Proceedings of the Prague Stringology Conference 2023

Edited by Jan Holub and Jan Žďárek

August 2023

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

ISBN 978-80-01-07206-6

Preface

The proceedings in your hands contain a collection of papers presented in the Prague Stringology Conference 2023 (PSC 2023) held on August 28–29, 2023 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 ten papers in this proceedings made the cut and were selected for regular presentation at the conference.

The PSC 2023 was organized in both present and remote form. Speakers we required to present their papers in person. Non-speakers could decide whether to arrive in Prague or to participate remotely.

The Prague Stringology Conference has a long tradition. PSC 2023 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, 2008–2021 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on the 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 at the Czech Technical University in Prague. The goal of the Prague Stringology Club is to study algorithms on strings, sequences, and trees with an emphasis on automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful of invited talks. However, since PSCW'97 the papers and talks are selected by a rigorous peer review process. The objective is not only to present new results in stringology and related areas but also to facilitate personal contacts among the people working on these problems.

We would like to thank all those who had submitted papers for PSC 2023 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 2023. 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 2023

Jan Holub and Solon Pissis

Conference Organisation

Program Committee

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

Organising Committee

Ondřej Guth	Štěpán Plachý	Jan Trávníček, *Co-chair*
Jan Holub, *Co-chair*	Regina Šmídová	Jan Žďárek
Tomáš Pecka		

External Referees

Laurent Bulteau	Lucian Ilie	Takuya Mieno
Diptarama Hendrian		

Table of Contents

Invited Talk

Contributed Talks

Theoretical Perspectives on Algorithmic Choices Made in Programming Languages (*Abstract*)

Cyril Nicaud

Université Paris-Est
Cité Descartes 5
Champs-sur-Marne
77454 Marne-la-Vallée Cedex 2
France
cyril.nicaud@u-pem.fr

All contemporary programming languages offer implementations of classical algorithms and classical data structures such as lists, hash tables, sorting, etc. These are basic building blocks that are used to develop larger programs. Efficient algorithms for dealing with such issues have been known for several decades, since the beginning of computing, often with several variants proposed in the literature. However, there are many surprising choices made by engineers in the implementations of these algorithms in programming languages such as Python, Java, Lua. In this talk, we will investigate several cases where some innovation were introduced, and explain how we can develop a theoretical approach to provide insights on the efficiency of these new ideas.

Cyril Nicaud: Theoretical Perspectives on Algorithmic Choices Made in Programming Languages, p. 1.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

Computing SEQ-IC-LCS of Labeled Graphs

Yuki Yonemoto[1], Yuto Nakashima[2], and Shunsuke Inenaga[2]

[1] Department of Information Science and Technology, Kyushu University
yonemoto.yuuki.240@s.kyushu-u.ac.jp
[2] Department of Informatics, Kyushu University
{nakashima.yuto.003, inenaga.shunsuke.380}@m.kyushu-u.ac.jp

Abstract. We consider labeled directed graphs where each vertex is labeled with a non-empty string. Such labeled graphs are also known as non-linear texts in the literature. In this paper, we introduce a new problem of comparing two given labeled graphs, called the SEQ-IC-LCS problem on labeled graphs. The goal of SEQ-IC-LCS is to compute the the length of the longest common subsequence (LCS) Z of two target labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ that includes some string in the constraint labeled graph $G_3 = (V_3, E_3)$ as its subsequence. Firstly, we consider the case where G_1, G_2 and G_3 are all acyclic, and present algorithms for computing their SEQ-IC-LCS in $O(|E_1||E_2||E_3|)$ time and $O(|V_1||V_2||V_3|)$ space. Secondly, we consider the case where G_1 and G_2 can be cyclic and G_3 is acyclic, and present algorithms for computing their SEQ-IC-LCS in $O(|E_1||E_2||E_3| + |V_1||V_2||V_3| \log |\Sigma|)$ time and $O(|V_1||V_2||V_3|)$ space, where Σ is the alphabet.

1 Introduction

We consider *labeled (directed) graphs* where each vertex is labeled with a non-empty string. Such labeled graphs are also known as *non-linear texts* or *hypertexts* in the literature. Labeled graphs are a natural generalization of usual (unary-path) strings, which can also be regarded as a compact representation of a set of strings. After introduced by the Database community [13], labeled graphs were then considered by the string matching community [21,23,2,22,16,17,10]. Recently, graph representations of large-scale string sets appear in the real-world applications including graph databases [3] and pan-genomics [14]. For instance, *elastic degenerate strings* [18,4,8,19,7], which recently gain attention with bioinformatics background, can be regarded as a special case of labeled graphs. In the best case, a single labeled graph can represent exponentially many strings. Thus, efficient string algorithms that directly work on labeled graphs without expansion are of significance both in theory and in practice.

Shimohira et al. [24] introduced the problem of computing the *longest common subsequence* (*LCS*) of two given labeled graphs, which, to our knowledge, the first and the only known similarity measure of labeled graphs. Since we can easily convert any labeled graph with string labels to an equivalent labeled graph with single character labels (see Figure 1), in what follows, we evaluate the size of a labeled graph by the number of vertices and edges in the (converted) graph. Given two labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, Shimohira et al. [24] showed how to solve the LCS problem on labeled graphs in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space when both G_1 and G_2 are acyclic, and in $O(|E_1||E_2| + |V_1||V_2| \log |\Sigma|)$ time and $O(|V_1||V_2|)$ space when G_1 and G_2 can be cyclic, where Σ is the alphabet. It is noteworthy that their solution is almost optimal since the quadratic $O((|A||B|)^{1-\epsilon})$-time conditional lower bound [1,9] with any constant $\epsilon > 0$ for the LCS problem on two strings A, B also applies to the LCS problem on labeled graphs.

Yuki Yonemoto, Yuto Nakashima, Shunsuke Inenaga: Computing SEQ-IC-LCS of Labeled Graphs, pp. 3–17.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

The *constrained LCS problems* on strings, which were first proposed by Tsai [25] and then extensively studied in the literature [25,12,6,11,15,27,28], use a third input string P which introduces a-priori knowledge of the user to the solution string Z to output. The task here is to compute the longest common subsequence Z of two target strings A and B that meets the condition w.r.t. P, such that

STR-IC-LCS: Z includes (contains) P as substring;
STR-EC-LCS: Z excludes (does not contain) P as substring;
SEQ-IC-LCS: Z includes (contains) P as subsequence;
SEQ-EC-LCS: Z excludes (does not contain) P as subsequence.

While STR-IC-LCS can be solved in $O(|A||B|)$ time [15], the state-of-the-art solutions to STR-EC-LCS and SEQ-IC/EC-LCS run in $O(|A||B||P|)$ time [12,6,11,27].

In this paper, we consider the SEQ-IC-LCS problems on labeled graphs, where the inputs are two target labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, and a constraint text $G_3 = (V_3, E_3)$, and the output is (the length of) a longest common subsequence of G_1 and G_2 such that Z includes as subsequence some string that is represented by G_3. Firstly, we consider the case where G_1, G_2 and G_3 are all acyclic, and present algorithms for computing their SEQ-IC-LCS in $O(|E_1||E_2||E_3|)$ time and $O(|V_1||V_2||V_3|)$ space. Secondly, we consider the case where G_1 and G_2 can be cyclic and G_3 is acyclic, and present algorithms for computing their SEQ-IC-LCS in $O(|E_1||E_2||E_3| + |V_1||V_2||V_3| \log |\Sigma|)$ time and $O(|V_1||V_2||V_3|)$ space, where Σ is the alphabet. The time complexities of our algorithms and related work are summarized in Table 1. Our algorithms for solving SEQ-IC-LCS on labeled graphs are based on the solutions to SEQ-IC-LCS of usual strings proposed by Chin et al. [12]. We emphasize that a faster $o(|E_1||E_2||E_3|)$-time solution to the SEQ-IC-LCS problems implies a major improvement over the SEQ-IC-LCS problems for strings whose best known solutions require cubic time.

A related work is the *regular language constrained sequence alignment* (*RLCSA*) problem [5] for two input strings A and B in which the constraint is given as an NFA. It is known that this problem can be solved in $O(|A||B||V|^3/\log|V|)$ time [20], where $|V|$ denotes the number of states in the NFA.

problem	text-1	text-2	text-3	time complexity	
LCS	string	string	-	$O(\|E_1\|\|E_2\|)$	[26]
	DAG	DAG	-	$O(\|E_1\|\|E_2\|)$	[24]
	graph	graph	-	$O(\|E_1\|\|E_2\| + \|V_1\|\|V_2\| \log \|\Sigma\|)$	[24]
SEQ-IC-LCS	string	string	string	$O(\|E_1\|\|E_2\|\|E_3\|)$	[12,6]
	DAG	DAG	DAG	$O(\|E_1\|\|E_2\|\|E_3\|)$	[this work]
	graph	graph	DAG	$O(\|E_1\|\|E_2\|\|E_3\| + \|V_1\|\|V_2\|\|V_3\| \log \|\Sigma\|)$	[this work]
SEQ-EC-LCS	string	string	string	$O(\|E_1\|\|E_2\|\|E_3\|)$	[11]
STR-IC-LCS	string	string	-	$O(\|E_1\|\|E_2\|)$	[15]
STR-EC-LCS	string	string	-	$O(\|E_1\|\|E_2\|)$	[27]
RLCSA	string	string	NFA	$O(\|E_1\|\|E_2\|\|V_3\|^3/\log\|V_3\|)$	[20]

Table 1. Time complexities of algorithms for labeled graph/usual string comparisons, for inputs text-1 $G_1 = (V_1, E_1)$, text-2 $G_2 = (V_2, E_2)$, and text-3 $G_3 = (V_3, E_3)$. Here, a string input of length n is regarded as a unary path graph $G = (V, E)$ with $|E| = n$.

2 Preliminaries

2.1 Strings and Graphs

Let Σ be an alphabet. An element of Σ^* is called a *string*. The *length* of a string w is denoted by $|w|$. The *empty string*, denoted by ε, is a string of length 0. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For a string $w = xyz$ with $x, y, z \in \Sigma^*$, strings x, y, and z are called a *prefix*, *substring*, and *suffix* of string w, respectively. The ith character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. A string u is a *subsequence* of another string w if $u = \varepsilon$ or there exists a sequence of integers $i_1, \ldots, i_{|u|}$ such that $1 \leq i_1 < \cdots < i_{|u|} \leq |w|$ and $u = w[i_1] \cdots w[i_{|u|}]$.

A *directed graph* G is an ordered pair (V, E) of the set V of *vertices* and the set $E \subseteq V \times V$ of *edges*. The *in-degree* of a vertex v is denoted by $\mathsf{in\_deg}(v) = |\{u \mid (u, v) \in E\}|$. A *path* in a (directed) graph $G = (V, E)$ is a sequence $v_0, \ldots, v_k$ of vertices such that $(v_{i-1}, v_i) \in E$ for every $i = 1, \ldots, k$. A path $\pi = v_0, \ldots, v_k$ in graph G is said to be *left-maximal* if its left-end vertex v_0 has no in-coming edges, and π is said to be *right-maximal* if its right-end vertex v_k has no out-going edges. A path π is said to be *maximal* if π is both left-maximal and right-maximal. For any vertex $v \in V$, let $\mathsf{P}(v)$ denote the set of all paths ending at vertex v, and $\mathsf{LMP}(v)$ denote the set of left-maximal paths ending at v. The set of all paths in $G = (V, E)$ is denoted by $\mathsf{P}(G) = \{\mathsf{P}(v) \mid v \in V\}$. Let $\mathsf{MP}(G)$ denote the set of maximal paths in G.

2.2 Longest Common Subsequence (LCS) of Strings

The *longest common subsequence* (LCS) problem for two given strings A and B is to compute (the length of) the longest string Z that is a subsequences of both A and B. It is well-known that LCS can be solved in $O(|A||B|)$ time by using the following recurrence [26]:

$$C_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0; \\ 1 + C_{i-1,j-1} & \text{if } i, j > 0 \text{ and } x[i] = y[j]; \\ \max(C_{i-1,j}, C_{i,j-1}) & \text{if } i, j > 0 \text{ and } x[i] \neq y[j], \end{cases}$$

where $C_{i,j}$ is the LCS length of $A[1..i]$ and $B[1..j]$.

2.3 SEQ-IC-LCS of Strings

Let A, B, and P be strings. A string Z is said to be an *SEQ-IC-LCS* of two target strings A and B *including* the pattern P if Z is a longest string such that P is a subsequence of Z and that Z is a common subsequence of A and B. Chin et al. [12] solved this problem in $O(|A||B||P|)$ time by using the following recurrence:

$$C_{i,j,k} = \begin{cases} 0 & \text{if } k = 0 \text{ and } (i = 0 \text{ or } j = 0); \\ -\infty & \text{if } k \neq 0 \text{ and } (i = 0 \text{ or } j = 0); \\ C_{i-1,j-1,k-1} + 1 & \text{if } i, j, k > 0 \text{ and } A[i] = B[j] = P[k]; \\ C_{i-1,j-1,k} + 1 & \text{if } i, j > 0 \text{ and } A[i] = B[j] \neq P[k]; \\ \max(C_{i-1,j,k}, C_{i,j-1,k}) & \text{if } i, j > 0 \text{ and } A[i] \neq B[j], \end{cases} \quad (1)$$

where $C_{i,j,k}$ is the SEQ-IC-LCS length of $A[1..i]$, $B[1..j]$, and $P[1..k]$.

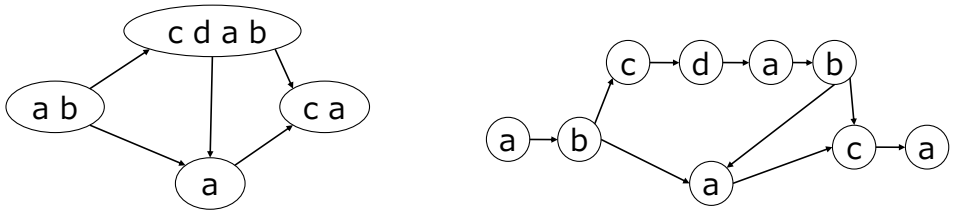


Figure 1. A labeled graph $G = (V, E, L)$ with $L : V \to \Sigma^+$ and its corresponding atomic labeled graph $G' = (V', E', L')$ with $L' : V' \to \Sigma$.

2.4 Labeled Graphs

A *labeled graph* is a directed graph with vertices labeled by strings, namely, it is a directed graph $G = (V, E, L)$ where V is the set of vertices, E is the set of edges, and $L : V \to \Sigma^+$ is a labeling function that maps nodes $v \in V$ to non-empty strings $L(v) \in \Sigma^+$. For a path $\pi = v_0, \ldots, v_k \in \mathsf{P}(G)$, let $L(\pi)$ denote the string spelled out by w, namely $L(\pi) = L(v_0) \cdots L(v_k)$. The size $|G|$ of a labeled graph $G = (V, E, L)$ is $|V| + |E| + \sum_{v \in V} |L(v)|$. Let $\mathsf{Subseq}(G) = \{\mathsf{Subseq}(L(\pi)) \mid \pi \in \mathsf{P}(G)\}$ denote the set of subsequences of a labeled graph $G = (V, E, L)$. For a set $P \in \mathsf{P}(G)$ of paths in G, let $L(P) = \{L(\pi) \mid \pi \in P\}$ denote the set of string labels for the paths in P.

For a labeled graph $G = (V, E, L)$, consider an "atomic" labeled graph $G' = (V', E', L')$ such that $L' : V' \to \Sigma$,

$$V' = \{v_{i,j} \mid L'(v_{i,j}) = L(v_i)[j], v_i \in V, 1 \le j \le |L(v_i)|\}, \text{ and}$$
$$E' = \{(v_{i,|L(v_i)|}, v_{k,1}) \mid (v_i, v_k) \in E\} \cup \{(v_{i,j}, v_{i,j+1}) \mid v_i \in V, 1 \le j < |L(v_i)|\},$$

that is, G' is a labeled graph with each vertex being labeled by a single character, which represents the same set of strings as G. An example is shown in Figure 1. Since $|V'| = \sum_{v \in V} |L(v)|$, $|E'| = |E| + \sum_{v \in V} (|L(v)| - 1)$, and $\sum_{v' \in V'} |L(v')| = \sum_{v \in V} |L(v)|$, we have $|G'| = O(|G|)$. We remark that given G, we can easily construct G' in $O(|G|)$ time. Observe that $\mathsf{Subseq}(G) = \mathsf{Subseq}(G')$ also holds.

In the sequel we only consider atomic labeled graphs where each vertex is labeled with a single character.

2.5 LCS of Acyclic Labeled Graphs

The problem of computing the length of longest common subsequence of two input acyclic labeled graphs is formalized by Shimohira et al. [24] as follows.

Problem 1 (Longest common subsequence problem for acyclic labeled graphs).

Input: Labeled graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$.
Output: The length of a longest string in $\mathsf{Subseq}(G_1) \cap \mathsf{Subseq}(G_2)$.

This problem can be solved in $O(|E_1||E_2|)$ time and $O(|V_1||V_2|)$ space by sorting G_1 and G_2 topologically and using the following recurrence:

$$C'_{i,j} = \begin{cases} 1 + \max(\{C'_{k,\ell} \mid (v_{1,k}, v_{1,i}) \in E_1, (v_{2,\ell}, v_{2,j}) \in E_2\} \cup \{0\}) & \text{if } L_1(v_{1,i}) = L_2(v_{2,j}); \\ \max\left(\begin{array}{l}\{C'_{k,j} \mid (v_{1,k}, v_{1,i}) \in E_1\} \cup \\ \{C'_{i,\ell} \mid (v_{2,\ell}, v_{2,j}) \in E_2\} \cup \{0\}\end{array}\right) & \text{otherwise,} \end{cases} \tag{2}$$

where $v_{1,i}$ and $v_{2,j}$ are respectively the ith and jth vertices of G_1 and in G_2 in topological order, for $1 \leq i \leq |V_1|$ and $1 \leq j \leq |V_2|$, and $C'_{i,j}$ is the length of a longest string in $\mathsf{Subseq}(L_1(\mathsf{P}(v_{1,i}))) \cap \mathsf{Subseq}(L_2(\mathsf{P}(v_{2,j})))$.

2.6 LCS of Cyclic Labeled Graphs

Here we consider a generalized version of Problem 1 where the input labeled graphs G_1 and/or G_2 can be cyclic. In this problem, the output is ∞ if there is a string $s \in \mathsf{Subseq}(G_1) \cap \mathsf{Subseq}(G_2)$ such that $|s| = \infty$, and that is the length of a longest string in $\mathsf{Subseq}(G_1) \cap \mathsf{Subseq}(G_2)$. Shimohira et al. [24] proposed an $O(|E_1||E_2| + |V_1||V_2| \log |\Sigma|)$ time and $O(|V_1||V_2|)$ space algorithm solving this problem. Their algorithm judges whether the output is ∞ by using a balanced tree, and computes the length of the solution by using Equation (2) and the balanced tree if the output is not ∞.

3 The SEQ-IC-LCS Problem for Labeled Graphs

In this paper, we tackle the problem of computing the SEQ-IC-LCS length of three labeled graphs, which formalized as follows:

Problem 2 (SEQ-IC-LCS problem for labeled graphs).

Input: Labeled graphs $G_1 = (V_1, E_1, L_1)$, $G_2 = (V_2, E_2, L_2)$, and $G_3 = (V_3, E_3, L_3)$.
Output: The length of a longest string in the set
$\{z \mid \exists\, q \in L_3(\mathsf{MP}(G_3))$ such that $q \in \mathsf{Subseq}(z)$ and $z \in \mathsf{Subseq}(G_1) \cap \mathsf{Subseq}(G_2)\}$.

Intuitively, Problem 2 asks to compute a longest string z such that z is a subsequence occurring in both G_1 and G_2 and that there exists a string q which corresponds to a maximal path of G_3 and is a subsequence of z.

For a concrete example, see the labeled graphs G_1, G_2 and G_3 of Figure 2. String `cdba` is a common subsequence of G_1 and G_2 and that contains an element `ba` of a maximal path string in $L_3(\mathsf{MP}(G_3))$. Since `cdba` is such a longest string, we ouput the SEQ-IC-LCS length $|\mathtt{cdba}| = 4$ as the solution to this instance.

In the sequel, Section 4 presents our solution to the case where the all input labeled graphs are acyclic, and Section 5 presents our solutions case where G_1 and/or G_2 can be cyclic and G_3 is acyclic.

4 Computing SEQ-IC-LCS of Acyclic Labeled Graphs

In this section, we present our algorithm which solves Problem 2 in the case where all of G_1, G_2 and G_3 are acyclic. The following is our result:

Theorem 3. *Problem 2 with acyclic labeled graphs G_1, G_2 and G_3 can be solved in $O(|E_1||E_2||E_3|)$ time and $O(|V_1||V_2||V_3|)$ space.*

Proof. We perform topological sort to the vertices of G_1, G_2, and G_3 in $O(|E_1| + |E_2| + |E_3|)$ time and $O(|V_1| + |V_2| + |V_3|)$ space. For $1 \leq i \leq |V_1|$, $1 \leq j \leq |V_2|$, and $1 \leq k \leq |V_3|$, let $v_{1,i}$, $v_{2,j}$, $v_{3,k}$ denote the ith, jth, and kth vertices in G_1, G_2, and G_3 in topological order, respectively. Let

$$\mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k}) = \left\{ z \;\middle|\; \begin{array}{l} \exists q \in L_3(\mathsf{LMP}(v_{3,k})) \text{ such that } q \in \mathsf{Subseq}(z) \\ \text{and } z \in \mathsf{Subseq}(L_1(\mathsf{P}(v_{1,i}))) \cap \mathsf{Subseq}(L_2(\mathsf{P}(v_{2,j}))) \end{array} \right\}$$

be the set of candidates of SEQ-IC-LCS strings for the maximal induced graphs of G_1, G_2, and G_3 whose sinks are $v_{1,i}$, $v_{2,j}$, and $v_{3,k}$, respectively. Let $D_{i,j,k}$ denote the length of a longest string in $\mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k})$. The solution to Problem 2 (the SEQ-IC-LCS length) is the maximum value of $D_{i,j,k}$ for which $v_{3,k}$ does not have out-going edges (i.e. $v_{3,k}$ is the end of a maximal path in G_3).

When $k = 0$, then the problem is equivalent to Problem 1 of computing SEQ-IC-LCS of strings. In that follows, we show how to compute $D_{i,j,k}$ for $k > 0$:

1. If $L_1(v_{1,i}) = L_2(v_{2,j}) = L_3(v_{3,k})$, there are three cases to consider:
   (a) If $v_{1,i}$ does not have in-coming edges or $v_{2,j}$ does not have in-coming edges, and if $v_{3,k}$ does not have in-coming edges (i.e., $\mathsf{in\_deg}(v_{1,i}) = \mathsf{in\_deg}(v_{3,k}) = 0$, or $\mathsf{in\_deg}(v_{2,j}) = \mathsf{in\_deg}(v_{3,k}) = 0$), then clearly $D_{i,j,k} = 1$.
   (b) If $v_{1,i}$ does not have in-coming edges or $v_{2,j}$ does not have in-coming edges, and if $v_{3,k}$ has some in-coming edge(s) (i.e., $\mathsf{in\_deg}(v_{1,i}) = 0$ and $\mathsf{in\_deg}(v_{3,k}) \geq 1$, or $\mathsf{in\_deg}(v_{2,j}) = 0$ and $\mathsf{in\_deg}(v_{3,k}) \geq 1$), then clearly $D_{i,j,k} = -\infty$.
   (c) If both $v_{1,i}$ and $v_{2,j}$ have some in-coming edge(s) and $v_{3,k}$ does not have in-coming edges (i.e., $\mathsf{in\_deg}(v_{1,i}) \geq 1$, $\mathsf{in\_deg}(v_{2,j}) \geq 1$, and $\mathsf{in\_deg}(v_{3,k}) = 0$), then let $v_{1,x}$ and $v_{2,y}$ be any nodes s.t. $(v_{1,x}, v_{1,i}) \in E_1$, and $(v_{2,y}, v_{2,j}) \in E_2$, respectively. Let s be a longest string in $\mathsf{Subseq}(L_1(\mathsf{P}(v_{1,i}))) \cap \mathsf{Subseq}(L_2(\mathsf{P}(v_{2,j})))$. Assume on the contrary that there exists a string $t \in \mathsf{Subseq}(L_1(\mathsf{P}(v_{1,x}))) \cap \mathsf{Subseq}(L_2(\mathsf{P}(v_{2,y})))$ such that $|t| > |s| - 1$. This contradicts that s is a longest common subsequence of $L_1(\mathsf{P}(v_{1,i}))$ and $L_2(\mathsf{P}(v_{2,j}))$, since $L_1(v_{1,i}) = L_2(v_{2,j})$. Hence $|t| \leq |s| - 1$. If $v_{1,x}$ and $v_{2,y}$ are vertices satisfying $C'_{x,y,0} = |s| - 1$, then $C'_{i,j,k} = C'_{x,y,0} + 1$. Note that such nodes $v_{1,x}$ and $v_{2,y}$ always exist.
   (d) Otherwise (all $v_{1,i}$, $v_{2,j}$, and $v_{3,k}$ have some in-coming edge(s)), let $v_{1,x}$, $v_{2,y}$ and $v_{3,z}$ be any nodes s.t. $(v_{1,x}, v_{1,i}) \in E_1$, $(v_{2,y}, v_{2,j}) \in E_2$ and $(v_{3,z}, v_{3,k}) \in E_3$, respectively. Let s be a longest string in $\mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k})$. Assume on the contrary that there exists a string $t \in \mathsf{S}_{\mathrm{IC}}(v_{1,x}, v_{2,y}, v_{3,z})$ such that $|t| > |s| - 1$. This contradicts that s is a SEQ-IC-LCS of $L_1(\mathsf{P}(v_{1,i}))$, $L_2(\mathsf{P}(v_{2,j}))$ and $L_3(\mathsf{LMP}(v_{3,k}))$, since $L_1(v_{1,i}) = L_2(v_{2,j}) = L_3(v_{3,k})$. Hence $|t| \leq |s| - 1$. If $v_{1,x}$, $v_{2,y}$ and $v_{3,z}$ are vertices satisfying $D_{x,y,z} = |s| - 1$, then $D_{i,j,k} = D_{x,y,z} + 1$. Note that such nodes $v_{1,x}$, $v_{2,y}$ and $v_{3,z}$ always exist.
2. If $L_1(v_{1,i}) = L_2(v_{2,j}) \neq L_3(v_{3,k})$, there are two cases to consider:
   (a) If $v_{1,i}$ does not have in-coming edges or $v_{2,j}$ does not have-incoming edges (i.e., $\mathsf{in\_deg}(v_{1,i}) = 0$ or $\mathsf{in\_deg}(v_{2,j}) = 0$), then clearly $D_{i,j,k}$ does not exist and let $D_{i,j,k} = -\infty$.
   (b) Otherwise (both $v_{1,i}$ and $v_{2,j}$ have in-coming edge(s)), let $v_{1,x}$ and $v_{2,y}$ be any nodes s.t. $(v_{1,x}, v_{1,i}) \in E_1$ and $(v_{2,y}, v_{2,j}) \in E_2$, respectively. Let s be a longest string in $\mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k})$. Assume on the contrary that there exists a string $t \in \mathsf{S}_{\mathrm{IC}}(v_{1,x}, v_{2,y}, v_{3,k})$ such that $|t| > |s| - 1$. This contradicts that s is a SEQ-IC-LCS of $L_1(\mathsf{P}(v_{1,i}))$, $L_2(\mathsf{P}(v_{2,j}))$ and $L_3(\mathsf{LMP}(v_{3,k}))$, since $L_1(v_{1,i}) = L_2(v_{2,j})$. Hence $|t| \leq |s| - 1$. If $v_{1,x}$, $v_{2,y}$ and $v_{3,k}$ are vertices satisfying $D_{x,y,k} = |s| - 1$, then $D_{i,j,k} = D_{x,y,k} + 1$. Note that such nodes $v_{1,x}$, $v_{2,y}$ and $v_{3,k}$ always exist.
3. If $L_1(v_{1,i}) \neq L_2(v_{2,j})$, there are two cases to consider:
   (a) If $v_{1,i}$ does not have in-coming edges and $v_{2,j}$ does not have in-coming edges (i.e., $\mathsf{in\_deg}(v_{1,i}) = \mathsf{in\_deg}(v_{2,j}) = 0$), then clearly $D_{i,j,k}$ does not exist and let $D_{i,j,k} = -\infty$.
   (b) Otherwise ($v_{1,i}$ has some in-coming edge(s) or $v_{2,j}$ has some in-coming edge(s)), let $v_{1,x}$ and $v_{2,y}$ be any nodes such that $(v_{1,x}, v_{1,i}) \in E_1$ and $(v_{2,y}, v_{2,j}) \in E_2$, respectively. Let s be a $\mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k})$. Assume on the contrary that there

exists a string $t \in \mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k})$ such that $|t| > |s|$. This contradicts that s is a SEQ-IC-LCS of $L_1(\mathsf{P}(v_{1,i}))$, $L_2(\mathsf{P}(v_{2,j}))$ and $L_3(\mathsf{LMP}(v_{3,k}))$, since $\mathsf{S}_{\mathrm{IC}}(v_{1,x}, v_{2,y}, v_{3,k}) \subseteq \mathsf{S}_{\mathrm{IC}}(v_{1,i}, v_{2,j}, v_{3,k})$. Hence $|t| \leq |s|$. If $v_{1,x}$ is a vertex satisfying $D_{x,j,k} = |z|$, then $D_{i,j,k} = D_{x,j,k}$. Similarly, if $v_{2,y}$ is a vertex satisfying $D_{i,y,k} = |s|$, then $D_{i,j,k} = D_{i,y,k}$. Note that such node $v_{1,x}$ or $v_{2,y}$ always exists.

Consequently we obtain the following recurrence:

$$D_{i,j,k} = \begin{cases} \text{Recurrence in Equation (2)} & \text{if } k = 0; \\ 1 + \max\left(\left\{ D_{x,y,z} \;\middle|\; \begin{array}{l}(v_{1,x}, v_{1,i}) \in E_1,\\ (v_{2,y}, v_{2,j}) \in E_2,\\ (v_{3,z}, v_{3,k}) \in E_3,\\ \text{ or } z = 0\end{array} \right\} \cup \{\gamma\}\right) & \begin{array}{l}\text{if } k > 0 \text{ and}\\ L_1(v_{1,i}) = L_2(v_{2,j})\\ = L_3(v_{3,k});\end{array} \\ \max\left(\left\{ 1 + D_{x,y,k} \;\middle|\; \begin{array}{l}(v_{1,x}, v_{1,i}) \in E_1,\\ (v_{2,y}, v_{2,j}) \in E_2\end{array} \right\} \cup \{-\infty\}\right) & \begin{array}{l}\text{if } k > 0 \text{ and}\\ L_1(v_{1,i}) = L_2(v_{2,j})\\ \neq L_3(v_{3,k});\end{array} \\ \max\left(\begin{array}{l}\{D_{x,j,k} \mid (v_{1,x}, v_{1,i}) \in E_1\} \cup\\ \{D_{i,y,k} \mid (v_{2,y}, v_{2,j}) \in E_2\} \cup \{-\infty\}\end{array}\right) & \text{otherwise.} \end{cases} \tag{3}$$

where

$$\gamma = \begin{cases} 0 & \text{if } v_{1,i} \text{ does not have in-coming edges at all or } v_{2,j} \text{ does not have} \\ & \text{in-coming edges at all, and } v_{3,k} \text{ does not have in-coming edges;} \\ -\infty & \text{otherwise.} \end{cases}$$

We compute $D_{i,j,k}$ for all $1 \leq i \leq |V_1|$, $1 \leq j \leq |V_2|$ and $0 \leq k \leq |V_3|$, using a dynamic programming table of size $O(|V_1||V_2||V_3|)$.

Below we analyze the time complexity for computing $D_{i,j,k}$ with the recurrence:

- The first case with Equation (2) takes $O(|E_1||E_2|)$ time (Section 2.5).
- Second, let us analyze the time cost for computing

$$M_{i,j,k} = \max\{D_{x,y,z} \mid (v_{1,x}, v_{1,i}) \in E_1, (v_{2,y}, v_{2,j}) \in E_2, (v_{3,z}, v_{3,k}) \in E_3, \text{ or } z = 0\}$$

  in the second case of the recurrence for all i, j, k. For each fixed pair of $(v_{1,x}, v_{1,i}) \in E_1$ and $(v_{2,y}, v_{2,j}) \in E_2$, we refer the value of $D_{x,y,z}$ for all $1 \leq z < k$ such that $(v_{3,z}, v_{3,k}) \in E_3$, in $O(|E_3|)$ time. For each fixed $(v_{1,x}, v_{1,i}) \in E_1$, we refer the value of $D_{x,y,z}$ for all $1 \leq y < j$ such that $(v_{2,y}, v_{2,j}) \in E_2$ and all $1 \leq z < k$ such that $(v_{3,z}, v_{3,k}) \in E_3$, in $O(|E_2||E_3|)$ time. Therefore, the total time complexity for computing all $M_{i,j,k}$ for all i, j, k is $O(|E_1||E_2||E_3|)$.
- Third, let us analyze the time cost for computing

$$M'_{i,j,k} = \max\{D_{x,y,k} \mid (v_{1,x}, v_{1,i}) \in E_1, (v_{2,y}, v_{2,j}) \in E_2\}$$

  in the third case of the recurrence for all i, j, k. For each fixed pair of $(v_{1,x}, v_{1,i}) \in E_1$ and $(v_{2,y}, v_{2,j}) \in E_2$, we refer the value of $D_{x,y,k}$ for all $1 \leq k \leq |V_3|$, in $O(|V_3|)$ time. For each fixed $(v_{1,x}, v_{1,i}) \in E_1$, we refer the value of $D_{x,y,k}$ for all $1 \leq y < j$ such that $(v_{2,y}, v_{2,j}) \in E_2$ and all $1 \leq k \leq |V_3|$, in $O(|E_2||V_3|)$ time. Therefore, the total time complexity for computing $M'_{i,j,k}$ for all i, j, k is $O(|E_1||E_2||V_3|) \subseteq O(|E_1||E_2||E_3|)$.

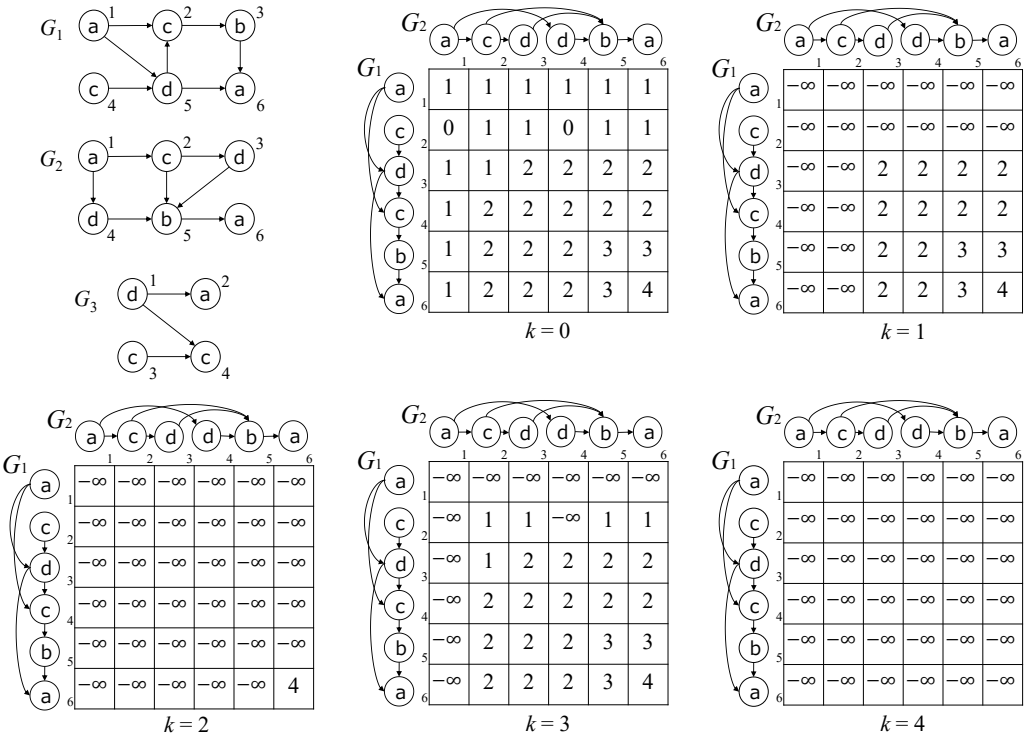


Figure 2. Example of dynamic programming table D for computing the SEQ-IC-LCS length of acyclic labeled graphs G_1, G_2 and G_3. Each vertex is annotated with its topological order. In this example, $v_{3,2}$ and $v_{3,4}$ with $k \in \{2, 4\}$ in G_3 are vertices with no out-going edges. The maximum value of $D_{i,j,k}$ with $k \in \{2, 4\}$ is $D_{6,6,2} = 4$, and the corresponding SEQ-IC-LCS is `cdba` of length 4.

– Fourth, let us analyze the time cost for computing

$$M''_{i,j,k} = \max\{D_{x,j,k}, D_{i,y,k} \mid (v_{1,x}, v_{1,i}) \in E_1, (v_{2,y}, v_{2,j}) \in E_2\}$$

in the fourth case of the recurrence for all i, j, k. For each fixed $(v_{1,x}, v_{1,i}) \in E_1$, we refer the value of $D_{x,j,k}$ for all $1 \leq j \leq |V_2|$ and all $1 \leq k \leq |V_3|$ in $O(|V_2||V_3|)$ time. Similarly, for each fixed $(v_{2,y}, v_{2,j}) \in E_2$, we refer the value of $D_{i,y,k}$ for all $1 \leq i \leq |V_1|$ and all $1 \leq k \leq |V_3|$ in $O(|V_1||V_3|)$ time. Therefore, the total time cost for computing $M''_{i,j,k}$ for all i, j, k is $O(|V_3|(|V_2||E_1| + |V_1||E_2|)) \subseteq O(|E_1||E_2||E_3|)$.

Thus the total time complexity is $O(|E_1||E_2||E_3|)$. □

An example of computing $D_{i,j,k}$ using dynamic programming is show in Figure 2. We remark that the recurrence in Equation (3) is a natural generalization of the recurrence in Equation (1) for computing the SEQ-IC-LCS length of given two strings.

5 Computing SEQ-IC-LCS of Cyclic Labeled Graphs

In this section, we present an algorithm to solve Problem 2 in case where G_1 and/or G_2 can be cyclic and G_3 is acyclic. We output ∞ if the set of output candidates in Problem 2 contains a string of infinite length, and outputs the (finite) SEQ-IC-LCS length otherwise.

To deal with cyclic graphs, we follow the approach by Shimohira et al. [24] which transforms a cyclic labeled graph $G = (V, E, L)$ into an acyclic labeled graph $\hat{G} = (\hat{V}, \hat{E}, \hat{L})$ based on the strongly connected components.

For each vertex $v \in V$, let $[v]$ denote the set of vertices that belong to the same strongly connected component. Formally, $\hat{G} = (\hat{V}, \hat{E}, \hat{L})$ is defined by

$$\hat{V} = \{[v] \mid v \in V\},$$
$$\hat{E} = \{([v],[u]) \mid [v] \neq [u], (\hat{v},\hat{u}) \in E \text{ for some } \hat{v} \in [v],\ \hat{u} \in [u]\} \cup \{(v,v) \mid |[v]| \geq 2\},$$

and $\hat{L}([v]) = \{L(v) \mid v \in [v]\} \subseteq \Sigma$. We regard each $[v]$ as a single vertex that is contracted from vertices in $[v]$. Observe that $\mathsf{Subseq}(\hat{G}) = \mathsf{Subseq}(G)$. An example of transformed acyclic labeled graphs is shown in Figure 3.

It is possible that a vertex $\hat{v} \in \hat{V}$ in the transformed graph $\hat{G}$ has a self-loop. We regard that a self-loop $(\hat{v}, \hat{v})$ is also an in-coming edge of vertex $\hat{v}$. We say that vertex $\hat{v}$ does not have in-coming edges *at all*, if $\hat{v}$ does not have in-coming edges from *any* vertex in $\hat{V}$ (including $\hat{v}$).

Our main result of this section follows:

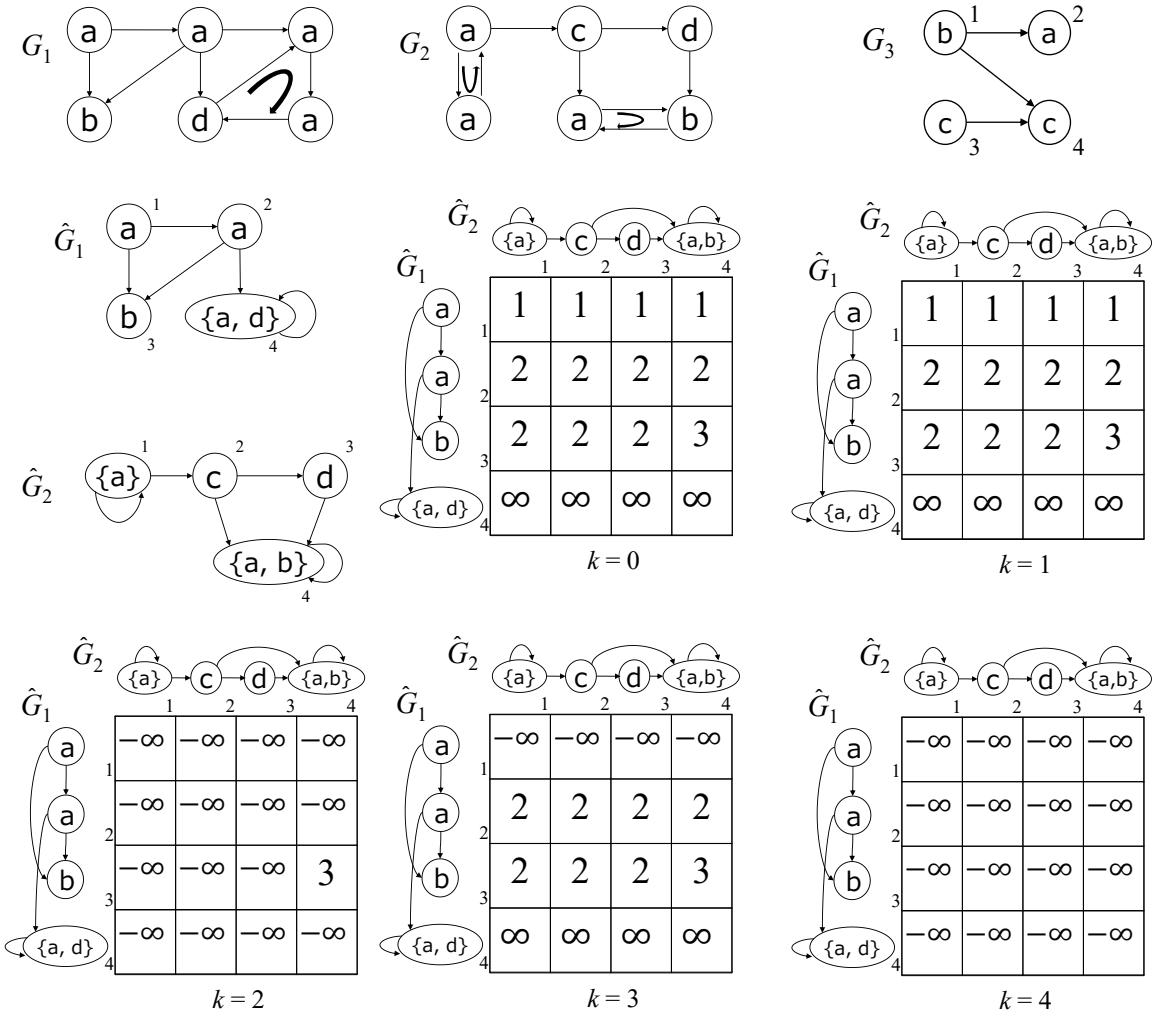


Figure 3. Example of dynamic programming table $\hat{D}$ for computing the SEQ-IC-LCS length of cyclic labeled graphs G_1 and G_2, and acyclic labeled graph G_3. $\hat{G}_1$ and $\hat{G}_2$ are the labeled graphs which are transformed from G_1 and G_2 by grouping vertices into strongly connected components. Each vertex is annotated with its topological order. In this example, $v_{3,2}$ and $v_{3,4}$ with $k \in \{2,4\}$ in G_3 are vertices with no out-going edges. The maximum value of $\hat{D}_{i,j,k}$ with $k \in \{2,4\}$ is $\hat{D}_{4,3,2} = 3$, and the corresponding SEQ-IC-LCS is `aab` of length 3.

Theorem 4. *Problem 2 with G_1 and/or G_2 cyclic and G_3 acyclic can be solved in $O(|E_1||E_2||E_3| + |V_1||V_2||V_3| \log |\Sigma|)$ time and $O(|V_1||V_2||V_3|)$ space.*

Proof. We first transform cyclic labeled graphs G_1 and G_2 into corresponding acyclic labeled graphs $\hat{G}_1$ and $\hat{G}_2$, as described previously. For $1 \leq i \leq |\hat{V}_1|$ and $1 \leq j \leq |\hat{V}_2|$, let $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ respectively denote the ith and jth vertices in $\hat{G}_1$ and $\hat{G}_2$ in topological order. Let $v_{3,k}$ denote the k-th vertex in topological ordering in G_3 for $1 \leq k \leq |V_3|$.

Let

$$\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k}) = \left\{ z \;\middle|\; \begin{array}{l} \exists q \in L_3(\mathsf{MP}(v_{3,k})) \text{ such that } q \in \mathsf{Subseq}(z) \\ \text{and } z \in \mathsf{Subseq}(\hat{L}_1(\mathsf{P}(\hat{v}_{1,i}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{P}(\hat{v}_{2,j}))) \end{array} \right\}.$$

Let $\hat{D}_{i,j,k}$ denote the length of a longest string in $\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k})$. For convenience, we let $\hat{D}_{i,j,k} = -\infty$ if $\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k}) = \emptyset$. The solution to Problem 2 (the SEQ-IC-LCS length) is the maximum value of $\hat{D}_{i,j,k}$ for which $v_{3,k}$ has no out-going edges (i.e. $v_{3,k}$ is the end of a maximal path in G_3).

$\hat{D}_{i,j,k}$ can be computed as follows:

1. If both $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ are cyclic vertices (i.e. $|[\hat{v}_{1,i}]| \geq 2$ and $|[\hat{v}_{2,j}]| \geq 2$), then remark that both $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ have some self-loop(s). There are four cases to consider:
   (a) If $k = 0$, there are two cases to consider:
      i. If $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$, then clearly $\hat{D}_{i,j,k} = \infty$.
      ii. Otherwise, there are two cases to consider:
         A. If the in-coming edges of $\hat{v}_{1,i}$ are $\hat{v}_{2,j}$ only self-loops, then clearly $\hat{D}_{i,j,k} = 0$.
         B. Otherwise ($\hat{v}_{1,i}$ has some in-coming edge(s) other than self-loops, or $\hat{v}_{2,j}$ has some in-coming edge(s) other than self-loops), let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively. Let s be a longest string in the set $\mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j})))$. Assume on the contrary that there is a string $t \in \mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,x}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j})))$ such that $|t| > |s|$. This contradicts that s is a longest common subsequence of $\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))$ and $\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))$, since $\mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,x}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))) \subseteq \mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j})))$. Hence $|t| \leq |s|$. If $\hat{v}_{1,x}$ is a vertex satisfying $\hat{D}_{x,j,k} = |s|$, then $\hat{D}_{i,j,k} = \hat{D}_{x,j,k}$. Similarly, if $\hat{v}_{2,y}$ is a vertex satisfying $\hat{D}_{i,y,k} = |s|$, then $\hat{D}_{i,j,k} = \hat{D}_{i,y,k}$. Note that such $\hat{v}_{1,x}$ or $\hat{v}_{2,y}$ always exists.
   (b) If $k > 0$ and $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap \{L_3(v_{3,k})\} \neq \emptyset$, there are two cases to consider:
      i. If $v_{3,k}$ has no in-coming edges, let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively (these edges may be self-loops). If $\hat{D}_{x,y,0} = -\infty$ for all $1 \leq x < i$ and $1 \leq y < j$, then clearly $\hat{D}_{i,j,k} = -\infty$. Otherwise, clearly $\hat{D}_{i,j,k} = \infty$.
      ii. Otherwise ($v_{3,k}$ has some in-coming edge(s)), let $\hat{v}_{1,x}$, $\hat{v}_{2,y}$ and $v_{3,z}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$, $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$ and $(v_{3,z}, v_{3,k}) \in E_3$, respectively (the first two edges may be self-loops). If $\hat{D}_{x,y,z} = -\infty$ for all $1 \leq x < i$ and $1 \leq y < j$, then clearly $\hat{D}_{i,j,k} = -\infty$. Otherwise, $\hat{D}_{i,j,k} = \infty$.
   (c) If $k > 0$ and $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap \{L_3(v_{3,k})\} = \emptyset$ and $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$, there are two cases to consider:

i. If the in-coming edges of $\hat{v}_{1,i}$ are $\hat{v}_{2,j}$ only self-loops, then clearly $\hat{D}_{i,j,k} = -\infty$.
ii. Otherwise ($\hat{v}_{1,i}$ has some in-coming edge(s) other than self-loops, or $\hat{v}_{2,j}$ has some in-coming edge(s) other than self-loops), let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively. If all $\hat{D}_{x,y,k} = -\infty$, then clearly $\hat{D}_{i,j,k} = -\infty$. Otherwise, clearly $\hat{D}_{i,j,k} = \infty$.

(d) If $k > 0$ and $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) = \emptyset$, there are two cases to consider:
i. If the in-coming edges of $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ are only self-loops, then clearly $\hat{D}_{i,j,k} = -\infty$.
ii. Otherwise ($\hat{v}_{1,i}$ has some in-coming edge(s) other than self-loops, or $\hat{v}_{2,j}$ has some in-coming edge(s) other than self-loops), let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively. Let s be a longest string in $\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k})$. Assume on the contrary that there exists a string $t \in \hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k})$ such that $|t| > |s|$. This contradicts that s is a SEQ-IC-LCS of $\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))$, $\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))$ and $L_3(\mathsf{MP}(v_{3,k}))$, since $\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,x}, \hat{v}_{2,y}, v_{3,k}) \subseteq \hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k})$. Hence $|t| \leq |s|$. If $\hat{v}_{1,x}$ is a vertex satisfying $\hat{D}_{x,j,k} = |z|$, then $\hat{D}_{i,j,k} = \hat{D}_{x,j,k}$. Similarly, if $\hat{v}_{2,y}$ is a vertex satisfying $\hat{D}_{i,y,k} = |s|$, then $\hat{D}_{i,j,k} = \hat{D}_{i,y,k}$. Note that such $\hat{v}_{1,x}$ or $\hat{v}_{2,y}$ always exists.

2. Otherwise ($v_{1,i}$ is not a cyclic vertex and/or $v_{2,j}$ is not a cyclic vertex), there are four cases to consider:

(a) If $k = 0$, there are two cases to consider:
i. If $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$, there are two cases to consider:
A. If $\hat{v}_{1,i}$ does not have in-coming edges at all or $\hat{v}_{2,j}$ does not have in-coming edges at all, then clearly $\hat{D}_{i,j,k} = 1$.
B. Otherwise (both $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ have some in-coming edge(s) including self-loops), let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively. Let s be a longest string in the set $\mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j})))$. Assume on the contrary that there is a string $t \in \mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,x}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,y})))$ such that $|t| > |s| - 1$. This contradicts that s is a longest common subsequence of $\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))$ and $\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))$, since $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$. Hence $|t| \leq |s| - 1$. If $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ are vertices satisfying $\hat{D}_{x,y,k} = |s| - 1$, then $\hat{D}_{i,j,k} = \hat{D}_{x,y,k} + 1$. Note that such $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ always exist.
ii. Otherwise, then this case is the same as Case 1(a)ii.

(b) If $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap \{L_3(v_{3,k})\} \neq \emptyset$, there are three cases to consider:
i. If $\hat{v}_{1,i}$ does not have in-coming edges at all or $\hat{v}_{2,j}$ does not have in-coming edges at all, and if $v_{3,k}$ does not have in-coming edges, then clearly $\hat{D}_{i,j,k} = 1$.
ii. If $\hat{v}_{1,i}$ does not have in-coming edges at all or $\hat{v}_{2,j}$ does not have in-coming edge at all, and if $v_{3,k}$ has some in-coming edge(s), then clearly $\hat{D}_{i,j,k} = -\infty$.
iii. If both $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ have some in-coming edge(s) including self-loops and $v_{3,k}$ does not have in-coming edges, let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively. Let s be a longest string in the set $\mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j})))$. Assume on the contrary that there exists a string $t \in \mathsf{Subseq}(\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,x}))) \cap \mathsf{Subseq}(\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,y})))$ such that $|t| > |s| - 1$. This contradicts that s is

a longest common subsequence of $\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))$ and $\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))$, since $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$. Hence $|t| \leq |s| - 1$. If $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ are vertices satisfying $\hat{D}_{x,y,0} = |s| - 1$, then $\hat{D}_{i,j,k} = \hat{D}_{x,y,0} + 1$. Note that such $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ always exist.

iv. Otherwise (all $\hat{v}_{1,i}$, $\hat{v}_{2,j}$, and $\hat{v}_{3,k}$ have some in-coming edge(s) including self-loops), let $\hat{v}_{1,x}$, $\hat{v}_{2,y}$ and $v_{3,z}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$, $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, and $(v_{3,z}, v_{3,k}) \in E_3$, respectively. Let s be a longest string in $\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k})$. Assume on the contrary that there exists a string $t \in \hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,x}, \hat{v}_{2,y}, v_{3,z})$ such that $|t| > |s| - 1$. This contradicts that s is a SEQ-IC-LCS of $\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))$, $\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))$ and $L_3(\mathsf{MP}(v_{3,k}))$, since $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap L_3(v_{3,k}) \neq \emptyset$. Hence $|t| \leq |s| - 1$. If $\hat{v}_{1,x}$, $\hat{v}_{2,y}$ and $v_{3,z}$ are vertices satisfying $\hat{D}_{x,y,z} = |s| - 1$, then $\hat{D}_{i,j,k} = \hat{D}_{x,y,z} + 1$. Note that such $\hat{v}_{1,x}$, $\hat{v}_{2,y}$ and $v_{3,z}$ always exist.

(c) If $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap \{L_3(v_{3,k})\} = \emptyset$ and $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$, there are two cases to consider:

i. If $\hat{v}_{1,i}$ does not have in-coming edges at all or $\hat{v}_{2,j}$ does not have in-coming edges at all, then clearly $\hat{D}_{i,j,k} = -\infty$.

ii. Otherwise (both $\hat{v}_{1,i}$ and $\hat{v}_{2,j}$ have some in-coming edges including self-loops), let $\hat{v}_{1,x}$ and $\hat{v}_{2,y}$ be any nodes such that $(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1$ and $(\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2$, respectively. Let s be a longest string in $\hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,i}, \hat{v}_{2,j}, v_{3,k})$. Assume on the contrary that there exists a string $t \in \hat{\mathsf{S}}_{\mathrm{IC}}(\hat{v}_{1,x}, \hat{v}_{2,y}, v_{3,k})$ such that $|t| > |s| - 1$. This contradicts that s is a SEQ-IC-LCS of $\hat{L}_1(\mathsf{LMP}(\hat{v}_{1,i}))$, $\hat{L}_2(\mathsf{LMP}(\hat{v}_{2,j}))$ and $L_3(\mathsf{MP}(v_{3,k}))$, since $\hat{L}_1(\hat{v}_{1,i}] \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset$. Hence $|t| \leq |s| - 1$. If $\hat{v}_{1,x}$, $\hat{v}_{2,y}$ and $v_{3,k}$ are vertices satisfying $\hat{D}_{x,y,k} = |s| - 1$, then $\hat{D}_{i,j,k} = \hat{D}_{x,y,k} + 1$. Note that such $\hat{v}_{1,x}$, $\hat{v}_{2,y}$ and $v_{3,k}$ always exist.

(d) If $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) = \emptyset$, then this case is the same as Case 1d.

The above arguments lead us to the following recurrence:

$$\hat{D}_{i,j,k} = \begin{cases} \delta + \max\left(\left\{ \hat{D}_{x,y,k} \;\middle|\; \begin{array}{l}(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1,\\ (\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2\end{array} \right\} \cup \{0\}\right) & \text{if } k = 0 \text{ and } \hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset; \\ \max\left(\begin{array}{l}\{\hat{D}_{x,j,k} \mid (\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1\} \cup \\ \{\hat{D}_{i,y,k} \mid (\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2\} \cup \{0\}\end{array}\right) & \text{if } k = 0 \text{ and } \hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) = \emptyset; \\ \delta + \max\left(\left\{ \hat{D}_{x,y,z} \;\middle|\; \begin{array}{l}(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1,\\ (\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2,\\ (v_{3,z}, v_{3,k}) \in E_3\\ \text{or } z = 0\end{array} \right\} \cup \{\gamma\}\right) & \text{if } k > 0 \text{ and } \hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap \{L_3([v_{3,k}])\} \neq \emptyset; \\ \max\left(\left\{ \delta + \hat{D}_{x,y,k} \;\middle|\; \begin{array}{l}(\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1,\\ (\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2\end{array} \right\} \cup \{-\infty\}\right) & \text{if } k > 0, \; \hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap \{L_3(v_{3,k})\} = \emptyset, \text{ and } \hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \neq \emptyset; \\ \max\left(\begin{array}{l}\{\hat{D}_{x,j,k} \mid (\hat{v}_{1,x}, \hat{v}_{1,i}) \in \hat{E}_1\} \cup \\ \{\hat{D}_{i,y,k} \mid (\hat{v}_{2,y}, \hat{v}_{2,j}) \in \hat{E}_2\} \cup \{-\infty\}\end{array}\right) & \text{otherwise,} \end{cases}$$

where

$$\delta = \begin{cases} \infty & \text{if both } \hat{L}_1(\hat{v}_{1,i}) \text{ and } \hat{L}_2(\hat{v}_{2,j}) \text{ are cyclic vertices;} \\ 1 & \text{otherwise,} \end{cases}$$

$$\gamma = \begin{cases} 0 & \text{if } \hat{v}_{1,i} \text{ does not have in-coming edges at all or } \hat{v}_{2,j} \text{ does not have} \\ & \text{in-coming edges at all, and } v_{3,k} \text{ does not have in-coming edges;} \\ -\infty & \text{otherwise.} \end{cases}$$

In the above recurrence, we use a convention that $\infty + (-\infty) = -\infty$.

We perform preprocessing which transforms G_1 and G_2 into $\hat{G}_1$ and $\hat{G}_2$ in $O(|E_1|+|E_2|)$ time with $O(|V_1|+|V_2|)$ space, based on strongly connected components.

To examine the conditions in the above recurrence, we explicitly construct the intersection of the character labels of the given vertices $\hat{v}_{1,i} \in \hat{V}_1$, $\hat{v}_{2,j} \in \hat{V}_2$, and $\hat{v}_{3,k} \in V_3$ by using balanced trees, as follows:

- Checking whether $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) = \emptyset$ or $\neq \emptyset$: Let Σ_1 and Σ_2 be the sets of characters that appear in G_1 and G_2, respectively. For every node $\hat{v}_{1,i} \in \hat{V}_1$ of the transformed graph $\hat{G}_1$, we build a balanced tree $\mathcal{T}_i$ which consists of the characters in $\hat{L}_1(\hat{v}_i)$. Since the total number of characters in the original graph $G_1 = (V_1, E_1)$ is equal to $|V_1|$, we can build the balanced trees $\mathcal{T}_i$ for all i in a total of $O(|V_1| \log |\Sigma_1|)$ time and $O(|V_1|)$ space. Then, for each fixed $\hat{L}_1(\hat{v}_{1,i}) \in \hat{V}_1$, by using its balanced tree, the intersection $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j})$ can be computed in $O(|V_2| \log |\Sigma_1|)$ time for all $\hat{L}_2(\hat{v}_{2,j}) \in V_2$. Therefore, $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j})$ for all $1 \leq i \leq |\hat{V}_1|$ and $1 \leq j \leq |\hat{V}_2|$ can be computed in $O(|V_1||V_2| \log |\Sigma_1|)$ total time.
- Checking whether $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap L_3(v_{3,k}) = \emptyset$ or $\neq \emptyset$: While computing $\Sigma_{i,j} = \hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j})$ in the above, we also build another balanced tree $\mathcal{T}_{i,j}$ which consists of the characters in $\Sigma_{i,j}$ for every $1 \leq i \leq |\hat{V}_1|$ and $1 \leq j \leq |\hat{V}_2|$. This can be done in $O(|V_1||V_2| \log |\Sigma_1|)$ total time and $O(|V_1||V_2|)$ space. Then, for each fixed $1 \leq i \leq |\hat{V}_1|$ and $1 \leq j \leq |\hat{V}_2|$, $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap L_3(v_{3,k})$ can be computed in a total of $O(|V_3| \log |\Sigma_{i,j}|)$ time. Therefore, $\hat{L}_1(\hat{v}_{1,i}) \cap \hat{L}_2(\hat{v}_{2,j}) \cap L_3(v_{3,k})$ for all $1 \leq i \leq |\hat{V}_1|$, $1 \leq j \leq |\hat{V}_2|$ and, $1 \leq k \leq |V_3|$ can be computed in $O(|V_1||V_2||V_3| \log |\Sigma|)$ time.

Assuming that the above preprocessing for the conditions in the recurrence are all done, we can compute $\hat{D}_{i,j,k}$ for all $1 \leq i \leq |\hat{V}_1|$, $1 \leq j \leq |\hat{V}_2|$ and $1 \leq k \leq |V_3|$ using dynamic programming of size $O(|\hat{V}_1||\hat{V}_2||V_3|)$ in $O(|\hat{E}_1||\hat{E}_2||E_3|)$ time, in a similar way to the acyclic case for Theorem 3.

Overall, the total time complexity is $O(|E_1| + |E_2| + |E_3| + |\hat{V}_1||\hat{V}_2| \log |\Sigma_1| + |\hat{V}_1||\hat{V}_2||V_3| \log |\Sigma| + |\hat{E}_1||\hat{E}_2||E_3|) \subseteq O(|E_1||E_2||E_3| + |V_1||V_2||V_3| \log |\Sigma|)$.

The total space complexity is $O(|V_1||V_2| + |\hat{V}_1||\hat{V}_2||V_3|) \subseteq O(|V_1||V_2||V_3|)$. □

An example of computing $\hat{D}_{i,j,k}$ using dynamic programming is shown in Figure 3.

6 Conclusions and Open Questions

In this paper, we introduced the new problem of computing the SEQ-IC-LCS on labeled graphs. We showed that when the all the input labeled graphs are acyclic, the

problem can be solved in $O(|E_1||E_2||E_3)$ time and $O(|V_1||V_2||V_3|)$ space by a dynamic programming approach. Furthermore, we extend our algorithm to a more general case where the two target labeled graphs can contain cycles, and presented an efficient algorithm that runs in $O(|E_1||E_2||E_3| + |V_1||V_2||V_3| \log |\Sigma|)$ time and $O(|V_1||V_2||V_3|)$ space.

Interesting open questions are whether one can extend the framework of our methods to the other variants STR-IC/EC-LCS and SEQ-EC-LCS of the constrained LCS problems in the case of labeled graph inputs. We believe that SEQ-EC-LCS for labeled graphs can be solved by similar methods to our SEQ-IC-LCS methods, within the same bounds.

References

1. A. Abboud, A. Backurs, and V. V. Williams: *Tight hardness results for LCS and other sequence similarity measures*, in FOCS 2015, 2015, pp. 59–78.
2. A. Amir, M. Lewenstein, and N. Lewenstein: *Pattern matching in hypertext*, in WADS 1997, vol. 1272 of LNCS, 1997, pp. 160–173.
3. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc: *Foundations of modern query languages for graph databases.* ACM Comput. Surv., 50(5) 2017, pp. 68:1–68:40.
4. K. Aoyama, Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Faster online elastic degenerate string matching*, in CPM 2018, vol. 105 of LIPIcs, 2018, pp. 9:1–9:10.
5. A. N. Arslan: *Regular expression constrained sequence alignment.* Journal of Discrete Algorithms, 5(4) 2007, pp. 647–661, Selected papers from Combinatorial Pattern Matching 2005.
6. A. N. Arslan and Ö. Egecioglu: *Algorithms for the constrained longest common subsequence problems.* Int. J. Found. Comput. Sci., 16(6) 2005, pp. 1099–1109.
7. G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone: *Elastic-degenerate string matching via fast matrix multiplication.* SIAM J. Comput., 51(3) 2022, pp. 549–576.
8. G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone: *Approximate pattern matching on elastic-degenerate text.* Theor. Comput. Sci., 812 2020, pp. 109–122.
9. K. Bringmann and M. Künnemann: *Quadratic conditional lower bounds for string problems and dynamic time warping*, in FOCS 2015, 2015, pp. 79–97.
10. M. Cáceres: *Parameterized algorithms for string matching to DAGs: Funnels and beyond*, in CPM 2023, vol. 259 of LIPIcs, 2023, pp. 7:1–7:19.
11. Y.-C. Chen and K.-M. Chao: *On the generalized constrained longest common subsequence problems.* Journal of Combinatorial Optimization, 21(3) Apr 2011, pp. 383–392.
12. F. Y. Chin, A. D. Santis, A. L. Ferrara, N. Ho, and S. Kim: *A simple algorithm for the constrained sequence problems.* Information Processing Letters, 90(4) 2004, pp. 175 – 179.
13. J. Conklin: *Hypertext: An introduction and survey.* IEEE Computer, 20(9) 1987, pp. 17–41.
14. T. C. P.-G. Consortium: *Computational pan-genomics: status, promises and challenges.* Briefings in Bioinformatics, 19(1) 2016, pp. 118–135.
15. S. Deorowicz: *Quadratic-time algorithm for a string constrained lcs problem.* Information Processing Letters, 112(11) 2012, pp. 423 – 426.
16. M. Equi, R. Grossi, V. Mäkinen, and A. I. Tomescu: *On the complexity of string matching for graphs*, in ICALP 2019, vol. 132 of LIPIcs, 2019, pp. 55:1–55:15.
17. M. Equi, V. Mäkinen, and A. I. Tomescu: *Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails*, in SOFSEM 2021, vol. 12607 of Lecture Notes in Computer Science, 2021, pp. 608–622.
18. R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, and L. Versari: *On-line pattern matching on similar texts*, in CPM 2017, vol. 78 of LIPIcs, 2017, pp. 9:1–9:14.
19. C. S. Iliopoulos, R. Kundu, and S. P. Pissis: *Efficient pattern matching in elastic-degenerate strings.* Inf. Comput., 279 2021, p. 104616.

20. G. Kucherov, T. Pinhas, and M. Ziv-Ukelson: *Regular language constrained sequence alignment revisited*, in IWOCA 2011, 2011, pp. 404–415.
21. U. Manber and S. Wu: *Approximate string matching with arbitrary costs for text and hypertext*, in Proc. IAPR, 1992, pp. 22–33.
22. G. Navarro: *Improved approximate pattern matching on hypertext.* Theoretial Computer Science, 237(1–2) 2000, pp. 455–463.
23. K. Park and D. K. Kim: *String matching in hypertext*, in Proc. CPM'95, 1995, pp. 318–329.
24. K. Shimohira, S. Inenaga, H. Bannai, and M. Takeda: *Computing longest common substring/subsequence of non-linear texts*, in PSC 2011, 2011, pp. 197–208.
25. Y.-T. Tsai: *The constrained longest common subsequence problem.* Information Processing Letters, 88(4) 2003, pp. 173 – 176.
26. R. A. Wagner and M. J. Fischer: *The string-to-string correction problem.* J. ACM, 21(1) Jan. 1974, pp. 168–173.
27. L. Wang, X. Wang, Y. Wu, and D. Zhu: *A dynamic programming solution to a generalized LCS problem.* Inf. Process. Lett., 113(19-21) 2013, pp. 723–728.
28. Y. Yonemoto, Y. Nakashima, S. Inenaga, and H. Bannai: *Space-efficient STR-IC-LCS computation*, in SOFSEM 2023, vol. 13878 of LNCS, 2023, pp. 372–384.

Tandem Duplication Parameterized by the Length Difference

Peter Damaschke

Department of Computer Science and Engineering, Chalmers University
41296 Göteborg, Sweden
ptr@chalmers.se

Abstract. A tandem duplication in a string takes a substring and inserts another copy of it right beside it. Given two strings, we want to find a shortest sequence of tandem duplications that transform the shorter string into the longer one, or recognize that no such sequence exists. The problem, in particular with short tandem duplications, is of interest in genomics, and a number of complexity results are known. First we improve a recent simple XP algorithm. However, our main technical contributions are an FPT algorithm, where the parameter is the difference of lengths of the two given strings, and a polynomial kernel.

Keywords: tandem duplication, edit distance, periodic string, ynamic programming, parameterized algorithm, alignment graph

1 Introduction

1.1 Definitions and Problem

In this work we give combinatorial and algorithmic results for a particular problem on strings, i.e., sequences of symbols from an alphabet. For understanding the problem, some basic definitions are needed first.

We sometimes use $|X|$ to denote the length of a string X, that is, the number of symbols in X (where multiple occurrences of a symbol are counted that many times). A string where all symbols are distinct is called *exemplar*. A *square* is a string of the form XX, in other words, a concatenation of two equal strings. A *substring* of X consists of some consecutive symbols of X, that is, we use the term substring in the strict sense.

A *tandem duplication (TD)* transforms a string of the form AXB into $AXXB$, that is, it inserts another copy of a substring X besides the existing occurrence of X. The *TD distance* of a string T from a string S is the minimum number k of TDs needed to transform S into T. We define $k = \infty$ if no such sequence of TDs exists. For clarity we state our problem formally:

Given: two strings S and T, where $|S| \leq |T|$.
Find: a sequence with a minimum number of TDs that turns S into T.

We use $n := |T|$ for the length of the target string T.

Short tandem repeats appear to be a type of mutations of particular interest in genomics. To quote from [10], "Short tandem repeat (STR) ... are abundant throughout the human genome, and specific repeat expansions may be associated with human diseases. ... Thus, the knowledge of the normal repeat ranges of STRs is critically important to determine pathogenicity of observed repeats in known STRs or to discover novel disease-relevant repeat expansions", and from [11], "Very short tandem

Peter Damaschke: Tandem Duplication Parameterized by the Length Difference, pp. 18–29.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

repeats bear substantial genetic, evolutional, and pathological significance in genome analyses." For a given sample of genomic strings we may want to recognize whether some strings result from others by sequences of short TDs, in order to figure out normal and irregular amounts of TDs. Also note that k TDs of length at most some ℓ can expand a string by at most $d \leq k\ell$ symbols.

The computational complexity of computing TD distances is only partly understood. To begin with, NP-hardness has been shown only rather recently: Computing the TD distance is NP-hard even when the alphabet size is 5 [2] or when S is an exemplar string [9]. The problem parameterized by the number k of duplications is fixed-parameter tractable (FPT) for exemplar strings S [9]. The latter paper also mentions a simple XP algorithm for arbitrary strings S and T that runs in $O(n^{2k})$ time. It is based on the operation opposite to TD:

A *contraction* transforms a string of the form $AXXB$ into AXB. In the present paper we call AXB a *contraction result* of $AXXB$.

The aforementioned XP algorithm simply branches at most k times on all possible contractions, whose number is trivially bounded by $n^2/2$. Hence the number of different sequences of contractions is at most $(n^2/2)^k$.

1.2 Contributions

First we improve the XP algorithm from [9]. By using some combinatorics of periods in strings, the time is reduced by essentially a factor n^k.

In the main part of the paper we first present an FPT algorithm for minimizing the number of TDs, parameterized by the length difference of T and S. This parameter d may be practically motivated as mentioned above, and from the parameterized complexity point of view, d is just a natural parameter to start with.

Intuitively one would expect this problem to be in FPT, since a small length difference allows only a few short TDs, and together with the linear structure of strings it should be possible to apply dynamic programming on subsets or a related technique. The basic idea is to decide which symbols in T shall be kept (matched) or deleted (unmatched), and (if possible) to delete exactly the unmatched symbols by a minimum number of contractions. However, it is not so obvious how to do the "local" technical details in the most efficient way, as one must care about dependencies among overlapping squares.

We also construct a polynomial kernel, which needs even more effort. Part of the preprocessing is an acyclic directed grid graph that encodes all possible alignments of the input strings, including invalid ones, and whose geometry allows to derive data reduction rules.

We remark that our problem can be seen as a variant of string editing, with TDs as edit operations. It is well known that string editing problems with insertions, deletions, and replacements as edit steps can be easily rephrased as shortest path problems in a certain alignment graph. For TD minimization we have to use an alignment graph in a more elaborated way (where periods in strings play an essential role), indicating that the concept might prove useful also for other string editing variants.

We conclude the paper with open questions.

1.3 Periods

We provide some terminology concerning periods in strings. For a positive integer p, a string $R = r_1 \dots r_m$ is said to have a *period* p if $r_i = r_{i+p}$ holds for every index i with $1 \leq i < i + p \leq m$. A *p-run* in a string is a substring that has a period p and length at least $2p$, and is maximal with these properties, in the sense that adding another adjacent symbol would destroy the period p. Note that every substring of length $2p$ in a p-run is a square. A substring which is a p-run for some number p is also simply called a *run*, without specifying p. The *exponent* of a run R is its length $|R|$ divided by its shortest period; note that this is in general a fractional number.

1.4 Fixed-Parameter Tractability and Problem Kernels

Technical introductions to parameterized algorithms and complexity can be found in a number of textbooks. In a nutshell, a problem with input size n and another input parameter d is in XP and in FPT, if some algorithm can solve it in time $O(n^{f(d)})$ and $O(f(d) \cdot n^{O(1)})$, respectively, where f is some computable function.

Given an instance of a parameterized problem, a *kernel* is another instance with the following properties: It is equivalent to the given instance, in that it yields the same output, its size is bounded by some function of d, and it is computable from the given instance in a time being polynomial in n. A problem is in FPT if and only if it possesses a kernel. But the kernel size is not always polynomial in d.

The O^* notation for XP and FPT time bounds depending on the input size n and a parameter d suppresses factors that are polynomial in n (with a constant exponent not depending on d) such that it focusses on the more critical dependency on the parameter.

2 Improved Branching for Arbitrary Strings

Not very surprisingly, the brute-force bound in [9] is a bit of an overestimate. Our idea is to avoid this brute-force branching on all substrings and potential contractions, and to branch only on contractions that produce *different* strings, taking advantage of some nice properties of periods. We begin with some combinatorial lemma.

Lemma 1. *All contractions of squares of length $2p$ in a p-run yield the same contraction result.*

Proof. We choose a square XX of length $2p$ as indicated, and write the p-run accordingly as $AXXB$. Let $CAXXBD$ be the entire string. Note that each of A, B, C, D might be empty. The result $CAXBD$ of the contraction can be obtained by deleting the right X and moving BD by p positions to the left. (If BD is empty, nothing happens in this step.) Since p is a period of $AXXB$, the prefix $CAXB$ of the contraction result has, at every position, the same symbol as $CAXXB$ had, and the suffix D is always the same, independently of the position of XX. □

This gives already an improvement of the trivial $O(n^2)$ bound on the number of different contraction results:

Proposition 2. *A string of length n has at most $n(\ln n + 1)$ different contraction results.*

Proof. First we observe, for every p, that any two squares of length $2p$ that overlap in at least p positions belong to the same p-run. Namely, let $T = t_1 \dots t_n$ be a string, and let $t_{i+1} \dots t_{i+2p}$ and $t_{i+k+1} \dots t_{i+k+2p}$ be squares, with some positive integer $k \leq p$. We have to show $t_j = t_{j+p}$ for all j with $i+1 \leq j < j+p \leq i+k+2p$, or simpler, $i+1 \leq j \leq i+k+p$. For $j \leq i+p$ this holds because of the first square. For $j \geq i+k+1$ this holds because of the second square. Every j in our range satisfies at least one of these conditions, since $j \geq i+p+1$ implies $j \geq i+k+1$ by $k \leq p$.

To prove the actual assertion, consider any fixed positive integer $p \leq n/2$, and all squares of length $2p$ in the string. Whenever the left ends of two squares are at most p positions away, they belong to the same p-run, and by Lemma 1 they yield the same contraction result. By contraposition this can be formulated also in this way: Two squares of length $2p$ can yield different contraction results only if their left ends have a distance at least p. Hence at most n/p different such contraction results can exist. Finally, summing over all p yields the claim, using the well-known sum of the harmonic series. □

However, we can do even better, using deeper combinatorics of strings: In [4] it was shown that the sum of exponents of all runs in a string of length n is at most $4.1n$, thus improving linear upper bounds with larger constants from earlier papers. This further reduces the XP time bound considerably:

Theorem 3. *For a string T of length n and another string S we can find a sequence of at most k TDs that transform S into T (or report that none exist) in $O^*((2.05n)^k)$ time. Moreover, there exist at most $(2.05n)^k$ different sequences of strings of the form $S = R_1, R_2 \dots, R_j = T$, where $j \leq k$, and every R_{i+1} is obtained from R_i by some TD.*

Proof. Simply branch on all possible contraction results, in a search tree with T at the root and with depth k, keep only the contraction sequences that result in S. The branching factor, i.e., the base of the time bound, is the maximum number of different contraction results in a string.

Proposition 2 would already yield $O^*((n \log n)^k)$. For the better linear base, observe that the number of different periods $p \leq k/2$ of a run of length k is at most half its exponent. Lemma 1 can be rephrased in the way that the contractions of any squares of the same fixed length $2p$ in the run, where the numbers $p \leq k/2$ are periods of the run, yield the same contraction results. Hence the number of distinct contraction results overall is at most half the sum of the exponents of all runs. With the bound $4.1n$ [4] we obtain $2.05n$. □

Remark 4. The bound of $O(n)$ distinct squares ($2n$ in [6], later improved by several authors like [7,5] to eventually less than n in [1]) does not simply imply a time bound as in Theorem 3, because squares that have different locations but are equal as strings are not counted there as distinct, but they can produce different contraction results. A similar remark holds for the known result that every string has only $O(n)$ different runs [8,3]. The catch is that a p-run (as defined here) is also a q-run for all integer multiples q of p until the half length of the run, and squares of different lengths yield, of course, different contraction results. Therefore, the linear bound on the sum of exponents of the runs is needed.

Remark 5. Since the $O(n^{2k})$ bound in [9] was used after kernelization in their FPT result when S is an exemplar string and k is the parameter, Theorem 3 improves the

running time for solving this problem kernel accordingly. Probably Theorem 3 can also improve other related results.

3 Length Difference as Parameter

Next we consider our TD minimization problem parameterized by the length difference $d = |T| - |S|$. (In the rest of the paper, symbol d is reserved for the exact difference.) Since the number k of TDs that transform S into T trivially satisfies $k \leq d$, Theorem 3 yields an $O^*((2.05n)^d)$ time bound for finding a shortest sequence of TDs. However, below we will obtain an FPT algorithm for the parameter d. The idea is quite natural: Since most of the symbols are not involved in TDs, certain sub-instances of the problem, consisting of certain pairs of substrings of T and S, can be solved independently, while a dynamic programming process cares of separators of matching substrings.

Theorem 6. *For a string T of length n and another string S, with length difference $d = |T| - |S|$, we can find a sequence with a minimum number of TDs that transforms S into T (or report that none exist) in $O^*((2d)^d)$ time, more precisely, in $O(d^3(2d)^d n)$ time.*

The remainder of this section is devoted to the proof of Theorem 6. First some more preparations and definitions are needed. Remember that a substring consists of consecutive symbols in a string. Sometimes we use $r_1 \dots r_n$ with $n = 0$ to denote an empty string.

Lemma 7. *Consider any sequence of contractions of T that deletes d symbols in total. Then there exists a partitioning of T into substrings of length at most $2d$, such that every contracted square is entirely contained in one of these substrings.*

Proof. We call a symbol in T active when it participates in some contracted square (no matter whether it belongs to the deleted or undeleted half of that square). All other symbols are called inactive. Since d symbols are deleted in total, at most $2d$ symbols are active. (It could be fewer than $2d$ symbols when the contracted squares overlap.) Any maximal substring of active symbols is called a block.

Consider any subset Q of symbols in T that becomes a contracted square sometimes during our contraction sequence, and consider any inactive symbol u in T. Assume that symbols of Q appear both to the left and to the right of u. Since, by definition, inactive symbols are never deleted in a sequence of contractions, u remains present all the time, such that Q will never become a substring (i.e., consist of consecutive symbols), which contradicts the assumption that Q becomes a contracted square.

It follows that every contracted square is contained in some block. Now we simply make every block a substring of our claimed partitioning, and divide the inactive symbols arbitrarily into substrings of length at most $2d$. □

Dynamic programming function. Given $S = s_1 \dots s_m$ and $T = t_1 \dots t_n$, we define $c(i, j)$ to be the minimum number of contractions needed to transform $t_1 \dots t_j$ into $s_1 \dots s_i$, and $c(i, j) := \infty$ if no such transformation exists. To account for empty prefixes we also define $c(0, 0) = 0$.

Principle of optimality. Let i and j be any indices such that $c(i,j) < \infty$, and consider some corresponding optimal solution, i.e., minimum sequence of contractions. By Lemma 7 there exists some index $b \geq j - 2d$ such that for every contracted square Q, the symbols of Q are either all in $t_1 \dots t_b$ or all in $t_{b+1} \dots t_j$. Let e be the index such that $t_1 \dots t_b$ is transformed into $s_1 \dots s_e$. It follows that $c(i,j)$ is the sum of $c(e,b)$ and the minimum number of contractions needed to transform $t_{b+1} \dots t_j$ into $s_{e+1} \dots s_i$.

Computing the optimal values. Based on this observation, we now describe the computation of $c(i,j)$ for any given pair of indices (i,j). If $i > j$ then $c(i,j) = \infty$. Values for $i = j$ are also clear: If $t_1 \dots t_j = s_1 \dots s_j$ then $c(j,j) = 0$, else $c(j,j) = \infty$.

Due to the principle of optimality, we may proceed as follows. Try all possible index pairs (e,b), where $j - 2d \leq b < j$ and $e \leq b \leq e + d$. For every such pair, transform $t_{b+1} \dots t_j$ into $s_{e+1} \dots s_i$ using a minimum number of contractions. Add this number to $c(e,b)$. Finally, $c(i,j)$ is the minimum of these sums, for all index pairs. The mentioned index pairs (e,b) are sufficient, since $b \geq j - 2d$ was shown in the paragraph above, the inequalities $e \leq b \leq j$ are trivial by the definitions, and e can differ from b by at most d, since a string of length b can be contracted only to strings of length at least $b - d$ (and $c(e,b)$ would be infinite otherwise).

Complexity analysis. We check $O(d^2)$ pairs of indices (e,b). Now we bound the time needed to optimally transform $t_{b+1} \dots t_j$ into $s_{e+1} \dots s_i$. The length of the former string is at most $2d$ and the number k of contractions is bounded by d. The XP algorithm from [9] enumerates all possible sequences of contractions, and takes the shortest one.

A sequence of, say, k contractions can be uniquely specified, e.g., by the left ends and the lengths $l_1, \dots, l_k$ of the deleted substrings. For the left ends we have (rather generously) at most $(2d-1)^k$ choices. Since $l_1 + \dots + l_k \leq d$, the number of possible sequences of k lengths is $\binom{d}{k}$. Hence, by the binomial formula, the total number of choices for all $k \leq d$ is at most $((2d-1)+1)^d = (2d)^d$.

Since $j \leq n$ and $i \leq j \leq i + d$, we must compute $O(dn)$ values $c(i,j)$. (To see that only these values are needed, remember the definition of $c(i,j)$ and of the parameter d.) The product of the three terms above yields the claimed overall time bound. This concludes the proof of Theorem 6.

From the optimal values, a specific solution can be reconstructed by backtracing in the standard way.

Remark 8. Direct application of the trivial $O(n^{2k})$ bound from [9] to $n \leq 2d$ and $k \leq d$ would only yield $O^*((2d)^{2d})$ time, but another counting argument above gave $O^*((2d)^d)$, since $2d$ is close to d. Also note that the direct application of Theorem 3, that was made for general n and k, would yield $O^*((2.05d)^d)$ in our case, which is slightly worse. The picture would change with an improved bound on the sum of exponents of runs.

4 Kernelization

In this section we construct a kernel that is polynomial in the parameter $d = |T| - |S|$. Recall the notation $S = s_1 \dots s_m$ and $T = t_1 \dots t_n$. With the help of an alignment graph that depicts possible "paths of insertions" of symbols, we find, in any large

enough instance, either two matching substrings not involved in TDs, or periodic substrings whose deletion does not change the result of the instance.

4.1 Alignment Graph Construction

As a first step we define a directed graph G that we call the *alignment graph*. Its vertices are certain (but not all) pairs of integers (i, j) in the rectangle specified by $0 \leq i \leq m = |S|$ and $0 \leq j \leq n = |T|$. They can be imagined as grid points in a Cartesian coordinate system. Also note that $d = n - m$.

We will create certain directed edges of the form either $(i, j-1) \to (i, j)$ or $(i-1, j-1) \to (i, j)$, called *vertical* and *diagonal edges*, respectively.

Before specifying exactly which vertices and edges exist in G, we outline the idea behind the graph: Traversing $(i, j-1) \to (i, j)$ shall model the deletion of t_j, and traversing $(i-1, j-1) \to (i, j)$ shall model the action of matching symbol s_i to symbol t_j. Thus every solution with exactly d unmatched symbols corresponds to some directed path from $(0, 0)$ to (m, n) with at most d vertical edges. Of course, the converse of the last statement is far from being true. We also "purify" our graph by not creating some obviously useless vertices and edges. Now we describe the actual construction. See also Figure 1 for the position of the alignment graph in the i-j coordinate system.

Definition 9. *The alignment graph of two strings S and T is obtained as follows. We (tentatively) create all vertices (i, j) with $0 \leq i \leq j \leq i + d$, and all vertical and diagonal edges between them. Next, any diagonal edge $(i-1, j-1) \to (i, j)$ is retained only if $s_i = t_j$, and otherwise deleted. Finally we also delete all vertices (and all their incident edges) that cannot be reached from $(0, 0)$ or cannot reach (m, n) via directed paths.*

The construction can be done in $O(dn)$ time by standard techniques: First, the graph has obviously $O(dn)$ vertices and edges, respectively, and it is a directed acyclic graph. For every diagonal edge it is decided locally, by testing the equality of two symbols, whether it is retained or deleted. Finally, the vertices reachable from $(0, 0)$ or (reversely to the edge orientations) from (m, n) can be determined by breadth-first search in linear time.

In the so obtained alignment graph G, the directed paths from $(0, 0)$ to (m, n) describe exactly all possible alignments of S and T after deleting d symbols from T, but still without caring whether these deletions can be realized by contractions of squares.

The alignment graph has two special directed paths from $(0, 0)$ to (m, n) that we call the *left* and *right greedy path*, defined as follows.

Definition 10. *The left greedy path always traverses a vertical edge, and whenever the vertical edge from the current vertex does not exist, it traverses the diagonal edge instead. Similarly, the right greedy path always traverses a diagonal edge, and whenever the diagonal edge from the current vertex does not exist, it traverses the vertical edge instead.*

Note that the alternative edge always exists by construction, because G contains only vertices from which (m, n) is reachable. That is, we never get stuck. We also observe that all vertices of the alignment graph are in the region of the plane bounded

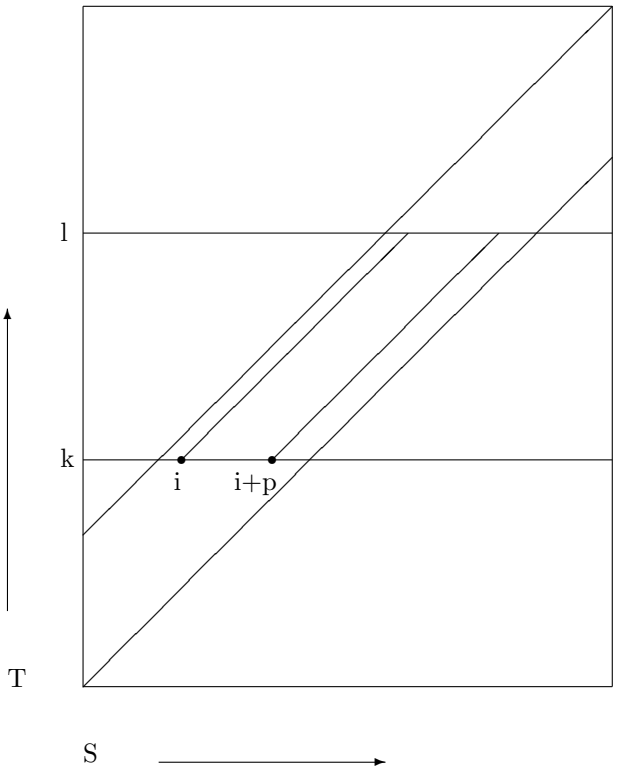


Figure 1. Diagram of the alignment graph. Its vertices are located in the region between the two long diagonal lines. The breadth of this stripe is d, both in horizontal and vertical direction. The diagram also depicts a pair of diagonal paths with coordinates as in Lemma 11.

by these two greedy paths. Otherwise some directed edge must leave this region, which would contradict the definition of the greedy paths.

We call any directed path merely consisting of diagonal edges a *diagonal path.*

4.2 Periods and Alignment Graph Properties

For convenience we say that a vertex (i, j) in the alignment graph is in column i and row j (see Figure 1).

Lemma 11. *Consider two rows k and l with $l-k > d$ in the alignment graph. Suppose that two (not necessarily distinct) directed paths from row k to row l are diagonal. Then either these paths are identical, or they have the form*

$$(i,k)\ldots(i+l-k,l) \text{ and } (i+p,k)\ldots(i+p+l-k,l)$$

for some integers i $(k-d \le i \le k)$ and p $(1 \le p \le d-(i-k))$, and in the latter case, both $t_{k+1}\ldots t_l$ and $s_{i+1}\ldots s_{i+p+l-k}$ have a period p.

Proof. We only have to show periodicity in the latter case. It follows directly from the condition for the existence of diagonal edges. Namely, for every q $(0 < q < p)$ we now obtain $s_{i+q} = t_{k+q} = s_{i+q+p} = t_{k+q+p} = s_{i+q+2p} = t_{k+q+2p} = \ldots$ (and so on in this way, as long as the indices are in the given interval), which yields the assertion. □

For formal clarity we need some further technical definitions.

A *valid sequence* for $T = t_1\ldots t_n$ and $S = s_1\ldots s_m$ means any sequence of contractions of squares that transforms T into S. Recall that every contracted square

has a length of at most $2d$. Without loss of generality we can assume that every contraction deletes the left half of the contracted square. With this convention, any valid sequence defines a partial function c from $\{t_1, \ldots, t_n\}$ onto $\{s_1, \ldots, s_m\}$, where $c(t_i) = s_j$ means that t_i is turned into s_j by the valid sequence, and $c(t_i)$ is undefined if t_i gets deleted by some contraction. This function can be naturally extended to substrings T^* and S^* of T and S, respectively: $c(T^*) = S^*$ means that T^* is turned into S^*. Note that this is possible only if $|T^*| = |S^*|$.

A *fresh symbol* is a symbol that does not yet appear in the strings at hand, that is, it may even extend the alphabet.

Lemma 12. *Let T^* be a substring of T, and let S^* be a substring of S, with the following properties:*

- $|T^*| = |S^*| \geq d + 2$.
- *In every valid sequence for T and S it holds that $c(T^*) = S^*$. (In particular, all symbols in T^* are matched symbols, in every valid sequence.)*

Next, we write T^* as a concatenation $T_1^*T_2^*$ where $|T_1^*| = d$, replace T_2^* (in T) with one fresh symbol f, and denote the resulting string T'. Similarly, we write S^* as a concatenation $S_1^*S_2^*$ where $|S_1^*| = d$, replace S_2^* (in S) with the same symbol f, and denote the resulting string S'.
Then, the valid sequences for T and S are exactly the same as the valid sequences for T' and S', in the sense that they contract the same squares, and in the same order.

Proof. The two directions of the equivalence are similar in structure, but they have to use slightly different arguments:

Consider any valid sequence for T and S. The assumption implies that every contracted square is completely to the right of T^* or contains at most the d leftmost symbols of T^*. Hence the same sequence is also valid for T' and S'.

Consider any valid sequence for T' and S'. Since f occurs only once in T' and S', it follows that f is a matched symbol, and f in T' is matched onto f in S'. Furthermore, every contracted square is completely to the right or to the left of f. Hence the same sequence is also valid for T and S. □

In the following we use a nice and simple property of periodic strings. Let X be any string that has a period p: Let us delete any substring P of length p and concatenate the two remaining substrings of X. Then the resulting string does not depend on the choice of P. Basically this was already stated in different phrasing in Lemma 1, but here we add a similar observation for the reverse operation: Let us choose any position between two neighbored symbols of X and insert there the suitable (and uniquely determined) string of length p that preserves periodicity. Then the resulting string does not depend on the choice of that position. We refer to these two operations as *shortening* and *enlarging* a p-periodic string X by p consecutive symbols.

We remark that, in the assumptions of the following lemma, C "starts d positions earlier" than D, while their end positions are the same. This is intended, as we need that to account for the deletion of up to d symbols from T until position $j + l$. Also remember that d denotes the exact difference $|T| - |S|$, not only an upper bound.

Lemma 13. *Suppose that the string T contains a substring $D = t_j \ldots t_{j+l}$ having a start position $j > d$, the length $l + 1 > d(d + 1) + 2d = d^2 + 3d$ and a period*

$p \le d$, and that the string S contains a substring $C = s_{j-d} \dots s_{j+l}$, also with period p. Then, shortening both D and C by p consecutive symbols yields an equivalent problem instance, that is, an instance with the same optimal number of contractions.

Proof. Consider any valid sequence for T and S. Since every contraction of a square of length $2q$ deletes q unmatched symbols, and d unmatched symbols exist in total, the (at most d) contracted squares together cover at most $2d$ symbols. There remain at least $d(d+1)$ symbols in D which are not covered by any contracted squares. We further observe that D can be uniquely partitioned into maximal substrings of covered symbols and uncovered symbols, respectively. Since at most d contractions are done, this partitioning has at most d substrings of covered symbols, hence at most $d+1$ substrings of uncovered symbols. It follows that some substring of uncovered symbols has a length at least d. In other words, D has some substring M of d symbols that are not involved in any contracted square. In particular, M is a substring of matched symbols.

The aforementioned substring M of T is matched onto a substring of S that we denote N. That is, $c(M) = N$. More specifically, N is a substring of C (since C starts at position $j-d$, and at most d symbols are deleted from D). Let us remove some substring P of length exactly $p \le d$ from M, and also remove the corresponding substring (its matching partner $c(P)$) from N. Let T' and S' denote the resulting strings, after these deletions from T and S, respectively. Then our valid sequence is also valid for T' and S', since M is not involved in any contraction.

Now remember the above definition of shortening and enlarging, and note that removing P from M and $c(P)$ from N means to shorten the p-periodic strings D and C, respectively, by p symbols. Similarly, we may insert, at corresponding positions in M and N, two equal substrings of length exactly p that preserve the period. The effect is that the p-periodic strings D and C are enlarged by p symbols. and our valid sequence is also valid for the resulting strings T' and S', for the same reason (M is not involved in any contraction).

Finally consider a valid sequence doing the optimal number k of contractions for transforming T into S. To summarize the observations above, shortening D and C yields a valid sequence with k contractions, furthermore, after this shortening there cannot exist another valid sequence with less than k contractions, since this would imply such a sequence also after enlarging D and C again (thus recovering the given T and S), which contradicts the minimality of k. This shows the assertion. □

4.3 The Kernel

Using the previous lemmas we can now finish up. Tandem duplication admits a kernel of size $O(d^3)$, or in more detail:

Theorem 14. *For a string T of length n and another string S, with length difference $d = |T| - |S|$, the problem of finding a sequence with a minimum number of TDs that transforms S into T (or report that none exist) has a kernel of size $O(d^3)$ that can be computed in $O(dn)$ time.*

Proof. We can assume $d > 3$ and $n > 6d^3$ (for simplicity with a generous constant factor), otherwise there is nothing to prove. For better orientation in the proof see Figure 1 again.

First we construct the alignment graph G. Since each of the two greedy paths uses d vertical edges, both greedy paths together can use at most $2d$ vertical edges.

Trivially, these $2d$ vertical edges $(i, j-1) \to (i, j)$ can use at most $2d$ different coordinates j on the T-axis, and these positions j cut the T-axis into at most $2d+1$ intervals. Furthermore, within these intervals, the greedy paths can use only diagonal edges. From these observations it follows the existence of an interval $D \subset [0, n]$ of length $n/(2d+1) > n/(3d)$ such that the sub-paths of both greedy paths restricted to the indices $j \in D$ are diagonal paths. By Lemma 11, either the two greedy paths restricted to D are identical, or T and S have aligned substrings with a period at most d and length $n/3d$. Since $n > 6d^3$, this length is larger than $2d^2 \geq d^2 + 4d > d + 2$.

In the former case, since G is bounded left and right by the two greedy paths, all alignments must use the greedy path restricted to D. Thus we are in the situation of Lemma 12, that is, we know two substrings of length at least $d+2$ that are necessarily matched to each other. As described there, using a fresh symbol we can shorten the instance to an equivalent one.

In the latter case we use Lemma 13 to shorten the instance to an equivalent one. (The substrings D and C specified there have lengths of at least $d^2 + 4d$.) This can be done only $O(n)$ times until $n \leq 6d^3$.

Thus we eventually obtain a kernel of size $O(d^3)$. As for the time for this kernelization, we first remark that polynomial time is evident. For the claimed specific time bound we observe: The initial alignment graph can be constructed in $O(dn)$ time, as noticed earlier, and the greedy paths are found in linear time. The shortening operations together take only linear (not quadratic) time as well, since they all cut away pairwise disjoint parts of the strings and the alignment graph, and updates after every shortening operation. are local □

A fully worked out substantial example would be lengthy, but a simple example just illustrating the principle could be a pair of strings like $S = aaaacacbbababababab$ and $T = aaaaacacbbababababababab$. The kernelization algorithm would recognize that *cacbb* must be mapped to *cacbb* and replace this pattern with a fresh symbol, and shorten the periodic substrings to the left and to the right of this pattern, by cutting out the same periods in both S and T, and the kernel would still reveal that some a and some ab must be doubled at the left and right end, in order to convert S into T.

In Lemma 12, for simplicity we did not care about the number of different fresh symbols. Substrings with more than d positions in between may reuse the same fresh symbols. Since at most d symbols of T are deleted in total, these remote occurrences of the same symbol would not interfere, and still the correct symbols would be matched to each other. Thus, extending the alphabet by only $O(d)$ different fresh symbols suffices. A technically more challenging question is whether one can avoid any extension of the alphabet, without sacrificing the kernel size, or at least lower the number of different fresh symbols further, e.g., by using carefully designed substrings instead of the fresh symbols.

5 Concluding Discussions

To our best knowledge, it is open whether the tandem duplication problem for arbitrary strings, parameterized by the number k of contractions, is fixed-parameter tractable (or perhaps W[1]-hard). A positive answer would imply our FPT result for parameter d, but even in that case, the question of complexity bounds would remain interesting. E.g., recall that the dependence of the time bound on d might be further improved.

We have studied the length difference d as parameter, but note that $k \leq d$, and a certain weakness of parameter d is that k can be arbitrarily smaller. A refinement worth investigating is the combined parameter (k, ℓ) where ℓ denotes the maximum length of substrings to be duplicated. Note that $d \leq k\ell$, hence our problem is in FPT also with parameter $k\ell$. Since trivially $\ell \leq d$, parameter ℓ alone would be much stronger than d, but the question whether tandem duplication parameterized by ℓ is in FPT seems to be as challenging as for k. Also a more fine-grained analysis in the parameter (k, ℓ) rather than d does not appear to be straightforward. But hardness results (provided that they hold) might be easier to prove for stronger parameters.

Finally, in our results we have not fixed the alphabet size. Would fixed alphabet sizes allow stronger time bounds? Can one construct a kernel without extending the alphabet?

Acknowledgements

Part of the algorithmic ideas have been developed during the supervision of the master's thesis project of Belmin Dervisevic and Mateo Raspudic. I would like to thank the former students for inspiring discussions, an anonymous reviewer of an earlier version for drawing my attention to exponents of runs, and the anonymous reviewers at PSC for several small corrections.

References

1. S. Brlek and S. Li: *On the number of distinct squares in finite sequences: Some old and new results*, in Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings, A. E. Frid and R. Mercas, eds., vol. 13899 of Lecture Notes in Computer Science, Springer, 2023, pp. 35–44.
2. F. Cicalese and N. Pilati: *The tandem duplication distance problem is hard over bounded alphabets*, in Combinatorial Algorithms - 32nd International Workshop, IWOCA 2021, Ottawa, ON, Canada, July 5-7, 2021, Proceedings, P. Flocchini and L. Moura, eds., vol. 12757 of Lecture Notes in Computer Science, Springer, 2021, pp. 179–193.
3. M. Crochemore, L. Ilie, and W. Rytter: *Repetitions in strings: Algorithms and combinatorics.* Theor. Comput. Sci., 410(50) 2009, pp. 5227–5235.
4. M. Crochemore, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *On the maximal sum of exponents of runs in a string.* J. Discrete Algorithms, 14 2012, pp. 29–36.
5. A. Deza, F. Franek, and A. Thierry: *How many double squares can a string contain?* Discret. Appl. Math., 180 2015, pp. 52–69.
6. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* J. Comb. Theory, Ser. A, 82(1) 1998, pp. 112–120.
7. L. Ilie: *A note on the number of squares in a word.* Theor. Comput. Sci., 380(3) 2007, pp. 373–376.
8. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA, IEEE Computer Society, 1999, pp. 596–604.
9. M. Lafond, B. Zhu, and P. Zou: *Computing the tandem duplication distance is np-hard.* SIAM J. Discret. Math., 36(1) 2022, pp. 64–91.
10. Q. Liu, Y. Tong, and K. Wang: *Genome-wide detection of short tandem repeat expansions by long-read sequencing.* BMC Bioinform., 21-S(21) 2020, p. 542.
11. H. Yu, S. Zhao, S. Ness, H. Kang, Q. Sheng, D. C. Samuels, O. Oyebamiji, Y. Zhao, and Y. Guo: *Non-canonical RNA-DNA differences and other human genomic features are enriched within very short tandem repeats.* PLoS Comput. Biol., 16(6) 2020.

Improved Practical Algorithms to Compute Maximal Covers

Holly Koponen, Neerja Mhaskar, and W. F. Smyth*

Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
koponeh@mcmaster.ca, pophlin@mcmaster.ca, smyth@mcmaster.ca

Abstract. A ***cover*** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position in $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$. Since very few strings possess a cover, it becomes interesting to compute various kinds of cover generalizations. Here we describe algorithms to compute a ***maximal cover***[1]; that is, a repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that covers $M = M_{\boldsymbol{x}}$ positions, the maximum coverage attained by any repeating substring of $\boldsymbol{x}$. In 2015, an $O(n \log n)$-time algorithm to compute a maximal cover was proposed; but the algorithm was complex, making use of annotated suffix trees. In 2022, an $O(n^2)$-time maximal cover algorithm was implemented and evaluated on protein sequences. In this paper, we propose two simple $O(n^2)$-time algorithms for this problem that, nonetheless, as we show by experiment, execute in linear time in many cases that arise in practice, and are much faster than the algorithm recently implemented in 2022. On the other hand, when experiments are restricted to the highly repetitive Fibonacci strings, the behaviour of both algorithms is clearly quadratic.

Keywords: string, cover, Fibonacci, algorithm

1 Introduction

As introduced in [2,3], a ***cover*** of a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position of $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$ — thus a cover occurs at least twice in $\boldsymbol{x}$. For example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaaba$. Even though all the covers of every prefix of $\boldsymbol{x}$ can be computed, whenever they exist, in $O(n)$ time [12], nevertheless, since very few strings possess a cover, in order to provide a compact representation of $\boldsymbol{x}$, various alternate covering structures have been proposed:

- *k-cover* [9]: a minimum collection of substrings of $\boldsymbol{x}$, each of given length $k < n$, that covers $\boldsymbol{x}$ — this computation turns out to be NP-hard [5];
- *enhanced cover* [7,1]: the border $\boldsymbol{u}$ (both prefix and suffix) of $\boldsymbol{x}$ that, over all borders of $\boldsymbol{x}$, covers a maximum number of positions — computable in $\Theta(n)$ time, but its effectiveness depends on some border also being a good cover — a rare occurrence.
- *α-partial cover* [10]: the shortest substring $\boldsymbol{u}$ of $\boldsymbol{x}$ (if it exists) that, for $\alpha = 1, 2, \ldots, n$, covers at least α positions in $\boldsymbol{x}$ — this computation uses only $\mathcal{O}(n \log n)$ time over all values of α and thus also computes the *maximal cover* (next entry), but on the other hand requires space-consuming data structures ("augmented and annotated" suffix trees);

* Supported by Grant No. 105–36797 from the Natural Sciences & Engineering Research Council of Canada (NSERC).

[1] The term "optimal cover" was used in [13].

Holly Koponen, Neerja Mhaskar, W. F. Smyth: Improved Practical Algorithms to Compute Maximal Covers, pp. 30–41.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

- *maximal cover* [14]: a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that occurs at least twice and that, over all substrings, covers a maximum number $M_{\boldsymbol{x}}$ of positions — to compute $\boldsymbol{u}$, an $\mathcal{O}(n^2)$-time and $\Theta(n)$-space algorithm is described[2];
- *frequency cover* [13]: any substring $\boldsymbol{u}$ of $\boldsymbol{x}$, of length greater than 1, that occurs most frequently — this requires only $\Theta(n)$ time, but yields a good cover only if $\boldsymbol{x}$ contains many short repeating substrings;

For example, given $\boldsymbol{x} = ababaaaba$ of length 9, there is no cover as defined above, but the set $\{aba, aab\}$ is a 3-cover; aba is a maximal cover, also an 8-partial cover (there is no 9-partial cover); ab and aba are frequency covers; aba is an enhanced cover. For further reading, see the survey [15].

In this paper, we introduce terminology and symbolism in Section 2. Then in Section 3 we outline the overall methodology for calculating maximal covers of $\boldsymbol{x}$ from the "Overlapping Positions" $\mathcal{OLP}$ array, which is the fundamental data structure underlying this computation. Section 4 describes the three competing $\mathcal{O}(n^2)$ algorithms[3] that compute $\mathcal{OLP}$, one already published in [14], and implemented in [8]. Here we describe two new ones introduced in [11] and provide computational evidence that they are faster in practice, requiring only linear time on average over a wide range of test cases. Section 5 compares the execution times of all three $\mathcal{OLP}$ algorithms applied to random strings with alphabet sizes $\sigma = \{2, 3, 4\}$, to Fibonacci sequences, and to protein sequences. Section 6 summarizes our findings and proposes future work.

2 Preliminaries

A ***string*** $\boldsymbol{x}[1..n]$ is a concatenation of symbols drawn from a totally ordered set Σ, called an ***alphabet***, where $\sigma = |\Sigma|$. The ***length*** of $\boldsymbol{x}$ is $|\boldsymbol{x}| = n$. A string of length zero is called an ***empty string*** and denoted by ε. A string $\boldsymbol{u}$ is called a ***substring*** of $\boldsymbol{x}[1..n]$ if $\boldsymbol{u} = \varepsilon$ or $\boldsymbol{u} = \boldsymbol{x}[i..j]$ and $1 \leq i \leq j \leq n$, a ***proper substring*** if $|\boldsymbol{u}| < n$. A substring $\boldsymbol{u}$ of $\boldsymbol{x}[1..n]$ is called a ***prefix (suffix)*** of $\boldsymbol{x}$ if $\boldsymbol{u} = \varepsilon$ or $\boldsymbol{u} = \boldsymbol{x}[1..i]$ and $1 \leq i \leq n$ ($\boldsymbol{u} = \boldsymbol{x}[j..n]$ and $1 \leq j \leq n$). A ***border*** $\boldsymbol{u}$ of $\boldsymbol{x}$ is a proper substring of $\boldsymbol{x}$ that is both a prefix and suffix of $\boldsymbol{x}$. For example, $\boldsymbol{x} = abacaba$ has borders aba, a and ε. A substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is a ***repeating substring*** of $\boldsymbol{x}$ if it occurs at least twice in $\boldsymbol{x}$. A ***left (right) extendible*** repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is a repeating substring of $\boldsymbol{x}$ whose each occurrence in $\boldsymbol{x}$ is preceded (followed) by the same symbol. If every occurrence of $\boldsymbol{u}$ is not preceded (followed) by the same symbol then it is called a ***non-left (non-right) extendible, NLE (NRE)*** repeating substring of $\boldsymbol{x}$.

A ***run*** in $\boldsymbol{x}$ is a substring $\boldsymbol{x}[i..j] = \boldsymbol{u}^e\boldsymbol{u'}$, where $e \geq 2$, $\boldsymbol{u'}$ is a prefix of $\boldsymbol{u}$, and the periodicity $|\boldsymbol{u}|$ cannot be extended left or right [18].

In this paper, we describe and compare algorithms that compute maximal covers $\boldsymbol{u}$; that is, repeating substrings that cover a maximum $M = M_{\boldsymbol{x}}$ positions of a given string $\boldsymbol{x}$. It may be that more than one substring $\boldsymbol{u}$ covers M positions; for example, $\boldsymbol{x} = aabaababaaba$ of length 12 has three maximal covers: $\boldsymbol{u}_1 = aba$, $\boldsymbol{u}_2 = aaba$, $\boldsymbol{u}_3 = (aba)^2$. Thus the longest (shortest) maximal cover may be of interest: the longest could provide more information about the structure of $\boldsymbol{x}$, while the shortest is more compact.

[2] An $\mathcal{O}(n \log n)$-time algorithm proposed in [14] was incorrect.

[3] All algorithms described in this paper are assumed to run on a word RAM model with word size $= k$ bits, $k \leq \log n$, where n is the input size.

The maximal cover algorithms proposed here both require two well-known data structures: the $\mathcal{SA}$ and $\mathcal{LCP}$ arrays. The ***suffix array*** $\mathcal{SA}$ is an integer array of length $|\boldsymbol{x}|$ such that $\mathcal{SA}[i]$ is the starting position of the i-th lexicographically least suffix in $\boldsymbol{x}$. The ***longest common prefix array*** $\mathcal{LCP}$ is also an integer array of length $|\boldsymbol{x}|$, where $\mathcal{LCP}[1] = 0$ and $\mathcal{LCP}[i]$, $i \in [2..n]$, is the length of the longest common prefix between the suffixes of $\boldsymbol{x}$ starting at $\mathcal{SA}[i-1]$ and $\mathcal{SA}[i]$. For convenience, we define $\boldsymbol{v}_i$ to be the NRE repeating substring of length $\mathcal{LCP}[i]$ occurring at positions $\mathcal{SA}[i-1], \mathcal{SA}[i]$ in $\boldsymbol{x}$ — that is,

$$\boldsymbol{v}_i = \boldsymbol{x}[SA[i] \,..\, \mathcal{SA}[i] + \mathcal{LCP}[i] - 1]. \tag{1}$$

3 Computing Maximal Covers

In order to compute maximal covers of $\boldsymbol{x}$, we require the $\mathcal{SA}_x$ and $\mathcal{LCP}_x$ arrays, defined above. Also required are the following:

- The ***repeating substring frequency array*** $\mathcal{RSF}$ (introduced in [13]) is an integer array of length $|\boldsymbol{x}| = n$ such that $\mathcal{RSF}[i]$ is the number of occurrences, that is 'frequency', of the NRE repeating substring v_i in x. Thus, if $\mathcal{RSF}[i] = m$, then there exist m consecutive index positions $r \in \{r_1, r_1 + 1, ..., r_m - 1, r_m\}$ in $\mathcal{SA}$ such that $\boldsymbol{x}[\mathcal{SA}[r]..n]$ has prefix $\boldsymbol{v}_i$. Note that for $\mathcal{LCP}[i] = 0$, $\mathcal{RSF}[i] = 0$.
- The ***overlapping positions array*** $\mathcal{OLP}$ (introduced in [14]) is an integer array of length $|\boldsymbol{x}|$ such that $\mathcal{OLP}[i]$ is the total number of overlapping positions between adjacent occurrences of the NRE repeating substring $\boldsymbol{v}_i$ in x.
- The ***repeating substring positions covered array*** $\mathcal{RSPC}$ (introduced in [14]) is an integer array of length $|\boldsymbol{x}|$ such that $\mathcal{RSPC}[i]$ is the number of distinct positions covered by $\boldsymbol{v}_i$. Hence,

$$\mathcal{RSPC}[i] = \mathcal{LCP}[i] * \mathcal{RSF}[i] - \mathcal{OLP}[i], \tag{2}$$

  a straightforward linear-time computation.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\boldsymbol{x}$	a	b	a	c	a	b	a	b	a	c	a	b	a	c	a	b	a
$\mathcal{SA}$	17	15	5	11	1	7	13	3	9	16	6	12	2	8	14	4	10
$\mathcal{LCP}$	0	1	3	3	7	7	1	5	5	0	2	2	6	6	0	4	4
$\mathcal{RSF}$	0	9	5	5	3	3	9	3	3	0	5	5	3	3	0	3	3
$\mathcal{OLP}$	0	0	1	1	4	4	0	1	1	0	0	0	2	2	0	0	0
$\mathcal{RSPC}$	0	9	14	14	17	17	9	14	14	0	10	10	16	16	0	12	12

Figure 1: $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, $\mathcal{OLP}$ and $\mathcal{RSPC}$ arrays computed for the string $\boldsymbol{x} =$ *abacababacabacaba*.

The $\mathcal{SA}$ and $\mathcal{LCP}$ arrays are computed in linear time using the well-known implementations of Yuta Mori[4] and Puglisi [17], respectively. To compute the $\mathcal{RSF}$ array, we use the $\Theta(n)$-time algorithm given in [13]. For the $\mathcal{OLP}$ array, we describe below two simple $O(n^2)$-time algorithms from [11], comparing their efficiency with the algorithm given in [14], and implemented in [8]. We optimize the computation of the $\mathcal{RSF}$ and $\mathcal{OLP}$ arrays by replacing duplicate values with zeros to avoid recomputing them for identical substrings $\boldsymbol{v}_i$.

Finally, the $\mathcal{RSPC}$ array is computed based on Equation 2, then scanned greedily [14, Algorithm 4] in linear time to complete the expected linear-time calculation of maximal cover. For more details on the MAXCOVER software architecture and data structures, see [11][5].

4 Computing the Overlapping Positions Array

In this section, we compare three worst-case $\mathcal{O}(n^2)$ algorithms to compute the $\mathcal{OLP}$ array that underlies the computation of the maximal cover.

4.1 Original $\mathcal{OLP}$ Algorithm

We call the original algorithm proposed in [14, Algorithm 1] and implemented in [8] the **OLP-1** algorithm. For completeness, we present it as Algorithm 1 and briefly explain it here.

Algorithm 1 OLP-1 Algorithm

```
procedure COMPUTE_OLP*_ARRAY_QUADRATIC(R1, RM)
    i ← 1
    for i ← 1 to n do                          ▷ u is a distinct NRE repeating substring
        if (RSF*[i] = 0) then OLP*[i] = 0
        else                                   ▷ u occurs in SA in range [R1[i], RM[i]]
            OLP*[i] ← Compute_OLPi*(i, R1[i], RM[i])
    return OLP*
```

The OLP-1 algorithm takes $R1$ and RM integer arrays of length n as input. The $R1[i]$ and $RM[i]$ stores the r_1 and r_m values for $\boldsymbol{v}_i$. In the implementation by [11], $R1[i]$ and $RM[i]$ are computed in linear time by traversing the $\mathcal{LCP}$ array and keeping track of the rise and fall of values using a stack. Then, as OLP-1 traverses the $\mathcal{LCP}$ array from left to right, for each $\boldsymbol{v}_i$, it computes the $\mathcal{OLP}[i]$ value using the $Compute\_\mathcal{OLP}i^*()$ procedure.

The $Compute\_\mathcal{OLP}i^*()$ procedure, first copies the m starting positions of $\boldsymbol{v}_i$ found in the range $\mathcal{SA}[r_1..r_m]$, to a temporary array $\mathcal{SA}^*[r_1..r_m]$ and sorts it. Then for an occurrence of $\boldsymbol{v}_i$ in the ordered range $\mathcal{SA}^*[r_1..r_m]$ (starting at index position r_1), it checks for an adjacent overlapping occurrence of $\boldsymbol{v}_i$ using the *exrun* function introduced and defined in [4] . The *exrun* function primarily returns the run $\boldsymbol{r}$ containing the adjacent occurrences of $\boldsymbol{v}_i$, if it exists. In which case, the procedure computes the

[4] Based on the SA-IS algorithm by [16] accessed from: `https://sites.google.com/site/yuta256/`

[5] **Availability:** Open source code, binaries, and test data are available on Github at `https://github.com/hollykoponen/MAXCOVER`. The software currently runs on Linux and is untested on other OS.

procedure COMPUTE_$\mathcal{OLCP}i^*(index,\ r_1,\ r_m)$
  $\ell \leftarrow \mathcal{LCP}[index]$ ▷ ℓ is the length of $\boldsymbol{v}_i$
  $OLPi \leftarrow 0$
  Copy elements $\mathcal{SA}[r_1..r_m]$ to $\mathcal{SA}^*[r_1..r_m]$ and sort in ascending order.
  $k \leftarrow r_1$
  while $k < r_m$ **do**
    if $(\mathcal{SA}^*[k+1] - \mathcal{SA}^*[k] < \ell)$ **then** ▷ Test for overlap
      $\boldsymbol{r} \leftarrow exrun(\mathcal{SA}^*[k], \mathcal{SA}^*[k+1] + \ell - 1)$ ▷ $\boldsymbol{r} = (i, j, p)$
      $f_{\boldsymbol{r},\boldsymbol{v}_i} \leftarrow$ frequency of $\boldsymbol{v}_i$ in run $\boldsymbol{r}$
      $OLPi \leftarrow OLPi + (\ell - p) \times (f_{\boldsymbol{r},\boldsymbol{v}_i} - 1)$
      $k \leftarrow k + f_{\boldsymbol{r},\boldsymbol{v}_i} - 1$
    else
      $k \leftarrow k + 1$
    return $OLPi$

Figure 2: Compute the set of eligible runs for an NRE repeating substring

frequency of $\boldsymbol{v}_i$ in the run $\boldsymbol{r}$ and computes the total overlapping positions between adjacent occurrences of $\boldsymbol{v}_i$ in the run $\boldsymbol{r}$. Note that, this results in skipping some of the positions in $\mathcal{SA}^*[r_1..r_m]$.

The OLP-1 algorithm executes in $\mathcal{O}(n^2)$ time. The *exrun* queries are answered in constant time by preprocessing the given string in linear time. However, the quadratic execution time results from multiple traversals of a range in $\mathcal{SA}[r_1..r_m]$ (possibly n times).

4.2 Improved Algorithms: OLP-2 and OLP-3

Here we propose two improved algorithms, **OLP-2** and **OLP-3**, that use simple techniques and data structures to compute the $\mathcal{OLCP}$ array more efficiently, even though the worst-case time complexity remains $\mathcal{O}(n^2)$.

Similar to OLP-1, both algorithms traverse the $\mathcal{LCP}$ array to compute the overlaps between adjacent occurrences of $\boldsymbol{v}_i$. They also compute the values r_1 and $r_m = r_1 + m - 1$, then copy the values in $\mathcal{SA}[r_1..r_m]$ to $\mathcal{SA}^*[r_1..r_m]$ and sort them to compute the total $\mathcal{OLCP}$ overlap.

Both OLP-2 and OLP-3 use:

$$\sum_{j=1}^{m-1} (\max\{0, \mathcal{LCP}[i] - \mathcal{SA}^*[r_1 + j] + \mathcal{SA}^*[r_1 + j - 1]\}) \tag{3}$$

to determine the total overlap, if any, between adjacent occurrences of $\boldsymbol{v}_i$.

However, OLP-2 and OLP-3 do not require complicated data structures to compute overlaps — thus significantly reducing execution time. The primary difference between OLP-2 and OLP-3 is the methodology used to compute r_1 for each $\boldsymbol{v}_i$ while tracking the changes in the $\mathcal{LCP}$ array values. OLP-2 traverses the $\mathcal{LCP}$ array in reverse multiple times to compute r_1, while OLP-3 (inspired by OLP-1) computes r_1 using a stack.

In OLP-3, we use a stack that stores pairs of integer values in the format: $(index,\ r_1)$. For a $\boldsymbol{v}_i$ when a pair of values is pushed onto the stack $index = i$ and r_1 identifies the starting position of $\boldsymbol{v}_i$ in $\mathcal{SA}$. When we reference the top of the stack we write $index = top_i$ and $r_1 = top_{r1}$.

We push the index i onto the stack for each increase in the $\mathcal{LCP}$ array values (i.e. $\mathcal{LCP}[top_i] < \mathcal{LCP}[i]$). We pop the stack when the $\mathcal{LCP}$ value decreases (i.e.

Algorithm 2 OLP-2: Compute $\mathcal{OLP}$ Using $\mathcal{LCP}$ Traversal

▷ *Note: arrays begin from 0 rather than 1, so $\mathcal{SA}$ values are decremented by one.*

```
 1: procedure COMPUTE_OLP()
 2:     Initialize arrays OLP and SA* of size n full of 0's.
 3:     for i from 1 to n − 1 do
 4:         v_i ← x[SA[i]..SA[i] + LCP[i] − 1]
 5:         if (RSF[i] > 1
 6:               and ((LCP[i] = 2 and v_i[0] = v_i[1]) or LCP[i] > 2)
 7:               and MAXBORDER(v_i)) then
                        ▷ Modified MAXBORDER(v_i) returns the longest border length for v_i [18].
 8:             i' ← i
 9:             while (LCP[i' − 1] ≥ LCP[i]) do
10:                 i' ← i' − 1
11:             r_1 ← i' − 1
12:             r_m ← r_1 + RSF[i] − 1
13:             Copy SA[r_1..r_m] to SA*[r_1..r_m]
14:             Sort SA*[r_1..r_m] in ascending order
15:             sum ← 0
16:             for k from 1 to RSF[i] − 1 do
17:                 diff ← SA*[r_1 + k] − SA*[r_1 + k − 1]
18:                 if (diff < LCP[i]) then
19:                     sum ← sum + LCP[i] − diff
20:             OLP[i] ← sum
21:     return OLP
```

$\mathcal{LCP}[top_i] > \mathcal{LCP}[i]$), which means we found r_m for the current NRE substring, $\boldsymbol{v}_i$. When we pop the stack, we compute the total overlap using the ProcessStack() procedure, which is a near replica of OLP-2. We also process any remaining values on the stack at the final index using ProcessStack().

The ordered pair on the top of the stack (top_i, top_{r1}) represent the values for $\boldsymbol{v}_{top_i}$. When we pop (top_i, top_{r1}) of the stack, we know that v_i represented by $\mathcal{LCP}[i]$ is a prefix of $\boldsymbol{v}_{top_i}$. In which case, the r_1 value for $\boldsymbol{v}_i$ would at least be equal to the r_1 value of $\boldsymbol{v}_{top_i}$. We store this tentative value of r_1 in $prev_{r1}$. We stop popping elements from the stack when $\boldsymbol{v}_i$ is no longer a prefix of $\boldsymbol{v}_{top_i}$ (i.e. when the stack is empty or $LCP[top_i] < LCP[i]$)). In which case, $prev_{r1}$ stores the final r_1 value of $\boldsymbol{v}_i$.

OLP-2 requires $\mathcal{O}(n^2)$ time in the worst case. The outer ***for*** loop executes $n - 2$ times. Thus, for each $\boldsymbol{v}_i$, we sort $\mathcal{SA}^*[r_1..r_m]$ using Radix Sort in $\mathcal{O}(mk)$ time, where $m = RSF[i]$, and $k = \log_2 q$ is the key length in bits, and q is the largest value in $SA^*[r_1..r_m]$, which provides an upper bound on the number of bits in each element of $\mathcal{SA}$. However, in practice, we can treat k as a constant limited by computing capabilities. In our experiments, $k = 64$, sufficient for strings of length $n \leq 10^{18}$. Therefore, in practice, the time complexity may be treated as $\mathcal{O}(n^2)$.

Similarly, OLP-3 requires $\mathcal{O}(n^2)$ time. For each $\boldsymbol{v}_i$, we call ProcessStack() at most n times. This procedure also sorts the values in $\mathcal{SA}^*[r_1..r_m]$ for each $\boldsymbol{v}_i$ in $\mathcal{O}(mk)$ time, again yielding overall time complexity $\mathcal{O}(n^2)$.

Algorithm 3 OLP-3: Compute $\mathcal{OLCP}$ using Stack.

```
                ▷ Note: arrays begin from 0 rather than 1, so SA values are decremented by one.
 1: procedure ComputeOLCP()
 2:     Initialize arrays OLCP and SA* of size n full of 0's.
 3:     Declare a stack of ordered pairs (index, r1) for a v_i      ▷ stack.top returns (top_i, top_r1)
 4:     push(1, 0)
 5:     prev_r1 ← 0
 6:     for i from 2 to n do
 7:         if (LCP[top_i] < LCP[i]) then
 8:             push(i, i − 1)
 9:         else if (LCP[top_i] > LCP[i]) then
10:             while (stack ≠ empty and LCP[top_i] > LCP[i]) do
11:                 prev_r1 ← top_r1
12:                 ProcessStack()                              ▷ A near replica of OLP-2 [11].
13:                 pop()
14:             if (stack = empty or LCP[top_i] < LCP[i]) then
15:                 push(i, prev_r1)
16:     while (stack ≠ empty) do
17:         ProcessStack()                                      ▷ A near replica of OLP-2 [11].
18:         pop()
19:     return OLCP
```

```
procedure ProcessStack()
    v_top_i ← input[SA[top_i]..SA[top_i] + LCP[top_i] − 1]      ▷ stack.top returns (top_i, top_r1)
    if (RSF[top_i] > 1)
          and (LCP[top_i] > 2 or (LCP[top_i] = 2 and v_top_i[0] = v_top_i[1]))
          and maxborder(v_top_i, LCP[top_i])) then
        r1 ← top_r1
        rm ← r1 + RSF[top_i] − 1
        Copy SA[r1..rm] to SA*[r1..rm]
        Sort SA*[r1..rm] in ascending order
        sum ← 0
        for k from 1 to RSF[top_i] do
            diff ← SA*[r1 + k] − SA*[r1 + k − 1]
            if (diff < LCP[top_i]) then
                sum ← sum + LCP[top_i] − diff
        OLP[top_i] ← sum
```

Figure 3: ProcessStack() procedure to process the pairs $(index, r_1)$ on the stack. A near replica of OLP-2.

5 Comparison of OLP algorithms

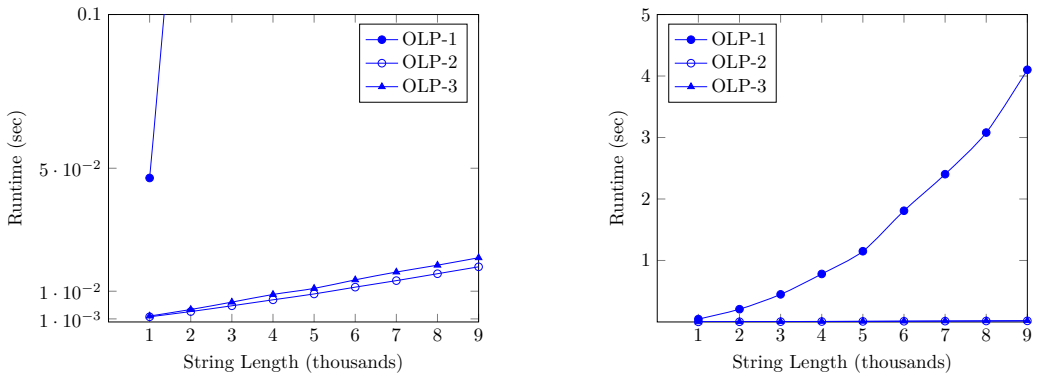
All algorithms were developed in C++ and run on a Microsoft Windows 10 Pro machine with Intel Core i9-10980XE CPU @3.00 GHz (3000 MHz, 18 Cores, 36 Logical Processors) and CORSAIR Vengeance RGB PRO 128GB (4x32GB) DDR4 3600 (PC4-28800) RAM. Testing was performed on an Ubuntu Virtual Machine (Oracle VM VirtualBox Manager) with 81804 MB Base Memory and 18 Processors.

5.1 Random Strings

We created test data of 54 unique randomly generated strings. Although ideally, an average over multiple tests per string type would be performed, only one test was performed per string because of the time it took to compute *(see analysis below)*.

This data varied based on 18 string lengths: half in the thousands ($|\boldsymbol{x}| \in \{1000, .., 9000\}$) and the other half in the millions ($|\boldsymbol{x}| \in \{1\text{M}, .., 9\text{M}\}$). These datasets also varied in alphabet size ($|\Sigma| = \{2, 3, 4\}$) to simulate binary strings, triples, and DNA strings.

The execution time of OLP-1 was significantly longer than the improved algorithms (OLP-2 and OLP-3). For instance, for $|\boldsymbol{x}| \in \{1000, .., 9000\}$, OLP-1 took up to 5 seconds to compute. In contrast, the improved algorithms (OLP-2 and OLP-3) took no more than 0.03 seconds. For $|\boldsymbol{x}| \in \{1\text{M}, .., 9\text{M}\}$, OLP-1 took too long to complete computation. In particular, for 9 million long strings the execution time was ~ 55 hours ≈ 2.3 days. As a result, its computation has several missing data points to compare with the improved algorithms. In contrast, for 9 million long strings, OLP-2 and OLP-3 took under a minute (maximum time taken was ~ 45.7 seconds) to execute.



(a) Scale between [0.001 - 0.1] seconds.

(b) Scale between [0 - 5] seconds.

Figure 4: Scaled charts (a) zoomed in; and (b) zoomed out; of runtime comparison to compute maximal covers using OLP-1, OLP-2, or OLP-3 on random binary strings. See [11] for more charts and data tables.

As noted earlier, although all three algorithms have quadratic running times, the overall overhead for OLP-2 and OLP-3 is significantly reduced due to simplified algorithm structure and fewer and simpler data structures — thus resulting in linear execution time for very long strings in practice. This is most clearly seen when comparing the algorithms on random strings over a binary alphabet $|\Sigma| = 2$ (see Figure 4).

It might be assumed that OLP-3 would perform better in practice than OLP-2 due to the use of a stack to reduce duplication in the computation of r_1. However, this was not the case for the test data used. OLP-2 performed only marginally better – most clearly seen when the string lengths are 9M. This is most likely due to the overhead required to set up the stack and to implement the push and pop routines.

Notice also that the larger the alphabet size, the faster the OLP-2 and OLP-3 algorithms perform. This is because smaller alphabet sizes are more likely to result in highly repetitive strings, which require more processing to compute overlaps.

5.2 Fibonacci Strings

We generated the Fibonacci strings using $F_0 = b$, $F_1 = a$, $F_2 = ab$, $F_k = F_{k-1}F_{k-2}$. These strings ranged in length from 3 (F_3) to 121,393 (F_{25}). We excluded the first four Fibonacci strings as they do not contain any repetitions. We stopped at F_{25} due to space limitations.

The experiments show that OLP-1 takes an order-of-magnitude longer to execute: nearly 9 hours to compute the maximal covers for F_{25}. In contrast, OLP-2 and OLP-3 take only 37 and 32 seconds, respectively.

Nevertheless, in Figure 5 we see that all three perform quadratically on Fibonacci strings, which are worst-case strings due to their highly repetitive structure. Interestingly, we see that OLP-2 performs slightly faster than OLP-3 on shorter Fibonacci strings (F_{3-15}), but slightly slower on longer Fibonacci strings (F_{16-25}). This is because:

1. OLP-2 traverses the $\mathcal{LCP}$ array backwards multiple times to determine r_1 — unlike OLP-3, which uses a stack to keep track of r_1 values during traversal, thereby eliminating duplicate computation;
2. OLP-2 performs $|x| = n$ computations of the border for v_i, whereas OLP-3 only does so when we pop off the stack, i.e. when there is a fall in $\mathcal{LCP}$ values.

We conclude that OLP-3 will be faster than OLP-2 on highly repetitive strings.

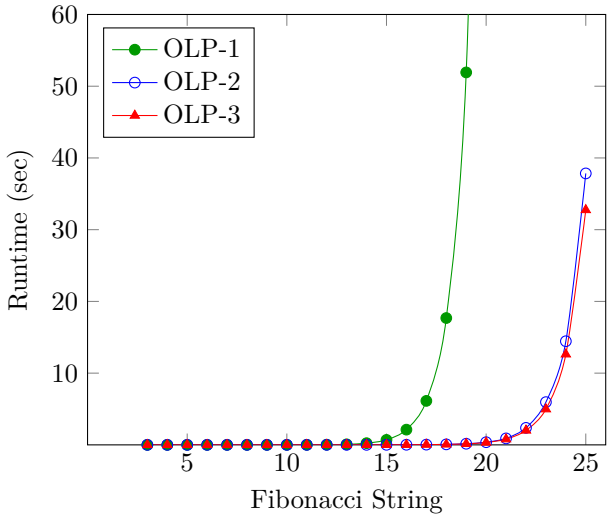


Figure 5: Plot of total runtime (seconds) when computing maximal covers of Fibonacci strings using OLP-1, OLP-2, or OLP-3.

5.3 Protein Sequences

We acquired FASTA files of protein sequence datasets from the NCBI (National Center for Biotechnology Information) databases of four taxonomically distinct model organisms: (1) *Arabidopsis thaliana* (thale cress plant), (2) *Caenorhabditis elegans* (roundworm), (3) *Drosophila melanogaster* (common fruit fly), and (4) *Homo sapiens* (human). Each protein dataset contained several thousand protein sequences (See Table 1) ranging in length from 19 to ˜36k amino acids.

Note that some amino acid letters represented are indeterminate, i.e. they could represent multiple amino acids. For example, 'Z' represents either glutamine ('Q') or

glutamate ('E'). In addition, 'X' represents 'all proteins' or gaps. For simplicity, we treated them as regular strings where each letter is treated distinctly. In future work, this analysis can be improved by addressing the ambiguous nature of these letters for better matching.

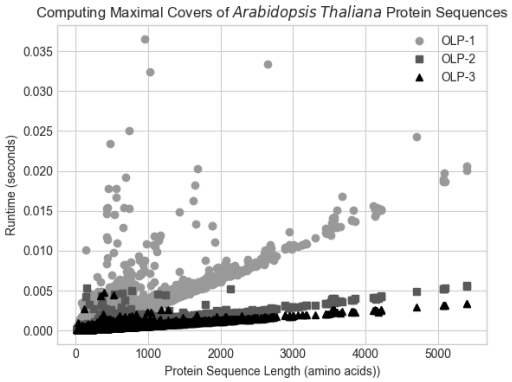


Figure 6: Runtime in seconds taken to compute maximal covers of *Arabidopsis Thaliana* protein sequences using OLP-1, OLP-2, or OLP-3 algorithms.

Species	No. of protein seq.	Protein seq. Length	Total Runtime in Seconds		
			OLP-1	OLP-2	OLP-3
Arabidopsis	~48K	19 - 5,399	64.2 s	22.3 s	16.5 s
C. elegans	~28K	19 - 15,187	47.4 s	14.5 s	10.8 s
D. melanogaster	~30K	20 - 22,948	73.7 s	23.6 s	15.3 s
human	~116K	23 - 35,990	246.3 s	79.7 s	56.6 s

Table 1: Total runtime (seconds) when computing maximal covers on sets of protein sequences of 4 different species using OLP-1, OLP-2, or OLP-3.

We computed maximal covers for each protein sequence within each set dedicated to a particular species, using OLP-1, OLP-2 and OLP-3. The algorithms perform in linear time when there are few repetitions, as shown in Fig. 6. However, when the protein sequence contains large repeats, they are processed quadratically. The outliers shown in Fig. 6 are examples of protein sequences with periodicity, for which OLP-1 took longer to compute maximal covers, unlike OLP-2 and OLP-3, which remained relatively close to linear time computation.

From Table 1, we see that OLP-3 was the fastest in each case. This is likely due to the stack data structure that better handles highly repetitive strings, as also shown by the results for Fibonacci Strings.

6 Conclusion

We conclude that OLP-2 and OLP-3 enable us to compute maximal covers in linear time for random strings and biological sequences, treated as regular strings rather than indeterminate. However, all OLP algorithms perform in quadratic time for highly repetitive strings, with OLP-2 and OLP-3 still significantly faster than OLP-1.

In the future, we plan to further improve the performance of OLP-3 by using the system stack instead of the program stack. Recently, Czajka & Radoszewski in [6]

gives an $\mathcal{O}(n \log n)$ implementation to compute maximal covers. We plan to compare the performance of our algorithms with this implementation soon. Moreover, it is worth considering that OLP-3 could potentially exhibit improved performance when utilizing a static stack instead of a dynamic stack. Therefore, conducting additional experiments to compare the two approaches would be beneficial.

Furthermore, we can conduct additional experiments to obtain an average by performing multiple tests for each string type. Additionally, when evaluating maximal covers, we can improve experiments involving protein sequences by incorporating considerations for indeterminate string matching.

An immediate question arises whether maximal cover computation of a string $\boldsymbol{x}$ (at least those on small σ) might be an interesting compression technique. However, our experimental evaluations show that several classes of strings have very short maximal covers (e.g. $|u| \leq 2$). This indicates that maximal covers may not be useful as a compression technique.

Another interesting question arises whether computing an *iterated* maximal cover set — that is, a set of covers covering the string $\boldsymbol{x}$ — might be useful as a compression technique. We plan to investigate this in future work.

7 Acknowledgements

The authors thank Viktor Melnyk for his contributions to this paper. In particular, for setting up the software to compute maximal covers using OLP-1, which included the incorporation of the $\mathcal{SA}/\mathcal{LCP}$ software, implementation of the optimal cover algorithm, including the implementation of the OLP-1 algorithm, and preliminary testing of its effectiveness.

References

1. A. Alatabbi, A. S. M. S. Islam, M. S. Rahman, J. Simpson, and W. F. Smyth: *Enhanced covers of regular & indeterminate strings using prefix tables.* J. Automata, Languages & Combinatorics, 21(3) 2016, pp. 131–147.
2. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasi–periodicities in strings*, Tech. Rep. 90.5, The Leonadro Fibonacci Institute, Trento, Italy, 1990.
3. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* Theoret. Comput. Sci., 119(2) 1993, pp. 247–265.
4. 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.
5. R. Cole, C. S. Iliopoulos, M. Mohamed, W. F. Smyth, and L. Yang: *The complexity of the minimum k-cover problem.* J. Automata, Languages & Combinatorics, 10-5/6 2005, pp. 641–653.
6. P. Czajka and J. Radoszewski: *Experimental evaluation of algorithms for computing quasiperiods.* Theoretical Computer Science, 854 2021, pp. 17–29.
7. T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, and W. Tyczyński: *Enhanced string covering.* Theoretical Computer Science, 506 2013, pp. 102–114.
8. G. B. Golding, H. Koponen, N. Mhaskar, and W. F. Smyth: *Computing maximal covers for protein sequences.* Journal of Computational Biology, 30(2) 2023, pp. 149–160.
9. C. S. Iliopoulos and W. F. Smyth: *On-line algorithms for k-covering*, in Proc. 9th Australasian Workshop on Combinatorial Algs. (AWOCA), 1998, pp. 97–106.
10. T. Kociumaka, J. Radoszewski, W. Rytter, S. P. Pissis, and T. Waleń: *Fast algorithm for partial covers in words.* Algorithmica, 73(1) 2015, pp. 217 – 233.

11. H. Koponen: *Efficient implementation & application of maximal string covering algorithms.* MSc Thesis, McMaster University, 2022, p. 58.
12. Y. Li and W. F. Smyth: *Computing the cover array in linear time.* Algorithmica, 32(1) 2002, pp. 95–106.
13. N. Mhaskar and W. F. Smyth: *Frequency covers for strings.* Fundamenta Informaticae, 163(3) 2018, pp. 275–289.
14. N. Mhaskar and W. F. Smyth: *String covering with optimal covers.* Journal of Discrete Algorithms, 51 2018, pp. 26–38.
15. N. Mhaskar and W. F. Smyth: *String covering: A survey.* submitted for publication, 2022.
16. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced–sorting.* Data Compression Conference, 0 2009, pp. 193–202.
17. S. J. Puglisi and A. Turpin: *Spacetime tradeoffs for longest-common-prefix array computation.* Proc. 19th Internat. Symp. Algs. & Comp., 2008, pp. 124–135.
18. B. Smyth: *Computing Patterns in Strings*, Pearson/Addison–Wesley, 2003.

Periodicity of Degenerate Strings

Estéban Gabory[1], Eric Rivals[2], Michelle Sweering[1], Hilde Verbeek[1], and Pengfei Wang[2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands*
{esteban.gabory,michelle.sweering,hilde.verbeek}@cwi.nl
[2] LIRMM, Université Montpellier, CNRS, Montpellier, France**
{rivals,pengfei.wang}@limmr.fr

Abstract. The notion of periods is key in stringology, word combinatorics, and pattern matching algorithms. A string has period p if every two letters at distance p from each other are equal.
There has been a growing interest in more general models of sequences which can describe uncertainty. An important model of sequences with uncertainty are degenerate strings. A degenerate string is a string with "undetermined" symbols, which can denote arbitrary subsets of the alphabet Σ. Degenerate strings have been extensively used to describe uncertainty in DNA, RNA, and protein sequences using the IUPAC code (Biochemistry, 1970).
In this work, we extend the work of Blanchet-Sadri et al. (2010) to obtain the following results about the combinatorial aspects of periodicity for degenerate strings:

- We compare three natural generalizations of periodicity for degenerate strings, which we refer to as weak, medium and strong periodicity. We define the concept of total autocorrelations, which are quaternary vectors indicating these three notions of periodicity.
- We characterize the three families of period sets, as well as the family of total autocorrelations, for each alphabet size. In particular, we prove necessary conditions period sets should satisfy and, to prove sufficiency, we show how to construct a degenerate string which gives rise to particular period sets.
- For each notion of periodicity, we (asymptotically) count the number of period sets, by combining known techniques from partial words with recent results from number theory.
- Moreover, we show that all families of period sets, as well as the family of total autocorrelations, form lattices under a suitably defined partial ordering.
- We compute the population of weak, medium and strong period sets (i.e., the number of strings with that period set). We also compute the population of total autocorrelations.

Keywords: Periodicity, Degenerate string, Indeterminate string, Autocorrelation.

1 Introduction

Sequences of letters taken over an alphabet Σ, also called strings or words, are used to represent texts in natural languages, biomolecules such as DNA, RNA or proteins, or the sequence of states in dynamical systems. The notion of periodicity proves to be crucial for investigating word combinatorics [18], the properties of symbolic dynamical systems [19], or to design efficient pattern matching algorithms [9].

However, the classic notion of a string is insufficient to handle undetermination. Instead, sequences of sets of letters were considered and several definitions that generalize the classic notion of strings have been proposed such as partial strings and

* Address: P.O. Box 94079 - 1090 GB Amsterdam THE NETHERLANDS
** Address: 161 rue Ada - 34095 Montpellier cedex 5 FRANCE

Estéban Gabory, Eric Rivals, Michelle Sweering, Hilde Verbeek, Pengfei Wang: Periodicity of Degenerate Strings, pp. 42–56.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

degenerate strings. In partial strings, the undetermined symbol $\diamond$ can represent any letter in Σ. In degenerate strings, we can have multiple different undetermined symbols, each representing a specified non-empty subset of Σ from which we must choose a letter. Degenerate strings thus generalize partial strings.

Regarding representations of biomolecules, reasons or causes of undetermination are multiple. First, undetermination appears in DNA/RNA sequences when sequencing machines fail to identify a precise nucleotide due to a noisy signal (which is frequent with the third generation of deep sequencing technologies, like Oxford Nanopore [21]. Second, undetermined symbols are used to represent binding sites (sequence regions at which biomolecules chemically bind to each other) at positions where alternative residues are observed. There exist databases of binding site representations using degenerate strings (JASPAR [5], HOCOMOCO [16]), which serve to identify new binding sites in genomes. Third, in the context of pangenomics, which investigates the genetic variations observed within a population, undetermined symbols serve to represent the variant nucleotides at a given genomic position in this population [28]. If only nucleotidic substitutions are considered, degenerate strings are adequate, but if multiple insertions/deletions need to be represented then elastic degenerate strings are preferred [13].

Related works. In classical finite strings, a period denotes the possibility of a word to self-overlap. In seminal articles Guibas and Odlyzko introduced the notion of period set of a finite word (and its binary representation the autocorrelation), proposed a characterization of it, investigated how the autocorrelation controls the probability of absence of a word in random texts, and extended it into correlation to study overlaps between pair of words. An alternative simpler proof of the "Fine and Wilf" theorem for period sets was given [10] and the set of period sets for words of length n and related combinatorics was investigated [24], while the asymptotic convergence on the number of period sets has recently been solved [25].

Our goal is to study combinatorics of period sets for string definitions allowing undetermined symbols. In a first step, Blanchet-Sadri et al conducted a combinatorial study in the case of partial strings [4,3] (similar to that on classical strings [23]), and investigated related algorithmic questions [2]. However, partial strings are inadequate to represent undetermination arising in biomolecules, while degenerate strings are. As degenerate strings generalize partial strings, we study combinatorics of period sets in the case of degenerate strings. In a related work, Iliopoulos and Radoszewski[14] showed that the weak period array of a degenerate string can be computed in $O(n\sqrt{n})$ and $O(n)$ space, while its strong period array cannot be computed in $O(n^{2-\epsilon}|\Sigma|^{O(1)})$ time if the Strong Exponential Time Hypothesis holds. Other algorithmic questions related to degenerate strings have also been investigated [6,12,11,26].

Contributions. For degenerate strings, three notions of period sets (weak, medium, and strong) are necessary and we characterize period sets for each, exhibiting necessary and sufficient conditions (Section 3). Then, we count the number of period sets for degenerate strings of length n for each notion and study its convergence using recent results from number theory (Section 4). We investigate the structure of the set of period sets in Section 5, and how many degenerate strings share a given period set (i.e., the population of a period set) extending the graph approach proposed in [3]. Finally, we outline some directions for future work (Section 7).

2 Preliminaries

A *classic string* $u = u[0\,..\,n-1] \in \Sigma^n$ of length n is a sequence of n letters over a non-empty finite *alphabet* Σ. For any $0 \leq i \leq j \leq n-1$, we denote the *substring* starting at position i and ending at position j with $u[i\,..\,j]$. In particular, $u[0\,..\,j]$ denotes a *prefix* of u and $u[i\,..\,n-1]$ a *suffix*. Throughout this paper, all our strings and vectors will be zero-indexed.

2.1 Degenerate strings

A *degenerate alphabet* Δ over Σ is a set of subsets of Σ, i.e., $\Delta \subseteq \mathcal{P}(\Sigma)$, where $\mathcal{P}(\Sigma)$ is the power set of Σ. We call the elements of a degenerate alphabet undetermined symbols, or *symbols* for short. A *degenerate string* $\hat{w} = \hat{w}[0\,..\,n-1] \in \Delta^n$ is a string of length n over the degenerate alphabet Δ. We define the *size* of $\hat{w}$ as the sum of the cardinalities of its symbols $\|\hat{w}\| = \sum_{i=0}^{n-1} |\hat{w}[i]|$.

Degenerate strings are used to model uncertainty. Undetermined symbols are used to denote all possible letters at a given position. This way, the degenerate string defines a language of words over the original alphabet Σ. Specifically, we define the *language* of a degenerate string $\hat{w}$ of length n over degenerate alphabet $\Delta \subseteq \mathcal{P}(\Sigma)$ as

$$\mathcal{L}(\hat{w}) = \{w \in \Sigma^n \mid \forall i \in \{0, \ldots, n-1\} \quad w[i] \in \hat{w}[i]\}\,.$$

Example 1. Let $\hat{w} = \left\{\begin{matrix}\mathtt{a}\\ \mathtt{b}\end{matrix}\right\} \cdot \left\{\begin{matrix}\mathtt{b}\\ \mathtt{c}\end{matrix}\right\} \cdot \{\mathtt{c}\}$. Note that $\hat{w}$ has length $|\hat{w}| = 3$, size $\|\hat{w}\| = 5$ and language $\mathcal{L}(\hat{w}) = \{\mathtt{abc}, \mathtt{acc}, \mathtt{bbc}, \mathtt{bcc}\}$.

A *hollow string* $\hat{w}$ is a degenerate string such that $\hat{w}[i] = \emptyset$ for at least one $i \in \{0, \ldots, n-1\}$, or equivalently a degenerate string such that $\mathcal{L}(\hat{w}) = \emptyset$. We say two degenerate strings $\hat{x}$ and $\hat{y}$ of length n over the same degenerate alphabet *match*, if for all $i \in \{0, \ldots, n-1\}$ the intersection $\hat{x}[i] \cap \hat{y}[i]$ is non-empty.

Sometimes, there are some restrictions on the degenerate alphabet Δ. In motif searching [22,27] for example, the k-motif which is a k-length degenerate string, consists of symbols such that the union of them is Σ and no symbol is a subset of another symbol. We will take $\Delta = \mathcal{P}(\Sigma) \setminus \emptyset$ unless stated otherwise, i.e., we have an undetermined symbol for every non-empty subset of Σ. This is the most general choice of Δ which excludes hollow strings. Hollow strings have an empty language, which is not very interesting when studying periodicity. Moreover, a nice consequence of excluding hollow strings is that now no two degenerate strings correspond to the same language.

2.2 Periodicity

Before we introduce the notion of periodicity in degenerate strings, we first recall its definition in the case of classic strings over the alphabet Σ. One such definition is as follows.

Definition 2 (Period of a string). *A string $u = u[0\,..\,n-1]$ has period $p \in \{0, 1, \ldots, n-1\}$ if and only if $u[0\,..\,n-p-1] = u[p\,..\,n-1]$, i.e., for all $0 \leq i \leq n-p-1$, we have $u[i] = u[i+p]$.*

There are several other equivalent definitions, e.g. one could require that $u[i] = u[j]$ whenever $i \equiv j \mod p$. Generalizing the notion of periodicity to degenerate strings is therefore not straightforward. Holub and Smyth introduced the concept of quantum and deterministic periods [11]. Blanchet-Sadri et al. call the same concepts weak and strong periods in the context of partial words [3]. We use the naming convention from Blanchet-Sadri et al. with the difference that we additionally define the concept of medium periodicity, which coincides with strong periodicity in the case of partial words but exhibits a different behaviour in the case of degenerate strings.

First, we recall the definition of weak periodicity.

Definition 3 (Weak period of a degenerate string). *A degenerate string* $\hat{w} = \hat{w}[0\,..\,n-1]$ *has weak period* $p \in \{0, 1, \ldots, n-1\}$ *if and only if* $\hat{w}[0\,..\,n-p-1]$ *matches* $\hat{w}[p\,..\,n-1]$*, i.e., for all* $0 \leq i \leq n-p-1$ *we have* $\hat{w}[i] \cap \hat{w}[i+p] \neq \emptyset$*.*

This is the most flexible type of periodicity, for which we want two strings in the language to overlap by $n - p$ letters. This type is most suitable when we use the degenerate string to model variations in a set of related strings.

Although periodicity p implies periodicity kp for classic strings, this is not the case for weak periods in degenerate strings (see Example 6). If we want to require this, we need a second stronger notion: medium periodicity.

Definition 4 (Medium period of a degenerate string). *A degenerate string* $\hat{w} = \hat{w}[0\,..\,n-1]$ *has medium period* $p \in \{0, 1, \ldots, n-1\}$ *if and only if for any* $0 \leq i, j \leq n-1$ *such that* $i \equiv j \pmod p$ *we have* $\hat{w}[i] \cap \hat{w}[j] \neq \emptyset$*.*

An equivalent definition is: a degenerate string $\hat{w}$ has medium period p if every multiple kp with $k \in \mathbb{N}$ is a weak period of $\hat{w}$. First notice that 0 is both medium period and weak period by definition.

Finally, we define strong periodicity.

Definition 5 (Strong period of a degenerate string). *A degenerate string* $\hat{w}$ *has strong period* p *if there exists a string* $w \in \mathcal{L}(\hat{w})$ *with period* p*.*

This is the most restrictive type of periodicity, where we require a word in the language to overlap itself. This type is most suitable when we use the degenerate string to model one specific string, of which letters are not precisely known.

Given a degenerate string $\hat{w}$, we denote its sets of weak, medium, and strong periods by $P^w(\hat{w})$, $P^m(\hat{w})$ and $P^s(\hat{w})$ respectively. From the definitions, we can easily see that $P^s \subseteq P^m \subseteq P^w$. We illustrate the difference between the different types of period sets with the following example.

Example 6. Let $\hat{w} = \begin{Bmatrix}\mathtt{a}\\\mathtt{b}\end{Bmatrix} \cdot \begin{Bmatrix}\mathtt{b}\\\mathtt{c}\end{Bmatrix} \cdot \begin{Bmatrix}\mathtt{b}\\\mathtt{c}\end{Bmatrix} \cdot \{\mathtt{c}\} \cdot \begin{Bmatrix}\mathtt{a}\\\mathtt{c}\end{Bmatrix}$. Then $P^w(\hat{w}) = \{0, 1, 2, 4\}$, $P^m(\hat{w}) = \{0, 2, 4\}$ and $P^s(\hat{w}) = \{0, 4\}$.

Finally, we denote the set of all possible weak, medium and strong period sets of degenerate strings of length n by Ω_n^w, Ω_n^m and Ω_n^s respectively.

2.3 Autocorrelations

One useful way to represent period sets is using autocorrelations, a concept introduced in 1981 by Guibas and Odlyzko [8]. The autocorrelation of a string $w \in \Sigma^n$ is the binary vector $s \in \{0, 1\}^n$ indicating its period set. We extend this definition by defining different autocorrelations for degenerate strings corresponding to different types of period sets.

Definition 7 (Autocorrelation of degenerate string). *For every degenerate string $\hat{w}$, its weak (resp. medium, resp. strong) autocorrelation is the binary vector $s \in \{0,1\}^n$ such that*

$$s[i] = \begin{cases} 1 & \text{if } i \text{ is a weak (resp. medium,} \\ & \text{resp. strong) period of } \hat{w} \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in \{0,\ldots,n-1\}.$$

We will denote the weak, medium and strong autocorrelations by $\hat{s}^w$, $\hat{s}^m$ and $\hat{s}^s$ respectively.

In [3], Blanchet-Sadri et al. take advantage of ternary vectors to simultaneously represent the weak and strong period sets of partial words. In our work, we introduce the concept of a total autocorrelation as a quaternary vector indicating these three notions of autocorrelations.

Definition 8 (Total autocorrelation of degenerate string). *For a degenerate string $\hat{w}$, its total autocorrelation is the sum of the weak, medium and strong autocorrelation $\hat{s} = \hat{s}^w + \hat{s}^m + \hat{s}^s$.*

We can equivalently define $\hat{s} \in \{0,1,2,3\}^n$ to be the vector such that

$$\hat{s}[i] = \begin{cases} 0 & \text{if } i \notin P^w & \text{(not a period)} \\ 1 & \text{if } i \in P^w \setminus P^m & \text{(weak period)} \\ 2 & \text{if } i \in P^m \setminus P^s & \text{(weak and medium period)} \\ 3 & \text{if } i \in P^s & \text{(weak, medium and strong period)} \end{cases}$$

for all $i \in \{0,\ldots,n-1\}$. To illustrate the weak, medium, strong and total autocorrelations, we review the degenerate string from example 6.

Example 9. Let $\hat{w} = \left\{\begin{matrix}\mathtt{a}\\\mathtt{b}\end{matrix}\right\} \cdot \left\{\begin{matrix}\mathtt{b}\\\mathtt{c}\end{matrix}\right\} \cdot \left\{\begin{matrix}\mathtt{b}\\\mathtt{c}\end{matrix}\right\} \cdot \{\mathtt{c}\} \cdot \left\{\begin{matrix}\mathtt{a}\\\mathtt{c}\end{matrix}\right\}$. Then $\hat{w}$ has weak autocorrelation $\hat{s}^w = 11101$, medium autocorrelation $\hat{s}^m = 10101$, strong autocorrelation $\hat{s}^s = 10001$ and total autocorrelation $\hat{s} = 31203$.

3 Characterization of total autocorrelations

In this section, we characterize the total (and hence also the weak, medium and strong) autocorrelation vectors of degenerate strings.

First, note that if the alphabet is unary, there exists a unique degenerate string of length n, which has total autocorrelation 3^n. Thus, we will henceforth assume that $|\Sigma| \geq 2$.

Theorem 10. *Let $P^s \subseteq P^m \subseteq P^w \subseteq \{0,\ldots,n-1\}$. Then P^w, P^m and P^s are respectively the weak, medium, and strong period sets of some non-hollow degenerate string $\hat{w}$ of length n if and only if*

A. $0 \in P^s$,
B. for all $p \in P^w$ *we have* $p \geq n/2 \implies p \in P^s$,
C. $p \in P^m$ *if and only if for all* $k \in \mathbb{N}$ *with* $kp \in \{0,\ldots,n-1\}$ *we have* $kp \in P^w$, *and*

D. $p \in P^s$ *if and only if for all* $k \in \mathbb{N}$ *with* $kp \in \{0, \ldots, n-1\}$ *we have* $kp \in P^s$.

Furthermore, these conditions are sufficient for any specific alphabet Σ *of cardinality at least 3. For a binary alphabet, we additionally require that* $P^m = P^s$.

Proof. We will first prove the necessity of these four properties. Let $\hat{w}$ be a degenerate string with weak, medium and strong period sets P^w, P^m and P^s respectively.

(I) Since $\hat{w}$ is not hollow, there exists a string $w \in \mathcal{L}(\hat{w})$. Since w has period 0, the degenerate string $\hat{w}$ has strong period 0.

(II) For every $p \in P^w$, there exist two strings $w_1, w_2 \in \mathcal{L}(\hat{w})$ such that $w_1[p\,..\,n-1] = w_2[0\,..\,n-1-p]$. Note that $w =: w_2[0\,..\,p-1]w_1[p\,..\,n-1] \in \mathcal{L}(\hat{w})$ as well. Moreover, since $p \geq n/2$, we have that $i \equiv j \mod p$ implies
- $i = j$ and hence $w[i] = w[j]$, or
- $j = i + p$ in which case $w[i] = w_2[i] = w_1[i+p] = w[i+p] = w[j]$, or
- $i = j + p$ and analogously $w[i] = w[j]$.

Thus, w has period p. Consequently, $\hat{w}$ has strong period p.

(III) This is the definition of medium periodicity.

(IV) Since $p \in P^s$, there exists $w \in \mathcal{L}(\hat{w})$ such that w has period p. If $kp < n$, then w also has period kp. Therefore kp is also a strong period of $\hat{w}$. Conversely, if kp is a strong period for all natural k such that $kp < n$, then trivially $1 \cdot p$ is a strong period as well.

To prove sufficiency, assume that $P^s \subseteq P^m \subseteq P^w \subseteq \{0, \ldots, n-1\}$ satisfy the four properties. We construct the degenerate string $\hat{w}$ such that

$$\hat{w}[i] = \begin{cases} \{\mathtt{a}, \mathtt{b}\} & \text{if } i = 0 \\ \{\mathtt{a}, \mathtt{c}\} & \text{if } i \in P^s \setminus \{0\} \\ \{\mathtt{b}, \mathtt{c}\} & \text{if } i \in P^w \setminus P^s \\ \{\mathtt{c}\} & \text{otherwise} \end{cases}$$

and verify that it has weak, medium and strong period sets P^w, P^m and P^s respectively.

- Note that every pair of sets intersects, except for $\{\mathtt{a}, \mathtt{b}\}$ and $\{\mathtt{c}\}$. Thus p is a weak period if and only if $\hat{w}[p] \neq \{\mathtt{c}\}$, which is indeed if and only if $p \in P^w$.
- The medium period set is defined by the weak period set by property (III). Thus, since $\hat{w}$ has the specified weak period set P^s, it also has the corresponding medium period set P^m.
- Note that for all $p \in P^s \setminus \{0\}$, the classic string $w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^n$ such that

$$w[i] = \begin{cases} \mathtt{a} & \text{if } p \mid i \\ \mathtt{c} & \text{otherwise} \end{cases}$$

is in $\mathcal{L}(\hat{w})$. Therefore $\hat{w}$ has strong period p. However, if $p \notin P^s$, then either $p \notin P^w$ (in which case p is not a weak period and thus not a strong period either) or there exists $k \in \mathbb{N}$ such that kp is a strong period with $n/2 \leq kp \leq n-1$ by property (II). It follows that

$$\hat{w}[0] \cap \hat{w}[p] \cap \hat{w}[kp] = \{\mathtt{a}, \mathtt{b}\} \cap \{\mathtt{b}, \mathtt{c}\} \cap \{\mathtt{a}, \mathtt{c}\} = \emptyset.$$

Therefore p is not a strong period.

We conclude that the four properties characterize the three period sets.

Note that the construction above uses an alphabet of size 3 and thus characterizes all possible total autocorrelations, even if we restrict to some specific alphabet Σ of cardinality at least 3. For binary alphabets, note that degenerate strings are the same as partial words, because they both have the same degenerate alphabet $\Delta = \{\{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}\}$. Thus, every medium period is a strong period. In other words, the autocorrelation is in $\{0, 1, 3\}^n$. Conversely, any such autocorrelation is the autocorrelation of the binary degenerate string $\hat{w} \in \Delta^n$ such that

$$\hat{w}[i] = \begin{cases} \{\mathtt{a}\} & \text{if } i = 0 \\ \{\mathtt{a}, \mathtt{b}\} & \text{if } i \in P^w \setminus \{0\} \\ \{\mathtt{b}\} & \text{otherwise,} \end{cases}$$

because p is a weak period of $\hat{w}$ if and only if $p \in P^w$, and because the medium — and in the binary case also strong — periods are defined by the weak periods by property (III). □

4 Structure of autocorrelations

In this section, we take a closer look at the structure and number of weak, medium and strong autocorrelations.

4.1 Weak autocorrelations

We show that Ω_n^w, the set of autocorrelations of degenerate strings of length n with respect to weak periodicity, equals $\{1\}\{0,1\}^{n-1}$. This result holds irrespective of (non-unary) alphabet size.

Theorem 11. $\Omega_n^w = \{1\}\{0,1\}^{n-1}$

Proof. Let $s \in \{1\}\{0,1\}^{n-1}$. We construct a corresponding degenerate string over a binary alphabet $\{\mathtt{a}, \mathtt{b}\}$. We set $\hat{w}[0] = \{\mathtt{a}\}$, and for every $1 \leq i \leq n-1$, we set $\hat{w}[i] = \{\mathtt{b}\}$ if $s[i] = 0$ and $\left\{\begin{matrix}\mathtt{a}\\ \mathtt{b}\end{matrix}\right\}$ if $s[i] = 1$. It can easily be seen that s is the weak autocorrelation of $\hat{w}$. Note that any pair of symbols at position $i, j \geq 1$ in $\hat{w}$ has nonempty intersection $\{\mathtt{b}\}$. Therefore, we only need to observe that $\hat{w}[0]$ and $\hat{w}[p]$ match if and only if $s[p] = 1$. □

4.2 Medium and strong autocorrelations

Blanchet-Sadri et al. define $R(v)$ as the irreducible period set of partial word v and Φ_n to be the set of all irreducible period sets of partial words of length n [3]. They show that $R(v)$ is a primitive set, a set wherein no two numbers divide each other, and that any primitive subset of $\{1, \ldots, n-1\}$ is an irreducible period set. They also show that there is a one-to-one mapping between Φ_n and the number of period sets of partial words of length n. Consequently, to count the number of period sets, it is sufficient to count the number of primitive subsets of $\{1, \ldots, n-1\}$. In this section, we will similarly characterize the sets of medium and strong autocorrelations of degenerate strings.

Let us fix an integer interval $I = [0\,..\,n-1]$. Given a subset $P \subseteq I$, we write $\langle P\rangle = \{kp \in I \mid p \in P,\ k \in \mathbb{Z}_{\geq 0}\}$ and say that P *generates* $\langle P\rangle$. We say that P is *closed under multiplication* if $\langle P\rangle = P$. Note that this implies in particular that $0 \in P$.

This is a direct reformulation of Theorem 10 in the case of medium and strong autocorrelations:

Corollary 12. *The subsets of* $[0\,..\,n-1]$ *that are medium (resp. strong) period sets of a degenerate string* $\hat{w}$ *having length* n *over any fixed alphabet of cardinality at least* 2 *are exactly the multiplicative subsets of* $[0\,..\,n-1]$. *In particular, one has* $\Omega_n^m = \Omega_n^s$ *for any* $n \geq 1$.

We say that a set P of integers is *primitive* if it does not contain a pair $i \neq j$ such that i divides j. Equivalently, that means that $\langle P\rangle = \langle P'\rangle$ only if $P \subseteq P'$. Note that if P is a primitive set containing 0, then $P = \{0\}$.

Lemma 13. *Let* $I = [0\,..\,n-1]$. *Any set* $P \subseteq I$ *which is closed under multiplication contains a unique minimum set* P_{prim} *generating it, and this set is primitive.*

Therefore, primitive subsets in I *are in a* 1*-to-*1 *correspondence with multiplicative sets in* I.

Proof. The subset P_{prim} can be obtained by taking every pair $i \neq j$ with i dividing j and removing j. The order of removal does not affect the result by transitivity of the divisibility relation. Note that if $P \neq \{0\}$, then 0 will be removed from P as it is a multiple of every integer. The resulting set generates P and is primitive by construction. It is also the minimum generating set because if $\langle P_{\text{prim}}\rangle = P = \langle P'\rangle$ for some $P' \subseteq I$ then $P_{\text{prim}} \subseteq P'$ from the definition of a primitive set.

The reciprocal mappings $P \mapsto P_{\text{prim}}$ and $P \mapsto \langle P\rangle$ hence form a 1-to-1 correspondence. □

4.3 Counting the number of period sets

We have seen that weak period sets can be any subset of $\{0, \ldots, n-1\}$ containing 0. It follows that there are exactly $|\Omega_n^w| = 2^{n-1}$ weak period sets of strings of length n. Counting the number of medium and strong period sets is a lot more complex, but luckily we can rely on tools from the literature.

In [3], Blanchet-Sadri et al. provide upper and lower bounds on the number of autocorrelations of partial words of length n. They use a result by Erdős [7] to determine the logarithm of the number of primitive sets with elements smaller than n (and hence the number of autocorrelations of partial words of length n) up to a factor of two. However, recently there have been major developments concerning the number of primitive sets. Let $Q(n)$ be the number of primitive sets with largest element at most n. Angelo proved that $\ln(Q(n))/n$ converges to some constant α [1]. Liu, Pach and Palincza [17] and McNew [20] proved that α is effectively computable and computed upper and lower bounds on them.

Theorem 14 (Liu, Pach and Palincza [17], McNew [20]). *For any* $\epsilon > 0$, *we have*

$$Q(n) = \alpha^{n\left(1+O\left(\exp\left((-1+\epsilon)\sqrt{\log n \log\log n}\right)\right)\right)}.$$

The constant α *is effectively computable and* $1.5729 < \alpha < 1.5745$.

Since $|\Omega_n^m| = |\Omega_n^s| = Q(n-1)$, this implies the same asymptotic behaviour for the number of medium and strong period sets.

Corollary 15. *For any $\epsilon > 0$, we have*

$$|\Omega_n^m| = |\Omega_n^s| = \alpha^{n\left(1+O\left(\exp\left((-1+\epsilon)\sqrt{\log n \log\log n}\right)\right)\right)}.$$

The constant α is effectively computable and $1.5729 < \alpha < 1.5745$.

Proof. By Theorem 14, it follows directly that

$$|\Omega_n^m| = |\Omega_n^s| = \alpha^{(n-1)\left(1+O\left(\exp\left((-1+\epsilon)\sqrt{\log(n-1)\log\log(n-1)}\right)\right)\right)}.$$

Then one can notice that

$$\begin{aligned}
&(n-1)\left(1+O\left(\exp\left((-1+\epsilon)\sqrt{\log(n-1)\log\log(n-1)}\right)\right)\right)\\
&= n\left(1-\frac{1}{n}+\frac{n-1}{n}O\left(\exp\left((-1+\epsilon)\sqrt{\log n\log\log n}\right)\right)\right)\\
&= n\left(1+O\left(\exp\left((-1+\epsilon)\sqrt{\log n\log\log n}\right)\right)\right),
\end{aligned}$$

because $\frac{1}{n} = \exp(-\log n) = \exp\left(-\sqrt{\log^2 n}\right) = O\left(\exp\left(-\sqrt{\log n \log\log n}\right)\right)$. □

5 Lattice structure

Blanchet-Sadri et al. show that the sets of all binary and ternary autocorrelations of partial words of length n both form lattices under set inclusion of the corresponding period sets [3]. Moreover, they show these lattices satisfy the Jordan-Dedekind condition.

We investigate the structure of individual autocorrelations, as well as the total autocorrelation of degenerate strings. We show that Ω_n^w, Ω_n^m and Ω_n^s all follow lattice structure under set intersection and set union, and hence satisfy the Jordan-Dedekind condition. We also show the set of total autocorrelations is a lattice with respect to product order. Due to similarity with [3], we refer to Appendix A for the definitions of respectively the weak, medium, and strong autocorrelations, and for the proof of Theorem 16.

Theorem 16. *$(\Omega_n^w, \subseteq)$, $(\Omega_n^m, \subseteq)$ and $(\Omega_n^s, \subseteq)$ are lattices with respect to the inclusion order.*

In a poset (and hence also in a lattice), a *chain* is defined as a subset of totally ordered elements. The length of a chain is its cardinality minus one. The Jordan-Dedekind condition requires that all maximal chains between the same elements have equal length. If a lattice is distributive (i.e., $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ for all x, y, z in the lattice) and finite, then it satisfies the Jordan-Dedekind condition.

Since the meet and join of weak, medium and strong period sets correspond to set intersection and set union, we have the following corollary.

Corollary 17. *The lattices $(\Omega_n^w, \subseteq)$, $(\Omega_n^m, \subseteq)$ and $(\Omega_n^s, \subseteq)$ are all distributive and thus satisfy the Jordan-Dedekind condition.*

Let Ψ_n be the set of all total autocorrelations of length n. We will now show that Ψ_n is also a lattice with respect to product order (i.e., $u \leq v$ if and only if $u_i \leq v_i$ for all indices i) using the results for the individual families of period sets.

Theorem 18. *$(\Psi_n, \leq)$ is a lattice with respect to product order.*

Proof. To show that this is a lattice, we need to show that its meet ($\wedge$) and join operations ($\vee$) are well-defined. In this case, the meet will be the pointwise minimum of two total autocorrelations, while the join will be the minimum of all total autocorrelations greater than both. Formally,

$$u \wedge v = \min(u, v) \quad \text{and} \quad u \vee v = \bigwedge_{w \in \Psi_n \text{ s.t. } w \geq u,v} w.$$

Meet Let u and v be two total autocorrelations. Let P^w, P^m, P^s and Q^w, Q^m, Q^s be the weak, medium and strong period sets of u and v respectively. We define $R^w = P^w \cap Q^w$, $R^m = P^m \cap Q^m$ and $R^s = P^s \cap Q^s$. Note that $R^s \subseteq R^m \subseteq R^w \subseteq \{0, \ldots, n-1\}$ and
- $0 \in P^s \cap Q^s = R^s$,
- for all $p \in R^w = P^w \cap Q^w$ we have $p \geq n/2 \implies p \in P^s \cap Q^s = R^s$,
- $p \in R^m = P^m \cap Q^m$ if and only if for all $k \in \mathbb{N}$ with $kp \in \{0, \ldots, n-1\}$ we have $kp \in P^w \cap Q^w = R^w$, and
- $p \in R^s = P^s \cap Q^s$ if and only if for all $k \in \mathbb{N}$ with $kp \in \{0, \ldots, n-1\}$ we have $kp \in P^s \cap Q^s = R^s$.

Therefore there exists a degenerate string with weak, medium and strong period sets R^w, R^m and R^s respectively. Since we are taking intersections of the individual period sets, the corresponding total autocorrelation is the minimum of u and v.

Join Let u and v be two total autocorrelations. The join is the minimum of the autocorrelations greater than both u and v. Note that the minimum of all greater or equal autocorrelations is greater or equal than both u and v and not greater than any autocorrelation $w \geq u, v$. Observe that this join is well-defined since there is always at least one autocorrelation greater than or equal to both (namely 3^n) and that the join is an autocorrelation as well (because it is the meet of autocorrelations).

We conclude that $(\Psi_n, \leq)$ is a lattice. □

6 Population of autocorrelations

In this section we will give formulae to compute the population of autocorrelations of degenerate strings, in the case $\Delta = \mathcal{P}(\Sigma) \setminus \{\emptyset\}$. The population of an autocorrelation (resp. period set) is defined as the number of degenerate strings with this autocorrelation (resp. period set). We will follow the work of Blanchet-Sadri et al. [3], who compute the population number of partial words using graph theory. However, instead of looking at graph colourings, we look at independent sets to account for arbitrary sets of letters at each position of the degenerate string. We will first give the formulae for weak periods, and then explain how these can be adapted to find the population of medium periods. Finally, we discuss the case of strong periodicity and give an analogous hypergraph formulation to illustrate our difficulty in generalizing the result.

6.1 Weak and medium period sets

We are given a set $P \subseteq \{0, 1, \ldots, n-1\}$ and would like to compute how many degenerate strings there are over Σ with weak (resp. medium) period set P.

We define a graph on the set of positions $\{0, 1, \ldots, n-1\}$, with an edge connecting two vertices if and only if they differ by a period $p \in P$. We will first compute how many strings there are that have these periods (and possibly more periods). This is the number of ways we can assign subsets of Σ to the vertices such that

(a) no vertex is assigned the empty set, and
(b) the sets assigned to any two adjacent vertices have non-empty intersection.

We will first count the number of sets satisfying property (b) using the inclusion-exclusion principle. For each subgraph H we compute how many assignments there are where *all* pairs of adjacent vertices have *no* letter in common. For each letter there are $i(H)$ ways to assign it, where $i(H)$ is the number of independent sets in H. This gives $i(H)^{|\Sigma|}$ ways in total for the subgraph. There are $2^{(|V(G)|-|V(H)|)\cdot|\Sigma|}$ assignments for the rest of G. The number of assignments where every pair of adjacent positions has a letter in common — those satisfying property (b) — is thus

$$\sum_{H \subseteq G} (-1)^{|E(H)|} 2^{(|V(G)|-|V(H)|)\cdot|\Sigma|} i(H)^{|\Sigma|}.$$

Now, if every pair of adjacent vertices has a letter in common, all non-isolate vertices are assigned at least one letter. The isolate vertices are completely independent however, so we need to adjust for the chance of them being assigned the empty set, as this would result in a hollow string. Let $I(G)$ be the number of isolated vertices of G. By construction of the graph $I(G) = \max(2 \cdot p_{\min} - n, 0)$, where $p_{\min}$ is the smallest non-zero period in P (and n if it has no non-zero period). Removing the hollow strings we get

$$\left(\frac{2^{|\Sigma|}-1}{2^{|\Sigma|}}\right)^{I(G)} \cdot \sum_{H \subseteq G} (-1)^{|E(H)|} 2^{(|V(G)|-|V(H)|)\cdot|\Sigma|} i(H)^{|\Sigma|}$$

degenerate strings satisfying properties (a) and (b). This number contains all strings that have the given period set P as a subset of their period set. Thus to get the precise period, we must subtract bigger period sets using the inclusion-exclusion principle.

$$\sum_{P \subseteq Q \in \Omega_n} (-1)^{|Q|-|P|} \left(\frac{2^{|\Sigma|}-1}{2^{|\Sigma|}}\right)^{I(G_Q)} \cdot \sum_{H \subseteq G_Q} (-1)^{|E(H)|} 2^{(|V(G_Q)|-|V(H)|)\cdot|\Sigma|} i(H)^{|\Sigma|}$$

Here Ω_n is the set of all period sets and differs between the weak and medium cases.

6.2 Strong period sets

For strong periodicity, we can use the same technique. However, now we want that all positions with the same index modulo p have a letter in common. To model this, we can use the hypergraph $G = (V, E)$, where $V = \{0, \ldots, n-1\}$ and $E = \{\{j \in \{0, \ldots, n-1\} \mid j \equiv i \mod p\} \mid p \in P,\ i \in \{1, \ldots, p\}\}$.

We want to assign symbols to vertices such that for each hyperedge there exists a letter, which is in all symbols. Here things get more complex: if we want to use the inclusion-exclusion principle, we need to count the number of ways the constraints

on a certain set of hyperedges are violated. That is, for each such hyperedge and each letter, we do not want to assign the letter to all its vertices. Equivalently, the non-assigned vertices cover the hyperedges. Thus, if we define we define $i'(H)$ to be the number of vertex covers (also known as transversals) of H, then we can apply the same formula.

$$\sum_{P \subseteq Q \in \Omega_n^s} (-1)^{|Q|-|P|} \left(\frac{2^{|\Sigma|}-1}{2^{|\Sigma|}} \right)^{I(G_Q)} \sum_{H \subseteq G_Q} (-1)^{|E(H)|} 2^{(|V(G_Q)|-|V(H)|)\cdot|\Sigma|} i'(H)^{|\Sigma|}$$

Remark: Since $\Omega_n^m = \Omega_n^s$ for any $n \geq 1$, and some degenerate strings have a different medium and strong period sets, the population of a given period set should differ in medium and strong case. This is not the case for partial strings.

6.3 Total autocorrelations

To find the population of a total autocorrelation, we can use the same technique. Here, we choose the graph to be (V, E), where $V = \{0, \ldots, n-1\}$ and $E = E^w \cup E^m \cup E^s$, where E^w, E^m and E^s are the (hyper)edge sets corresponding to the weak, medium and strong period sets as defined above. The formula follows analogously.

$$\sum_{P \subseteq Q \in \Psi_n} (-1)^{|Q|-|P|} \left(\frac{2^{|\Sigma|}-1}{2^{|\Sigma|}} \right)^{I(G_Q)} \sum_{H \subseteq G_Q} (-1)^{|E(H)|} 2^{(|V(G_Q)|-|V(H)|)\cdot|\Sigma|} i'(H)^{|\Sigma|}$$

Remark: Note that these formulas are costly to compute. However, if we want to compute multiple populations, we can obtain slight speed ups using dynamic programming and memoization. For example, we can compute the number of independent sets $i(H)$, in terms of the number of independent sets of its subgraphs.

7 Future Work

In future work, we would like to explore how the concept of periodicity translates from degenerate strings to different families of languages. In particular, we would like to generalize our definitions to apply to *any language*, i.e., any set of strings. We want to investigate which combinatorial results carry over to this more general setting, and if not, which additional conditions must be met.

Moreover, we are interested in studying the algorithmic aspects of the periodicity of languages. One question would be the complexity of determining period sets of degenerate strings; while naïve algorithms are already close to optimal, as shown by the lower bound proven in [14], there might be room for improvement in certain cases, such as the restriction of the alphabet size. A second area of interest is the application of periodicity to matching algorithms on degenerate strings. Similarly to how periodicity is applied to the Knuth-Morris-Pratt algorithm for matching in classical strings, it may be possible to carry over the same concepts to degenerate string matching using our defined terminology for periodicity.

Acknowledgements This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and No 956229, from the Netherlands Organisation for Scientific Research (NWO) through

Gravitation-grant NETWORKS-024.002.003 and from the Constance van Eeden PhD Fellowship. Moreover, we would like to thank Solon P. Pissis for his helpful advice and suggestions.

References

1. R. Angelo: *A Cameron and Erdős conjecture on counting primitive sets.* INTEGERS, 18 2018, p. 2.
2. F. Blanchet-Sadri: *Algorithmic Combinatorics on Partial Words*, Discrete mathematics and its applications, CRC Press, 2008.
3. F. Blanchet-Sadri, J. Fowler, J. D. Gafni, and K. H. Wilson: *Combinatorics on partial word correlations.* Journal of Combinatorial Theory, Series A, 117(1) 2010, pp. 607–624.
4. F. Blanchet-Sadri, J. D. Gafni, and K. H. Wilson: *Correlations of partial words*, in STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings, W. Thomas and P. Weil, eds., vol. 4393 of Lecture Notes in Computer Science, Springer, 2007, pp. 97–108.
5. J. C. Bryne, E. Valen, M.-H. E. Tang, T. Marstrand, O. Winther, I. da Piedade, A. Krogh, B. Lenhard, and A. Sandelin: *JASPAR, the open access database of transcription factor-binding profiles: new content and tools in the 2008 update.* Nucleic Acids Research, 36(suppl 1) 2008, pp. D102–D106.
6. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen: *Covering problems for partial words and for indeterminate strings.* Theoretical Computer Science, 698 2017, pp. 25–39.
7. P. Erdős: *Note on sequences of integers no one of which is divisible by any other.* Journal of the London Mathematical Society, 10(1) 1935, pp. 126–128.
8. L. Guibas and A. Odlyzko: *Periods in strings.* Journal of Combinatorial Theory, Series A, 30 1981, pp. 19–43.
9. D. Gusfield: *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, 1997.
10. V. Halava, T. Harju, and L. Ilie: *Periods and binary words.* Journal of Combinatorial Theory, Series A, 89(2) 2000, pp. 298–303.
11. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings.* In Proceedings of 14th Australasian Workshop on Combinatorial Algorithms, 2003, pp. 36–45.
12. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings.* Journal of Discrete Algorithms, 6(1) 2008, pp. 37–50.
13. C. S. Iliopoulos, R. Kundu, and S. P. Pissis: *Efficient pattern matching in elastic-degenerate strings.* CoRR, abs/1610.08111 2016.
14. C. S. Iliopoulos and J. Radoszewski: *Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties*, in 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel, R. Grossi and M. Lewenstein, eds., vol. 54 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 8:1–8:12.
15. IUPAC-IUB Commission on Biochemical Nomenclature: *Abbreviations and symbols for the description of the conformation of polypeptide chains. Tentative rules (1969).* Biochemistry, 9(18) 1970, pp. 3471–3479.
16. I. V. Kulakovskiy, I. E. Vorontsov, I. S. Yevshin, R. N. Sharipov, A. D. Fedorova, E. I. Rumynskiy, Y. A. Medvedeva, A. Magana-Mora, V. B. Bajic, D. A. Papatsenko, and et al.: *HOCOMOCO: towards a complete collection of transcription factor binding models for human and mouse via large-scale ChIP-Seq analysis.* Nucleic Acids Research, 46(D1) Nov 2018, p. D252–D259.
17. H. Liu, P. P. Pach, and R. Palincza: *The number of maximum primitive sets of integers.* Combinatorics, Probability and Computing, 30(5) 2021, p. 781–795.
18. M. Lothaire, ed., *Combinatorics on Words*, Cambridge University Press, second ed., 1997.
19. M. Lothaire: *Algebraic Combinatorics on Words*, Cambridge University Press, Cambridge, 2002.
20. N. McNew: *Counting primitive subsets and other statistics of the divisor graph of 1,2,...,n.* European Journal of Combinatorics, 92 2021, p. 103237.

21. F. Pfeiffer, C. Gröber, M. Blank, K. Händler, M. Beyer, J. L. Schultze, and G. Mayer: *Systematic evaluation of error rates and causes in short samples in next-generation sequencing.* Scientific Reports, 8(1) Jul 2018.
22. N. Pisanti, H. Soldano, and M. Carpentier: *Incremental inference of relational motifs with a degenerate alphabet*, in Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005, Proceedings, A. Apostolico, M. Crochemore, and K. Park, eds., vol. 3537 of Lecture Notes in Computer Science, Springer, 2005, pp. 229–240.
23. S. Rahmann and E. Rivals: *On the distribution of the number of missing words in random texts.* Combinatorics, Probability and Computing, 12(01) Jan 2003.
24. E. Rivals and S. Rahmann: *Combinatorics of periods in strings.* Journal of Combinatorial Theory, Series A, 104(1) Oct 2003, pp. 95–113.
25. E. Rivals, M. Sweering, and P. Wang: *Convergence of the Number of Period Sets in Strings*, in 50th International Colloquium on Automata, Languages, and Programming (ICALP 2023), K. Etessami, U. Feige, and G. Puppis, eds., vol. 261 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2023, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 100:1–100:14.
26. W. F. Smyth and S. Wang: *New perspectives on the prefix array*, in String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10-12, 2008. Proceedings, A. Amir, A. Turpin, and A. Moffat, eds., vol. 5280 of Lecture Notes in Computer Science, Springer, 2008, pp. 133–143.
27. H. Soldano, A. Viari, and M. Champesme: *Searching for flexible repeated patterns using a non-transitive similarity relation.* Pattern Recognition Letters, 16(3) 1995, pp. 233–246.
28. The Computational Pan-Genomics Consortium: *Computational pan-genomics: status, promises and challenges.* Briefings in Bioinformatics, 19(1) 2018, pp. 118–135.

A More about lattices

In this appendix, we prove that Ω_n^w, Ω_n^m and Ω_n^s lattices under set intersection and set union, and hence satisfy the Jordan-Dedekind condition. Before we start, we first review some important concepts. We start by recalling the definition of meet and join in terms of posets (partially ordered sets).

Definition (Meet and join). *Given a poset $(A, \leq)$ and $x, y \in A$. We say m is the meet (greatest lower bound or infimum) of x and y denoted by $x \wedge y$, if m satisfies the following conditions.*
1. $m \in A$
2. $m \leq x$ *and* $m \leq y$
3. *For all $w \in A$, if $w \leq x$ and $w \leq y$, then $w \leq m$.*

We say j is the join (least upper bound or supremum) of x and y denoted by $x \vee y$, if j satisfies the following conditions.
1. $j \in A$
2. $x \leq j$ *and* $y \leq j$
3. *For all $w \in A$, if $x \leq w$ and $y \leq w$, then $j \leq w$.*

Definition (Lattice). *Poset $(A, \leq)$ is a lattice if and only if all $x, y \in A$ have both a meet and join.*

Let Ω_n^w, Ω_n^m and Ω_n^s denote the families of weak, medium and strong period sets. In this section, we show that Ω_n^w, Ω_n^m and Ω_n^s are all lattices partially ordered by inclusion.

Theorem 16. *$(\Omega_n^w, \subseteq)$, $(\Omega_n^m, \subseteq)$ and $(\Omega_n^s, \subseteq)$ are lattices with respect to the inclusion order.*

Proof. To show that these posets are lattices, we need to show that their meet and join operations are well-defined. Specifically, since we order their elements with respect to inclusion, we need to show that Ω_n^w, Ω_n^m and Ω_n^s are closed under intersection and union (conditions 2 and 3 are trivially met).

Weak Let $U, V \in \Omega_n^w$ be two weak period sets. Then $0 \in U \subseteq \{0, \ldots, n-1\}$ and $0 \in V \subseteq \{0, \ldots, n-1\}$. It follows that $0 \in U \cup V \subseteq \{0, \ldots, n-1\}$ and $0 \in U \cap V \subseteq \{0, \ldots, n-1\}$. Thus $U \cup V \in \Omega_n^w$ and $U \cap V \in \Omega_n^w$. We conclude that $(\Omega_n^w, \subseteq)$ is a lattice.

Medium Let $U, V \in \Omega_n^m$ be two medium period sets. Equivalently, U and V are two subsets of $\{0, \ldots, n-1\}$ containing 0 and closed under multiplication. It follows that $U \cap V$ and $U \cup V$ also contain 0 and are closed under multiplication. Thus $U \cup V \in \Omega_n^m$ and $U \cap V \in \Omega_n^m$. We conclude that $(\Omega_n^m, \subseteq)$ is a lattice.

Strong Since $\Omega^s = \Omega^m$, the poset of strong period sets $(\Omega_n^s, \subseteq)$ also form a lattice under ordering by inclusion.

We conclude that $(\Omega_n^w, \subseteq)$, $(\Omega_n^m, \subseteq)$ and $(\Omega_n^s, \subseteq)$ are all lattices with respect to the inclusion order. □

Approximate String Searching with AVX2 and AVX-512

Tamanna Chhabra[1], Sukhpal Singh Ghuman[1], and Jorma Tarhio[2]

[1] Faculty of Applied Science and Technology
Sheridan College, Ontario, Canada
firstname.lastname@sheridancollege.ca
[2] Department of Computer Science
Aalto University, Finland
firstname.lastname@aalto.fi

Abstract We present new algorithms for the k mismatches version of approximate string matching. Our algorithms utilize the SIMD (Single Instruction Multiple Data) instruction set extensions, particularly AVX2 and AVX-512 instructions. Our approach is an extension of an earlier algorithm for exact string matching with SSE2 and AVX2. In addition, we modify this exact string matching algorithm to work with AVX-512. We demonstrate the competitiveness of our solutions by practical experiments. Our experimental results show that our algorithms outperform earlier algorithms for both exact and approximate string matching on various benchmark data sets.

Keywords: Approximate string matching, Hamming distance, exact string matching, SIMD computing, experimental comparison

1 Introduction

String matching [5] is a widely studied problem in Computer Science. The problem of string matching consists of two strings, a text and a pattern, and the task is to find all occurrences of the pattern in the text.

There have been numerous developments in this field in the recent past. Many variations of this problem have appeared, such as exact string matching [5], approximate string matching [10], order preserving matching [2], jumbled pattern matching [7] and many more.

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, the problem of exact string matching is defined as follows: to find all the positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_0 p_1 \cdots p_{m-1}$. In this paper we consider the k mismatches variation of the problem where P', a substring of T, is an occurrence of P, if $|P'| = |P|$ holds and P' has at most k mismatches with P, $0 \leq k < m$. The mismatch distance of two strings of equal length is also called the Hamming distance. For example, if $x = ababb$ and $y = abbab$, then the Hamming distance between x and y is 2.

In our study, we propose algorithms that make use of SIMD (Single Instruction Multiple Data) computing for approximate string matching. By harnessing the AVX2 and AVX-512 features found in modern processors, our algorithms can process multiple characters simultaneously. Especially AVX2 is widely available in new Intel and AMD processors. To build upon existing work, we start with a simple algorithm [12] that already utilizes SIMD for exact string matching. We extend and modify this algorithm to handle mismatches, thereby allowing approximate string matching.

Our main focus is on demonstrating the practical efficiency of the new algorithms. Our algorithms count the number of occurrences with up to five mismatches. To

Tamanna Chhabra, Sukhpal Singh Ghuman, Jorma Tarhio: Approximate String Searching with AVX2 and AVX-512, pp. 57–67.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

show their competitiveness, we conduct practical experiments that validate their performance. As a result, we not only achieve faster approximate string matching, but also surpass the speed of earlier algorithms designed for exact string matching. The improvement in approximate string matching is significant: In English data, when permitting one mismatch, our algorithm is approximately six times faster than the reference method.

The rest of the paper is organized as follows: Section 2 presents the background. Section 3 introduces our algorithms for approximate string matching, Section 4 describes adaptation of the approach to AVX-512, and Section 5 depicts the results of our practical experiments, and Section 6 concludes the article.

2 Background

For exact string matching Tarhio et al. [12] presented a naive algorithm (shown as Algorithm 1) which uses the SIMD instruction architecture. The algorithm compares α characters in parallel, where α is 16 or 32. In the following, the names N16 and N32 are used for these variations. N16 uses the SSE2 instruction set and N32 the AVX2 instruction set.

Algorithm 1: SIMD-naive-search

```
1 construct vector(c) for each c ∈ Σ
2 count ← 0; i ← 0
3 while i ≤ n − m do
4     found ← 2^α − 1
5     for j ← 0 to m − 1 do
6         found ← found and SIMDcompare(t_{i+j}, vector(p_j), α)
7         if found = 0 then goto out
8     count ← count + popcount(found)
9     out: i ← i + α
```

The key idea of Algorithm 1 is to test α consecutive potential occurrences of the pattern in parallel. For that purpose, a comparison vector containing α copies of the same character is constructed in line 1 for each character of the alphabet. In the case of AVX2, α is 32, and the algorithm first compares the vector of p_0 with $t_0 \dots t_{31}$, and then it compares the vector of p_1 with $t_1 \dots t_{32}$ and so on. The bitvector *found* of 32 bits keeps track of active match candidates. The intrinsic function `_mm_popcnt_u32` [9] is used for counting matches in line 8.

The SIMDcompare function for the AVX2 architecture uses three intrinsic functions [9] described below:

- `_mm256_loadu_si256`: The function loads 256 bits of integer data from memory into the destination. The memory address does not need to be aligned on the particular boundary.
- `_mm256_cmpeq_epi8`: The function compares a 32 8-bit integer elements in two 256-bit vectors and sets the corresponding bit in the output vector to 1 if the two elements are equal, and to 0 otherwise. The result is a 256-bit vector where each bit represents the result of a single comparison operation.
- `_mm256_movemask_epi8`: The function creates a 256-bit vector of 32 8-bit integer elements and returns a 32-bit integer value where the ith bit is set to 1 if the ith element of the vector has its most significant bit set, and to 0 otherwise. This

function creates a mask from the most significant bit of the comparison result as a 32-bit integer.

With these intrinsic functions, SIMDcompare for AVX2 is implemented as follows:

```
SIMDcompare(x, y, 32)
   x_ptr = _mm256_loadu_si256(x)
   y_ptr = _mm256_loadu_si256(y)
   return _mm256_movemask_epi8(_mm256_cmpeq_epi8(x_ptr, y_ptr))
```

For N16, the corresponding intrinsic functions for loading, comparing, and creating a mask for comparison are used in the SSE2 instruction set architecture.

Tarhio et al. [12] used three different orders for comparing the characters of the pattern: plain order, fixed order, and reverse English frequency order. In the case of the 32 byte version, we use the following names for these variations: N32, N32F, and N32E. The plain order advances from left to right in the pattern. The fixed order applies the following heuristic order: $p_0, p_{m-1}, p_3, p_6, \ldots, p_2, p_5, \ldots, p_1, p_4, \ldots$ excluding space characters which are compared last. The reverse frequency order could be applied to any type of data, but only English has been used in experiments. Algorithm 1 uses the plain order. In the case of other orders, the call of the SIMDcompare function in line 6 is in the form

$$\text{SIMDcompare}(t_{i+\pi(j)}, \text{vector}(p_{\pi(j)}),\ \alpha)$$

where π is a permutation of pattern positions. The algorithm uses $\alpha \cdot |\Sigma|$ bytes extra space for the vectors.

In addition to the SIMD instructions, loop peeling has a key role in the efficiency of Algorithm 1. In loop peeling, a number of iterations is moved in front of the loop. As a result, the code becomes faster because of fewer loop tests. In loop unrolling, the whole loop is peeled. In the following, we call the number of the moved iterations the peeling factor r. Tarhio et al. [12] used $r = 2$ or 3 for English and $r = 5$ for DNA.

3 Algorithms for approximate matching

Our aim is to develop algorithms for approximate string matching. Algorithm 2 (as shown below) is used to count all the occurrences of a given pattern string P in a text string T, with at most k mismatches. To perform the comparisons efficiently, the algorithm uses SIMD (Single Instruction Multiple Data) instructions, which can compare multiple characters in parallel. The algorithm is a variation of N32 extended with mismatch counting. It works by handling α consecutive starting positions of an occurrence candidate of P in parallel. In the case of AVX2, α is 32. Bitvectors $found[0], \ldots, found[k]$ of α bits are used to keep track of mismatches. Initially, every bit of each $found[i]$ is set. During computation, if the jth bit of $found[i]$ becomes zero, then more than i mismatches has been found while checking the jth candidate. If all the bits of $found[k]$ become zero, then none of the α candidates can be an occurrence of the pattern. The comparison vectors for each position of the pattern are computed before search in line 1. Each comparison vector contains α copies of the corresponding character. The popcount function counts the number of set bits in a vector.

Because $n - m + 1$ is not divisible by α in a general case, the last execution of line 11 may add extra matches to *count* in some rare cases. For example, this may

Algorithm 2: SIMD-approximate-search

```
1  for j ← 0 to m − 1 do construct vector(j) for p_j
2  count ← 0; i ← 0
3  while i ≤ n − m do
4      for j ← 0 to k do found[j] ← 2^α − 1
5      for j ← 0 to m − 1 do
6          c ← SIMDcompare(t_{i+j}, vector(j), α)
7          for s ← k downto 1 do
8              found[s] ← found[s] and (found[s − 1] or c)
9          found[0] ← found[0] and c
10         if found[k] = 0 then goto out
11     count ← count + popcount(found[k])
12     out: i ← i + α
13 count ← count − popcount(found[k] >> (n − m − i + α + 1))
```

happen when searching for aaaaa with $k \geq 1$ mismatches in a text ending with aaaa. Line 13 eliminates such extra matches from *count*. Because *found*$[k]$ is reversed at the implementation level, the vector is shifted to the right in order to hide real matches. Here we assume that it is allowed to access some text positions beyond t_{n-1}. If that is not the case, texts shorter than $\alpha + m - 1$ and the end of a text should be processed with another algorithm.

Algorithm 1 and Algorithm 2 use different approaches for constructing comparison vectors. In Algorithm 1, vectors are constructed for each character of the alphabet, while in Algorithm 2, vectors are constructed for each position of the pattern. Thus Algorithm 2 needs $\alpha \cdot m$ bytes extra space for the vectors. This approach saves space when m is less than $|\Sigma|$.

Algorithm 2 uses the plain order. If other orders are used (see Section 2), the call of the SIMDcompare function in line 6 is in the form

$$\text{SIMDcompare}(t_{i+\pi(j)}, \text{vector}(\pi(j)),\ \alpha)$$

where π is a permutation of pattern positions. Algorithm 2 solves the counting version of approximate string matching with k mismatches. It can be transformed into the reporting version by printing positions in line 11.

Proof of the counting method

Without losing generality, we can assume that positions of candidates are processed in order from left to right. Let $f_{i,k}$ be the bit of *found*$[k]$ corresponding a candidate starting from t_j after $p_0 \cdots p_i$ has been processed. According to the construction, $f_{i,k} \geq f_{i,k-1}$ holds.

Proposition: $f_{i,k} = 1$ holds if $t_j \cdots t_{j+i}$ contains at most k character mismatches with $p_0 \cdots p_i$, and otherwise $f_{i,k} = 0$ holds.

Proof by induction: If t_j is p_0, then we have $f_{0,0} = f_{0,1} = \cdots = f_{0,k} = 1$. If t_j is not p_0, then we have $f_{0,0} = 0$ and $f_{0,1} = \cdots = f_{0,k} = 1$.

Let us assume that the proposition holds for $f_{i-1,k-1}$. If t_{j+i} is p_i, then we have $f_{i,k} = f_{i-1,k-1}$ and the proposition holds. If t_{j+i} is not p_i, we have two cases. If $f_{i-1,k-1} = 0$ holds, then we have $f_{i,k} = f_{i-1,k-1}$ and the proposition holds. If $f_{i-1,k-1} = 1$ holds, then $f_{i-1,k} = 1$ holds. So $f_{i,k} = 1$ is satisfied. Because $t_j \cdots t_{j+i-1}$ contains at most $k - 1$ mismatches according to the induction assumption, $t_j \cdots t_{j+i}$ contains at most k mismatches.

Let us consider an example. The bolded entry in Table 1 shows the value of $f_{i,3}$ after processing aabbab in the text for P = aaaaaa. Here vectors are shown in the order of the text.

Table 1. An example of computation of $f_{i,3}$. P = aaaaaa, $k = 3$.

	a	a	b	b	a	b
$f_{i,3}$	1	1	1	1	1	**1**
$f_{i,2}$	1	1	1	1	1	0
$f_{i,1}$	1	1	1	0	0	0
$f_{i,0}$	1	1	0	0	0	0

Tuning up

The pseudocode of Algorithm 2 presents the principles that can be applied to any scenario for $k < m$. Recognizing that the approach is primarily advantageous for small values of k, we developed algorithms for fixed $k = 1, 2, \ldots, 5$. By combining these algorithms, we were able to craft a more efficient implementation. First we split the for loop in line 5 of Algorithm 2 into two parts and reduce some unnecessary assignments. The outcome is shown in Algorithm 3.

Algorithm 3 Loop (line 5) of Alg. 2 split.

```
1  for j ← 0 to k do
2      c ← SIMDcompare(t_{i+j}, vector(p_j), α)
3      for s ← j downto 1 do
4          found[s] ← found[s] and (found[s − 1] or c)
5      found[0] ← found[0] and c
6  for j ← k + 1 to m − 1 do
7      c ← SIMDcompare(t_{i+j}, vector(p_j), α)
8      for s ← k downto 1 do
9          found[s] ← found[s] and (found[s − 1] or c)
10     found[0] ← found[0] and c
11     if found[k] = 0 then goto out
```

When k is fixed, we can unroll the three loops in lines 1, 3, and 8 of Algorithm 3 and the initialization loop in line 4 of Algorithm 2. After these changes, the code still contains computations that are not essential, but the compiler can fairly efficiently eliminate them.

We call the tuned version N32A. Exactly in the same way as in the case of exact string matching explained in Section 2, we get the variations N32FA and N32EA for handling pattern positions in the fixed heuristic order and the reverse English frequency order.

4 Adaptation to AVX-512

We decided to adapt Algorithm 2 to leverage AVX-512 extensions, which allow us to compare 64 bytes in parallel. The set of AVX-512 intrinsic functions does not contain a 512-bit counterpart for the `_mm256_cmpeq_epi8` intrinsic function which was applied in SIMDcompare for AVX2. Therefore we selected another intrinsic function computing

the mask as well, eliminating the need for an extra intrinsic function. Here is the redesigned SIMDcompare function for AVX-512:

```
SIMDcompare(x, y, 64)
   x_ptr =  _mm512_loadu_si512(x)
   y_ptr =  _mm512_loadu_si512(y)
   return _mm512_cmpeq_epi8_mask(x_ptr, y_ptr)
```

In addition, we use `_mm_popcnt_u64` for counting matches. We describe how these 512-bit intrinsic functions [9] work. The function `_mm512_loadu_si512` works correspondingly to `_mm512_loadu_si512`, which was explained in Section 2.

The intrinsic function `_mm512_cmpeq_epi8_mask`[1] performs an element-wise comparison of two 512-bit registers containing 64 8-bit integer elements each. It returns a 64-bit mask, where each bit represents the result of the comparison of the corresponding 8-bit integer element in the input registers. If the two elements are equal, the corresponding bit in the mask is set to 1, otherwise it is set to 0.

SIMDcompare for AVX-512 is lighter than SIMDcompare for SSE2 or AVX2, because SIMDcompare for AVX-512 has one intrinsic function less than the others.

The same adaptation into the AVX-512 platform applies naturally also for Algorithm 1 for exact matching. Therefore, we present experimental results of exact string matching in the next section in addition to the results of approximate string matching.

5 Experimental Results

We present experimental results in order to compare the behavior of our algorithms against the best known solutions in the literature for approximate and exact string searching.

5.1 Setting

All the algorithms were implemented[2] using the C programming language and compiled with Apple Clang 14.0.0 and run in the testing framework of Hume and Sunday [8]. The processor used was Intel Core i5-1030NG7 with 6 MB cache and 8 GB RAM. The operating system used was MacOS Ventura 13.0.1.

We used three texts: English (the KJV Bible, 12 MB), DNA (the genome of E. Coli, 10 MB), and random binary ($|\Sigma| = 2$, 12 MB) for testing. We chose the length of the text to be at least 1.5 times the cache size (by concatenating the multiples of the text) in order to avoid cache interference with running times [11]. Sets of patterns of lengths 5, 8, 10, 16, and 32 were randomly taken from the texts. Each set contains 200 patterns. The tests were made with 99 repeated runs. Speedup is reported as a ratio of the running times of the reference algorithm and a new algorithm.

5.2 Approximate matching

We compared our algorithms (N32A and N64A for the plain order, N32FA and N64FA for the fixed order, N32EA and N64EA for the reverse English frequency order)

[1] Note that `_mm256_cmpeq_epi8_mask` is not available in AVX2 but only in AVX-512. Therefore it was not used in Algorithm 1.

[2] The codes are available at https://users.aalto.fi/tarhio/hamming/.

against ANS2B, BYPSB, and BYPSC [6] for $5 \leq m \leq 32$. Fiori et al. [6] tested twelve algorithms for the k mismatches problem, and ANS2B, BYPSB, and BYPSC were clearly the best among them. Because all these algorithms apply SIMD, we also present test results of TWSA [3], which is one of the best non-SIMD algorithms for the k mismatches problem.

We carried out the experiments for $k = 1$, 3, and 5 as shown in Tables 2, 3, and 4. The best time for each pattern set has been boxed and the used peeling factor for each run has been super-scripted in the tables. From the results, it is clear that the new algorithms outperform earlier algorithms with a wide margin. For the English dataset, the speedup of N64FA over ANS2B is about six for $k = 1$, indicating a significant improvement in performance. The speedup AVX-512 offers over AVX2 is typically 1.5 or more, i.e. speedup of the variations of N64A over the corresponding versions of N32A.

When k increases, our algorithms become slower. As an example, Figure 1 shows the search times of ANS2B for $k = 1$ and N64FA for $k = 1$, 3, and 5 in the English dataset. The 64-byte algorithms stay competitive at least until $k = 5$.

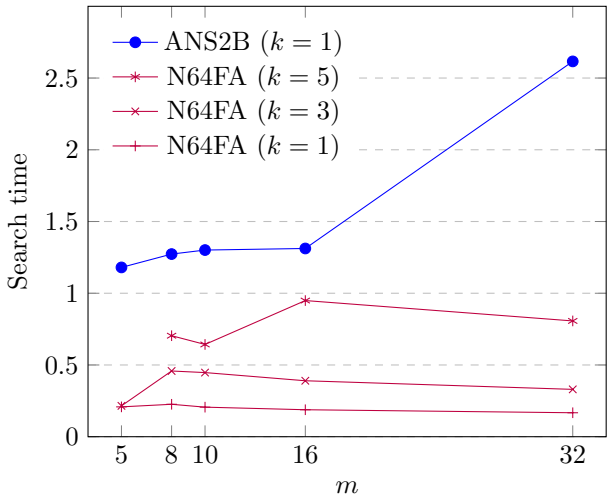


Figure 1. Search times of ANS2B for $k = 1$ and N64FA for $k = 1$, 3, and 5 in the English dataset.

In most of the cases N64EA and N64FA are almost equally fast for English data as well as N64A and N64FA for binary and DNA data.

There is no upper limit for the pattern size our algorithms can handle, but the speed does not change much when the pattern gets longer.

One noteworthy finding is that the advantage of N64A over N32A increases when patterns do not appear in the text. This observation is useful for scenarios checking for pattern absence.

5.3 Effect of peeling factor

We analyzed the performance of the N64FA algorithm with various peeling factors for $k = 1$. The results are presented in Table 5. The optimal choice of the peeling factor r depends on the nature of the dataset. For English data, lower r values produce better speed, while for DNA and binary data, higher r values yield improved performance.

Table 2. Approximate search times, $k = 1$.

		$m = 5$	8	10	16	32
English	ANS2B	1.18	1.27	1.30	1.31	2.62
	BYPSC	—	3.14	3.19	0.807	0.43
	TWSA	4.53	3.25	2.77	1.83	0.886
	N32FA	$0.336^{\{5\}}$	$0.295^{\{4\}}$	$0.283^{\{4\}}$	$0.265^{\{4\}}$	$0.222^{\{4\}}$
	N32EA	$0.357^{\{5\}}$	$0.325^{\{4\}}$	$0.305^{\{4\}}$	$0.245^{\{3\}}$	$0.209^{\{3\}}$
	N64FA	$\boxed{0.208}^{\{5\}}$	$0.226^{\{4\}}$	$0.206^{\{5\}}$	$\boxed{0.188}^{\{4\}}$	$0.167^{\{4\}}$
	N64EA	$0.219^{\{5\}}$	$\boxed{0.217}^{\{4\}}$	$\boxed{0.200}^{\{4\}}$	$\boxed{0.188}^{\{4\}}$	$\boxed{0.154}^{\{4\}}$
DNA	ANS2B	1.02	1.09	1.12	1.12	2.15
	BYPSB	—	3.58	3.22	0.81	0.35
	TWSA	5.71	3.72	3.20	1.90	0.909
	N32A	$0.273^{\{5\}}$	$0.448^{\{8\}}$	$0.443^{\{8\}}$	$0.452^{\{8\}}$	$0.428^{\{8\}}$
	N32FA	$0.277^{\{5\}}$	$0.370^{\{7\}}$	$0.373^{\{7\}}$	$0.398^{\{7\}}$	$0.368^{\{7\}}$
	N64A	$\boxed{0.189}^{\{5\}}$	$0.271^{\{8\}}$	$0.271^{\{8\}}$	$\boxed{0.269}^{\{8\}}$	$0.266^{\{8\}}$
	N64FA	$0.205^{\{5\}}$	$\boxed{0.249}^{\{8\}}$	$\boxed{0.262}^{\{8\}}$	$0.277^{\{8\}}$	$\boxed{0.252}^{\{8\}}$
Binary	ANS2B	1.26	1.34	1.35	1.38	2.75
	BYPSB	—	11.91	8.20	5.01	0.699
	TWSA	—	—	—	3.85	1.91
	N32A	$0.323^{\{5\}}$	$0.520^{\{8\}}$	$0.654^{\{10\}}$	$0.965^{\{8\}}$	$0.930^{\{11\}}$
	N32FA	$0.334^{\{5\}}$	$0.539^{\{8\}}$	$0.704^{\{11\}}$	$0.988^{\{13\}}$	$0.915^{\{13\}}$
	N64A	$\boxed{0.213}^{\{5\}}$	$0.304^{\{8\}}$	$\boxed{0.385}^{\{10\}}$	$\boxed{0.536}^{\{15\}}$	$0.549^{\{15\}}$
	N64FA	$0.221^{\{5\}}$	$\boxed{0.299}^{\{8\}}$	$0.392^{\{10\}}$	$0.539^{\{14\}}$	$\boxed{0.530}^{\{14\}}$

Table 3. Approximate search times, $k = 3$.

		$m = 5$	8	10	16	32
English	ANS2B	1.25	1.325	1.23	1.27	2.70
	BYPSC	—	—	—	4.24	1.17
	TWSA	6.23	5.18	4.65	2.83	—
	N32FA	$0.322^{\{5\}}$	$0.706^{\{6\}}$	$0.693^{\{6\}}$	$0.589^{\{6\}}$	$0.832^{\{6\}}$
	N32EA	$0.335^{\{5\}}$	$0.771^{\{6\}}$	$0.652^{\{6\}}$	$0.526^{\{6\}}$	$0.427^{\{6\}}$
	N64FA	$0.215^{\{5\}}$	$0.458^{\{8\}}$	$0.447^{\{7\}}$	$0.390^{\{6\}}$	$0.330^{\{6\}}$
	N64EA	$\boxed{0.213}^{\{5\}}$	$\boxed{0.415}^{\{8\}}$	$\boxed{0.445}^{\{7\}}$	$\boxed{0.349}^{\{6\}}$	$\boxed{0.260}^{\{6\}}$
DNA	ANS2B	1.09	1.08	1.22	1.12	2.18
	BYPSB	—	—	—	4.37	1.09
	TWSA	3.87	5.32	4.62	2.92	—
	N32A	$0.245^{\{5\}}$	$0.656^{\{8\}}$	$0.945^{\{10\}}$	$1.06^{\{10\}}$	$1.07^{\{10\}}$
	N32FA	$0.267^{\{5\}}$	$0.663^{\{8\}}$	$1.07^{\{10\}}$	$1.12^{\{10\}}$	$1.04^{\{10\}}$
	N64A	$\boxed{0.170}^{\{5\}}$	$\boxed{0.343}^{\{8\}}$	$\boxed{0.531}^{\{10\}}$	$\boxed{0.615}^{\{10\}}$	$0.615^{\{10\}}$
	N64FA	$0.189^{\{5\}}$	$\boxed{0.343}^{\{8\}}$	$0.542^{\{10\}}$	$0.629^{\{10\}}$	$\boxed{0.600}^{\{10\}}$
Binary	ANS2B	1.38	1.30	1.32	1.38	3.62
	BYPSB	—	—	—	17.19	6.86
	TWSA	4.91	5.16	5.25	5.81	—
	N32A	$0.316^{\{5\}}$	$0.744^{\{8\}}$	$1.16^{\{10\}}$	$2.67^{\{8\}}$	$3.09^{\{8\}}$
	N32FA	$0.321^{\{5\}}$	$0.823^{\{8\}}$	$1.26^{\{10\}}$	$2.93^{\{12\}}$	$3.60^{\{12\}}$
	N64A	$\boxed{0.202}^{\{5\}}$	$\boxed{0.419}^{\{8\}}$	$\boxed{0.581}^{\{10\}}$	$\boxed{1.22}^{\{12\}}$	$\boxed{1.57}^{\{12\}}$
	N64FA	$0.202^{\{5\}}$	$0.429^{\{8\}}$	$0.598^{\{10\}}$	$1.38^{\{12\}}$	$1.70^{\{12\}}$

In general, adjusting the r value may lead to significant savings in search times—by doubling the search speed in many cases.

Table 4. Approximate search times, $k = 5$.

		$m = 8$	10	16	32
English	ANS2B	1.35	1.37	1.29	2.72
	BYPSC	—	—	—	3.86
	TWSA	6.07	5.36	3.50	—
	N32FA	$1.14^{\{8\}}$	$1.44^{\{10\}}$	$1.15^{\{9\}}$	$0.955^{\{9\}}$
	N32EA	$1.02^{\{8\}}$	$1.40^{\{10\}}$	$0.983^{\{9\}}$	$0.983^{\{9\}}$
	N64FA	$0.419^{\{8\}}$	$0.686^{\{10\}}$	$0.949^{\{7\}}$	$0.807^{\{7\}}$
	N64EA	$\boxed{0.381}^{\{8\}}$	$\boxed{0.653}^{\{10\}}$	$\boxed{0.833}^{\{7\}}$	$\boxed{0.664}^{\{7\}}$
DNA	ANS2B	1.33	1.10	1.13	2.23
	BYPSB	—	—	—	4.144
	TWSA	3.85	4.67	3.60	—
	N32A	$0.587^{\{8\}}$	$1.74^{\{10\}}$	$2.52^{\{8\}}$	$2.57^{\{8\}}$
	N32FA	$0.611^{\{8\}}$	$1.12^{\{10\}}$	$2.45^{\{8\}}$	$2.70^{\{8\}}$
	N64A	$\boxed{0.301}^{\{8\}}$	$\boxed{0.493}^{\{10\}}$	$\boxed{1.28}^{\{9\}}$	$\boxed{1.28}^{\{9\}}$
	N64FA	$0.332^{\{8\}}$	$0.548^{\{10\}}$	$\boxed{1.28}^{\{9\}}$	$1.37^{\{9\}}$
Binary	ANS2B	1.27	1.21	1.26	6.62
	BYPSB	—	—	—	18.78
	TWSA	4.74	4.84	5.31	—
	N32A	$0.747^{\{8\}}$	$1.30^{\{10\}}$	$2.93^{\{4\}}$	$4.66^{\{3\}}$
	N32FA	$0.704^{\{8\}}$	$1.26^{\{10\}}$	$3.07^{\{4\}}$	$5.07^{\{3\}}$
	N64A	$\boxed{0.375}^{\{8\}}$	$\boxed{0.649}^{\{10\}}$	$\boxed{1.57}^{\{7\}}$	$\boxed{2.71}^{\{7\}}$
	N64FA	$0.379^{\{8\}}$	$0.651^{\{10\}}$	$1.62^{\{7\}}$	$2.95^{\{7\}}$

Table 5. Search times of N64FA with varied peeling factor, $k = 1$.

	r	$m = 5$	8	10	16		r	$m = 5$	8	10	16
English	2	0.388	0.385	0.407	0.363	Binary	4	0.233	0.422	0.760	1.10
	3	0.376	0.363	0.373	0.339		5	$\boxed{0.208}$	0.412	0.732	1.05
	4	0.248	0.226	0.219	$\boxed{0.188}$		6	—	0.400	0.704	1.054
	5	$\boxed{0.208}$	$\boxed{0.220}$	$\boxed{0.206}$	0.196		7	—	0.376	0.637	0.995
	6	—	0.242	0.246	0.252		8	—	$\boxed{0.319}$	0.647	0.997
DNA	2	0.371	0.624	0.639	0.648		9	—	—	0.625	0.988
	3	0.376	0.684	0.703	0.738		10	—	—	$\boxed{0.384}$	1.03
	4	0.346	0.639	0.653	0.663		11	—	—	—	0.785
	5	$\boxed{0.205}$	0.539	0.559	0.559		12	—	—	—	0.657
	6	—	0.379	0.384	0.385		13	—	—	—	0.594
	7	—	0.280	0.278	0.283		14	—	—	—	$\boxed{0.539}$
	8	—	$\boxed{0.249}$	$\boxed{0.262}$	$\boxed{0.277}$		15	—	—	—	0.543
	9	—	—	0.279	0.290		16	—	—	—	0.581

5.4 Choice of comparison vectors

Algorithm 1 constructs comparison vectors for each character of the alphabet, whereas Algorithm 2 constructs vectors for each position of the pattern. The latter approach offers computational advantage during the search process, potentially resulting in faster running times. To verify this, we tested N64FA and N32FA on our main test processor equipped with AVX-512. Surprisingly, both approaches performed equally well for both algorithms on this processor.

To further investigate the performance on different processors, we tested N32FA with both approaches on three other processors without AVX-512 but with AVX2. In

this scenario, the latter approach (used in Algorithm 2) exhibited a speed improvement of approximately 5–10 percent compared to the former approach.

5.5 Exact matching

We compared the 64 byte variations of Algorithm 1 (N64 for the plain order, N64F for the fixed order, and N64E for the reverse English frequency order) against EPSM [4], EPSMA [1], and the corresponding variations of N32 [12]. EPSM and EPSMA were clearly the best for $m \leq 16$ in the extensive experimental comparison of [1].

Table 6 demonstrates that the N64 variations are clearly faster than the N32 and EPSM variations for $5 \leq m \leq 16$. The speedup becomes particularly noticeable in the case of binary data. However, the gain of the AVX-512 technology is smaller than in the case of approximate matching. For example, the speedup of N64F over N32F is 1.17 but the speedup of N64FA over N32FA is 1.41 for $k = 1$ both in the case of English data for $m = 16$.

Table 6. Exact search times.

		$m = 5$	8	10	16
English	EPSM	0.429	0.409	0.451	0.386
	EPSMA	0.280	0.306	0.330	0.239
	N32F	$0.180^{\{3\}}$	$0.163^{\{3\}}$	$0.170^{\{3\}}$	$0.164^{\{3\}}$
	N64F	$0.164^{\{4\}}$	$0.148^{\{3\}}$	$\boxed{0.148}^{\{3\}}$	$\boxed{0.141}^{\{3\}}$
	N64E	$\boxed{0.154}^{\{4\}}$	$\boxed{0.145}^{\{4\}}$	$0.150^{\{4\}}$	$0.143^{\{4\}}$
DNA	EPSM	0.469	0.476	0.495	0.314
	EPSMA	0.234	0.456	0.463	0.265
	N32	$0.168^{\{5\}}$	$0.205^{\{5\}}$	$0.205^{\{5\}}$	$0.212^{\{5\}}$
	N32F	$0.168^{\{5\}}$	$0.196^{\{5\}}$	$0.203^{\{5\}}$	$0.204^{\{5\}}$
	N64	$\boxed{0.152}^{\{5\}}$	$\boxed{0.165}^{\{6\}}$	$\boxed{0.165}^{\{6\}}$	$\boxed{0.159}^{\{6\}}$
	N64F	$0.153^{\{5\}}$	$0.167^{\{6\}}$	$0.169^{\{6\}}$	$0.173^{\{6\}}$
Binary	EPSM	4.92	4.85	5.15	0.550
	EPSMA	0.280	4.87	4.91	1.35
	N32	$0.177^{\{5\}}$	$0.237^{\{8\}}$	$0.323^{\{10\}}$	$0.344^{\{11\}}$
	N32F	$0.191^{\{5\}}$	$0.277^{\{8\}}$	$0.327^{\{10\}}$	$0.460^{\{11\}}$
	N64	$0.162^{\{5\}}$	$\boxed{0.198}^{\{8\}}$	$\boxed{0.230}^{\{10\}}$	$\boxed{0.259}^{\{16\}}$
	N64F	$\boxed{0.160}^{\{5\}}$	$0.202^{\{8\}}$	$0.236^{\{10\}}$	$0.282^{\{11\}}$

6 Conclusions

We have introduced new algorithms for the k mismatches problem. N32A and its variations utilize SIMD instructions based on the AVX2 technology. We adapted N32A into N64A which applies the AVX-512 technology. As a side result, we got N64, an adaptation of an earlier algorithm, for exact string matching with AVX-512.

We have presented an experimental analysis of variations of N32A, N64A, and N64. Through comparisons with earlier algorithms, we have demonstrated their excellent performance for short patterns. Notably, we have observed a substantial speed improvement by executing instructions that process 64 bytes simultaneously. Additionally, our experiment underscores the critical role of loop peeling in enhancing the performance of these new algorithms.

References

1. M. A. Aydogmus and M. O. Külekci: *Optimizing packed string matching on AVX2 platform*, in High Performance Computing for Computational Science — VECPAR 2018 — 13th International Conference, São Pedro, Brazil, September 17-19, 2018, Revised Selected Papers, H. Senger, O. Marques, R. E. Garcia, T. P. de Brito, R. Iope, S. L. Stanzani, and V. Gil-Costa, eds., vol. 11333 of Lecture Notes in Computer Science, Springer, 2018, pp. 45–61.
2. T. Chhabra, S. Faro, M. O. Külekci, and J. Tarhio: *Engineering order-preserving pattern matching with SIMD parallelism.* Software: Practice and Experience, 47(5) 2017, pp. 731–739.
3. B. Durian, T. Chhabra, S. S. Ghuman, T. Hirvola, H. Peltola, and J. Tarhio: *Improved two-way bit-parallel search*, in Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 71–83.
4. 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.
5. 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.
6. F. J. Fiori, W. Pakalén, and J. Tarhio: *Approximate string matching with SIMD.* Comput. J., 65(6) 2022, pp. 1472–1488.
7. S. S. Ghuman, J. Tarhio, and T. Chhabra: *Improved online algorithms for jumbled matching.* Discret. Appl. Math., 274 2020, pp. 54–66.
8. A. Hume and D. Sunday: *Fast string searching.* Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
9. Intel: *Intel intrinsics guide*, https://www.intel.com/content/www/us/en/docs/intrinsics-guide, Accessed: 2023-05-20.
10. G. Navarro: *A guided tour to approximate string matching.* ACM Comput. Surv., 33(1) 2001, pp. 31–88.
11. W. Pakalén, H. Peltola, J. Tarhio, and B. W. Watson: *Pitfalls of algorithm comparison*, in Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021, pp. 16–29.
12. J. Tarhio, J. Holub, and E. Giaquinta: *Technology beats algorithms (in exact string matching).* Software: Practice and Experience, 47(12) 2017, pp. 1877–1885.

On Expressive Power of Regular Expressions with Subroutine Calls and Lookaround Assertions

Ondřej Guth

Czech Technical University in Prague
Thákurova 9
16000 Praha
Czech Republic
ondrej.guth@fit.cvut.cz

Abstract. Many regular expression engines employ syntactical extensions to provide simple, expressive support for real-world needs. These features are subroutine calls, zero-width lookaround assertions, DEFINE rules, and named parenthesised expressions. A subroutine call executes a specified subpattern where the call is placed, possibly recursively. Lookaround assertions are either lookahead or lookbehind: a lookahead is a conditional within a subpattern: when it fails, the match at the current position of the whole subpattern fails, while a lookahead itself does not consume any input; a lookbehind works as a lookahead except it checks the input prior to the current position. A DEFINE rule introduces a subpattern for use by a subroutine call, while not involved in matching where the rule is placed. A named parenthesised expression can be executed by its name in addition to the parenthesis number. This paper presents a formalisation of subroutine calls, DEFINE rules, and named parenthesised expressions using the matching relation while attempting to mimic the behaviour of real-world regular expression engines. Also, we give an alternative constructive proof of equivalence of expressive power of regular expressions extended with subroutine calls and the class of context-free languages: a conversion between such expressions and context-free grammars. Finally, the question of whether regular expressions with operations lookaround assertion combined with subroutine call have greater expressive power than expressions with only subroutine call is answered positively.

1 Introduction

Regular expressions were introduced by Kleene[13] as a theoretical concept with expressive power equivalent to regular languages (these are further referred to as classical regular expressions or RE). This concept plays an important role in pattern matching and its variants with multiple (finite or infinite) patterns (see the taxonomy of pattern matching problems by Melichar and Holub[16]). So-called regular expressions have been implemented in many tools (e.g., UNIX text filters, text editors), programming languages, and libraries (these expressions are often referred to as extended regular expressions, practical regular expressions, or regexes). Unlike classical regular expressions, regexes "seem to have been invented entirely on the level of software implementation, without prior theoretical formalisation" (Schmid[22]). Moreover, both the syntax and semantics of the regex flavours used in implementations differ from each other (differences among the flavours were described by Friedl[8]). However, researchers have been exploring the algorithmic and language properties of practical regular expressions. Research results address properties of combinations of particular features used in regexes rather than complete flavours.

One of the fields in which this paper is concerned focuses on the expressive power of regexes and its relation to known language classes. Some syntactic constructs (not

Ondřej Guth: On Expressive Power of Regular Expressions with Subroutine Calls and Lookaround Assertions, pp. 68–82.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

used in classical regular expressions) are known to be mere syntax sugar: they can be rewritten to equivalent REs. Among these constructs are positive iteration (e.g., `a+`), character class (e.g., `[abc]`), or counting constraint (interval quantifier, e.g., `a{3,8}`) as pointed out, among others, by Câmpeanu et al.[4]. However, some features of regexes impact their expressive power: nonregular languages can be matched. A backreference indicates that a substring matched by a corresponding parenthesised subpattern should be matched again at the positions where the backreference is placed. Regexes with backreferences can match a proper subset of the class of context-sensitive languages (the expressive power of backreferences was studied, among others, by Câmpeanu et al.[4], Berglund et al.[2], or Schmid[22]). The expressive power of a practical regular expression with features of RE extended by lookahead stays within regular languages (Berglund et al.[3]). Regexes with subroutine calls describe context-free languages (addressed in master's thesis by Hruša[12]). When a practical regular expression with backreferences is extended by lookaheads, its expressive power supersedes the expressive power of regex with only backreferences (Chida and Terauchi[5]). Therefore, it is natural to wonder whether adding lookahead to a regex with subroutine calls also impacts expressive power. This question is addressed in this paper.

The formalisation of syntax and semantics of features of practical regular expressions is an essential part of proving their expressive power. Aho[1] gave a relatively informal definition of a regex with backreferences: the definition uses named variables, while any variable can be reassigned. Câmpeanu, Salomaa, and Yu[4] precisely formalised the numbered backreferences. Another formalisation of backreferences, factor-referencing, was introduced by Schmid[22]: it uses named variables which can be reassigned, and unlike the first formalisation[1], it deals with details of both syntax and semantics. Regexes with lookahead were originally formalised by Morihata[17] according to Berglund et al.[3]: the definition uses lookahead language. Chida and Terauchi[6] formalised the regexes with lookaheads and numbered backreferences using the matching relation. The syntax and semantics of the regexes with numbered subroutine calls were defined by Hruša[12]. To the author's knowledge, there is no formalisation of DEFINE rules, named parenthesised expressions, or named subroutine calls. This paper fills this gap by extending the notion matching relation.

Finding the expressive power of regex flavours is motivated by more than scientific curiosity. Users of pattern matching tools need to know what can and can not be matched by particular flavours of practical regular expressions[1].

To the author's knowledge, there has been almost no research on the expressive power of regexes with subroutine calls. The first known text dealing with this gap was published as a blog post by Popov[19] providing a sketch of a reduction of context-free grammars to regexes with DEFINE rules and named subroutine calls. Popov's claim of the equivalence of expressive power of regexes with subroutine calls and context-free languages was later formally proved by Hruša[12] while using numbered-only subroutine calls. This paper gives an alternative proof to the Hruša's while using the matching relation and regexes with DEFINE rules and named subroutine calls. We hope that our approach is more straightforward and extensible in future research.

[1] As shown by several discussions at Stack Overflow, for example, `https://stackoverflow.com/q/35449863`, `https://stackoverflow.com/q/2974210`, or `https://stackoverflow.com/q/4840988`.

To the author's knowledge, there is no peer-reviewed publication on the expressive power of practical regular expressions with both subroutine calls and lookaround assertions. The problem seems to have a solution due to Popov's[19] sketch (or, more precisely, an idea) of a reduction from context-sensitive grammars to regexes[2]. However, we show a counterexample. In addition, we present proof that the expressive power of practical regular expressions with lookaround assertions and subroutine calls is greater than the expressive power of expressions with only subroutine calls.

This paper focuses on the expressive power of regex with subroutine calls (subpattern recursion) and lookaround assertions. Our contributions are the following:

- We give a formalisation of practical regular expressions by extending the notion of the matching relation. In particular, this paper gives the formalisation of constructs named parenthesised expression, DEFINE rule, and both numbered and named subroutine calls. Our formalisation mimics the syntax and semantics of Perl-Compatible Regular Expressions (PCRE2, as documented on the manual page[10] and as we experimentally verified). At the same time, it also works for Perl regular expressions[18] and Ruby Regexp class[20].
- We prove that the expressive power of regex with concatenation, alternative, DEFINE rule, and subroutine call is equal to the class of context-free languages. This proof is based on the matching relation and works for regex with numbered and named subroutines.
- We show that adding lookaround assertions to regexes with subroutine calls extends their expressive power beyond context-free languages. In addition, we also show that the equivalence of the expressive power of these regexes to the class of context-sensitive languages remains an open problem.

This paper is structured as follows. In section 2, we give notational preliminaries. Section 3 contains the formalisation of practical regular expressions with subroutine calls. In section 4, we present proof that the expressive power of regexes with subroutine calls is equal to context-free languages: a conversion between such a regex and context-free grammar. Section 5 contains proof that adding lookaround assertion extends the expressive power of practical regular expressions with subroutine call. In section 6, we conclude the paper and discuss future work.

2 Preliminaries

The set of natural numbers is denoted by $\mathbb{N}$ and is without zero. The mathematical symbols $\emptyset$, $\cup$, $\cap$, $\setminus$, $\wedge$, $\forall$, and $\nexists$ denote the empty set, set union, set intersection, set difference, logical conjunction, universal quantifier, and negated existential quantifier, respectively. If $\mathcal{V}$ and $\mathcal{X}$ are sets, $\mathcal{V}$ being a subset (or a strict subset) of $\mathcal{X}$ is denoted by $\mathcal{V} \subseteq \mathcal{X}$ (or $\mathcal{V} \subsetneq \mathcal{X}$, respectively). An *alphabet*, denoted by $\mathcal{A}$, is a finite nonempty set whose elements are called *symbols*. A *string* over $\mathcal{A}$ is a finite sequence of elements of $\mathcal{A}$. The empty sequence is called the *empty string* and is denoted by ε. *The set of all strings* over $\mathcal{A}$ is denoted by $\mathcal{A}^*$ and *the set of all nonempty strings* over $\mathcal{A}$ is denoted by $\mathcal{A}^+$. The *length* of a string $\boldsymbol{y}$ is the length of the sequence associated with

[2] The reduction of context-sensitive grammars to regexes seem to be trusted: for example, a Stack Exchange contributor claims that regexes with subroutine calls and lookaround assertions can express any context-sensitive language using Popov's argument: `https://cs.stackexchange.com/q/143221`.

$\boldsymbol{y}$ and is denoted by $|\boldsymbol{y}|$. By $\boldsymbol{y}[i]$, where $i \in \mathbb{N} \wedge i \in \{1, \ldots, |\boldsymbol{y}|\}$, we denote the symbol at the index i of $\boldsymbol{y}$. The concatenation of strings $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ is denoted by $\boldsymbol{y}_1\boldsymbol{y}_2$. Thus, $\boldsymbol{y} = \boldsymbol{y}[1]\boldsymbol{y}[2]\ldots\boldsymbol{y}[|\boldsymbol{y}|]$. The substring of $\boldsymbol{y}$ that starts at the index i and ends at the index g is denoted by $\boldsymbol{y}[i..g]$; that is, $\boldsymbol{y}[i..g] = \boldsymbol{y}[i]\boldsymbol{y}[i+1]\ldots\boldsymbol{y}[g]$. A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$, denoted by $\mathcal{L} \subseteq \mathcal{A}^*$. The *concatenation of languages* $\mathcal{L}_1, \mathcal{L}_2$ is denoted by $\mathcal{L}_1 \cdot \mathcal{L}_2$ and is defined as $\mathcal{L}_1 \cdot \mathcal{L}_2 = \{\boldsymbol{y} = \boldsymbol{y}_1\boldsymbol{y}_2 : \boldsymbol{y}_1 \in \mathcal{L}_1 \wedge \boldsymbol{y}_2 \in \mathcal{L}_2\}$. The *closure of a language* $\mathcal{L}$ is denoted by $\mathcal{L}^*$ and is defined as $\bigcup_{g\geq 0} \mathcal{L}^g$ where $\mathcal{L}^0 = \{\varepsilon\}$, $\mathcal{L}^1 = \mathcal{L}$, and for $g > 1$: $\mathcal{L}^g = \mathcal{L} \cdot \mathcal{L}^{g-1}$.

The following grammar-related notions follow the conventions of Hopcroft et al.[11] and Mateescu et al.[15]. A *grammar* is a quadruple $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ where $\mathcal{A} \cap \mathcal{V} = \emptyset$, $S \in \mathcal{V}$, and $\mathcal{R}$ is a set of pairs $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ where $\boldsymbol{v}_1 \in (\mathcal{A} \cup \mathcal{V})^* \cap \mathcal{V}^+$ and $\boldsymbol{v}_2 \in (\mathcal{A} \cup \mathcal{V})^*$. The sets $\mathcal{V}$ (nonterminals), $\mathcal{A}$, and $\mathcal{R}$ are finite. We use the following naming and typographic conventions: $a \in \mathcal{A}$, $N \in \mathcal{V}$, $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{A}^*$, and $\boldsymbol{v} \in (\mathcal{A} \cup \mathcal{V})^*$ (bold italic sans serif for a string that can contain a nonterminal). The members of the set $\mathcal{R}$ are called *productions* and are written with $\to$ as a delimiter of the left- and right-hand side. Multiple productions with the same left-hand side can be contracted: for instance, if $\boldsymbol{v} \to \boldsymbol{v}_1, \boldsymbol{v} \to \boldsymbol{v}_2 \in \mathcal{R}$ then we can write $\boldsymbol{v} \to \boldsymbol{v}_1 \mid \boldsymbol{v}_2 \in \mathcal{R}$. A *derivation step* in grammar $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ is denoted by $\underset{\mathfrak{G}}{\Rightarrow}$ and defined as follows. If $\boldsymbol{v}_1 \in (\mathcal{A} \cup \mathcal{V})^+$, $\boldsymbol{p}, \boldsymbol{s}, \boldsymbol{v}_2 \in (\mathcal{A} \cup \mathcal{V})^*$, and $\boldsymbol{v}_1 \to \boldsymbol{v}_2 \in \mathcal{R}$ then $\boldsymbol{p}\boldsymbol{v}_1\boldsymbol{s} \underset{\mathfrak{G}}{\Rightarrow} \boldsymbol{p}\boldsymbol{v}_2\boldsymbol{s}$. The transitive closure of $\underset{\mathfrak{G}}{\Rightarrow}$ is denoted by $\underset{\mathfrak{G}}{\Rightarrow}^+$, and the reflective and transitive closure is denoted by $\underset{\mathfrak{G}}{\Rightarrow}^*$. If $S \underset{\mathfrak{G}}{\Rightarrow}^* \boldsymbol{v}$ then the string $\boldsymbol{v}$ is called a *sentential form* and $S \underset{\mathfrak{G}}{\Rightarrow}^* \boldsymbol{v}$ is called a *derivation* of $\boldsymbol{v}$ in $\mathfrak{G}$. The *language generated by grammar* $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ is denoted by $\mathrm{L}(\mathfrak{G})$ and is defined as $\mathrm{L}(\mathfrak{G}) = \{\boldsymbol{x} \in \mathcal{A}^* : S \underset{\mathfrak{G}}{\Rightarrow}^* \boldsymbol{x}\}$.

We relate language classes of practical regular expressions to the Chomsky hierarchy ([7]). A *context-sensitive grammar* is a grammar $(\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ where every member of $\mathcal{R}$ is of the form $\boldsymbol{p}N\boldsymbol{s} \to \boldsymbol{p}\boldsymbol{v}\boldsymbol{s}$ where $\boldsymbol{v} \neq \varepsilon$, and $\boldsymbol{p}, \boldsymbol{s} \in (\mathcal{A} \cup \mathcal{V})^*$. A *context-free grammar* is a grammar $(\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ where all members of $\mathcal{R}$ are of the form $N \to \boldsymbol{v}$.

A derivation $S \underset{\mathfrak{Gl}}{\Rightarrow}^* \boldsymbol{v}$ in a context-free grammar $\mathfrak{G}$ is called *the leftmost* if at each derivation step we replace the leftmost nonterminal (in a sentential form) by the right-hand side of one of its productions. A context-free grammar $(\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ is in *Greibach normal form* if every production is of the form $N \to a\boldsymbol{v}$. [9],[14, lecture 21]

A *context-sensitive language* is a language that is generated by some context-sensitive grammar. Context-free languages are defined likewise. The class of context-free languages is denoted by $\mathbb{L}_{\mathrm{CF}}$.

2.1 Regular expressions

The set of all classical regular expressions over an alphabet $\mathcal{A}$ is denoted by $\mathbb{E}_{\mathrm{C},\mathcal{A}}$. The syntax of classical regular expressions is given as follows (as defined in the literature[11,13] and adapted to conform conventions used in tools and libraries; operators are ordered by their precedence from the highest):

1. $\emptyset$ and ε are regular expressions,
2. for $a \in \mathcal{A}$, a is a regular expression,
3. for $\boldsymbol{r} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $(\boldsymbol{r})$ (parenthesised expression) is a regular expression,
4. for $\boldsymbol{r} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\boldsymbol{r}^*$ (iteration, Kleene star) is a regular expression.
5. for $\boldsymbol{r_1}, \boldsymbol{r_2} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\boldsymbol{r_1} \cdot \boldsymbol{r_2}$ or $\boldsymbol{r_1}\boldsymbol{r_2}$ (concatenation) is a regular expression,
6. for $\boldsymbol{r_1}, \boldsymbol{r_2} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\boldsymbol{r_1} \mid \boldsymbol{r_2}$ (alternative) is a regular expression.

The semantics of classical regular expressions (where the language matched by RE $\boldsymbol{r}$ is denoted by $\mathrm{L}(\boldsymbol{r})$) is given as follows[11]:

- $\mathrm{L}(\emptyset) = \emptyset$,
- $\mathrm{L}(\varepsilon) = \{\varepsilon\}$,
- for $a \in \mathcal{A}$, $\mathrm{L}(a) = \{a\}$,
- for $\boldsymbol{r} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\mathrm{L}((\boldsymbol{r})) = \mathrm{L}(\boldsymbol{r})$,
- for $\boldsymbol{r} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\mathrm{L}(\boldsymbol{r}^*) = \Big(\mathrm{L}(\boldsymbol{r})\Big)^*$,
- for $\boldsymbol{r_1}, \boldsymbol{r_2} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\mathrm{L}(\boldsymbol{r_1 r_2}) = \mathrm{L}(\boldsymbol{r_1}) \cdot \mathrm{L}(\boldsymbol{r_2})$,
- for $\boldsymbol{r_1}, \boldsymbol{r_2} \in \mathbb{E}_{\mathrm{C},\mathcal{A}}$, $\mathrm{L}(\boldsymbol{r_1} \mid \boldsymbol{r_2}) = \mathrm{L}(\boldsymbol{r_1}) \cup \mathrm{L}(\boldsymbol{r_2})$.

The set of all practical regular expressions with operations iteration, concatenation, alternative, DEFINE rule, lookaround assertion, subroutine calls, and numbered and named parenthesised subexpressions over alphabet $\mathcal{A}$ is denoted by $\mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ where the set of labels of named parenthesised expressions is denoted by $\mathcal{X}$ ($\mathcal{X} \cap \mathcal{A} = \emptyset$). The set of all regexes without lookaround assertions is denoted by $\mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$. Each practical regular expression consists of characters that may occur in the input string (i.e., $a \in \mathcal{A}$) and metacharacters that cannot occur in the input[3]: $(,), ?, =, <, > \notin \mathcal{A}$. For brevity, we write parentheses that denote a parenthesised expression with their assigned number, e.g., $({}_l)_l$. Our syntax closely follows the flavour PCRE2:

- the empty string, a character, iteration, concatenation, and alternative are defined the same way as for classical regular expressions,
- $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}} \wedge l \in \mathbb{N} : ({}_l\boldsymbol{r})_l \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$,
- $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}} \wedge l \in \mathbb{N} \wedge N \in \mathcal{X} : ({}_l?{<}N{>}\boldsymbol{r})_l \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ (parenthesised expression named N and numbered l),
- $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}} : (?(\mathrm{DEFINE})\boldsymbol{r}) \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ (DEFINE rule),
- $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}} : (?{=}\boldsymbol{r}) \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ (lookahead),
- $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}} : (?{<}{=}\boldsymbol{r}) \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ (lookbehind),
- $l \in \mathbb{N} : (?l) \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ (numbered subroutine call),
- $N \in \mathcal{X} : (?\mathrm{P}{>}N) \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ (named subroutine call).

The numbering of parenthesised expressions, both named and unnamed, is unique. (Note that neither parentheses around lookahead, lookbehind, nor subroutine call delimit a parenthesised expression.)

The semantics of regexes with numbered backreferences and lookaround assertions was defined using the matching relation by Chida and Terauchi[5,6]. We closely follow their definition[4]. A matching relation $\rightsquigarrow$ is of the form $(\boldsymbol{r}, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}$ where $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$, $\boldsymbol{x} \in \mathcal{A}^*$, $i \in \mathbb{N} \wedge i \leq |\boldsymbol{x}|$, and $\mathcal{R} = \{i : i \in \mathbb{N} \wedge i \leq |\boldsymbol{x}| + 1\}$ (matching result). The rules for deriving the matching relation for practical regular expressions with lookaround assertions are as follows:

$$\frac{}{(\emptyset, \boldsymbol{x}, i) \rightsquigarrow \emptyset}$$

$$\frac{}{(\varepsilon, \boldsymbol{x}, i) \rightsquigarrow \{i\}}$$

[3] For brevity, we deviate from the way real-world engines treat metacharacters: they can occur in the input and can be matched in a regex when following a special escaping metacharacter (some flavours can match some metacharacters even without escaping), mostly the backslash. We refer the reader to Friedl[8].

[4] We omit capturing environment as this paper does not deal with backreferences.

$$\frac{a \in \mathcal{A} \wedge i \leq |\boldsymbol{x}| \wedge \boldsymbol{x}[i] = a}{(a, \boldsymbol{x}, i) \rightsquigarrow \{i+1\}}, \frac{a \in \mathcal{A} \wedge (i > |\boldsymbol{x}| \vee \boldsymbol{x}[i] \neq a)}{(a, \boldsymbol{x}, i) \rightsquigarrow \emptyset}$$

$$\frac{(\boldsymbol{r}, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R} \wedge \forall i_h \in \mathcal{R} \setminus \{i\} : (\boldsymbol{r}^*, \boldsymbol{x}, i_h) \rightsquigarrow \mathcal{R}_h}{(\boldsymbol{r}^*, \boldsymbol{x}, i) \rightsquigarrow \{i\} \cup \bigcup_{1 \leq h \leq |\mathcal{R} \setminus \{i\}|} \mathcal{R}_h} \tag{1}$$

$$\frac{(\boldsymbol{r}_1, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R} \wedge \forall (i_h) \in \mathcal{R} : (\boldsymbol{r}_2, \boldsymbol{x}, i_h) \rightsquigarrow \mathcal{R}_h}{(\boldsymbol{r}_1 \boldsymbol{r}_2, \boldsymbol{x}, i) \rightsquigarrow \bigcup_{1 \leq h \leq |\mathcal{R}|} \mathcal{R}_h} \tag{2}$$

$$\frac{(\boldsymbol{r}_1, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}_1 \wedge (\boldsymbol{r}_2, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}_2}{(\boldsymbol{r}_1 \mid \boldsymbol{r}_2, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}_1 \cup \mathcal{R}_2} \tag{3}$$

$$\frac{(\boldsymbol{r}, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}{((\boldsymbol{r}), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}} \tag{4}$$

$$\frac{(\boldsymbol{r}, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}{((?{=}\boldsymbol{r}), \boldsymbol{x}, i) \rightsquigarrow \{i \wedge \mathcal{R} \neq \emptyset\}} \tag{5}$$

$$\frac{\boldsymbol{y} \in \mathcal{A}^* \wedge (\boldsymbol{y}, \boldsymbol{x}[i - |\boldsymbol{y}|..i-1], 1) \rightsquigarrow \mathcal{R}}{((?{<}{=}\boldsymbol{y}), \boldsymbol{x}, i) \rightsquigarrow \{i \wedge \mathcal{R} \neq \emptyset\}} \tag{6}$$

The language of a regex $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ is $\mathrm{L}(\boldsymbol{r}) = \{\boldsymbol{x} : (\boldsymbol{r}, \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R} \wedge |\boldsymbol{x}| + 1 \in \mathcal{R}\}$. We also say that a string $\boldsymbol{x} \in \mathrm{L}(\boldsymbol{r})$ *matches* a regex $\boldsymbol{r}$ (similarly, $\mathrm{L}(\boldsymbol{r})$ is the language matched by $\boldsymbol{r}$). The class of all languages that can be matched by the regexes of $\mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ is denoted by $\mathbb{L}_{\mathrm{E_{LS}}}$. The class of all languages that can be matched by the regexes of $\mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ is denoted by $\mathbb{L}_{\mathrm{E_S}}$.

3 Formalizing expressions with subroutine calls and lookaround assertions

We now formally define the semantics of practical regular expressions with named parenthesised expressions, DEFINE rules, and numbered and named subroutine calls. Our definition is an extension of the matching relation in the previous section. In this section, the following notation is used: $i, l \in \mathbb{N}; N \in \mathcal{X}; \boldsymbol{r}, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3 \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}; \boldsymbol{x} \in \mathcal{A}^*$.

The subroutine call attempts to match a given parenthesised expression at a current position. To be able to use the subexpression given the parenthesis number, the partial function $\sigma_{\boldsymbol{r}}$ is computed before the matching of regex $\boldsymbol{r}$ starts; it is defined as follows:

$$\sigma_{\boldsymbol{r}} : \mathbb{N} \to \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}} \wedge \boldsymbol{r} = \boldsymbol{r}_1({}_l\boldsymbol{r}_2)_l\boldsymbol{r}_3 \text{ implies } \sigma_{\boldsymbol{r}}(l) = \boldsymbol{r}_2$$

If no confusion can arise, we use σ for simplicity. In addition to being unambiguous, the numbering of parenthesised expressions (both named and unnamed) is not important for our results. Our definition conforms to some flavours of practical regular expressions.[5]

Matching a numbered subroutine call means matching the subpattern given in l-th parentheses from the current position. The subroutine call can be located anywhere related to the referred subpattern (i.e., both forward and recursive calls are valid).

$$\frac{(\sigma(l), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}{((?l), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}$$

[5] Namely PCRE2 and Perl. Both PCRE2 and Perl even support enabling duplicate parenthesis numbers (it is not the default). Our definition needs to be modified for the Ruby Regexp class: numbers cannot be used when at least one named expression is present.

Matching a named subroutine call uses the parenthesised expression assigned to the given name. Named parenthesised expressions can be identified by either their name or number. Thus, a name is just an alias for the parenthesis number. The partial function $\nu_{\boldsymbol{r}}$ is computed before the matching of $\boldsymbol{r}$ starts. If the name is used for multiple parenthesised expressions, $\nu_{\boldsymbol{r}}$ assigns the name to the leftmost parentheses.

$\nu_{\boldsymbol{r}} : \mathcal{X} \to \mathbb{N} \wedge \boldsymbol{r} = \boldsymbol{r}_1({}_l?{<}N{>}\boldsymbol{r}_2)_l\boldsymbol{r}_3$ implies $\nu_{\boldsymbol{r}}(N) = l \wedge \nexists l' < l : \boldsymbol{r} = \boldsymbol{r}'_1({}_{l'}?{<}N{>}\boldsymbol{r}'_2)_{l'}\boldsymbol{r}'_3$

If no confusion can arise, we use ν for simplicity.

$$\frac{(\sigma(\nu(N)), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}{((?\mathrm{P}{>}N), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}} \tag{7}$$

This matching relation for named subroutine call closely mimics the semantics of PCRE2, Perl, and Ruby.

Any regex can be wrapped in a DEFINE rule. In addition to the possibility of extending ν or σ, the DEFINE rule does not affect the matching.

$$\frac{}{((?(\mathrm{DEFINE})\boldsymbol{r}), \boldsymbol{x}, i) \rightsquigarrow \{i\}} \tag{8}$$

DEFINE rules with the above-defined semantics are supported by PCRE2 and Perl.

4 Expressive power of subroutine call

We give a rigorous proof of the equivalence of expressive power of context-free languages and practical regular expressions with subroutine calls. Our proof is based on the matching relation and extends Hruša's work[12], which is based on Popov's claim[19].

Theorem 1. $\mathbb{L}_{\mathrm{E_S}} = \mathbb{L}_{\mathrm{CF}}$

To prove the class equivalence, we first show that every context-free grammar can be converted to a regex with subroutine calls. Later, we show that every such regex can be converted into context-free grammar.

Lemma 2. $\mathbb{L}_{\mathrm{CF}} \subseteq \mathbb{L}_{\mathrm{E_S}}$.

We show that every context-free grammar can be expressed by a practical regular expression with the following sufficient operations: concatenation, alternative, DEFINE rule with named parenthesised expression, and named subroutine call. Intuitively, such a regex contains all the building blocks of context-free grammar: concatenation, alternative, and the ability to reuse a subexpression, even recursively. The conversion is formally defined by algorithm 1 and definition 3. The restriction of conversion to Greibach normal form grammar does not change the expressive power: Any context-free grammar (and therefore any context-free language) can be expressed by a Greibach normal form grammar using a known transformation.[9][14, lecture 21]

Definition 3. *Let us have a context-free grammar* $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ *and a regex* $\boldsymbol{r} \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ *where* $\mathcal{V} = \mathcal{X}$. *The function* $\mathrm{rx} : (\mathcal{V} \cup \mathcal{A})^* \to \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ *is defined as follows: let* $\boldsymbol{v}_1, \boldsymbol{v}_2 \in (\mathcal{A} \cup \mathcal{V})^*$, $a \in \mathcal{A}$, *and* $N \in \mathcal{V}$ *then* $\mathrm{rx}(\varepsilon) = \varepsilon$, $\mathrm{rx}(a) = a$, $\mathrm{rx}(N) = (?\mathrm{P}{>}N)$, *and* $\mathrm{rx}(\boldsymbol{v}_1\boldsymbol{v}_2) = \mathrm{rx}(\boldsymbol{v}_1)\,\mathrm{rx}(\boldsymbol{v}_2)$.

Algorithm 1 Conversion of a context-free grammar to a regex

Input: a context-free grammar $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ in Greibach normal form
Output: a regex $\boldsymbol{r} \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ such that $\mathrm{L}(\mathfrak{G}) = \mathrm{L}(\boldsymbol{r})$

1. initialize $\boldsymbol{r} = \varepsilon$ and consider $\mathcal{X} = \mathcal{V}$
2. for all productions with a non-terminal N on the left-hand side ($N \to \boldsymbol{v}_{N1} \mid \cdots \mid \boldsymbol{v}_{Nm_N} \in \mathcal{R}$):
   (a) let $\boldsymbol{r}_{Ng}$ ($1 \leq g \leq m_N$) be constructed from the g-th right-hand side of the production for N ($\boldsymbol{v}_{Ng}$) by replacing the non-terminals with subroutine call: $\boldsymbol{r}_{Ng} = \mathrm{rx}(\boldsymbol{v}_{Ng})$
   (b) add a DEFINE rule with parenthesised expression named N containing the strings $\boldsymbol{r}_{Ng}$ constructed from right-hand side of these productions, i.e., let $\boldsymbol{r} = \boldsymbol{r}(?(\mathrm{DEFINE})(?{<}N{>}\boldsymbol{r}_{N1} \mid \boldsymbol{r}_{N2} \mid \cdots \mid \boldsymbol{r}_{Nm_N}))$
3. add the matching of the initial symbol, i.e., let $\boldsymbol{r} = \boldsymbol{r}(?\mathrm{P}{>}S)$

Lemma 4. *Let us have a Greibach normal form grammar $\mathfrak{G}$ and a practical regular expression $\boldsymbol{r}$ such that $\boldsymbol{r}$ is constructed by the algorithm 1 from $\mathfrak{G}$. For any possible derivation step in $\mathfrak{G}$, a possible step exists in the matching relation for $\boldsymbol{r}$.*

Proof. Let the grammar be $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$. Without loss of generality, we can assume that only the leftmost derivations are used to derive any sentential form. Let $\boldsymbol{p}N\boldsymbol{s} \underset{\mathfrak{Gl}}{\Longrightarrow} \boldsymbol{p}\boldsymbol{v}\boldsymbol{s}$ be a derivation step (recall that this derivation step is possible only if $N \to \boldsymbol{v} \in \mathcal{R}$) where $\boldsymbol{p} \in \mathcal{A}^*$, $\boldsymbol{s} \in \mathcal{V}^*$ and $\boldsymbol{v} \in (\mathcal{V} \cup \mathcal{A})^+$. Recall that due to algorithm 1, $\boldsymbol{r}$ is in the following form:

$$(?(\mathrm{DEFINE})\ldots)\cdots(?(\mathrm{DEFINE})(?{<}N{>}\cdots \mid \mathrm{rx}(\boldsymbol{v}) \mid \cdots))\cdots(?\mathrm{P}{>}S) \tag{9}$$

The following steps of the matching relation show that an expression of the form $\boldsymbol{p}(?\mathrm{P}{>}N)\,\mathrm{rx}(\boldsymbol{s})$ (from step 2a follows that $\mathrm{rx}(\boldsymbol{p}) = \boldsymbol{p}$) can always be resolved by the concatenation of $\boldsymbol{p}$, $\mathrm{rx}(\boldsymbol{v})$, and $\mathrm{rx}(\boldsymbol{s})$.

$$\dfrac{\dfrac{\dfrac{\ldots}{(\mathrm{rx}(\boldsymbol{v}_{N1}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hN1}} \cdots \dfrac{(\mathrm{rx}(\boldsymbol{v}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hNg}}{(\mathrm{rx}(\boldsymbol{v}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hNg}} \cdots \dfrac{\ldots}{(\mathrm{rx}(\boldsymbol{v}_{Nm_N}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hNm_N}}}{(\mathrm{rx}(\boldsymbol{v}_{N1}) \mid \cdots \mid \mathrm{rx}(\boldsymbol{v}) \mid \cdots \mid \mathrm{rx}(\boldsymbol{v}_{Nm_N}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hN} = \bigcup_{1\leq h'\leq m_N} \mathcal{R}_{hNh'}} \qquad \dfrac{\dfrac{(\sigma(l),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hN}}{(\sigma(\nu(N)),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hN}} \quad \dfrac{\ldots}{\forall i_{hN} \in \mathcal{R}_{hN} : (\mathrm{rx}(\boldsymbol{s}),\boldsymbol{x},i_{hN}) \rightsquigarrow \mathcal{R}_{h_{\boldsymbol{s}}N}}}{(?\mathrm{P}{>}N),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_{hN} \quad \forall i_{hN} \in \mathcal{R}_{hN} : ((?\mathrm{P}{>}N)\,\mathrm{rx}(\boldsymbol{s}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_h = \bigcup_{1\leq h_{\boldsymbol{s}}\leq|\mathcal{R}_{hN}|} \mathcal{R}_{h_{\boldsymbol{s}}N}}}{(\boldsymbol{p},\boldsymbol{x},i) \rightsquigarrow \mathcal{R}'} \quad \dfrac{\ldots}{\forall i_h \in \mathcal{R}' : ((?\mathrm{P}{>}N)\,\mathrm{rx}(\boldsymbol{s}),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_h = \bigcup_{1\leq h_{\boldsymbol{s}}\leq|\mathcal{R}_{hN}|} \mathcal{R}_{h_{\boldsymbol{s}}N}}}{(\boldsymbol{p}(?\mathrm{P}{>}N)\,\mathrm{rx}(\boldsymbol{s}),\boldsymbol{x},i) \rightsquigarrow \bigcup_{1\leq h\leq|\mathcal{R}'|} \mathcal{R}_h}$$

Note that $\mathcal{R}' = \{i + |\boldsymbol{p}|\}$ if $\boldsymbol{x}[i..i + |\boldsymbol{p}| - 1] = \boldsymbol{p}$ and $\mathcal{R}' = \emptyset$ otherwise. Furthermore, the matching result for $\mathrm{rx}(\boldsymbol{v})$, $\mathcal{R}_{hNg}$, is involved in the expression matching. □

The following lemma holds due to lemma 4.

Lemma 5. *Let us have a Greibach normal form grammar $\mathfrak{G}$ and a practical regular expression $\boldsymbol{r}$ such that $\boldsymbol{r}$ is constructed by the algorithm 1 from $\mathfrak{G}$. Then $\mathrm{L}(\mathfrak{G}) \subseteq \mathrm{L}(\boldsymbol{r})$.*

Lemma 6. *Let us have a Greibach normal form grammar $\mathfrak{G}$ and a practical regular expression $\boldsymbol{r}$ such that $\boldsymbol{r}$ is constructed by the algorithm 1 from $\mathfrak{G}$. Then $\mathrm{L}(\boldsymbol{r}) \subseteq \mathrm{L}(\mathfrak{G})$.*

Proof. Let the grammar be $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$ and $\boldsymbol{x} \in \mathrm{L}(\mathfrak{G})$. The general form of a regex constructed by the algorithm is (9) while all subpatterns $\mathrm{rx}(\boldsymbol{v})$ of $\boldsymbol{r}$ are of

the form $a_1\,\mathrm{rx}(\boldsymbol{s})$ where $a_1 \in \mathcal{A}$ and $\boldsymbol{s} \in \mathcal{V}^*$ (due to the normal Greibach form). Such a subpattern can exist only if $N \to a_1\boldsymbol{s} \in \mathcal{R}$ (due to step 2a). Following the matching relation, the initial step in matching, that is, (?P>S) which applies some subpattern of named parentheses S, $a_2\,\mathrm{rx}(\boldsymbol{s}')$, is possible only when $\boldsymbol{x}[1] = a_2$; such a subpattern is constructed only if $S \to a_2\boldsymbol{s}' \in \mathcal{R}$. Suppose, for contradiction, that $a_2\,\mathrm{rx}(\boldsymbol{s}')$ matches $\boldsymbol{x}$ and $\boldsymbol{x} \notin \mathrm{L}(\mathfrak{G})$. As $\boldsymbol{s}' = N_1N_2\ldots N_{|\boldsymbol{s}'|}$, the only possibility is that any of (?P>N_g) matches a substring $\boldsymbol{y}$ of $\boldsymbol{x}$ and $N_g \underset{\mathfrak{G}}{\Rightarrow}^* \boldsymbol{y}$ is not possible. However, due to the algorithm 1, the named parenthesised expressions in $\boldsymbol{r}$ contain only subpatterns corresponding to the right-hand sides of the productions in $\mathfrak{G}$. Due to the Greibach normal form, $\boldsymbol{r}$ can only match $\boldsymbol{x}$ using subpatterns in the same order as productions of $\mathfrak{G}$ are applied when generating $\boldsymbol{x}$. In any situation when a subroutine call (?P>N_g) occurs, $\mathfrak{G}$ can use any production for N_g, because the application of productions in context-free grammar is not restricted by their order or context. □

The validity of lemma 2 was shown by lemmas 5 and 6: any context-free language can be expressed by a regex with subroutine calls.

Lemma 7. $\mathbb{L}_{\mathrm{E_S}} \subseteq \mathbb{L}_{\mathrm{CF}}$.

We show that an equivalent context-free grammar can express any practical regular expression with operations concatenation, alternative, DEFINE rule with named parenthesised expression, and named subroutine call. We begin by showing that removing the other valid operations in $\mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ does not change the expressive power.

Lemma 8 (Redundancy of Kleene star). *Every regex of the form $\boldsymbol{r}^* \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ can be expressed as*

$$(?(\mathrm{DEFINE})(?<N>\varepsilon \mid \boldsymbol{r}(?\mathrm{P}>N)))(?\mathrm{P}>N) \tag{10}$$

where $N \notin \mathcal{X}$.

Proof. The matching relation can be derived as follows:

$$\dfrac{\dfrac{\dfrac{}{(\varepsilon,\boldsymbol{x},i) \rightsquigarrow \{i\}} \quad \dfrac{(\boldsymbol{r},\boldsymbol{x},i) \rightsquigarrow \mathcal{R}' \wedge \forall i_h \in \mathcal{R}' : ((?\mathrm{P}>N),\boldsymbol{x},i_h) \rightsquigarrow \mathcal{R}_h}{(\boldsymbol{r}(?\mathrm{P}>N),\boldsymbol{x},i) \rightsquigarrow \mathcal{R} = \bigcup_{1\le h\le|\mathcal{R}'|}\mathcal{R}_h}}{(\varepsilon \mid \boldsymbol{r}(?\mathrm{P}>N),\boldsymbol{x},i) \rightsquigarrow \mathcal{R}\cup\{i\}}}{\dfrac{((?(\mathrm{DEFINE})(?<N>\varepsilon \mid \boldsymbol{r}(?\mathrm{P}>N))),\boldsymbol{x},i) \rightsquigarrow \{i\} \quad \dfrac{(\sigma(\nu(N)),\boldsymbol{x},i) \rightsquigarrow \mathcal{R}\cup\{i\}}{((?\mathrm{P}>N),\boldsymbol{x},i) \rightsquigarrow \mathcal{R}\cup\{i\}}}{((10),\boldsymbol{x},i) \rightsquigarrow \mathcal{R}\cup\{i\}}}$$

It is clear that the matching result is the same as that of the Kleene star (1). Although the matching relation of the Kleene star (1) excludes i from rematching $\boldsymbol{r}^*$, this exclusion does not affect the positions in its matching result. □

The following two redundancies of a standalone parenthesised expression are straightforward and thus are left without proof. Each occurrence of parenthesised expression put inside a DEFINE rule retains its parenthesis number and name, and thus it does not affect any subroutine call.

Lemma 9 (Redundancy of named parenthesised expression). *A regex in the form $(?<N>\boldsymbol{r}) \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ (named parenthesised expression outside the DEFINE rule) can be expressed as*

$$(?(\mathrm{DEFINE})(?<N>\boldsymbol{r}))(?\mathrm{P}>N)$$

Lemma 10 (Redundancy of numbered parenthesised expression). *A regex in the form* $({}_l\boldsymbol{r})_l \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ *(numbered parenthesised expression outside the DEFINE rule) can be expressed as*

$$(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r})_l)(?\mathrm{P}{>}N)$$

where $N \notin \mathcal{X}$.

The redundancy of the numbered subroutine call (and its replacement with the named subroutine call) is straightforward and thus left without proof.

Lemma 11 (Redundancy of numbered subroutine call). *Let* $\boldsymbol{r} \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ *be a regex with operations concatenation, alternative, DEFINE rule with named parenthesised expression, named subroutine call, and numbered subroutine call. To construct a regex* $\boldsymbol{r}' \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ *such that* $\mathrm{L}(\boldsymbol{r}) = \mathrm{L}(\boldsymbol{r}')$ *and* $\boldsymbol{r}'$ *does not contain numbered subroutine call,* $\boldsymbol{r}'$ *is the same as* $\boldsymbol{r}$ *with the following modifications: any occurrence of* $(?l)$ *from* $\boldsymbol{r}$ *is replaced with* $(?\mathrm{P}{>}N)$ *in* $\boldsymbol{r}'$ *where* l *refers to* $(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r}_l)_l)$.

The conversion of a regex to a context-free grammar is formally defined in algorithm 2, which is inspired by Thompson's[23] pattern matching algorithm as presented by Hopcroft et al.[11, theorem 3.7] and closure properties of context-free languages studied by Scheinberg[21] as presented by Kozen[14].

Algorithm 2 Conversion of a regex to a context-free grammar

Input: a practical regular expression $\boldsymbol{r} \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ with operations concatenation, alternative, DEFINE rule with named parenthesised expression, and named subroutine call
Output: a context-free grammar $\mathfrak{G}_{\boldsymbol{r}} = (\mathcal{V}_{\boldsymbol{r}}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}}, S_{\boldsymbol{r}})$ such that $\mathrm{L}(\boldsymbol{r}) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}})$

1. construct grammars for elementary expressions
   - $\mathfrak{G}_{\emptyset} = (\{S_{\emptyset}\}, \mathcal{A}, \emptyset, S_{\emptyset})$
   - $\mathfrak{G}_{\varepsilon} = (\{S_{\varepsilon}\}, \mathcal{A}, \{S_{\varepsilon} \to \varepsilon\}, S_{\varepsilon}\})$
   - $\mathfrak{G}_{a} = (\{S_a\}, \mathcal{A}, \{S_a \to a\}, S_a) : a \in \mathcal{A}$
   - $\mathfrak{G}_{(?\mathrm{P}>N)} = (\{S_{(?\mathrm{P}>N)}, N\}, \mathcal{A}, \{S_{(?\mathrm{P}>N)} \to N\}, S_{(?\mathrm{P}>N)}) : N \in \mathcal{X}$
2. iteratively construct grammars for operations in a regex (suppose $\mathfrak{G}_{\boldsymbol{r}_1} = (\mathcal{V}_{\boldsymbol{r}_1}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}_1}, S_{\boldsymbol{r}_1})$ and $\mathfrak{G}_{\boldsymbol{r}_2} = (\mathcal{V}_{\boldsymbol{r}_2}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}_2}, S_{\boldsymbol{r}_2})$ are already constructed for $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, respectively)
   - $\mathfrak{G}_{\boldsymbol{r}_1\boldsymbol{r}_2} = (\mathcal{V}_{\boldsymbol{r}_1} \cup \mathcal{V}_{\boldsymbol{r}_2} \cup \{S_{\boldsymbol{r}_1\boldsymbol{r}_2}\}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}_1} \cup \mathcal{R}_{\boldsymbol{r}_2} \cup \{S_{\boldsymbol{r}_1\boldsymbol{r}_2} \to S_{\boldsymbol{r}_1}S_{\boldsymbol{r}_2}\}, S_{\boldsymbol{r}_1\boldsymbol{r}_2}) : S_{\boldsymbol{r}_1\boldsymbol{r}_2} \notin \mathcal{V}_{\boldsymbol{r}_1} \cup \mathcal{V}_{\boldsymbol{r}_2}$
   - $\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2} = (\mathcal{V}_{\boldsymbol{r}_1} \cup \mathcal{V}_{\boldsymbol{r}_2} \cup \{S_{\boldsymbol{r}_1|\boldsymbol{r}_2}\}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}_1} \cup \mathcal{R}_{\boldsymbol{r}_2} \cup \{S_{\boldsymbol{r}_1|\boldsymbol{r}_2} \to S_{\boldsymbol{r}_1} \mid S_{\boldsymbol{r}_2}\}, S_{\boldsymbol{r}_1|\boldsymbol{r}_2}) : S_{\boldsymbol{r}_1|\boldsymbol{r}_2} \notin \mathcal{V}_{\boldsymbol{r}_1} \cup \mathcal{V}_{\boldsymbol{r}_2}$
   - $\mathfrak{G}_{(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r}_1)_l)} = (\mathcal{V}_{\boldsymbol{r}_1} \cup \{S_{(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r}_1)_l)}, N_l\}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}_1} \cup \{N_l \to S_{\boldsymbol{r}_1}, S_{(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r}_1)_l)} \to \varepsilon\}, S_{(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r}_1)_l)}) : S_{(?(\mathrm{DEFINE})({}_l?{<}N{>}\boldsymbol{r}_1)_l)}, N_l \notin \mathcal{V}_{\boldsymbol{r}_1}$
3. having grammar $\mathfrak{G}_{\boldsymbol{r}} = (\mathcal{V}_{\boldsymbol{r}}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}}, S_{\boldsymbol{r}})$, for every nonterminal $N \in \mathcal{X} \cap \mathcal{V}_{\boldsymbol{r}}$: if $N_{\nu(N)} \in \mathcal{V}_{\boldsymbol{r}}$ then replace all occurrences of N in the right-hand sides of productions $\mathcal{R}_{\boldsymbol{r}}$ with $N_{\nu(N)}$
4. return $\mathfrak{G}_{\boldsymbol{r}}$

Lemma 12. *Let* $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$. *Let* $\mathfrak{G}_{\boldsymbol{r}_1}, \mathfrak{G}_{\boldsymbol{r}_2}$ *be the grammars constructed by the algorithm 2 from* $\boldsymbol{r}_1, \boldsymbol{r}_2$, *respectively. If* $\mathrm{L}(\boldsymbol{r}_1) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1})$ *and* $\mathrm{L}(\boldsymbol{r}_2) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_2})$, *then* $\mathrm{L}(\boldsymbol{r}_1 \mid \boldsymbol{r}_2) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2})$ *and* $\mathrm{L}(\boldsymbol{r}_1\boldsymbol{r}_2) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1\boldsymbol{r}_2})$.

Proof. For the alternative of regexes $\boldsymbol{r}_1, \boldsymbol{r}_2$, following the matching relation (3), $\boldsymbol{r}_1 \mid \boldsymbol{r}_2$ matches some $\boldsymbol{x}$ if at least one of $\boldsymbol{r}_1, \boldsymbol{r}_2$ matches $\boldsymbol{x}$. Clearly, $\boldsymbol{x} \in \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2})$ if $\boldsymbol{x} \in \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1}) \vee \boldsymbol{x} \in \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_2})$. Following the well-known construction of a context-free grammar for the union of languages[21], $\mathrm{L}(\boldsymbol{r}_1 \mid \boldsymbol{r}_2) \subseteq \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2})$ because $\mathcal{V}_{\boldsymbol{r}_1} \cap \mathcal{V}_{\boldsymbol{r}_2}$ can be nonempty. Suppose, for contradiction, that $\boldsymbol{x} \notin \mathrm{L}(\boldsymbol{r}_1 \mid \boldsymbol{r}_2) \wedge \boldsymbol{x} \in \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2})$ while

$\mathrm{L}(\boldsymbol{r}_1) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1})$ and $\mathrm{L}(\boldsymbol{r}_2) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_2})$. The only way it can happen is $S_{\boldsymbol{r}_1|\boldsymbol{r}_2} \overset{}{\underset{\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2}}{\Longrightarrow}}^{+} \boldsymbol{p}_1 N_1 \boldsymbol{s}_1 \underset{\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2}}{\Longrightarrow} \boldsymbol{p}_1 \boldsymbol{v}_1 \boldsymbol{s}_1 \underset{\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2}}{\Longrightarrow}^{*} \boldsymbol{p}_2 N_2 \boldsymbol{s}_2 \underset{\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2}}{\Longrightarrow} \boldsymbol{p}_2 \boldsymbol{v}_2 \boldsymbol{s}_2 \underset{\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2}}{\Longrightarrow}^{*} \boldsymbol{x}$ where $N_1 \to \boldsymbol{v}_1$ and $N_2 \to \boldsymbol{v}_2$ are not from the same grammar $\mathfrak{G}_{\boldsymbol{r}_1}, \mathfrak{G}_{\boldsymbol{r}_2}$; in other words, $N_2 \in \mathcal{V}_{\boldsymbol{r}_1} \cap \mathcal{V}_{\boldsymbol{r}_2} \wedge N_2 \to \boldsymbol{v}_2 \notin \mathcal{R}_{\boldsymbol{r}_1} \cap \mathcal{R}_{\boldsymbol{r}_2}$. All grammars from step 2 introduce unique nonterminals that cannot appear in both $\mathcal{V}_{\boldsymbol{r}_1}$ and $\mathcal{V}_{\boldsymbol{r}_2}$. The nonterminals of grammars $\mathfrak{G}_\emptyset$, $\mathfrak{G}_\varepsilon$, and $\mathfrak{G}_a$ clearly cannot have different right-hand sides of productions in different grammars $\mathfrak{G}_{\boldsymbol{r}_1}, \mathfrak{G}_{\boldsymbol{r}_2}$. Both nonterminals introduced by $\mathfrak{G}_{(?\mathrm{P}>N)}$ can appear in both $\mathcal{V}_{\boldsymbol{r}_1}$ and $\mathcal{V}_{\boldsymbol{r}_2}$, however, due to step 3, every N from $\mathfrak{G}_{(?\mathrm{P}>N)}$ is replaced by a single N_l that rewrites to a unique nonterminal determined by a single DEFINE rule. Thus, it is not possible to achieve $N_2 \in \mathcal{V}_{\boldsymbol{r}_1} \cap \mathcal{V}_{\boldsymbol{r}_2} \wedge N_2 \to \boldsymbol{v}_2 \notin \mathcal{R}_{\boldsymbol{r}_1} \cap \mathcal{R}_{\boldsymbol{r}_2}$.

Similar arguments can be used to prove that $\mathrm{L}(\boldsymbol{r}_1\boldsymbol{r}_2) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1\boldsymbol{r}_2})$. □

Lemma 13. *If $\mathfrak{G}_{\boldsymbol{r}} = (\mathcal{V}_{\boldsymbol{r}}, \mathcal{A}, \mathcal{R}_{\boldsymbol{r}}, S_{\boldsymbol{r}})$ is constructed from $\boldsymbol{r}$ by algorithm 2 then $\mathrm{L}(\boldsymbol{r}) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}})$ for any $\boldsymbol{r} \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ (with operations concatenation, alternative, DEFINE rule with named parenthesised expression, and named subroutine call) and $\mathfrak{G}_{\boldsymbol{r}}$ is context-free.*

Proof. The grammars are clearly correct for the cases of elementary expressions $\emptyset, \varepsilon, a \in \mathcal{A}$. Assume that for $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, $\mathrm{L}(\boldsymbol{r}_1) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_1})$ and $\mathrm{L}(\boldsymbol{r}_2) = \mathrm{L}(\mathfrak{G}_{\boldsymbol{r}_2})$, respectively. For a subroutine call $(?\mathrm{P}{>}N)$, the matching relation (7) is

$$\frac{\frac{(\boldsymbol{r}_1, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}{(\sigma(l), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}}{\frac{(\sigma(\nu(N)), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}{((?\mathrm{P}{>}N), \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}}}$$

where $\mathcal{R}$ contains all i' such that $\boldsymbol{x}[i..i'-1]$ matches $\boldsymbol{r}_1$ and l identifies the leftmost parenthesised expression named N. Production $S_{(?\mathrm{P}>N)} \to N$ of $\mathfrak{G}_{(?\mathrm{P}>N)}$ is effectively $S_{(?\mathrm{P}>N)} \to N_{\nu(N)}$ due to step 3. Nonterminal $N_{\nu(N)}$ rewrites to $S_{\boldsymbol{r}_1}$. Therefore, the grammar $\mathfrak{G}_{(?\mathrm{P}>N)}$ generates the same language as is matched by $(?\mathrm{P}{>}N)$.

The grammar $\mathfrak{G}_{(?(\mathrm{DEFINE})(_l?<N>\boldsymbol{r}_1)_l)}$ follows the matching relation for the DEFINE rule (8). The correctness of both $\mathfrak{G}_{\boldsymbol{r}_1|\boldsymbol{r}_2}$ and $\mathfrak{G}_{\boldsymbol{r}_1\boldsymbol{r}_2}$ follows from lemma 12. Therefore, step 2 constructs the correct grammars.

All grammars add only productions with a single nonterminal on the left-hand side; therefore, all grammars constructed by the algorithm 2 are context-free. □

Lemma 2 is proved by lemmas 5 and 6; also, lemma 7 is proved by lemmas 8, 9, 10, 11, and 13. Therefore, theorem 1 is proved.

5 Expressive power of subroutine call combined with lookaround assertions

We show that lookaround assertion combined with subroutine call has greater expressive power than subroutine call alone. We use an example of such regex inspired by Popov's blog post[19] and arguments by Scheinberg[21].

Theorem 14. $\mathbb{L}_{\mathrm{E_S}} \subsetneq \mathbb{L}_{\mathrm{E_{LS}}}$

Proof. We show a regex $\boldsymbol{r} \in \mathbb{E}_{\mathrm{LS},\mathcal{A},\mathcal{X}}$ that matches a language that is not context-free (the equality of $\mathbb{L}_{\mathrm{CF}}$ and $\mathbb{L}_{\mathrm{E_S}}$ is shown by theorem 1). Language $\mathcal{L} = \{\mathtt{a}^g\mathtt{b}^g\mathtt{c}^g : g \in \mathbb{N}\}$ is a well-known language that is not context-free[11, example 7.19]. We show that for $\boldsymbol{r} = (?{=}(?{<}N_1{>}\mathtt{a}(\varepsilon \mid (?\mathrm{P}{>}N_1))\mathtt{b})\mathtt{c})\mathtt{a}\mathtt{a}^*(?{<}N_2{>}\mathtt{b}(\varepsilon \mid (?\mathrm{P}{>}N_2))\mathtt{c})$, $\mathcal{L} = \mathrm{L}(\boldsymbol{r})$. Let $\boldsymbol{r}_1 = (?{<}N_1{>}\mathtt{a}(\varepsilon \mid (?\mathrm{P}{>}N_1))\mathtt{b})\mathtt{c}, \boldsymbol{r}_2 = \mathtt{a}\mathtt{a}^*, \boldsymbol{r}_{N_2} = \mathtt{b}(\varepsilon \mid (?\mathrm{P}{>}N_2))\mathtt{c}$, and thus $\boldsymbol{r} = (?{=}\boldsymbol{r}_1)\boldsymbol{r}_2(?{<}N_2{>}\boldsymbol{r}_{N_2})$. Let $\boldsymbol{r}_{N_1} = \mathtt{a}(\varepsilon \mid (?\mathrm{P}{>}N_1))\mathtt{b}$. The matching relation for $\boldsymbol{r}_1$ can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{(\boldsymbol{r}_{N_1}, \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R}}}{((?{<}N_1{>}\boldsymbol{r}_{N_1}), \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R}} \quad \dfrac{\boldsymbol{x}[i] = \mathtt{c}}{\forall i \in \mathcal{R} : (\mathtt{c}, \boldsymbol{x}, i) \rightsquigarrow \{i+1 : \boldsymbol{x}[i] = \mathtt{c}\}}}{((?{<}N_1{>}\boldsymbol{r}_{N_1})\mathtt{c}, \boldsymbol{x}, 1) \rightsquigarrow \bigcup_{i\in\mathcal{R}}\{i+1 : \boldsymbol{x}[i] = \mathtt{c}\}}$$

Therefore, $\mathrm{L}(\boldsymbol{r}_1) = \mathrm{L}(\boldsymbol{r}_{N_1}) \cdot \{\mathtt{c}\}$. Let us derive the matching relation for $\boldsymbol{r}_{N_1}$:

$$\dfrac{\boldsymbol{x}[1] = \mathtt{a}}{(\mathtt{a}, \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R}_{\mathtt{a}}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dots}{(\boldsymbol{r}_{N_1}, \boldsymbol{x}, 2) \rightsquigarrow \mathcal{R}_{N_{1'}}}}{(\varepsilon, \boldsymbol{x}, 2) \rightsquigarrow \{2\}} \quad \dfrac{}{((?\mathrm{P}{>}N_1), \boldsymbol{x}, 2) \rightsquigarrow \mathcal{R}_{N_{1'}}}}{(\varepsilon \mid (?\mathrm{P}{>}N_1), \boldsymbol{x}, 2) \rightsquigarrow \mathcal{R}_{N_1} = \{2\} \cup \mathcal{R}_{N_{1'}}} \quad \dfrac{\boldsymbol{x}[i] = \mathtt{b}}{\forall i \in \mathcal{R}_{N_1} : (\mathtt{b}, \boldsymbol{x}, i) \rightsquigarrow \mathcal{R}_{\mathtt{b}_i}}}{((\varepsilon \mid (?\mathrm{P}{>}N_1)), \boldsymbol{x}, 2) \rightsquigarrow \mathcal{R}_{N_1} \quad \text{if } \boldsymbol{x}[1] = \mathtt{a} : ((\varepsilon \mid (?\mathrm{P}{>}N_1))\mathtt{b}, \boldsymbol{x}, 2) \rightsquigarrow \mathcal{R} = \bigcup_{i\in\mathcal{R}_{N_1}} \mathcal{R}_{\mathtt{b}_i}}$$

$$(\mathtt{a}(\varepsilon \mid (?\mathrm{P}{>}N_1))\mathtt{b}, \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R} \text{ if } \boldsymbol{x}[1] = \mathtt{a}$$

The regex $\boldsymbol{r}_{N_1}$ matches only if a prefix of $\boldsymbol{x}$ has the form $\mathtt{a}\dots\mathtt{a}\mathtt{b}\dots\mathtt{b}$. Furthermore, because as and bs are matched only within the same parenthesised expression and the same number of times, $\mathrm{L}(\boldsymbol{r}_{N_1}) = \{\mathtt{a}^g\mathtt{b}^g : g \in \mathbb{N}\}$. Clearly, $\mathrm{L}(\boldsymbol{r}_2) = \{\mathtt{a}\}^+$. Following similar arguments as for $\mathrm{L}(\boldsymbol{r}_{N_1})$, $\mathrm{L}(\boldsymbol{r}_{N_2}) = \{\mathtt{b}^g\mathtt{c}^g : g \in \mathbb{N}\}$. The matching relation for $\boldsymbol{r}$ can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{(\boldsymbol{r}_1, \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R}_1}}{((?{=}\boldsymbol{r}_1), \boldsymbol{x}, 1) \rightsquigarrow \{1 : \mathcal{R}_1 \neq \emptyset\}} \quad \dfrac{\dots}{\text{if } \mathcal{R}_1 \neq \emptyset : (\boldsymbol{r}_2(?{<}N_2{>}\boldsymbol{r}_{N_2}), \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R}}}{((?{=}(?{<}N_1{>}\mathtt{a}(\varepsilon \mid (?\mathrm{P}{>}N_1))\mathtt{b})\mathtt{c})\mathtt{a}\mathtt{a}^*(?{<}N_2{>}\mathtt{b}(\varepsilon \mid (?\mathrm{P}{>}N_2))\mathtt{c}), \boldsymbol{x}, 1) \rightsquigarrow \mathcal{R}}$$

The lookahead matches, following the matching relation (5), only if the regex $\boldsymbol{r}_1$ matches, while the current position in $\boldsymbol{x}$ is unchanged. Thus, $\boldsymbol{r}$ matches $\boldsymbol{x}$ if $\boldsymbol{x}$ starts with $\mathtt{a}^g\mathtt{b}^g\mathtt{c}$ and also has the form $\mathtt{a}^{g'}\mathtt{b}^g\mathtt{c}^g$. In other words, $\mathrm{L}(\boldsymbol{r}) = (\{\mathtt{a}^g\mathtt{b}^g\mathtt{c} : g \in \mathbb{N}\} \cdot \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*) \cap (\{\mathtt{a}^g : g \in \mathbb{N}\} \cdot \{\mathtt{b}^g\mathtt{c}^g : g \in \mathbb{N}\})$. □

5.1 Relation with context-sensitive languages

To the author's knowledge, there is no peer-reviewed or academic publication concerning the expressive power of practical regular expressions with both lookaround assertions and subroutine calls. The only known text on this topic is due to Popov[19]: an idea of what a reduction of context-sensitive grammars to regexes might look like.

Popov claims that having a context-sensitive grammar $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$, any production in the form $\boldsymbol{p}N\boldsymbol{s} \to \boldsymbol{p}\boldsymbol{v}\boldsymbol{s}$ can be converted into a DEFINE rule of the form $(?(\mathrm{DEFINE})(?{<}N{>}(?{<}{=}\,\mathrm{rx}(\boldsymbol{p}))\,\mathrm{rx}(\boldsymbol{v})(?{=}\,\mathrm{rx}(\boldsymbol{s}))))$. However, this alone does not work for all context-sensitive grammars. Let us attempt to formalize the conversion in algorithm 3 as a modification of algorithm 1:

Although a regex $\boldsymbol{r}$ matches any string generated by grammar $\mathfrak{G}$, it is still not correct because, in general, it can match more. To apply a production of the form

Algorithm 3 Conversion of a context-sensitive grammar to a regex ([19], incorrect)

Input: a context-sensitive grammar $\mathfrak{G} = (\mathcal{V}, \mathcal{A}, \mathcal{R}, S)$
Output: a regex $\boldsymbol{r} \in \mathbb{E}_{\mathrm{S},\mathcal{A},\mathcal{X}}$ such that $\mathrm{L}(\mathfrak{G}) = \mathrm{L}(\boldsymbol{r})$

1. initialize $\boldsymbol{r} = \varepsilon$ and consider $\mathcal{X} = \mathcal{V}$
2. for all $N \in \mathcal{V}$:
   (a) let $\boldsymbol{r}_N = \varepsilon$
   (b) for all productions with N on the left-hand side and particular left and right context ($\boldsymbol{p}N\boldsymbol{s} \to \boldsymbol{p}\boldsymbol{v}_{N1}\boldsymbol{s} \mid \cdots \mid \boldsymbol{p}\boldsymbol{v}_{Nm_N}\boldsymbol{s} \in \mathcal{R}$):
      i. let $\boldsymbol{r}_{Ng}$ $(1 \leq g \leq m_N)$ be constructed from $\boldsymbol{v}_{Ng}$ for $\boldsymbol{p}N\boldsymbol{s}$ ($\boldsymbol{v}_{Ng}$): $\boldsymbol{r}_{Ng} = \mathrm{rx}(\boldsymbol{v}_{Ng})$
      ii. if $\boldsymbol{r}_N = \varepsilon$ then $\boldsymbol{r}_N = (?<=\boldsymbol{p})\boldsymbol{r}_{Ng}(?=\boldsymbol{s})$ else $\boldsymbol{r}_N = \boldsymbol{r}_N \mid (?<=\boldsymbol{p})\boldsymbol{r}_{Ng}(?=\boldsymbol{s})$
   (c) let $\boldsymbol{r} = \boldsymbol{r}(?(\mathrm{DEFINE})(?<N>\boldsymbol{r}_N))$
3. add the matching of the initial symbol, i.e., let $\boldsymbol{r} = \boldsymbol{r}((?\mathrm{P}>S))$

$\boldsymbol{p}N\boldsymbol{s} \to \boldsymbol{p}\boldsymbol{v}\boldsymbol{s}$ in the generation of string $\boldsymbol{x}$, the left-hand side of the production must appear in a sentential form, that is, $S \underset{\mathfrak{G}}{\Rightarrow}^* \boldsymbol{p}'\boldsymbol{p}N\boldsymbol{s}\boldsymbol{s}' \underset{\mathfrak{G}}{\Rightarrow} \boldsymbol{p}'\boldsymbol{p}\boldsymbol{v}\boldsymbol{s}\boldsymbol{s}' \underset{\mathfrak{G}}{\Rightarrow}^* \boldsymbol{x}$. In other words, the context $(\boldsymbol{p}, \boldsymbol{s})$ of the production must already be present in the sentential form. Similarly to $\boldsymbol{p}N\boldsymbol{s} \to \boldsymbol{p}\boldsymbol{v}\boldsymbol{s}$ being part of generating a substring of $\boldsymbol{x}$, the subpattern $(?<=\boldsymbol{p})\boldsymbol{v}(?=\boldsymbol{s})$ matches a substring of $\boldsymbol{x}$. However, $(?<=\boldsymbol{p})\boldsymbol{v}(?=\boldsymbol{s})$ can match a substring of another $\boldsymbol{x}' \notin \mathrm{L}(\mathfrak{G})$ because during the matching of a regex, the order of the use of subpatterns is independent of the derivations of sentential forms by $\mathfrak{G}$. The following example illustrates this.

Example 15. Let us have $\mathfrak{G} = (\{N_1, N_2, S\}, \{\mathtt{a}, \mathtt{c}\}, \{S \to N_2N_1N_1, N_1 \to \mathtt{a}, \mathtt{a}N_1 \to \mathtt{aaa}, N_2\mathtt{a} \to \mathtt{caa}\}, S)$. Clearly, $\mathrm{L}(\mathfrak{G}) = \{\mathtt{caaa}, \mathtt{caaaa}\}$. After applying algorithm 3,

$$\boldsymbol{r} = (?(\mathrm{DEFINE})(?<N_1>\mathtt{a} \mid (?<=\mathtt{a})\mathtt{aa}))(?(\mathrm{DEFINE})(?<N_2>\mathtt{ca}(?=\mathtt{a}))) \cdot \\ \cdot(?(\mathrm{DEFINE})(?<S>(?\mathrm{P}>N_2)(?\mathrm{P}>N_1)(?\mathrm{P}>N_1)))(?\mathrm{P}>S)$$

and $\mathrm{L}(\mathfrak{G}) \subsetneq \mathrm{L}(\boldsymbol{r})$, as $\mathtt{caaaaa} \in \mathrm{L}(\boldsymbol{r})$: $\mathtt{caaaaa}[1..2]$ is matched by the subpattern $\mathtt{ca}(?=\mathtt{a})$, and both $\mathtt{caaaaa}[3..4]$ and $\mathtt{caaaaa}[5..6]$ are matched by the subpattern $(?<=\mathtt{a})\mathtt{aa}$. However, in $\mathfrak{G}$, there is no way to apply production $\mathtt{a}N_1 \to \mathtt{aaa}$ twice, as there must first be symbol $\mathtt{a}$ in a sentential form (which consumes one N_1).

As a result, the relation between the class of context-sensitive languages and the class of languages expressed by regexes with both subroutine calls and lookaround assertions remains an open problem.

6 Conclusions

We presented a formalisation of syntax and semantics of certain features of practical regular expressions using the matching relation: subroutine call, named parenthesised expression, and DEFINE rule. We attempted to mimic documented (and real) behaviour of certain flavours of practical regular expressions: Perl-compatible regular expressions, Perl, and Ruby Regexp class.

This paper showed the equivalence of context-free languages and languages expressed by practical regular expressions with concatenation, alternative, and subroutine call. This result applies to flavours that support subroutine calls. We presented an alternative constructive proof employing named subroutine calls, DEFINE rules, and the matching relation: a conversion between such practical regular expressions and context-free grammar.

We showed that adding zero-width lookaround assertions to practical regular expressions with operations concatenation, alternative, and subroutine call extends their expressive power beyond context-free languages. However, the relation of the language class expressed by such expressions to some non-context-free languages, particularly the class of context-sensitive languages, remains an open problem.

We hope that our results stimulate more work on the expressive power of specific combinations of operations used in practical regular expressions, such as backreferences, subroutine calls, lookaround assertions, or atomic groups.

References

1. A. V. Aho: *Algorithms for Finding Patterns in Strings*, in Algorithms and Complexity, J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier, 1990, pp. 255–300.
2. M. Berglund and B. van der Merwe: *Regular Expressions with Backreferences Re-examined*, in Proceedings of the Prague Stringology Conference 2017, Czech Technical University in Prague, 2017, pp. 30–41.
3. M. Berglund, B. Van Der Merwe, and S. van Litsenborgh: *Regular Expressions with Lookahead*. Journal of Universal Computer Science, 27(4) 2021, pp. 324–340.
4. C. Câmpeanu, K. Salomaa, and S. Yu: *A formal study of practical regular expressions*. International Journal of Foundations of Computer Science, 14(06) Dec. 2003, pp. 1007–1018.
5. N. Chida and T. Terauchi: *On lookaheads in regular expressions with backreferences*, in 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022), A. P. Felty, ed., vol. 228 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2022, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 15:1–15:18.
6. N. Chida and T. Terauchi: *Repairing DoS vulnerability of real-world regexes*, in 2022 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, May 2022, IEEE Computer Society, pp. 1049–1066.
7. N. Chomsky: *Three models for the description of language*. IEEE Transactions on Information Theory, 2(3) Sept. 1956, pp. 113–124.
8. J. E. F. Friedl: *Mastering Regular Expressions*, O'Reilly, Sebastapol, CA, 3rd ed., 2006.
9. S. A. Greibach: *A New Normal-Form Theorem for Context-Free Phrase Structure Grammars*. Journal of the ACM, 12(1) Jan. 1965, pp. 42–52.
10. P. Hazel: *PCRE2 — Perl-compatible regular expressions (revised API)*, University of Cambridge, Cambridge CB2 3QH, England, 2022, UNIX manual page PCRE2PATTERN(3) referring to PCRE2 10.40.
11. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Always Learning / Pearson, Pearson Education, Harlow, 3rd ed., 2014.
12. V. Hruša: *Regular Expressions with Subpattern Recursion*, Master's thesis, Czech Technical University in Prague, 2021.
13. S. C. Kleene: *Representation of Events in Nerve Nets and Finite Automata*, in Automata Studies. (AM-34), C. E. Shannon and J. McCarthy, eds., Princeton University Press, Dec. 1956, pp. 3–42.
14. D. C. Kozen: *Automata and Computability*, Undergraduate Texts in Computer Science, Springer, New York, Oct. 2012.
15. A. Mateescu and A. Salomaa: *Aspects of Classical Language Theory*, in Handbook of Formal Languages, G. Rozenberg and A. Salomaa, eds., Springer Berlin Heidelberg, 1997, pp. 175–251.
16. B. Melichar and J. Holub: *6D Classification of Pattern Matching Problems*, in Proceedings of the Prague Stringology Club Workshop '97, Czech Technical University in Prague, 1997.
17. A. Morihata: *Translation of regular expression with lookahead into finite state automaton*. Computer Software, 12(1) 2012, pp. 148–158.
18. *perlre — Perl regular expressions*, Perl Programmers Reference Guide, 2022, UNIX manual page PERLRE(1) referring to perl v5.36.0.
19. N. Popov: *The true power of regular expressions*. https://www.npopov.com/2012/06/15/The-true-power-of-regular-expressions.html, June 2012.

20. *class Regexp*, https://docs.ruby-lang.org/en/3.1/Regexp.html, The API documentation for Ruby 3.1.
21. S. Scheinberg: *Note on the boolean properties of context free languages.* Information and Control, 3(4) Dec. 1960, pp. 372–375.
22. M. L. Schmid: *Characterising REGEX languages by regular languages equipped with factor-referencing.* Information and Computation, 249 Oct. 2016.
23. K. Thompson: *Programming Techniques: Regular expression search algorithm.* Communications of the ACM, 11(6) June 1968, pp. 419–422.

Efficient Integer Retrieval from Unordered Compressed Sequences

Igor Zavadskyi

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorzavadskyi@knu.ua

Abstract. We investigate the problem of fast direct access to elements of an integer sequence given in a compressed form. If integers are sorted in ascending order, it can be reduced to performing the 'select' operation on a bitmap, which is very well investigated. We focus on the more general and more complicated case of unordered integer sequence and propose to represent it with the help of variable-length Reverse Multi-Delimiter (RMD) codes. When applied to data compression, these codes combine a good compression ratio with fast decoding. In this paper, another property of RMD codes is researched - the ability of direct access to codewords in the encoded bitstream. We present the method allowing us to extract and decode a codeword from an RMD-bitstream in almost constant time and experiment on its application to natural language text compression. Due to the properties of RMD codes and compactness of auxiliary direct access structures, our method appears to be very space efficient, only a few percent above the Shannon entropy.

1 Introduction

A compressed representation of integer sequences is the key element of different data compression techniques. It is used in frequency-based compression, when alphabet symbols are numbered so that smaller numbers are assigned to more frequent symbols, in the compact representation of suffix trees and arrays, offsets and lengths in LZ77-type compression, to mention a few areas. In most cases, the distribution of integers is biased in the direction of smaller ones. The variable length codes (VLC) provide a simple, space-efficient solution to the problem. However, often not only is compression itself required but also performing different operations on compressed integer sequence, such as sequential decoding or extracting the element with a given index. While VLC are well-suited for sequential decoding, the direct access to elements of a VLC-bitstream is not obvious and straightforward. It requires using auxiliary data structures and/or a special code construction.

Suppose integers are arranged in ascending order, and deltas between them are small enough. In that case, the problem of direct access is reduced to performing the *select* operation on a bitmap, where i-th bit is set if the number i belongs to the sequence (as usual, we denote by $select(B, i)$ the position of the i-th one in the bitstream B, while $rank(B, i)$ is equal to the number of ones from the beginning of the bitstream B up to position i). There are a lot of solutions concerning rank and select on bitmaps; the overview of recent results can be found in [13] and [15]. Less attention was paid to extracting an element of an unordered number sequence given in a compressed form. However, this operation is quite useful, e.g., in manipulating compressed texts or calculating values of the Ψ function used in compressed suffix

Igor Zavadskyi: Efficient Integer Retrieval from Unordered Compressed Sequences, pp. 83–96.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

arrays. Construction of data structures for space-efficient representation of unordered integer sequences allowing fast access to their elements is the goal of this paper.

Generally, we can distinguish five approaches to solve the mentioned problem more or less efficiently in terms of space and time. Here and below, we denote by n the number of elements in the integer sequence and assume the RAM model is used, allowing constant time reading values from memory.

1. Encode the sequence using universal codes, such as Elias codes [8]. Sample each h-th codeword and store pointers to sampled elements. To get the x-th element, obtain the position of the $\lfloor x/h \rfloor$-th sampled element and then search the bitstream sequentially. This approach is quite straightforward and requires $O(h)$ time to access an element.

2. Encode numbers with all binary strings of the shortest possible length and concatenate these codes. Construct an auxiliary bit sequence of the same length as the main bitstream denoting by '1' the starting positions of codewords. Then a given codeword can be extracted in constant time via known 'select' techniques applied to an auxiliary bit sequence. This method is discussed in [10] and called *Simple Dense Coding* (SDC). Its drawback is obvious: although the main bitstream can be quite succinct, the auxiliary bitstream doubles the space, which is more than significant.

3. In a dense sampling scheme [9] sequence elements are also encoded with all possible binary strings constituting non-uniquely decodable but very dense code. To access the elements, two types of pointers are stored: absolute pointers to every $O(\log n)$-th element and relative pointers to the rest. If the integers distribution is not too positively skewed, the constant time direct access is possible at the cost of less extra space to the main bitstream compared with the previous approach.

4. Split binary representations of integers into chunks of a fixed length. Construct the first bitstream from all first chunks, the second bitstream from all second chunks, etc. Then the i-th element of any bitstream can be accessed on the RAM model in constant time as an array element. However, since the number of chunks in the bit representation of an integer is variable, an extra bitmap B_j is used together with the j-th bitstream. $B_j[i] = 1$ iff the j-th chunk of some integer is stored in the i-th element of the j-th bitstream, and this integer has the $(j + 1)$-th chunk. Thus, to reconstruct the i-th integer, we directly get its first chunk from the first bitstream and then check the $B_1[i]$. If it is 0, we've got the result; otherwise, we compute the $rank(B_1, i)$ to get the position of the integer's second chunk in the second bitstream and so on. This technique is investigated in [6] and called the *Directly Addressable variable-length Codes* (DAC). Also, it is known as *Reordered Vbyte* since it generalizes the Vbyte code idea [17]. Extraction of a random codeword requires at most $\lceil \frac{\log S}{b} \rceil$ rank operations, where S is the largest integer in the sequence and b is the chunk size. The space overhead for all rank data structures is $O(\frac{n \log \log n}{\log n})$.

5. Use the variable-length codes with delimiters, for example, Fibonacci codes [3], as proposed in another schema from [10]. The classical constant-time and $o(n)$-space 'select' algorithm, proposed by Clark [7], can be extended to this case.

Our method of fast direct access to elements of a compressed integer sequence is based on the use of variable-length Reverse Multi-delimiter (RMD) Codes introduced in [19]. These codes are self-delimited, which allows us to avoid using an auxiliary bit-vector indicating codeword boundaries. As well as for Fibonacci codes, it is easy to extend the classical Jacobson's [12] and Clark's [7] results regarding constant time

rank and select on bit-vectors with $o(n)$ extra space to the case of RMD codes. However, the select operation in [7] requires $\frac{3n}{\lceil \lg \lg n \rceil} + O(n^{\frac{1}{2}} \lg n \lg \lg n)$ bits of extra space. On the real-world data, for bit sequences of length up to 4Gb, this value exceeds 60% of the code itself, which we consider too big. To reduce space overhead, we combined approaches 1 and 5. Namely, we store the absolute position of each 'Level 1' block of RMD-codewords, then use a kind of linear approximation to get the position of a smaller 'Level 2' block (a similar idea has been implemented in [5] and [13]) and search the codeword inside this block sequentially. Properties of RMD codes allow us to perform this search and decoding significantly faster and use smaller space than Elias or Fibonacci codes.

The variable-length data compression RMD codes and their properties are briefly discussed in Section 2. The main method of integer retrieval is presented in Section 3. Its space complexity is estimated in Section 4. In Section 5, we discuss experiments in compression and extracting elements from an integer sequence generated in the process of English text compression. As shown in [19], RMD codes outperform Fibonacci codes in natural language text compression ratio and decoding speed. That is why we did not include in experiments the Fibonacci-based approach [10]. Also, in practical English text compression, DAC outperforms the dense sampling both by space and time, as shown in [6]. Thus, we also exclude the dense sampling scheme from our experimental set, where the SDC encoding, DAC, and our new method remain. Also, as the basis of comparison, we experimented with naive approach 1 implemented using the Elias delta code.

2 Reverse Multi-Delimiter Codes

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

Definition 1 *The reverse multi-delimiter code $R_{m_1,\ldots,m_t}$ consists of all the words of the form $01^{m_i}, i = 1, \ldots, t$ and all other words that meet the following requirements:*

(i) for any $m_i \in \mathcal{M}$ a word does not end with the sequence 01^{m_i};
(ii) for any $m_i \in \mathcal{M}$ a word can contain the sequence $01^{m_i}0$ only as a prefix;
(iii) a word starts with the prefix $01^{m_i}0$ for some $m_i \in \mathcal{M}$.

The given definition implies that code delimiters in $R_{m_1,\ldots,m_t}$ are sequences of the form $01^{m_i}0$. However, the code also contains shorter words of the form 01^{m_i} that form a delimiter together with the first zero of the next codeword. The set of codewords of the code $R_{2,4,5}$ of length not greater than 7 is shown in Fig. 1, where codewords of the second type are highlighted in grey.

In the sequel, we refer to the "infinite" versions of RMD codes, notably $R_{2-\infty}$ and $R_{2,4-\infty}$, as they demonstrate the best compression ratio. They use all delimiters containing $2, 3, \ldots$ or $2, 4, 5, \ldots$ ones, respectively. However, in practice, limiting the lengths of delimiters by some relatively large number is enough, defined by the maximal codeword length for a specific application.

Any reverse multi-delimiter code $R_{m_1,\ldots,m_t}$ contains the same number of codewords of a given length as the "direct" multi-delimiter code $D_{m_1,\ldots,m_t}$ discussed in [1]. Thus, reverse MD-codes possess all properties of MD-codes, such as completeness and universality, as well as their asymptotic densities. For MD codes, these properties were proven in [1]. Also, we refer to [1] for the analysis of asymptotic densities and quantities of short codewords in multi-delimiter codes.

L=3	L=4	L=5	L=6	L=7
011	0110	01100	011000	0110000
		01101	011010	0110100
		01111	011110	0111100
			011001	0110010
			011111	0111110
				0110001
				0110101
				0111101
				0110111

Figure 1. $R_{2,4,5}$ codewords of length ≤ 7

The main advantage of RMD codes over the "direct" multi-delimiter code is that for RMD codes, there exists a simple monotonic encoding mapping from the set of natural numbers to the set of codewords. Also, the reverse decoding mapping can be built and represented as a finite automaton with a small number of states recognizing codeword bits from left to right and calculating the index of a codeword in the ordered codeword set. A simple decoding principle is a key point since, as mentioned above, we must decode the RMD-encoded numbers quickly to retrieve them from a compressed sequence efficiently.

The decoding automata for codes $R_{2-\infty}$, $R_{3-\infty}$, and $R_{2,4-\infty}$ are given and discussed in [20]. However, they process a text bit-by-bit, which is quite slow. The main idea of a fast decoding algorithm is a "quantification" of a decoding automaton so that it reads bytes of a code and produces the corresponding output numbers. Such an algorithm has been proposed in [19] and its improved version in [2]. In this paper, we search for a codeword and, after it is found, decode it. Thus, we need a simplified version of the fast decoding method intended to decode only one codeword. This simplification of the decoding algorithm from [2] we use in Algorithm 2 described in the next Section, which is a part of general integer retrieving Algorithm 1.

3 Integer Retrieving Technique

Below we describe Algorithm 1 calculating the value of the element with a given index in the integer sequence encoded with RMD codes. From now on, we consider codes $R_{2,x}$ having the shortest delimiter 0110; they are the best representatives of an RMD family in natural language text compression. Although an RMD-encoded sequence is a bitstream, we operate on a byte level to make the method fast, getting all required bit-level data from lookup tables. The algorithm idea and notations are the following.

- Split the encoded bitstream into level 1 blocks containing L_1 codewords each. Store the number of the first byte of each block in the array $L1byte$.
- Split each L1-block into level 2 blocks containing L_2 codewords each. Let $L2Length[i]$ be the average length in bytes of an L2-block in the i-th L1-block. Also, store the array $\Delta_b[i][j] = b_{ij} - \lfloor j \cdot L2Length[i] \rfloor$, where b_{ij} is the position of the first byte of the j-th level 2 block relative to i-th level 1 block. Then we can calculate the number of the leftmost byte of the L2-block by the formula $L1byte[i] + j \cdot L2Length[i] + \Delta_b[i][j]$. As shown in [19], no more than three codewords of an RMD-code $R_{2,x}$ can start in one byte. Therefore, we also need the

2-bit value $\Delta_c[i][j]$ indicating which codeword inside the byte is the first codeword of the j-th level 2 block.

– When we know exactly where the L2 block starts, seek the codeword inside the block, processing it byte-by-byte. It can be done from the beginning of the block in the left-to-right direction or from the beginning of the next block right-to-left, depending on which way is shorter.

– When we find the leftmost byte of a required codeword, decode it using a fast byte-aligned decoding technique, e.g., as discussed in [2].

Algorithm 1: Decoding an element of the RMD-encoded sequence

input : The index t of the element.
output: The value of the t-th element, *out*.

1 $n_1 \leftarrow t \operatorname{div} L_1$; `// Number of the L1-block`
2 $e_1 \leftarrow t \bmod L_1$; `// Number of the element inside the L1-block`
3 $n_2 \leftarrow e_1 \operatorname{div} L_2$; `// Number of the L2-block inside the L1-block`
`// Number of the codeword inside the byte-aligned L2-block`
4 $e_2 \leftarrow e_1 \bmod L_2 + \Delta_c[n_1][n_2]$;
`// Find the byte where the codeword starts`
5 **if** $e_2 < L_2/2$ **then**
`// Number of the byte where the L2-block starts`
6 $i \leftarrow L1byte[n_1] + L2Length[n_1] \cdot n_2 + \Delta_b[n_1][n_2]$;
7 $e \leftarrow 0$;
8 **else**
9 $n_2 \leftarrow n_2 + 1$; `// Search from the next L2-block right to left`
10 $i \leftarrow L1byte[n_1] + L2Length[n_1] \cdot n_2 + \Delta_b[n_1][n_2] - 1$;
11 $e \leftarrow L_2 + \Delta_c[n_1][n_2 - 1] - \Delta_c[n_1][n_2]$;
12 **while** $e \geq e_2$ **do**
13 $e \leftarrow e - Words(i)$;
14 $i \leftarrow i - 1$;
15 **while** $e < e_2$ **do**
16 $e \leftarrow e + Words(i)$;
17 $i \leftarrow i + 1$;
`// Starting from the` $(i-1)$`-th byte of a code, skip` $e - e_2$ `codewords, and decode the next codeword`
18 $out \leftarrow Decode\_number(i - 1, e - e_2)$;

Note that, in general, the j-th L2-block position can be approximated by the formula $kj + b$, where parameters k and b are calculated with the ordinary least squares technique. However, we intentionally fix b as $L1byte[i]$ since experiments show that this approach is a bit less space-consuming.

Let us explain in detail how Alg. 1 works. Given the index t of a required element (codeword), in lines 1 and 2, we get the number of the containing L1-block, n_1, and the relative number of the element inside this L1-block, e_1. Consider the L2 block containing the required codeword. Its number n_2 relative to the containing L1 block is calculated in line 3 of Alg. 1.

We search a codeword inside the L2-block byte-by-byte. However, an L2 block may start not from the first codeword in the byte but from the second or third. Then we extend the discussed L2-block to the left by including all full codewords

from its first byte and call this extended block a *byte-aligned L2-block*. We store the difference between the codeword positions in the byte-aligned and original L2-blocks in the array Δ_c, and get the number of the required codeword inside the byte-aligned L2-block in line 4 of Alg. 1 denoting it by e_2.

In line 5, we analyze if the codeword is in the left half of the L2 block. If so, using a kind of linear approximation, in line 6, we get the number of the byte where this L2 block starts. Then in lines 7 and 15-17, the number i of the first byte of a required codeword is calculated by sequential processing bytes of L2-block from left to right. The function $Words(i)$ returns the number of codewords starting in the i-th byte of a code. It is summed up in the variable e until it becomes no less than the required value e_2. The correspondence between the estimated and actual beginning of an L2 block, as well as values from Δ_b and Δ_c arrays, are shown in Fig. 2.

If the required codeword is in the right half of the L2 block, the right-to-left search from the beginning of the next L2 block would be faster (lines 9-14). In this case, we increment the number of the L2 block (line 9), get the number of the byte before its beginning (line 10), and the number of codewords the right-to-left search to be started from (line 11). After the search finishes, the value e may become too small, and a few iterations of the left-to-right search may be needed (lines 15-17).

In both cases, after line 17, the index $i-1$ points to the byte where the required codeword starts and it is the $(e-e_2)$-th codeword in this byte if we count from 0 and the right edge of the byte. Thus, we skip $e-e_2$ codewords from the right edge of the byte and return the result of decoding the next codeword to the left of them (line 18). This is done in the function *Decode_number*, which is described in Alg. 2. Its idea resembles the fast byte-aligned decoding method presented in [2].

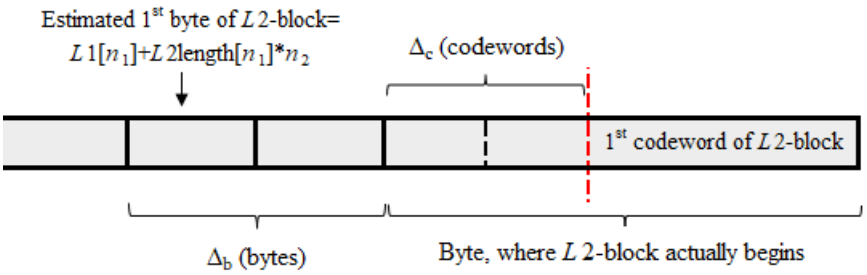


Figure 2. Calculating the position of an L2-block

Codewords for real-world texts are at most 35-40 bits. In line 1 of Alg. 2, we load into the variable *val*64 8 bytes of a code containing the whole codeword to be decoded. In line 2, this codeword is shifted to the right edge of a 64-bit machine word. Then we split the bit representation of *val*64 into chunks and process it chunk-by-chunk, accumulating the resultant value in the variable *out*. Alg. 2 shows this process on a little-endian machine, where bytes of a value are loaded from memory to a processor register in the reverse order. Thus, the bits inside bytes of a code should also be put in the reverse order.

The chunks of an RMD-encoded bitstream are recognized by quantified finite automatons, as described in [19]. The result of the chunk decoding depends on the chunk's content, the number of the chunk, and the state of the decoding automaton at the beginning of chunk processing. The last two parameters are stored in the variable *ptr* used in lines 6 and 7. In line 6, we shift its value by the chunk bit size to the left and add the chunk content to it. This way, we obtain the value v containing the full

information to decode the current chunk. Then, in line 7, we get the value *ptr* for the next chunk and increment the current result by the value $Out[v]$ in line 8. At last, in line 9, the value $val64$ is shifted by the chunk size to the right to process the next chunk. The loop repeats until the flag $N[v]$ signals that we met a delimiter and the codeword has been decoded.

Algorithm 2: Function $Decode\_number(i, s)$ - decoding the $(s+1)$-th codeword starting from the right edge of the i-th byte of a code

input : i - the number of the byte of a code; $s \leq 2$ - the number of codewords to be skipped.
output: The decoded number, *out*.

1 $val64 \leftarrow Code[i...i+7]$; `// Read 8 bytes of a code`
2 $val64 \leftarrow Align(val64, s)$; `// Align the` $(s+1)$`-th codeword to the right edge of a 64-bit word`
3 $ptr \leftarrow 0$; $out \leftarrow 0$;
4 $chunk\_mask \leftarrow 2^{chunk\_size} - 1$;
5 **repeat**
6 $\quad v \leftarrow ptr << chunk\_size + val64 \& chunk\_mask$;
7 $\quad ptr \leftarrow Pointers[v]$;
8 $\quad out \leftarrow out + Out[v]$;
9 $\quad val64 \leftarrow val64 >> chunk\_size$;
10 **until** $N[v] = 0$;

Example 1. Assume $L_1 = 2^{10} = 1024$, $L_2 = 2^5 = 32$ and retrieve the 1060-th element from the compressed integer sequence encoded by $R_{2,4-\infty}$. At first, we get $n_1 = 1060 \operatorname{div} 1024 = 1$ (the number of the L1-block), $e_1 = 1060 \bmod 1024 = 36$ (the number of the byte in the L1-block), and $n_2 = 36 \operatorname{div} 32 = 1$ (the number of the L2-block). Then, assume the first L1-block occupies 1200 full bytes of an RMD-bitstream, i.e., $L1byte[1] = 1200$, and the average length of an L2-block inside the second L1-block is 40 bytes, i.e., $L2Length[1] = 40$. However, the actual byte length of the first L2 block inside the second L1 block can be different, say 39. Then $\Delta_b[1][1] = 39 - 40 = -1$. E.g., this block can occupy some rightmost bits of the 1200-th byte, full bytes $1201 - 1238$, and five leftmost bits of the byte 1239, as shown in Fig. 3.

Now, assume the binary representation of the 1239-th byte is 00011011. The leftmost 2 bits represent the ending of the 1055-th codeword and, together with the 1056-th codeword 011, belong to the first L2 block inside the second L1 block. Then the last 3 bits 011 are the starting bits of the next L2 block, which interest us. Since this L2-block starts from the second codeword in the byte 1239, $\Delta_c[1][1] = 1$, i.e., we should skip one full codeword in the byte 1239 to get to the beginning of the L2-block. Then $e_2 = e_1 \bmod L_2 + \Delta_c[n_1][n_2] = (36 \bmod 32) + 1 = 5$ is the number of the target codeword if we start counting from the first full codeword in the byte 1239.

Since $e_2 < L_2/2$, we execute lines 6 and 7 of Algorithm 1: $i \leftarrow L1byte[n_1] + L2Length[n_1] \cdot n_2 + \Delta_b[n_1][n_2] = 1200 + 40 \cdot 1 - 1 = 1239$, $e = 0$. Then we execute iterations of the loop in lines $15-17$ assuming the bitstream is shown in Fig. 3, where even bytes are highlighted with grey.

1. $Words(1239) = 2$, $e = 2$, $i = 1240$;
2. $Words(1240) = 2$, $e = 4$, $i = 1241$;
3. $Words(1241) = 0$, $e = 4$, $i = 1242$;
4. $Words(1242) = 2$, $e = 6$, $i = 1243$;

At last, we call the function $Decode\_number(i-1, e-e_2) = Decode\_number(1242, 1)$. It skips one codeword (1061-th) from the right edge of the byte 1242, aligns the 1060-th codeword 01101 with the right edge of a 64-bit machine word, and returns its decoded value, i.e., 3 (see Fig. 1).

Codeword		1056	1057	1058	1059	1060	1061
Bitstream	0 0	0 1 1	0 1 1 0	0 1 1 1 1	0 1 1 0 1 1 1 0 1 0	0 1 1 0 1	0 1 1 0 1 0 0 0
Byte	1239			1240	1241	1242	1243
L2-block	1st			2nd			

Figure 3. Fragment of an $R_{2,4-\infty}$-bitstream

4 Space complexity

Now, let us estimate the space required by Algorithms 1 and 2, apart from the size of an RMD-encoded file itself. Assume the encoded sequence fits into 4GB, and n is the number of elements in it. Then 4 bytes are enough to store an $L1byte$ array element, or $4n/L_1$ bytes for the whole $L1byte$ array. If we reserve 4 bytes to store the linear approximation ratio $L2Length[n_1]$, the array $L2Length$ will occupy the same space. As mentioned above, no more than three codewords of R_{2-x} code can start in one byte. Therefore, 2 bits are enough for an element of the array Δ_c, or $n/4L_2$ bytes for the whole array.

The function $Word(i)$ uses the lookup table consisting of the number of codewords starting in the byte $code[i]$. Analyzing the byte itself, it is not possible to determine how many codewords start in it. For example, if the byte ends with a 0 bit, it can be either the first bit of the next codeword or a continuation of the current one. However, to answer this question for the code $R_{2-\infty}$, we need to analyze only 2 bits following the current byte and 4 bits for the code $R_{2,4-\infty}$. Namely, the sequences 0|11 in $R_{2-\infty}$, and 0|110 or 0|1111 in $R_{2,4-\infty}$ begin the new codeword, while all other bit combinations after the ending 0 mean that the current codeword continues. For ending 1, we need to analyze even fewer extra bits. Thus, the index of the mentioned lookup table can be a 12-bit integer, and the table consists of 4096 2-bit elements (numbers between 0 and 3). To decrease the number of bit-level operations, we reserve 1 byte for each element, and 4096 bytes will be enough to store the whole table.

To estimate the space complexity of Alg. 2, we should calculate the maximal value of the variable v. If a codeword consists of not more than max_len bits, it contains not more than $c = \lceil \frac{max\_len}{chunk\_size} \rceil$ chunks. As mentioned above, the decoding result depends on the number of the chunk, its content, and the state of the decoding automaton. Thus, $v \leq c \cdot n\_states \cdot 2^{chunk\_size}$, where n_states is the number of states of the decoding automaton (3 for $R_{2-\infty}$ and 5 for $R_{2,4-\infty}$ [20]). Assuming the realistic codeword length does not exceed 35-40 bits and $chunk\_size = 7$, which gives the lowest decoding time in experiments, we get $4000 - 5000$ as an upper bound for v. Each element of the arrays $Pointers$ and N takes 1 byte, while $Out[v]$ requires 4 bytes. Therefore, the total size of the lookup tables for Algorithm 2 does not exceed $25 - 30$KB.

The array Δ_b occupies the biggest space. These delta values can vary in different ranges for different L1 blocks. That is why we allocate the different number of bits

for elements of different $\Delta_b[i]$ subarrays and store these bit lengths in the special array Bit_ranges. We store all Δ_b values for an $L1$-block as a bitstream, keeping a pointer to it in the array Δptr. One byte is enough for a Bit_ranges array element and 4 bytes for a pointer. Of course, this approach involves some extra bit operations. Nonetheless, it allows us to save $40-50\%$ space occupied by data structures needed for direct access and does not affect the overall time much because the biggest time consumption is accounted for the loops in lines $12-17$ of Algorithm 1. Also, using smaller data structures accelerates an algorithm thanks to fewer cache mismatches.

In total, we need $25-30$KB of memory for Alg. 2, $13n/L_1$ bytes for level 1 structures, $n/4L_2$ bytes for the array Δ_c, and a variable space for the array Δ_b. As shown in experiments described in the next section, the optimal value L_1 can be between 2^{14} and 2^{17}, while L_2 is between 2^6 and 2^8. This makes the space for $L1$-structures and Δ_c almost insignificant, about tenths of 1 percent of a code itself, while Δ_b occupies about $1-3\%$ of the code size.

5 Experiments

We tested our solution on integer sequences obtained by applying two known natural language compression schemes to 200 MB English text from Pizza&Chili corpus.

- In the first scheme, words of the text are considered as alphabet symbols. In the dictionary, they are arranged in the order of descending frequencies. Then we replace words in the text with their indices in the dictionary. The text consists of 37,003,242 words and has the entropy H0 52,805 KB.
- The second scheme was proposed by Ferragina and Venturini [9] to compress a sequence of n characters to its high-order entropy so that a $O(\log n)$-bit substring can be decoded in constant time. The text is split into blocks of $\frac{1}{2}\log n$ bits, which are sorted by frequency and encoded as in the first scheme. In our test $n = 209,715,202$, which implies $\frac{1}{2}\log n \approx 14$. To reduce the volume of bit-level operations, we rounded the block size to 2 bytes. Since the alphabet is constructed of pairs of characters, the compressed text size should be compared with the entropy H1, which is 106,754 KB.

We measured the element extraction time and the space occupied by the encoded text together with auxiliary structures. The time was averaged over 100 million extractions of a random integer sequence element. To reduce the number of divisions in Algorithm 1, we chose the size both of L1 and L2-blocks as powers of 2: $L_1 = 2^{l_1}$ and $L_2 = 2^{l_2}$. The optimal chunk size in Algorithm 2 was determined experimentally and equals 7 bits for all tests.

Parameters l_1 and l_2 constitute a space/time trade-off shown in Fig. 4. Parameter l_2 has more impact because it defines the average number of iterations of the loops in Algorithm 1 and the size of arrays Δ_c and Δ_b. As mentioned above, the most prominent values of l_2 for our data are between 6 and 8. When l_2 decreases, arrays Δ_c and Δ_b become bigger, but loops have fewer iterations. This speeds up the algorithm by the cost of space until arrays become too big to fit into the L2 or L3 cache, which causes many cache mismatches. The latter situation is illustrated in Fig. 4b, where the element extraction for $l_2 = 6$ executes longer and requires more space than for $l_2 = 7$.

Two competitive solutions discussed in the Introduction were tested for the comparison: the Directly Addressable variable-length Codes (DAC) [6] and the Simple

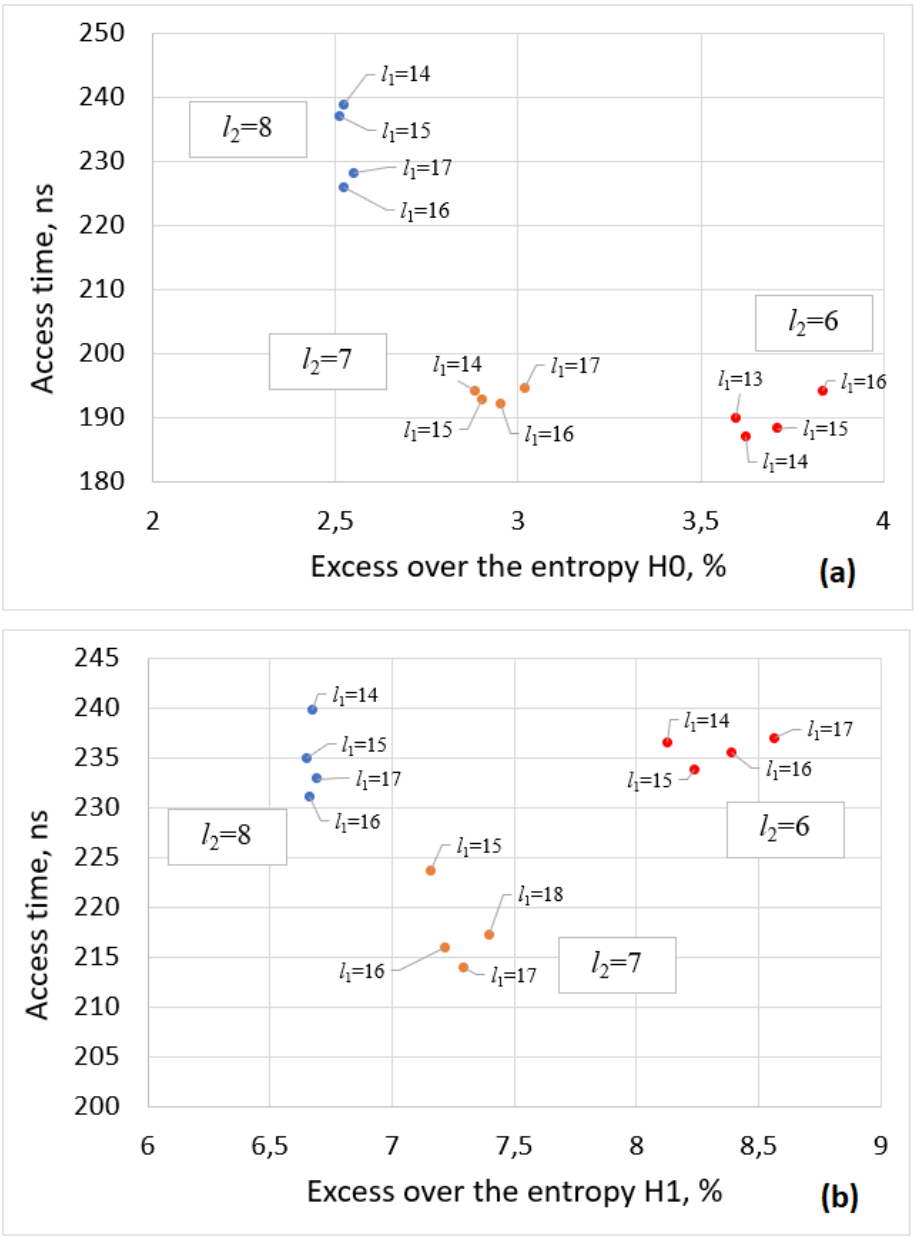


Figure 4. Element extraction from the RMD-encoded integer sequence: (a) word-based alphabet, (b) character-based alphabet

Dense Coding (SDC) [10]. The DAC relies on a 'black-box' rank operation for a bit-sequence, while random access via the SDC structure requires a select. The comparison of the best recent approaches to computing rank and select for binary sequences is given in [13] and [15]. In both sources, the two fastest methods to compute rank appear to be the Rank9-V1 [18] and its variation, the so-called IL (interleaving) [11], where the original bit-vector is interleaved with rank data. They also require relatively small space overhead (usually the Rank9-V1 uses somewhat bigger space than the IL).

As reported in [13], very fast 'select' algorithms operating at approximately the same speed are provided with SD [14], MCL, RSAA [13], and LA [5] structures. However, the space complexity highly depends on the percentage of ones in a bit-

vector. In the first scheme of our test set, it is about 13% and about 19% in the second scheme. For bit-vectors with a low percentage of ones, SD and LA select structures occupy more attractive positions on the space/time plane than the other two mentioned solutions. To implement them, we stored the simple dense code of the integer sequence as-is, while the auxiliary binary sequence needed for constant time random access is given in the form of a compressed LA- or SD-vector.

Also, we tested the naive method mentioned as "approach 1" in Introduction. The integer sequences were encoded with the Elias δ-code, and the position of each s-th codeword was sampled. To retrieve an element, we perform a sequential search from the sampled position.

The compressed file sizes, together with auxiliary data structures as well as average integer extraction times, are shown in Table 1 and in Fig. 5 (except for the naive approach in the Figure as it goes beyond the scale). The excess over the H0 entropy for the word alphabet and over the H1 entropy for the character-based one is shown in percentage. All data is stored in RAM. To build our and other solutions, we used the g++ compiler v9.4.0 with the -O3 optimization flag. We got IL and SD implementations from the SDSL library [16] and LA implementation from [4]. Tests have been run on a computer with an AMD Athlon 3000G processor, 32 KB of L1 cache, 512 KB of L2 cache, 4 MB of L3 cache, 16 GB RAM, and OS Ubuntu 20.04 LTS. The source code can be downloaded from [21].

We tested methods with different parameters representing different points in the space/time trade-off. For the first scheme, the code $R_{2,4-\infty}$ gives the best compression ratio, while for the second scheme, it is $R_{2-\infty}$. In each case, we show two pairs of parameters l_1, l_2 giving the best time and the best space, as well as space or time optimal LA parameters for the SDC+LA scheme. The DAC code is parameterized by the bit size of a chunk, b. The extraction time for the Elias δ-code depends on the sampling interval s.

As seen, the Elias codes are obviously space inefficient. Even the code itself exceeds the entropy H0 by 34% for the word-based alphabet and by 38% for the character-based. However, the extraction time can be quite low if the sampling rate is high. In fact, this way, we approach the uncompressed integer sequence.

Our data structure based on RMD codes is significantly more compact than all competitive solutions both for word-based and character-based alphabets. Also, our direct access method is faster than SDC in combination with the space-optimal LA or SD on both alphabets. However, the SDC+LA scheme may become faster at the cost of extra space ($bpc = 7$).

We tested DAC with different bit sizes of a chunk: $b = 4$ or $b = 8$. This parameter represents a space/time trade-off: operating whole bytes ($b = 8$) is much faster but requires much more space than for $b = 4$. The value $b = 2$ appears to be inefficient and is not shown in the tables.

Our method is better than DAC-4 in space and time on the word-based alphabet: $3-10\%$ shorter and $1-25\%$ faster, depending on parameters l_1, l_2 and DAC variations. Enlarging the chunk size to 8 bits makes DAC $2-2.7$ times faster than RMD by the cost of space (it becomes $15-19\%$ larger).

On the character-based alphabet, the encoded text becomes larger, and the size of the arrays $L1Byte$, $L2Length$, Δ_c, and Δ_b also grows, causing more cache mismatches and slowing down our algorithm. At the same time, encoded integers are smaller, requiring less number of streams in DAC. As a result, on the character-based alphabet, our method becomes slower even than DAC-4. However, its space outperformance

Integer sequence generated from the word-based alphabet

Algorithm	Parameters	Size, KB	Time, ns
Elias δ	$s = 4$	107,835 (104.9%)	127
	$s = 512$	71,121 (34.69%)	1999
RMD, $R_{2,4-\infty}$	$l_1 = 14$, $l_2 = 6$	54,719 (3.62%)	187
	$l_1 = 16$, $l_2 = 8$	54,138 (2.52%)	226
DAC	$b = 8$, IL	63,163 (19.61%)	86
	$b = 8$, Rank9-v1	64,387 (21.93%)	85
	$b = 4$, IL	57,595 (9.07%)	233
	$b = 4$, Rank9-v1	59,628 (12.92%)	228
SDC+LA	$bpc = 7$	67,634 (28.08%)	137
	v-opt	65,500 (24.04%)	253
SDC+SD		64,164 (21.51%)	300

Integer sequence generated from the character-based alphabet

Elias δ	$s = 4$	252,732 (136.7%)	135
	$s = 512$	148,694 (39.3%)	1932
RMD, $R_{2-\infty}$	$l_1 = 17$, $l_2 = 7$	114,538 (7.29%)	214
	$l_1 = 16$, $l_2 = 8$	113,863 (6.65%)	231
DAC	$b = 8$, IL	136,444 (27.81%)	47
	$b = 8$, Rank9-v1	135,935 (27.33%)	53
	$b = 4$, IL	124,133 (16.28%)	175
	$b = 4$, Rank9-v1	129,734 (21.5%)	150
SDC+LA	$bpc = 7$	159,914 (49.8%)	146
	v-opt	143,696 (34.61%)	323
SDC+SD		143,301 (34.23%)	433

Table 1. Experiments on integer compression and extraction

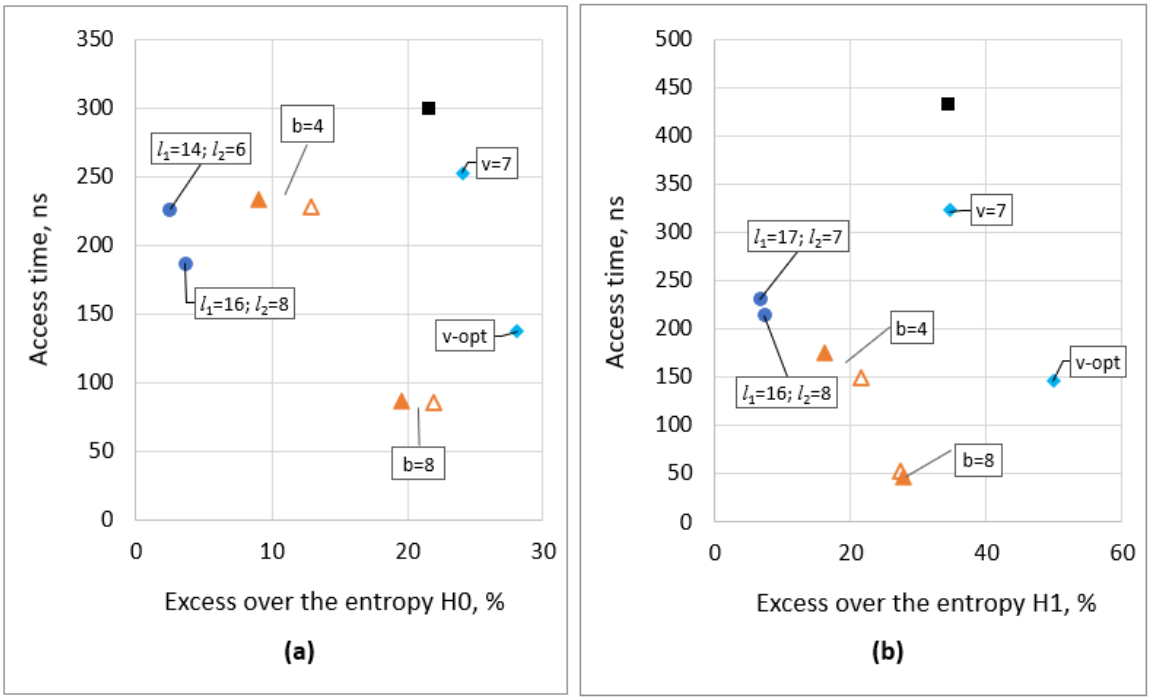


Figure 5. Experiments with different approaches: (a) word-based alphabet, (b) character-based alphabet

increases (the space optimal RMD structure is shorter than DAC by $9 - 14\%$ for $b = 4$ and $19 - 20\%$ for $b = 8$).

6 Conclusion

We presented a fast method of extracting an element of an unordered integer sequence compressed using the Reverse Multi-Delimiter codes. By exploiting the recently developed technique of linear approximation of a codeword block position and properties of the RMD codes, we achieved a very good compression ratio for integer sequences taken from a frequency-based compression of the 200MB English text. Together with all data structures required for fast direct access, the size of the compressed file exceeds the zero-order entropy on the word-based alphabet by $2.5 - 3.5\%$ and the first-order entropy on the character-based alphabet by $6.5 - 7.5\%$. At the same time, our method provides a decent speed of element extraction, being the fastest among competitive solutions that compress the text with a ratio exceeding the entropy by less than 15%.

References

1. A. Anisimov and I. Zavadskyi: *Variable-length prefix codes with multiple delimiters.* IEEE Transactions Information Theory, 63(5) 2017, pp. 2885–2895.
2. A. Anisimov, I. Zavadskyi, and T. Chudakov: *Practical word-based text compression using the reverse multi-delimiter codes*, in Information Technology and Implementation (ITI-2022), CEUR Workshop Proc., 2022, pp. 175—183.
3. A. Apostolico and A. S. Fraenkel: *Robust transmission of unbounded strings using Fibonacci representations.* IEEE Transactions Information Theory, 33 1987, pp. 238–245.
4. A. Boffa, P. Ferragina, and G. Vinciguerra: *Learned-compressed-rank-select.* https://github.com/aboffa/Learned-Compressed-Rank-Select-TALG22.
5. A. Boffa, P. Ferragina, and G. Vinciguerra: *A learned approach to design compressed rank/select data structures.* ACM Transactions on Algorithms, 2022.
6. N. R. Brisaboa, S. Ladra, and G. Navarro: *Directly addressable variable-length codes*, in String Processing and Information Retrieval, J. Karlgren, J. Tarhio, and H. Hyyrö, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 122–130.
7. D. R. Clark: *Compact Pat Trees*, PhD thesis, University of Waterloo, 1996.
8. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Transactions Information Theory, 21 1975, pp. 194–203.
9. P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds*, in Proc. 18th SODA, 2007, pp. 690—696.
10. K. Fredriksson and F. Nikitin: *Simple compression code supporting random access and fast string matching*, in International Workshop on Experimental and Efficient Algorithms, 2007, pp. 203—216.
11. S. Gog and M. Petri: *Optimized succinct data structures for massive data.* Software: Practice and Experience, 44 2014, pp. 1287 – 1314.
12. G. Jacobson: *Succinct Static Data Structures. Ph.D. Thesis*, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.
13. O. Kulekci: *Counting with prediction: Rank and select queries with adjusted anchoring*, in 2022 Data Compression Conference, 2022, pp. 409–418.
14. D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, Louisiana, USA, 2007, Society for Industrial and Applied Mathematics, pp. 60–70.
15. G. Pibiri and S. Kanda: *Rank/select queries over mutable bitmaps.* Information Systems, 99 2021, p. 101756.
16. *Succinct data structure library*: https://github.com/simongog/sdsl/.
17. L. Thiel and H. Heaps: *Program design for retrospective searches on large data bases.* Information Storage and Retrieval, 8(1) 1972, p. 1–20.
18. S. Vigna: *Broadword implementation of rank/select queries*, in Proceedings of the International Workshop on Experimental and Efficient Algorithms, 2008, pp. 154—-168.

19. I. Zavadskyi and A. Anisimov: *Reverse multi-delimiter compression codes*, in 2020 Data Compression Conference, 2020, pp. 173–182.
20. I. Zavadskyi and V. Zavadska: *Reverse multi-delimiter codes in English and Ukrainian natural language text compression*, in CEUR Workshop Proc., 2022, pp. 211—219.
21. I. O. Zavadskyi: *Direct access to RMD-encoded sequence elements. Implementation in C programming language.* https://github.com/zavadsky/DirectAccess.

Selective Weighted Adaptive Coding

Yoav Gross[1], Shmuel T. Klein[2], Elina Opalinsky[1], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
yodgimmel@gmail.com, elinao@ariel.ac.il, shapird@g.ariel.ac.il
[2] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. Motivated by improving the processing time of the weighted compressor introduced recently, while only marginally affecting its compression performance, we propose a selective-weighted method that restricts the model based positions and/or the times the model gets updated. That is, the adaptive statistical weight oriented method combines the ability to assign higher priority to symbols that are closer to those being encoded, while extending the model to the probability distribution of symbols that are not necessarily located in consecutive positions. Several variants for selecting the representative positions for determining the model are suggested, and all are empirically shown to fulfill both objectives for small skips.

1 Introduction

The most noticeable advantage of *adaptive* compression techniques is their ability to adjust the encoding to the local changes in the input file. Static methods, on the other hand, such as Huffman coding [9] or Elias' universal codes [3], use the same set of codewords throughout the entire file, and the main way to adapt the encoding to the given input is via a preprocessing stage.

Statistical dynamic methods, like dynamic Huffman [14] or adaptive arithmetic coding [15], attempt to adjust to the local probability distribution by collecting statistics of the already processed portion of the file. Traditional statistical adaptive compression algorithms encode the next to be processed character according to the current model, and then update the model by incrementing the frequency of the currently read symbol. These algorithms are usually heuristics that are based on the following two strategies:

- all positions in the input file are treated equally, that is, with no consideration for their distance from the currently processed symbol;
- the distribution of the elements in *consecutive positions* in a "sliding window", just preceding the current processed character, is used to estimate the distribution of the elements still to come.

In this paper we suggest to use different policies to determine the varying probability distributions, first, by giving higher priority to symbols that are closer to those being encoded, second, by using the frequency counts of symbols that are not necessarily located in consecutive positions. The goal of our first policy is to improve the compression efficiency, while the goal of the second is to enhance the processing times while only marginally affecting the compression performance. We refer to this new method as selective-weighted.

Unlike the uniform treatment of symbols in different locations of the file, a new approach is taken in the *weighted* dynamic compression method suggested in [5], which assigns higher priority to closer to be encoded symbols by means of an increasing

Yoav Gross, Shmuel T. Klein, Elina Opalinsky, Dana Shapira: Selective Weighted Adaptive Coding, pp. 97–106.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

weight function. The weighted method is especially suited for the encoding of files with locally skewed distributions. We distinguish between two weighted schemes, a *forward weighted* coding [7], and a *backward weighted* coding [5]. The weights assigned to the positions by the former scheme are generated by a non increasing function f, and the weight for each symbol σ is the sum of the values of f over all the positions where σ occurs in the portion of the input file that is still to be encoded. The latter is, in fact, a heuristic defined similarly to the forward-weighted distribution, but it is calculated over positions that have already been processed. As a result, in the forward approach, there is a need to transmit the exact character frequencies to the decoder, for example in a header, while for the backward approach, these frequencies can be learned incrementally and need not to be given. Empirical results have shown that backward weighted techniques can improve beyond the lower bound given by the entropy for static encoding.

An unweighted forward-looking dynamic algorithm has been suggested in [11]; it uses the true distribution of the characters within the remaining portion of the file by decrementing the frequency counts, rather than incrementing the frequency count of the current processed element as is done in the standard, backward looking, adaptive coding techniques. A bidirectional method, which combines both traditional and forward-looking methods, has then been proposed in [6] and the frequencies of the elements are transmitted progressively, whenever a new symbol is encountered, rather than as a bulk, in a header of the file. The combination of the weighted algorithm with PPM has been studied in [1].

The idea of the backward dynamic methods is based on the assumption that the more data is collected in the already processed portion of the input file, the better can one predict the corresponding probability distribution. However, this assumption is not necessarily true, and the distribution over a *subset* of the elements of the file can at times serve as a good approximation for further encodings [10] as shown in our following experiment.

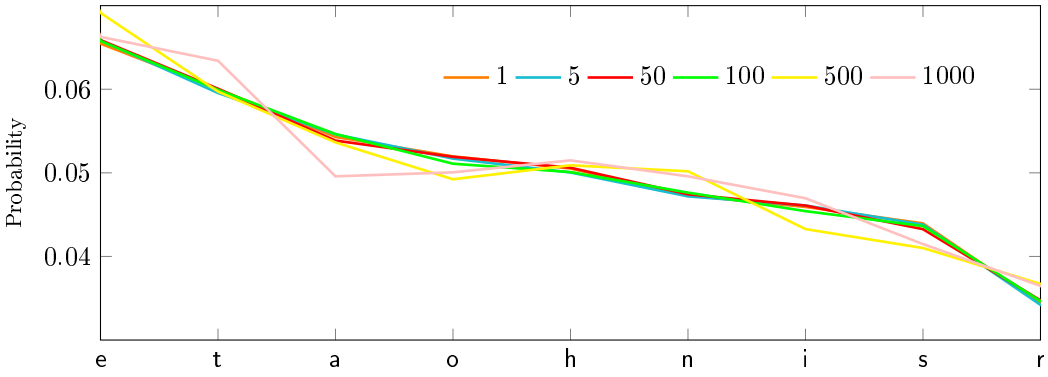


Figure 1. The probabilities of some of the most frequent letters of the file english, in positions with skips with various skip sizes.

We considered the file english, taken from the Pizza & Chili corpus[1], and sorted its letters in non decreasing order of their frequencies. The most frequent letters in this file are, in order, \b, e, t, a, o, h, n, i, s, r, d, ... , where \b stands for blank. Figure 1 depicts on the y-axis the probability of occurrence of the letters that appear on the x-axis, given in this order. The probabilities are based on occurrences of all characters

[1] http://pizzachili.dcc.uchile.cl/

(gap size 1), or on selected subsets of positions, choosing the characters with gap sizes of 5, 50, 100, 500 and 1000 characters. As can be seen, there are obviously fluctuations, but the general forms of the distribution graphs remain similar, even for quite sparse subsets of the inspected positions within the file. The impact on the corresponding codewords lengths in a Huffman or other encoding will even be smaller.

Basing the encoding of the current character on non-consecutive positions in the already processed portion of the file might be straightforward for homogeneous files, but one must carefully avoid some extreme cases. For example, a skip size equal to the fixed length of a line in a text document, or a fixed length field in a database, could result in a completely biased alphabet. We therefore also suggest varying skip values.

Our paper is constructed as follows. In Section 2 we briefly recall the details of the backward weighted compression scheme and propose several selection algorithms. Section 3 presents our experimental results.

2 The Selective Weighted Variants

We concentrate on the *backward weighted* variant, a special case of weighted coding that considers all the positions that have already been processed. Let $T = x_1 \cdots x_n$ be an input file of size n over an alphabet Σ, and assume we have already processed the prefix of T up to position $i-1$ of T, and are about to encode x_i. Given is a function g, $g : [1, n] \longrightarrow \mathbb{R}^+$, which assigns a *non-negative* real number to each position $i \in [1, n]$ within T. A weight $W(g, \sigma, i)$ based on g is defined for each symbol $\sigma \in \Sigma$ and every position i, $i \in [1, n]$, as the sum of the values of the function g for all positions in the prefix $[1, i-1]$ at which σ occurs. Formally,

$$W(g, \sigma, i) \;=\; \sum_{\{j \;\mid\; 1 \le j \le i-1 \;\wedge\; x_j = \sigma\}} g(j).$$

The classic non-weighted adaptive backward compression algorithms, e.g., adaptive arithmetic coding and the one-pass methods based on Huffman coding of the FGK algorithm by Faller [4], Gallager [8] and Knuth [12] and the enhanced algorithm by Vitter [14], are the special case in which g is the constant function $g = \mathbb{1} \equiv g(i) = 1$ for all i.

A simple weighted adaptive coding introduced and named b-2 in [7], divides all the frequencies by 2 at the end of every block of k characters, for some given parameter k, so that the occurrences of characters at the beginning of T contribute to W less than those closer to the current position. Furthermore, all positions within the same block contribute equally to W, and their weights are twice as large as those assigned to the indices in the preceding block. Therefore, the corresponding function g, denoted by $g_{\text{b-2}}$, maintains the equality $g_{\text{b-2}}(i+k) = 2g_{\text{b-2}}(i)$, for each pair of indices i and $i+k$.

Another family of weighted coding schemes, named b-w, is based on the function $g_{\text{b-w}}(i) = (\sqrt[k]{2})^{i-1}$ for $i \ge 1$, for a given parameter k. As for b-2, the function $g_{\text{b-w}}$ still provides a fixed ratio of 2 between blocks but with rather smoother differences at the block borders.

Table 1 compares the classic backward coding, denoted by b-adp, with b-w, on the running example $T = x_1 \cdots x_{12} =$ dbcabcbcaaaa over the alphabet $\Sigma =$ {a, b, c, d}. The table presents for each method the following information:

– $g(i)$: the value of g for position i, of the specific method;

- $W(g, x_i, i)$: the specific weight of the character $\sigma = x_i$ up to (and not including) the column i of the table; that is, the sum of W for those indices $j < i$ at which the character σ occurs, including the initial 1 values.
- $CW[1, i]$: the cumulative W weights for all the characters $\sigma \in \Sigma$ up to (and not including) the column i;
- p_i: the ratio of $W(g, x_i, i)$ and $CW[1, i]$;
- IC: the *Information content*, $-\log p_i$ bits, for each position i.

For b-adp, the values of $g(i)$ are just 1 for every i, whereas for b-w they are $(\sqrt{2})^{i-1}$, taking $k = 2$. The sum of the IC values is a lower bound on, and can be used as an estimate of, the storage requirement by the corresponding method, since it can be closely approximated by arithmetic coding. This sum is 25.588 bit and 24.447 bit for b-adp and b-w, respectively.

	i	1	2	3	4	5	6	7	8	9	10	11	12
	T	d	b	c	a	b	c	b	c	a	a	a	a
b-adp	$g(i)$	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	W	1.000	1.000	1.000	1.000	2.000	2.000	3.000	3.000	2.000	3.000	4.000	5.000
	CW	4.000	5.000	6.000	7.000	8.000	9.000	10.000	11.000	12.000	13.000	14.000	15.000
	p_i	0.250	0.200	0.167	0.143	0.250	0.222	0.300	0.273	0.167	0.231	0.286	0.333
	IC	2.000	2.322	2.585	2.807	2.000	2.170	1.737	1.874	2.585	2.115	1.807	1.585
b-w	$g(i)$	1.000	1.414	2.000	2.828	4.000	5.657	8.000	11.314	16.000	22.627	32.000	45.255
	W	1.000	1.000	1.000	1.000	2.414	3.000	6.414	8.657	3.828	19.828	42.456	74.456
	CW	4.000	5.000	6.414	8.414	11.243	15.243	20.899	28.899	40.213	56.213	78.841	110.841
	p_i	0.250	0.200	0.156	0.119	0.215	0.197	0.307	0.300	0.095	0.353	0.539	0.672
	IC	2.000	2.322	2.681	3.073	2.219	2.345	1.704	1.739	3.393	1.503	0.893	0.574

Table 1. *Classic* b-adp *algorithm vs.* b-w *for the example* $T = x_1 \cdots x_{12} =$ dbcabcbcaaaa.

Motivated by trying to enhance the processing times, even at the price of possibly reduced compression efficiency, we suggest a new encoding scheme based on a periodic selection process, which is controlled by a skip-function f. In a first stage we consider only the special case in which the skip-function is a constant c, and the model gets updated every $f(s) = c$ characters. We distinguish between two different strategies.

(a) The complete-selective algorithm uses the entire input file to compute the probability distributions, as usually done in adaptive methods, but updates the model only every $f(s)$ characters.
(b) The subset-selective algorithm encodes the entire input file T based on the probability distributions of characters appearing at positions selected according to $f(s)$. That is, it only uses a sub-sequence of the input file to determine the model for the encoding of the entire file.

Algorithm 1 brings the formal descriptions of the encoding procedures for both COMPLETE and SUBSET. The only difference is the addition of zeroing the $g(i)$ function in the last lines for the latter. Thereby, the model is updated at steps indexed by $f(s)$, where s is the number of updates so far. While the COMPLETE variant remembers all the changes from the last update, the SUBSET variant skips over the non-selected values. Decoding is just the reverse process.

Table 2 and 3 continue our running example with $s = 3$, the first for COMPLETE and the second for SUBSET. For the first, the sum of the IC values is 24.564 for b-adp

Algorithm 1: COMPLETE (SUBSET) SELECTIVE

COMPLETE (SUBSET)-SELECTIVE $(T = x_1 \cdots x_n,\ g,\ f)$

```
1 s ← 0; last ← 0 ; Initialize the model according to the uniform distribution on Σ
2 for i ← 1 to n do
3     encode x_i according to the current model
4     if i − last = f(s) then
5         update the model according to the distribution of the characters in Σ, given by the
            probabilities {W(g, σ, i+1)/CW[1, i+1]}_{σ∈Σ}
6         s ← s + 1
7         last ← i
8     else
9         g(i) ← 0
```

	i	1	2	3	4	5	6	7	8	9	10	11	12
	T	d	b	c	a	b	c	b	c	a	a	a	a
b-adp	$g(i)$	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	W	1.000	1.000	1.000	1.000	2.000	2.000	3.000	3.000	2.000	3.000	3.000	3.000
	CW	4.000	4.000	4.000	7.000	7.000	7.000	10.000	10.000	10.000	13.000	13.000	13.000
	p_i	0.250	0.250	0.250	0.143	0.286	0.286	0.300	0.300	0.200	0.231	0.231	0.231
	IC	2.000	2.000	2.000	2.807	1.807	1.807	1.737	1.737	2.322	2.115	2.115	2.115
b-w	$g(i)$	1.000	1.414	2.000	2.828	4.000	5.657	8.000	11.314	16.000	22.627	32.000	45.255
	W	1.000	1.000	1.000	1.000	2.414	3.000	6.414	8.657	3.828	19.828	19.828	19.828
	CW	4.000	4.000	4.000	8.414	8.414	8.414	20.899	20.899	20.899	56.213	56.213	56.213
	p_i	0.250	0.250	0.250	0.119	0.287	0.357	0.307	0.414	0.183	0.353	0.353	0.353
	IC	2.000	2.000	2.000	3.073	1.801	1.488	1.704	1.272	2.449	1.503	1.503	1.503

Table 2. Complete Selective *with* $s = 3$ *for* b-adp *and* b-w *on the running example* $T = x_1 \cdots x_{12} =$ dbcabcbcaaaa.

and 22.296 for b-w. For the second, $g(i)$ is assigned 0 at every position that is not a multiple of 3, which yields a sum of IC values of 23.558 for b-adp and 21.792 for b-w. These examples show that there are special cases for which even the compression efficiency may improve.

The following experiment is based on defining the distance separating consecutive choices in the selective approach by a varying function. We have to balance between the following, opposing, requirements.

1. On the one hand, the weighted approach calls for giving priority to positions close to the one currently processed. This would imply that the selected elements should be denser at the end than at the beginning of the already treated prefix of the file.
2. On the other hand, there is a need to attain as soon as possible a critical mass of selected items, from which a reliable estimate of the true probability distribution may be derived. It is therefore at the beginning of the file that the selected items should be more frequent.

The first option is hard to implement, because the file is processed progressively. We therefore opt for the second one, and try to control the density of the selected items by choosing different parameters for the skip function.

Our next suggestion is a selective method which is tuned by the function g of the weights. The intuition is that the model should not be updated as long as no

	i	1	2	3	4	5	6	7	8	9	10	11	12
	T	d	b	c	a	b	c	b	c	a	a	a	a
b-adp	$g(i)$	0.000	0.000	1.000	0.000	0.000	1.000	0.000	0.000	1.000	0.000	0.000	1.000
	W	1.000	1.000	1.000	1.000	1.000	2.000	1.000	3.000	1.000	2.000	2.000	2.000
	CW	4.000	4.000	4.000	5.000	5.000	5.000	6.000	6.000	6.000	7.000	7.000	7.000
	p_i	0.250	0.250	0.250	0.200	0.200	0.400	0.167	0.500	0.167	0.286	0.286	0.286
	IC	2.000	2.000	2.000	2.322	2.322	1.322	2.585	1.000	2.585	1.807	1.807	1.807
b-w	$g(i)$	0.000	0.000	2.000	0.000	0.000	5.657	0.000	0.000	16.000	0.000	0.000	45.255
	W	1.000	1.000	1.000	1.000	1.000	3.000	1.000	8.657	1.000	17.000	17.000	17.000
	CW	4.000	4.000	4.000	6.000	6.000	6.000	11.657	11.657	11.657	27.657	27.657	27.657
	p_i	0.250	0.250	0.250	0.167	0.167	0.500	0.086	0.743	0.086	0.615	0.615	0.615
	IC	2.000	2.000	2.000	2.585	2.585	1.000	3.543	0.429	3.543	0.702	0.702	0.702

Table 3. Subset Selective *with* $s = 3$ *for* b-adp *and* b-w *on the running example* $T = x_1 \cdots x_{12} =$ dbcabcbcaaaa.

significant mass of weights has been accumulated that can modify it. We only update the model when the ratio of the sum of the weights since the last update, to the total sum of the weights in the model, crosses a certain threshold, as shown in Algorithm 3. The tuned algorithm provides a single method that assigns a decreasing number of updates for b-adp while at the same time it is almost equivalent to a selection with fixed intervals for b-w. The different behaviour of b-adp and b-w by the tuned selection can be seen in Figure 2, that plots the indices where the model updates have been performed for *threshold* = 1.

Algorithm 2: TUNED SELECTIVE

TUNED-SELECTIVE($T = x_1 \cdots x_n$, g, *threshold*)

1 $cum \leftarrow 0$
2 **for** $i \leftarrow 1$ **to** n **do**
3 encode x_i according to the current model
4 $cum \leftarrow cum + g(i)/\big(g(1) + \cdots + g(i-1)\big)$
5 **if** $cum \geq threshold$ **then**
6 update the model by the probability distribution, at position i, of the characters in Σ: $\{W(g, \sigma, i+1)/CW[1, i+1]\}_{\sigma \in \Sigma}$
7 $cum \leftarrow 0$

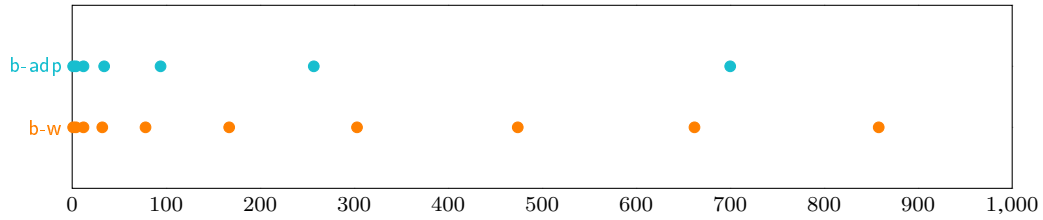


Figure 2. The different behaviour of b-adp and b-w by the tuned selection.

While the tuned selective process is controlled solely by the weight function g, we might wish to modify the selection pace via a chosen parameter. Let $f(j)$ be a function describing the distance from the j-th selected location to the following one. We shall explore the functions $f(j) = j^\alpha$, rounded to the nearest integer, for various values of the parameter α. Choosing $\alpha = 1$ would imply a linear increase in the distance between consecutive selected points. If s locations are selected in the prefix of size i of the text, one gets that $\sum_{j=1}^{s} j$ must be bounded by i, so that $s \leq \sqrt{2i}$. For general α, the corresponding bound is

$$i \geq \sum_{j=1}^{s} j^\alpha \simeq \int^{s} x^\alpha dx \simeq \frac{1}{\alpha+1} s^{\alpha+1}$$

from which one can derive $s \leq \left\lceil (\alpha+1)i \right\rceil^{\frac{1}{\alpha+1}}$. Table 4 shows the first few selected indices for various values of α, where the line for $\alpha = 0$ has been grayed as this is the special case with no selection at all, that is, all the positions are chosen. The values for $\alpha = 1$ are emphasized, and the corresponding method appears in the experimental results as incremental. The column headed s shows the number of selected positions for a text of size $n = 1000$.

α	s	indices of selected points
0	1000	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0.25	300	1 2 3 4 5 7 9 11 13 15 17 19 21 23 25 27 29
0.5	131	1 2 4 6 8 10 13 16 19 22 25 28 32 36 40 44 48
0.75	71	1 3 5 8 11 15 19 24 29 35 41 47 54 61 69 77 85
1	**45**	**1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153**
1.25	31	1 3 7 13 20 29 40 53 69 87 107 129 154 181 211 243 278
1.5	23	1 4 9 17 28 43 62 85 112 144 180 222 269 321 379 443 513

Table 4. *Sample of selected indices for various values of* α.

3 Experimental Results

In order to evaluate the selective methods, we have considered several datasets downloaded from the Pizza & Chili Corpus and report our outcomes here on only two representative files.

- *english* – a concatenation of English texts from the Gutenberg Project;
- *dna* – a sequence of gene DNA sequences obtained from the Gutenberg Project.

As mentioned above, the weighted approach is especially suitable for the encoding of files with locally skewed distributions and is most effective when it is applied on files that have been pre-processed by the *Burrows-Wheeler Transform* (BWT) [2]. To improve the time complexity of this transformation via the use of a suffix array [13], BWT is applied in blocks. We therefore consider only a 4M prefix of the above files. The experiments were conducted on a machine running 64 bit Windows 10 with an Intel Core i5-8250 @ 1.60GHz processor, 6144K L3 cache, and 8GB of main memory.

The plots in Figure 3 summarize the experiments, with those on the left corresponding to english and those on the right to dna. They show the compression ratio

(size of the compressed file divided by the size of the original) and encoding and decoding times in seconds, as a function of the skip size $f(s)$ between consecutive selected positions. This skip size is constant for the basic complete and subset strategies, and represents the *average* interval size for those with varying distances, labelled tuned, α and incremental, which is the special case $\alpha = 1$. As a benchmark, we also added the values of gzip.

As can be seen, there is a slight increase in the size of the compressed file with growing $f(s)$, that is, when the selected positions become sparser, with almost no difference on the performance of the different methods keeping the full statistics on the dna input, and slowly diverging values on the english file. The loss of compression efficiency is more accentuated for the subset approach, which can be explained by the fact that it did not accumulate enough data to get reliable estimates. Note that for these examples, the compression is still better than that of gzip.

In parallel to the slight loss in compression efficiency, there is, with increasing skip size $f(s)$, a significant improvement in both encoding and decoding times, again with similar performance for all the methods, except that based on the selection of a subset, for which the gain in execution time is even stronger. None of these times are comparable with the performance of the highly optimized gzip.

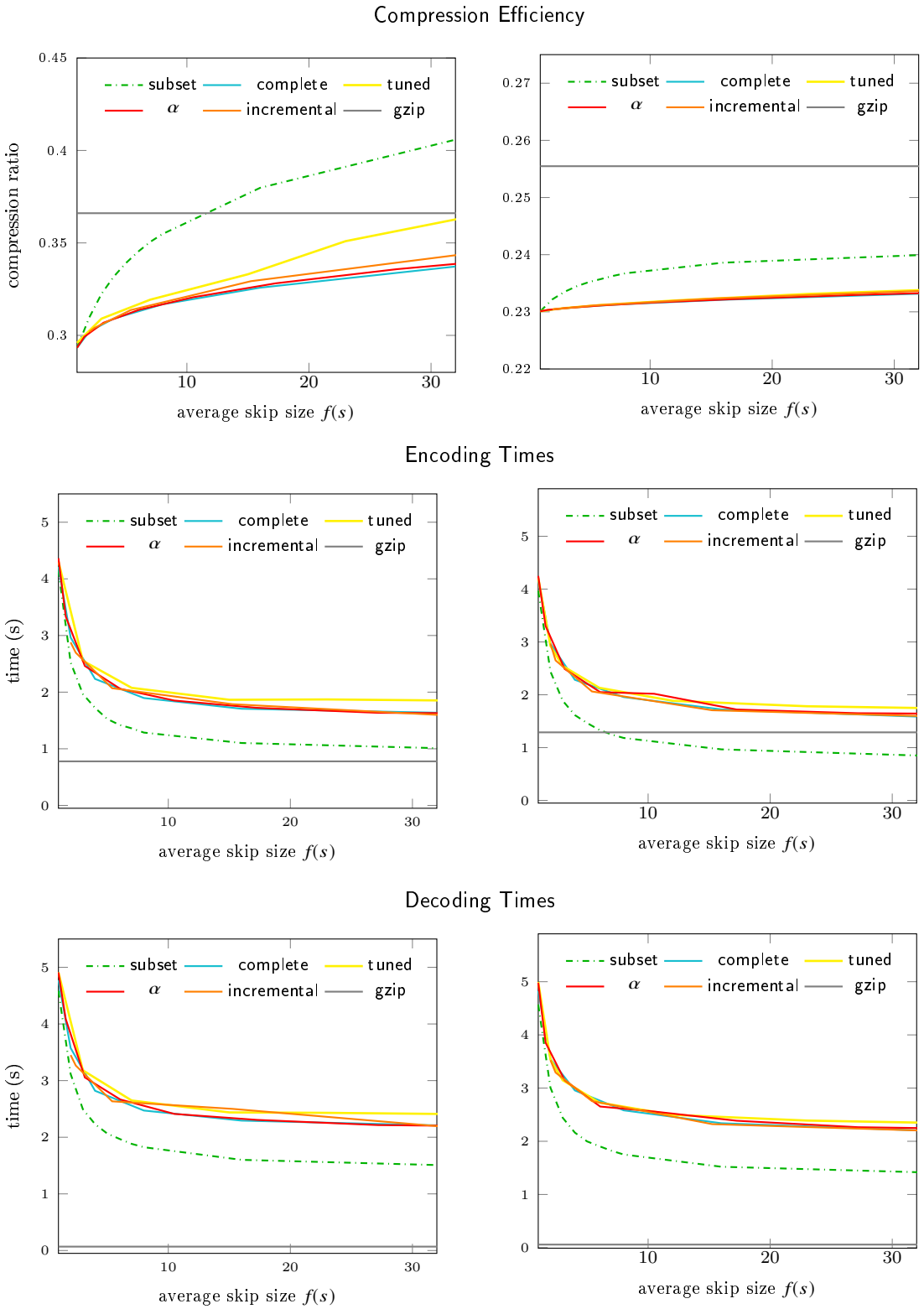


Figure 3: Experimental results for selective variants of b-w as a function of the average skip size $f(s)$ on the 4M *english* and *dna* BWT transformed files. The compression ratio and encoding and decoding times in seconds on the 4M *english* (left) and *dna* (right) BWT transformed files, as a function of the skip size $f(s)$ between consecutive selected positions. This skip size is constant for the basic complete and subset strategies, and represents the *average* interval size for those with varying distances, labelled tuned, α, incremental ($\alpha = 1$) and gzip.

4 Conclusion

We extended the recently introduced weighted adaptive compression paradigm to variants basing the model, on the basis of which the encoding is derived, on various selective approaches. Our empirical tests indicate that the time performance can be significantly improved by the selective methods, while only marginally affecting the compression.

References

1. R. M. Avrunin, S. T. Klein, and D. Shapira: *Combining forward compression with PPM.* SN Comput. Sci., 3(239) 2022.
2. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, 1994.
3. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.
4. N. Faller: *An adaptive system for data compression*, in Record of the 7-th Asilomar Conference on Circuits, Systems and Computers, 1973, pp. 593–597.
5. A. Fruchtman, Y. Gross, S. T. Klein, and D. Shapira: *Backward weighted coding*, in 31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23-26, 2021, IEEE, 2021, pp. 93–102.
6. A. Fruchtman, Y. Gross, S. T. Klein, and D. Shapira: *Bidirectional adaptive compression.* Discret. Appl. Math., 330 2023, pp. 40–50.
7. A. Fruchtman, Y. Gross, S. T. Klein, and D. Shapira: *Weighted Burrows-Wheeler compression.* SN Comput. Sci., 4(265) 2023.
8. R. Gallager: *Variations on a theme by Huffman.* IEEE Transactions on Information Theory, 24(6) 1978, pp. 668–674.
9. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) 1952, pp. 1098–1101.
10. S. T. Klein, E. Opalinsky, and D. Shapira: *Selective dynamic compression*, in Proceedings of the Prague Stringology Conference, Czech Technical University in Prague, Czech Republic, 2019.
11. S. T. Klein, S. Saadia, and D. Shapira: *Forward looking Huffman coding.* Theory of Computing Systems, 2020, pp. 1–20.
12. D. E. Knuth: *Dynamic Huffman coding.* Journal of Algorithms, 6(2) 1985, pp. 163–180.
13. 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.
14. J. S. Vitter: *Design and analysis of dynamic Huffman codes.* JACM, 34(4) 1987, pp. 825–845.
15. I. H. Witten, R. M. Neal, and J. G. Cleary: *Arithmetic coding for data compression.* Commun. ACM, 30(6) 1987, pp. 520–540.

A Worst Case Analysis of the LZ2 Compression Algorithm with Bounded Size Dictionaries

Sergio De Agostino

Computer Science Department
Sapienza University of Rome
Via Salaria 113, 00198 Rome, Italy
deagostino@di.uniroma1.it

Abstract. We make a worst case analysis of practical implementations of LZ2 compression, where the work space remains constant with the increase of the data size and the optimal solution must work with the same on-line decoder. The memory bound implies an off-line standard polynomial time optimal solution with huge multiplicative constants and we show that an on-line approach approximates with a large factor, leaving the design of an effective and more efficient off-line coding as an open problem in this context.

Keywords: factorization, dictionary, optimality, approximation.

1 Introduction

Sheinwald, Lempel and Ziv [17] proved that the power of off-line coding is not useful if we want on-line decodable files, as far as asymptotical results are concerned. Such result extends the asymptotical optimality results of Lempel and Ziv for ergodic sources in a non-constructive way, where the on-line reading of the data from left to right works with a sublinearly bounded buffer length. In the finite case, De Agostino and Storer [8] introduced the notion of on-line decodable Ziv-Lempel (LZ2) optimal coding and proved its NP-completeness. Moreover, a sublogarithmic factor approximation algorithm cannot be realized on-line and the greedy LZ2 compression algorithm is an $O(n^{\frac{1}{4}})$ approximation with binary worst case examples, where n is the string length [7,8]. Therefore, for finite strings, one could afford the extra-cost of off-line coding in exchange of some gain in compression of data stored on a read-only memory. Considering this point, we make in this paper a worst case analysis of practical implementations of LZ2 compression, where the work space remains constant with the increase of the data size. The memory bound implies a standard polynomial time optimal solution with huge multiplicative constants and we show that the on-line approach still approximates with a large factor, leaving the design of an effective and more efficient off-line coding as an open problem in this context. In other words, the trade off between effectiveness (good compression ratio) and efficiency (practical running time) must be improved. This depends on the improvement of the LZ2 string factorization process [20] while on-line greedy LZ1 factorization [15] is already structurally optimal and the only possible improvements are at the coding level as we will discuss in the next sections.

Such need for further theoretical analysis of the power of off-line encoding that must produce on-line decodable files is motivated, as previously mentioned, by applications to read-only memories. Indeed, the design of an on-line decodable off-line

Sergio De Agostino: A Worst Case Analysis of the LZ2 Compression Algorithm with Bounded Size Dictionaries, p. 107.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

coding approximating the optimal solution does not need a real time or even slower algorithm to be considered practical since compression must be performed only once. Therefore, we can accept a time cost that could not be permitted when on-line computation is required as for decoding or on-the fly coding. Typically, LZ2 compression is more efficient but less effective than LZ1 but optimal or nearly optimal LZ2 compression might be more effective than LZ1 on several specific types of data.

In Section 2, Lempel-Ziv compression is described in relation to the LZ1 and LZ2 string factorization methods in the unbounded and bounded memory cases. Then, the on-line versus off-line computation and the greedy versus optimal solution issues are discussed for the unbounded memory case. Such issues are discussed and analyzed for the bounded memory case in Section 3 after giving the polynomial time optimal algorithm. Conclusion and future work are given in Section 4.

2 LZ2 Compression with Bounded Size Dictionaries

Lempel-Ziv compression is a dictionary-based technique [14,15,20], using a string factorization process where the factors of the string are substituted by *pointers* to copies stored in a *dictionary*, which are called *targets*.

2.1 LZ Factorizations

Given an alphabet A and a string S in A^*, the LZ1 factorization of S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the shortest substring which does not occur previously in the proper prefix $f_1 f_2 \cdots f_i$ for $1 \leq i \leq k$ [15]. LZ2 is easier to implement but less effective. The standard LZ2 greedy factorization of a string S, on which practical implementations of LZ2 compression are based, is $S = f_1 f_2 \cdots f_i \cdots f_m$ where each factor f_i is the longest match with the concatenation of a previous factor and the next character [19,20]. f_i is encoded by a pointer q_i whose target is such concatenation. Regardless of memory issues, LZ2 compression can be implemented in real time by storing the dictionary of targets with a trie data structure. When the string length goes to infinity, also the dictionary size does.

2.2 Bounded Size Dictionaries

In practical implementations the dictionary size is bounded by a constant and the pointers have equal size [18]. Let d be the cardinality of the fixed size dictionary (alphabet characters are always dictionary elements). With the most naive approach, there is a first phase of the factorization process where the dictionary is filled up and "frozen". Afterwards, the factorization continues in a non-adaptive way using the factors of the frozen dictionary. In other words, the factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the longest match with the concatenation of a previous factor f_j and the next character, where $j \leq d - \alpha$ and α is the alphabet size. The shortcoming of this heuristic is that after processing the string for a while the dictionary often becomes obsolete. Therefore, after the dictionary is filled up, the compression ratio is monitored. When the ratio deteriorates, a better heuristic deletes all the elements from the dictionary but the alphabet characters and restarts new adaptive and non-adaptive phases. Let $S = f_1 f_2 \cdots f_j \cdots f_i \cdots f_k$ be the factorization of the input string S computed by the LZ2 compression algorithm using such heuristic to bound the dictionary. Let j be the highest index less than i where a restarting

operation happens. Then, f_j is an alphabet character and f_i is the longest match with the concatenation of a previous factor f_h, with $h \geq j$, and the next character (1 and $m+1$ are considered restarting positions by default). This LZ2 compression heuristic with constant work space is called LZC [3] (a variation of LZW compression where the dictionary is restarted with no monitoring) and it is used by the Unix command Compress since it has a good compression effectiveness and it is easy to implement.

2.3 Optimal Factorizations

In the unbounded case, the pointer encoding the factor f_i has a size increasing with the index i. This means that the lower is the number of factors for a string of a given length, the better is the compression. The factorizations described in the previous subsections are produced by greedy algorithms. The question is whether the greedy approach is always optimal, that is, if we relax the assumption that each factor is the longest match, can we do better than greedy? The answer is negative with suffix dictionaries (every suffix of a dictionary element is a dictionary element) as for LZ1 compression. On the other hand, the greedy approach is not always optimal for LZ2 compression. We define $S = f_1 \cdots f_k$ a *feasible* LZ2 factorization if each factor f_i is equal to the concatenation of f_j and the next character, for some $j < i$. A feasible LZ2 factorization with the smallest number of factors is an *optimal* LZ2 factorization. Obviously, the coding of every feasible LZ2 factorization works with the same decoder of the standard greedy LZ2 coding.

As mentioned in the introduction, the optimal approach is NP-complete [8] and the greedy algorithm approximates with an $O(n^{\frac{1}{4}})$ multiplicative factor the optimal solution [7]. Moreover, a sublogarithmic factor approximation algorithm cannot be realized on-line [8]. Although these results can be viewed to be in some sense negative, they serve to motivate the need for further theoretical analysis of the power of off-line encoding that must produce on-line decodable files, as pointed out in [8]. The design of a practical on-line decodable off-line approximation algorithm has important applications to read-only memories. So, we produce further theoretical analysis for the bounded case in the next section.

3 Bounded Memory Greedy versus Optimal Analysis

The memory bound implies a standard polynomial time optimal solution with huge multiplicative constants and we show that an on-line approach still approximates with a large factor. We show the polynomial time algorithm in the first subsection. The second subsection discusses the on-line versus off-line computation issue, while the greedy versus optimal analysis is given in the third subsection.

3.1 The Polynomial Time Algorithm

A *feasible* d-LZC factorization $S = f_1 \cdots f_k$ is such that the number of different concatenations of a factor with the next character between f_h and f_t (f_t is not counted) is less or equal than d decreased by the alphabet size, with h and t two consecutive positions where the restarting operation happens (no restarting between h and t), and each factor f_i with $h < i < t$ is equal to $f_j c$, where c is the first character of f_{j+1} and $h \leq j < i$ (1 and $k+1$ are considered restarting positions by default, meaning that

frozen dictionaries relate to special cases of feasible factorizations). We define *optimal* the feasible d-LZC factorization with the smallest number of factors. The *greedy* d-LZC factorization is the one described in the previous section (the Unix command Compress works with $d = 2^{16}$). We assume, as it happens in practical implementations as well, that the coding inserts a special character when the restarting operation happens. This avoids monitoring of the compression ratio in the standard applications and makes the coding of every feasible factorization decodable (the decrement by one unit of the dictionary size caused by the special character is obviously irrelevant for the compression effectiveness).

A practical algorithm to compute the optimal solution is not known. In order to have a polynomial time optimal algorithm, the bound to the dictionary size should be sublogarithmic. However, the number of all the possible dictionaries induced by a feasible d-LZC factorization of any input string is constant since d and the alphabet cardinality are constant. Given an input string, pair each of these dictionaries with each position of such string. Link the pair (p, D), where p is a position of the string and D is one of the dictionaries, to the pair $(p + \ell + 1, D')$ if ℓ is the length of a dictionary element matching the string in p and D' is the updating of D corresponding to the choice of such dictionary element as factor of a feasible d-LZC factorization. If the match ends the string, the pair links to a special node $\overline{v}$. Also, link the pair (p, D) to (p, A) where A is the dictionary comprising only the alphabet characters, in order to have the possibility of picking p as restarting position. The optimal d-LZC factorization and the sequence of pointers is given by the shortest path from $(1, A)$ to $\overline{v}$. Such polynomial time algorithm is, obviously, unpractical since the number of nodes (pairs) is linear in the string length but the number of dictionaries is a huge multiplicative constant.

3.2 On-line versus Off-line Computation

A trivial upper bound to the approximation multiplicative factor of a feasible factorization with respect to the optimal one is the maximum factor length of the optimal solution, that is, the height of the trie storing the dictionary. Such upper bound is $\Theta(d)$ in the worst case, where d is the dictionary size ($O(d)$ follows from the feasibility of the factorization and $\Omega(d)$ from the factorization of the unary string). In practice, a dictionary comprises thousands of elements and even a logarithmic approximation multiplicative factor is too large. In [8], it is shown in the unbounded case that a sublogarithmic approximation of the optimal LZ2 coding cannot be realized by means of an example binary string $X = pref(n)suff(n)$, where the prefix $pref(n)$ of length $\Omega(n)$ is such that, for any on-line algorithm applied on it, an appropriate suffix $suff(n)$ of length $\Omega(n)$ can be concatenated in order to fool the on-line strategy. From the definition of feasible d-LZC factorization, we know that a dictionary between two restarting positions might comprise less than d elements.

Definition 1. *We call Δ the highest number of elements a dictionary is composed of between two restarting positions in an optimal d-LZC factorization.*

We can adapt the example string X in [8] to the bounded case by considering d the number of different factors selected by the on-line strategy on X, so that it will be at least a logarithmic approximation of Δ. Moreover, we can append to X a sequence of characters where the dictionary performs very badly in order to have a string Y such that $|Y|$ is $\Theta(|X|)$ and LZC compression applied to an arbitrarily

long string $YYY\cdots Y$ has restarting positions on the first character of Y. Then, any on-line approach produces an $\Omega(\log(\Delta))$ approximation of the optimal LZC coding of Y^t for any positive integer t.

3.3 The Greedy versus Optimal Analysis

The proof of the following theorem employs techniques similar to the ones for the unbounded dictionary case of [7]. A preliminay version of this theorem was shown in [6] without giving worst case examples. Such examples are described in this subsection after the proof of the theorem (we suggest to study first the proofs of theorem 3.1 and theorem 3.2 in [7] for the unbounded cases).

Theorem 2. *The greedy d-LZC factorization is an $O(\sqrt{\Delta})$ approximation of the optimal one.*

Proof. Let S be the input string and let R be a substring of S given by the concatenation of the factors of the greedy d-LZC factorization between two consecutive positions where the restarting operation happens. Let T be the trie storing the set I of strings corresponding to factors of the optimal d-LZC factorization of S contained in R. Let Φ be the number of occurrences of all these factors in R. We call an element of the dictionary built by the greedy d-LZC factorization of S an *internal occurrence* if it corresponds to a substring of a factor of I in R. We denote with M_T the number of internal occurrences. The number of non-internal occurrences is less than $|I|$. Therefore, we can consider only the internal ones. For each factor $f \in I$, an internal occurrence corresponding to f is represented by a subpath of the path representing f in T. Let u be the endpoint at the lower level in T of this subpath (which, obviously, represents a prefix of f). Let $d(u)$ be the number of subpaths representing internal occurences with endpoint u and let $c(u)$ be the total sum of their lengths. Since the occurrences (internal or not) are different from each other between two consecutive positions where the restarting operation happens and two equal length subpaths with the same endpoint represent the same factor, we have $c(u) \geq d(u)(d(u)+1)/2$. Therefore

$$1/2 \sum_{u \in T} d(u)(d(u)+1) \leq \sum_{u \in T} c(u) \leq 2|S| \leq 2H_T\Phi$$

where H_T is the height of T and the multiplicative factor 2 is due to the fact that occurrences of dictionary elements may overlap. Since $M_T = \sum_{u \in T} d(u)$, we have

$$M_T^2 \leq |I| \sum_{u \in T} d(u)^2 \leq |I| \sum_{u \in T} d(u)(d(u)+1) \leq 4|T|H_T\Phi$$

where the first inequality follows from the fact that the arithmetic mean is less than the quadratic mean. Then

$$M_T \leq \sqrt{4|I|H_T\Phi} = \Phi\sqrt{\frac{4|I|H_T}{\Phi}} \leq 2\Phi\sqrt{H_T}$$

Since the trie height is $O(\Delta)$, the theorem statement follows. □

We need to adapt the worst case example for the unbounded dictionary case [7] in order to have one for the bounded case. To reformulate such result in our context,

if we let d be the number of factors of the LZ2 factorization of a string of length n in the unbounded case, there exists a binary string X of lenght n on which the optimal factorization working with the same decoder has a number of factors and produces a number of dictionary elements, which are both $\Theta(d^{2/3})$. As in the previous subsection, we can append to X a sequence of characters where the dictionary performs very badly in order to have a string Y such that $|Y|$ is $\Theta(|X|)$ and LZC compression applied to an arbitrarily long string as $YYY \cdots Y$ has restarting positions on the first character of Y. So, the optimal solution employs two thirds of the dictionary space on the input blocks up to a multiplicative contant. On the other hand, the greedy solution cost approximates the optimal solution cost with a multiplicative approximation factor which is, up to a multiplicative constant, greater than the square root of the actual dictionary size needed by the optimal approach. So, it follows from the proof of Theorem 1 that the square root of the actual dictionary size needed by the optimal approach is a tight bound to the approximation factor of the greedy approach, up to multiplicative constants.

4 Conclusion

The gap between on-line and off-line computation, shown in this paper, has its stringological reason in the structure of the dictionary which might not contain all the suffixes of its elements. Differently, with LZ1 coding dictionaries have this property and greedy factorizations are optimal. However, the coding is more expensive and on-line improved variants exist employing either fixed-length codewords [4,16] or variable-length ones [5,9,10,11,12,13]. Moreover, there are off-line approaches to improve LZ1 coding working with on-line decoders [1,2]. As previously pointed out, with LZ2 coding practical off-line string factorizations approximating on-line decodable optimal solutions could be more effective than LZ1 compression on several specific types of data.

References

1. A. Apostolico and S. Lonardi: *Compression of biological sequences by greedy off-line textual substitution*, in Proceedings IEEE Data Compression Conference, 2000, pp. 143–152.
2. A. Apostolico and S. Lonardi: *Off-line compression by greedy textual substitution*, in IEEE Proceedings, vol. 88, 2000, pp. 1733–1744.
3. T. C. Bell and I. H. Witten: *Text Compression*, Prentice Hall, 1990.
4. M. Crochemore, A. Langiu, and F. Mignosii: *Note on the greedy parsing optimality for dictionary-based text compression.* Theoretical Computer Science, 525 2014, pp. 55–59.
5. M. Crochemore, G. M., A. Langiu, F. Mignosi, and A. Restivo: *Dictionary symbolwise flexible parsing.* Journal of Discrete Algorithms, 14 2012, pp. 74–90.
6. S. DeAgostino: *Greedy versus optimal analysis of bounded size dictionary compression and on-the-fly distributed computing*, in Proceedings Prague Stringology Conference, 2020, pp. 74–83.
7. S. DeAgostino and R. Silvestri: *A worst case analysis of the lz2 compression algorithm.* Information and Computation, 139 1997, pp. 258–268.
8. S. DeAgostino and J. A. Storer: *On-line versus off-line computation in dynamic text compression.* Information Processing Letters, 59 1996, pp. 169–174.
9. A. Farrugia, P. Ferragina, A. Frangioni, and R. Venturini: *Bicriteria data compression*, in Proceedings SIAM-ACM Symposium on Discrete Algorithms (SODA 14), 2014, pp. 1582–1585.
10. P. Ferragina, I. Nitto, and R. Venturini: *On optimally partitioning a text to improve its compression.* Algorithmica, 61 2011, pp. 51–74.

11. P. Ferragina, I. Nitto, and R. Venturini: *On the bit-complexity of lempel-ziv compression.* SIAM Journal on Computing, 42 2013, pp. 1521–1541.
12. D. Kosolobov: *Relations between greedy and bit-optimal lz77 encodings*, in Proceedings Symposium on Theoretical Aspect of Computer Science, 2018, pp. 46:1–46:14.
13. A. Langiu: *On parsing optimality for dictionary-based text compression - the zip case.* Journal of Discrete Algorithms, 20 2013, pp. 65–70.
14. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 22 1976, pp. 75–81.
15. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
16. Y. Matias and C. S. Sahinalp: *On the optimality of parsing in dynamic dictionary-based data compression*, in Proceedings SIAM-ACM Symposium on Discrete Algorithms (SODA 99), 1999, pp. 943–944.
17. D. Sheinwald, A. Lempel, and J. Ziv: *On encoding and decoding with two - way head machines.* Information and Computation, 116 1995, pp. 128–133.
18. J. A. Storer: *Data Compression: Methods and Theory*, Computer Science Press, 1988.
19. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
20. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.

Turning Compression Schemes into Crypto-Systems

Kfir Cohen[1], Yonatan Feigel[1], Shmuel T. Klein[1], and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
{kfirco12,feyon9}@gmail.com, tomi@cs.biu.ac.il
[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@g.ariel.ac.il

Abstract. Some techniques are presented turning several of the classical compression schemes into crypto-systems, not only reducing their space, but also securing their content by means of encryption based on the knowledge of a secret key. Only a single bit of the key is consumed at every encoding step, and the cumulative impact of applying these steps is shown to produce almost random output. Only the techniques are presented; security issues are deferred to future work.

1 Background

The ever increasing number of cyber attacks necessitates the protection of all transmitted personal data and, in particular, delicate information. The challenge is to protect the data without hurting the network's throughput rate. A *Compression Cryptosystem*, which performs compression and encryption simultaneously, is meant to overcome these challenges. This combined system is accomplished by either inserting compression tools into encryption algorithms (see [22] for example), or by embedding cryptography into the compression systems (see [8] and [5]).

Encrypted data usually cannot be distinguished from a randomly generated file, as empirically shown by Sharma and Bollavarapu [15] and by Carpentieri [2]. Therefore, when both compression and encryption are desired, compression cannot be applied after encryption, only before or in parallel. Most existing solutions perform compression and encryption sequentially. Contrarily, in this paper we propose a *single* process that produces secure outputs against unauthorized eavesdroppers and also reduces the memory usage. Unlike the compression cryptosystem of [8], that used the secret key to control the coding of the *model*, the current research follows the approach of [5] and controls the code generation itself.

A *One Time Pad* (OTP) is a process for encrypting any message, sequentially, by XORing it with a one-time secret key of length at least the size of the message. Singh et al. [16] apply OTP on a file that has been compressed by arithmetic coding. Empirical experiments show that the processing times of simultaneous compression and encryption are shorter than applying them sequentially. Raju [12] has examined this fact for OTP with arithmetic coding, while Sangwan [13] tested it for a particular algorithm that combines reversed static Huffman codewords with different secret keys. Setyaningsih and Wardoyo [14] provide a thorough survey concerning the combination of cryptography and lossless and lossy compression methods, and they mention that most of such research concentrates on image security rather than on compression efficiency.

Our previous work concentrated on Huffman and arithmetic coding and performed a sequence of small perturbations steps, which cumulatively had the impact of scrambling the output enough to turn the decoding into an (almost) impossible task without

Kfir Cohen, Yonatan Feigel, Shmuel T. Klein, Dana Shapira: Turning Compression Schemes into Crypto-Systems, pp. 114–123.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

the full knowledge of the secret key $\mathcal{K}$. Yet, the number of bits of $\mathcal{K}$ consumed by each perturbation step is of the order of $\log n$, where n is the size of the encoded alphabet. Since we do not limit this alphabet to consist only of single characters, n might in fact be quite large. In many applications, especially when processing large Information Retrieval corpora, the term *alphabet* should be understood in a broader sense, and it may consist of the different words in the textual database, or of other variable length strings, see, e.g., [10] or [1].

We therefore turn in the current work to a different paradigm, in which we constrain the process to use at most a single bit of the secret key at each step, actually, a binary decision between two plausible alternatives. This reduction in the number of required secret bits may come at the price of a possible deterioration of the compression performance. However, our experiments indicate only a negligible loss in compression efficiency, as shown below.

The idea of shuffling elements of the compression model, according to a secret key, has already been explored is several studies. For example, Wang [19] shuffles such elements of several well known compression algorithms in a preprocessing stage. In particular, the initial *dictionary* is shuffled in LZW compression [20], the *order* of the alphabet symbols in the working interval is shuffled in case of arithmetic coding [21], and the Huffman *tree* gets scrambled when Huffman coding is used. A similar approach for Huffman coding, that shuffles the Huffman tree in a preprocessing stage, is performed in [18]. Zebari [23] converts a given message into a corresponding DNA file based on a secret function, followed by compressing the converted message and then encrypting it by an additional shuffling procedure. Kelley and Tamassia [6] suggest a compression cryptosystem based on LZW that periodically alters the dictionary. In our work, following the approach of [5], the shuffles are done iteratively, and not only in a preprocessing or postprocessing stage.

The resulting size of the ciphertext might sometimes become a burden for data protection. BREACH is an attack based on the size of the output ciphertext due to Gluck et al. [4]. Their idea is to gradually try to retrieve encrypted secrets from a HTTPS channel.

For our experiments, we used the King James version of the Bible, of size about 4.25MB, taken from the Gutenberg project[1], and the 50MB variants of the files *english*, *XML* and *DNA* of the Pizza & Chili corpus[2]. The fact that our results were similar on all the tests shows that they are not dependent of the nature of the input files.

In the next section, we explore our new ideas in detail for Huffman and arithmetic coding, as well as for the Ziv-Lempel dictionary based methods LZW and LZ77, and for compression after having applied the Burrows-Wheeler Transform. Section 3 concludes and suggests future work.

2 Adding binary decisions to the compression process

We assume that a secret key $\mathcal{K}$ of sufficient length has been exchanged between sender and receiver prior to the transmission of any encoded message $\mathcal{E}_{\mathcal{K}}(M)$ called the *ciphertext*, and that this key is not known to a potential opponent, whose aim it is to reveal the content of the original message $\mathcal{D}_{\mathcal{K}}(\mathcal{E}_{\mathcal{K}}(M)) = M$, known as the

[1] https://www.gutenberg.org/cache/epub/10/pg10.txt
[2] http://pizzachili.dcc.uchile.cl/

cleartext. To facilitate the description of our methods, we may assume that the size of the key is at least as long as the message itself, though this would then be equivalent to an ideal OTP. In practice, a secret key of any length can be produced by choosing a random seed for some random number generator, of length, say, 1000 bits, so that it cannot be guessed.

The common idea for the following methods is to let the compression algorithm depend on some binary choice, which on the one hand, should not at all, or only slightly, impact on the compression efficiency, but on the other hand, produce a completely different ciphertext. The methods differ in the details on where in the process such a choice may be applied. The challenge is in the design of the algorithm, the correctness and efficacy of which should be oblivious to the actual value of the secret key $\mathcal{K}$.

2.1 Huffman coding

Huffman codes are well known to yield optimal compression once it has been decided which elements to encode. The set of these elements is referred to as the *alphabet*, even if they are not just single characters. The additional constraint for optimality is that all the codewords consist of an integral number of bits, which is not obvious, as it may be overcome by arithmetic coding. There are, however, many different Huffman codes, or, equivalently, Huffman trees, for any single probability distribution. We consider *Mirror* and *Swap* transformations, already mentioned in [5].

Mirror: Given is a Huffman tree T and one of its internal nodes v, or, equivalently, a Huffman code C_T and a proper prefix α_v of some of its codewords. A mirror transformation $M(T)$ of T replaces the subtree T_v rooted at v by its mirror image, in which the left and right branches emanating from every internal node of the subtree T_v are interchanged. Equivalently, the codewords corresponding to the leaves of T_v have their suffixes following the prefix α_v complemented, that is, if $\alpha_v\beta \in C_T$, then $\alpha_v\overline{\beta} \in C_{M(T)}$, for all strings β. Since the lengths of the codewords did not change, their average is still optimal.

Swap: Using the same notation, a swap transformation $S(T)$ of T replaces the subtree T_v rooted at v by a subtree whose own left and right subtrees have changed sides, but without altering their shape. This is like the mirror transformation, but only for one level, without continuing recursively to the sub-subtrees. Equivalently, only the bit following the prefix α_v is complemented, for all the codewords corresponding to the leaves of T_v. That is, if $\alpha_v\gamma\beta \in C_T$, then $\alpha_v\overline{\gamma}\beta \in C_{S(T)}$, for $\gamma \in \{0,1\}$ and all strings β. Here again, the transformation does not change the lengths of the codewords.

Table 1 shows an illustrative example. The original code is given in the left column, and the chosen internal node v is the right child of the root of the tree, corresponding to the prefix $\alpha_v = 1$. The elements of the alphabet corresponding to the leaves of T_v are shown on shaded background, red or blue for the left or right subtrees of T_v, respectively.

The middle column shows the code after the mirror transformation: all the suffixes, shown in red or blue, of the codewords starting with 1 are complemented, and are then rearranged to keep the lexicographic order. As a result, the left to right order of the alphabet elements corresponding to leaves of the subtree T_v is reversed, as indicated by matching color shades.

Original		Mirror		Swap	
a	0 0	a	0 0	a	0 0
b	0 1 0 0	b	0 1 0 0	b	0 1 0 0
c	0 1 0 1 0	c	0 1 0 1 0	c	0 1 0 1 0
d	0 1 0 1 1	d	0 1 0 1 1	d	0 1 0 1 1
e	0 1 1	e	0 1 1	e	0 1 1
f	1 0 0	m	1 0 0 0	k	1 0 0
g	1 0 1 0 0	l	1 0 0 1	l	1 0 1 0
h	1 0 1 0 1	k	1 0 1	m	1 0 1 1
i	1 0 1 1 0	j	1 1 0 0 0	f	1 1 0
j	1 0 1 1 1	i	1 1 0 0 1	g	1 1 1 0 0
k	1 1 0	h	1 1 0 1 0	h	1 1 1 0 1
l	1 1 1 0	g	1 1 0 1 1	i	1 1 1 1 0
m	1 1 1 1	f	1 1 1	j	1 1 1 1 1

Table 1: *Mirror and swap transformations in a Huffman code.*

The right column corresponds to a swap. Here, only the single bit after the prefix $\alpha_v = 1$ is complemented, which moves the codewords of the left and right subtrees of T_v as solid blocks, without changing their internal order.

While in previous work, $\log n$ bits of the secret key were used at each step to point to one of the internal nodes v of the tree T at which the mirror or swap had to be applied, we now suggest to use just a single bit to decide whether the transformation will be of type mirror or swap. This supposes that the internal nodes of the tree are scanned in some predetermined fixed order, known to both encoder and decoder, and possibly also to a potential eavesdropper. The order could be generated randomly, to avoid anomalies in trees with particular layouts, but it is not assumed to be a part of the secret known only to the communicating parties.

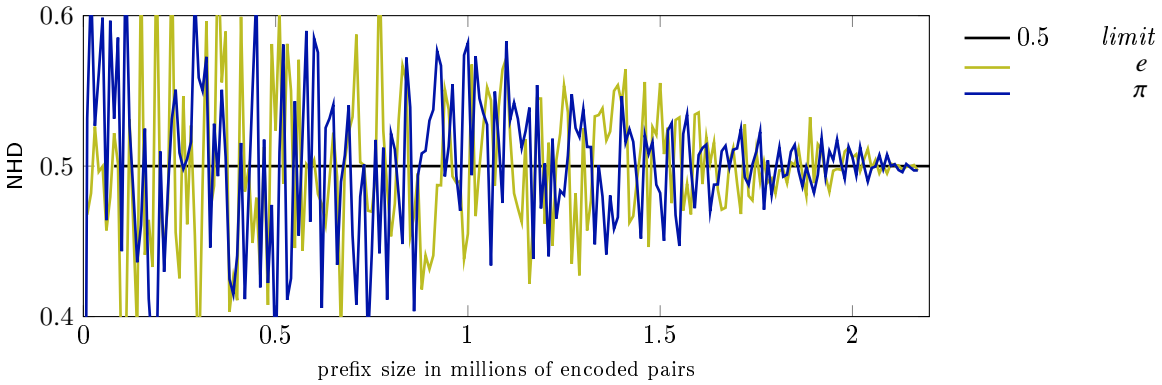


Figure 1. NHD for two runs on the same text with different keys for Huffman coding.

Given two binary files A and B, we define their *Normalized Hamming distance* as

$$\mathsf{NHD}(A,B) = \frac{1}{n}\sum_{i=1}^{n} A_i \text{ XOR } B_i,$$

where X_i is the ith bit of the file X, n is the size of the larger of the two files A and B, and X_i is defined as 0 for indices larger than the size of X. We use the NHD as a measure of the similarity between files, a perfect match corresponding to NHD = 1, while for completely unrelated files, the NHD should tend to $\frac{1}{2}$. Figure 1 shows the effect on the King James Bible of applying the suggested transformations with random keys generated by different seeds. In our example, we have chosen as seeds the first

few bits of the expansion of e or π. The figure plots the NHD between a file obtained by applying mirror and swap transformations and a file using the original Huffman code for all its codewords, as a function of the index within the files. We see that the values fluctuate symmetrically and ultimately zoom in on the expected limit $\frac{1}{2}$. Moreover, the two files using the transformations with different seeds are themselves completely unrelated.

2.2 Arithmetic coding

The output of an arithmetic coder is a subinterval within $[0, 1)$, or just a single real number in it. One starts with $[0, 1)$ and at each step, the current interval is narrowed according to the currently processed character. More precisely, the unit interval is partitioned into regions corresponding to the elements of the given alphabet, the sizes of the regions being proportional to the probabilities of the characters. Arithmetic coding is known to reach entropy, regardless of the initial assignment of the regions to the characters.

In previous work, the suggested transformations were swaps between adjacent regions, which are easy to implement as only a single borderline between regions has to be moved. The cumulative effect of a sufficiently large number of such swaps is that they end up in what could be deemed as a *random* permutation. The new binary variant we suggest in this work is to decide according to the current bit of the secret key, whether to perform the swap of a given interval with its left or right adjacent neighbor, working cyclically for the extreme intervals.

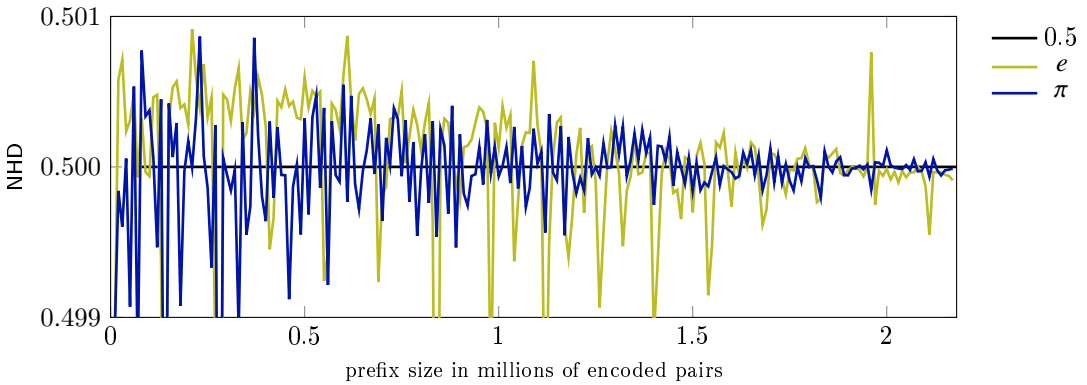


Figure 2. NHD for two runs on the same text with different keys for arithmetic coding.

The idea is similar to that of the Huffman coding seen in the previous section: even if an eavesdropper knows about the strategy of constantly swapping adjacent intervals, decoding will not be possible without knowing the side of the swap, and guessing it is impossible. Figure 2 illustrates the NHD of the encrypted file with one produced without this encryption, again for the two seeds e and π used above. Here the convergence to the expected limit $\frac{1}{2}$ is even faster (note the scale on the y-axis), as expected for arithmetic coding, whose output, even without encryption, is known to be especially close to random [7].

To verify that the convergence to $\frac{1}{2}$ is not due to the difference of the expansions of the irrational numbers e and π, but rather to the random fluctuations introduced by the swapping process, we repeated the above experiments for both Huffman and arithmetic coding using almost identical secret keys: the binary expansion of e on the

one hand, and the same string, in which we have flipped the tenth bit, on the other hand. As can be seen in Table 2, the produced files are still as different as if they were randomly generated, even though they follow both almost the same perturbation sequences.

	Huffman	arithmetic
Bible	0.5009	0.49979
English	0.5005	0.49994
DNA	0.4998	0.49995

Table 2: *Limit values of NHD comparing runs with secret keys differing only in one bit.*

An alternative way to apply small perturbations to arithmetic coding, and in fact also to other compression methods, is to tell *small lies*, that is, not always transmitting the true values. The idea here is to occasionally, instead of entering the intended sub-interval, use a predetermined *other* one, that may be the one next in size or the one adjacent in terms of character encoding. The choice of whether or not to do so is again solely guided by the bits of the secret key.

It is known that if some sort of mistake took place during the transmission of an arithmetically encoded message, this will not only result in a wrong character being perceived by the decoder in the place where the error has occurred, but also all the subsequent characters are irreversibly lost as well. And so, if an evil listener were to intercept the transmission of such an encrypted message, where we intentionally lead him into the wrong sub-intervals every so often, he would not be able to decode any significant part of the message. The opponent would need to guess correctly at every single letter if it was flipped, otherwise the decoded message will very quickly turn into random noise.

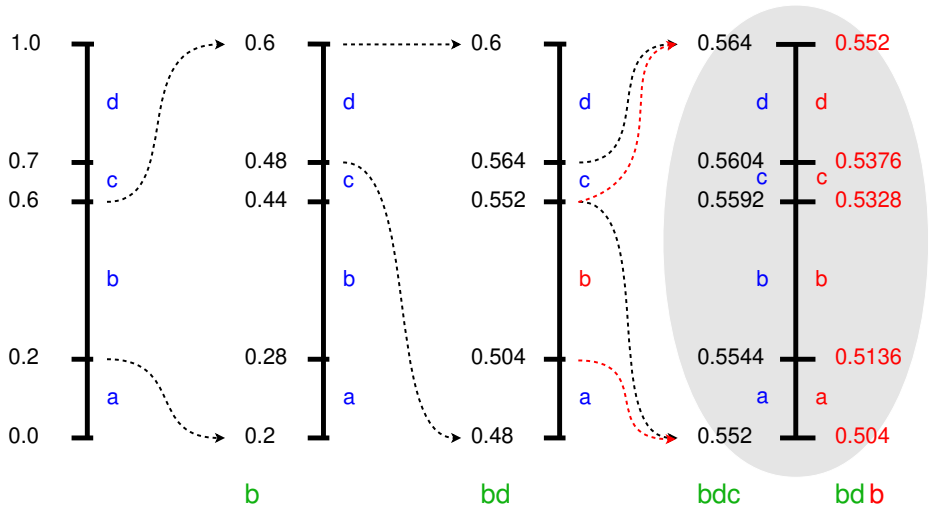


Figure 3: *Example of arithmetic coding. Each bar is a refinement of the chosen sub-interval to its left. In the rightmost step, we should have used the (black) sub-interval corresponding to character* c*, but we use the (red) sub-interval instead, corresponding to* b*. This has as effect that the partition process continues with the red partition, written to the right of the bar, instead of with the black partition, written to its left.*

An illustration of such a transmitted *lie* is given in Figure 3, depicting a simple example with an alphabet of four letters {a, b, c, d}, appearing with probabilities 0.2, 0.4, 0.1 and 0.3, respectively. Suppose the message to be encoded starts with bdc$\cdots$.

The first two steps recall the mechanism of arithmetic coding, narrowing the initial interval $[0, 1)$ first to $[0.2, 0.6)$ according to the first character b, and then further to $[0.48, 0.6)$ for the second character d. In the regular process, the third character c would then yield the interval $[0.552, 0.564)$, but if we instead decide to choose the adjacent interval corresponding to b, the new interval would be $[0, 504, 0.552)$; the new partitions are depicted in black and red in the left and right parts of the shaded area.

An unauthorized decoder, who has no access to the secret key, would not know that the decoded letter b should have actually be replaced by the following letter in line instead, c in our example. Even without repeating the perturbation process, the decoding would already be mislead into a completely different path.

A possible attack would be, if it is known that the decoded output is not reliable, and that occasionally, a deliberately wrong character is transmitted, to try an exhaustive search through all the possible alternatives. This might work if only a single or very few such lies are used, but incurs a prohibitive search cost if we apply such perturbations possibly after each character.

It should be noted that the compression efficiency is necessarily hurt by introducing these lies, as opposed to the techniques mentioned before for Huffman coding, or for arithmetic coding with swapping the position of adjacent intervals, that preserve the compression optimality. This is so because the given input file defines for the characters to be encoded a probability distribution $p_1, \ldots, p_n$. By encoding from time to time different characters, we deviate from the original to a different probability distribution $q_1, \ldots, q_n$. The size of the arithmetically encoded file using this lying strategy is thus $-\sum_{i=1}^{n} p_i \log q_i$, but by a well know theorem, the size S of the original encoded file satisfies

$$S = -\sum_{i=1}^{n} p_i \log p_i \leq -\sum_{i=1}^{n} p_i \log q_i, \tag{1}$$

with equality only if $p_i = q_i$ for all i.

Pushing the idea of transmitting sporadically a different character even further, consider a strategy in which the characters of the alphabet $c_1, \ldots, c_n$ are ordered by their probabilities $p_1 \leq \cdots \leq p_n$ and in which one transmits consistently the next character with the next higher probability, that is, c_{i+1} is encoded instead of c_i for $1 \leq i < n$, and c_0 instead of c_n. This may look as a promising compression method for itself at first sight, even without any connection to encryption, because for almost all characters, there is a gain: actually using a higher probability results in narrowing less the current interval, and ultimately should require less bits for the encoding of the entire message.

Unfortunately, the most frequent character c_n being replaced by the rarest one c_1, necessarily cancels the entire gain, because of the theorem mentioned in eq. (1). The loss could even be significant. A refined solution could thus be to partition the ordered sequence of characters into several adjacency regions and to apply the cyclic shift within each region rather than globally. Formally, we define k subsequences of the n indices, 1 to n_1, $n_1 + 1$ to n_2, …, $n_{k-1} + 1$ to $n_k = n$. The shifts are then from c_i to c_{i+1} and from $c_{n_{j+1}}$ to c_{n_j+1}, for all $n_j + 1 \leq i < n_{j+1}$, with $0 \leq j < k$ and $n_0 = 0$. Table 3 presents some of the compression results for arithmetic coding on two 50MB test files from the *Pizza & Chili* corpus, with all file sizes given in MB.

	alphabet size	compressed no encryption	compressed 1 cycle	compressed 5 cycles	compressed 10 cycles
English	99	26.95	27.83	27.01	26.97
XML	97	32.69	32.99	32.79	32.72

Table 3: *Arithmetic compression performance on test files. The column headed* no compression *refers to pure arithmetic compression without the lying heuristic.*

As can be seen, while using a single cycle incurs an increase of 1–3% in the size of the compressed file, partitioning the alphabet into 5–10 equisized cycles reduces this loss to less than 0.1%.

2.3 LZW

Welch's variant of the LZ78 algorithm [20] builds a dictionary $\mathcal{D}$ that is initialized by the single characters, and grows dynamically by the addition, at each stage, of the longest newly encountered substring. The algorithm consists of two independent, yet interleaving, processes:

1. greedily parsing the given text T by finding the longest element $A \in \mathcal{D}$ matching the characters following the current position in T;
2. inserting a new element $B = Ax$ into $\mathcal{D}$, where x is the first character that caused a mismatch.

The processes are independent, because one could at any point, and in particular when the dictionary fills up, stop the updating of the second process and continue with the first alone, as in a static dictionary parsing. The idea to turn LZW into a compression-crypto system is to perform the update process of the second point selectively, according to the 1-bits of the secret key $\mathcal{K}$. This is similar to the selective update process of the statistics in dynamic arithmetic coding [8], where instead of basing the model on the last seen m characters, one chooses randomly m of the $2m$ last characters.

There is, however, a compression deterioration of 1–3% in this variant of LZW on our test files, while for arithmetic coding the fluctuations in the size of the compressed file were hardly noticeable. Figure 4 plots the NHD in the same format as above, showing again convergence to the expected $\frac{1}{2}$. As already noted in [7], the output of LZW is not fully random, since it consists of pointers of predetermined sizes (the first 256 pointers use 9 bits, the next 512 have 10 bits, etc.), and their leading bits have a higher chance to be 0 than 1. We therefore extracted the two first bits of all the codewords into a separate file, that could be compressed by arithmetic coding. The output of this combined process has been used to generate the graphs in Figure 4.

2.4 LZ77

The output of the variant LZSS [17] of the LZ77 algorithm is a sequence of items, each of which can be either a single character or an (offset, length) pair. A 1-bit flag is used to differentiate between the alternatives. In practice, after a sufficiently long prefix, the output consists only of a sequence of pairs.

We adapt the strategy of not always telling the truth by adding some small integer d to the offset, if and only if the current bit in the secret key $\mathcal{K}$ is 1. If such a disruption

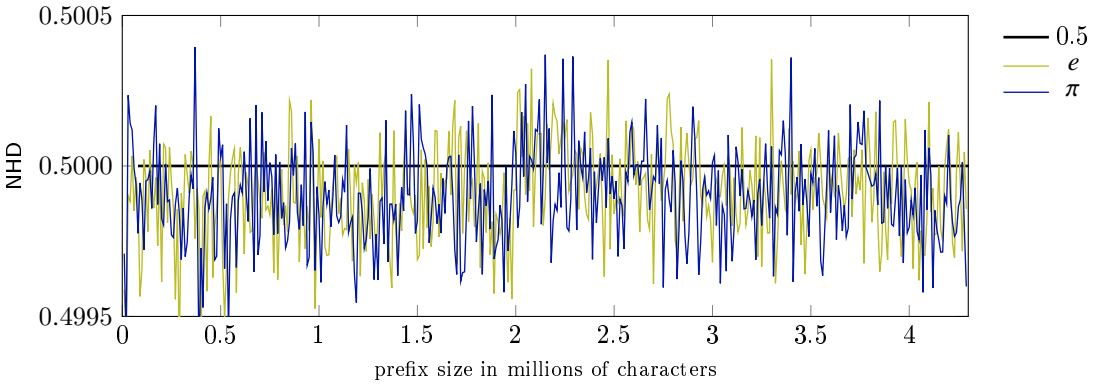


Figure 4: NHD for two runs on the same text with different keys for LZW.

occurs only once, an opponent who does not know that the offset has been shifted, will copy some wrong characters. This may seem as a local perturbation, while a single wrong bit in the output of arithmetic coding generally plays havoc with the rest of the file. However, even for LZ77, the wrong characters may be referenced later, adding more errors, and this may lead to a snowball effect, destroying eventually the message completely. The slight increase of the offsets caused a compression loss of 2-4% on our test files. The NHD graphs also show convergence to the expected $\frac{1}{2}$ and are omitted.

An additional twist to complicate malicious decoding attempts even further is to let the shift parameter d also depend on $\mathcal{K}$: one could, for instance, reuse four of the last processed bits of the key and define d accordingly as an integer between 1 and 16.

2.5 Burrows-Wheeler Transform

The Burrows-Wheeler transform is not a compression method on its own and only permutes its input. It has, however, a tendency to produce long runs of identical characters, and its output is generally much more compressible, which is why it is cascaded with techniques like Move-to-front (MTF) followed by run-length-encoding in popular software like bzip2.

Previous work with the BWT includes Külekci [9], who proposes a compression cryptosystem based on a random permutation of the alphabet in the BWT; the system supports pattern matching on the compressed form, while still retaining its security. Similar alterations to the BWT are mentioned in Teuhola [11] with applications to clustering.

Our suggestion here is to decide according to $\mathcal{K}$ whether to apply MTF or its variant Move-to-middle (MTM). Indeed, both heuristics are plausible: MTM reacts slower to dramatic changes in the text, but might be preferable for sporadic appearances of rare characters. On our test files, MTM gave slightly lower compression, and the suggested MTF+MTM approach yielded an increase of up to 8% in size, and similar NHD graphs.

3 Conclusion and future work

We presented techniques to turn some of the classical compression methods into crypto-systems, based on a secret key shared by sender and receiver, at the cost of

a mostly negligible loss in compression efficiency. Of course, the randomness alone of the output does not imply that the systems are secure against attacks and we shall work on this aspect, for example by trying to show the NP-completeness of the problem of breaking the code, as done in [3] or [5].

References

1. N. Brisaboa, S. Ladra, and G. Navarro: *DACs: Bringing direct access to variable-length codes.* Inf. Process. Manag., 49(1) 2013, pp. 392–404.
2. B. Carpentieri: *Efficient compression and encryption for digital data transmission.* Security and Communication Networks, 9591768 2018, pp. 1–9.
3. A. Fraenkel and S. Klein: *Complexity aspects of guessing prefix codes.* Algorithmica, 12(4) 1994, pp. 409–419.
4. Y. Gluck, N. Harris, and A. Prado: *Breach: Reviving the crime attack.* Black Hat Conference, Las Vegas, USA, July 27–August 1, 2013.
5. Y. Gross, S. Klein, E. Opalinsky, R. Revivo, and D. Shapira: *A Huffman code based crypto-system*, in Data Compression Conference, DCC'22, Snowbird, UT, USA, March 22-25, 2022, pp. 133–142.
6. J. Kelley and R. Tamassia: *Secure compression: Theory & Practice.* IACR Cryptol. ePrint Arch., 2014, p. 113.
7. S. Klein and D. Shapira: *On the randomness of compressed data.* Inf., 11(4) 2020, p. 196.
8. S. Klein and D. Shapira: *Integrated encryption in dynamic arithmetic compression.* Inf. Comput., 279:104617 2021.
9. M. O. Külekci: *On scrambling the Burrows-Wheeler transform to provide privacy in lossless compression.* Comput. Secur., 31(1) 2012, pp. 26–32.
10. A. Moffat: *Word-based text compression.* Softw. Pract. Exp., 19(2) 1989, pp. 185–198.
11. A. Niemi and J. Teuhola: *Burrows-Wheeler post-transformation with effective clustering and interpolative coding.* Softw. Pract. Exp., 50(9) 2020, pp. 1858–1874.
12. J. Raju: *A study of joint lossless compression and encryption scheme*, in International Conference on Circuit, Power and Computing Technologies, IEEE, 2017, pp. 1–6.
13. N. Sangwan: *Combining Huffman text compression with new double encryption algorithm*, in C2SPCA Conference, IEEE, 2013, pp. 1–6.
14. E. Setyaningsih and R. Wardoyo: *Review of image compression and encryption techniques.* Intern. J. of Advanced Computer Science and Applications, 8(2) 2017, pp. 83–94.
15. R. Sharma and S. Bollavarapu: *Data security using compression and cryptography techniques.* International J. of Computer Applications, 117(14) 2015.
16. A. Singh and R. Gilhotra: *Data security using private key encryption system based on arithmetic coding.* International Journal of Network Security & Its Applications (IJNSA), 3(3) 2011, pp. 58–67.
17. J. Storer and T. Szymanski: *Data compression via textual substitution.* J. ACM, 29(4) 1982, pp. 928–951.
18. T. Subhamastan Rao, M. Soujanya, T. Hemalatha, and T. Revathi: *Simultaneous data compression and encryption.* International Journal of Computer Science and Information Technologies, 2(5) 2011, pp. 2369–2374.
19. C. Wang: *Cryptography in data compression.* Code-Breakers Journal, 2(3) 2006.
20. T. Welch: *A technique for high-performance data compression.* IEEE Computer, 17(6) 1984, pp. 8–19.
21. K. Wong, Q. Lin, and J. Chen: *Simultaneous arithmetic coding and encryption using chaotic maps.* IEEE Trans. on Circ. and Syst.–II Express Briefs, 57(2) 2010, pp. 146–150.
22. K. Wong and C. Yuen: *Embedding compression in chaos based cryptography.* IEEE Trans. on Circuits and Systems–II Express Briefs, 55(11) 2008, pp. 1193–1197.
23. D. Zebari, H. Haron, D. Zeebaree, and A. Zain: *A simultaneous approach for compression and encryption techniques using deoxyribonucleic acid*, in 2019 13th SKIMA Conference, IEEE, 2019, pp. 1–6.

Author Index

Proceedings of the Prague Stringology Conference 2023
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 978-80-01-07206-6

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