ with

$$u = a_1^{(1)} a_2^{(1)} a_3^{(1)} a_4^{(1)} a_5^{(1)} a_6^{(1)} a_7^{(1)},$$
$$v = a_1^{(2)} a_2^{(2)} a_3^{(2)} a_4^{(2)} a_5^{(2)} a_6^{(2)} a_7^{(2)} a_8^{(2)} a_9^{(2)},$$
$$w = a_1^{(3)} a_2^{(3)} a_3^{(3)}.$$

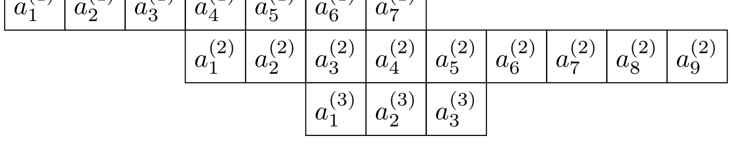

$a_1^{(1)}$ $a_2^{(1)}$ $a_3^{(1)}$ $a_4^{(1)}$ $a_5^{(1)}$ $a_6^{(1)}$ $a_7^{(1)}$
$a_1^{(2)}$ $a_2^{(2)}$ $a_3^{(2)}$ $a_4^{(2)}$ $a_5^{(2)}$ $a_6^{(2)}$ $a_7^{(2)}$ $a_8^{(2)}$ $a_9^{(2)}$
$a_1^{(3)}$ $a_2^{(3)}$ $a_3^{(3)}$

Figure 1. Graphical depiction of a word in $\shuffle_2^3(u, v, w)$. See Example 13.

Now, we can show that the two introduced n-ary literal shuffle variants generalize the initial literal shuffle and the literal shuffle from [2, 3].

Lemma 14. *Let $U, V \subseteq \Sigma^*$ be two languages. Then*

$$\sqcup\!\sqcup_1(U, V) = \sqcup\!\sqcup_1^2(U, V) \quad \textit{and} \quad \sqcup\!\sqcup_2(U, V) = \sqcup\!\sqcup_2^2(U, V).$$

We can also write the second n-ary variant in terms of the first and the mirror operation, as stated in the next lemma.

Lemma 15. *Let $u_1, \ldots, u_n \in \Sigma^*$. Then*

$$\sqcup\!\sqcup_2^n(u_1, \ldots, u_n) = \bigcup_{\substack{x_1,\ldots,x_n \in \Sigma^* \\ y_1,\ldots,y_n \in \Sigma^* \\ u_i = x_i y_i}} \{(\sqcup\!\sqcup_1^n(x_1^R, \ldots, x_n^R))^R \cdot \sqcup\!\sqcup_1^n(y_1, \ldots, y_n)\}$$

With these multiple-argument versions, we define an "iterated" version, where iteration is not meant in the usual sense because of the lack of associativity for the binary argument variants.

Definition 16. *Let $L \subseteq \Sigma^*$ be a language. Then, for $i \in \{1, 2\}$, define*

$$L^{\sqcup\!\sqcup_i, \circledast} = \{\varepsilon\} \cup \bigcup_{n \geq 1} \sqcup\!\sqcup_i^n(L, \ldots, L).$$

For any $i \in \{1, 2\}$, as $\sqcup\!\sqcup_i^1(L) = L$, we have $L \subseteq L^{\sqcup\!\sqcup_i, \circledast}$. Now, let us investigate some properties of the operations under consideration.

Proposition 17. *Let $L_1, \ldots, L_n \subseteq \Sigma^*$ and $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ a permutation. Then*

1. *$L_{\pi(1)} \cdots L_{\pi(n)} \subseteq \sqcup\!\sqcup_2^n(L_1, \ldots, L_n)$.*
2. *Let $k \in \mathbb{N}_0$. Then $\sqcup\!\sqcup_2^n(L_1, \ldots, L_n) = \sqcup\!\sqcup_2^n(L_{((1+k-1) \bmod n)+1}, \ldots, L_{((n+k-1) \bmod n)+1})$.*
3. *$\sqcup\!\sqcup_1^n(L_1, \ldots, L_n) \subseteq \sqcup\!\sqcup_2^n(L_1, \ldots, L_n) \subseteq L_1 \sqcup\!\sqcup \cdots \sqcup\!\sqcup L_n$;*
4. *$L_1^* \subseteq L_1^{\sqcup\!\sqcup_2, \circledast}$;*
5. *$\Sigma^* = \Sigma^{\sqcup\!\sqcup_i, \circledast}$ for $i \in \{1, 2\}$;*
6. *$L_1^{\sqcup\!\sqcup_1, \circledast} \subseteq L_1^{\sqcup\!\sqcup_2, \circledast} \subseteq L_1^{\sqcup\!\sqcup, *}$;*
7. *for $u_1, \ldots, u_n, u \in \Sigma^*$, if $u \in \sqcup\!\sqcup_i^n(\{u_1\}, \ldots, \{u_n\})$, then $|u| = |u_1| + \cdots + |u_n|$.*

Proof (sketch). We only give a rough outline for Property 2. Let $x_{i,j} \in \Sigma$ for $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$. Then, the main idea is to use the equations

$$\begin{aligned}
&h(I(x_{1,1} \cdots x_{1,m}, x_{2,1} \cdots x_{2,m}, \ldots, x_{n,1} \cdots x_{n,m})) \\
&= h((x_{1,1} \cdots x_{n,1})(x_{1,2} \cdots x_{n,2}) \cdots (x_{1,m} \cdots x_{n,m})) \\
&= h(\$^n(x_{1,1} \cdots x_{n,1})(x_{1,2} \cdots x_{n,2}) \cdots (x_{1,m} \cdots x_{n,m})) \\
&= h((\$^{n-1} x_{1,1})(x_{2,1} \cdots x_{n,1} x_{1,2}) \cdots (x_{2,m} \cdots x_{n,m}\$)) \\
&= h(I(\$x_{2,1} \cdots x_{2,m}, \$x_{3,1} \cdots x_{3,m}, \ldots, \$x_{n,1} \cdots x_{n,m}, x_{1,1} \cdots x_{1,m}\$))
\end{aligned}$$

and $\sqcup\!\sqcup_2^n(u_1, \ldots, u_n) = h(\{I(v_1, \ldots, v_n) \mid \exists m \forall i \in \{1, \ldots, n\} : v_i \in \$^* u_i \$^* \cap \Sigma^m\})$ with $h : (\Sigma \cup \{\$\})^* \to \Sigma^*$ as in Definition 11. $\square$

Remark 18. By the first two properties, all permutations of concatenations of arguments are in $\sqcup\!\sqcup_2^n$, but this shuffle variant itself is only invariant under a cyclic permutation of its arguments. Note that Example 12 shows that it is not invariant under arbitrary permutations of its arguments. For $n = 2$, where it equals the literal shuffle by Lemma 14, as interchanging is the only non-trivial permutation, which is a cyclic one, this product is commutative, as was noted in [2, 3]. But as shown above, this property only extends to cyclic permutation for more than two arguments.

With both iterated variants we can describe languages that are not context-free, as shown by the next proposition. Comparing Proposition 19 with Proposition 10, we see that these operations are more powerful than the iterated initial literal shuffle from [2, 3], in the sense that we can leave the family of regular languages for finite input languages.

Proposition 19. $(abc)^{\sqcup\!\sqcup_1,\circledast} = (abc)^{\sqcup\!\sqcup_2,\circledast} \cap a^*b^*c^* = \{a^m b^m c^m \mid m \geq 0\}$.

5 Closure Properties

We first show that, when adding the full trio operations, the iterated version of our shuffle variants are as powerful as the shuffle, or as powerful as the iterated variants of the binary versions of the initial literal and literal shuffle introduced in [2, 3] by Proposition 9. But before, and for establishing our closure properties, we state the next result.

Lemma 20. *Let $\mathcal{L}$ be a family of languages. If $\mathcal{L}$ is closed under intersection with regular languages, isomorphic mappings and (general) shuffle, then $\mathcal{L}$ is closed under each shuffle variant $\sqcup\!\sqcup_i^n$ for $i \in \{1, 2\}$.*

With the previous lemma, we can derive the next result.

Proposition 21. *Let $\mathcal{L}$ be a full trio. The following properties are equivalent:*

1. *$\mathcal{L}$ is closed under shuffle.*
2. *$\mathcal{L}$ is closed under $\sqcup\!\sqcup_i^n$ for some $i \in \{1, 2\}$ and $n \geq 2$.*

In a full trio, we can express the iterated shuffle with our iterated versions of the n-ary shuffle variants.

Proposition 22. *Let $L \subseteq \Sigma^*$ be a language, $\$ \notin \Sigma$, and $h : (\Sigma \cup \$)^* \to \Sigma^*$ the homomorphism defined by: $h(x) = x$ if $x \in \Sigma$, $h(\$) = \varepsilon$. Then, for $i \in \{1, 2\}$,*

$$L^{\sqcup\!\sqcup,*} = h(h^{-1}(L)^{\sqcup\!\sqcup_i,\circledast}).$$

It is well-known that the regular languages are closed under shuffle [5]. Also, the context-sensitive languages are closed under shuffle [17]. A full trio is closed under intersection if and only if it is closed under shuffle [9]. As the recursively enumerable languages are a full trio that is closed under intersection, this family of languages is also closed under shuffle. As the context-free languages are a full trio that is not closed under intersection [12], it is also not closed under shuffle. The last result is also implied by Proposition 19 and Proposition 21. The recursive languages are only closed under non-erasing homomorphisms, so we could not reason similarly. Nevertheless, this family of languages is closed under shuffle.

Proposition 23. *The family of recursive languages is closed under shuffle.*

We now state the closure properties of the families of regular, context-sensitive, recursive and recursively enumerable languages.

Proposition 24. *The families of regular, context-sensitive, recursive and recursively enumerable languages are closed under $\sqcup\!\sqcup_i^n$ for $i \in \{1, 2\}$. Furthermore, the families of context-sensitive, recursive and recursively enumerable languages are closed under the iterated versions, i.e., if L is context-sensitive, recursive or recursively enumerable, then $L^{\sqcup\!\sqcup_i,\circledast}$, $i \in \{1, 2\}$, is context-sensitive, recursive or recursively enumerable, respectively.*

Proof (sketch). The closure of all mentioned language families under $\sqcup\!\sqcup_i^n$ with $i \in \{1, 2\}$ is implied by Lemma 20, as they are all closed under intersection with regular languages and shuffle by Proposition 23 and the considerations before this statement. Now, we give a sketch for the iterated variant $L^{\sqcup\!\sqcup_1,\circledast}$.

Let $M = (\Sigma, \Gamma, Q, \delta, s_0, F)$ be a Turing machine for L. The following construction will work for all language classes. More specifically, if given a context-sensitive, recursive or recursively enumerable language L with an appropriate machine M, it could be modified to give a machine that describes a language in the corresponding class, but the basic idea is the same in all three cases. Recall that the context-sensitive languages could be characterized by linear bounded automata [12].

We construct a 3-tape Turing machine, with one input tape, that simulates M and has three working tapes. Intuitively,

1. the input tape stores the input u;
2. on the first working tape, the input is written in a decomposed way, and on certain parts, the machine M is simulated;
3. on the second working tape, for each simulation run of M, a state of M is saved;
4. the last working tape is used to guess and store a number $0 < n \leq |u|$.

We sketch the working of the machine. First, it non-deterministically guesses a number $0 < n \leq |u|$ and stores it on the last tape. Then, it parses the input u in several passes, each pass takes $0 < k \leq n$ symbols from the front of u and puts them in an ordered way on the second working tape, and, non-deterministically, decreases k or does not decrease k. More specifically, on the second working tape, the machine writes a word, with a special separation sign #,

$$\#u_1\#u_2\#\cdots\#u_n\#$$

where $\sqcup\!\sqcup_1^n(u_1, \ldots, u_n)$ equals the input parsed so far. When the input word is completey parsed, it simulates M to check if each word u_i on the second working tape is contained in L. □

Lastly, we can characterize the family of non-empty finite languages using $\sqcup\!\sqcup_1^n$.

Proposition 25. *The family of non-empty finite languages is the smallest family of languages $\mathcal{L}$ such that*

1. *$\{w\} \in \mathcal{L}$ for some word $w \neq \varepsilon$ with all symbols in w distinct, i.e., $w = a_1 \cdots a_m$ with $a_i \neq a_j$ for $1 \leq i \neq j \leq m$ and $a_i \in \Sigma$ for $i \in \{1, \ldots, m\}$,*
2. *closed under union,*
3. *closed under homomorphisms $h : \Sigma^* \to \Gamma^*$ such that $|h(x)| \leq 1$ for $x \in \Sigma$,*

4. closed under $\sqcup\!\sqcup_1^n$ for some $n \geq 2$.

And, without closure under any homomorphic mappings.

Proposition 26. *The family of non-empty finite languages is the smallest family of languages $\mathcal{L}$ such that (1) $\{\{\varepsilon\}\} \cup \{\{a\} \mid a \in \Sigma\} \subseteq \mathcal{L}$ and which is (2) closed under union and $\sqcup\!\sqcup_1^n$ for some $n \geq 2$.*

6 Computational Complexity

Here, we consider various decision problems for both shuffle variants motivated by similar problems for the ordinary shuffle operation [4, 25, 29, 31]. It could be noted that all problems considered are tractable when considered for $\sqcup\!\sqcup_1$. However, for $\sqcup\!\sqcup_2$, most problems considered are tractable except one that is NP-complete. Hence, the ability to vary the starting positions of different words when interlacing them consecutively, in an alternating fashion, seems to introduce computional hardness. For $\sqcup\!\sqcup_1$, we find the following:

Proposition 27. *Given $L \subseteq \Sigma^*$ represented by a non-deterministic*[1] *automaton and words $w_1, \ldots, w_n \in \Sigma^*$, it is decidable in polynomial time if $\sqcup\!\sqcup_1^n(w_1, \ldots, w_n) \in L$.*

Proposition 28. *Given words $w \in \Sigma^*$ and $v \in \Sigma^*$, it is decidable in polynomial time if $w \in \{v\}^{\sqcup\!\sqcup_1, \circledast}$.*

The non-uniform membership for a languages $L \subseteq \Sigma^*$ is the computational problem to decide for a given word $w \in \Sigma^*$ if $w \in L$. In [25] it was shown that the shuffle of two deterministic context-free languages can yield a language that has an NP-complete non-uniform membership problem. This result was improved in [4] by showing that there even exist linear deterministic context-free languages whose shuffle gives an intractable non-uniform membership problem.

Next, we show that for the initial literal and the literal shuffle, this could not happen if the original languages have a tractable membership problem, which is the case for context-free languages [12].

Proposition 29. *Let $U, V \subseteq \Sigma^*$ be languages whose membership problem is solvable in polynomial time. Then, also the membership problems for $U \sqcup\!\sqcup_1 V$ and $U \sqcup\!\sqcup_2 V$ are solvable in polynomial time.*

Proof. Let w be a given word and write $w = w_1 \cdots w_n$ with $w_i \in \Sigma$ for $i \in \{1, \ldots, n\}$.

Then, to check if $w \in U \sqcup\!\sqcup_2 V$, we try all decompositions $w = xyz$ with $x, y, z \in \Sigma^*$ and $|y|$ even. For $w = xyz$, write $y = y_1 \cdots y_{2n}$ with $y_i \in \Sigma$ and $n \geq 0$. Then test if $xy_1y_3 \cdots y_{2n-1} \in U$ and $y_2 \cdots y_{2n}z \in V$, or $y_1y_3 \cdots y_{2n-1}z \in U$ and $xy_2 \cdots y_{2n} \in V$, or $xy_1y_3 \cdots y_{2n-1}z \in U$ and $y_2 \cdots y_{2n} \in V$, or $y_1y_3 \cdots y_{2n-1} \in U$ and $xy_2 \cdots y_{2n}z \in V$ As $U \sqcup\!\sqcup_2 V = V \sqcup\!\sqcup_2 U$ this is sufficient to find out if $w \in U \sqcup\!\sqcup_2 V$.

For $U \sqcup\!\sqcup_1 V$, first check if $w \in U$ and $\varepsilon \in V$, or if $\varepsilon \in U$ and $w \in V$. If neither of the previous checks give a YES-answer, then try all decompositions $w = yz$ with $y = y_1 \cdots y_{2n}$ for $y_i \in \Sigma$ and $n > 0$. Then, test if $y_1y_3 \cdots y_{2n-1}z \in U$ and $y_2 \cdots y_{2n} \in V$, or if $y_1y_3 \cdots y_{2n-1}z \in V$ and $y_2 \cdots y_{2n} \in U$. If at least one of these tests gives a YES-answer, we have $w \in U \sqcup\!\sqcup_1 V$, otherwise $w \notin U \sqcup\!\sqcup_1 V$.

In all cases, only polynomially many tests were necessary. □

[1] In a non-deterministic automaton the transitions are represented by a relation instead of a function, see [12].

A similar procedure could be given for any fixed number n and $L_1, \ldots, L_n \subseteq \Sigma^*$ to decide the membership problem for $\sqcup\!\sqcup_i^n(L_1, \ldots, L_n)$, $i \in \{1, 2\}$ in polynomial time.

Lastly, the following is an intractable problem for the second shuffle variant.

Proposition 30. *Suppose $|\Sigma| \geq 3$. Given a finite language $L \subseteq \Sigma^*$ represented by a deterministic automaton and words $w_1, \ldots, w_n \in \Sigma^*$, it is* NP*-complete to decide if $\sqcup\!\sqcup_2^n(w_1, \ldots, w_n) \cap L \neq \emptyset$.*

Proof (sketch). We give the basic idea for the hardness proof. Similarly as in [31] for the corresponding problem in case of the ordinary shuffle and a single word $L = \{w\}$ as input, we can use a reduction from 3-Partition. This problem is known to be strongly NP-complete, i.e., it is NP-complete even when the input numbers are encoded in unary [8].

3-Partition
Input: A sequence of natural numbers $S = \{n_1, \ldots, n_{3m}\}$ such that $B = (\sum_{i=1}^{3m} n_i)/m \in \mathbb{N}_0$ and for each i, $1 \leq i \leq 3m$, $B/4 < n_i < B/2$.
Question: Can S be partitioned into m disjoint subsequences $S_1, \ldots, S_m$ such that for each k, $1 \leq k \leq m$, S_k has exactly three elements and $\sum_{n \in S_k} n = B$.

Let $S = \{n_1, \ldots, n_{3m}\}$ be an instance of 3-Partition. Set

$$L = \{aaauc \in \{a, b, c\}^* \mid |u|_b = B, |u|_a = 0, |u|_c = 2\}^m.$$

We can construct a deterministic automaton for L in polynomial time. Then, the given instance of 3-Partition has a solution if and only if

$$L \cap \sqcup\!\sqcup_2^n(ab^{n_1}c, ab^{n_2}c, \ldots, ab^{n_{3m}}c) \neq \emptyset. \quad \square$$

Lastly, as the constructions in the proof of Lemma 20 are all effective, and the inclusion problem for regular languages is decidable [12], we can decide if a given regular language is preserved under any of the shuffle variants.

As the inclusion problem is undecidable even for context-free languages [12], we cannot derive an analogous result for the other families of languages in the same way.

Proposition 31. *For every regular language $L \subseteq \Sigma^*$ and $i \in \{1, 2\}$, we can decide whether L is closed under $\sqcup\!\sqcup_i^n$, i.e, if $\sqcup\!\sqcup_i^n(L, \ldots, L) \subseteq L$ holds.*

7 Permuting Arguments

If we permute the arguments of the first shuffle variant $\sqcup\!\sqcup_1^n$ we may get different results. Also, for the second variant, see Proposition 2, only permuting the arguments cyclically does not change the result, but permuting the arguments arbitrarily might change the result, see Example 12.

Here, we introduce two variants of $\sqcup\!\sqcup_1$ that are indifferent to the order of the arguments, i.e., permuting the arguments does not change the result, by considering all possibilities in which the strings could be interlaced. A similar definition is possible for the second variant.

Definition 32 (n-ary symmetric initial literal shuffle). *Let $u_1, \ldots, u_n \in \Sigma^*$ and $x_1, \ldots, x_n \in \Sigma$. Then the function $\sqcup\!\sqcup_3^n : (\Sigma^*)^n \to \mathcal{P}(\Sigma^*)$ is given by*

$$\sqcup\!\sqcup_3^n(u_1, \ldots, u_n) = \bigcup_{\pi \in \mathcal{S}_n} \{\sqcup\!\sqcup_1^n(u_{\pi(1)}, \ldots, u_{\pi(n)})\}.$$

An even stronger form as the previous definition do we get, if we do not care in what order we put the letters at each step.

Definition 33 (n-ary non-ordered initial literal shuffle). *Let $u_1, \ldots, u_n \in \Sigma^*$ and $x_1, \ldots, x_n \in \Sigma$. Then define*

$$\begin{aligned} ш_4^n(x_1u_1, \ldots, x_nu_n) &= \bigcup_{\pi \in \mathcal{S}_n} x_{\pi(1)} \cdots x_{\pi(n)} \, ш_4^n \,(u_1, \ldots, u_n) \\ ш_4^n(u_1, \ldots, u_{j-1}, \varepsilon, u_{j+1}, \ldots, u_n) &= ш_4^{n-1}(u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_n) \\ ш_4^1(u_1) &= \{u_1\}. \end{aligned}$$

Similarly as in Definition 16 we can define iterated versions $L^{ш_3, \circledast}$ and $L^{ш_4, \circledast}$ for $L \subseteq \Sigma^*$. The following properties follow readily.

Proposition 34. *Let $L_1, \ldots, L_n \subseteq \Sigma^*$, $\pi \in \mathcal{S}_n$ and $i \in \{3, 4\}$. Then*

1. $ш_3^n(L_1, \ldots, L_n) = ш_3^n(L_{\pi(1)}, \ldots, L_{\pi(n)})$;
2. $ш_4^n(L_1, \ldots, L_n) = ш_4^n(L_{\pi(1)}, \ldots, L_{\pi(n)})$;
3. $\{ш_1^n(L_1, \ldots, L_n)\} \subseteq ш_3^n(L_1, \ldots, L_n)\} \subseteq ш_4^n(L_1, \ldots, L_n)$;
4. $ш_i^n(L_1, \ldots, L_n) \subseteq L_1 ш \cdots ш L_n$;
5. $\Sigma^* = \Sigma^{ш_i, \circledast}$ *for* $i \in \{3, 4\}$;
6. $L_1^{ш_1, \circledast} \subseteq L_1^{ш_3, \circledast} \subseteq L_1^{ш_4, \circledast} \subseteq L_1^{ш, *}$;
7. *for* $u_1, \ldots, u_n, u \in \Sigma^*$, *if* $u \in ш_i^n(\{u_1\}, \ldots, \{u_n\})$, *then* $|u| = |u_1| + \cdots + |u_n|$.

With these properties, we find that for the iteration the first and third shuffle variants give the same language operator.

Lemma 35. *For languages $L \subseteq \Sigma^*$ we have $ш_1^n(L, \ldots, L) = ш_3^n(L, \ldots, L)$.*

For the iterated version, this gives that the first and third variant are equal.

Corollary 36. *Let $L \subseteq \Sigma^*$ be a language. Then $L^{ш_1, \circledast} = L^{ш_3, \circledast}$.*

Hence, with Proposition 19, we can deduce that $(abc)^{ш_3, \circledast} = (abc)^{ш_4, \circledast} \cap a^*b^*c^* = \{a^m b^m c^m \mid m \geq 0\}$ and so even for finite languages, the iterated shuffles yield languages that are not context-free.

We find that Lemma 20, Proposition 21, Proposition 22 and Proposition 24 also hold for the third and fourth shuffle variant. To summarize:

Proposition 37. *Let $i \in \{3, 4\}$. Then:*

1. *If $\mathcal{L}$ is a family of languages closed under intersection with regular languages, isomorphic mappings and (general) shuffle, then $\mathcal{L}$ is closed under $ш_i^n$ for $i \in \{3, 4\}$ and each $n \geq 1$.*
2. *If $\mathcal{L}$ is a full trio, then $\mathcal{L}$ is closed under shuffle if and only if it is closed under $ш_i^n$.*
3. *For regular $L \subseteq \Sigma^*$, it is decidable if $ш_i^n(L, \ldots, L) \subseteq L$.*
4. *For $L \subseteq \Sigma^*$ and the homomorphism $h : (\Sigma \cup \$)^* \to \Sigma^*$ given by $h(x) = x$ for $x \in \Sigma$ and $h(\$) = \varepsilon$, we have $L^{ш, *} = h(h^{-1}(L)^{ш_i, \circledast})$.*
5. *The families of regular, context-sensitive, recursive and recursively enumerable languages are closed under $ш_i^n$ and the families of context-sensitive, recursive and recursively enumerable languages are closed for $L^{ш_i, \circledast}$.*

Lastly, we give two examples.

Example 38. Set $L = \{ab, ba\}$. Define the homomorphism $g : \{a,b\}^* \to \{a,b\}^*$ by $g(a) = b$ and $g(b) = a$, i.e., interchanging the symbols.

1. $L^{ш_1,\circledast} = L^{ш_3,\circledast} = \{ug(u) \mid u \in \{a,b\}^*\}$.

   Proof. Let $u_1, \ldots, u_n \in L$ and $w = ш_1^n(u_1, \ldots, u_n)$. Then $w = I(u_1, \ldots, u_n)$. As $|u_1| = \ldots = |u_n| = 2$, we can write $w = x_1 \cdots x_{2n}$ with $x_i \in \{a,b\}$ for $i \in \{1, \ldots, 2n\}$. By Definition 3 of the I-operator, we have, for $i \in \{1, \ldots, n\}$, $u_i = x_i x_{i+n}$. So, if $u_i = ab$, then $x_i = a$ and $x_{i+n} = b$, and if $u_i = ba$, then $x_i = b$ and $x_{i+n} = a$. By Corollary 36, $L^{ш_1,\circledast} = L^{ш_3,\circledast}$.

2. $L^{ш_4,\circledast} = \{uv \mid u, v \in \{a,b\}^*, |u|_a = |v|_b, |u|_b = |v|_a\}$.

   Proof. Let $u_1, \ldots, u_n \in L$ and $w = ш_4^n(u_1, \ldots, u_n)$. Then, using the inductive Definition 33 twice, we find $w = uv$, where u contains all the first symbols of the arguments $u_1, \ldots, u_n$ in some order, and v all the second symbols. Hence, for each a in u, we must have a b in v and vice versa. So, w is contained in the set on the right hand side. Conversely, suppose $w = uv$ with $|u|_a = |v|_b$ and $|u|_b = |v|_a$. Then set $n = |u| = |v|$, $u = x_1 \cdots x_n$, $v = y_1 \cdots y_n$ with $x_i, y_i \in \Sigma$ for $i \in \{1, \ldots, n\}$. We can reorder the letters to match up, i.e., an a with a b and vice versa. More specifically, we find a permutation $\pi \in \mathcal{S}_n$ such that $w = I(x_{\pi(1)}y_1, \ldots, x_{\pi(n)}y_n) \in ш_4^n(x_1y_1, \ldots, x_ny_n) \in L^{ш_4,\circledast}$.

8 Conclusion and Summary

The literal and the initial literal shuffle were introduced with the idea to describe the execution histories of step-wise synchronized processes. However, a closer investigation revealed that they only achieve this for two processes, and, mathematically, the lack of associativity of these operations prevented usage for more than two processes. Also, iterated variants derived from this non-associative binary operations depend on a fixed bracketing and does not reflect the synchronization of n processes. The author of the original papers [2, 3] does not discuss this issue, but is more concerned with the formal properties themselves. Here, we have introduced two operations that lift the binary variant to an arbitrary number of arguments, hence allowing simultaneous step-wise synchronization of an arbitrary number of processes. We have also introduced iterative variants, which are more natural than the previous ones for the non-associative binary operations. In summary, we have

1. investigated the formal properties and relations between our shuffle variant operations,
2. we have found out that some properties are preserved in analogy to the binary case, but others are not, for example commutativity, see Proposition 17;
3. we have shown various closure or non-closure properties for the family of languages from the Chomsky hierarchy and the recursive languages;
4. used one shuffle variant to characterize the family of finite languages;
5. in case of a full trio, we have shown that their expressive power coincides with the general shuffle, and, by results from [2, 3], with the initial literal and literal shuffle;
6. we have investigated various decision problems, some of them are tractable even if an analogous decision problem with the general shuffle operation is intractable. However, we have also identified an intractable decision problem for our second n-ary shuffle variant.

As seen in Proposition 30 for the NP-complete decision problem, we needed an alphabet of size at least three. For an alphabet of size one, i.e., an unary alphabet, the n-ary shuffle for any of the variants considered reduces to the (n-times) concatenation of the arguments, which is easily computable. Then, deciding if the result of this concatenation is contained in a given regular language could be done in polynomial time. So, a natural question is if the problem formulated in Proposition 30 remains NP-complete for binary alphabets only. Also, it is unknown what happens if we alter the problem by not allowing a finite language represented by a deterministic automaton as input, but only a single word, i.e., $|L| = 1$. For the general shuffle, this problem is NP-complete, but it is open what happens if we use the second n-ary variant. Also, it is unknown if the problem remains NP-complete if we represent the input not by an automaton, but by a finite list of words.

Acknowledgement
I thank anonymous reviewers of a previous version for feedback and remarks that helped to improve the presentation. I also thank the reviewers of the current version for careful reading and pointing out typos and some unclear formulations. Due to the strict page limit, I cannot put all proofs into the paper. I have supplied proof sketches where possible. Also, an extended version with all the missing proofs is in preparation.

Bibliography

[1] T. Araki and N. Tokura: *Flow languages equal recursively enumerable languages.* Acta Informatica, 15 1981, pp. 209–217.

[2] B. Bérard: *Formal properties of literal shuffle.* Acta Cyb., 8(1) 1987, pp. 27–39.

[3] B. Bérard: *Literal shuffle.* Theor. Comput. Sci., 51 1987, pp. 281–299.

[4] M. Berglund, H. Björklund, and J. Björklund: *Shuffled languages - representation and recognition.* Theor. Comput. Sci., 489-490 2013, pp. 1–20.

[5] J. A. Brzozowski, G. Jirásková, B. Liu, A. Rajasekaran, and M. Szykuła: *On the state complexity of the shuffle of regular languages*, in Descrip. Compl. of Formal Systems - 18th IFIP WG 1.2 International Conference, DCFS 2016, Bucharest, Romania, July 5-8, 2016. Proceedings, C. Câmpeanu, F. Manea, and J. Shallit, eds., vol. 9777 of Lecture Notes in Computer Science, Springer, 2016, pp. 73–86.

[6] S. Buss and M. Soltys: *Unshuffling a square is NP-hard.* J. Comput. Syst. Sci., 80(4) 2014, pp. 766–776.

[7] R. H. Campbell and A. N. Habermann: *The specification of process synchronization by path expressions*, in Operating Systems OS, E. Gelenbe and C. Kaiser, eds., vol. 16 of LNCS, Springer, 1974, pp. 89–102.

[8] M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, W. H. Freeman, first edition ed., 1979.

[9] S. Ginsburg: *Algebraic and Automata-Theoretic Properties of Formal Languages*, Elsevier Science Inc., USA, 1975.

[10] S. Ginsburg and S. Greibach: *Abstract families of languages*, in 8th Annual Symposium on Switching and Automata Theory (SWAT 1967), 1967, pp. 128–139.

[11] D. Henshall, N. Rampersad, and J. O. Shallit: *Shuffling and unshuffling.* Bull. EATCS, 107 2012, pp. 131–142.

[12] J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979.

[13] K. Iwama: *Unique decomposability of shuffled strings: A formal treatment of asynchronous time-multiplexed communication*, in Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC '83, New York, NY, USA, 1983, Association for Computing Machinery, p. 374–381.

[14] K. Iwama: *The universe problem for unrestricted flow languages.* Acta Informatica, 19(1) Apr. 1983, pp. 85–96.

[15] M. Jantzen: *The power of synchronizing operations on strings.* Theor. Comput. Sci., 14 1981, pp. 127–154.

[16] M. Jantzen: *Extending regular expressions with iterated shuffle.* Theor. Comput. Sci., 38 1985, pp. 223–247.

[17] J. Jedrzejowicz: *On the enlargement of the class of regular languages by the shuffle closure.* Inf. Process. Lett., 16(2) 1983, pp. 51–54.

[18] J. Jedrzejowicz and A. Szepietowski: *Shuffle languages are in P.* Theor. Comput. Sci., 250(1-2) 2001, pp. 31–53.

[19] M. Kudlek and N. E. Flick: *Properties of languages with catenation and shuffle.* Fundam. Inform., 129(1-2) 2014, pp. 117–132.

[20] M. Latteux: *Cônes rationnels commutatifs.* J. Comp. Sy. Sc., 18(3) 1979, pp. 307–333.

[21] M. Latteux: *Behaviors of processes and synchronized systems of processes*, in Theoretical Foundations of Programming Methodology, S. G. Broy M., ed., vol. 91 of NATO Advanced Study Institutes Series (Series C — Mathematical and Physical Sciences), Springer, Dordrecht, 1982, pp. 473–551.

[22] A. Mateescu, G. Rozenberg, and A. Salomaa: *Shuffle on trajectories: Syntactic constraints.* Theor. Comput. Sci., 197(1-2) 1998, pp. 1–56.

[23] A. W. Mazurkiewicz: *Parallel recursive program schemes*, in Mathematical Foundations of Computer Science 1975, 4th Symposium, Mariánské Lázne, Czechoslovakia, September 1-5, 1975, Proceedings, J. Becvár, ed., vol. 32 of Lecture Notes in Computer Science, Springer, 1975, pp. 75–87.

[24] M. Nivat: *Transductions des langages de chomsky.* Annales de l'Institut Fourier, 18(1) 1968, pp. 339–455.

[25] W. F. Ogden, W. E. Riddle, and W. C. Round: *Complexity of expressions allowing concurrency*, in Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '78, New York, NY, USA, 1978, Association for Computing Machinery, p. 185–194.

[26] W. E. Riddle: *An approach to software system behavior description.* Comput. Lang., 4(1) 1979, pp. 29–47.

[27] W. E. Riddle: *An approach to software system modelling and analysis.* Comput. Lang., 4(1) 1979, pp. 49–66.

[28] A. C. Shaw: *Software descriptions with flow expressions.* IEEE Trans. Softw. Eng., 4 1978, pp. 242–254.

[29] L. J. Stockmeyer and A. R. Meyer: *Word problems requiring exponential time (preliminary report)*, in Proceedings of the fifth annual ACM Symposium on Theory of Computing, STOC, ACM, 1973, pp. 1–9.

[30] M. H. ter Beek, C. Martín-Vide, and V. Mitrana: *Synchronized shuffles.* Theor. Comput. Sci., 341(1-3) 2005, pp. 263–275.

[31] M. K. Warmuth and D. Haussler: *On the complexity of iterated shuffle.* J. Comput. Syst. Sci., 28(3) 1984, pp. 345–358.

Towards an Efficient Text Sampling Approach for Exact and Approximate Matching

Simone Faro[1], Francesco Pio Marino[1], Arianna Pavone[2], and Antonio Scardace[1]

[1] Dipartimento di Matematica e Informatica,
Università di Catania, viale A. Doria n. 6, 95125, Catania, Italia
[2] Dipartimento di Scienze Cognitive,
Università di Messina, via Concezione n.6, 98122, Messina, Italia

Abstract. Text-sampling is an efficient approach for the string matching problem recently introduced in order to overcome the prohibitive space requirements of indexed matching, on the one hand, and drastically reduce searching time for the online solutions, on the other hand. Known solutions to sampled string matching are very efficient in practical cases being able to improve standard online string matching algorithms up to 99.6% using less than 1% of the original text size. However at present text sampling is designed to work only in the case of exact string matching.
In this paper we present some preliminary results obtained in the attempt to extend sampled-string matching to the general case of approximate string matching. Specifically we introduce a new sampling approach which turns out to be suitable for both exact and approximate matching and evaluate it in the context of a specific case of approximate matching, the order preserving pattern matching problem.
Our preliminary experimental results show that the new approach is extremely competitive both in terms of space and running time, and for both approximate and exact matching. We also discuss the applicability of the new approach to different approximate string matching problems.

1 Introduction

String matching, in both its *exact* and *approximate* form, is a fundamental problem in computer science and in the wide domain of text processing. It consists in finding all the occurrences of a given pattern x, of length m, in a large text y, of length n, where both sequences are composed by characters drawn from the same alphabet Σ.

As the size of data increases the space needed to store it is constantly increasing too, for this reasons the need for new efficient approaches to the problem capable of significantly improving the performance of existing algorithms by limiting the space used to achieve this as much as possible.

In this paper it is assumed that the text is a sequence of elements taken from a set on which a relation of total order is defined. In general we will try to simplify the discussion by assuming that the text is a sequence of numbers. Such a situation can be assumed for many practical applications since even a character can often be interpreted as a number.

Applications require two kinds of solutions: *online* and *offline* string matching. Solutions based on the first approach assume that the text is not pre-processed and thus they need to scan the input sequence *online*, when searching. Their worst case time complexity is $\Theta(n)$, and was achieved for the first time by the well known Knuth-Morris-Pratt (KMP) algorithm [19], while the optimal average time complexity of the problem is $\Theta(n \log_\sigma m/m)$ [24], achieved for example by the Backward-Dawg-Matching algorithm [9]. Many string matching solutions have been also developed in order to obtain sub-linear performance in practice [11]. Among them the Boyer-Moore-Horspool algorithm [3,17] deserves a special mention, since it has inspired much work. Memory requirements of this class of algorithms are very low and generally limited to a precomputed table of size $O(m\sigma)$ or $O(\sigma^2)$ [11]. However their searching time is always proportional to the length of the text and thus their performances may stay poor in many practical cases, especially for huge texts and short patterns.

Solutions based on the second approach try to drastically speed up searching by preprocessing the text and building a data structure that allows searching in time proportional to the length of the pattern. This method is called *indexed searching* [20,16]. However, despite their optimal time performances, space requirements of such data structures are from 4 to 20 times the size of the text, which may be too large for many practical applications.

Leaving aside other different approaches, like those based on *compressed string matching* [21,4], an effective alternative solution to the problem is *sampled string matching*, introduced in 1991 by Vishkin [23]. It consists in the construction of a succinct sampled version of the text (which must be maintained together with the original text) and in the application of an online searching procedure directly on the sampled sequence which acts as a filter method in order to limit the search only on a limited set of candidate occurrences. Although any candidate occurrence of the pattern may be found more efficiently, the drawback of this approach is that any occurrence reported in the sampled-text requires to be verified in the original text. Apart from this point a sampled-text approach may have a lot of good features: it may be easy to implement if compared with other succint matching approaches, it may require very small extra space and may allow fast searching. Additionally it may also allow fast updates of the data structure.

The first practical solution to sampled string matching has been introduced by Claude *et al.* [8] and is based on an alphabet reduction. Their solution has an extra space requirement which is only 14% of text size and turns out to be up to 5 times faster than standard online string matching on English texts. In this paper we refer to this algorithm as Occurrence Text Sampling (OTS).

More recently Faro *et al.* presented a more effective sampling approach based on *character distance sampling* [14,13] (CDS), obtaining in practice a speed up by a factor of up to 9 on English texts, using limited additional space whose amount goes from 11% to 2.8% of the text size, with a gain in searching time up to 50% if compared against the previous solution.

1.1 Our Results and organization of the paper

Known solutions to sampled-string matching prove to work efficiently only in the case of natural language texts or, in general, when searching on input sequences over large alphabets, while their performances degrade when the size of the underlying alphabets decreases. In addition they have been designed to work for solving the exact string matching problem, being inflexible in case they have to be applied to approximate string matching problems.

In this paper we present a new text sampling technique, called Monotonic Run Length Scaling, based on the length of the monotonic sub-sequences formed by the characters of the text when the latter is made up of elements of a finite and totally ordered alphabet. The new technique is original and turns out to be very flexible for its application in both exact and approximate matching. Specifically we also present a preliminary evaluation of the technique in the case of exact string matching (ESM) and order preserving pattern matching (OPPM), as a case study for the approximate pattern matching.

In the second part of the paper we improve the new approach in practice by proposing a further technique called Monotonic Run Length Sampling and based on the sampling of the lengths of the monotonic sequences in the input text.

From our experimental results it turns out that the new approach, although still in a preliminary phase of formalization and in an early implementation stage, is particularly efficient and flexible in its application, obtaining results that improve up to 12 times standard solutions for the exact string matching problem and up to 40 times known solutions for order preserving pattern matching problem.

The paper is organized as follows. In Section 2 we introduce the Monotonic Run Length Scaling while in Section 3 we introduce the Monotonic Run Length Sampling. In both cases we present a first naïve algorithm for searching a text using the new proposed partial indexes. Then we present our preliminary experimental evaluation in Section 4 testing our proposed sampling approach in terms of space consumption and running times for both exact strung matching and order preserving pattern matching. Finally we draw our conclusions and discuss some future works in Section 5.

2 Monotonic Run Length Scaling

Definition 1 (Monotonic Run). *Let y be a text of length n over a finite and totally ordered alphabet Σ of size σ. A Monotonic Increasing Run of y is a not extendable sub-sequence w of y whose elements are arranged in increasing order. Formally, if $w = y[i..i+k-1]$ is a monotonic increasing run of y of length k, we have:*

- $w[j-1] < w[j]$, *for each* $0 < j < k$;
- $y[i] \leq y[i-1]$;
- $y[i+k-1] \geq y[i+k]$.

Symmetrically a Monotonic Non-Increasing Run of y is a not extendable sub-sequence w of y whose elements are arranged in non-increasing order. Formally, if $w = y[i..i+k-1]$ is a monotonic non-increasing run of y of length k, we have:

- $w[j-1] \geq w[j]$, *for each* $0 < j < k$;
- $y[i] > y[i-1]$;
- $y[i+k-1] < y[i+k]$.

We will indicate with the general term Monotonic Run any sub-sequence of y that can be both monotonic increasing and monotonic non-increasing.

By definition two adjacent monotonic sub-sequences of a string have a single overlapping character, i.e. the rightmost character of the first sub-sequence is also the leftmost character of the second sub-sequence.

Example 2. Let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15. We can identify the following monotonic runs in y: $y[0..2] = \langle 4, 5, 11\rangle$ is a monotonic increasing run; $y[2..5] = \langle 11, 7, 6, 6\rangle$ is a monotonic non-increasing run; $y[5..6] = \langle 6, 12\rangle$ is a monotonic increasing run; $y[6..8] = \langle 12, 12, 2\rangle$ is a monotonic non-increasing run; $y[8..9] = \langle 2, 9\rangle$ is a monotonic increasing run; $y[9..11] = \langle 9, 8, 6\rangle$ is a monotonic non-increasing run; finally, $y[11..14] = \langle 6, 7, 10, 13\rangle$ is a monotonic increasing run.

```
Function 1: Monotonic-Run-Length-Scaling($x, m$)
Data: a string $x$ of length $m$
Result: The Scaled version $\widetilde{x}$ of $x$
$\widetilde{x} \longleftarrow \langle\rangle$;
$\mu \longleftarrow 1$;
$d \longleftarrow 1$;
if $(x[1] - x[0] \leq 0)$ then $d \longleftarrow 1$ ;
$i \longleftarrow 2$;
while $(i < n)$ do
    if $((d = 1$ and $x[i] - x[i-1] > 0)$ or $(d = -1$ and $x[i] - x[i-1] \leq 0))$ then
        $i \longleftarrow i + 1$;
        $\mu \longleftarrow \mu + 1$;
    else
        $\widetilde{x} \longleftarrow \widetilde{x} + \langle\mu\rangle$;
        $\mu \longleftarrow 0$;
        $d \longleftarrow d \times -1$;
    end
end
$\widetilde{x} \longleftarrow \widetilde{x} + \langle\mu\rangle$;
return $\widetilde{x}$
```

We now define the process of *Monotonic Run Length Scaling* (MRLX) of a string y, which consists in decomposing the string in a set of adjacent monotonic runs. The resulting sequence, which we call *monotonic run length scaled version* of y, is the numeric sequence of the lengths of the monotonic runs given by the MRLX process.

Definition 3 (Monotonic Run Length Scaling). *Let y be a text of length n over a finite and totally ordered alphabet Σ of size σ. Let $\langle\rho_0, \rho_1, \ldots, \rho_{k-1}\rangle$ the sequence of adjacent monotonic runs of y such that ρ_0 starts at position 0 of y and ρ_{k-1} ends at position $n-1$ of y. The monotonic run length scaled version of y, indicated by $\widetilde{y}$, is a numeric sequence, defined as $\widetilde{y} = \langle|\rho_0|, |\rho_1|, .., |\rho_{k-1}|\rangle$.*

It is straightforward to prove that there exists only a unique monotonic run length scaled version of a given string y. In addition we observe that

$$\left[\sum_{i=0}^{k-1} |\rho_i|\right] - k + 1 = n$$

Example 4. As in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15. Then the scaled version of y is the sequence $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4\rangle$ of length 7. We have therefore that $3+4+2+3+2+3+4-7+1 = 15$.

Function 1 depicts the pseudo-code of the algorithm which computes the MRL scaled version of an input string x of length m. It constructs the sequence $\widetilde{x}$ incrementally by scanning the input string x character by character, proceeding from left to right. It is straightforward to prove that its time complexity is $\mathcal{O}(m)$.

Figure 1 shows the average and maximal length of a monotonic run on a random text over an alphabet of size 2^δ, with $2 \leq \delta \leq 8$, and where the ordinates show the length values while the ordinates show the values of δ.

It is easy to observe that the length of each monotonic run (whose value is in any case bounded at the top by the size of the alphabet) never exceeds ten characters.

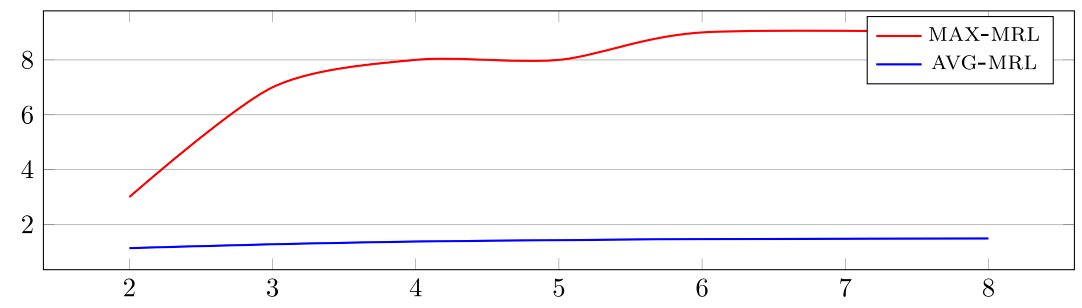


Figure 1. Average and maximal length of a monotonic run on a random text over an alphabet of size 2^δ. The ordinates show the length values while the ordinates show the values of δ.

Furthermore, we observe that the average length of the monotonic sub-sequences is much lower being always between 1.10 (for small alphabets) and 1.50 (for large alphabets). This implies that the average length of the monotonic run length scaled version of the text is on average between 66% and 90% of the length of the original sequence, a result not particularly exciting when compared with those obtained by previous sampling techniques such as CDS and OTS.

However, the fact that the length of each monotonic sub-sequence is up to 10 allows us to represent the monotonic run length scaled version of the text using only 4 bits for each element of the sequence, instead of the 8 bits needed in the OTS representation and the 32 bits needed in the CDS representation. Thus the average memory consumption required by the monotonic run length scaled version of the text is on average between 33% and 45% of the memory needed to store the original sequence.

2.1 Searching Using Monotonic Run Length Scaling

In this section we discuss the use of monotonic run length scaling as a sampling approach with application to exact and approximate string matching. In this preliminary work, for the case of approximate string matching, we take the Order Preserving Pattern Matching problem [18,6,7,2,10] as an case study, leaving section 5 with a broader discussion on the applicability of this method to other non-standard string matching problems.

Specifically, we present a naïve searching procedure designed to use the monotonic scaled version of the text as a partial index in order to speed up the search for the occurrences of a given pattern within the original text. Like any other solution based on text-sampling, the solution proposed in this section requires that the partial index is used in a preliminary filter phase and that each candidate occurrence identified in this first phase is then verified within the original text using a simple verification procedure.

The pseudo-code of the naïve searching procedure is depicted in Algorithm 1. The preprocessing phase of the algorithm consists in computing the monotonic run length scaled version $\widetilde{x}$ of the input pattern x. Let k be the length of the sequence $\widetilde{y}$ and let h be the length of the sequence $\widetilde{x}$. In addition let d be the length of the first monotonic run of x, i.e. $d = \widetilde{x}[0]$.

During the searching phase the algorithm naively searches for all occurrences of the sub-sequence $\widetilde{x}[1..h-2]$ along the scaled version of the text. We discard the first

Algorithm 1: Naïve algorithm based on Monotonic Run Length Scaling

```
Data: a pattern $x$ of length $m$, a text $y$ of length $n$ and its scaled version $\widetilde{y}$ of length $k$
Result: all positions $0 \leq i < n$ such that $x$ occurs in $y$ starting at position $i$
$\widetilde{x} \longleftarrow$ Monotonic-Run-Length-Scaling($x, m$);
$h \longleftarrow |\widetilde{x}|$;
$d \longleftarrow \widetilde{x}[0]$;
$r \longleftarrow \widetilde{y}[0]$;
for $s \longleftarrow 1$ to $k - h$ do
    $j \longleftarrow 1$;
    while ($j < h - 1$ and $\widetilde{x}[j] = \widetilde{y}[s + j]$) do $j \longleftarrow j + 1$;
    if ($j = h - 1$) and then
        if Verify($y, n, x, m, r - d$) then Output($r - d$);
    end
    $r \longleftarrow r + \widetilde{y}[s] - 1$;
end
```

and the last element of $\widetilde{x}$ since they are allowed to match any any element in the text which is greater than or equal.

The main **for** loop of the algorithm iterates over a shift value s, which is initialized to 1 at the first iteration and is incremented by one when passing from one iteration to the next. During each iteration the current window of the text $\widetilde{y}[s..s + h - 2]$ is attempted for a candidate occurrence. An additional variable r is maintained, representing the shift position in the original text y corresponding to the current window $\widetilde{y}[s..s + h - 2]$. Thus at the beginning of each iteration of the main **for** loop the following invariant holds: $r = \widetilde{y}[0] + \sum_{i=1}^{s-1}(\widetilde{y}[i]) - 1)$.

At each iteration the window of the text $\widetilde{y}[s..s + h - 2]$ is compared, character by character, against the sub-sequence $\widetilde{x}[1..h - 2]$, proceeding form left to right. If a match is found then a candidate occurrence of the pattern is located at position $r - d$ of the original text and a verification phase is called in order to check if the sub-string $y[r - d...r - d + m - 1]$ corresponds to a full occurrence of the pattern x. In all cases at the end of each attempt the value of r is increased by $\widetilde{y}[s]$. and the value of the shift s is increased by one position.

Example 5. As in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15 (the text) and let $x = \langle 12, 2, 9, 8, 6, 7, 10\rangle$ be a pattern of length 7. Then we have $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4\rangle$ and $\widetilde{x} = \langle 2, 2, 3, 3\rangle$, with $d = 2$.

The algorithm naively searches the text $\widetilde{y}$ for any occurrence of the sub-sequence $\widetilde{x}[1..2] = \langle 2, 3\rangle$, finding two candidate occurrences at position 2 and 4, respectively.

We obtain that:

– at the beginning of the third iteration of the **for** main loop, with $s = 2$, we have $r = 3 + 4 - 1 = 6$. Thus the verification procedure is run to compare x against the sub-sequence at position $r - d = 4$ in the original text, i.e. $\widetilde{y}[4..10] = \langle 6, 6, 12, 12, 2, 9, 8\rangle$. Unfortunately at position 6 the verification procedure would find neither an exact match nor an order preserving match.
– at the beginning of the fifth iteration of the **for** main loop, with $s = 4$, we have $r = 3 + 4 + 2 + 3 - 3 = 8$. Thus the verification procedure is run to compare x against the sub-sequence at position $r - d = 6$ in the original text, i.e. $\widetilde{y}[6..12] = \langle 12, 12, 2, 9, 8, 6, 7, 10\rangle$. Thus at position 6 the verification phase would find both an exact match and an order preserving match.

Function 2: Monotonic-Run-Length-Sampling($\widetilde{x}, h, q$)

```
Data: The Monotonic Run Length Scaled version $\widetilde{x}$ of the string $x$, with length $h$
Result: The Monotonic Run Length Sampled version $\widetilde{x}^q$ of $x$
$\widetilde{x}^q \longleftarrow \langle\rangle$;
$r \longleftarrow 0$;
for $i \longleftarrow 0$ to $h-1$ do
    if $(\widetilde{x}[i] = q)$ then $\widetilde{x}^q \longleftarrow \widetilde{x}^q + \langle r\rangle$;
    $r \longleftarrow r + \widetilde{x}[i]$;
end
return $\widetilde{x}^q$
```

Regarding the complexity issues it is straightforward to observe that, if the verification phase can be run in $O(m)$ time, Algorithm 1 achieves a $\mathcal{O}(nm)$ worst case time complexity and requires only $\mathcal{O}(m)$ additional space for maintaining the scaled version of the pattern $\widetilde{x}$.

3 Monotonic Run Length Sampling

As observed above, the space consumption for representing the partial index obtained by monotonic run length scaling is not particularly satisfactory. This influences also the performance of the search algorithms in practical cases, as we will see in Section 4 which presents a preliminary experimental evaluation.

In this section we propose the application of an approach similar to that used by CDS sampling in order to obtain a partial index requiring a reduced amount of space, on the one hand, and is able to improve the performance of the search procedure, on the other hand.

For what we should present in this section it is useful to introduce some further notions. Given a monotonic run w of y, and assuming $w = y[i..i+k-1]$, we use the symbol $\mu(w)$ to indicate its starting position i in the text y.

The following definition introduces the *Monotonic Run Length Sampling* (MRLS) process which, given an input string y, constructs a partial index $\widetilde{y}^q$, which is the numeric sequence of all (and only) starting positions of any monotonic runs of y with length equal to q, for a given parameter $q > 1$.

Definition 6 (Monotonic Run Length Sampling). *Let y be a text of length n and let $\widetilde{y} = \langle|\rho_0|, |\rho_1|, .., |\rho_{k-1}|\rangle$ be the monotonic run length scaled version of y, with $|\widetilde{y}| = k$. In addition let ℓ be the maximal length of a monotonic run in y, i.e. $\ell = \max(|\rho_i| \ : \ 0 \le i < k)$. If q is an integer value, with $2 \le q \le \ell$, we define the Monotonic Run Length Sampled version of y, with pivot length q, as the numeric sequence $\widetilde{y}^q$, defined as $\widetilde{y}^q = \langle|\rho_{i_0}|, |\rho_{i_1}|, .., |\rho_{i_{h-1}}|\rangle$, where $h \le k$, $i_{j-1} < i_j$ for each $0 < j < h$, and $|\rho_{i_j}| = q$ for each $0 \le j < h$.*

Example 7. Again, as in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15. As already observed $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4\rangle$. Thus we have: $\widetilde{y}^2 = \langle 5, 8\rangle$, since $\mu(\langle 6, 12\rangle) = 5$ and $\mu(\langle 2, 9\rangle) = 8$; $\widetilde{y}^3 = \langle 0, 6, 9\rangle$, since $\mu(\langle 4, 5, 11\rangle) = 0$, $\mu(\langle 12, 12, 2\rangle) = 6$ and $\mu(\langle 9, 8, 6\rangle) = 9$; $\widetilde{y}^4 = \langle 2, 11\rangle$, since $\mu(\langle 11, 7, 6, 6\rangle) = 2$ and $\mu(\langle 6, 7, 10, 13\rangle) = 11$.

Function 2 depicts the pseudo-code of the algorithm which computes the MRL sampled version of an input string x of length m. It gets as input the monotonic run

Algorithm 2: Naïve algorithm based on Monotonic Run Length Sampling

```
Data: a pattern $x$ of length $m$, a text $y$ of length $n$ and its MRLS version $\widetilde{y}^q$ of length $k$
Result: all positions $0 \leq i < n$ such that $x$ occurs in $y$ starting at position $i$
$\widetilde{x} \longleftarrow$ Monotonic-Run-Length-Scaling($x, m$);
$\widetilde{x}^q \longleftarrow$ Monotonic-Run-Length-Sampling($x, m$);
$d \longleftarrow \widetilde{x}^*[0]$;
for $s \longleftarrow 0$ to $k - h$ do
    $r \longleftarrow \widetilde{y}^*[s]$;
    if $(r - d \geq 0$ and $r - d + m - 1 < n)$ then
        if Verify($y, n, x, m, r - d$) then Output($r - d$);
    end
end
```

length scaled version $\widetilde{x}$ of the string, its length h and the pivot length q. Then it constructs the sequence $\widetilde{x}^q$ incrementally by scanning the input sequence $\widetilde{x}$ element by element, proceeding from left to right. It is straightforward to prove that, also in this case, the worst case time complexity of the procedure is $\mathcal{O}(m)$.

3.1 Searching Using Monotonic Run Length Sampling

In this section we shortly describe a simple naïve procedure to search for all occurrences (in their exact or approximate version) of a pattern x of length m inside a text y of length n. The pseudo-code of such procedure is depicted in Algorithm 2.

The algorithm takes as input both the text y and its MRLS version $\widetilde{y}^q$ of length k. During the preprocessing phase the algorithm first computes the scaled version $\widetilde{x}$ of the input pattern x and, subsequently, computes its monotonic run length sampled version $\widetilde{x}^q$. Let k be the length of the sequence $\widetilde{y}$ and let h be the length of the sequence $\widetilde{x}^q$. In addition let d be the starting position of the first monotonic run length of x of length q, i.e. $d = \widetilde{x}^q[0]$.

The searching phase of the algorithm consists in a main **for** loop which iterates over the sequence $\widetilde{y}^q$, proceeding form left to right. For each element $\widetilde{y}^q[s]$, for $0 \leq s < h$, the algorithm calls the verification procedure to check an occurrence beginning at position $\widetilde{y}^q[s] - d$ in the original text. Roughly speaking, the algorithm aligns the first monotonic of length q in the pattern with all monotonic runs of length q in the text.

Example 8. As in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15 (the text) and let $x = \langle 12, 2, 9, 8, 6, 7, 10\rangle$ be a pattern of length 7. Then we have $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4\rangle$ and $\widetilde{x} = \langle 2, 2, 3, 3\rangle$. Assuming $q = 3$ we have also $\widetilde{y}^q = \langle 0, 6, 9\rangle$, $\widetilde{x}^q = \langle 2, 4\rangle$ and $d = 2$.

The algorithm considers any position $r \in \widetilde{y}^q$ as a candidate occurrence of the pattern and naively checks the whole pattern against the sub-sequence $y[r - d...r - d + m - 1]$ of the original text. Thus we have

- at the first iteration of the main **for** loop we have $s = 0$ and the algorithm would run a verification for the window starting at $\widetilde{y}^q[0] - d = 0 - 2 = -2$. However, since $-2 < 0$, such window is skipped.
- at the second iteration of the main **for** loop we have $s = 1$ and the algorithm would run a verification for the window starting at $\widetilde{y}^q[1] - d = 6 - 2 = 4$. Thus the pattern x is compared with the sub-sequence $y[4...10] = \langle 6, 6, 12, 12, 2, 9, 8\rangle$. However in

this case neither an exact occurrence nor an order preserving occurrence would be found.

– finally, at the third iteration of the main **for** loop we have $s = 2$ and the algorithm would run a verification for the window starting at $\widetilde{y}^q[9] - d = 9 - 2 = 7$. Thus the pattern x is compared with the sub-sequence $y[7...13] = \langle 12, 2, 9, 8, 6, 7, 10\rangle$. In this case the verification procedure would find both an exact occurrence and an order preserving match.

Regarding the complexity issues it is straightforward to observe that, if the verification phase can be run in $O(m)$ time, also Algorithm 1 achieves a $\mathcal{O}(nm)$ worst case time complexity and requires only $\mathcal{O}(m)$ additional space for maintaining the sampled version of the pattern $\widetilde{x}^q$.

4 Experimental Evaluation

In this section, we present experimental results in order to evaluate the performances of the sampling approaches presented in this paper for both exact string matching and order preserving pattern matching.

The algorithms have been implemented using the C programming language, and have been tested using the SMART tool [12][1] and executed locally on a MacBook Pro with 4 Cores, a 2.7 GHz Intel Core i7 processor, 16 GB RAM 2133 MHz LPDDR3, 256 KB of L2 Cache and 8 MB of Cache L3.[2] During the compilation we use the -O3 optimization option. .

For both exact and approximate string matching comparisons have been performed in terms of searching times. For our tests, we used six RAND-δ sequences of short integer values (each element of the sequence is an integer in the range [0...256]) varying around a fixed mean equal to 100 with a variability of δ, with $\delta \in \{4, 8, 16, 32, 64, 128, 250\}$. All sequences have a size of 3MB and are provided by the SMART research tool, available online for download. For each sequence in the set, we randomly selected 100 patterns, extracted from the text, and computed the average running time over the 100 runs.

4.1 Space Consumption

The first evaluation we discuss in this section relates to the space used to maintain the partial index. This evaluation is independent of the application for which the index is used and has the same value for both exact and approximate string matching.

Figure 2 shows the space consumption of the new proposed sampling approaches compared against the CDS and OTS methods. Data are reported as percentage values with respect to the size of the original text on which the index is built. Values are computed on six random texts with a uniform distribution and built on an alphabet of size 2^δ (abscissa) with $2 \leq \delta \leq 8$.

From the data shown in the Figure 2 it is possible to observe how the best results in terms of space are obtained by the MRLS and CDS. However, while the latter tends to have good results only for large alphabets, the MRLS approach proves to be

[1] In the case of OPPM experimental evaluations the tool has been properly tuned for testing string matching algorithms based on the OPPM approach

[2] The SMART tool is available online for download at `http://www.dmi.unict.it/~faro/smart/` or at `https://github.com/smart-tool/smart`.

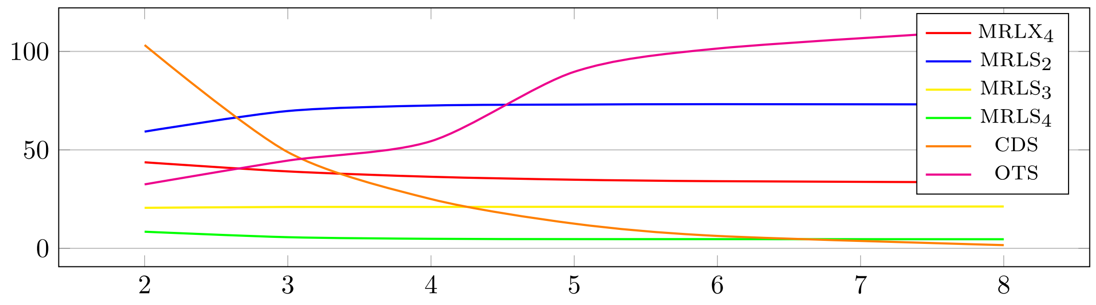


Figure 2. Space Consumption of sampling approaches to text searching. Data are reported as percentage values with respect to the size of the original text on which the index is built. Values are computed on six random texts with a uniform distribution and built on an alphabet of size 2^δ (abscissa) with $2 \leq \delta \leq 8$.

more flexible, obtaining good results also for small alphabets. However, it should be noted that these results were obtained on random texts with a uniform distribution of characters, a condition not favorable to the best performances for CDS and OTS.

4.2 Running Times for the OPPM problem

For the OPPM problem we took the standard NRq algorithm [6] as a reference point for our evaluation, since it is one of the most effective solution known in literature. Specifically we evaluated the following text-sampling solutions:

- (MRLX$_4$) The Monotonic Run Length Scaling approach (Section 2) using a compact representation of the elements (4 bits for each run length) and implemented using the Naïve algorithm.
- (MRLX$_8$) The Monotonic Run Length Scaling approach (Section 2) using a relaxed representation of the elements (an 8-bits char for each run length) and implemented using the Naïve algorithm.
- (MRLS$_q$) The Monotonic Run Length Sampling approach approach (Section 3) using a 32-bits integer value for each text position and implemented by sampling runs of length q, with $2 \leq q \leq 4$.

Table 1 shows the experimental evaluation for the OPPM problem on the Rand-δ short integer sequences where running times are expressed in milliseconds.

To better highlight the improvements obtained by the new proposed solutions, in Table 1 we show the running times only for the reference NRq algorithm, while we show the speed-up obtained against the latter for all the other tested algorithms. In this context a value greater than 1 indicates a speed-up of the running times proportional to the reported value, while a value less than 1 indicates a slowdown in performance. Best and second best results have been enhanced for a better visualization.

From our experimental results it turns out that the best solution in almost all the cases analyzed is the MRLS$_q$ algorithm which is almost always twice as fast as the NRq algorithm and reaches impressive speed-ups for very long patterns, up to 40 times faster than the reference algorithm. The MRLS$_q$ algorithm is second only to the MRLX$_8$ algorithm for short patterns ($m = 8$).

More specifically the MRLX$_8$ algorithm allows speed-ups compared to NRq, however these oscillate between one and a half times and twice as fast as the reference algorithm. In general, its performance is not that impressive. The MRLX$_4$ algorithm

δ	m	NRq	MRLX$_4$	MRLX$_8$	MRLS$_2$	MRLS$_3$	MRLS$_4$
	8	5.40	0.59	**1.19**	0.91	0.72	0.70
	16	5.21	0.74	1.34	**1.50**	0.76	0.73
	32	5.25	0.78	1.40	**3.65**	1.52	0.71
	64	5.07	0.71	1.25	3.57	**3.67**	1.16
4	128	5.21	0.81	1.37	3.59	**9.65**	2.52
	256	5.61	0.77	1.40	3.98	10.58	**14.38**
	512	5.71	0.81	1.46	4.05	10.77	**22.84**
	1024	5.20	0.78	1.42	3.59	9.45	**18.57**
	8	4.70	0.67	**1.21**	1.06	0.78	0.71
	16	4.89	0.92	1.49	**2.24**	0.82	0.74
	32	4.90	0.93	1.50	**3.29**	1.29	0.85
	64	4.96	1.01	1.57	3.67	**5.51**	1.10
8	128	5.03	1.03	1.58	3.70	**11.18**	1.76
	256	5.14	1.01	1.52	3.75	**11.42**	3.67
	512	4.73	0.92	1.44	3.50	10.28	**15.26**
	1024	5.20	0.95	1.57	3.91	11.56	**30.59**
	8	4.90	0.84	**1.52**	1.19	0.78	0.78
	16	5.24	1.05	1.88	**1.93**	1.15	0.82
	32	4.67	1.00	1.59	**3.38**	1.64	0.80
	64	4.87	0.99	1.61	3.87	**4.35**	1.07
16	128	4.98	1.02	1.63	4.02	**9.76**	2.06
	256	4.94	1.01	1.60	3.98	**12.05**	4.57
	512	5.20	1.16	1.95	4.19	12.68	**37.14**
	1024	4.96	1.05	1.68	4.03	12.10	**35.43**
	8	5.00	0.96	**1.64**	1.10	0.81	0.82
	16	4.66	0.99	1.77	**2.59**	0.92	0.77
	32	4.90	1.14	1.99	**3.38**	1.72	0.87
	64	4.83	1.09	1.90	**3.83**	3.43	1.04
32	128	4.88	1.08	1.85	4.17	**12.84**	1.51
	256	4.76	1.05	1.93	4.10	**12.53**	3.81
	512	5.07	1.15	1.96	4.30	13.34	**39.00**
	1024	5.23	1.13	1.99	4.59	13.41	**40.23**
	8	5.12	1.02	**1.78**	1.32	0.89	0.87
	16	4.93	1.02	1.83	**2.77**	0.89	0.81
	32	4.91	1.10	2.00	**3.48**	1.66	0.83
	64	4.90	1.12	2.04	3.60	**6.53**	0.98
64	128	4.84	1.08	1.94	3.87	**11.80**	1.58
	256	4.97	1.13	2.05	4.28	**13.08**	3.88
	512	4.66	1.02	1.83	4.09	12.26	**38.83**
	1024	4.82	1.07	1.84	4.38	12.68	**40.17**
	8	4.99	0.99	**1.81**	1.23	0.89	0.85
	16	4.83	1.05	1.92	**2.60**	0.93	0.77
	32	5.01	1.12	2.09	**3.48**	1.69	0.83
	64	4.76	1.08	1.91	3.33	**6.52**	0.86
256	128	4.96	1.09	1.95	3.59	**8.70**	1.58
	256	4.72	1.07	1.93	3.75	**11.51**	3.87
	512	4.79	1.06	1.94	3.89	**11.97**	10.41
	1024	4.89	1.12	2.08	4.37	12.54	**37.62**

Table 1. Experimental results for the OPPM problem on six Rand-δ short integer sequence, for $4 \leq \delta \leq 256$. Running times of the NRq are expressed in milliseconds. Results for all other algorithm are expressed in terms of speed-up obtained against the reference NRq algorithm. Best results and second best results have been enhanced.

performs worse and almost never brings improvements over NRq, probably due to the trade-off introduced by the compact representation of the partial index.

4.3 Running Times for the ESM problem

For the ESM problem, in accordance with what has been done in previous publications on the subject, we took the standard the Horspool (HOR) algorithm [17] as a reference point for our evaluation. We evaluated the following text-sampling solutions:

– (OTS) The Occurrence Text Sampling approach [8] introduced by Claude *et al.* and implemented by removing the first 8 characters of the alphabet, with the exception of the case $\delta = 4$, for which we removed the first 3 characters, and the case $\delta = 8$, for which we removed the first 7 characters.
– (CDS) The Character Distance Sampling approach [14] introduced by Faro *et al.*, implemented by selecting the 8th character of the alphabet, with the exception of the case $\delta = 4$, for which we selected the 4th character.
– (MRLS$_q$) The Monotonic Run Length Position Sampling approach (Section 3) using a 32-bits integer value for each text position and implemented using the sampling of runs of length q, with $2 \leq q \leq 4$.

Table 2 shows the experimental evaluation for the ESM problem on the Rand-δ short integer sequences where running times are expressed in milliseconds.

Also in this case to better highlight the improvements obtained by the new proposed solutions, in Table 2 we show the running times only for the reference HOR algorithm, while we show the speed-up obtained against the latter for all the other tested algorithms.

Our experimental results show that in the case of medium and large-sized alphabets, the MRLS$_q$ algorithm does not have the same performance as the CDS approach. However, it is very powerful in the case of small alphabets, a case in which the previous solutions suffered particularly and showed not exciting results.

It is also interesting to observe how the MRLS$_q$ algorithm proves to be competitive in the general case, always obtaining the second best results. The speed-up obtained by comparing it with the reference HOR algorithm reaches a factor of 2, in the case of small alphabets, and a factor of 6 for large alphabets.

5 Conclusions and Future Works

This article presents the first results relating to a work in progress. Specifically, we presented a new technique for text sampling called Monotonic Run Length Scaling (MRLX), an approach based on the length of the monotonic runs present within the text, flexible enough to be used both for exact string matching and for approximate string matching. A further improvement was obtained by sampling through the sampling of the lengths of the monotonic runs present in the text, an approach that we have called Monotonic Run Length Sampling (MRLS). In this work we also presented some first experimental tests for the evaluation of the two sampling approaches and implemented using naive search algorithms, focusing on two case studies: exact string matching and order preserving pattern matching.

The first experimental results obtained showed how the approaches are particularly versatile, obtaining considerable speed-ups on execution times, reaching gain

δ	m	HOR	CDS	OTS	MRLS$_2$	MRLS$_3$	MRLS$_4$
4	8	2.43	1.21	1.31	**1.48**	1.23	1.19
	16	2.10	1.19	1.31	**1.98**	1.27	1.25
	32	1.99	1.21	1.33	**2.65**	2.40	1.24
	64	2.16	1.23	1.33	2.96	**4.50**	2.00
	128	2.10	1.20	1.29	2.84	**6.77**	3.96
	256	1.99	1.23	1.33	2.62	6.42	**10.47**
	512	2.01	1.19	1.32	2.75	6.48	**12.56**
	1024	1.97	1.22	1.31	2.70	5.97	**12.31**
8	8	1.40	**2.19**	1.44	1.65	1.33	1.19
	16	1.11	**2.92**	1.52	1.88	1.50	1.39
	32	0.99	**3.67**	1.62	1.83	2.02	1.60
	64	1.01	**6.73**	1.58	1.84	4.04	1.87
	128	1.02	**9.27**	1.62	1.89	4.64	2.83
	256	1.04	**11.56**	1.58	1.96	4.73	4.52
	512	1.01	**14.43**	1.55	1.87	4.59	9.18
	1024	1.01	**16.83**	1.63	1.87	5.05	11.22
16	8	1.04	**1.65**	0.94	1.35	1.41	1.39
	16	0.79	**2.26**	1.27	1.76	1.98	1.52
	32	0.68	**4.00**	1.70	1.58	2.52	1.79
	64	0.62	**6.20**	1.72	1.48	3.26	2.07
	128	0.61	**10.17**	1.85	1.45	3.59	3.21
	256	0.61	**12.20**	1.69	1.49	3.81	4.69
	512	0.63	**15.75**	1.66	1.50	3.94	7.88
	1024	0.61	**20.33**	1.74	1.45	3.81	7.62
32	8	0.92	**1.53**	1.39	1.39	1.46	1.48
	16	0.66	**1.94**	1.83	1.74	1.94	1.74
	32	0.55	**3.06**	2.20	1.49	2.75	2.04
	64	0.53	**6.62**	2.52	1.43	3.31	2.41
	128	0.50	**8.33**	2.63	1.32	3.57	3.12
	256	0.50	**12.50**	2.78	1.35	3.57	4.55
	512	0.47	**15.67**	2.94	1.27	3.62	6.71
	1024	0.50	**25.00**	2.78	1.39	3.33	7.14
64	8	0.85	**1.60**	1.52	1.47	1.52	1.49
	16	0.60	**2.07**	2.00	1.76	1.88	1.71
	32	0.50	2.78	2.50	1.28	**2.94**	2.27
	64	0.46	**5.11**	3.07	1.39	3.29	2.71
	128	0.44	**11.00**	3.38	1.29	3.38	4.00
	256	0.43	**14.33**	3.31	1.30	3.31	4.78
	512	0.43	**21.50**	3.91	1.30	3.31	6.14
	1024	0.42	**21.00**	3.50	1.24	3.23	6.00
256	8	0.77	**1.54**	1.54	1.71	1.57	1.57
	16	0.58	**2.23**	2.07	1.81	2.07	1.93
	32	0.46	**3.29**	2.56	1.39	2.88	2.19
	64	0.45	**5.00**	3.45	1.36	3.46	3.21
	128	0.44	**11.00**	3.37	1.33	3.38	4.40
	256	0.45	**15.00**	4.90	1.32	3.21	5.00
	512	0.47	**23.50**	5.85	1.42	3.62	5.87
	1024	0.47	**23.50**	5.86	1.38	3.36	5.87

Table 2. Experimental results for the ESM problem on six Rand-δ short integer sequence, for $4 \leq \delta \leq 256$. Running times of the HOR are expressed in milliseconds. Results for all other algorithm are expressed in terms of speed-up obtained against the reference HOR algorithm. Best results and second best results have been enhanced.

factors of 40, in particularly favorable conditions. The new techniques presented therefore serve as good starting points for improvements in future investigations. The first aspect of the research that can be carried out for future improvements is the choice of the underlying search algorithms used for the implementation of the partial index filtering procedure. The MRLX approach requires algorithms that scan the text, reading every single character, in order not to lose the information relating to the alignment with the position of the occurrences in the original text. This can be done using more efficient string matching algorithms such as the KMP [19] or the Shift-And [1] algorithms. The MRLS approach, using text positions as the primary information of the numerical sequence, can afford the use of more efficient string matching algorithms that skip portions of the text during the search. In this case, we expect more significant improvements in execution times.

However, one of the most interesting aspects to be analyzed in a future work is the applicability of the new sampling approach to other approximate string matching problems. We can now say that the MRLX technique is well suited to solve other problems such as Cartesian-tree pattern matching [22] or shape preserving pattern matching [6], which have a strong relationship with the OPPM problem. Furthermore we argue that an approximate search within the partial index can also allow to obtain solutions for problems in which the occurrence of the pattern is found within the text in the form of some kind of permutation of its characters. And many non-standard string matching problems belong to this category, such as swap matching [15], string matching with inversions and/or moves [5] as well as the jumbled matching itself.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) Oct. 1992, p. 74–82.
2. D. Belazzougui, A. Pierrot, M. Raffinot, and S. Vialette: *Single and multiple consecutive permutation motif search*, in Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings, vol. 8283 of Lecture Notes in Computer Science, Springer, 2013, pp. 66–77.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
4. D. Cantone, S. Faro, and E. Giaquinta: *Adapting boyer-moore-like algorithms for searching huffman encoded texts.* Int. J. Found. Comput. Sci., 23(2) 2012, pp. 343–356.
5. D. Cantone, S. Faro, and E. Giaquinta: *Text searching allowing for inversions and translocations of factors.* Discret. Appl. Math., 163 2014, pp. 247–257.
6. D. Cantone, S. Faro, and M. O. Külekci: *Shape-preserving pattern matching*, in Proceedings of the 21st Italian Conference on Theoretical Computer Science 2020, vol. 2756 of CEUR Workshop Proceedings, CEUR-WS.org, 2020, pp. 137–148.
7. S. Cho, J. C. Na, K. Park, and J. S. Sim: *Fast order-preserving pattern matching*, in Combinatorial Optimization and Applications - 7th International Conference, COCOA 2013, Proceedings, vol. 8287 of Lecture Notes in Computer Science, Springer, 2013, pp. 295–305.
8. F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio: *String matching with alphabet sampling.* J. Discrete Algorithms, 11 2012, pp. 37–50.
9. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: *Speeding up two string-matching algorithms.* Algorithmica, 12(4/5) 1994, pp. 247–267.
10. S. Faro and M. O. Külekci: *Efficient algorithms for the order preserving pattern matching problem*, in Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Proceedings, vol. 9778 of Lecture Notes in Computer Science, Springer, 2016, pp. 185–196.

11. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results*. ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
12. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 99–111.
13. S. Faro and F. P. Marino: *Reducing time and space in indexed string matching by characters distance text sampling*, in Prague Stringology Conference 2020, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020, pp. 148–159.
14. S. Faro, F. P. Marino, and A. Pavone: *Efficient online string matching based on characters distance text sampling*. Algorithmica, 82(11) 2020, pp. 3390–3412.
15. S. Faro and A. Pavone: *An efficient skip-search approach to swap matching*. Comput. J., 61(9) 2018, pp. 1351–1360.
16. P. Ferragina and G. Manzini: *Indexing compressed text*. J. ACM, 52(4) 2005, pp. 552–581.
17. R. N. Horspool: *Practical fast searching in strings*. Softw. Pract. Exp., 10(6) 1980, pp. 501–506.
18. J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order-preserving matching*. Theor. Comput. Sci., 525 2014, pp. 68–79.
19. D. E. Knuth, J. H. M. Jr., and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
20. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches*. SIAM J. Comput., 22(5) 1993, pp. 935–948.
21. G. Navarro and J. Tarhio: *Lzgrep: a boyer-moore string matching tool for ziv-lempel compressed text*. Softw. Pract. Exp., 35(12) 2005, pp. 1107–1130.
22. S. Song, G. Gu, C. Ryu, S. Faro, T. Lecroq, and K. Park: *Fast algorithms for single and multiple pattern cartesian tree matching*. Theor. Comput. Sci., 849 2021, pp. 47–63.
23. U. Vishkin: *Deterministic sampling - A new technique for fast pattern matching*. SIAM J. Comput., 20(1) 1991, pp. 22–40.
24. A. C. Yao: *The complexity of pattern matching for a random string*. SIAM J. Comput., 8(3) 1979, pp. 368–387.

Searching with Extended Guard and Pivot Loop

Waltteri Pakalén[1], Jorma Tarhio[1], and Bruce W. Watson[2]

[1] Department of Computer Science
Aalto University, Finland
[2] Information Science, Centre for AI Research
School for Data-Science & Computational Thinking
Stellenbosch University, South Africa

Abstract We explore practical optimizations on comparison-based exact string matching algorithms. We present a *guard test* that compares q-grams between the pattern and the text before entering the match loop, and evaluate experimentally the benefit of optimization of this kind. As a result, the *Brute Force* algorithm gained most from the guard test, and it became faster than many other algorithms for short patterns. In addition, we present variations of a recent algorithm that uses a special skip loop where a pivot, a selected position of the pattern, is tested at each alignment of the pattern and in case of failure; the pattern is shifted based on the last character of the alignment. The variations include alternatives for the pivot and the shift function. We show the competitiveness of the new algorithm variations by practical experiments.

Keywords: exact string matching, tune-up of algorithms, guard test, skip loop, experimental comparison

1 Introduction

Searching for occurrences of a string pattern in a text is a fundamental task in computer science. It finds use in many domains such as text processing, bioinformatics, computer vision, and intrusion detection. Depending on the problem definition, the pattern occurrences can be exact, approximate, permuted, or any other variation. Here, we consider the *exact online string matching problem*, where the occurrences are exact and the pattern may be preprocessed but not the text. Formally, the problem is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an alphabet Σ, find all the occurrences (including overlapping ones) of P in T. String matching is an extensively studied problem with over a hundred published algorithms in the literature, see e.g. Faro and Lecroq [15].

Guard test [20,26,27] is a widely used technique to speed-up comparison-based string matching algorithms. The idea is to test certain pattern positions before entering a match loop. Guard test is a representative of a general optimization technique called loop peeling, where a number of iterations are moved in front of the loop. As a result, the computation becomes faster because of fewer loop tests. Original guard tests deal with single characters — here we consider extended guards: q-grams that are substrings of q characters.

Processor (CPU) development has gradually improved the speed of multicharacter reads — especially the penalty for misaligned memory accesses has disappeared. In our earlier paper [32], we applied q-gram guards to the Dead-Zone algorithm [33] and we anticipated that the guard test with a q-gram might improve the performance of some other algorithms as well. In this paper, we show that this is true. Especially, the transformed Brute Force algorithm is faster than many other algorithms for patterns

$m \leq 16$. Testing of q-grams as entities has been used before by Faro and Külekci [13], Sharfuddin and Feng [28] as well as Khan [22].

A few years ago, Al-Ssulami [1] introduced an interesting algorithm called SSM (short for Simple String Matching) which utilizes a special skip loop where a pivot, a selected position of the pattern, is tested at each alignment of the pattern and in case of failure, the pattern is shifted based on the last character of the alignment. This algorithm is not widely known, but it contains a unique shift heuristic that is worth observing. In this paper, we introduce several variations of SSM including alternatives for the pivot and the shift function.

Our emphasis is on the practical efficiency of algorithms and we show the competitiveness of the new algorithm variations by practical experiments.

The rest of the paper is organized as follows: Section 2 reviews typical loop structures of exact string matching algorithms. Section 3 presents how guard test with a q-gram is implemented, Section 4 presents the principles of the SSM algorithm, and Section 5 introduces our variations of SSM. Section 6 shows the results of our practical experiments, and the discussion of Section 7 concludes the article.

2 Loops in String Matching

Let us consider Algorithm 1, which is a general model of string matching of Boyer–Moore type [5]. Two phases alternate during execution. The first phase is the while loop in line 3 which goes through the alignments of the pattern. This loop is called a *skip loop* [20], which is aimed at forwarding the pattern quickly rightwards. The variable j is associated with the alignments: t_j is the last character of an alignment. When the skip loop finds an alignment which is a potential occurrence of the pattern, that alignment is checked in second phase in line 4 and the skip loop is resumed after moving the pattern in line 5. In the preprocessing phase executed before Algorithm 1, the shift functions $\mathbf{shift}_1$ and $\mathbf{shift}_2$ are computed based on the pattern.

Algorithm 1
```
1  $j \leftarrow m - 1$
2  while $j < n$ do
3      while **condition** do $j \leftarrow j + \mathbf{shift}_1$
4      if $t_{j-m+1} \cdots t_j = P$ then report occurrence
5      $j \leftarrow j + \mathbf{shift}_2$
```

In line 1, the pattern is placed at the first position of the text. Line 3 can be missing as it is the case in the original Boyer–Moore algorithm [5] or it can be in a reduced form

if **condition** then

where the then-branch contains lines 4 and 5 as in Horspool's algorithm [18]. In order to reduce tests in the skip loop, a copy of the pattern may be concatenated to the end of the text as a stopper. The **condition** of line 3 examines a suffix of the alignment window. In certain algorithms (Cantone and Faro [8], Peltola and Tarhio [25]), even some characters following the alignment are considered. The displacement $\mathbf{shift}_1$ of the pattern is typically a constant [25], but it can be function based on a character or a q-gram at a constant distance (Faro and Lecroq [14]).

The test in line 4 can be implemented as a match loop in various orders: backward, forward, reversed frequency etc. (Hume and Sunday [20]) or with the memcmp library function. In the algorithms of BNDM type (Ďurian et al. [11]) the match loop does not compare characters but updates a bit vector. Some algorithms (Hume and Sunday [20], Raita [26,27]) contain guard tests which are located between lines 3 and 4.

The displacement $\mathbf{shift}_2$ of the pattern in line 5 can be a constant (Ďurian et al. [11]) or a function (Boyer and Moore [5]). In some algorithms, there is a separate shift in case of a match in line 4 (Ďurian et al. [11]).

The skip loop of Alg. 1 in line 3 has a unique form in the SSM algorithm [1]:

$$\text{while } p_c \neq t_{j-m+1+c} \text{ do } j \leftarrow j + h[t_j]$$

where p_c is a *pivot character* p_c in P, $0 \leq c < m-1$. SSM uses Horspool's [18] shift table h which carries out the bad character heuristic for p_{m-1}. If a character x does not occur in $p_0 \cdots p_{m-2}$, $h[x]$ is m. Otherwise, $h[x]$ is $m-1-i$, where $p_i = x$ is the rightmost occurrence of x in $p_0 \cdots p_{m-2}$. In the following, we call a skip loop of this kind a *pivot loop*. In other algorithms than SSM, the test character of the skip loop is either the last character of the pattern or a character close to the last character. In SSM, the pivot position is selected so that the shift is long after the pivot loop.

3 Extended Guard Test

The match loop is subject to various optimizations that add or move logic from the match loop into a filter. For instance, the Fast-Search algorithm [7] shifts whenever the right-most character of an alignment mismatches. Verifying this mismatch does not require an explicit comparison between the characters. Instead, the mismatch is encoded in a shift table during preprocessing. Thus, the logic is moved from the match loop into a filter that determines if a match loop is necessary to perform based on the results of the shift table lookup.

Similarly, a guard test is a line of optimizations that compares particular character(s) between the pattern and the alignment window to determine whether a match loop is necessary. Hume and Sunday [20] presented a guard test that compares the least frequent character of the pattern (over the alphabet Σ) with the corresponding text character.

More recently, Khan [22] presented a transformation of the match loop on comparison-based algorithms that involves testing q-grams. A similar approach was earlier developed by Sharfuddin and Feng [28] for Horspool's algorithm [18].

On 64-bit processors, reading a q-gram can be performed in one instruction for $q = 2^i$ for $i = 0, 1, 2, 3$. We implemented the extended guard test as follows. The first q-gram of the pattern is stored in a variable during preprocessing. The first q-gram of an alignment window is stored in another variable. If the values of these variables differ, an occurrence is impossible and the match loop is skipped. If they match, the match loop is executed to check for an occurrence. The match loop can now skip the first q characters because the characters already matched in the guard test.

The processor word size limits q to be less than or equal to that size in bytes. Additionally, the pattern cannot be shorter than the q-gram or otherwise the guard test matches characters outside the right ends of the pattern and the alignment window. An implementation can be adapted to run for any pattern length by branching to a non-guard version of the algorithm in the beginning. Lastly, the guard test is not applicable to every comparison-based algorithm. A skip loop or a similar fast loop [20] may ruin the benefit of a guard test. And some algorithms, such as the original Boyer–Moore algorithm [5] and the Fast-Search algorithm [7], require knowing the position of the first mismatching character in order to shift correctly.

4 Original SSM

4.1 Algorithm

The main idea of the SSM algorithm [1] is to select a pivot character p_c that will allow a long shift in case of found p_c. As far we know, the shift heuristic of SSM is different from earlier algorithms. The pseudocode of SSM is presented as Algorithm 2. Let the pattern P be aligned with the text T so that p_{m-1} is at t_j. As in the model algorithm Algorithm 1, two phases alternate during execution. The first phase is the pivot loop where the pivot p_c is compared with a text character $t_{j-(m-1)+c}$ and Horspool's [18] shift $h[t_j]$ is taken in case of mismatch (line 4). In case of a match, a potential occurrence of P has been found and it is checked in the second phase in a match loop (lines 6–9). After the match loop, P is shifted according to a proprietary shift table s (line 10) and the pivot loop is resumed. In order to be able to stop the pivot loop, a copy of P is placed at t_n as a stopper (line 1).

Algorithm 2: SSM

```
1  place a copy of $P$ at $t_n$
2  $j \leftarrow m-1$
3  while $j < n$ do
4      while $p_c \neq t_{j-m+1+c}$ do $j \leftarrow j + h[t_j]$
5      $i \leftarrow m-1$
6      while $i \geq 0$ and $p_i = t_{j-m+1+i}$ do $i \leftarrow i-1$
7      if $i < 0$ then
8          if $j < n$ then report match at $k$
9          $i \leftarrow i+1$
10     $j \leftarrow j + s[i]$
```

4.2 Pivot Character and Shift Tables

The pivot character $p_c, 0 \leq c \leq m-2$, is selected to enable a long safe shift in case of a character match at p_c. Note that p_{m-1} is not allowed to be the pivot in SSM. When the pivot loop stops it is known that p_c was found, and the shift table s utilizes this fact. In order to determine p_c, a distance array[1] $d[i]$ is computed where $d[i]$ is the distance of p_i to its next occurrence to the left or $i+1$ if no such an occurrence exists. Formally,

$$d[i] = \begin{cases} \min(i+1, \min\{k > 0 \mid p_i = p_{i-k}\}) & \text{if } i < m-1 \\ 0 & \text{if } i = m-1 \end{cases}$$

Let i with the largest $d[i]$ be c. If there are more than one such indices, the smallest one is selected. In other words, c is $\min\{i \mid d[i] \geq d[j] \text{ for j} = 0, \ldots, \text{m} - 2\}$.

The shift table s applies two heuristics which consider runs, i.e. sequences of equal characters. If $p_j \cdots p_k$ is a run and $p_{j-1} \neq p_j$ or $j = 0$ holds, then $s_1[i]$ is $i - j + 1$

[1] Sunday [30] used a similar construction in his MS algorithm.

for $i = j, \dots, k$. Informaly, $s_1[i]$ is a shift to get a different character in P at the text position that caused a mismatch. If $p_{j-1} \neq p_j$ and $p_k \neq p_{k+1}$ or $k = m-1$ holds, then $s_2[j-1]$ is $k-j+2$. Otherwise, $s_2[j]$ is 1. Informaly, $s_2[i]$ shifts the pattern over a complete run. Finally[2], $s[i]$ is $\max(s_1[i], s_2[i], d[c])$. See an example in Table 1, where P is abbbbf and $p_c = p_1$.

		j			k	
i	0	1	2	3	4	5
p_i	a	b	b	b	b	f
d	1	2	1	1	1	0
s_1	1	1	2	3	4	1
s_2	5	1	1	1	2	1
s	5	2	2	3	4	2

Table 1. Data structures of SSM for P = abbbbf.

4.3 Remarks

Al-Ssulami's experiments [1] show that SSM is faster than Horspool's algorithm (Hor) [18]. The comparison is a bit unfair because SSM has a skip loop and Hor does not contain one. The time complexity of SSM is $O(mn)$ in the worst case for $P = a^m$ and $T = a^n$. However, SSM works in linear time for several cases that are in $O(mn)$ for many algorithms of Boyer-Moore type, for example $P = a^{m/2-1}ba^{m/2}$ and $T = a^n$.

The example of Table 1 suggests that the s_2 heuristic could be improved. For example, the value of $s_2[2]$ could be 3 with a relaxed definition. However, we will keep the original s in the following.

The HSSMq algorithm by Al-Ssulami et al. [3] uses a q-gram as a pivot. However, the loop structure of HSSMq is different from SSM because the value of the tested q-gram is used for shifting whereas the shift of the pivot loop in SSM is based on another place of the alignment. Thus HSSMq does not contain a similar pivot loop. Moreover, HSSMq has been designed for long patterns in a small alphabet. Al-Ssulami's third algorithm FHASHq [2] applying q-grams has also a different loop structure. We chose FHASH2 to be one of the reference methods in our experimental comparison.

5 Variations of SSM

As far as we know, the pivot loop of SSM is of a new type of skip loop which has not been presented earlier. The pivot loop opens possibilities for variations in the following features:

– the pivot character
– the shift function of the pivot loop
– the shift function of the match loop

In this section, we will present several variations of SSM. Our aim is to improve the performance of SSM on short English patterns. Each alternative of a feature is given a unique letter, which will be concatenated to the name of a variation. For example, the variation SSM-UBC contains the alternatives U, B, and C.

[2] In the original article [1], s is defined as an outcome of an algorithm without s_1 and s_2.

5.1 Variations of Pivot

In SSM the pivot is chosen so that the shift after the match loop could be long. Another principle would be to minimize the number of exits from the pivot loop. Then the least frequent character of P is a good choice. Let F denote this alternative. The least frequent character is utilized in many algorithms [18,20,23,30], mostly in the match loop. Instead of a single character, the least frequent q-gram could be used, but the size of the required frequency table is impractical for large alphabets. Külekci [23] considers even the use of discontinuous q-grams.

The cost of reading a q-gram from the memory for $q = 2, 4$, and 8 is almost equal to the cost of reading a single character in modern processors. Also implementing a match loop as a single call of the library function memcmp is faster in some processors than a character by character loop. We tried the following q-gram pivots.

- $p_{m-4} \dots p_{m-1}$ (alternative U)
- $p_{m-8} \dots p_{m-1}$ (alternative V)
- $p_0 \dots p_3$ and $p_{m-4} \dots p_{m-1}$ using a short circuit (alternative W)
- P with memcmp (alternative M)

The alternatives U and W work for $m \geq 4$ and V for $m \geq 8$. In the alternatives U, V, and W, the match loop checks the remaining characters of P. In the alternative M there is no match loop at all, and the shift in case of a match is $m - o$ where o is the length of the overlap of P with itself.

5.2 Variations of Shift of the Pivot Loop

If the pivot does not match, we know that the current alignment of P does not hold an occurrence. Therefore the shift need not necessarily be based on t_j but Sunday's shift [30] based on t_{j+1} and Berry and Ravindran's shift [4] based on a 2-gram $t_{j+1}t_{j+2}$ can be applied as well. Let S denote Sunday's shift and let B denote Berry and Ravindran's shift with 16-bit reads, i.e. a 2-gram is read in a single operation. Kalsi et al. [21] show that the shift based on $t_j t_{j+1}$ is better than Berry and Ravindran's shift on DNA data. Let X denote this shift with 16-bit reads.

The restriction that p_{m-1} cannot be a pivot is unnecessary. The algorithm becomes slightly faster without this restriction for large alphabets. Especially patterns like $a^{m-1}b$ can be found faster. Let A denote the variation where p_{m-1} is allowed to be a pivot.

5.3 Variations of Shift of the Match Loop

The shift of SSM for patterns like $(ab)^{m/2}$ is shorter than in the Boyer-Moore algorithm [5] because SSM does not apply the good suffix shift. The good suffix shift is easy to preprocess in linear time and to combine it with the original shift table s. Let C denote the combined shift which includes the good suffix shift.

5.4 Special Variations

A guard test of a single character before the match loop is denoted by G.

Horspool [18] presented the SLFC algorithm which searches for occurrences of the least frequent character of P and checks the alignments of P associated with each found occurrence. We made a variation L, where the shift of the pivot loop is replaced by the strchr library instruction which searches for the next occurrence of the least frequent character of P. From SSM-WB we made a hybrid algorithm SSM-WBZ for English data, where SSM-L is used when the frequency of the least frequent character of P is below a threshold and SSM-WB is used otherwise.

Let us assume that there is a run of k characters in the pattern and the last character of the run has been chosen as the pivot. In this situation, the shift of the pivot loop is "$j = j + k - 1 + h[t_{j+k-1}]$" in the case of Horspool's shift. Sunday's shift and Berry and Ravindran's shift are treated correspondingly. This modification is faster for patterns containing a run of four or more characters, but it is, unfortunately, slower in the average case.

6 Experimental Results

6.1 Guard Test

We added the guard test to several comparison-based string matching algorithms in order to experimentally evaluate its effectiveness. Table 2 lists the selected algorithms. Each algorithm was transformed with possible values of q. That is, on a 64-bit processor, four values 1, 2, 4, and 8 were possible for q.

ASKIP [9]
Brute Force (BF)
BR [4]
GRASPm [10]
HOR [18]
NSN [17]
RAITA [26]
SMITH [29]
TS [6]
TSW [19]

Table 2. Transformed algorithms.

The experiments were run in the SMART framework [16,12], with default configurations (e.g. the text size was 1 MiB). The processor used was Intel Core i7-6500U with 4 MB L3 cache and 16 GB RAM; this CPU has a Skylake microarchitecture and has none of the misaligned access performance penalties found on some other microarchitectures. The operating system was Ubuntu 16.04 LTS.

The base implementations were taken from the repository of SMART [12]. The results are reported as speed-ups: the ratio of running times of the transformed algorithm and the original one.

Tables 3–6 show the speed-ups on English text, genome sequence, rand2, and rand250, respectively, for $q = 8$ when $m > 4$ and for $q = 4$ when $m = 4$. The other q values have been left out because they were either on par with $q = 8$ or slower. However, there were exceptions for $q = 1$ and $q = 4$ (not shown in the tables): $q = 1$ interestingly displayed up to 1.14 speed-up on both BF and TS, and $q = 4$ was almost always neck and neck with $q = 8$, except on rand2, where $q = 8$ was substantially faster.

The larger the alphabet is, the smaller the speed-ups are. This is evident when comparing rand2 and genome to English and rand250. Table 5 for rand2 exhibits

m	4	8	16	32	64	128	256	512	1024
ASKIP	0.99	1.02	1.02	0.97	1.01	1.01	1.00	1.00	0.99
BF	3.20	3.11	3.25	3.38	3.27	3.21	3.22	3.22	3.23
BR	1.19	1.12	1.10	1.07	1.02	1.00	1.02	1.02	1.00
GRASPM	1.01	1.03	1.01	1.04	1.00	0.96	1.02	1.02	1.04
HOR	1.34	1.26	1.16	1.16	1.13	1.09	1.13	1.10	1.02
NSN	1.25	1.29	1.34	1.37	1.35	1.32	1.32	1.31	1.32
RAITA	1.15	1.14	1.10	1.06	1.05	1.03	1.00	1.06	1.06
SMITH	1.35	1.31	1.23	1.19	1.17	1.07	1.02	1.08	1.06
TS	1.20	1.15	1.13	1.07	1.01	1.01	1.00	1.00	1.04
TSW	1.32	1.25	1.18	1.09	1.08	1.00	1.02	1.00	1.00

Table 3. Speed-ups on English text.

m	4	8	16	32	64	128	256	512	1024
ASKIP	1.17	1.14	1.06	1.03	1.04	1.01	1.00	1.00	1.00
BF	6.71	7.10	7.10	7.32	7.22	7.08	7.03	7.32	7.38
BR	1.72	1.61	1.43	1.30	1.26	1.31	1.27	1.27	1.29
GRASPM	1.12	1.15	1.18	1.28	1.41	1.55	1.56	1.50	1.52
HOR	1.88	1.73	1.64	1.63	1.64	1.63	1.64	1.64	1.63
NSN	1.56	1.65	1.71	1.74	1.72	1.66	1.65	1.65	1.65
RAITA	1.65	1.61	1.53	1.54	1.54	1.53	1.54	1.53	1.54
SMITH	1.73	1.61	1.51	1.57	1.56	1.52	1.52	1.52	1.51
TS	1.50	1.52	1.47	1.43	1.42	1.42	1.36	1.34	1.31
TSW	2.17	1.96	1.72	1.56	1.49	1.50	1.51	1.49	1.49

Table 4. Speed-ups on genome sequence.

m	4	8	16	32	64	128	256	512	1024
ASKIP	1.33	1.63	1.47	1.35	1.26	1.14	1.12	1.03	1.01
BF	4.37	9.81	9.38	9.38	9.16	9.14	9.37	9.37	9.26
BR	1.47	2.01	2.05	2.05	2.06	2.05	2.06	2.06	2.06
GRASPM	1.25	1.97	2.53	3.12	3.51	3.50	3.53	3.45	3.39
HOR	1.64	2.29	2.38	2.39	2.40	2.39	2.40	2.40	2.40
NSN	1.54	1.63	1.69	1.72	1.70	1.62	1.62	1.61	1.61
RAITA	1.76	2.19	2.42	2.55	2.55	2.57	2.56	2.57	2.58
SMITH	1.49	1.96	2.02	2.11	2.09	2.06	2.05	2.04	2.05
TS	1.41	1.81	1.83	1.90	1.92	1.92	1.91	1.86	1.85
TSW	1.77	2.66	2.71	2.67	2.70	2.67	2.69	2.70	2.66

Table 5. Speed-ups on rand2.

significant speed-ups, even for some algorithms that normally enter the match loop relatively rarely (e.g. ASKIP). Almost all the transformed algorithms exhibit substantial speed-ups for rand2. Slightly similar observations apply to genome sequence but to a much lesser extent. In this case, for many algorithms the speed-ups are in range of 1.4 to 1.7. On the other hand, rand250 shows almost no speed-up in any case other than BF. RAITA and SMITH exceed a speedup of 1.1 for a few pattern lengths but otherwise a mere few cases reach 1.05. The English text is somewhere between the other texts. It reaches speed-ups of up to 1.3. In many cases, the speed-ups hover around 1.0 to 1.15. For English, the speed-ups additionally seem to decrease for the longest patterns.

The reason that smaller alphabets work better can be found in the probabilities to match a pair of characters between the pattern and the text. Such a pairwise comparison on average is more probable to match as the alphabet size goes down.

m	4	8	16	32	64	128	256	512	1024
ASKIP	1.02	1.01	1.06	1.03	1.04	1.02	1.01	1.01	1.00
BF	2.03	2.03	1.92	1.97	2.03	2.04	2.06	2.04	1.97
BR	0.98	0.98	0.98	1.00	1.00	1.02	1.02	1.00	1.00
GRASPM	1.00	0.99	0.96	1.02	1.00	1.00	1.00	1.04	0.98
HOR	1.06	1.04	1.01	1.04	1.02	1.00	1.00	1.06	1.00
NSN	0.99	1.00	1.04	1.03	1.04	1.01	1.03	1.03	1.03
RAITA	0.99	1.00	1.00	1.00	0.98	1.02	1.15	1.13	1.06
SMITH	1.13	1.11	1.07	1.07	0.98	1.00	1.02	1.00	0.98
TS	1.04	1.01	1.01	1.03	1.02	1.01	0.99	1.03	1.04
TSW	1.02	0.99	1.01	1.00	0.99	1.01	1.00	0.98	1.00

Table 6. Speed-ups on rand250.

Thus, the original algorithms must carry out multiple pairwise comparisons before coming across a mismatch. Meanwhile, the guard test does not care whether one, two, three, or more pairs match between the q-grams. As long as one of them mismatches, the guard test fails altogether. Hence, there are more savings in execution for smaller alphabets. Similar reasoning applies as to why $q = 4$ and $q = 8$ perform neck and neck. A comparison between a pair of four characters mismatches often enough that the difference is nearly negligible between $q = 4$ and $q = 8$.

While we have included BF in the tables, its speed-ups are somewhat incomparable to the other speed-ups because of its nature. BF is evidently going to be the one benefitting the most from such an optimization. However, note that it became quite competent with the guard test. The transformed BF was the fastest algorithm in the test set on English text, genome sequence and rand2 for $m \leq 16$. This is a remarkable result because short patterns are most important in practice. This is also an example of how technology can beat complicated algorithms (see another example [31] of that).

	m=4		8		16		32
BF4	0.74	BF8	0.75	BF8	0.72	SMITH8	0.67
TSW4	1.18	TSW8	0.92	TSW8	0.77	GRASPM8	0.69
TSW	1.56	SMITH8	1.07	SMITH8	0.80	RAITA8	0.69
TS4	1.57	RAITA8	1.11	RAITA8	0.84	TSW8	0.69
SMITH4	1.61	HOR8	1.13	HOR8	0.86	HOR8	0.69
BR4	1.64	TSW	1.15	GRASPM8	0.87	BF8	0.71
RAITA4	1.67	GRASPM8	1.18	GRASPM	0.88	GRASPM	0.72
NSN4	1.71	GRASPM	1.21	TSW	0.91	BR8	0.72
HOR4	1.71	BR8	1.22	BR8	0.91	RAITA	0.73
GRASPM4	1.81	TS8	1.23	RAITA	0.92	ASKIP	0.74
GRASPM	1.82	RAITA	1.26	SMITH	0.98	TSW	0.75
TS	1.88	BR	1.37	HOR	1.00	ASKIP8	0.76
RAITA	1.92	SMITH	1.40	BR	1.00	BR	0.77
BR	1.95	HOR	1.42	ASKIP8	1.02	SMITH	0.80
NSN	2.13	TS	1.42	TS8	1.04	HOR	0.80
SMITH	2.18	NSN8	1.67	ASKIP	1.04	TS8	0.91
HOR	2.29	ASKIP8	1.70	TS	1.17	TS	0.97
BF	2.37	ASKIP	1.73	NSN8	1.70	NSN8	1.70
ASKIP	2.46	NSN	2.15	NSN	2.28	NSN	2.33
ASKIP4	2.49	BF	2.33	BF	2.34	BF	2.40

Table 7. Ranks of algorithms according to average running times of 500 English patterns in milliseconds for $m = 4, 8, 16, 32$.

Besides BF, we did not notice any drastic changes in the algorithm rankings. Table 7 shows algorithm rankings for English for $m = 4, 8, 16, 32$. The suffix 4 or 8 refer to the transformed algorithm with $q = 4$ or $q = 8$, respectively.

In addition, we tried the guard test with the HASH3, HASH5, and HASH8 algorithms [24]. The results did not show any improvement except with rand2. This was expected, because those algorithms contain an already-efficient skip loop. For instance, HASH3 enters the match loop only when the hash value of the last 3-gram of an alignment window is equal to the hash value of the last 3-gram of the pattern. This means that only a very small number of alignments are checked. As for other tested algorithms, the transformed BF was faster than the HASH3, HASH5, and HASH8 algorithms for $m \leq 16$.

Finally, we ran BF8 against all the 195 algorithms in the SMART repository. BF8 was faster than all the others for $m = 8$ on rand2. Its rank was #16 for $m = 8$ on English.

We also ran experiments with BF8b, a variation of BF8, for $8 \leq m \leq 16$. BF8b tests two 8-grams, the first and last 8-gram of an alignment, with a short-circuit AND instead of a match loop. BF8b was about 25% faster than BF8 on rand2.

In order to test the reliability of our results, we repeated the experiments of HOR in the testing environment of Hume & Sunday (HS) [20]. Table 8 shows speed-ups on genome sequence in both HS and SMART. The table shows concretely that $q = 2$ is inferior and that $q = 4$ and $q = 8$ are very similar. Moreover, the differences between HS and SMART are notable with up to a 12 percentage point difference in the speed-ups.

	m	4	8	16	32	64	128	256
HS	HOR2	1.40	1.46	1.43	1.43	1.44	1.42	1.43
	HOR4	1.97	1.80	1.71	1.69	1.71	1.68	1.68
	HOR8	-	1.85	1.72	1.71	1.71	1.70	1.70
SMART	HOR2	1.38	1.39	1.39	1.37	1.38	1.39	1.38
	HOR4	1.88	1.70	1.62	1.63	1.62	1.61	1.63
	HOR8	-	1.73	1.64	1.63	1.64	1.63	1.64

Table 8. Speed-ups on genome sequence in HS and SMART for HOR with $q \in \{2, 4, 8\}$.

Khan [22] applied q-gram reading inside the match loop. Speed-ups he achieved were much smaller than ours.

6.2 Variations of SSM

The experiments with SSM were run on Intel Core i7-4578U. Algorithms were written in the C programming language and compiled with gcc 5.4.0 using the O3 optimization level. Testing was done in the framework of Hume and Sunday [20]. We used two texts: English (the KJV Bible, 4.0 MB) and DNA (the genome of E. Coli, 4.6 MB) for testing. The texts were taken from the SMART repository. Sets of patterns of lengths 5, 10, and 20 were randomly taken from both texts. Each set contains 200 patterns.

Table 9 lists the alternatives introduced in Section 4. Table 10 shows the running times of the original SSM together with twelve variations. We tested even more alternatives (e.g. SSM-L) but we do not show their times because they were not competitive. Besides SSM we ran experiments with two other reference methods:

id	Alternative
A	p_{m-1} allowed as a pivot
B	shift based on $t_{j+1}t_{j+2}$
C	SSM shift combined with the good suffix shift
F	the least frequent character of P as a pivot
G	a guard test
L	strchr on a pivot
M	P as a pivot
S	shift based on t_{j+1}
U	$p_{m-4} \dots p_{m-1}$ as a pivot
V	$p_{m-8} \dots p_{m-1}$ as a pivot
W	$p_0 \dots p_3$ and $p_{m-4} \dots p_{m-1}$ as a pivot
X	shift based on $t_j t_{j+1}$
Z	strchr in case of an infrequent character in P

Table 9. Summary of alternative features of SSM.

	English			DNA		
m	5	10	20	5	10	20
SSM	89	51	31	206	133	103
SSM-ASC	79	48	30	199	135	105
SSM-AFSC	68	42	28	208	160	144
SSM-ASGC	67	40	26	206	158	142
SSM-USC	59	40	29	105	91	88
SSM-UBC	49	30	18	79	52	38
SSM-UXC	53	30	17	71	45	32
SSM-VBC	–	29	17	–	51	38
SSM-VXC	–	30	17	–	43	32
SSM-MB	63	37	22	166	105	71
SSM-WB	49	29	18	79	51	38
SSM-WX	51	29	16	68	43	31
SSM-WBZ	39	27	23	–	–	–
SBNDM4	31	11	7	38	16	11
FHASH2	36	29	24	149	94	61

Table 10. Running times (in units of 10 ms) of algorithms for sets of 200 patterns.

SBNDM4 [11] with 16-bit reads, and FHASH2 [2]. SBNDM4 is an example of a simple and efficient algorithm, and FHASH2 is an advanced algorithm co-authored by the developer of SSM.

From Table 10 one can see that SSM-WBZ and SSM-WX are the best variations of SSM. SSM-WBZ is the fastest on English data for $m = 5$ and 10. The character 'm' was used as a threshold for SSM-WBZ. If one selects a more frequent threshold, this algorithm becomes slightly faster for $m = 5$ and slightly slower for $m = 20$. SSM-WX is the fastest on English data for $m = 20$ and on DNA data. SSM-WBZ and SSM-WX are 47–70% faster than the original SSM. However, these variations are much slower than SBNDM4 which in turn is slower than recent SIMD-based algorithms like EPSM [13] (the times of EPSM are not shown here).

Table 10 confirms that the alternative X is faster than B on DNA data. Note that no SSM variation was faster than FHASH2 for English patterns of five characters.

7 Conclusions

We applied a guard test on comparison-based string matching algorithms. The test compares multiple characters between the pattern and an alignment window before the match loop. The guard test led to notable speed-ups as shown with experiments. The Brute Force algorithm benefits most from the guard test, and it was faster than other comparison-based algorithms of the test set for short patterns $m \leq 16$. In addition, the variation BF8 was faster than any of the algorithms in the SMART repository for $m = 8$ on binary text.

Our experiments show that most of the new variations of SSM are faster than the original SSM. Although the pivot loop is an inspiring tool, we learned that it hardly can lead to the level of SBNDM4 in efficiency. The obvious reason is that the pivot loop makes two separate accesses to the text in a round: the pivot and the base of shift. A typical skip loop accesses only the base of shift which consists of a single character or a q-gram. However, the positive results with variations testing q-grams (the alternatives U, V, and W) support the usefulness of q-gram guards.

References

1. A. M. Al-Ssulami: *Hybrid string matching algorithm with a pivot.* J. Information Science, 41(1) 2015, pp. 82–88.
2. A. M. Al-Ssulami and H. Mathkour: *Faster string matching based on hashing and bit-parallelism.* Inf. Process. Lett., 123 2017, pp. 51–55.
3. A. M. Al-Ssulami, H. Mathkour, and M. A. Arafah: *Efficient string matching algorithm for searching large DNA and binary texts.* Int. J. Semantic Web Inf. Syst., 13(4) 2017, pp. 198–220.
4. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop 1999, J. Holub and M. Simánek, eds., Prague, Czech Republic, 1999, pp. 16–28.
5. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
6. D. Cantone and S. Faro: *"It's economy, stupid!": Searching for a substring with constant extra-space complexity*, in Proceedings of Third International Conference on Fun with algorithms, P. Ferragina and R. Grossi, eds., Tuscany, Italy, 2004, pp. 118–131.
7. D. Cantone and S. Faro: *Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 589–608.
8. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics.* J. Discrete Algorithms, 28 2014, pp. 73–84.
9. C. Charras, T. Lecroq, and J. Pehoushek: *A very fast string matching algorithm for small alphabets and long patterns*, in CPM 1998: Combinatorial Pattern Matching, M. Farach-Colton, ed., vol. 1448 of Lecture Notes in Computer Science, Piscataway, New Jersey, USA, 1998, Springer, Berlin, Heidelberg, pp. 55–64.
10. S. Deusdado and P. Carvalho: *GRASPm: An efficient algorithm for exact pattern-matching in genomic sequences.* International journal of bioinformatics research and applications, 5(4) 2009, pp. 385–401.
11. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
12. S. Faro: *SMART.* `https://github.com/smart-tool/smart`, 2016, Commit cd7464526d41396e11912c6a681eddb965e17f58. Accessed 12.6.2020.
13. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, 2013, pp. 113–121.

14. S. Faro and T. Lecroq: *Efficient variants of the backward-oracle-matching algorithm.* Int. J. Found. Comput. Sci., 20(6) 2009, pp. 967–984.
15. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
16. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111.
17. C. Hancart: *Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte*, PhD thesis, University Paris 7, 1993.
18. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
19. 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.
20. A. Hume and D. Sunday: *Fast string searching.* Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
21. P. Kalsi, H. Peltola, and J. Tarhio: *Comparison of exact string matching algorithms for biological sequences*, in Bioinformatics Research and Development, Second International Conference, BIRD 2008, Vienna, Austria, July 7-9, 2008, Proceedings, 2008, pp. 417–426.
22. M. A. Khan: *A transformation for optimizing string-matching algorithms for long patterns.* The Computer Journal, 59(12) 2016, pp. 1749–1759.
23. M. O. Külekci: *An empirical analysis of pattern scan order in pattern matching*, in Proceedings of the World Congress on Engineering, WCE 2007, London, UK, 2-4 July, 2007, 2007, pp. 337–341.
24. T. Lecroq: *Fast exact string matching algorithms.* Information Processing Letters, 102(6) 2007, pp. 229–235.
25. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163 2014, pp. 352–360.
26. T. Raita: *Tuning the Boyer-Moore-Horspool string searching algorithm.* Software: Practice and Experience, 22(10) 1992, pp. 879–884.
27. T. Raita: *On guards and symbol dependencies in substring search.* Software: Practice and Experience, 29(11) 1999, pp. 931–941.
28. A. Sharfuddin and X. Feng: *Improving Boyer-Moore-Horspool using machine-words for comparison*, in Proceedings of the 48th Annual Southeast Regional Conference, 2010, Oxford, MS, USA, April 15-17, 2010, H. C. Cunningham, P. Ruth, and N. A. Kraft, eds., ACM, 2010, pp. 1–5.
29. P. D. Smith: *Experiments with a very fast substring search algorithm.* Software: Practice and Experience, 21(10) 1991, pp. 1065–1074.
30. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
31. J. Tarhio, J. Holub, and E. Giaquinta: *Technology beats algorithms (in exact string matching).* Software: Practice and Experience, 47(12) 2017, pp. 1877–1885.
32. J. Tarhio and B. W. Watson: *Tune-up for the Dead-Zone algorithm*, in Proceedings of the Prague Stringology Conference 2020, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2020, pp. 160–167.
33. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of Dead-Zone single keyword pattern matching*, in Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.

Author Index

Chauve, Cedric, 30

Faro, Simone, 75
Franek, Frantisek, 41

Guth, Ondřej, 1

Hirakawa, Ryo, 53
Hoffmann, Stefan, 61

Inenaga, Shunsuke, 53

Janoušek, Jan, 1

Liut, Michael, 41

Marino, Francesco Pio, 75
Mishna, Marni, 30

Nakashima, Yuto, 53

Pakalén, Waltteri, 16, 90
Paquet-Nadeau, France, 30
Pavone, Arianna, 75
Peltola, Hannu, 16

Scardace, Antonio, 75
Šestáková, Eliška, 1

Takeda, Masayuki, 53
Tarhio, Jorma, 16, 90

Watson, Bruce W., 16, 90

Proceedings of the Prague Stringology Conference 2021
Edited by Jan Holub and Jan Žďárek
Published by: Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science
Prague Stringology Club
Thákurova 9, Praha 6, 160 00, Czech Republic.
First edition.

ISBN 978-80-01-06869-4

URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811

Printed by powerprint s.r.o.
Brandejsovo nám. 1219/1, Praha 6 Suchdol, 165 00, Czech Republic