Proceedings of the Prague Stringology Conference 2024

Edited by Jan Holub and Jan Žďárek

August 2024

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

ISBN 978-80-01-07328-5

Preface

The proceedings in your hands contain a collection of papers presented in the Prague Stringology Conference 2024 (PSC 2024) held on August 26–27, 2024 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 seven papers in this proceedings made the cut and were selected for regular presentation at the conference.

The PSC 2024 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 2024 is the twenty-seventh 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, 2023 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 2024 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 2024. 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 2024

Jan Holub and Dominik Köppl

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)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Dominik Köppl, *Co-chair*	(Tokyo Medical and Dental University, Japan)
Thierry Lecroq	(Université de Rouen, France)
Robert Mercas	(Loughborough University, United Kingdom)
Yuto Nakashima	(Kyushu University, Japan)
Solon Pissis	(CWI, The Netherlands)
William F. Smyth	(McMaster University, Canada)
Bruce W. Watson	(National Security Centre of Excellence, Canada)
Jan Žďárek	(Czech Technical University in Prague, Czech Republic)

Organising Committee

Dominika Draesslerová	Tomáš Pecka	Jan Trávníček
Ondřej Guth, *Co-chair*	Josef Erik Sedláček	Jan Žďárek
Jan Holub, *Co-chair*	Regina Šmídová	

External Referee

Arnaud Lefebvre

Table of Contents

Invited Talk

Contributed Talks

The Discreet Charm of Multi-Pattern Codes
(*Abstract*)

Igor Zavadskyi

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

Probably the most important trade-off in data compression is between the compression ratio and code processing speed. When it comes to compression ratio, methods approaching the theoretical bound of entropy encoding have been known for decades, such as optimal arithmetic encoding and quasi-optimal Huffman codes. The latter codes can be processed times faster, while their compression efficiency can vary from optimal to significantly suboptimal depending on the properties of the source alphabet. In 2014, J. Duda et al. proposed the encoding based on Asymmetric Numeral Systems, providing a compromise solution nearly as good as arithmetic encoding in terms of compression ratio and nearly as fast as Huffman codes.

Yet, these impressive solutions tend to overshadow codes that prioritize fast decoding, leaving many intriguing questions unanswered in this less-explored domain. Can we develop a code that can be processed significantly faster than Huffman codes? If so, what are the trade-offs? What specific structural properties of a codeword set influence the speed of decoding? For instance, a step structure of codeword length distribution may accelerate code processing. In byte-aligned codes (ETDC, SCDC, or RPBC), invented in the early 2000s, codewords are composed of whole bytes and thus can be processed easily and quickly. However, this is achieved at the cost of a 12–15% loss in compression ratio. Fibonacci codes are not so stepped, thus becoming denser but slower.

Our presentation delves into a series of recently developed multi-pattern variable-length data compression codes. While trading a small percentage of compression ratio, these codes offer an order-of-magnitude acceleration of code processing, surpassing both byte-aligned and Fibonacci codes in terms of compression efficiency and processing speed. We explore the structural, algorithmic, and technical aspects of fast decoding and showcase several other benefits of these innovative codes, including the potential for fast Boyer-Moore-style search in a compressed file and the ability to represent integer sequences in a space-efficient manner with nearly constant time direct access.

Igor Zavadskyi: The Discreet Charm of Multi-Pattern Codes, p. 1.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

Fast Matching Statistics for Sets of Long Similar Strings

Zsuzsanna Lipták[1], Martina Lucà[1], Francesco Masillo[1], and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Verona, Italy
zsuzsanna.liptak@univr.it
martina.luca@studenti.univr.it
francesco.masillo@univr.it
[2] Department of Computer Science, University of Helsinki, Finland
simon.puglisi@helsinki.fi

Abstract. Matching statistics (*MS*) computation is at the heart of numerous bioinformatics applications, from read alignment to computing phylogenies of a set of genomes or even speeding up the computation of core data structures on collections of genomes. Many of these datasets have the property of being highly similar to the reference, which itself, however, may not be very repetitive. Some heuristics based on sequence-to-sequence similarity have already been studied in [Lipták et al., Alg. Mol. Biol. 2024], leading to a significant speedup in the computation of the matching statistics. In this paper, we introduce a new heuristic that further speeds MS computation. The core idea is to take advantage of existing similarities between the input sequences and the reference. We give an implementation making use of this heuristic, which also allows the use of multiple threads to parallelize *MS* computation. We give an experimental evaluation of our tool, `LRF-ms`, comparing it to other *MS* computation tools, on publicly available genomic datasets, and show that it is the fastest when the collection of genomes is highly similar to the reference string, while keeping a comparably low memory footprint.

Keywords: matching statistics, suffix array, parallel algorithms, LCP-array

1 Introduction

Given two strings S and R, over the same alphabet Σ, the matching statistics of S relative to R consists of an array of length $|S|$ whose ith entry contains the length of the longest substring of S starting in position i which has an occurrence also in R. Matching statistics were introduced by Chang and Lawler [11] in 1994 as an algorithmic tool for approximate pattern matching. It has since found many applications, for example, in DNA chip design [37], computation of string kernels [40,34], whole-genome phylogenies [12,44], and detection of SNVs or sequencing errors in read collections [35]. The classic book by Gusfield [21] contains several applications of matching statistics, among these longest common substrings, exact matches, longest prefix matching, and several others. Matching statistics have recently been used as a bridge to faster computation of the suffix array and Burrows-Wheeler Transform (BWT) [9] of string collections [27,28,30].

In contemporary genomics, a common type of dataset consists of many similar copies of essentially the same string. Most recent sequencing projects no longer aim to identify the genomic sequence of a species, but to identify the biological variation of individuals of a given species, such as the 100K Human Genome Project [43], the 1001 Arabidopsis Project [42], and the 3,000 Rice Genomes Project (3K RGP) [39]. Thus, the typical type of biological dataset no longer consists of one very long sequence

Zsuzsanna Lipták, Martina Lucà, Francesco Masillo, Simon J. Puglisi: Fast Matching Statistics for Sets of Long Similar Strings, pp. 3–15.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

(or even one genome consisting of several chromosomes), but of a large number of highly-similar seqeuences.

Recently, a subset of the current authors introduced two heuristics for computing the matching statistics, which proved highly effective, and were therefore suggested to be of independent interest [27,28]. In this paper, we explore these heuristics and add another, which we show to be even more successful in speeding up computation. All of these heuristics exploit the fact that most datasets are highly repetitive *in a particular manner*, namely they consist of many similar copies of essentially the same string. Thus, while the collection of strings S is very similar to the reference string R, the reference string itself may not necessarily be very repetitive. This reflects itself on a fairly low maximal LCP-value (for precise definitions, see Section 2) on the reference string, while resulting in very long matches between S and R.

We implemented this heuristic in a tool for matching statistics computation. We compare our tool, `LRF-ms`, to other publicly available tools for matching statistics computation, on different datasets with varying properties. Surprisingly, we find that our tool outperforms the others by orders of magnitude w.r.t. running time, while also using less or comparable amount of peak memory.

The paper is organized as follows. In Section 2, we introduce the necessary definitions and terminology, in Section 3, we briefly recall matching statistics computation using the suffix array, followed by a presentation of the different heuristics. In Section 4, we give details of our implemention, followed by the results of the experiments in Section 5, comparing the different heuristics, and comparing our tool to competitor tools. We close with future work in Section 6.

2 Basics

Let Σ be an ordered alphabet of size σ. A string T over Σ is a finite sequence of characters from Σ. The ith character of T is denoted $T[i]$, its length is $|T| = n$, and $T[i..j]$ denotes the substring $T[i] \cdots T[j]$. If $i > j$, then $T[i..j]$ is the empty string ϵ. The suffix $T[i..] = T[i..n]$ is referred to as the ith suffix $suf_i(T)$, and $T[..i] = T[1..i]$ is the ith prefix $pref_i(T)$. When T is clear from the context, we write suf_i for $suf_i(T)$. We assume that the last character of T is the *sentinel character*, denoted \$, which does not occur elsewhere in the string and is assumed to be smaller than all characters from Σ. Note that we index strings from 1.

The *suffix array* SA of a string T is a permutation of the set $\{1, \ldots, n\}$ such that $SA[i] = j$ if $suf_j(T)$ is the ith in lexicographic order among all suffixes. For a substring Y of T, all suffixes prefixed by Y appear contiguously in SA; the interval $[s, e]$ of the SA containing all occurrences of Y is called *Y-interval* or *SA-range of Y*. It is well known that the suffix array can be computed in linear time [22,24,25] (see also the classic survey [36] and [3,7] for more recent overviews). Some recent algorithms include [32,31,4,19,26], with SA-IS [32] by far the most popular linear-time SACA, being both simple and fast in practice.

The *inverse suffix array* ISA is the inverse permutation of SA, namely, for all $1 \leq i \leq n$, $ISA[SA[i]] = i$. The *longest common prefix* (*lcp*) of two strings T and S is the longest string U which is a prefix of both T and S. The *longest-common-prefix array* LCP is another array closely related to the SA. It is given by: $LCP[1] = 0$, and for $i > 1$, $LCP[i]$ is the length of the longest common prefix of the two suffixes $suf_{SA[i-1]}$ and $suf_{SA[i]}$, which are consecutive in the SA. The LCP-array can be also be computed in linear time [23].

Let R and S be two strings, and let us denote the sentinel character of R as # and of S as \$, where # < \$. The *matching statistics* [11] MS of S w.r.t. R is an array of length $|S|$, whose entries are integer pairs defined as follows. For $1 \leq i \leq |S|$, let U_i be the longest prefix of suffix $suf_i(S)$ which occurs as a substring in R; we refer to U_i as the *matching factor* at position i. Then $MS[i] = (p_i, \ell_i)$, where $\ell_i = |U_i|$, and p_i is an occurrence of U_i in R if $U_i \neq \epsilon$, and $p_i = -1$ otherwise.

For an integer array A of length n and an index i, the *previous smaller values* PSV and *next smaller values* NSV are defined as follows: $PSV(A, i) = \max\{i' < i \ : \ A[i'] < A[i]\}$, $NSV(A, i) = \min\{i' > i \ : \ A[i'] < A[i]\}$, where $\min \emptyset = -\infty$ and $\max \emptyset = +\infty$. It is known that there is a data structure of size $n \log(3 + 2\sqrt{2}) + o(n)$ bits which answers both PSV and NSV queries in constant time and can be built in $\mathcal{O}(n)$ time [15]. A similar type of query, which we refer to as *extended PSV* and *extended NSV*, can be defined as follows: Let x be an integer and define $PSV(A, i, x) = \max\{i' < i \ : \ A[i'] < x\}$ and $NSV(A, i, x) = \min\{i' > i \ : \ A[i'] < x\}$, the previous respectively next smaller values *with respect to* x. As shown in [10], there is a data structure of size $16n/(2^B)$ bytes, which can be built in time $\mathcal{O}(n)$ and can answer these queries in $\mathcal{O}(B \log \frac{n}{B})$ time, where B is a parameter.

Given an integer array A of length n and two indices $1 \leq i \leq j \leq n$, a *range minimum query (RMQ)* on A is defined as $RMQ(A, i, j) = \arg\min\{A[k] \leq i \leq k \leq j\}$. Thus, an RMQ-query returns some position of the minimum value in the subarray $A[i..j]$. A data structure capable of answering RMQ-queries in constant time can be built in linear time and takes $2n + o(n)$ bits of space [14].

3 Matching Statistics computation

In the original publication introducing the matching statistics [11], Chang and Lawler gave a construction algorithm using the suffix tree of R, with space usage $\mathcal{O}(|R|)$ and running time $\mathcal{O}(|R| + |S| \log \sigma)$. Since then, it has been shown how to construct the matching statistics efficiently using compressed text indexes instead of the suffix tree [17,1,34,6].

In this section, we will first recall how to compute the matching statistics of a string S w.r.t. a reference string R using the suffix array. Then we will briefly explain two heuristics introduced in [27,28] (the MaxLCP and Block heuristics) and present a refinement which we refer to as LRF-heuristic. As we will see in Section 5, this latter heuristic far outperforms our previous heuristics, as well as all competing tools.

3.1 Plain version

In order to compute the matching statistics of S w.r.t. R, we will use SA_R, ISA_R, LCP_R, and the data structure of [10] (a heap-type tree) for extended *PSV-NSV*-queries on LCP_R (see Sec. 2). Constructing these data structures takes overall $\mathcal{O}(|R|)$ time and $\mathcal{O}(|R|)$ space.

In the following, we will use the terminology from [27,28]:

Definition 1 ([27,28]). *For a character c and a string Y, the computation of the Yc-interval from the Y-interval is called a* right extension *and the computation of the Y-interval from cY-interval is called a* left contraction.

A left contraction corresponds to following an implicit suffix link in the suffix tree of the text. In the following lemma, we show how to compute a left contraction using our data structures.[1]

Lemma 2. *Let cY be a substring of string T with SA-range $[s, e]$ and let $s' = ISA[SA[s] + 1]$ and $e' = ISA[SA[e] + 1]$. Then the SA-range of Y is $[x, y]$, where $x = PSV(LCP, s', |Y|)$ and $y = NSV(LCP, e', |Y|) - 1$.*

Proof. Let $i = SA[s]$ and $j = SA[e]$, thus both $suf_i(T)$ and $suf_j(T)$ have cY as prefix. Therefore, the following suffixes, $suf_{i+1}(T)$ and $suf_{j+1}(T)$ have Y as prefix, and thus both $s' = ISA[SA[s] + 1] = ISA[i + 1]$ and $e' = ISA[SA[e] + 1] = ISA[j + 1]$ lie in the SA-range $[x, y]$ of Y. Since $s \leq e$, thus $suf_i(T) \leq suf_j(T)$, and since both start with the same character c, therefore also $suf_{i+1}(T) \leq suf_{j+1}(T)$, implying $s' \leq e'$. Since both have prefix Y, it thus holds for all $s' \leq k \leq e'$ that $suf_k(T)$ has Y as prefix, i.e. k is an occurrence of Y. However, there could be other occurrences of Y whose corresponding suffixes are lexicographically smaller than suf_{i+1} resp. greater than suf_{j+1}. Now notice that for any two occurrences k, k' of Y, it holds that $lcp(suf_k(T), suf_{k'}(T)) \geq |Y|$. Therefore, the first position in which the LCP-value falls below $|Y|$ gives the extremities of the Y-interval of SA. Since $LCP[k]$ is defined as the lcp between the suffixes in position k and $k - 1$ of the SA, we get that $x = PSV(LCP, s', |Y|)$ gives us the position in the SA of the lexicographically smallest occurrence of Y, i.e. the one listed first in the SA, while $y = NSV(LCP, e', |Y|) - 1$ gives us the last one. □

The computation of each entry of the matching statistics can thus be seen as consisting of two distinct phases. Given entry $MS[i]$ and the U_i-interval $[s_i, e_i]$ of SA_R, we can split the computation of the next entry $MS[i + 1]$ into:

1. *left contraction:* Compute the SA-interval $[x, y]$ of U', where $U_i = cU'$ for character $c = S[i]$, and
2. *right extension:* Extend the factor U' to the right as long as the current prefix of $suf_i(S)$ occurs in R. The result is the range $[s_{i+1}, e_{i+1}]$ of the matching factor U_{i+1} (of which U' is a prefix).

Phase 1. Given the range $[s_i, e_i]$ for position i, we want to compute the range $[x, y]$ of factor U'. By Lemma 2, this interval equals

$$[x, y] = [PSV(LCP_R, s', |Y|), NSV(LCP_R, e', |Y|) - 1],$$

where $s' = ISA_R[SA_R[s_i] + 1]$ and $e' = ISA_R[SA_R[e_i] + 1]$.

Phase 2. The second phase can be done by searching for the longest prefix of $suf_i(S)$ which occurs in R, with two applications of binary search on SA_R, one for the left extremity of the interval, and one for the right one. The range is a pair of indices $[s_{i+1}, e_{i+1}]$, where $SA[k]$, for $s_{i+1} \leq k \leq e_{i+1}$, are all occurrences of U_{i+1} in R. We then set $MS[i + 1] = (s_{i+1}, |U_{i+1}|)$.

The total worst-case running time of the algorithm for computing the matching statistics is $\mathcal{O}(|S| \log |R|)$, due to the binary search in the right extension phase, and choosing the parameter $B = \mathcal{O}(1)$ for the extended PSV-NSV data structure.

[1] Note that this lemma is slightly different from the formula given in [27,28].

3.2 The *LRF*-heuristic

The two heuristics given in [27,28] both aimed at speeding up *MS*-computation using information stored in the *LCP*-array.

1. MaxLCP-heuristic: Compute $L = \max\{LCP[j] \mid 1 \leq j \leq |R|\}$, and for each $1 \leq i \leq |S|$ compare ℓ_i with L. If $\ell_i - 1 > L$, then this means that we only need a left contraction, i.e. $ISA[p_i + 1]$, and no right extension is necessary, leading to $MS[i+1] = (p_i + 1, \ell_i - 1)$. This is because the interval containing the matching factor U_{i+1} is a singleton interval in the SA_R (in other words, it can be viewed as a leaf branch in the suffix tree of R). Hence, there are no further occurrences in R of $S[i+1..\ell_i - 2]$ other than $p_i + 1$.
2. Block-heuristic: This is a further refinement of the MaxLCP-heuristic: Divide the *LCP*-array into blocks of size b and compute the maximum of each block. Locate the block that contains $SA[p_i]$ and compare ℓ_i to this value.

Here, we propose to further refine the second heuristic by noticing that by taking the extreme value of $b = 1$, one ends up having to query the *LCP* array directly, or rather, two consecutive values of the *LCP*-array (see Observation 1). But the access pattern to the *LCP* array for contiguous text positions is not cache-friendly, because the *lcp* information is in *SA* order rather than in text-order. Therefore, we introduce the following data structure:

Definition 3. *Let T be a string of length n over Σ. Define the* longest repeated factor array *LRF: $LRF[i] = \max\{|Y| : Y$ occurs both in position i and some $k \neq i$ in $T\}$.*

Note that the *LRF*-array is distinct from the well-known *LPF*-array [33], used, e.g. for computation of the LZ77 parsing of a string T, as that array contains the length of the longest *previous* factor, i.e. the longest factor occurring both in i and in some position $k < i$, while here, we want the longest factor occurring both in i and in *any* position $k \neq i$. It is easy to compute the *LRF*-array in linear time, using the following observation:

Observation 1 *For all i, $LRF[i] = \max\{LCP[ISA[i]], LCP[ISA[i]+1]\}$.*

Proof. Since $LCP[k]$ gives the longest common prefix of the suffix in position k and in position $k-1$, it follows that the longest repeated factor of the suffix which is position k is the maximum of $LCP[k]$ and $LCP[k+1]$, implying the claim. □

Since the *LCP*-array can be computed in linear time, e.g. using the algorithm by Kasai et al. [23], we can also compute the *LRF*-array in linear time. Alternatively, we can use a linear-time algorithm for computing the *PLCP*-array (*permutated LCP-array*) of [22], which is defined by $PLCP[i] = LCP[ISA[i]]$, i.e. the *LCP*-values given in text-order.

Now, checking whether we are in a singleton interval inside a leaf branch can be done by comparing $\ell_i - 1$ to $LRF[p_i + 1]$. The strength of this approach can be appreciated when we have consecutive positions in S that satisfy this check, leading to a very cache-efficient sequential access pattern in the *LRF*-array. A brief pseudocode for this heuristic is given in Algorithm 1.

An experimental evaluation of this heuristic against the other two showed that the *LRF*-heuristic far outperformed both the MaxLCP heuristic and the Block heuristic for all block sizes, see Figure 1.

Algorithm 1: *LRF*-heuristic

```
while ℓ_i − 1 > LRF[p_i + 1] do
    MS[i + 1] = (p_i + 1, ℓ_i − 1)
    i + +
end
```

4 Implementation details

To compute SA_R, LCP_R, and $PLCP_R$ we use a highly engineered version of the SA-IS algorithm [32], `libsais` [20]. We implemented the data structure for *PSV-NSV* and range minimum queries on the LCP_R, based on the work of Cánovas and Navarro [10], setting $B = 7$. We preprocessed the data to have only characters in the set $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. We did this by assigning to IUPAC characters one among their possible corresponding nucleotides. We use this assumption in several of our experiments later described in this section, and to allow all tools to run without crashing (as some assume that all characters present in the dataset S are also present in the reference R).

We implemented the two heuristics introduced in [27,28] (see Sec. 3.2). For the second one, which involves choosing a block size for computing local maxima, we tested $b = 512, 1024, 2048, 4096$. In a further step, we parallelized our *LRF*-based implementation by splitting the collection to evenly distribute the number of sequences to give to each thread.

We tested various other approaches to further increase efficiency, including:

- a k-mer lookup table ($k = 8$), where we have a precomputed range for every possible k-mer in R. With the assumption of $\sigma = 4$, we can use just two bits to represent each letter of a k-mer, allowing us to use a lookup table with 2^{16} entries, when choosing $k = 8$. We access the lookup table when ℓ drops below the chosen value of k;
- a cache, implemented as a hashtable, storing resulting ranges during right-extensions. The hashtable has (sp, ep, l, c) as key, where sp and ep are the start, respectively the end, of the initial binary search range, l is the length of the currently matched prefix, and c is the character to be found after applying the binary search procedure. The values stored for each key are (sp', ep'), which are the start, respectively the end, of the new range of the binary search. We also tried to store in l the value of c in base 4, taking only the two most significant bits of l. We set 100000 elements as the maximum capacity of these hashtables;
- we attempted to trade off some running time by approximating the *LCP* array using only $\log \log n$ bits, so effectively reducing the integers to two bytes instead of four. This effect is also reflected on the *RMQ* and *PSV-NSV* data structures.
- we implemented the improved version of the binary search procedure on *SA*, which uses *RMQ* queries on LCP_R, allowing to compute right-extensions in overall $\mathcal{O}(|S| + \log |R|)$ time. For the *RMQ* data structure, we tested all the available plug-in options in the SDSL [18], plus the data structure based on [10]. The latter achieved the best performance in our tests, probably because it was already in memory due to suffix link traversal.

These four approaches did not lead to further significant improvement in time and/or space (data not shown).

5 Experiments

We implemented our algorithm for computing the *MS* in `C++`, resulting in the tool `LRF-ms`. Our implementation is available at `https://github.com/fmasillo/lrf-ms`. The experiments were conducted on a desktop equipped with 64GB of RAM DDR4-3200MHz and an Intel(R) Core(R) i9-11900 @ 2.50GHz (with turbo speed @ 5GHz) with 16 MB of cache. The operating system was Ubuntu 22.04 LTS, the compiler used was `g++` version 11.3.0 with options `-std=c++20 -O3 -march=native` enabled.

5.1 Datasets

In our experiments, we used four publicly available datasets. The first dataset, which we refer to as `sars-cov2`, contains copies of SARS-CoV2 genomes taken from the COVID-19 Data Portal. The second dataset (`chr19`) consists of copies of the Human Chromosome 19 from the 1000 Genomes Project [41]. The third dataset (`salmonella`) is a collection of assembled genomes of Salmonella enterica, downloaded from NCBI Pathogens website[2]. Lastly, the fourth dataset, which we refer to as `rice`, contains variants of Chromosome 1 of the Nipponbare reference sequence (rice) downloaded from RiceVarMap [45] as vcf files[3]. For each dataset we have a collection of sequences of length 1 GB. Some additional metadata can be found in Table 1.

name	no. sequences	ref. seq. length	max LCP_R	mean LCP_R
`sars-cov2`	36 201	29 783	17	6
`chr19`	17	59 126 939	3 099 999	81 379
`salmonella`	216	4 506 055	145	10
`rice`	23	43 992 113	12 893	27

Table 1. Datasets used in the experiments. In column 2 the number of sequences in the dataset, in column 3 the reference sequence length, in column 4 the maximum value in LCP_R, and in column 5 the average of the values in LCP_R. The total dataset has size 1 GB and σ is always 4.

5.2 Other tools

We compared our implementations to the following four tools:

1. `indexed_ms` [13], a tool computing a compact version of matching statistics. At its core, it uses a compressed suffix tree of the forward and reverse reference string along with some further annotation containing a list of nodes of the suffix tree topology that are maximal repeats. This tool outputs a compact version of matching statistics. We ran the experiments with the following flags enabled: `-load_cst 1 -load_maxrep 1 -lazy_wl 1 -nthreads 16`.
2. `MONI` [38], builds an enhanced r-index [16,5] on the reference string. Other than the Burrows-Wheeler Transform of R and the suffix array samples at run-boundaries, the tool also computes some additional $\mathcal{O}(r)$ space information, called thresholds, to enable fast recovery in case of a mismatch. It also makes use of a grammar to allow fast random access to the original text. We ran the experiments with the following flags enabled: `-t 16 -g shaped`.

[2] `https://www.ncbi.nlm.nih.gov/pathogens/isolates/` with isolate number PDS000065758.863

[3] `https://ricevarmap.ncpgr.cn/download/`

3. `PHONI` [8], is another tool based on the r-index, but it does not need thresholds. This is done via a refinement of the process of finding the length of the matches with a grammar enabling fast longest common extension (LCE) queries. This enables a single scan of the pattern to compute the matching statistics. We ran the experiments with the following flag enabled: `-g shaped`.
4. `AUG-PHONI` [29], a tool based on `PHONI`. It stores thresholds for LCE, bypassing queries to the grammar, resulting in a faster computation, with a slight increase in space. We ran the experiments with the following flag enabled: `-g shaped`.

5.3 Running time and peak memory results

In this section, we first compare the different implementations of our algorithm using different heuristics. Then, we will show how the best implementation of our tool compares to the other state-of-the-art tools for computing matching statistics.

Comparison of different heuristics. Looking at Figure 1, we can see that, overall, the fastest implementation is the one making use of the LRF heuristic. More in detail, in both `chr19` and `rice` datasets we can see a large improvement between using the LRF heuristic and the fastest of the block-based heuristic, taking approximately a third of the time. In `sars-cov2` and `salmonella` datasets, we can see that using even the simplest heuristic, i.e. taking only the maximum value in the LCP, leads to a big speedup w.r.t. not using a heuristic at all. This is because, as reported in Table 1, the max LCP value is quite small, enabling a lot of skipping in the sequence of left-contractions even without using the exact value of LCP/LRF associated with p_i. `sars-cov2` is the only dataset in which we have a slight worsening due to the overhead of making a cache-miss when accessing an entry of LRF instead of comparing to a variable holding the max LCP.

Comparison with other tools. We have divided the comparison of `LRF-ms` and the other state-of-the-art tools based on whether they support multi-threading when computing MS. In Figure 2, we compare `LRF-ms` with `indexed_ms` and `MONI`. Over the four datasets, `LRF-ms` is always the winner, followed by `indexed_ms`. On `chr19` and `rice` where the difference is largest, `LRF-ms` is around three times faster, while on `sars-cov2` and `salmonella` is around 1.5 to 2 times faster than `indexed_ms`. `MONI` takes, on average, 6.5 times more time than `LRF-ms`. Moving to Figure 3, we compare to `PHONI` and `AUG-PHONI`, which allow only for serial computation of MS. `AUG-PHONI` and `PHONI` are 32-52 respectively 44-382 times slower than `LRF-ms`.

The running results reported here are in line with the expectations on the indexes used for R. Our implementation makes use of mostly uncompressed data structures, so the overhead for each step of the computation is set to a minimum. `indexed_ms` uses suffix tree topology over the Burrows-Wheeler Transform (BWT) [9] of R, which is supposedly smaller than the total size of our data structures. Using compact data structures also implemented in SDSL, this tool is always between `LRF-ms` and the rest of the tools when comparing running time. `MONI`, `PHONI`, and `AUG-PHONI` are all based on the r-index, and therefore, due to the compressed nature of such index, the running time is affected negatively. In general, BWT-based approaches require a procedure called LF-mapping based on rank queries. LF-steps are known to be not very cache-efficient due to unpredictable jumps in the data structure. Here, extra work must be performed to recover from a mismatch during an LF-step, lacking the explicit suffix

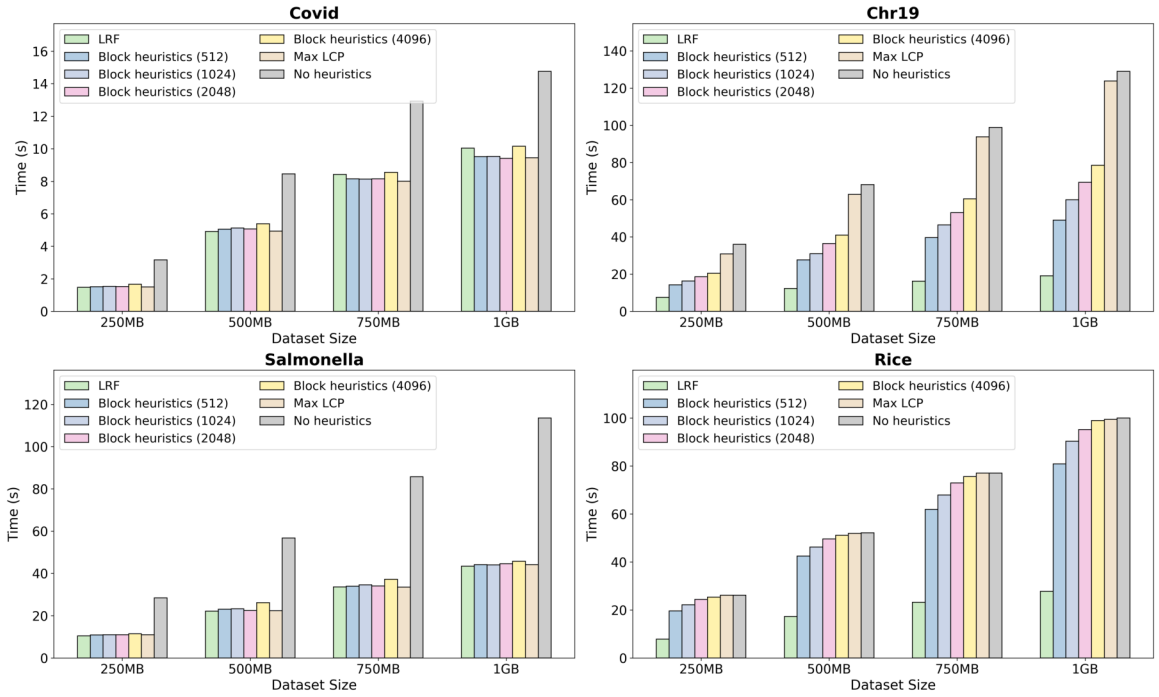

Figure 1. Comparison of different heuristics using as a base implementation the plain binary search version. Every subfigure represents a different dataset, and on the x-axis the size of the dataset is shown, while on the y-axis the running time is reported in seconds.

tree topology. On the other hand, both theoretically and experimentally, the index takes $\mathcal{O}(r)$ space, where r is the number of runs in the BWT of R, a well-known repetitiveness measure. (Here a *run* is defined as a maximal substring consisting of the same character.) This ensures the possibility of computing the index for extremely large references, as reported in each paper.

Looking at memory consumption, we divided the comparison into two phases: indexing of R and computation of *MS*. In Table 2, the peak memory in kilobytes is measured for the indexing phase. Across the four datasets, our tool uses the least memory overall. On `sars-cov2`, `LRF-ms` uses less than half the memory compared to the other tools due to $|R|$ being extremely small. Likewise, on `salmonella`, we take less space than the competitors, a third less in general. On `chr19` and `rice`, we take almost the same space as `indexed_ms` which is around a third less than the other three tools. Moving to Table 3, we recorded the peak memory of *MS* computation on 1GB of data for each dataset, using 16 threads if the tool can run in multi-threaded mode. We consistently use an additional 100MB of memory w.r.t. the indexing phase due to buffering I/O operations, which are the main bottleneck on `sars-cov2` and `chr19` datasets. On average, `indexed_ms` seems to be settling around 650 MB of internal memory, while `MONI` has a particular behaviour, probably related to the length of the individual sequences in the dataset. For example, on `chr19`, if we set the execution to single-threaded, the memory consumption drops to 1300 MB (data not shown) from around 13 GB when using 16 threads. Overall, in our setting, `LRF-ms` performs comparably well for space consumption, for example, being the lightest on `salmonella` and close to `indexed_ms` on `rice`.

In general, indexing a moderate-size reference with compressed or uncompressed data structures leads to similar memory consumption (due to the incompressibility of

the reference). When computing the *MS*, `indexed_ms` seems to take more space the more threads are allocated, as for `MONI`, but only around 40 MB per thread. Looking at r-index-based tools, even though the space is proportional to r, a single sequence is not so repetitive. The complex machinery used in these tools takes more space than our set of data structures when loaded in memory, as can be seen for every dataset except `sars-cov2`.

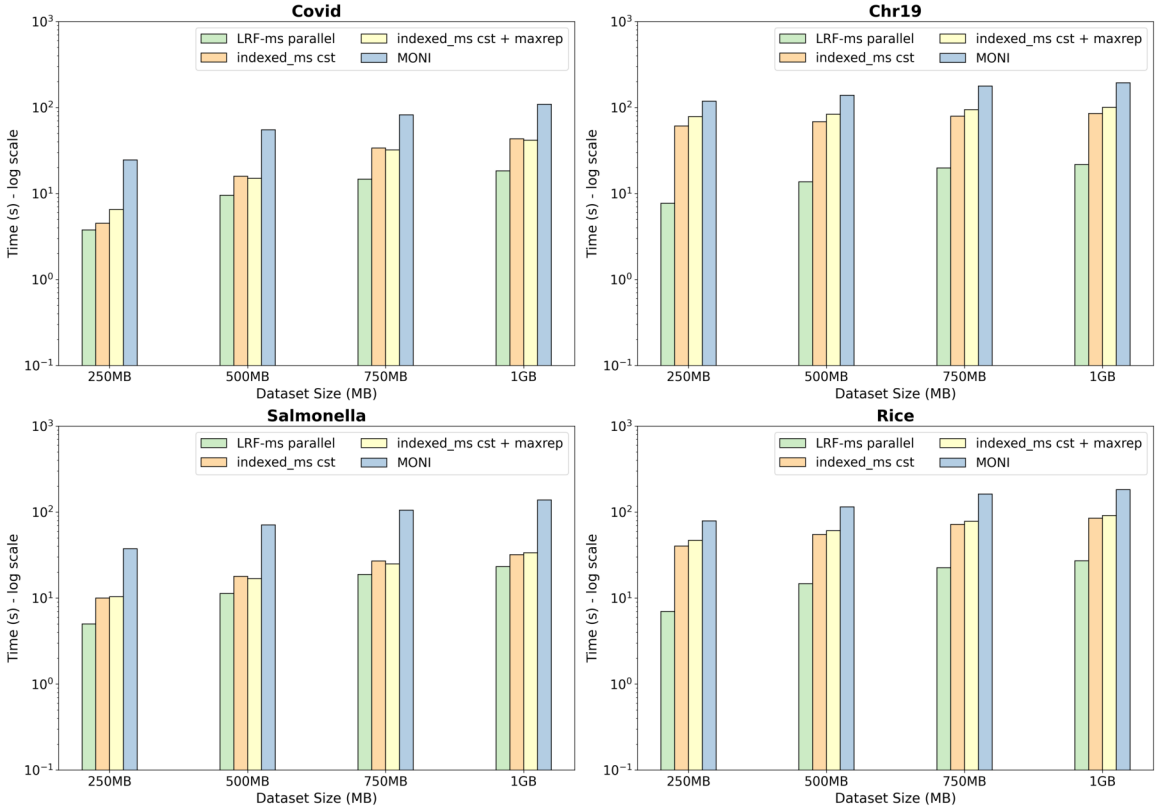

Figure 2. Comparison of different multi-threaded tools using 16 threads. Each subfigure represents a different dataset. We give the dataset size on the x-axis and the time (in seconds) on the y-axis (log-scale).

	LRF-ms	indexed_ms	MONI	PHONI	AUG-PHONI
sars-cov2	**4 992**	13 488	15 052	14 524	14 640
chr19	**991 240**	1 003 788	1 428 540	1 428 536	1 428 536
salmonella	**80 380**	102 996	112 760	112 760	112 760
rice	727 924	**709 864**	1 039 352	1 039 344	1 039 348

Table 2. Peak memory (in KB) of the different tools for indexing the reference string R. The lowest value for each row is highlighted in bold.

6 Conclusion

In this paper, we have presented a new simple and practical heuristic for speeding up the computation of matching statistics. This new heuristic exploits the similarity between the reference string and the individual strings in the collection. Compared to the previous best heuristic, our approach more often avoids the costly operation

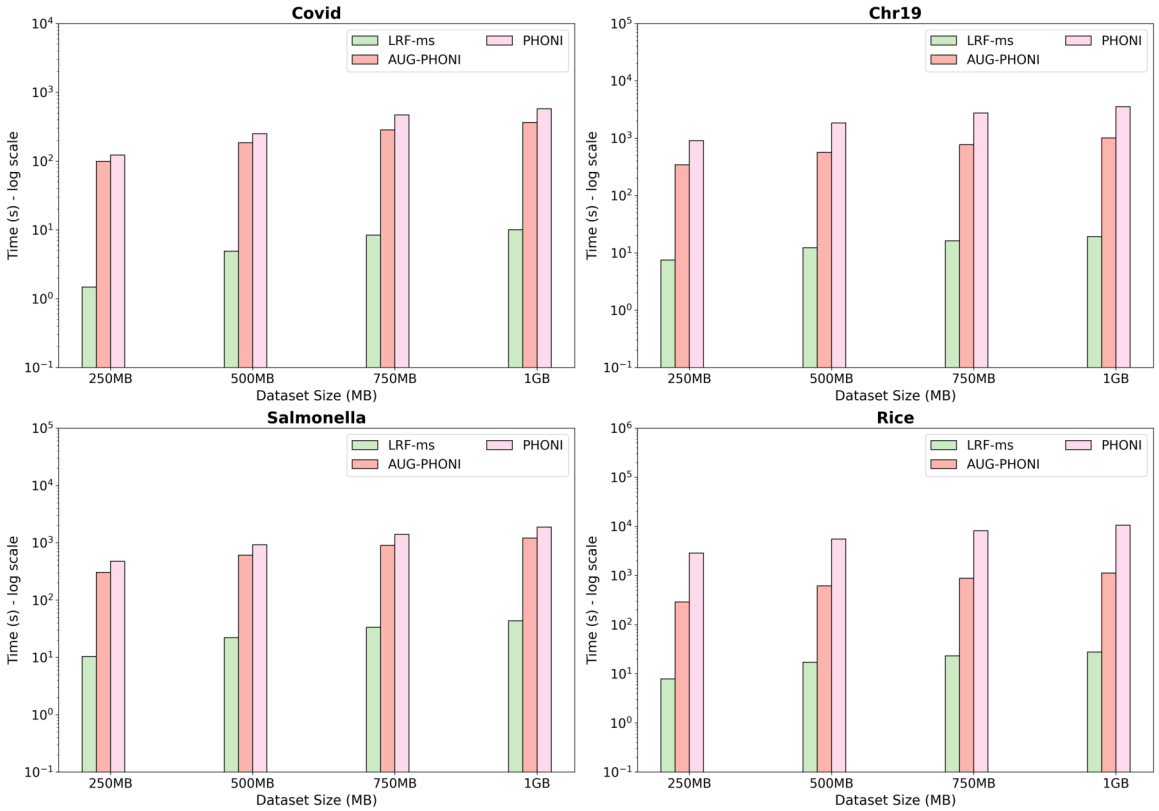

Figure 3. Comparison of different single-threaded tools. Each subfigure represents a different dataset. We give the dataset size on the x-axis and the time (in seconds) on the y-axis (log-scale).

	LRF-ms	indexed_ms	MONI	PHONI	AUG-PHONI
sars-cov2	90 716	617 820	17 044	**14 168**	14 180
chr19	1 077 920	**683 504**	13 078 520	2 175 716	2 393 404
salmonella	**166 552**	635 368	1 269 008	181 620	202 240
rice	814 900	**673 916**	10 859 492	1 628 136	1 816 560

Table 3. Peak memory (in KB) of the different tools for computing the matching statistics of the string collection w.r.t. R. The lowest value for each row is highlighted in bold.

of traversing a suffix link. The experimental evaluation against the state-of-the-art tools shows that our implementation `LRF-ms` is the winner in terms of running time, while having comparable space consumption.

Future work will focus on exploring the possibility of using the proposed heuristic on a more compact version of the text index of R, be it a compressed suffix tree or a variant of the r-index, called $\overline{r}$-index, which does not use backward search [2], possibly allowing a right-extension-like operation then followed by fast left-contractions.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* J. Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. O. Ahmed, A. Baláz, N. K. Brown, L. Depuydt, A. Goga, A. Petescia, M. Zakeri, J. Fostier, T. Gagie, B. Langmead, G. Navarro, and N. Prezza: *r-indexing without backward searching.* CoRR, abs/2312.01359 2023.
3. J. Bahne, N. Bertram, M. Böcker, J. Bode, J. Fischer, H. Foot, F. Grieskamp, F. Kurpicz, M. Löbel, O. Magiera, R. Pink, D. Piper, and C. Poeplau: *Sacabench:*

Benchmarking suffix array construction, in Proc. of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019), vol. 11811 of Lecture Notes in Computer Science, Springer, 2019, pp. 407–416.

4. U. Baier: *Linear-time suffix sorting - A new approach for suffix array construction*, in Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), vol. 54 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 23:1–23:12.
5. H. Bannai, T. Gagie, and T. I: *Refining the* r*-index*. Theor. Comput. Sci., 812 2020, pp. 96–108.
6. D. Belazzougui, F. Cunial, and O. Denas: *Fast matching statistics in small space*, in Proc. 17th International Symposium on Experimental Algorithms (SEA), vol. 103 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 17:1–17:14.
7. T. Bingmann: *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools*, PhD thesis, Karlsruhe Institute of Technology, Germany, 2018.
8. C. Boucher, T. Gagie, T. I, D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, and M. Rossi: *PHONI: streamed matching statistics with multi-genome references*, in Proc. of the 31st Data Compression Conference (DCC 2021), IEEE, 2021, pp. 193–202.
9. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., DIGITAL System Research Center, 1994.
10. R. Cánovas and G. Navarro: *Practical compressed suffix trees*, in Proc. of the 9th International Symposium Experimental Algorithms, SEA 2010, vol. 6049 of LNCS, Springer, 2010, pp. 94–105.
11. W. I. Chang and E. L. Lawler: *Sublinear approximate string matching and biological applications.* Algorithmica, 12(4/5) 1994, pp. 327–344.
12. E. Cohen and B. Chor: *Detecting phylogenetic signals in eukaryotic whole genome sequences.* J. Comput. Biol., 19(8) 2012, pp. 945–956.
13. F. Cunial, O. Denas, and D. Belazzougui: *Fast and compact matching statistics analytics.* Bioinform., 38(7) 2022, pp. 1838–1845.
14. J. Fischer: *Optimal succinctness for range minimum queries*, in Proc. of the 9th Latin American Symposium on Theoretical Informatics (LATIN 2010), vol. 6034 of Lecture Notes in Computer Science, Springer, 2010, pp. 158–169.
15. J. Fischer: *Combined data structure for previous- and next-smaller-values.* Theor. Comput. Sci., 412(22) 2011, pp. 2451–2456.
16. T. Gagie, G. Navarro, and N. Prezza: *Fully functional suffix trees and optimal text searching in BWT-runs bounded space.* J. ACM, 67(1) 2020, pp. 2:1–2:54.
17. Y. Gao: *Computing matching statistics on repetitive texts*, in Proc. of the 32nd Data Compression Conference (DCC 2022), IEEE, 2022, pp. 73–82.
18. S. Gog, T. Beller, A. Moffat, and M. Petri: *From theory to practice: Plug and play with succinct data structures*, in Proc. of the 13th International Symposium on Experimental Algorithms (SEA 2014), vol. 8504 of Lecture Notes in Computer Science, Springer, 2014, pp. 326–337.
19. K. Goto: *Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets*, in Proc. of the Prague Stringology Conference 2019, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019, pp. 111–125.
20. I. Grebnov: *Code for libsais*, `https://github.com/IlyaGrebnov/libsais`.
21. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
22. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* J. ACM, 53(6) 2006, pp. 918–936.
23. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings, vol. 2089 of Lecture Notes in Computer Science, Springer, 2001, pp. 181–192.
24. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time.* J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.
25. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays.* J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
26. Z. Li, J. Li, and H. Huo: *Optimal in-place suffix sorting.* Inf. Comput., 285(Part) 2022, p. 104818.

27. Zs. Lipták, F. Masillo, and S. J. Puglisi: *Suffix sorting via matching statistics*, in Proc. of the 22nd International Workshop on Algorithms in Bioinformatics (WABI 2022), vol. 242 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:15.
28. Zs. Lipták, F. Masillo, and S. J. Puglisi: *Suffix sorting via matching statistics.* Algorithms Mol. Biol., 19(1) 2024, pp. 11:1–11:18.
29. C. Martínez-Guardiola, N. K. Brown, F. Silva-Coira, D. Köppl, T. Gagie, and S. Ladra: *Augmented thresholds for MONI*, in Proc. of the 33rd Data Compression Conference (DCC 2023), IEEE, 2023, pp. 268–277.
30. F. Masillo: *Matching statistics speed up BWT construction*, in Proc. of the 31st Annual European Symposium on Algorithms (ESA 2023), vol. 274 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 83:1–83:15.
31. G. Nong: *Practical linear-time O(1)-workspace suffix sorting for constant alphabets.* ACM Trans. Inf. Syst., 31(3) 2013, p. 15.
32. G. Nong, S. Zhang, and W. H. Chan: *Two efficient algorithms for linear time suffix array construction.* IEEE Trans. Computers, 60(10) 2011, pp. 1471–1484.
33. E. Ohlebusch: *Bioinformatics Algorithms*, Oldenbusch Verlag, 2013.
34. E. Ohlebusch, S. Gog, and A. Kügel: *Computing matching statistics and maximal exact matches on compressed full-text indexes*, in Proc. of the 17th International Symposium on String Processing and Information Retrieval, SPIRE 2010, vol. 6393 of LNCS, Springer, 2010, pp. 347–358.
35. N. Philippe, M. Salson, T. Commes, and E. Rivals: *Crac: an integrated approach to the analysis of rna-seq reads.* Genome biology, 14 2013, pp. 1–16.
36. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39(2) 2007, p. 4.
37. S. Rahmann: *Fast and sensitive probe selection for DNA chips using jumps in matching statistics*, in Proc. of the 2nd IEEE Computer Society Bioinformatics Conference (CSB 2003), IEEE Computer Society, 2003, pp. 57–64.
38. M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher: *MONI: A pangenomic index for finding maximal exact matches.* J. Comput. Biol., 29(2) 2022, pp. 169–187.
39. C. Sun, Z. Hu, T. Zheng, K. Lu, Y. Zhao, W. Wang, J. Shi, C. Wang, J. Lu, D. Zhang, Z. Li, and C. Wei: *RPAN: rice pan-genome browser for 3000 rice genomes.* Nucleic Acids Research, 45(2) 2017, pp. 597–605.
40. C. H. Teo and S. V. N. Vishwanathan: *Fast and space efficient string kernels using suffix arrays*, in Proc. of the 23rd International Conference on Machine Learning (ICML 2006), vol. 148 of ACM International Conference Proceeding Series, ACM, 2006, pp. 929–936.
41. The 1000 Genomes Project Consortium: *A global reference for human genetic variation.* Nature, 526 2015, pp. 68–74.
42. The 1001 Genomes Consortium: *Epigenomic Diversity in a Global Collection of* Arabidopsis thaliana *Accessions.* Cell, 166(2) 2016, pp. 492–505.
43. C. Turnbull et al.: *The 100,000 genomes project: bringing whole genome sequencing to the NHS.* British Medical Journal, 361 2018.
44. I. Ulitsky, D. Burstein, T. Tuller, and B. Chor: *The average common substring approach to phylogenomic reconstruction.* J. Comput. Biol., 13(2) 2006, pp. 336–350.
45. H. Zhao, J. Li, L. Yang, G. Qin, C. Xia, X. Xu, Y. Su, Y. Liu, L. Ming, L.-L. Chen, et al.: *An inferred functional impact map of genetic variants in rice.* Molecular Plant, 14(9) 2021, pp. 1584–1599.

Beyond Horspool: A Comparative Analysis in Sampled Matching

Simone Faro[1], Francesco Pio Marino[1,2], and Andrea Moschetto[1]

[1] Dipartimento di Matematica e Informatica, Università di Catania, viale A.Doria n.6, 95125, Catania, Italia
[2] Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen F-76000, France

Abstract. The *exact online string matching* problem, pivotal in fields ranging from computational biology to data compression, involves identifying all instances of a specified pattern within a text. Despite extensive examination over the decades, this problem has remained computationally challenging due to the time and space limitations inherent in traditional online and offline methods, respectively. Introduced in 1991, *sampled string matching* has now emerged as a groundbreaking approach, ingeniously combining classical online string matching techniques with efficient text sampling methods. This approach not only addresses the spatial constraints of indexed string matching but also significantly reduces the search duration in online environments, achieving speed increases of up to hundreds of times while requiring less than 4% of the text size for its partial index. In this paper, we explore the adaptability of various online string matching algorithms within the framework of sampled string matching, which has traditionally relied on the Horspool algorithm. Our investigation reveals that integrating alternative string matching algorithms as subroutines markedly enhances overall performance. These findings highlight the potential for reevaluating established methodologies in light of newer, more dynamic solutions and set the stage for transformative impacts across multiple domains.

1 Introduction

Given a text y of length n and a pattern x of length m over some alphabet Σ of size σ, the string matching problem consists of finding all occurrences of the pattern x in the text y. String matching is a crucial topic in the broader domain of text processing [6], and algorithms for this problem serve as fundamental components in the implementation of practical software across various operating systems. Although data are stored in various formats, text remains the primary medium for exchanging information. This is particularly evident in the literature of linguistics, where data consist of extensive corpora and dictionaries. Similarly, in computer science, a significant amount of data is stored in linear files. This holds true in fields like molecular biology as well, where biological molecules are often represented as sequences of nucleotides or amino acids.

Applications necessitate two distinct approaches: *online* and *offline* string matching. The former deals with unprocessed text, requiring real-time scrutiny during the search operation. Its worst-case time complexity is $\Theta(n)$, a milestone initially achieved by the well-known Knuth-Morris-Pratt (KMP) algorithm [20]. However, the average time complexity, $\Theta(\frac{n \log_\sigma m}{m})$ [24], was first achieved by the Backward-Dawg-Matching (BDM) algorithm [5]. Many string matching solutions have also been developed in order to achieve sub-linear performance in practical cases [6]. Among them, the Boyer-Moore-Horspool algorithm [2,17] is worth special mention, as it has been the inspiration for much work.

Simone Faro, Francesco Pio Marino, Andrea Moschetto: Beyond Horspool: A Comparative Analysis in Sampled Matching, pp. 16–26.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

Conversely, solutions adopting the second approach aim to expedite searches through preprocessing, constructing data structures that facilitate search operations. Termed *indexed searching*, this methodology consists on various efficient solutions. Notable examples include those leveraging suffix trees [1], boasting a $O(m + occ)$ worst-case time, suffix arrays [21], offering a respectable $O(m + \log n + occ)$ [21], and the FM-index [14] (Full-text index in Minute space), a compressed marvel derived from the Burrows-Wheeler transform, adeptly balancing input compression with swift substring queries. However despite their time performance, full-index data structures such as the ones just mentioned require additional space ranging from 4 to 20 times the size of the original text.

1.1 Sampled String Matching

Another solution in the literature consists in the realm of *Sampled String Matching* algorithms, pioneered by Vishkin in 1991 [23]. This method involves creating a succinct version of the text and then applying online string matching algorithms directly to the new version. This technique enables faster discovery of pattern occurrences, but each discovery within the sampled version of the text requires subsequent verification within the original text. Moreover, the sampled-text approach possesses several characteristics: it typically necessitates straightforward implementation, demands only a limited amount of additional space, and enables fast search and update operations.

Besides Vishkin's theoretical results, a more practical solution of the sampled string matching appeared more recently by Claut *et al.* [4], presenting an alphabet reduction technique (OTS). Their solution requires an extra space of 14% of the original text size, while speeding up the searching procedure up to 5 times traditional online string matching algorithms on English texts. Moreover they also introduced a indexed version of the sampled text, adapting the suffix array by indexing the sampled positions of the text.

More recently, Faro *et al.* have presented several algorithms on the field of sampling, especially with their *Character Distance Sampling* (CDS) approach [8,10,11,12,13]. In practical terms, through sampling absolute positions of some specific characters in the text, called *pivot characters*, their method has reached speedups of up to a factor of 9 on English texts, while demanding a limited additional space, ranging from 11% to 2.8% of the text's size. Achieving a 50% reduction in search times compared to the previous approach (OTS).

1.2 Our Contribution

In this paper, we investigate both the *OTS* and *CDS* algorithms in the context of online string matching. Indeed, both of these algorithms have traditionally been implemented based on the *Horspool* algorithm. We will analyze some practical results regarding the adaptability of these sampling string matching algorithms with various online algorithms. This paper is organized as follows: in Section 2 and Section 3 we briefly talk about the two practical existing sampling methods. In Section 4 we introduce the Online Algorithms selected for this study providing a short description for each of them. Finally in Section 5 we show the experimental results. In Section 6 we draw our conclusions.

2 The Occurrence Text Sampling algorithm

In this section we briefly describe the efficient text-sampling approach proposed by Claude *et al.* [4]. We will refer to this solution as the Occurrence-Text-Sampling algorithm (OTS).

Let y be the input text, of length n, and let x be the input pattern, of length m, both over an alphabet Σ of size σ. The main idea of their sampling approach is to select a subset of the alphabet, $\hat{\Sigma} \subset \Sigma$ (the sampled alphabet), and then to construct a partial-index as the subsequence of the text (the sampled text) $\hat{y}$, of length $\hat{n}$, containing all (and only) the characters of the sampled alphabet $\hat{\Sigma}$. More formally $\hat{y}[i] \in \hat{\Sigma}$, for all $1 \leq i \leq \hat{n}$.

During the searching phase of the algorithm a sampled version of the input pattern, $\hat{x}$, of length $\hat{m}$, is constructed and searched in the sampled text. Since $\hat{y}$ contains partial informations, for each candidate position i returned by the search procedure on the sampled text, the algorithm has to verify the corresponding occurrence of x in the original text. For this reason a table ρ is maintained in order to map, at regular intervals, positions of the sampled text to their corresponding positions in the original text. The position mapping ρ has size $\lfloor \hat{n}/q \rfloor$, where q is the *interval factor*, and is such that $\rho[i] = j$ if character $y[j]$ corresponds to character $\hat{y}[q \times i]$. The value of $\rho[0]$ is set to 0. In their paper, on the basis of an accurate experimentation, the authors suggest to use values of q in the set $\{8, 16, 32\}$

Then, if the candidate occurrence position j is stored in the mapping table, i.e. if $\rho[i] = j$ for some $1 \leq i \leq \lfloor \hat{n}/q \rfloor$, the algorithm directly checks the corresponding position in y for the whole occurrence of x. Otherwise, if the sampled pattern is found in a position r of $\hat{y}$, which is not mapped in ρ, the algorithm has to check the substring of the original text which goes from position $\rho[r/q] + (r \mod q) - \alpha + 1$ to position $\rho[r/q + 1] - (q - (r \mod q)) - \alpha + 1$, where α is the first position in x such that $x[\alpha] \in \hat{\Sigma}$.

Notice that, if the input pattern does not contain characters of the sampled alphabet, the algorithm merely reduces to search for x in the original text y.

Example 1. Suppose $y =$ "abaacabdaacabcc" is a text of length 15 over the alphabet $\Sigma =$ {a,b,c,d}. Let $\hat{\Sigma} =$ {b,c,d} be the sampled alphabet, by omitting character "a". Thus the sampled text is $\hat{y} =$ "bcbdcbcc". If we map every $q = 2$ positions in the sampled text, the position mapping ρ is $\langle 5, 8, 13, 15 \rangle$. To search for the pattern $x =$ "acab" the algorithm constructs the sampled pattern $\hat{x} =$ "cb" and searches for it in the sampled text, finding two occurrences at position 2 and 5, respectively. We note that $\hat{y}[2]$ is mapped and thus it suffices to verify for an occurrence starting at position 4, finding a match. However, position $\hat{y}[5]$ is not mapped, thus we have to search in the substring $y[\rho(2) + 3 - 1..\rho(3)]$, finding the other match.

The above algorithm works well with most of the known pattern matching algorithms. However, since the sampled patterns tend to be short, the authors implemented the search phase using the Horspool algorithm, which has been found to be fast in such setting.

The real challenge in their algorithm is how to choose the best alphabet subset to sample. Based on some analytical results, supported by an experimental evaluation, they showed that it suffices in practice to sample the most frequent characters up to

some limit.[1] Under this assumption their algorithm has an extra space requirement which is only 14% of text size and is up to 5 times faster than standard online string matching on English texts.

We point out that, despite demonstrating commendable performance in searching texts composed in natural languages, which typically feature relatively large alphabets, this text sampling technique exhibits significant limitations when applied to texts characterized by smaller alphabets, such as those found in genomic sequences and proteins. More recently, an adaptation of the OTS approach was introduced in [11]. This modification involves artificially enlarging the alphabet, thereby achieving improved experimental outcomes through the use of q-grams.

For the sake of completeness it has to be noticed that in [4] the authors also consider indexing the sampled text. Specifically they build a suffix array indexing the sampled positions of the text, and get a sampled suffix array. This approach is similar to the sparse suffix array [18] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

3 Characters Distance Sampling in Brief

In this section, we provide concise description of the methodology employed to build partial-index in the *Character Distance Sampling* (CDS).

Let y be the input text, of length n, and let x be the input pattern, of length m, both over an alphabet Σ of size σ. We assume that all strings can be treated as vectors starting at position 1. Thus we refer to $x[i]$ as the i-th character of the string x, for $1 \leq i \leq m$, where m is the size of x.

The algorithm selects a sub-alphabet $C \subseteq \Sigma$ to serve as the *set of pivot characters*. Using these designated pivots, it is possible to sample the text y by calculating the distances between the n_c consecutive occurrences of any pivot character $c \in C$ within y. Formally, this sampling methodology is based on the definition of *position sampling* within a text. Given $\delta : \{1, .., n_c\} \to \{1, .., n\}$, where $\delta(i)$ is the position of the i-th occurrence of any pivot character c in y. Then the position sampled version of y, indicated by $\dot{y}$, is a numeric sequence, of length n_c, defined as $\dot{y} = \langle \delta(1), \delta(2), .., \delta(n_c) \rangle$.

Example 2. Suppose $y =$ "agaacgcagtata" is a sequence of length 13, over the alphabet $\Sigma =$ {a,c,g,t}. Let $C =$ {a} be the set of pivot characters. Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 8, 11, 13 \rangle$. Specifically the first occurrence of character "a" is at position 1 ($y[1] =$ "a"), its second occurrence is at position 3 ($y[3] =$ "a"), and so on.

We can now define the *Character Distance Function* defined by $\Delta(i) = \delta(i+1) - \delta(i)$, for $1 \leq i \leq n_c - 1$, as the distance between two consecutive occurrences of any pivot character in y. Then the *characters-distance sampled version* of the text y is a numeric sequence, indicated by $\bar{y}$, of length $n_c - 1$ defined as $\bar{y} = \langle \Delta(1), \Delta(2), .., \Delta(n_c - 1) \rangle = \langle \delta(2) - \delta(1), \delta(3) - \delta(2), .., \delta(n_c) - \delta(n_c - 1) \rangle$

Example 3. Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma =$ {a,c,g,t}. Let $C =$ {a} be the set of pivot characters. Thus the character distance

[1] According to their theoretical evaluation and their experimental results it turns out that, when searching on an English text, the best performances are obtained when the least 13 characters are removed from the original alphabet.

sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$. Specifically $\bar{y}[1] = \Delta(1) = \delta(2) - \delta(1) = 3 - 1 = 2$, while $\bar{y}[3] = \Delta(3) = \delta(4) - \delta(3) = 8 - 4 = 4$, and so on.

In practical scenarios, particularly when dealing with large alphabets, the set of pivot characters may comprise only one character. Consequently, for the sake of simplicity, we will frequently refer to the pivot character in the singular form, rather than mentioning the entire set of pivot characters.

The approach of sampled string matching utilizing CDS maintains a partial index, which is represented by the position-sampled version of the text y. The size of this index is $32n_c$ bits, assuming that this index resides in memory and is readily available for any search operation on the text. When there arises a need to search for a pattern x of length m within y, a preprocessing step is executed on the pattern to compute its sampled version $\bar{x}$. It can be straightforwardly proved that an occurrence of x in y corresponds to an occurrence of $\bar{x}$ in $\bar{y}$, hence it suffices to utilize any string matching algorithm to locate the occurrences of $\bar{x}$ in $\bar{y}$ to solve the problem. However, the reverse scenario is not necessarily true, implying that occurrences of $\bar{x}$ in $\bar{y}$ may not align with occurrences of x in y. Consequently, for each occurrence of $\bar{x}$ in $\bar{y}$, referred to as a candidate occurrence, a validation check in y is required.

Given that the validation process demands $O(m)$ computational time, the entire search operation will consume $O(mn)$ time. Nonetheless, envisioning modifications to the fundamental procedure to ensure that the overall search operation, despite the checks, remains linear in time is not challenging (for further details, refer to [10]).

An essential aspect to highlight in our discourse is that the CDS-based approach does not explicitly maintain the character-distance sampled version $\bar{y}$ of the text. Instead, it maintains the position-sampled version $\dot{x}$ of the text. Indeed, $\bar{y}$ solely retains the distances between the pivot characters and lacks direct ties to the original positions of these pivot characters within the text. Consequently, directly verifying every candidate occurrence becomes impractical. This issue is addressed by retaining the text $\dot{y}$, which holds the positions, and computing $\bar{y}$ on-the-fly during the search. The i-th element of $\bar{y}$ can indeed be computed in constant time using the relationship $\bar{y}(i) = \dot{y}(i + 1) - \dot{y}(i)$.

The CDS-based sampled string matching approach has demonstrated remarkable effectiveness in practical applications, boasting a significant reduction in search times by up to 40 times compared to standard online exact string matching techniques. Remarkably, this enhancement is achieved while incurring a relatively minimal cost, as it entails the construction of a partial index merely equivalent to 2% of the text size. Moreover, sampled string matching has exhibited exceptional flexibility, rendering it adept at addressing text searching challenges, even in the approximate realm. Notably, Faro *et al.*[13] recently introduced the run-length text sampling, tailored for approximate searches on texts. This technique proves particularly well-suited, for instance, for tasks such as *Order Preserving pattern matching* [19].

In addition to its commendable space and time efficiency, sampled string matching offers a plethora of other advantageous features. For instance, ease of programming stands out as a notable advantage, with the construction of the partial index typically being a swift and straightforward process. Moreover, the inherent flexibility of the data structure allows it to seamlessly adapt to text variations. This means that minor alterations in the text, such as character deletions or insertions, can be effortlessly reflected in the corresponding index.

However the one described above is not without its share of pitfalls or weaknesses. One such challenge is the variability in performance based on the choice of pivot character. Consequently, strategic consideration must be given to selecting the pivot character, striking a balance between partial index size and execution times. Research indicates that in the case of the English language the pivot character ranked 8th tends to offer best performances.

Another factor to consider is that if the pattern is exceptionally short and lacks occurrences of the pivot character, resorting to a standard string search within the text becomes necessary. Additionally, this method may not yield significant advantages when applied to texts with small alphabets, as the benefits in terms of space efficiency may not be realized. However, studies by Faro *et al.* [11] have proved the efficacy of a technique leveraging condensed alphabets to expand the underlying alphabet size and achieve markedly improved performance.

Recently new study conducted by the original authors showed new space and time improvement by using the fake distance representation [9].

4 Online String Matching Algorithms

In this section, we delve into the algorithms utilized to execute the searching phase within the context of the sampling methodologies. Our selection of algorithms aims to present a comparative analysis against the original sampling implemented through Horspool algorithm, considering their suitability across varied pattern and alphabet sizes. We have curated three distinct algorithms:

- Quick Search (QS) [22]: specifically tailored for large patterns and extensive alphabets.
- Weak Factor Recognition (WFR) [3]: for normal-sized patterns and smaller alphabets.
- Franek-Jennings-Smyth Algorithm (FJS) [16]: for very small patterns and large alphabets.

The subsequent sections provide an in-depth exploration of each algorithm, elucidating their underlying mechanisms and performance characteristics.

4.1 QS

The Quick-Search algorithm proposed by Sunday presents a simple variation of the Horspool algorithm. After each attempt, the shift is computed based on the character immediately following the current window of the text, denoted as $t[s+m]$. This corresponds to advancing the shift by $qbc_p(t[s+m])$ positions, where

$$qbc_p(c) = \min\{1 \le k \le m \mid p[m-k] = c\} \cup \{m+1\}$$

for all $c \in \Sigma$.

4.2 WFR

In the preprocessing phase, all factors of the pattern x are indexed to speed up the subsequent searching phase. Specifically, we define a hash function $h : \Sigma^* \to \{0, \ldots, 2^\alpha - 1\}$, which associates an integer value $0 \le v < 2^\alpha$ (for a fixed bound

α) with any string over the alphabet Σ. The value of α in the definition of the hash function h may depend on the target machine on which the algorithm is implemented. In our setting, the value of α has been fixed to 16 so that each hash value fits perfectly into a single 16-bit register. Although greater values of α are possible, we observed that the average number of false positives due to the hash function is negligible when the value of α is set to 16. A factor table F will store the hash values of the factors of x. It can be implemented as a table of Boolean values such that, for $0 \leq v < 2^{\alpha}$, $F[v]$ is set (i.e., $F[v] = \text{True}$) if and only if there exists a factor z of x such that $h(z) = v$.

During the searching phase, a window of size m slides along the text, starting at position 0. After each attempt, the window is shifted to the right until the end of the text is reached. Specifically, in the attempt at a given position i of the text, the window is opened on the substring $y[i..i+m-1]$. Thus, the `WFR` algorithm computes the hash values $h_{i,l}$, starting from $l = 1$ and for increasing values of l, until either $F[h_{i,l}] = \text{False}$ or $l = m$ (and, therefore, $F[h_{i,l}] = \text{True}$).

4.3 FJS

The Franek-Jennings-Smyth string matching algorithm (FJS for short) is a simple hybrid algorithm which mixes the linear worst-case time complexity of Knuth-Morris-Pratt algorithm and the sublinear average behavior of Quick-Search algorithm. Specifically the FJS algorithm searches for matches of p in t by shifting a window of size m from left to right along t. Each attempt of the algorithm is divided into two steps. During the first step, in accordance with the Quick-Search approach, the FJS algorithm first compares the rightmost character of the pattern, $p[m-1]$, with its corresponding character in the text, that is, $t[s+m-1]$. If a mismatch occurs, a Quick-Search shift is implemented, moving p along t until the rightmost occurrence in p of the character $t[s+m]$ is aligned with position $s+m$ in the text. At this new location, the rightmost character of p is again compared with the corresponding text position. Only when a match is found the FJS algorithm invokes the second step. Otherwise another Quick-Search shift occurs. The second step of the algorithm consists in a Knuth-Morris-Pratt pattern matching starting from the leftmost character $p[0]$ and, if no mismatch occurs, extending as far as $p[m-2]$. Then whether or not a match of p is found, a Knuth-Morris-Pratt shift is eventually performed followed by a return to the first step. The preprocessing time of the algorithm takes $O(m)$ time for computing the failure function of KMP, and $O(m+\sigma)$ time in order to compute the Quick-Search bad character rule. The authors showed that the worst-case number of character comparisons is bounded by $3n-2m$, so that the corresponding searching phase requires $O(n)$time. The space complexity of the algorithm is $O(m+\sigma)$

5 Experimental Results

In this section, we present experimental results to evaluate the performance of the sampled string matching approaches outlined in this paper.

The algorithms were implemented in the C programming language and tested using the Smart tool [7] on a MacBook Pro with 4 cores, a 2.7 GHz Intel Core i7 processor, 16 GB RAM 2133 MHz LPDDR3, 256 KB of L2 Cache, and 8 MB of Cache L3.[2] The algorithms were compiled using the -O3 optimization option. Performance

[2] The Smart tool can be downloaded from `http://www.dmi.unict.it/~faro/smart/` or from `https://github.com/smart-tool/smart`.

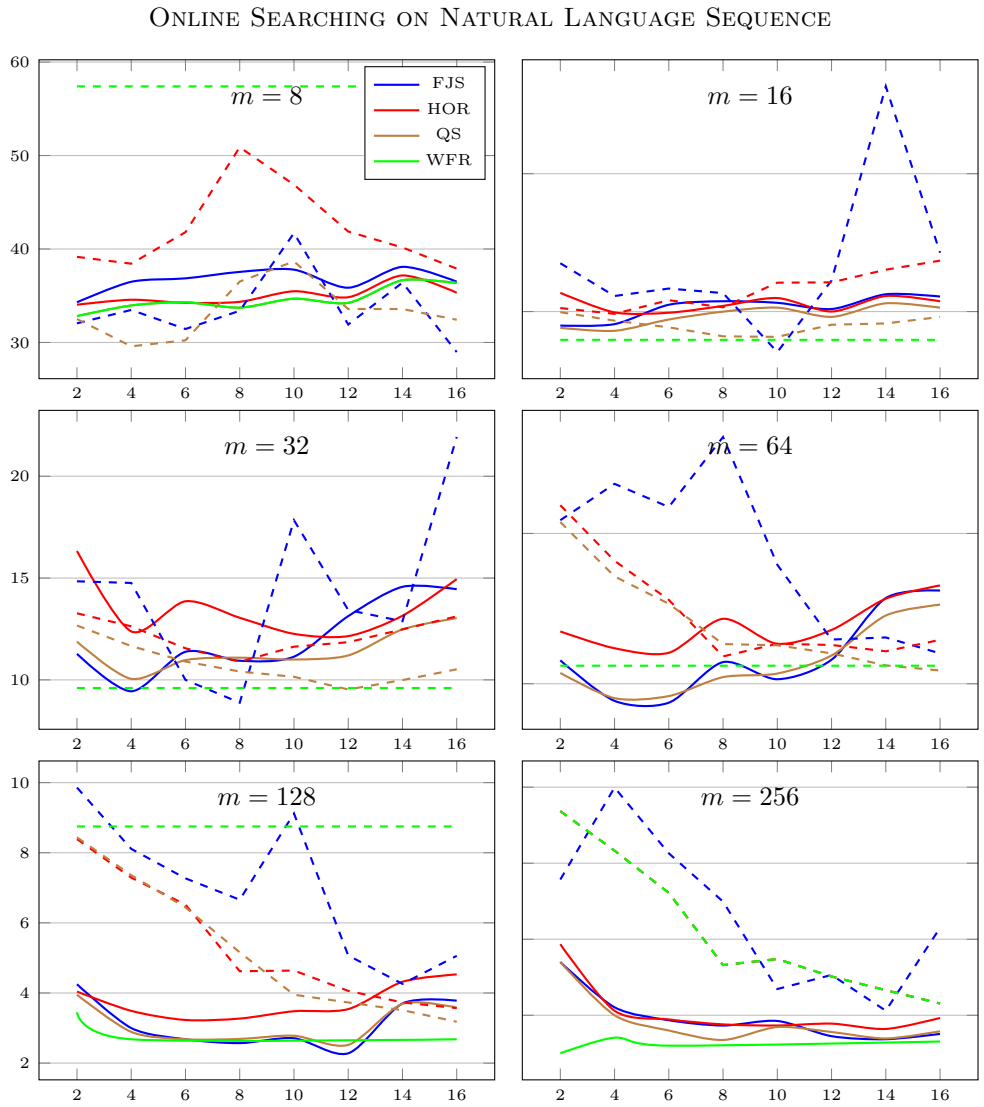

Figure 1. Experimental results of the sampling algorithms implemented through different online string matching algorithms, using English Texts. Straight lines represent the CDS algorithms while dashed lines represent OTS ones. Times are represented in the y axes in milliseconds (ms), while the x axes represents the rank of the pivot character $2 \leq K \leq 16$.

comparisons were made based on searching times and are expressed in milliseconds (ms).

Two text buffers, each with a size of 100 MB, were used, sourced from the *Pizza and Chili* dataset [15], available online for download. Specifically, the algorithms were tested using a genomics data sequence and a natural language text. For each sequence, 500 patterns were randomly selected from the text, and the average running time was computed over the 500 runs.
Moreover as previously pointed out in this paper, sampling technique exhibits significant limitations when applied to texts characterized by smaller alphabets, such as those found in genomics sequences and proteins.

For both the natural language dataset and the genomics sequences, experiments were conducted for normal-sized patterns ($8 \leq m \leq 256$). Furthermore, for algorithms that can be implemented with different values of the parameter q, tests were

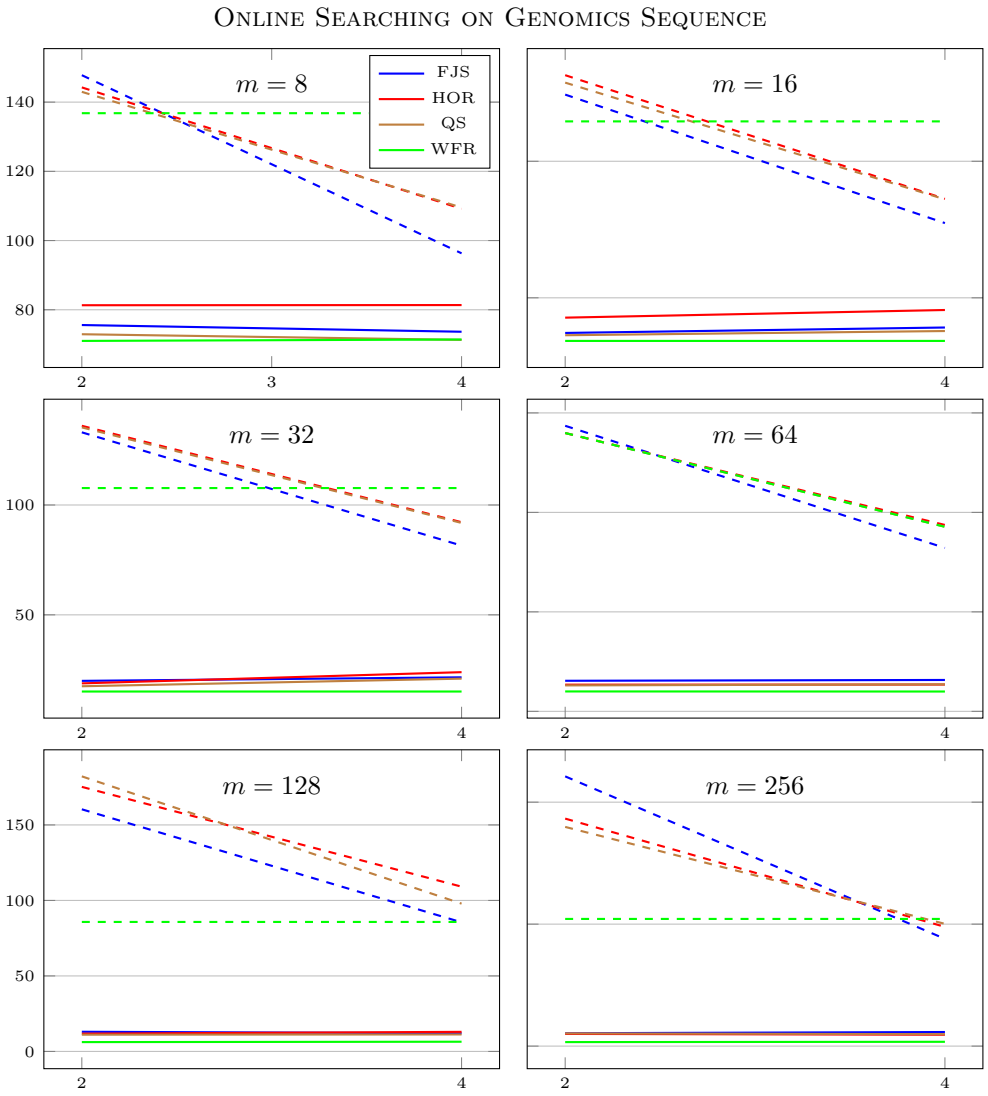

Figure 2. Experimental results of the sampling algorithms implemented through different online string matching algorithms, using Genomics Sequences. Straight lines represent the CDS algorithms while dashed lines represent OTS ones. Times are represented in the y axes in milliseconds (ms), while the x axes represents the rank of the pivot character $2 \leq K \leq 4$.

conducted for multiple values, generally $1 \leq q \leq 8$, and only the best result among the different values of q was considered for each pattern size. Let K be the rank of a character in a totally ordered alphabet, we conducted our tests for different values of K, indeed for the natural language dataset we used $2 \leq K \leq 16$, while for the genomics sequences given $|\Sigma| = 4$ we could use only $2 \leq K \leq 4$. In both Figure 1 and 2 the legend show the color of the algorithm used in the searching phase, while the smooth line represent the CDS algorithm has been used, and the dashed lines are used to represent the OTS algorithm.

6 Conclusions

In this paper we have presented an extension of the text sampling approaches, called Character Distance and Occurrence Text Sampling, to the case of applying different

searching algorithms. This extension was carried out using four different well known algorithms. Our results proved the efficacy of the sampled methods discussed in this paper and their versatility to be adapted to any online string matching algorithms without impacting their original performances.

Although our tests were limited to the exact string matching problem, obtaining excellent results, we believe that the approach can be effectively generalized even to non-standard string matching. Our future studies will focus in this direction in order to apply sampled string matching to other problems related to text processing.

Acknowledgements

Simone Faro was supported by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, affiliated to Spoke 10, co-founded by the European Union – NextGenerationEU.

References

1. A. Apostolico: *The myriad virtues of subword trees*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Berlin, Heidelberg, 1985, Springer Berlin Heidelberg, pp. 85–96.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) oct 1977, p. 762–772.
3. D. Cantone, S. Faro, and A. Pavone: *Speeding up string matching by weak factor recognition*, in Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017, pp. 42–50.
4. F. Claude Faust, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio: *String matching with alphabet sampling.* Journal of Discrete Algorithms, 11 12 2010.
5. M. Crochemore: *Speeding up two string-matching algorithms.* Algorithmica, 12(4) 1994, pp. 247–267.
6. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) mar 2013.
7. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 99–111.
8. S. Faro and F. P. Marino: *Reducing time and space in indexed string matching by characters distance text sampling*, in Prague Stringology Conference 2020, Prague, Czech Republic, August 31 - September 2, 2020, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020, pp. 148–159.
9. S. Faro, F. P. Marino, A. Moschetto, A. Pavone, and A. Scardace: *The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples*, in 12th International Conference on Fun with Algorithms (FUN 2024), A. Z. Broder and T. Tamir, eds., vol. 291 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2024, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 13:1–13:17.
10. S. Faro, F. P. Marino, and A. Pavone: *Efficient online string matching based on characters distance text sampling.* Algorithmica, 82(11) 2020, pp. 3390–3412.
11. S. Faro, F. P. Marino, and A. Pavone: *Enhancing characters distance text sampling by condensed alphabets*, in Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021, C. S. Coen and I. Salvo, eds., vol. 3072 of CEUR Workshop Proceedings, CEUR-WS.org, 2021, pp. 1–15.
12. S. Faro, F. P. Marino, and A. Pavone: *Improved characters distance sampling for online and offline text searching.* Theor. Comput. Sci., 946 2023, p. 113684.

13. S. Faro, F. P. Marino, A. Pavone, and A. Scardace: *Towards an efficient text sampling approach for exact and approximate matching*, 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. 75–89.
14. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) jul 2005, p. 552–581.
15. P. Ferragina and G. Navarro: *Pizza&Chili*, Available online: pizzachili.dcc.uchile.cl/, 2005.
16. F. Franek, C. G. Jennings, and W. F. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discrete Algorithms, 5(4) 2007, pp. 682–695.
17. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
18. J. Kärkkäinen and E. Ukkonen: *Sparse suffix trees*, in Computing and Combinatorics, J.-Y. Cai and C. K. Wong, eds., Berlin, Heidelberg, 1996, Springer Berlin Heidelberg, pp. 219–230.
19. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order-preserving matching.* Theoretical Computer Science, 525 2014, pp. 68–79, Advances in Stringology.
20. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
21. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
22. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
23. U. Vishkin: *Deterministic sampling–a new technique for fast pattern matching.* SIAM Journal on Computing, 20(1) 1991, pp. 22–40.
24. A. C.-C. Yao: *The complexity of pattern matching for a random string.* SIAM Journal on Computing, 8(3) 1979, pp. 368–387.

Refining SFDC Compression Scheme with Block Text Segmentation*

Simone Faro and Alfio Spoto

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

Abstract. The Succinct Format with Direct Accessibility (SFDC) is an encoding scheme originally designed for efficient data compression and quick access to elements within compressed sequences. While SFDC performs well under stable character frequency conditions, its efficacy diminishes in text corpora with high variability in character frequencies, typical of natural language environments. Addressing this limitation, this paper presents three variant of SFDC based on block segmentation methods, each offering unique enhancements over the original SFDC representation. By tailoring the segmentation process to the distribution of characters within the text, these methods aim to optimize compression efficiency and decoding performance. The paper presents experimental results demonstrating the effectiveness of these approaches, highlighting their ability to improve upon the original scheme in several scenarios. The findings underscore the potential of these advanced segmentation strategies to provide superior compression and performance across a range of text datasets.

Keywords: text processing, text compression, computational friendly data structures

1 Introduction

Text compression is the process of reducing the size of a given text to save storage space and speed up its transmission across digital systems. This is crucial for efficient data management and quick access to information, which is especially important in environments like online streaming, real-time communication, and large-scale data storage. Formally, the challenge is to modify the representation of a text y, of length n and whose characters are drawn from an alphabet Σ of size σ, such that the new format optimizes storage space usage. The basic premise in conventional compression is that every symbol in an uncompressed text is represented by $\lceil \log_2 \sigma \rceil$ bits, totaling $n\lceil \log_2 \sigma \rceil$ bits[1]. However, by employing a variable-length encoding, such as Huffman's optimal compression scheme [7], each character $c \in \Sigma$ can be represented by a code $\rho(c)$, with the length depending on the character's frequency $f(c)$ in the text, allowing the text to be encoded in fewer bits. In an optimal compression scheme, [2] like Huffman's, the total number of bits required for the compressed text is $N = \sum_{c \in \Sigma} f(c) \cdot |\rho(c)|$. This total N is minimized in such a way that it is less than or equal to the number of bits required in the conventional representation, i.e., $N \leqslant n\lceil \log_2 \sigma \rceil$ bits. Huffman's algorithm calculates an optimal prefix code based on character frequencies and creates a binary tree (Huffman tree) for efficient decoding. These Huffman trees, while not unique, are pivotal in compression as they simplify

* This work is supported by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, co-founded by the European Union - NextGenerationEU.

[1] Throughout the paper, all logarithms are intended in base 2, unless otherwise stated.

[2] An optimal compression scheme minimizes the number of bits required to represent the text.

Simone Faro, Alfio Spoto: Refining SFDC Compression Scheme with Block Text Segmentation, pp. 27–41.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

the decoding process. The leaves of the tree represent characters, and the branches are labeled in a binary manner, facilitating straightforward codeword retrieval. Despite their efficiency, variable-length codes like Huffman's pose a significant challenge: they do not allow direct access to the i-th codeword in the encoded string without processing previous characters. This limitation hampers quick data retrieval and affects algorithmic computations, especially in environments that benefit from rapid access. Several encoding schemes have been developed to address this issue, providing direct access while trying to maintain compactness. Notable examples include Dense Sampling [5], Elias-Fano codes [4], Interpolative coding [8,9], Wavelet Trees [6], and DACs [1,2]. These schemes vary in their approach and efficiency.

This work is based on the most recent SFDC format, which leverages variable-length codes to enable fast and direct access to any element within a compressed sequence. This format excels under conditions of low variability in character frequencies, providing consistent access times and superior compression ratios compared to other methods. SFDC's design is inherently flexible, making it suitable for a range of applications where the trade-off between efficiency and space consumption is critical. The scheme employs a total of $N + \mathcal{O}\big(n\,(\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1})\,\big) = N + \mathcal{O}(n)$ bits, where λ is parameter involved in the encoding, F_j represents the j-th Fibonacci number, achieving efficient access times that are merely proportional to the length of the character's encoding, with an expected additional time overhead $\mathcal{O}((F_{\sigma-\lambda+3} - 3)/F_{\sigma+1})$.

In addressing the limitations inherent to the traditional SFDC approach, particularly in scenarios with high variability in character frequency along the text, in this paper we introduce three block segmentation strategies within the SFDC framework. The first approach segments the text into fixed-length blocks and uses a single Huffman tree constructed from the character frequencies of the entire text. This method is straightforward but may not adapt well to varying character distributions within the text. The second approach also uses fixed-length blocks but constructs adaptively a separate Huffman tree for each block. By tailoring the Huffman tree to each block's character frequencies, this method aims to improve the compression factor. The third approach segments the text into variable-length blocks based on the occurrences of infrequent characters. This method adapts to the text's natural structure, potentially enhancing compression efficiency. All three approaches reduce the average delay in the SFDC scheme, mitigating its specific limitations. Their performance varies depending on the text characteristics. Our extensive experiments evaluate the performance of these approaches, demonstrating that the choice of segmentation strategy should be based on the specific text properties to optimize compression and decoding.

The structure of this article is as follows. Section 2 provides a comprehensive description of the SFDC representation. In Section 3, we examine a specific phenomenon that adversely affects the performance of the SFDC scheme. Section 4 introduces three new variants of the SFDC approach, which are based on block text segmentation. In Section 5, we present experimental results to assess the performance of these new variants. Finally, Section 6 offers our conclusions.

2 An Overview of the SFDC Representation Scheme

This section summarizes the SFDC representation [3], a recently introduced text compression scheme. SFDC is based on Huffman's algorithm to create optimal prefix

binary codes for characters of an alphabet Σ of size σ, with each character $c \in \Sigma$ represented by a binary code $\rho(c)$. Here, we assume characters are ordered in a non-decreasing frequency, with c_0 being the least frequent and $c_{\sigma-1}$ the most frequent. The SFDC scheme encodes a text y of length n, over an alphabet Σ, into an ordered collection of λ binary strings, where λ is a fixed integer satisfying $2 \leqslant \lambda \leqslant \max\{|\rho(c)| : c \in \Sigma\}$ and represents the size of the SFDC representation. The parameter λ is upper bounded by the length of the codeword of the alphabet character Σ, which is the longest among all characters, i.e., $\lambda \leqslant \max\{|\rho(c)| : c \in \Sigma\}$. This setup includes $\lambda - 1$ *fixed layers* and one additional *dynamic layer*, enhancing the encoding performance.

Each of the first $\lambda - 1$ layers, denoted by $\widehat{Y}_0, \widehat{Y}_1, \ldots, \widehat{Y}_{\lambda-2}$, contains the i-th bits of the encodings of the characters in y, arranged in the order they appear:

$$\widehat{Y}_i \coloneqq \big\langle \rho(y[0])[i],\ \rho(y[1])[i],\ \ldots,\ \rho(y[n-1])[i] \big\rangle,$$

where $\rho(y[j])[i] = 0$ if $i \geqslant |\rho(y[j])|$ for $0 \leqslant j < n$.

The final, dynamic layer, $\widehat{Y}_D \coloneqq \widehat{Y}_{\lambda-1}$, gathers all remaining bits of the character encodings exceeding $\lambda - 1$ in length, following a last-in first-out (LIFO) scheme. This layer adheres to specific storage rules to ensure efficient encoding and decoding:

1. Bits from the same encoding are stored sequentially from left to right.
2. Pending bits from earlier in the text precede those later in the text.
3. Pending bits are stored in the leftmost available position, respecting 1. and 2.

As proved in [3], this approach effectively manages the varying length of Huffman codes, ensuring efficient data compression and accessibility.

2.1 Encoding and Decoding procedures

During encoding, each of the n characters of the text is processed using a Last-In-First-Out (LIFO) strategy implemented with a stack $\mathcal{S}$ to manage the pending bits in the dynamic layer $\widehat{Y}_D$. At each iteration i, the algorithm places up to $\lambda - 1$ bits of $\rho(y[i])$ into the fixed layers. If $|\rho(y[i])| < \lambda - 1$, then $\lambda - |\rho(y[i])| - 1$ bits in the fixed layers remain unused. We refer to such positions as *idle bits* in the representation. If the encoding length of character $y[i]$ exceeds $\lambda - 1$, the excess bits (*pending bits*)

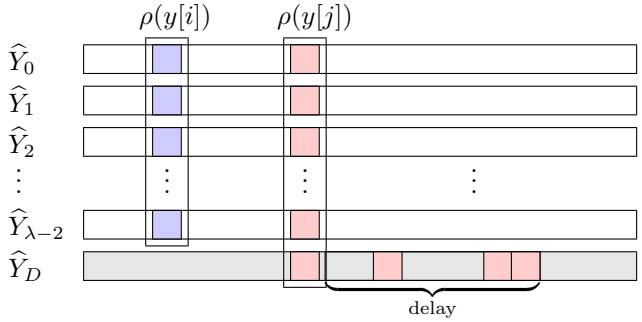

Figure 1. Examples of the reorganization of the bits of two character's code: $\rho(y[i])$ has length $\lambda - 1$ and fits within the λ layers of the representation; $\rho(y[j])$ has length $\lambda + 3$ and its 3 pending bits are arranged along the dynamic layer.

char	code	length
s	001	3
e	01	2
n	010	3
p	0110101	7
m	101	3
C	1100011010	10
o	1100111	7
i	11010	5
r	11101	5
_	00001	5

	0	1	2	3	4	5	6	7	8	9	10
	C	o	m	p	r	e	s	s	i	o	n
$\widehat{Y}_0$	1	1	1	0	1	0	0	0	1	1	0
$\widehat{Y}_1$	1	1	0	1	1	1	0	0	1	1	1
$\widehat{Y}_2$	0	0	1	1	1	-	1	1	0	0	0
$\widehat{Y}_3$	0	0	-	0	0	-	-	-	1	0	-
$\widehat{Y}_4$	0	1	-	1	1	-	-	-	0	1	-
$\widehat{Y}_D$	1	1		0						1	
	1	1		1						1	
	0										
	1										
	0										

	0	1	2	3	4	5	6	7	8	9	10
	C	o	m	p	r	e	s	s	i	o	n
$\widehat{Y}_0$	1	1	1	0	1	0	0	0	1	1	0
$\widehat{Y}_1$	1	1	0	1	1	1	0	0	1	1	1
$\widehat{Y}_2$	0	0	1	1	1	-	1	1	0	0	0
$\widehat{Y}_3$	0	0	-	0	0	-	-	-	1	0	-
$\widehat{Y}_4$	0	1	-	1	1	-	-	-	0	1	-
$\widehat{Y}_D$	1	1	1	0	1	1	0	1	0	1	1

Figure 2. The SFDC representation of the string `Compression` with $\lambda = 6$ layers (5 fixed layers and an additional dynamic layer).

are pushed onto a stack $\mathcal{S}$ in reverse order. Once the bits are processed, the top bit of the stack is then stored into the dynamic layer $\widehat{Y}_D$. If the stack $\mathcal{S}$ is empty at any iteration, the corresponding bit position in $\widehat{Y}_D$ will remain idle. After all characters are processed, any remaining bits in the stack are sequentially placed into the dynamic layer, potentially extending its length beyond that of the fixed layers.

The total execution time of the encoding algorithm for a text y of length n effectively amounts to $\mathcal{O}(n+N)$, where N is the total length of the encoded text. This accounts for the time to read the input text, which is $\mathcal{O}(n)$, and the time to generate the encoded text, which is $\mathcal{O}(N)$. Note that for highly compressible texts, N can be significantly smaller than n.

The decoding of characters from a text encoded with the SFDC representation allows direct access to the start position of the encoding, but decoding time is not necessarily constant. When decoding a character $y[i]$, additional characters might need to be decoded due to the *decoding delay*, which is the additional effort required to complete the decoding of $y[i]$. This occurs if the last bit of the encoding of $y[i]$ is located at position j in the dynamic layer, where $j \geqslant i$, necessitating the decoding of all characters from position i to j, including $y[j]$. In this context, the *decoding delay* is thus defined as the need to decode additional $j - i$ characters to fully decode $y[i]$.

Assume we want to decode the window of the text $y[i\,..\,j]$. Each character $y[i]$ is decoded by traversing the Huffman tree from the root to a leaf, selecting the left or right subtree based on the bits of $\rho(y[i])$. Each node x in the tree has a Boolean value $x.\mathsf{leaf}$ indicating whether it is a leaf, and if so, $x.\mathsf{symbol}$ provides the associated character. A stack is used to manage the pending bits from the dynamic layer, storing tree nodes related to partially decoded characters. Nodes in the stack are associated with their respective positions in the text, allowing efficient positioning once decoded. The stack follows a LIFO strategy, where nodes are added or removed based on their proximity to being fully decoded.

The procedure iteratively scans each layer from $\widehat{Y}_0$ to $\widehat{Y}_D$, attempting to decode each character. If a leaf is reached early, the character is immediately decoded; otherwise,

the node is placed on the stack for further processing. The loop continues until the stack is empty and $y[j]$ is successfully decoded. The complexity of decoding a single character is $\mathcal{O}(|\rho(y[i])| + d)$, and for a window $y[i..j]$, it is $\mathcal{O}(\sum_{k=i}^{j} |\rho(y[k]|) + d)$, where d represents the decoding delay. Estimates of the expected decoding delay can be computed through simulations, adjusting λ to achieve acceptable performance levels. Of course, the time complexity of these simulations is $\mathcal{O}(n+N)$, with N being the total length of encoded text.

2.2 Performances and Evaluation

Regarding space efficiency, Cantone and Faro proved in [3] that SFDC uses a total of $N + \mathcal{O}(n)$ bits, optimizing both space and access time in the encoded representation. More formally, assuming the worst case scenario, the expected number of idle bits in a SFDC encoding using λ layers can be estimated by the following formula $\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1}$, where F_j is the j-th Fibonacci number. As the size of the alphabet σ increases, it is easy to verify that the function quickly converges to the value $\lambda - 2.618$. Consequently, if we assume that the value λ represents a constant implementation-related parameter, the total space used by SFDC for encoding a sequence of n characters is equal to $N + \mathcal{O}\left(n\left(\lambda - (F_{\sigma+3} - 3/F_{\sigma+1})\right)\right) = N + \mathcal{O}(n)$.

Regarding the expected value of the decoding delay, whch is the number of additional characters that need to be decoded in order to obtain the full encoding of a character of the text, the authors proved in [3] that, for an alphabet of σ characters, the expected delay of our SFDC encoding with λ layers (where $4 \leqslant \lambda \leqslant \sigma - 1$) can be estimated, in the worst case, by $(F_{\sigma-\lambda+3} - 3)/F_{\sigma+1}$, which is constant.

From the experimental results it emerges that the SFDC representation scheme offers a compression very close to the optimal values and offers in many cases an extremely low average delay, showing a certain competitiveness with the best compression schemes that offer direct access. However, from experimental results it turns out also that the SFDC scheme suffers particularly when the text contains portions in which the character frequency diverges significantly from the overall frequency of the text. This is the case, for instance, of datasets consisting of the union of several natural language texts, with slightly different character frequencies, or in the case of datasets containing portions in which infrequent characters are found in contiguous sequences. These circumstances lead to a phenomenon that we call *LIFO Delay Amplification*. Such phenomenon was neither identified nor analyzed by Faro and Cantone. In this paper we highlight how the LDA effect can significantly impact the performance of the SFDC encoding by increasing the decoding delay for characters preceding rare characters in the text. By bringing attention to this previously unrecognized issue, we provide a deeper understanding of the challenges associated with the SFDC scheme and propose strategies to mitigate its effects.

The following section deals with this phenomenon in detail while the rest of the paper proposes solutions to significantly mitigate it.

3 Understanding LIFO Delay Amplification in SFDC

The SFDC representation scheme employs variable-length encodings for characters. In this framework, frequently occurring characters are assigned shorter codewords,

whereas characters that appear rarely in the text are given longer codewords. For the purposes of this paper, we define *rare markers* as those characters with very low relative frequency. In this context, the *LIFO Delay Amplification* (LDA) phenomenon in SFDC refers to the unintended increase in decoding delay for characters appearing in a block that precedes a rare character in the text. This phenomenon actually occurs because rare characters can induce significant delay to characters preceding them due to their long decoding paths. Such behavior is exacerbated by the LIFO strategy adopted in the SFDC scheme which assigns decoding priority to the rightmost characters in the text.

When a rare character is encoded, all preceding characters waiting in the stack for some pending bits must wait for the rare character's encoding to complete. And this occurs also for those characters with shorter codewords that wait for a single bit on the stack. Consequently, characters that normally would require minimal delay to decode suddenly experience an increased waiting time, thus amplifying the overall decoding delay for the block. Consider the example illustrated in Fig. 3, which demonstrates the application of the SFDC representation to the text "`poor poor Compression`", assuming the character encodings provided in Figure 2. It also indicates the length of the codeword for each character in the text and its corresponding decoding delay. In this instance, the rare marker is the character "`C`" located at position 10. It is noteworthy that the block of characters preceding the "`C`" is primarily composed of characters whose codeword length is equal to or slightly exceeds the number of layers. Despite this, the delay associated with these characters increases linearly as the distance from the rare character increases, illustrating the potentially adverse effects of the LDA phenomenon. Note that the block involved in the LDA phenomenon has the shape of a ladder that descends from left to right ending, at the far right, at the position of the rare marker.

In the example of Fig. 3, the average delay of each character is just over 4. However, it is curious to observe that if we move the character "`C`" in the last position of the text the average delay would suddenly drop below 1. This highlights how the primary implication of LDA is a decrease in the efficiency of the decoding process, particularly in texts where rare characters are unevenly distributed. This can lead to significant variability in the performance of the SFDC encoding scheme, affecting both its speed and its predictability. For applications requiring consistent and rapid decoding, such as real-time data processing, this can pose considerable challenges.

This phenomenon can be observed in the experimental results shown by Faro and Cantone in [3], where a particularly inefficient behavior of the SFDC representation scheme is highlighted in the case of the ENGLISH dataset, represented by a collection of texts written in natural language, the total length of which is 100 million characters. In this case, in fact, the average delay obtained on the dataset is particularly high, equal to approximately $44,300$ characters. Analyzing the performance of the algorithm in

$y[i]$	p	o	o	r	_	p	o	o	r	_	C	o	m	p	r	e	s	s	i	o	n
$\lvert\rho(y[i])\rvert$	7	7	7	5	6	7	7	7	5	6	10	7	3	7	5	2	3	3	5	7	3
$d(i)$	23	21	1	0	15	13	11	1	0	8	7	1	0	1	0	0	0	0	0	1	0

Figure 3. An example of an SFDC scheme in which the LDA phenomenon occurs: the rare marker "`C`" in position 10 induces a significant decoding delay in the entire block of text preceding it.

more detail for this dataset, it is highlighted that the maximum delay obtained on the text is equal to approximately $17,400,000$ characters. Assuming that such high values could have been generated by a single LDA phenomenon on the text, it would be sufficient to have a rare marker whose delay is equal to just over 17 million. If this marker is capable of amplifying the delay on a block of 260 thousand characters that precede it (just 0.26% of the length of the text), the observed values would be obtained. The histogram shown in Fig. 4 confirms our analysis. It shows the delay values associated with all 100 million characters of the ENGLISH dataset. It is easy to locate a block of considerable height whose structure is typical of LDA. The histogram associated with the PROTEIN dataset does not show the occurrence of this phenomenon.

LDA is a notable challenge in the SFDC encoding approach, particularly affecting its application in environments where decoding speed is crucial. Understanding and addressing this phenomenon is essential for optimizing the performance of the SFDC scheme and ensuring their practical utility in diverse computational contexts. Mitigating the effects of LIFO Delay Amplification involves strategic block management and possibly adjusting the Huffman coding process to minimize the occurrence of lengthy rare character codes at critical points within the text. One approach could be the introduction of adaptive Huffman trees that adjust more dynamically to the text's character frequency distribution, thereby potentially reducing the impact of rare characters on overall block decoding times. In the following sections we will try to mitigate the phenomenon through the use of a text block segmentation strategy.

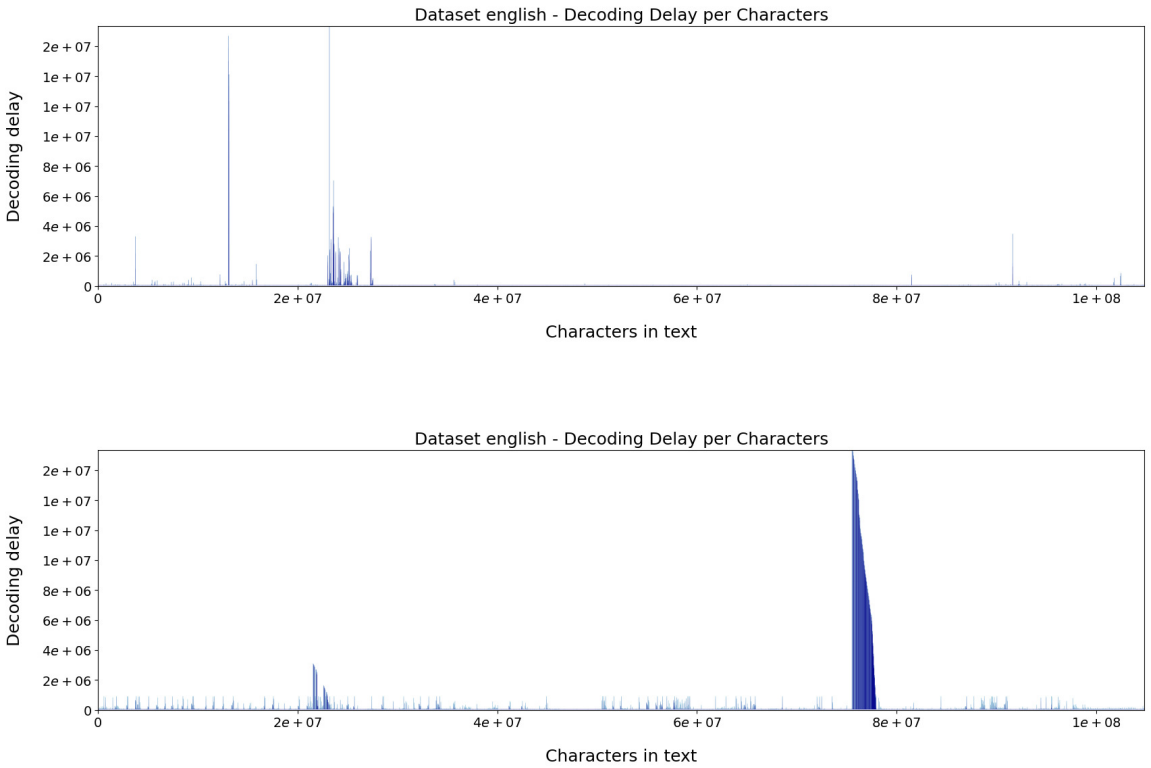

Figure 4. Histograms of the delay values associated with the 100 million characters of the PROTEIN and the ENGLISH datasets, respectively, obtained by the SFDC representation.

4 Evaluating Block Text Segmentation in SFDC

In this section we discuss some approaches based on text segmentation to address the challenges faced by LDA, especially in the context of datasets which exhibit substantial internal variability of character frequencies, like those consisting of various texts written in natural language. The text segmentation approach partitions the text into smaller blocks and compresses each block separately using the SFDC method. As a general effect, dividing the text into blocks can mitigate the effects of the LDA phenomenon by allowing the pending bits in the stack to be processed in advance. Therefore, closing a block enables the placement of all pending bits, thereby reducing the waiting times for the characters in the stack.

However, a text segmentation can be implemented in various ways. In this paper, we evaluate the following three primary segmentation strategies:

- Fixed Length Block Segmentation (Section 4.1);
- Adaptive Huffman Encoding in Fixed Length Block Segmentation (Section 4.2);
- Rare Markers Block Segmentation (Section 4.3).

4.1 Fixed Length Block Segmentation

The Fixed Length Block (FLB) is a segmentation strategy designed to divide text into blocks of a fixed length. This approach employs a single Huffman tree that is constructed over the entire dataset to define the codeword set used across all the blocks. By referncing this single Huffman tree, the encoding process remains consistent throughout the entire text. This method is straightforward and adheres closely to the original SFDC scheme, with the primary requirement being the division of text into blocks of fixed length. The use of a single Huffman tree for encoding characters within all blocks ensures that the compression factor remains equivalent to that achieved by the original SFDC. During the text encoding phase, the process under FLB is nearly identical to that of SFDC, with one key difference: at the end of each fixed-length block, there is an opportunity to empty the stack, thereby processing all pending bits of characters that are left waiting. This capability is advantageous for the performance of the representation scheme, as it tends to reduce the average delay experienced by characters within that block. As a result, a reduction in the overall average delay can be anticipated, with this decrease being more pronounced when smaller block sizes are considered. In the decoding phase of FLB, no significant modifications to the main algorithm are necessary. When decoding a specific character, it is sufficient to accurately identify the block in which the character's encoding resides. This identification process can be performed in constant time, ensuring efficiency in the decoding phase. FLB segmentation can be interpreted as a variant of the SFDC scheme. It introduces specific positions at regular intervals within the text where the stack of pending bits can be completely emptied. This insertion of emptying points is a highly efficient mechanism for addressing the challenges posed by LDA.

To summarize, the Fixed Length Block segmentation strategy simplifies the text processing procedure by utilizing fixed-length blocks and a single Huffman tree. This approach ensures consistent compression factors and offers an efficient mechanism for reducing delays in character processing, making it a robust and effective variant of the SFDC scheme in handling large datasets.

4.2 Adaptive Huffman Encoding in Fixed Length Block Segmentation

The idea of Adaptive Huffman Encoding in FLB (AFLB) Segmentation is to create a new Huffman tree for each block obtained from the segmentation of the text. This strategy ensures that the frequency function used for tree construction more accurately reflects the character frequencies within that specific block, thereby enabling more efficient character encoding and consequently reducing the average delay within the block. On the other hand, this method introduces the need to maintain multiple Huffman trees, one for each text block created by the partitioning process. This requirement can potentially complicate the management of resources and increase the computational overhead involved in the compression process.

In AFLB segmentation, let n represent the length of the text in characters, and k denote the block size. The text is divided into $\lfloor n/k \rfloor$ blocks, each containing k characters, with a final block containing the remaining $n \mod k$ characters if n is not divisible by k. Assuming the text uses the ASCII character set (256 elements) and that the average code length for Huffman encoding is approximately 4 bits, the memory requirement for one Huffman tree is around 1KB. This accounts for the storage of each character's Huffman code and the tree structure. The space overhead for maintaining a separate Huffman tree for each block is directly proportional to the number of blocks, amounting to approximately n/k KB. This relationship highlights the additional space required for separate Huffman trees, indicating the space efficiency and scalability of the compression method. However, this overhead can be significantly high in certain scenarios, potentially undermining the effectiveness of the compression scheme.

To mitigate this issue, we propose a strategy that allows for the reuse of the same Huffman tree across two or more adjacent blocks, provided the tree structures obtained from these blocks are identical or sufficiently similar. More formally, this involves constructing the tree for a new block and then checking whether its structure is similar to that of the tree from the previous block. If they are similar, the same tree can be reused; otherwise, a new tree must be constructed. This approach could significantly reduce the number of trees utilized during the compression phase, enhancing the efficiency and manageability of the SFDC encoding scheme in practical applications. This refined approach aims to balance the need for accurate, context-sensitive encoding with the practical considerations of computational efficiency and resource management, making it particularly well-suited for large-scale text analysis and data compression tasks.

In this context, we adopt the *cosine distance metric* to compute the similarity between the trees of two adjacent blocks. Cosine similarity is a critical measure in data analysis, used to determine the similarity between two non-zero vectors in an inner product space. This similarity is defined as the cosine of the angle between the vectors. Due to its ability to measure vector orientation without being affected by vector magnitude, Cosine similarity is essential in several fields. In information retrieval and text mining, it measures document similarity based on content orientation, independent of length. In data mining, it assesses cluster cohesion, replacing Euclidean distance in methods like k-means, especially for text clustering. In machine learning, it aids pattern recognition, classification, and neural network training by determining data object similarity. In the context of AFLB, cosine distance is employed to evaluate the similarity between Huffman trees derived from continuous text blocks. By trans-

forming these tree structures into vector forms we can then apply cosine distance to quantify how similar two trees are.

More formally, assume y_a and y_b are two adjacent text blocks for which we want to compute the similarity of their Huffman trees, T_a and T_b, respectively. Assume also that f_a and f_b are the two character frequency functions of y_a and y_b, respectively, and on which the Huffman trees T_a and T_b are built. The character frequency functions f_a and f_b can be interpreted as vectors over the ordered alphabet Σ. For example, if $\Sigma = \{a, b, c\}$, and the text block y_a contains the characters $\{a, a, b\}$, while y_b contains the characters $\{a, c, c\}$, then the frequency functions f_a and f_b would be defined as $f_a = (2, 1, 0)$ and $f_b = (1, 0, 2)$, where each position in the vectors corresponds to the characters in Σ in order. The distance, denoted as δ, is then expressed as:

$$\delta(T_a, T_b) = \frac{f_a \cdot f_b}{||f_a|| \times ||f_b||}$$

Here, $f_a \cdot f_b$ represents the dot product of the vectors f_a and f_b, and $||f_a||$ denotes the Euclidean norm of the vector f_a. Observe that, in this case, the value of such distance lies in the range $[0, 1]$, where 1 indicates that the vectors are proportional (identical direction), and 0 indicates that they are orthogonal (no similarity).[3] We say that two Huffman trees are similar if their distance is below or equal to a threshold γ, which in our paper is identified by the average cosine distance between all adjacent text blocks. Assuming the block size is K and $N = n/K$ is the number of blocks, we have

$$\gamma = \frac{1}{N-1} \sum_{i=0}^{N-2} \delta(T_i, T_{i+1})$$

where T_i and T_{i+1} are the Huffman trees for the i-th and $(i + 1)$-th text blocks, respectively.

It is important to consider that the cosine similarity measure can only be applied if the two blocks refer to the same character alphabet. If the second block contains characters that do not appear in the first block and, consequently, do not have defined paths in the Huffman tree, reuse of the tree is not feasible. In such circumstances, a new Huffman tree must be constructed to accommodate the unique characters present in the second block. The Adaptive Fixed-Length Block segmentation approach presented in this section, while requiring the maintenance of multiple Huffman trees, offers the potential for tailored compression for each block of text. This could lead to a reduction in the total space required for text compression. However, it is crucial to identify an optimal balance between the size of the blocks (and thus their number in the segmentation) and the overall space needed to maintain the Huffman

[3] In this paper, we evaluate the similarity between Huffman trees using cosine distance based on the frequency function of characters, rather than the more common approach of using codeword lengths. This method directly reflects the character distribution within the text and relates closely to compression efficiency, as Huffman trees are designed to minimize the weighted path length according to character frequencies. While this frequency-based approach provides a more accurate representation of the data structure it also has drawbacks. It can be more sensitive to minor variations in character distribution and more complex to interpret compared to using codeword lengths.

trees. Achieving this balance is essential for maximizing the efficiency of the AFLB approach.

4.3 Rare Marker Block Segmentation

Previous approaches, such as FLB and AFLB segmentation, perform text segmentation by dividing the text into fixed length blocks. This "blind" division does not consider the most advantageous points for ending one block and starting the next. In this section, we introduce an enhancement to the AFLB approach that segments text into variable-length blocks. Specifically, we present the *Rare Marker Block* (RMB) segmentation, a technique that utilizes the positions of infrequently occurring characters within the text as segmentation markers to define block boundaries. This method aims to improve the efficiency of the SFDC scheme by adjusting block sizes based on the distribution of low-frequency characters.

The RMB segmentation formally identifies characters $c \in \Sigma$ with a frequency $f(c)$ below a predefined threshold, termed *rare markers*. These rare markers are used to determine the points at which the text is segmented into blocks. Thus, the text y is divided into blocks such that each block ends immediately after the next occurrence of any rare marker. This approach ensures that infrequent characters act as natural dividers, optimizing the distribution of text codes within the SFDC layers. To prevent the creation of excessively small blocks when rare markers occur in close proximity, we introduce a parameter $\beta > 0$, which sets a minimum block size. Formally, a block is closed at the position of a rare marker only if the next rare marker is at least β characters away. This means that if two rare markers are found within β characters of each other, they are included in the same block, ensuring that no block is smaller than β characters. This approach helps maintain a more consistent and efficient block size, preventing the inefficiencies associated with handling very small blocks. The RMB segmentation offers several advantages:

- **Efficiency:** The segmentation adapts to the inherent structure of the text, optimizing compression performance by aligning block boundaries with the distribution of low-frequency characters.
- **Scalability:** The method scales effectively with text size and complexity, adjusting dynamically to variations in text composition and character distribution.
- **Simplicity:** The use of clearly defined markers simplifies both the encoding and decoding processes, making the method practical for large datasets.

The RMB segmentation strategy is particularly suited for texts with non-uniform character distributions, such as natural language corpora, where certain characters may appear with significantly lower frequency. This adaptability makes it an excellent choice for efficiently managing diverse text types in real-world applications.

The RMB segmentation strategy offers significant advantages for text compression but also presents several challenges. These include the potential for inconsistent block sizes, which can lead to inefficient data management, and the irregular distribution of rare characters, which may result in suboptimal compression and increased decoding times. Additionally, determining the optimal parameters for identifying rare characters and setting the minimum block size (β) is complex and dataset-specific. Non-optimal parameters can compromise performance. The approach also introduces

Text	σ	Max$\{\lvert\rho(y[i])\rvert\}$	Avg$\{\lvert\rho(y[i])\rvert\}$
protein	25	11	4.22
dblp	96	21	5.26
english	94	20	4.59

Table 1. Some noteworthy details regarding the datasets utilized in our experimental outcomes. All sequences comprise 104,857,600 elements.

computational overhead due to the need for dynamic segmentation and rare character identification, which can be problematic for real-time applications or systems with limited resources. Moreover, the increased implementation complexity compared to traditional fixed-length segmentation methods can lead to higher development and maintenance costs. Despite these challenges, addressing these issues can help fully realize the potential benefits of the RMB approach, enhancing its practical applicability and efficiency.

5 Experimental Results

In this section, we present experimental results that evaluate the performance of the three variants of the SFDC representation, which are based on block segmentation of the dataset. The evaluation criteria include average delay, the number of blocks generated, the number of Huffman trees utilized, and the compression factor achieved.[4]

Our experiments were performed on 3 real data sequences. Such real data sequences are text files of size 100 MB from the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl`), and specifically an XML file, an English text and a protein sequence. Following the notation proposed in [2], we denote by dblp the XML file containing bibliographic information on major computer science journals and proceedings. We denote by english the English text, which contains different English text files with slightly different character distributions. We denote by protein the protein text containing protein sequences (consisting of uppercase letters for the 20 amino acids). Some interesting information about these datasets are shown in Table 1.

The FLB and AFLB strategies are evaluated based on the block sizes used for segmentation, which range from 1 KB to 100 MB. For both approaches, the average delay is measured. In addition to the average delay, the AFLB strategy also includes measurements of the number of Huffman trees used in the representation and the spatial overhead generated by these trees. The RMB strategy, on the other hand, is evaluated based on the number of rare elements used as markers, which varies from 2 to 10. Similar to AFLB, the evaluation of RMB includes measurements of the average delay, the number of Huffman trees used in the representation, and the spatial overhead generated by these trees. By considering these metrics, we aim to provide a comprehensive assessment of the performance of each strategy under varying conditions.

[4] All the implementations of the encodings presented in this paper and all the datasets are available in a Google Drive repository, accessible at `https://drive.google.com/drive/folders/-1e4aPz_TR9m4BIYo5fXOKYQhbLNuW9WG8?usp=sharing`.

	Block Size (in KB)	Avg. Delay		Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
		FLB	AFLB				
Fixed Block	10^0	2.07	1.16	104,858	104,437	76,249,148	112.810 %
	10^1	2.15	1.48	10,486	8,915	9,405,724	13.780 %
	10^2	2.18	1.63	1,049	815	991,310	1.450 %
	10^3	2.17	2.03	105	98	138,124	0.200 %
	10^4	2.16	2.02	11	7	11,520	0.020 %
	10^5	2.84	2.82	2	2	3,334	0.005 %

	Rare Elements	Average Delay	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Rare Marker	2	1.93	59	49	69,746	0.09 %
	4	1.80	223	185	235,690	0.30 %
	6	1.81	323	273	331,508	0.42 %
	8	1.71	643	529	620,926	0.74 %
	10	1.66	1,171	1,002	1,021,286	1.19 %

Table 2. Experimental results obtained on the DBLP text using 6 layers. The results must be evaluated considering the standard version of SFDC shows an average delay equal to 2.19, and that the compressed text has a size of 68.91 MB.

	Block Size (in KB)	Avg. Delay		Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
		FLB	AFLB				
Fixed Block	10^0	10.06	3.71	104,858	104,559	55,760,356	95.420 %
	10^1	63.08	21.36	10,486	9,391	9,156,926	15.470 %
	10^2	502.03	197.77	1,049	655	802,572	1.350 %
	10^3	2,852.33	2,122.98	105	88	129,328	0.220 %
	10^4	16,957.22	12,615.84	11	8	13,070	0.020 %
	10^5	59,829.62	59,016.76	2	2	3,368	0.015 %

	Rare Elements	Average Delay	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Rare Marker	2	9,680.20	24	15	16,650	0.02 %
	4	568.66	263	200	246,444	0.33 %
	6	1,048.50	464	368	415,620	0.56 %
	8	135.82	1,749	1,398	1,275,560	1.47 %
	10	114.41	2,372	1,947	1,621,346	1.83 %

Table 3. Experimental results obtained on the ENGLISH text using 5 layers. The results must be evaluated considering the standard version of SFDC shows an average delay equal to 44,387.30, and that the compressed text has a size of 60.1 MB.

Table 2, Table 3 and Table 4 show the results obtained on the DBLP, ENGLISH and PROTEIN datasets, respectively. From experimental results it turns out that the FLB approach often achieves significant reductions in average delay compared to the original SFDC scheme. Generally, better results are observed with smaller block sizes. This is because the delay induced by rare characters is minimized due to the limited block size; the smaller the block, the lower the maximum potential delay caused by these rare markers. The most notable improvements are observed with the ENGLISH and PROTEIN datasets. For the ENGLISH dataset, using 1KB blocks results in an average delay of 10, an improvement of three orders of magnitude compared to SFDC. For the PROTEIN dataset, the average delay decreases to 0.26, which is 75% less than that achieved by SFDC.

	Block Size (in KB)	Avg. Delay FLB	Avg. Delay AFLB	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Fixed Block	10^0	0.26	0.16	104,858	39,610	6,378,498	11.760 %
	10^1	0.45	0.15	10,486	2,525	417,090	0.760 %
	10^2	0.94	0.21	1,049	289	50,034	0.090 %
	10^3	1.02	0.66	105	54	10,072	0.020 %
	10^4	1.01	0.98	11	4	876	0.012 %
	10^5	1.06	1.07	2	2	448	0.004 %

	Rare Elements	Average Delay	Number of Blocks	Huffman Trees	Tree Size (in Byte)	Space Overhead
Rare Marker	2	0.77	114	70	13,316	0.02 %
	4	0.61	4,244	1,495	258,920	0.34 %
	6	0.45	19,900	12,033	487,114	0.79 %
	8	0.39	22,794	19,458	863,314	1.26 %
	10	0.21	35,395	31,147	1,223,612	1.96 %

Table 4. Experimental results obtained on the PROTEIN text using 5 layers. The results must be evaluated considering the standard version of SFDC shows an average delay equal to 1.02, and that the compressed text has a size of 55.36 MB.

In the case of AFL segmentation, the results in terms of average delay are significantly better than those obtained with FLB. This improvement is due to the adaptive approach, which allows for more tailored and efficient coding for each block. The most substantial improvements are seen with smaller block sizes, with reductions of up to 60%, although this advantage diminishes as block size increases. However, AFLB must contend with the overhead of maintaining multiple Huffman trees. For 1KB block sizes, this overhead becomes prohibitive, reaching almost 113% of the compressed text size for the DBLP dataset. Therefore, larger block sizes are necessary to achieve a reasonable trade-off, allowing the overhead to decrease to around 1%. This optimal block size is 100 KB for both DBLP and ENGLISH and 10 KB for PROTEIN. Additionally, it is observed that utilizing the Huffman tree reuse technique based on cosine similarity can reduce the number of required trees by up to 20%. This reduction diminishes gradually as the number of blocks increases, justified by the fact that small adjacent blocks tend to have minimal variations in character frequency distribution. This trend is consistent across all datasets.

Lastly, the performance of the RMB approach did not meet expectations. Although this technique offers considerable improvements and achieves low spatial overhead, the average delay values are generally higher than those provided by AFLB for an equivalent number of blocks. This is attributed to the highly variable block sizes created by the RMB technique. In this context, very large blocks fail to deliver good performance in terms of delay, reducing the overall effectiveness of the approach.

In conclusion, for the DBLP dataset, the best performance is achieved with AFLB using 100KB blocks, resulting in an average delay of 1.63 with only a 1.45% increase in spatial overhead. For the ENGLISH dataset, the optimal performance is obtained with FLB using 1KB blocks, achieving an average delay of 10. Finally, for PROTEIN, the best results are achieved with AFLB using 10KB blocks, resulting in an average delay of 0.15 with less than a 1% increase in spatial overhead.

6 Conclusions and Future Work

In this article, we have explored three primary text compression strategies: Fixed-Length Block (FLB) segmentation, Adaptive Fixed-Length Block (AFLB) segmentation, and Rare Marker Block (RMB) segmentation. Each approach offers unique benefits and addresses different aspects of the text compression challenge. Looking forward, several promising directions for future research have been identified. One avenue is to investigate the use of rare markers as starting points of blocks rather than ending points. This adjustment could potentially optimize the segmentation process further and reduce the delay introduced by rare characters. Additionally, exploring the efficacy of a First-In, First-Out (FIFO) strategy as opposed to the Last-In, First-Out (LIFO) strategy currently used could provide insights into improving the decoding efficiency.

References

1. N. Brisaboa, S. Ladra, and G. Navarro: *Directly addressable variable-length codes*, in SPIRE 2009, vol. 5721 of LNCS, Springer, 2009, pp. 122–130.
2. N. R. Brisaboa, S. Ladra, and G. Navarro: *Dacs: Bringing direct access to variable-length codes.* Inf. Process. Manag., 49(1) 2013, pp. 392–404.
3. D. Cantone and S. Faro: *The many qualities of a new directly accessible compression scheme.* CoRR, abs/2303.18063 2023.
4. P. Elias: *Efficient storage and retrieval by content and address of static files.* J. ACM, 21(2) 1974, pp. 246–260.
5. P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds.* Theor. Comput. Sci., 372(1) 2007, pp. 115–121.
6. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM, 2003, pp. 841–850.
7. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the Institute of Radio Engineers, 40(9) September 1952, pp. 1098–1101.
8. A. Moffat and L. Stuiver: *Binary interpolative coding for effective index compression.* Inf. Retr., 3(1) 2000, pp. 25–47.
9. J. Teuhola: *Interpolative coding of integer sequences supporting log-time random access.* Inf. Process. Manag., 47(5) 2011, pp. 742–761.

On Practical Data Structures for Sorted Range Reporting

Golnaz Badkobeh[1], Sehar Naveed[2], and Simon J. Puglisi[3]

[1] Department of Computer Science, City University of London, United Kingdom
golnaz.badkobeh@city.ac.uk
[2] Department of Computer Science, Goldsmiths University of London, United Kingdom
snave001@gold.ac.uk
[3] Department of Computer Science, University of Helsinki
Helsinki Institute for Information Technology (HIIT), Helsinki, Finland
simon.puglisi@helsinki.fi

Abstract. Given an array $A[1, n]$ of integers, the sorted range reporting problem is to preprocess A in order to later answer queries of the form sort(i, j), which should return an array of length $j-i+1$ that contains the contents of $A[i, j]$ sorted in ascending order. When applied to the suffix array, sorted range reporting can be applied to solve several string processing problems, including, for example, non-overlapping pattern matching and variable-length gapped pattern matching. In this paper we explore the practical performance of solutions for sorted range reporting. Our experiments show that the choice of solution depends on the interval size. For very short intervals, manually sorting the range at query time beats asymptotically optimal methods, while for longer intervals, a data structure that we describe that precomputes sorted blocks of varying sizes which are then merged at query time is the method of choice.

Keywords: sorted range reporting, suffix array, gapped pattern matching

1 Introduction

The suffix array, $SA[1, n]$, of a string $T[1, n]$ is a permutation of the integers $[1, n]$ such that suffix $T[SA[i]..n]$ is the ith suffix in lexicographical order amongst all suffixes of T. For example, the suffix array of string

$$T = actagtatctcccgtagtac\$$$

is

$$SA = 21, 19, 1, 16, 4, 7, 20, 11, 12, 13, 2, 9, 17, 14, 5, 18, 15, 3, 6, 10, 8$$

Many string processing problems can be solved efficiently both in theory and in practice using the suffix array. A property of the SA that makes it useful for many problems is that all positions of occurrence of any substring that occurs in T are contiguous in SA. For example, in the above string, the substring ta occurs at positions 3, 6, 15, and 18 — and these positions occur together in SA, in the subarray $SA[16, 19] = 18, 15, 3, 6$. This is a consequence of placing the suffixes in lexicographical order. Observe, however, that while the occurrences of the substring do form a contiguous range in SA, the lexicographical order also means that in general the positions do not necessarily occur in the same order as they are in the string (string order).

For several important string processing problems it is desirable (even necessary) to have the occurrences in string order. A compelling example from computational

Golnaz Badkobeh, Sehar Naveed, Simon J. Puglisi: On Practical Data Structures for Sorted Range Reporting, pp. 42–49.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

biology is the variable-length gap pattern matching (VLG) problem [5], in which we seek matches to a regex-like pattern that is composed of subpatterns (strings) and bounds on the distance allowed between subpatterns. For example, *ta*[2,6]*ac* asks for positions in the string where an occurrence of *ta* is followed by an occurrence of *ac* at least 2 positions, but at most 6 positions away.

The suffix array allows us to easily find the occurrences of the two subpatterns, but it is not until we sort those occurrences into ascending order that we can easily apply the gap constraint to arrive at the answer (there is one such occurrence in the example string above, starting at position 15).

Another example is the *non-overlapping pattern matching* problem, given a pattern, the task is to output a maximal set of pattern occurrences in T that do not overlap each other. Clearly, being able to enumerate pattern occurrences in string order would be advantageous. Another compelling set of problems comes from document retrieval [9].

In this paper we examine methods that allow subarrays of an integer to be returned in sorted order—a problem known in the literature as *sorted range reporting* [4]. The methods we will consider can be applied to general arrays of integers, not necessarily the suffix array—our motivation, however, is to eventually apply these methods to problems in string matching, in particular VLG matching. Our focus is on methods that are performant in practice. We find that manually sorting the query range with each query is often faster in practice than an asymptotically optimal data structure due to Brodal et al. [4]. We also describe an alternative data structure to Brodal et al. [4] which presorts blocks of the input array and merges a relevant subset of them covering the query. Our experiments show our method beats the above methods on all but the short query ranges.

1.1 Related Work

The sorted range reporting problem is stated formally as follows. We are given an array, $A[1, n]$ of integers in the range $[1, n]$. We are allowed to preprocess the array and build a data structure to later help us answer queries of the form $\text{sort}(i, j)$ that should return an array containing the contents of $A[i, j]$ sorted in ascending order.

The problem was first directly studied by Mäkinen and Navarro [8], and later by Navarro and Nekrich [10]. The related problem of range predecessor queries was studied by Belazzougui and Puglisi [3]. Their $O(n)$ space $O(\sqrt{\log n})$-time query data structure implies a solution to sorted range reporting with query time $O(k\sqrt{\log n})$, where $k = j - i + 1$ is the size of the query range.

Brodal et al. [4] describe an optimal solution to sorted range reporting, proposing a RAM model data structure that takes $O(n)$ words of space and allows for queries to be answered in optimal $O(k)$ time, i.e., the query time is directly proportional to the number of elements reported. The preprocessing phase, which organizes the data to enable these efficient queries, also takes $O(n \log n)$, aligning with the optimal sorting time for n elements. We examine this data structure further in Section 2.2.

Another related problem is range selection queries (also known as range quantile queries). Akram and Saxena [2] present a linear space solution with $O(k \log k)$ query time and $O(n)$ preprocessing time, reporting output elements in non-decreasing order, using a data structure consisting of an RMQ structure and a binary min-heap and offering a straightforward and easy-to-implement alternative to the solution by Brodal

et al [4]. A broader range selection problem, where the output elements are not required to be reported in sorted order, is previousy studied in [1]. Skala [11] provides a comprehensive survey of array range query problems.

2 Sorted Range Reporting

In this section we describe the three solutions to the sorted range reporting problem that we implemented and measured in this study.

2.1 Simple Sorted Range Reporting

Perhaps the simplest approach to sorted range reporting one can think of is to just copy and sort the query range in order to answer each query. This method represents a natural baseline against which the performance of more sophisticated data structures for sorted range reporting should be measured.

We implemented this baseline, which we refer to as Simple Sorted Range Reporting (ssrr) as follows. For each index in the range from i to j (inclusive), the elements of A are copied to an array vector v. Once the elements are copied, the contents of v are sorted using the `std::sort` function and then v is returned.

The running time is $O(k \log k)$, where $k = j - i$ is the length of the query range.

Algorithm 1 Simple Sorted Range Reporting

```
function QUERY(i, j, v)
    Step 1: Copy elements from A to v within the specified range [i, j]
    std::copy( A.begin() + i, A.begin() + j + 1, v.begin())
    Step 2: Sort the elements in the v vector
    std::sort(v, j − i + 1)
    return v
end function
```

2.2 Brodal et al.'s Data Structure

We implemented the asymptotically optimal sorted range reporting approach of Brodal et al. [4]. To our knowledge this is the first implementation of their data structure. In the interest of being self contained, we provide an overview of the data structure below, but refer the reader to [4] for full details.

We construct local rank labellings for each $r \in \{0 \cdots \log\log n\}$ as follows (the rank of an element x in a set X is defined as $|\{y \in X | y < x\}|$). For each r, the input array is divided into $\lceil n/2^{2^r} \rceil$ consecutive subarrays each of size 2^{2^r} (except possibly the last subarray), and for each element $A[x]$ the r'th local rank labelling is defined as its rank in the subarray $A[\lfloor x/2^{2^r} \rfloor 2^{2^r} \cdots (\lfloor x/2^{2^r} \rfloor + 1)2^{2^r} - 1]$. Thus, the r'th local rank for an element $A[x]$ consists of 2^r bits. All local rank labels of length 2^r can be stored using space $O(n2^r + w)$ bits, where by w we denote the word length in bits, and assume that $w \geq logn$. For all $\lceil \log\log n \rceil$ local rank labellings, the total number of bits used is $O(w \log\log n + n \log n) = O(nw)$ bits. All local rank labellings can be built in $O(n \log n)$ time while performing mergesort on A. The r'th structure is built by writing out the sorted lists, when we reach level 2^r . Given a query for $k = j - i + 1$ elements, we find the r for which $2^{2^r} - 1 < k \leq 2^{2^r}$. Since each subarray in the r'th local rank labelling contains 2^{2^r} elements, we know that i and j are either in the same

or in two consecutive subarrays. If i and j are in consecutive subarrays, we compute the start index of the subarray where the index j belongs, i.e. $x = \lfloor j/2^{2^r} \rfloor 2^{2^r}$. We then radix sort the elements in $A[i, x-1]$ using the local rank labels of length 2^r. This can be done in $O(k)$ time using two passes by dividing the 2^r bits into two parts of 2^{r-1} bits each, since $2^{2^{r-1}} < k$. Similarly we radix sort the elements from $A[x, j]$ using the labels of length 2^r in $O(k)$ time. Finally, we merge these two sorted sequences in $O(k)$ time, and return the k smallest elements. If i and j are in the same subarray, we just radix sort $A[i, j]$.

2.3 Sorted Range Reporting via Presorted Blocks

We now describe a simple alternative to Brodal et al.'s data structure that takes $O(n \log n)$ words of space and answers queries in $O(k \log k)$ time, where k is the length of the query range. The structure and accompanying query algorithm are very simple, but, to our knowledge, novel.

2.4 Data Structure

We assume for ease of exposition that n is a power of 2 (the general case is easy to deal with in practice and without affecting asymptotic bounds).

The data consists of $\log n$ levels, each an array of n elements. Denote the array at the ith as L_i. At the ith level, the input array A is conceptually divided into $n/2^i$ blocks, each of length 2^i. At each level, the elements of each block are sorted and concatenated to form an array of length n and thus the overall data structure size is $n \log n$ words and construction time is $O(n \log^2 n)$. We keep an array of $\log n$ pointers so that the start of each level can be accessed in constant time. At a given level, the start of the jth block can be access in constant time at position $j2^i$.

2.5 Query Algorithm

To answer queries, we proceed in two phases. The first phase identifies a set of presorted blocks from different levels of the data structure that allow the range to be covered. For query range $[i, j]$, the cover can be determined recursively as follows. Let $k = j - i + 1$. We first determine the largest full block contained in the range. This takes $\log k$ time by simply testing each block size that can possibly fit in the range. We then recurse on the uncovered prefix and suffix.

We remark that to save space, lower levels of the data structure, where blocks are very short, can be omitted from the data structure and reconstructed on demand at query time. This idea is illustrated in Fig. 1.

The second phase of the query algorithm merges the contents of the sorted blocks that cover the query range as determined in the first phase. There are $O(\log n)$ ranges to merge. We allocate an output buffer of k elements, which will eventually contain the elements of query range sorted in ascending order. We insert pointers to the start of each of ranges into a min heap with key equal to the element at the start of each range. A standard multi-way merge is then performed: we extract min from the heap, add the element to the output array, increment the pointer to point to the next element of the block, and then (if elements remain in the block) reinsert the block to the heap with the next element as the sort key. This process is repeated until the output buffer is full. Fig. 2 shows an example.

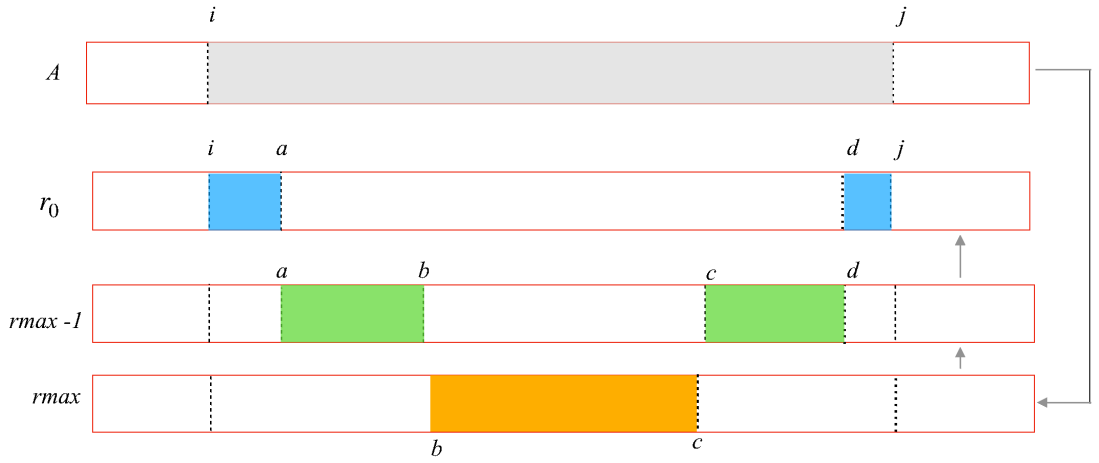

Figure 1. Block selection process in Block Sorted Range Reporting data structure. The orange and green ranges are the presorted blocks that best cover the range as identified in the first phase of the query algorithm. Blue areas at the start and end are manually sorted. Then all five regions are merged and returned in answer to the query.

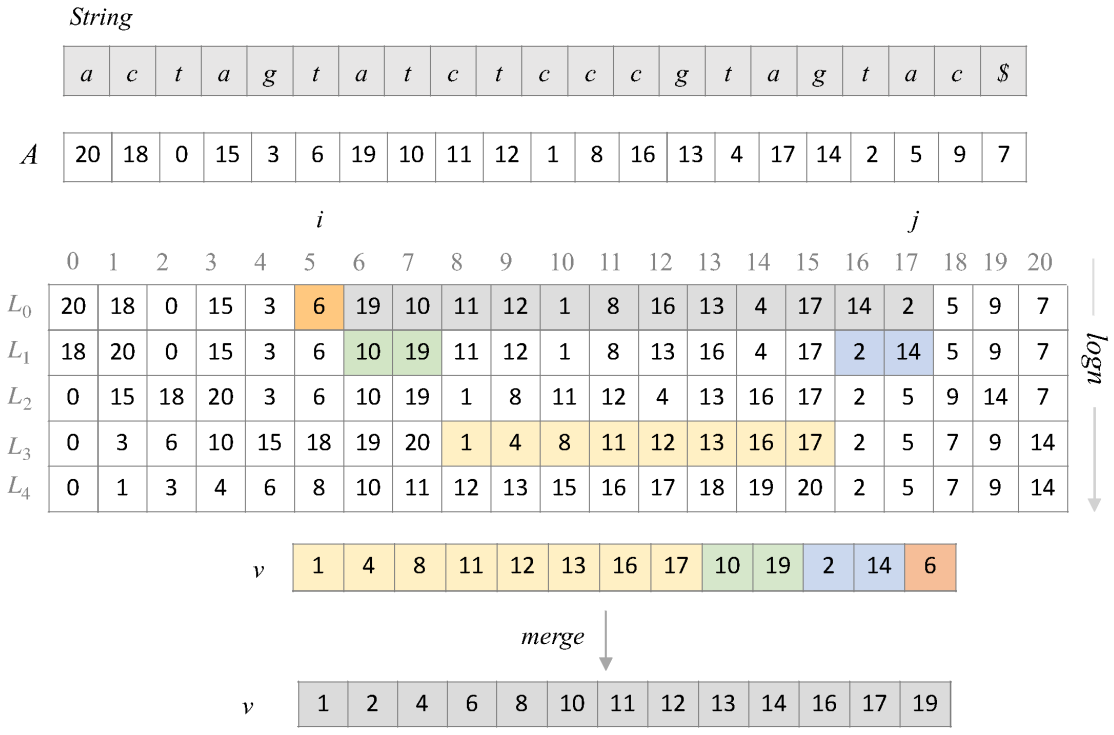

Figure 2. An example of using Block Sorted Range Reporting to report the $k = 13$ elements in sorted order for the text *actagtatctcccgtagtac* for a query $[i, j] = [5, 17]$.

3 Experimental Evaluation

We implemented the three methods described in the previous section and measured there performance in experiments described below.

3.1 Experimental setup

Environment. Our test machine comprises an Intel Core i7-4790 CPU @ 3.60GHz with a CPU cache size of 8 MiB (1 instance) and a total memory of 23Gi operated on Ubuntu 24.04 with an x86_64 architecture, running kernel version 6.8.0-31-generic and employing g++ 13.2.0 as the compiler version.

Test data. We generated suffix arrays from the texts below, from the Pizza & Chilli Corpus, which are available at `https://pizzachili.dcc.uchile.cl/texts.html`. Each text is 100MiB long. A brief description of each follows:

- *english* is the concatenation of English text files selected from various collections of Gutenberg Project. The file *english* are prefixes of the original text.
- *proteins* dataset comprises newline-separated protein sequences, devoid of descriptions, sourced from the Swissprot database. Each of the 20 amino acids is encoded as a single uppercase letter.
- *Para* dataset, provided by the Saccharomyces Genome Resequencing Project contains 36 sequences of Saccharomyces paradoxus. It includes four nucleotide bases: A, C, G, T.
- *kernel* dataset comprises the source code for all 332 Linux kernel versions downloaded from kernel.org. The dataset exhibits high redundancy, as only minor modifications are present between subsequent versions.
- *ecoli* is a dataset for protein localization. It contains 336 E.coli proteins split into 8 different classes.

Queries. Our experiments aimed to gauge the effect of query range size on query time. The queries were randomly generated with the fixed query lengths: $k_{100} = 100$, $k_{500} = 500$, $k_{1k} = 1000$, $k_{5k} = 5000$, $k_{10k} = 10000$, $k_{50k} = 50000$, where k is defined as $k = j - i + 1$, totaling a set of 10,000 queries for each separate run. To ensure that the evaluation covered a wide spectrum of potential real-world scenarios, a set of *mixed* random queries of varied lengths was also generated to test the performance and efficiency of the multi-level data structure.

3.2 Results

Measured runtimes are shown in Table 1. The experimental results demonstrate that the block-sorted range reporter (bsrr) significantly outperforms both simple sorted range reporter (ssrr) and the asymptotically optimal sorted range reporter (srr) across all datasets and query lengths. The performance of all methods is quite stable. High constant factors in the asymptotically optimal srr method cause it to yield to the others.

The gap between bsrr and ssrr widens as the query length increases, to about a factor of four. bsrr reports markedly better performance in handling mixed query lengths compared to ssrr and srr, achieving speeds that are three times as fast as ssrr and almost five times faster than srr.

Dataset	Query Lengths						
	k_{100}	k_{500}	k_{1k}	k_{5k}	k_{10k}	k_{50k}	mixed
english							
bsrr	**1.3691**	**5.6404**	**11.177**	**58.5259**	**120.516**	**597.531**	**129.371**
ssrr	1.9743	12.1864	26.6443	173.665	357.007	2004.59	412.025
srr	3.5908	27.1614	53.4557	263.955	520.584	2723.81	592.948
proteins							
bsrr	**1.3915**	**5.7007**	**11.1467**	**58.3191**	**120.345**	**597.666**	**129.51**
ssrr	1.9881	12.1868	28.2491	161.127	349.596	2002.54	411.85
srr	3.5665	27.0364	53.3736	263.589	520.454	2724.27	592.913
para							
bsrr	**1.3946**	**5.6618**	**11.4455**	**58.8152**	**121.534**	**608.432**	**131.641**
ssrr	2.0393	11.4455	27.4848	166.621	359.859	2056.76	424.777
srr	3.5805	27.2729	53.5191	262.754	519.137	2723.27	593.17
kernel							
bsrr	**1.2543**	**5.2102**	**9.912**	**53.956**	**111.399**	**582.988**	**123.459**
ssrr	1.9395	12.0159	26.5149	160.357	347.817	1985.71	408.186
srr	3.5612	27.0947	53.442	262.831	518.654	2698.25	588.144
E.coli							
bsrr	**1.4047**	**6.1667**	**11.3705**	**61.4457**	**122.473**	**610.411**	**132.034**
ssrr	2.0352	12.5351	27.4683	167.365	372.692	2068.67	423.631
srr	3.5742	27.0499	53.3785	263.879	518.184	2729.38	593.692

Table 1. The average query time in microseconds for the three methods block-sorted range reporter (bsrr), simple sorted range reporter (ssrr) and the asymptotically optimal sorted range reporter (srr) for query lengths $k_{100} = 100$, $k_{500} = 500$, $k_{1k} = 1000$, $k_{5k} = 5000$, $k_{10k} = 10000$, $k_{50k} = 50000$ and *mixed* random queries of varied lengths over a set of 10,000 queries.

4 Conclusions

This paper has explored the performance of sorted range reporting structures in practice. Our experiments have shown that our implementation of the asymptotically optimal structure, srr, of Brodal et al. [4] is beaten by both a simple method, ssrr, that simply sorts the contents of the query range each time, and by a data structure we introduce, bsrr, that presorts blocks of different sizes and merges these blocks at query time. This later method is consistently faster than the other methods. We remark that srr and bsrr complement each other and can be easily combined by leaving out the lower levels of the bsrr structure and simply resorting to sorting for short ranges.

In future work, we plan to sparsify our bsrr data structure so that it only includes some levels. This may require more merging to be done for some queries, but is an effective way to reduce data structure size. Moreover, we can reasonably expect a higher-degree merge to not adversely effect overall query time, as blocks are still few and processing of their contents will remain cache efficient. We aim to include an exploration of this approach in the full version of this paper.

Finally, one can further reduce the space usage of the bsrr data structure by storing sorted blocks in a compressed representation designed for sorted sequences, such as Elias-Fano [6,7] or by storing gaps between elements and using variable length codes. This should be particularly effective for large blocks.

References

1. P. Afshani, G. S. Brodal, and N. Zeh: *Ordered and unordered top-k range reporting in large data sets*, in Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2011, pp. 390–400.
2. W. Akram and S. Saxena: *Sorted range reporting and range minima queries.* arXiv preprint arXiv:2104.02461, 2021.
3. D. Belazzougui and S. J. Puglisi: *Range predecessor and lempel-ziv parsing*, in Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, R. Krauthgamer, ed., SIAM, 2016, pp. 2053–2071.
4. G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz: *Online sorted range reporting*, in Algorithms and Computation: 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings 20, Springer, 2009, pp. 173–182.
5. M. Cáceres, S. J. Puglisi, and B. Zhukova: *Fast indexes for gapped pattern matching*, in SOFSEM 2020: Theory and Practice of Computer Science: 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20–24, 2020, Proceedings 46, Springer, 2020, pp. 493–504.
6. P. Elias: *Efficient storage and retrieval by content and address of static files.* Journal of the ACM (JACM), 21(2) 1974, pp. 246–260.
7. R. M. Fano: *On the number of bits required to implement an associative memory*, Massachusetts Institute of Technology, Project MAC, 1971.
8. V. Mäkinen and G. Navarro: *Position-restricted substring searching*, in LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings, J. R. Correa, A. Hevia, and M. A. Kiwi, eds., vol. 3887 of Lecture Notes in Computer Science, Springer, 2006, pp. 703–714.
9. G. Navarro: *Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences.* ACM Computing Surveys, 46(4) 2014, p. article 52, 47 pages.
10. Y. Nekrich and G. Navarro: *Sorted range reporting*, in Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings, F. V. Fomin and P. Kaski, eds., vol. 7357 of Lecture Notes in Computer Science, Springer, 2012, pp. 271–282.
11. M. Skala: *Array Range Queries*, Springer Berlin Heidelberg, 2013, p. 333–350.

A Quantum Circuit for the Cyclic String Matching Problem

Arianna Pavone[1] and Caterina Viola[2⋆]

[1] Dipartimento di Matematica e Informatica, Università degli Studi di Palermo
Via Archirafi 34, 90123, Palermo, Italy
ariannamaria.pavone@unipa.it
[2] Dipartimento di Matematica e Informatica, Università degli Studi di Catania
Viale A. Doria n.6, 95125, Catania, Italy
caterina.viola@unict.it

Abstract. The cyclic string matching problem aims to detect an occurrence of any cyclical shift of a given sequence of characters within a (usually larger) one. This variant of the (classical) string matching problem arises in several real-world problems, such as DNA analysis and spoken natural language recognition, in which it is necessary to deal with sequences that lack a precise beginning or end and are equivalent up to cyclical shifting. We present a quantum algorithm, in the form of a quantum circuit, for solving the cyclic string matching problem. This algorithm requires quadratically fewer time steps than the most efficient counterpart algorithm running on a classical machine. Additionally, we provide a practical implementation of the presented algorithm using the Qiskit toolkit[1].

Keywords: quantum computing, non-standard text searching, text processing

1 Introduction

In the field of computing, the challenge of pattern identification in textual data spans various disciplines, including natural language processing, information retrieval, and computational biology. Known as *string matching*, this task involves identifying every occurrence of a specified pattern x, of length m, within a text y, of length n. Both sequences are composed of characters drawn from an alphabet Σ, with a size of σ symbols. This field encompasses both *exact* and *approximate* matching.

In *exact string matching*, the goal is to find exact alignments of the pattern within the text. *Approximate string matching* allows for some discrepancies, using a distance function to measure how closely a substring matches the pattern. Various methods exist for approximate matching, including Hamming distance, Levenshtein distance, order-preserving string matching, and Cartesian tree pattern matching. These methods are widely used across different applications, such as genetic analysis, data compression, and natural language processing.

In this paper, we focus on a specific approximation problem known as the *cyclic string matching* problem [18]. Here, the pattern is a *cyclic string*, meaning it can appear in the text in its original form or any of its cyclic rotations. For example, the strings *accba*, *cbaac*, and *baacc* are all variations of the same cyclic string.

⋆ A. Pavone is supported by PNRR project ITSERR - Italian Strengthening of the ESFRI RI RESILIENCE.
C. Viola is supported by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, affiliated to Spoke 10., co-founded by the European Union - NextGenerationEU.

[1] Qiskit is an open-source software development framework for working on utility-scale quantum computers, developed by IBM.

Arianna Pavone, Caterina Viola: A Quantum Circuit for the Cyclic String Matching Problem, pp. 50–70.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

Cyclic strings are significant in both computational and mathematical contexts, particularly within combinatorics. They lack distinct initial or terminal positions, resembling a continuous loop. This property makes cyclic strings particularly interesting and useful in various scientific and practical applications.

Cyclic rotations have various applications, including, for instance, image and signal processing, where they can be employed to perform cyclic shifts on images or signals, such as image rolling [16] or time delay of a signal. Cyclic rotations can be also used to design efficient algorithms sorting data [20]. In addition, sequences admitting cyclic rotations are also relevant in various biological contexts, including viruses [28,9] and bacteria [26]. Thus, the analysis of organisms with a cyclic structure can benefit from algorithms designed for strings that allow for cyclic rotations [17].

The *Cyclic String Matching* problem has numerous applications, including: Bioinformatics, in identifying repeated motifs or conserved sequences within genomic data; Data Compression, in detecting and removing repetitive patterns to enhance compression; Pattern Recognition, in identifying and comparing cyclically repeating patterns; Natural Language Processing, in analyzing literary texts for cyclically repeating phrases or word sequences; Control and Monitoring Systems, in monitoring biological signals or recognizing fault patterns in machinery.

In classical computational models, solutions to the *Cyclic String Matching* problem can be achieved in linear time [18].

1.1 Quantum Computing and String Matching

Quantum computing represents an innovative frontier in computer science, leveraging the principles of quantum mechanics to develop advanced computing systems that differ significantly from classical models. Unlike classical computers, which process information using discrete binary bits confined to states 0 or 1, quantum computing utilizes *quantum bits* or *qubits*. Qubits can exist in a *superposition* of states, allowing them to represent multiple values simultaneously. Additionally, *entanglement* enables two or more qubits to perform correlated operations, enhancing computational power.

Over the past few decades, quantum computing has emerged as a transformative technology with the potential to revolutionize various fields, including cryptography and materials science. This nascent field has achieved significant milestones, highlighting its ability to solve problems once considered intractable for classical computers. One of the most notable achievements is the demonstration of quantum supremacy, where quantum computers have outperformed the world's most powerful supercomputers on specific tasks.

Quantum algorithms have made substantial progress, particularly in optimization and simulation tasks. Grover's algorithm [14] and Shor's algorithm [25] provided early examples of quantum advantage, offering quadratic speedup in database searching and exponential speedup in integer factorization, respectively. These theoretical advancements have paved the way for practical applications in secure communications and complex problem-solving.

Recent studies [7] have successfully applied quantum computation to tackle NP-complete problems, which are notoriously challenging for classical algorithms [27]. In natural language processing, classifiers that integrate quantum and classical computing techniques are expected to have a substantial impact, especially in areas such as classification [6].

As quantum hardware continues to evolve and algorithms become more sophisticated, the integration of quantum computing into broader scientific and industrial applications becomes increasingly feasible. This progression promises not only to expand our computational capabilities but also to redefine problem-solving paradigms across numerous disciplines.

Recent research has focused on applying quantum computing to string matching, exploring the potential for sub-linear time complexities. Notably, Ramesh and Vinay [23] combined Grover's search with parallel string matching to develop a quantum algorithm with $\tilde{\mathcal{O}}(\sqrt{n})$ complexity[2]. Montanaro [19] highlighted the substantial gap between quantum and classical complexities in this context. In 2021, Niroula and Nam [21] proposed a quantum circuit model solution with $\tilde{\mathcal{O}}(\sqrt{n})$ complexity, advancing the field significantly.

In the field of quantum computation, the cyclic rotation of the states of a given register of qubits has been effectively used in solutions for text processing [21,8,13], and specifically for exact and approximate string matching [11,12]. The recent algorithm by Niroula and Nam [21] makes clever use of cyclic rotations of the registers encoding the input strings to achieve a superposition of all their possible alignments and to perform a parallel comparison against the pattern.

This idea was later generalized by Cantone *et al.* in [8] to get a quantum solution to the string matching problem allowing for swaps of adjacent characters, which is more time-efficient than the best known classical counterpart. In their paper, Niroula and Nam provide insight into the fact that a circuit performing a cyclic rotation of the states of a given register of qubits can be executed in time $\mathcal{O}(\log(n))$. The basic idea is that at each step of the algorithm that accomplishes the permutation, it is possible to place at least half of the qubits that still need to be moved to their final position. Since the number of qubits to be placed decreases by at least half at each iteration, $\mathcal{O}(\log(n))$ steps are needed to achieve the target permutation.

1.2 Our Contribution

We propose a novel quantum solution to the *Cyclic String Matching* problem, building on the algorithm by Niroula and Nam [21] to handle cyclically rotated patterns. Our approach utilizes the circuit-based quantum model, making it compatible with modern quantum computing platforms. We implement the algorithm using Qiskit, allowing for testing on both quantum simulators and actual quantum computers.

Our method offers significant computational efficiency, with a time-complexity of $\tilde{\mathcal{O}}(\sqrt{nm})$, where m and n are the lengths of the pattern and text, respectively. This represents a substantial speed-up over classical algorithms, particularly for patterns of constant length, promising enhanced performance in practical applications.

The paper is organized as follows. In Section 2 we introduce the basic notions and some essential notations for our discussion. In Section 3 we recall the construction by Pavone and Viola [22] of a quantum circuit for the cyclic shifting. In Section 4 we present the quantum algorithm for Cyclic String Matching and discuss its computational complexity. We draw our conclusions in Section 5.

In Appendix A, a simple implementation of the new algorithm using Qiskit is provided.

[2] The big-$\tilde{\mathcal{O}}$-notation works as the big-$\mathcal{O}$-notation except for hiding polylogarithmic factors

2 Preliminaries

2.1 Cyclic String Matchings

Given a string x, of length $n \geq 0$, we represent it as a finite array $x[0\,..\,n-1]$. The empty string is denoted by ε. We denote by $x[i]$ the $(i+1)$-st character of x, for $0 \leq i < n$, and by $x[i\,..\,j]$ the substring of x contained between the $(i+1)$-st and the $(j+1)$-st characters of x, for $0 \leq i \leq j < n$. For ease of notation, the $(i+1)$-th character of the string x will also be denoted by the symbol x_i so that $x = x_0 x_1 \ldots x_{n-1}$. Throughout our paper, we assume that all the strings we deal with have length equal to a power of 2. This can be assumed without loss of generality. Indeed, given a character string of length n, we can always add to the string a suitable number of copies of a special character outside of the alphabet so as to get a string of length 2^p for some $p \in \mathbb{N}$. A leftward *circular shift* (or rotation) of a string $x = x_0, \ldots, x_{m-1}$ by s positions is the string $R_s(x) = x_{(m-s) \mod m}, x_{(m-s+1) \mod m} \cdots, x_{(m-s+m-1) \mod m}$ for $0 \leq s < m$. There are m distinct cyclic rotations of x; more precisely, there is a cyclic rotation of x by i positions to the right for each $i \in \{0, \ldots, m-1\}$.

Let x and y be two strings of length m and n, respectively, with $m < n$. Cyclic String Matching is the problem of finding a substring of y that matches any leftward cyclic rotation of x; in other words, the objective of the problem is to find $s \in \{0, \ldots, m-1\}$ and a position $j \in \{0, \ldots, n-1\}$ such that $R_s(x)$ matches $y[j..j+m-1]$, that is $x[m-s+i] = y[j+i]$ for $0 \leq i < m$.

2.2 Quantum Computation

The fundamental unit in quantum computation is the *qubit* (or *quantum bit*). A qubit is a coherent superposition of the two orthonormal computational basis states, which are denoted by $|0\rangle$ and $|1\rangle$, using the conventional *braket* notation. Formally, a single qubit is an element from the *state space* $\mathcal{H}$, that is the two-dimensional Hilbert space on the complex numbers equipped with the inner product; therefore, the mathematical expression of a qubit $|\psi\rangle$ is a linear combination of the two basis states, i.e. $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where the values α and β are called *amplitudes*, are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$, representing the probability of measuring the qubit in the state $|0\rangle$ or $|1\rangle$, respectively. A *quantum measurement* is the only operation giving access to the information on the state of a qubit; however, this operation causes the qubit to collapse to one of the two basis states.

The symbols $|+\rangle$ and $|-\rangle$ denote the quantum states $\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$ and $\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$, respectively. Both of them are uniform superpositions of the two computational basis states $|0\rangle$ and $|1\rangle$; howbeit, $|+\rangle$ and $|-\rangle$ differ in their respective *relative phase*, which is positive for $|+\rangle$ and negative for $|-\rangle$.

Multiple qubits taken together are referred to as *quantum registers*. A quantum register $|\psi\rangle = |q_0, q_1, \ldots, q_{n-1}\rangle$ of size n is an element from the tensor product of n state spaces, $\mathcal{H}^{\otimes n}$, and thus it is expressed as a linear combination of the 2^n states in $\{0,1\}^n$, that is $|\psi\rangle = \sum_{k=0}^{2^n-1} \alpha_k\,|k\rangle$, where the values α_k represent the probability of measuring the register in the state $|k\rangle$, and $\otimes$ denotes the tensor product. If k is an integer value that can be represented by a binary string of length n, we use the symbol $|k\rangle$ to denote the register of n qubits such that $|k\rangle = \bigotimes_{i=0}^{n-1} |k_i\rangle$, where $|k_i\rangle$ takes the value of the i-th least significant binary digit of k. We use $|q\rangle^{\oplus n}$ to denote a quantum register of size n such that each of its constituent qubits is in the state $|q\rangle$. A *phase oracle* U_f for a function $f\colon \{0,1\}^n \to \{0,1\}$ takes as input a quantum register $|x\rangle$ of

size n and leaves its value unchanged, applying to it a negative global phase only if x is a solution for the function, that is $f(x) = 1$. Formally, $U_f |x\rangle = (-1)^{f(x)} |x\rangle$.

2.3 The Quantum Circuit Model

The model of computation we adopt in this paper is that of *quantum circuits*. Quantum circuits can be represented as a sequence of parallel wires (each corresponding to a qubit) passing through certain *gates* that perform quantum elementary operations on them. Indeed, a quantum circuit needs to be reversible and therefore, for each gate the number of input wires equals that of output wires. Furthermore, because of the No-Cloning Theorem [29], it is not possible to either copy or split (fan-out) the information carried by any wire. To use the information on a quantum state multiple times it is necessary to include some *ancillæ*[3].

There are two major measures of the computational time complexity in general circuital models of computation. One of them is the total number of basic gates, namely the *size*; the other one is the *depth* of the direct acyclic graph that represents the circuit. The latter is usually the measure of election for the complexity of quantum circuits because, in such circuits, it is possible to execute two or more gates in parallel whenever they operate on disjoint sets of qubits. The depth of a quantum circuit coincides with the number of time steps performed before the output, that is, with its computational time. The space complexity of a quantum circuit is the number of qubits that it involves, which can also be pictured as the number of wires.

There is a great variety of elementary quantum operators. We briefly list some of them, those that are used in this paper. To define such basic operations, we use the linearity of quantum maps. These elementary gates can be implemented in constant time by real quantum machines and, by definition, have depth $\Theta(1)$ in the quantum circuit model.

- The *Pauli*-X or *NOT* (X) gate is the quantum equivalent of the Boolean NOT gate. It operates on a single qubit, mapping $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$.
- The *Hadamard* gate (H) acts on a single qubit, mapping $|0\rangle$ and $|1\rangle$ to $|+\rangle$ and to $|-\rangle$.
- The *Pauli*-Z or *phase-flip gate* (Z) maps $|0\rangle$ to itself and $|1\rangle$ to $-|1\rangle$, by applying a negative phase to it. Based on the equivalence $Z = HXH$, the Z operator can be obtained from the previous two operators.
- The *controlled* NOT *gate* (CNOT) operates on a register of two qubits $|q_0, q_1\rangle$. If the control qubit $|q_0\rangle$ is set to $|1\rangle$, an X operator is applied on the qubit $|q_1\rangle$, otherwise both qubits stay unchanged. Formally, it maps $|q_0, q_1\rangle$ to $|q_0, q_0 \oplus q_1\rangle$, where $\oplus$ is the Boolean exclusive or, defined also as the sum $\mod 2$.
- The *Swap* gate (SWAP) is a two-qubit operator: it swaps the state of the two qubits $|q_0, q_1\rangle$ involved in the operation, mapping them to $|q_1, q_0\rangle$. Interestingly, the swap gate can be achieved by the application of three CNOT operators.
- The *Toffoli* gate (CCNOT) operates on 3 qubits: if the first two qubits are both equal to $|1\rangle$, an X operator is applied on the third qubit, otherwise all qubits are unchanged. Formally, $|q_0, q_1, q_2\rangle$ is mapped to $|q_0, q_1, q_0 q_1 \oplus q_2\rangle$.

[3] Ancilla qubits or ancillaæare additional qubits needed for the computation, usually they are initialized to the classic 0 state, and after the information they store has been used they are usually cleaned-up, i.e. reset to 0.

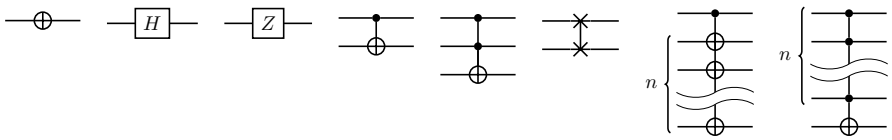

Figure 1. The representation of the following basic gates (from left to right): Pauli-X, Hadamard, Pauli-Z, CNOT, Toffoli, Swap, Fanout and Multicontrolled NOT.

Figure 1 contains a graphical representation of the listed basic gates, together with the graphical representation of the general fanout and the general multicontrolled operators that we are going to define.

Now we introduce three general constructions of non-elementary gates that we need for our algorithm, analyzing their depth.

Given a quantum register $|\psi\rangle = |q_0, q_1, \ldots, q_{n-1}\rangle$ of size n, a *fanout* operator simultaneously copies the control qubit $|q_0\rangle$ onto the $n-1$ target qubits $|q_i\rangle$, for $i \in \{1, \ldots, n-1\}$. Formally, the fanout operator maps $|q_0, q_1, q_2, \ldots, q_{n-1}\rangle$ to the register $|q_0, q_0 \oplus q_1, q_0 \oplus q_2, \ldots, q_0 \oplus q_{n-1}\rangle$. Although a constant time fanout can be obtained by the product of n controlled-not gates, the no-cloning theorem makes it difficult to directly fan qubits out in constant depth [15]. However, assuming that the target qubits are all initialized to $|0\rangle$, it is easy to fan the information in the control qubit out in $\Theta(\log(n))$ depth, by a divide-and-conquer strategy employing controlled-not gates and $\log(n)$ ancilla qubits [10], all initialized to $|0\rangle$.

A *multicontrolled operator* applies an elementary operation on the unique target if *all* the control qubits are set to $|1\rangle$. For example, the n-ary *multicontrolled CNOT* operator, which is a generalization of the Toffoli operator, maps $|q_0, q_1, q_2, \ldots, q_{n-1}\rangle$ to $|q_0, q_1, \ldots, q_{n-2}, (q_0 \cdot q_1 \cdots q_{n-2}) \oplus q_{n-1}\rangle$. Similarly, the n-ary *multicontrolled phase-flip* operator maps $|q_0, q_1, q_2, \ldots, q_{n-1}\rangle$ to $|q_0, q_1, \ldots, q_{n-2}, (-1)^{q_0 \cdot q_1 \cdots q_{n-2}} q_{n-1}\rangle$.

It is possible to obtain a logarithmic-depth quantum circuit performing a multicontrolled operator [2], by exploiting parallelism and using $n/2$ ancilla qubits. We also mention a recent result [24] that enables the implementation of multi-controlled NOT gates in constant time in architectures with trapped ions and neutral atoms.

Assume the need to execute a certain number, say t, of parallel gates controlled by the same qubit $|c\rangle$. Since the control qubit would be involved in all the t operations, the resulting gates would no longer be executable in parallel. Let G be an operator that consists in the parallel execution of the mutually disjoint gates $\{G_0, G_1, \ldots, G_{t-1}\}$, each operating on disjoint sets of qubits from a register of global size n in time $T(n)$. Let $|q_0, q_1, \ldots, q_{n-1}\rangle$ be the quantum register to which one wants to apply the t gates in parallel. The state of $|c\rangle$ is transferred into t ancilla qubits $|a_i\rangle$, all initialized to $|0\rangle$, for $0 \leq i < t$, through a fan-out operator. In this way, t copies of the state $|c\rangle$ are obtained. The ancilla qubit $|a_i\rangle$ is then used as a control qubit for applying the gate G_i. Finally, the ancilla qubits are cleaned up by a new fan-out operator, controlled again by $|c\rangle$. Such a technique applies the t gates in parallel in $T(n)$ time, but requires $\mathcal{O}(\log(t))$ time for the fanout operator.

2.4 Grover's Search Algorithm

Grover's search algorithm [14] is one of the most famous results proving theoretical quantum supremacy, together with Shor's algorithm [25]. Grover's algorithm searches a desired item within an unstructured dataset of n items in $\mathcal{O}(\sqrt{n})$ time, which is quadratically lower than the lower-bound to the runtime of an algorithm solving the same problem in a classical model of computation. The algorithm is inherently bounded-error, but we know there is no quantum algorithm with less than n queries that solves the problem with certainty for an arbitrary dataset [3].

Given a function $f\colon \{0,1\}^{\log(n)} \to \{0,1\}$ with a unique solution $w \in \{0,1\}^{\log(n)}$ such that $f(w) = 1$, the algorithm uses a rotation of the quantum register representing the superposition of all inputs $x \in \{0,1\}^{\log(n)}$ to increase the probability of getting the desired solution w when measuring the register.

More in detail, the algorithm starts with a register encoding the uniform superposition of all possible inputs $x \in \{0,1\}^{\log(n)}$. Note that the objective register, in which the unique solution w has amplitude 1 and all other items have amplitudes 0, is nearly orthogonal to the starting register. Thus the goal of the algorithm is to apply to the initial register a rotation as close as possible to $\pi/2$ to bring it closer to the objective register. A single iterative step of the algorithm consists of two rotations.

The first one consists in a *Phase Oracle gate* U_f for the function f, which flips the amplitude of the qubit $|w\rangle$ corresponding to the searched item, leaving the amplitudes of all other inputs unchanged. The second rotation is performed by the *Grover's Diffusion operator* (or *Diffuser*), which operates a reflection across $|+\rangle^{\oplus n}$. The overall rotation performed during a single Grover iteration is approximately equal to $2/\sqrt{n}$ radiants. Thus, $\pi/4\sqrt{n}$ iterations are necessary and sufficient. Figure 2 represents the circuital translation of Grover's algorithm, in which details on the circuit implementing the Diffuser used by the algorithm are provided.

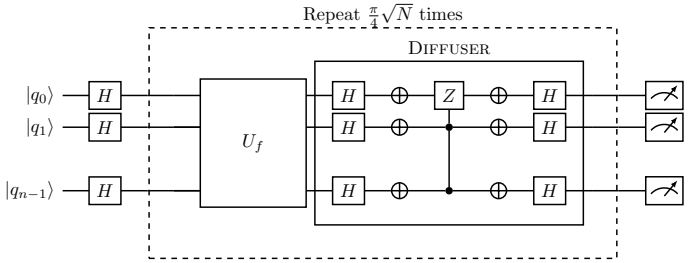

Figure 2. The circuit implementing Grover's algorithm.

The Diffuser requires a multicontrolled Z gate of size n; therefore, its depth is logarithmic in n. Thus, Grover's algorithm has depth $\mathcal{O}\left(\sqrt{n}(T(n)+\log(n))\right)$,[4] where $T(n)$ is the depth of the phase oracle gate.

In the case where the function f has r solutions, with $r > 1$, $\mathcal{O}(\sqrt{n})$ iterations are still enough to find a solution, but it can be shown that roughly $\frac{\pi}{4}\sqrt{\frac{n}{r}}$ iterations are required [5], and that this bound is optimal [4]. However, in general, the value r is not known a priori and can only be obtained through a complex procedure based on the Quantum Phase Estimation. Finally, if we have r solutions and we would like to find all of them, $\Theta(\sqrt{nr})$ iterations are sufficient and necessary [1].

[4] If we assume to be able to implement the multicontrolled Z gate in constant time [24], a Grover's search on a dataset of n items achieves $\mathcal{O}\left(\sqrt{n}T(n)\right)$ time.

CIRCULAR-SHIFT($|q\rangle, k$):
1. for $i \leftarrow 1$ to $\log(n) - \log(k)$ do
2. for $j \leftarrow 0$ to $\frac{n}{2^i k} - 1$ do (in parallel)
3. for $h \leftarrow jk2^i$ to $(j2^i + 1)k - 1$ do (in parallel)
8. Swap($|q_h\rangle, |q_{h-2^{i-1}k}\rangle$)

Figure 3. The pseudocode of the CIRCULAR-SHIFT algorithm for circularly rotating a quantum register $|q\rangle$ of size n by k positions.

3 A controlled circular shift operator

A *circular shift* gate (or *rotation* gate) R_s applies a leftward circular shift of s positions. Formally, the operator R_s applies the following permutation

$$R_s(|q_0, q_1, \ldots, q_{n-1}\rangle) = |R_s(q_0, q_1, \ldots, q_{n-1})\rangle = |q_{n-s}, q_{n-s+1}, \ldots, q_{n-1}, q_0, \ldots, q_{n-s-1}\rangle .$$

A circular shift gate can always be decomposed into (a sequence of) parallel Swap gates. Figure 3 contains the pseudocode of a quantum circuit circularly rotating a quantum register $|q\rangle$ of size n by s positions, for the case that s is a power of 2, proposed by [22,21][5]. Such a circuit has depth $\mathcal{O}(\log(n))$ and at each time-layer performs at most $\frac{n}{2}$ parallel swaps.

Next, we want to define a *controlled cyclic shift gate* that rotates a quantum register by a number of positions depending on an input value k. In this context, we suppose to operate on a circuit containing two quantum registers: the first register $|j\rangle$, of size $\lceil \log(n) \rceil$, encoding the input value related to the shift amount, and the second register $|q\rangle$, of size n, representing the register to be rotated. We denote such a controlled circuit by ROT, and its action on a register $|q\rangle$ of size n controlled by a register $|j\rangle$ (of size $\log(n)$) is formalized by

$$\text{ROT}(|j\rangle \otimes |q_0, q_1, \cdots, q_{n-1}\rangle) = |j\rangle \otimes |q_{n-j}, q_{n-j+1} \cdots, q_{n-j-1}\rangle = |j\rangle \otimes |R_j(q)\rangle .$$

Because the controlled cyclic shift gate does not change the control register, sometimes we abuse the notation writing ROT_j for $\text{ROT}(|j\rangle, \cdot)$.

Since a register of size n can be rotated by a number of positions between 0 and $n-1$, the $|k\rangle$ register can be implemented using $\log(n)$ qubits. We have then $|j\rangle = \bigotimes_{i=0}^{\log(n)-1} |k_i\rangle$, where $|j_i\rangle$ is initialized with the value of the i-th least significant bit of the binary representation of the value k. It will then be enough, for any value of i, such that $0 \leq i < \log(n)$, to apply to register $|q\rangle$ the rotation operator ROT_{2^i} controlled by the qubit $|j_i\rangle$. Note that the rotation operator ROT_{2^i} consists of a sequence of at most $\log(n)$ time-layers, each containing at most, $\frac{n}{2}$ parallel gates, controlled by the same qubit. Thus, to keep the parallelism of the cyclic shift operator in its controlled version, we apply the technique described in the last paragraph of Section 2.3 that involves the use of $\log(n)\frac{n}{2}$ ancilla qubits for the application of all parallel operators controlled by the same qubit, that is, for each $0 \leq i \leq \log(n)$, the cyclic shift controlled by 2^i positions we need $\frac{n}{2}$ ancilla qubits. The controlled cyclic shift by 2^i positions requires $\mathcal{O}(\log(n))$ time-steps to fan the information on the control qubit out to the ancillæ plus $\mathcal{O}(\log(n))$ to perform the cyclic rotation. Therefore, the overall controlled circular shift operator ROT_j over a register of size

[5] Observe that the algorithm in [22] operates a rightward cyclic shift, while we work with leftward cyclic shifts. However, translating a circuit for the rightward cyclic shift to a leftward one by the same number of positions, and vice versa, is a very easy task

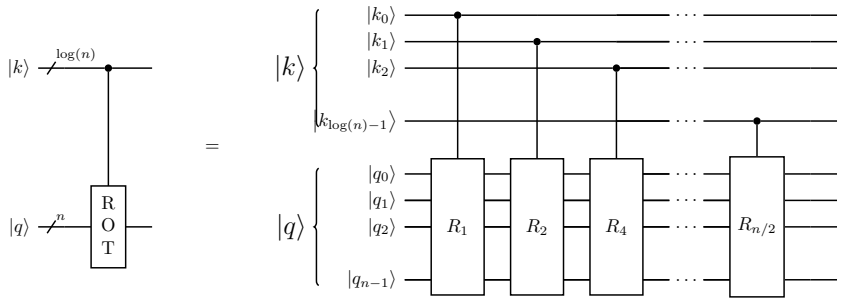

Figure 4. A circular shift operator on a n-qubit register $|q\rangle$ controlled by a $\log(n)$-qubit register $|k\rangle$. The circuit makes use of $\log(n)$ ancilla qubits which is not shown in this graphical representation. On the left of the figure, we show the succinct graphical representation used below in this paper.

n and dependent on the value of an input quantum register of size $\log(n)$ can be implemented by a quantum circuit with depth equal to $\mathcal{O}(\log^2(n))$. Figure 4 contains an illustration of such a circuit.

4 The Algorithm

Throughout the entire Section 4, let a pattern x, of length m, and a text y, of length n.

The problem of finding an occurrence of a cyclic rotation of the pattern x in y can be restated as the problem of finding an element in the subset $Cyc(x, y) \subseteq \{1, \ldots, m-1\} \times \{1, \ldots, n-1\}$ such that

$$Cyc(x, y) = \{(i, j) : 0 \leq i < m,\, 0 \leq j < n \text{ , and } y[j..j+m-1] = R_i(x)\}\,. \tag{1}$$

We point out that in such a definition of the computational problem, the objective is to find not only the position of the text at which a cyclic rotation of the pattern occurs but also the amplitude of the rotation. However, there is a slightly alternative version of the problem in which the wanted output of a solving algorithm is only the position of the text at which a cyclic rotation of the text occurs. The latter version of the cyclic string-matching problem can be restated as the problem of finding an element in the subset $Cyc'(x, y) \subseteq \{1, \ldots, n-1\}$ such that

$$Cyc'(x, y) = \{j \colon 0 \leq j < n \text{ and } \exists 0 \leq i < m \text{ such that } y[j..j+m-1] = R_i(x)\}\,. \tag{2}$$

Our algorithm solves the problem of finding an element in the set defined in Equation 1 and, as we show, it can be easily adapted to solving the problem of finding an element in the set defined in Equation 2.

We denote by $\chi_{Cyc(x,y)}$ the *characteristic function* of $Cyc(x, y)$, that is, the function $\chi_{Cyc(x,y)} \colon \{1, \ldots, m-1\} \times \{1, \ldots, n-1\} \to \{0, 1\}$ defined by

$$\chi_{Cyc(x,y)}(i, j) = \begin{cases} 1 & \text{whenever } (i, j) \in Cyc(x, y), \\ 0 & \text{otherwise.} \end{cases}$$

Analogously, the characteristic function of $Cyc'(x, y)$ is $\chi_{Cyc'(x,y)} \colon \{1, \ldots, n-1\} \to \{0, 1\}$ defined by

$$\chi'_{Cyc(x,y)}(j) = \begin{cases} 1 & \text{whenever } j \in Cyc'(x, y), \\ 0 & \text{otherwise.} \end{cases}$$

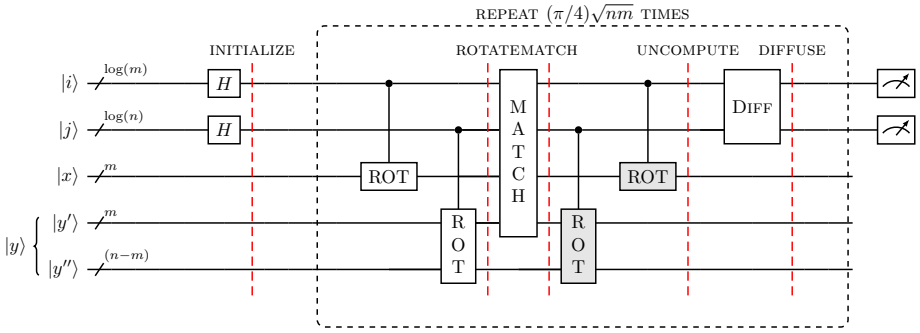

Figure 5. A simplified diagram describing the quantum circuit solving Cyclic String Matching.

The algorithm is an application of Grover's search algorithm (ref. to Section 2.4) for a solution to the function $\chi_{Cyc(x,y)}$, in the form of a quantum circuit. It consists of two steps that are iterated $\lfloor(\pi/4)\sqrt{nm}\rfloor$ times: the phase oracle for the function $\chi_{Cyc(x,y)}$; and the application of Grover's diffuser. The phase oracle of the function $\chi_{Cyc(x,y)}$ is implemented by the rotation and the match steps, and their respective uncomputing steps. In Figure 5, we give a simplified diagram illustrating the circuit, neglecting the ancilla registers needed in the computation.

In the following sections, we describe the computation giving the details of the main steps, alongside their correctness, and analyze the overall time complexity. Throughout the next sections, we will refer to the representation of the circuit shown in Figure 5.

4.1 The Initialization Step

Let us describe the quantum registers involved in the computation and their initialization.

- The register $|i\rangle$ has size $\log(m)$ and it retains the information on the amplitude of the circular rotation of the pattern. More precisely, the kth qubit of the register $|i\rangle$ is either 1, 0, or a coherent superposition of these, depending on the kth digit of the binary representation of the amplitude of the rotation, which is a value between 0 and $m-1$. Starting from $|0\rangle^{\otimes \log(m)}$, the register $|i\rangle$ is initialized to $|+\rangle^{\otimes \log(m)}$ through the application of $\log(m)$ Hadamard's gates so that it contains the superposition of all possible values in $\{0,...,m-1\}$. Formally, $|i\rangle = \frac{1}{\sqrt{m}} \sum_{s\in\{0,1\}^{\log(m)}} |s\rangle$.
- The register $|j\rangle$ has size $\log(n)$ and retains the information on the position $j \in \{0,\ldots,n-1\}$ of the substring to be examined within the text, that is, specifically, the (binary representation of the) value of the position j of the text where the first character of such a substring is found. Starting from $|0\rangle^{\otimes \log(n)}$, the register $|j\rangle$ is initialized to $|+\rangle^{\otimes \log(n)}$ through the application of $\log(n)$ Hadamard's gates so to contain the superposition of all possible values in $\{0,...,n-1\}$. Formally, $|j\rangle = \frac{1}{\sqrt{n}} \sum_{t\in\{0,1\}^{\log(n)}} |t\rangle$.
- The register $|x\rangle$ has size m and and its constituent qubits are initialized so that $|x_h\rangle$ contains the h-th bit of the pattern, for $0 \leq h < m$.
- Similarly, the register $|y\rangle$ has size n and its constituent qubits are initialized so that $|y_k\rangle$ contains the k-th bit of the text, for $0 \leq k < n$. For visualization purposes

the $|y\rangle$ register is divided into two registers, $|y'\rangle$ and $|y''\rangle$, of size m and $n-m$, respectively.
– Finally, we have three ancilla registers[6], $|a\rangle$, $|b\rangle$, and $|c\rangle$, that – for simplicity – are not depicted in Figure 5. The register $|a\rangle$ is an ancilla register that is needed to use the same control register for different operators (see Section 2.3). In particular, $|a\rangle$ has length $\log(m)+\log(n)$; the first $\log m$ qubits of $|a\rangle$ are needed for the implementation of the controlled cyclic shift operator ROT (see Section 3) to be applied to the pattern x, while the last $\log(n)$ qubits of $|a\rangle$ are needed for the implementation of the text y. All the qubits from $|a\rangle$ are initialized to $|0\rangle$. The register $|b\rangle$ consists of $\lceil \frac{m}{2} \rceil + 1$ qubits and is needed to implement the multicontrolled phase-flip gate needed in the implementation of the phase oracle for $\chi(x,y)$, and precisely within the matching step (see Section 4.4) and is initialized to $|0\rangle^{\otimes \lceil \frac{m}{2} \rceil + 1}$. Similarly, the register $|c\rangle$ consists of $\lceil \frac{\log(m)+\log(n)}{2} \rceil + 1$ qubits and is needed to implement the multicontrolled phase-flip gate needed in the implementation of the Grover's Diffuser (see Section 2.4) and is initialized to $|0\rangle^{\otimes \lceil \frac{m}{2} \rceil + 1}$.

Overall, the circuit needs $\mathcal{O}(\log(mn)+m+n)$ qubits, i.e., it has $\mathcal{O}(n)$-space complexity.

Observe that each of the operators applied during this step takes constant time, and since they can be applied in parallel, the initialization phase takes $\mathcal{O}(1)$ time.

We can define the *global* register

$$|q\rangle := |i\rangle \otimes |j\rangle \otimes |x\rangle \otimes |y\rangle \,,$$

whose constituent qubits are all those needed in the circuit (except for the ancillæ). Referring to $|q\rangle$, we can algebraically prove the correctness of the quantum algorithm described by the circuit in Figure 5. After the initialization we have

$$|\psi\rangle := \textsc{Initialize}\,|q\rangle = \frac{1}{\sqrt{m}} \sum_{s\in\{0,1\}^{\log(m)}} |s\rangle \otimes \frac{1}{\sqrt{n}} \sum_{t\in\{0,1\}^{\log(n)}} |t\rangle \otimes |x\rangle \otimes |y\rangle \,.$$

4.2 The Grover's Iterations

Since we are searching for solutions to $\chi_{cyc(x,y)}$ in a space of dimension $m\cdot n$, and since we do not know the number of solutions, we repeat the rotation step, the matching step, and the Grover's Diffuser $\lfloor \frac{\pi}{4}\sqrt{mn} \rfloor$ times. The phase oracle for the function $\chi_{cyc(x,y)}$ consists of the rotation step, the matching step, and the uncomputing step. After the $\lfloor \frac{\pi}{4}\sqrt{mn} \rfloor$ Grover's iterations we measure the first $m+n$ qubits of the register $|q\rangle$.

Already in Section 2.4, we presented the circuit implementing the Grover's Diffuser and discussed how a suitable number of Grover's iterations (i.e. phase oracle followed by Diffuser) maximizes - and makes very close to 1 - the probability of measuring a solution to the target function, which in our case is $\chi_{cyc(x,y)}$. Therefore, in order to prove the correctness of our procedure it is enough to prove that indeed the rotation step, the matching step, and the uncomputing step implement the oracle function for $\chi_{cyc(x,y)}$, that is the combined effect of a single application of each of these step transforms $|\psi\rangle$ by changing nothing but the phase of the state $|s\rangle\,|t\rangle$ corresponding to a pair (s,t) that is a solution $\chi_{cyc(x,y)}$, that is a pair (s,t) such that the cyclic rotation of x by s positions matches the substring $[y_j,\ldots,y_{j+m-1}]$ of the text.

[6] Ancilla registers are quantum registers made of ancilla qubits.

4.3 The Rotation Step

The Rotation Step consists of two controlled cyclic shift gates, see Section 3: the first of which performs a leftward cyclic rotation of the register $|x\rangle$ controlled by $|i\rangle$, while the second one performs a leftward cyclic rotation of the register $|y\rangle$ controlled by $|j\rangle$. Since the two sets containing the qubits manipulated by $\textsc{rot}(|i\rangle, |x\rangle)$ and $\textsc{rot}(|j\rangle, |y\rangle)$, respectively, are disjoint, we can execute the two gates in parallel. The depth of the circuit performing ROT on $|i\rangle \otimes |x\rangle$ is $\mathcal{O}(\log^2(m))$, and the depth of the circuit performing ROT on $|j\rangle \otimes |y\rangle$ is $\mathcal{O}(\log^2(n))$; therefore, overall, the rotation steps has time complexity $\mathcal{O}(\max(\log^2(m), \log^2(n))) = \mathcal{O}(\log^2(n))$. The joint action of $\textsc{rot}(|i\rangle, |x\rangle)$ and $\textsc{rot}(|j\rangle, |y\rangle)$ on the register $|\psi\rangle$ is described by

$$|\psi'\rangle = \textsc{Rotate}\,|\psi\rangle = \frac{1}{\sqrt{m}} \sum_{s\in\{0,1\}}^{\log(m)} |s\rangle \otimes \frac{1}{\sqrt{n}} \sum_{t\in\{0,1\}}^{\log(n)} |t\rangle \otimes \frac{1}{\sqrt{m}} \sum_{i=0}^{m} |R_s(x)\rangle \otimes \frac{1}{\sqrt{n}} \sum_{i=0}^{n} |R_t(y)\rangle\,.$$

4.4 The Matching Step

The Matching Step consists of a single gate (denoted by MATCH in Figure 5) that makes a *qubitwise* comparison between the two registers $|x\rangle$ and $|y'\rangle$, which at the moment before the application of MATCH are actually equal to $\frac{1}{\sqrt{m}} \sum_{i=0}^{m} |R_s(x)\rangle$ and $\frac{1}{\sqrt{n}} \sum_{i=0}^{n} |R_t(y)\rangle$; in the case that $\frac{1}{\sqrt{m}} \sum_{i=0}^{m} |R_s(x)\rangle$ and $\frac{1}{\sqrt{n}} \sum_{i=0}^{n} |R_t(y)\rangle$ are qubitwise equal, MATCH flip the phase of the first qubit of $|i\rangle$, which results in a phase-flip of the register $|i\rangle \otimes |j\rangle$. The quantum circuit implementing the MATCH gate is illustrated in Figure 6.

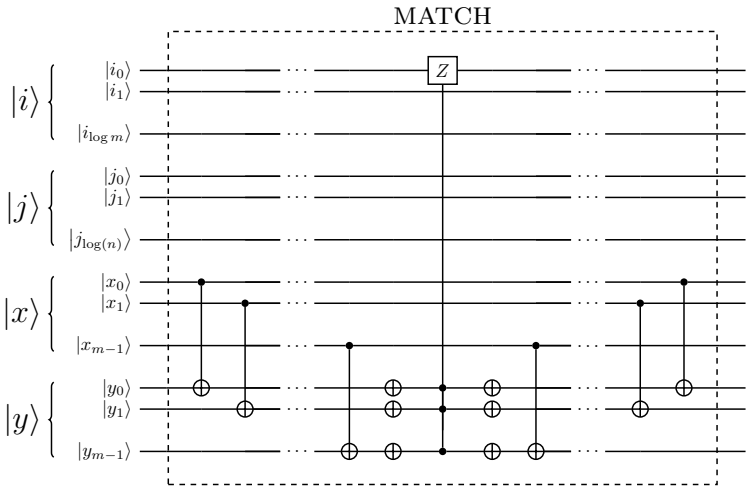

Figure 6. A circuit detecting an exact match between two strings of length m.

The comparisons between qubits can be performed in parallel. More precisely, for $0 \leq \ell < m$, the states of the qubits $|i_\ell\rangle$ and $|j_\ell\rangle$ are compared in $\mathcal{O}(\log(m))$-time by means of m parallel CNOT gates and a multiple controlled phase-flip gate with m controls. Specifically, the parallel application of the m parallel CNOT takes $\mathcal{O}(1)$ time, while the multiple controlled phase-flip gate needs $\mathcal{O}(\log(m))$ time (and $\frac{m}{2}$ ancillæ). Thus, the overall depth of the MATCH gate is $\mathcal{O}(\log(m))$. After the match phase we have

$$
\begin{aligned}
|\psi''\rangle =& \textsc{Matching}\,|\psi'\rangle \\
=& \frac{1}{\sqrt{mn}} \sum_{s\in\{0,1\}}^{\log(m)} \sum_{t\in\{0,1\}}^{\log(n)} (-1)^{\chi_{Cyc(x,y)}(s,t)} |s\rangle \otimes |t\rangle \otimes \frac{1}{\sqrt{m}} \sum_{i=0}^{m} |R_s(x)\rangle \otimes \frac{1}{\sqrt{n}} \sum_{i=0}^{n} |R_t(y)\rangle\,.
\end{aligned}
$$

4.5 The Uncomputing Step

The uncomputing step rotate the register $|x\rangle$ and $|y\rangle$ back to their initial configuration and it consists of a *rightward* cyclic rotation of $|x\rangle$ controlled by $|i\rangle$ and *rightward* cyclic rotation of $|y\rangle$ controlled by $|j\rangle$. Clearly, this step has time-complexity equal to that of the rotating step (see Section 4.3), that is $\mathcal{O}(\log^2(n))$. After the uncomputing step we have

$$
\begin{aligned}
|\psi'''\rangle =& \textsc{Uncomputing}\,|\psi''\rangle \\
=& \frac{1}{\sqrt{mn}} \sum_{s\in\{0,1\}}^{\log(m)} \sum_{t\in\{0,1\}}^{\log(n)} (-1)^{\chi_{Cyc(x,y)}(s,t)} |s\rangle \otimes |t\rangle \otimes \frac{1}{\sqrt{m}} \sum_{i=0}^{m} |R_{-s}R_s(x)\rangle \otimes \frac{1}{\sqrt{n}} \sum_{i=0}^{n} |R_{-t}R_t(y)\rangle \\
=& \frac{1}{\sqrt{mn}} \sum_{s\in\{0,1\}}^{\log(m)} \sum_{t\in\{0,1\}}^{\log(n)} (-1)^{\chi_{Cyc(x,y)}(s,t)} |s\rangle \otimes |t\rangle \otimes \frac{1}{\sqrt{m}} \sum_{i=0}^{m} |x\rangle \otimes \frac{1}{\sqrt{n}} \sum_{i=0}^{n} |y\rangle\,.
\end{aligned}
$$

Therefore, after the uncomputing step our register is unchanged except for the flipped phase of the states $|s\rangle\,|t\rangle$ corresponding to a pair (s,t) that is a solution $\chi_{Cyc(x,y)}$, that is we have proved that the rotating, the matching, and the uncomputing step implements the phase oracle for $\chi_{Cyc(x,y)}$, which is what we wanted to prove for correctness (see 4.2).

4.6 Time Complexity

The overall circuit implementing the phase oracle for $\chi_{Cyc(x,y)}$ has depth $\mathcal{O}(\log^2(n) + \log(m) = \mathcal{O}(\log^2(n))$. Since Grover's Diffuser has depth $\mathcal{O}(\log(m))$, and because we need $\lfloor \frac{n}{\pi}\sqrt{mn} \rfloor$ Grover's iterations, the circuit for cyclic string matching presented has depth equal to $\mathcal{O}\left(\sqrt{mn}(\log^2(n) + \log(m))\right) = \mathcal{O}\left(\sqrt{nm}(\log^2(n))\right)$. Using the big-$\tilde{\mathcal{O}}$-notation, which works as the big-$\mathcal{O}$-notation except for hiding polylogarithmic factors, we can write that the depth of the presented circuit presented equals $\tilde{\mathcal{O}}\left(\sqrt{nm}(\log^2(n))\right)$. Therefore, we get an almost quadratic speed-up against the most time-efficient algorithm for the problem considered.

In the appendix, we provide a practical implementation of the algorithm using Qiskit, an open-source toolkit developed by IBM and based on the Python language. The code shown in this section is available to browse and run within a public Google Colab Tutorial.[7]

5 Conclusions

In this paper, we have investigated the quantum cyclic string matching problem, presenting a novel approach that leverages quantum computing principles to enhance

[7] `https://colab.research.google.com/drive/-1bbRjsYl7UCVT6P4gNwJulARfd64L1RCT?usp=sharing`

computational efficiency. We began by discussing the foundational concepts of string matching, both exact and approximate, and highlighted the significance of cyclic strings in various applications, from bioinformatics to data compression and pattern recognition.

Our primary contribution is a quantum algorithm for cyclic string matching, building on the work of Niroula and Nam [21]. This algorithm utilizes the circuit-based quantum model and incorporates cyclic rotations to achieve a superposition of all possible alignments of the pattern within the text. Implemented using Qiskit, our method offers significant computational efficiency, with a complexity of $\tilde{\mathcal{O}}(\sqrt{nm})$, providing a substantial speed-up over classical algorithms.

As quantum hardware and algorithms continue to evolve, the integration of quantum computing into broader scientific and industrial applications appears increasingly promising. Our work demonstrates the potential of quantum computing to redefine problem-solving paradigms in string matching, paving the way for future innovations and applications in this and related fields.

References

1. A. Ambainis: *Quantum search algorithms.* SIGACT News, 35(2) 2004, pp. 22–35.
2. S. Balauca and A. Arusoaie: *Efficient constructions for simulating multi controlled quantum gates*, in Computational Science - ICCS 2022, vol. 13353 of LNCS, Springer, 2022, pp. 179–194.
3. R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf: *Quantum lower bounds by polynomials.* J. ACM, 48(4) 2001, pp. 778–797.
4. M. Boyer, G. Brassard, P. Høyer, and A. Tapp: *Tight bounds on quantum searching.* Fortschritte der Physik, 46(4-5) 1998, pp. 493–505.
5. G. Brassard, P. Høyer, M. Mosca, and A. Tapp: *Quantum amplitude amplification and estimation*, in Quantum Computation and Information, S. G. Lo Monaco and H. E. Brandt, eds., vol. 305 of Contemporary Mathematics, American Mathematical Society, 2002, pp. 53–74.
6. G. Buonaiuto, R. Guarasci, A. Minutolo, G. D. Pietro, and M. Esposito: *Quantum transfer learning for acceptability judgements.* Quantum Mach. Intell., 6(1) 2024, p. 13.
7. E. T. Campbell, A. Khurana, and A. Montanaro: *Applying quantum algorithms to constraint satisfaction problems.* Quantum, 2018.
8. D. Cantone, S. Faro, and A. Pavone: *Quantum string matching unfolded and extended*, in Reversible Computation - Proceedings, vol. 13960 of LNCS, Springer, 2023, pp. 117–133.
9. R. Dulbecco and M. Vogt: *Evidence for a ring structure of polyoma virus dna.* Proceedings of the National Academy of Sciences, 50(2) 1963, pp. 236–243.
10. M. Fang, S. Fenner, F. Green, S. Homer, and Y. Zhang: *Quantum lower bounds for fanout.* Quantum Info. Comput., 6(1) jan 2006, pp. 46–57.
11. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
12. S. Faro, F. P. Marino, and A. Pavone: *Efficient online string matching based on characters distance text sampling.* Algorithmica, 82(11) 2020, pp. 3390–3412.
13. S. Faro, A. Pavone, and C. Viola: *Quantum path parallelism: A circuit-based approach to text searching*, in Theory and Applications of Models of Computation - 18th Annual Conference, TAMC 2024, Hong Kong, China, May 13-15, 2024, Proceedings, X. Chen and B. Li, eds., vol. 14637 of Lecture Notes in Computer Science, Springer, 2024, pp. 247–259.
14. L. K. Grover: *A fast quantum mechanical algorithm for database search*, in Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, STOC '96, ACM, 1996, pp. 212–219.
15. P. Høyer and R. Spalek: *Quantum fan-out is powerful.* Theory C., 1 2005, pp. 81–103.
16. Y. Lao and O. Ait-Aider: *Rolling shutter homography and its applications.* IEEE Trans. Pattern Anal. Mach. Intell., 43(8) 2021, pp. 2780–2793.
17. F. Lisacek: *Algorithms on strings, trees and sequences: Dan gusfield.* Comput. Chem., 24(1) 2000, pp. 135–137.

18. M. Lothaire: *Applied Combinatorics on Words*, Cambridge University Press, 2005.
19. A. Montanaro: *Quantum pattern matching fast on average.* Algorithmica, 77(1) 2017, pp. 16–39.
20. D. R. Musser: *Introspective sorting and selection algorithms.* Softw. Pract. Exp., 27(8) 1997, pp. 983–993.
21. P. Niroula and Y. Nam: *A quantum algorithm for string matching.* npj Quantum Information, 7 02 2021, p. 37.
22. A. Pavone and C. Viola: *The quantum cyclic rotation gate*, in Proceedings of the 24th Italian Conference on Theoretical Computer Science (ITCTS 2023), vol. Vol-3587, Italy, September 13-15 2023, University of Palermo, pp. 206–218.
23. H. Ramesh and V. Vinay: *String matching in $\mathcal{O}(n+m)$ quantum time.* Journal of Discrete Algorithms, 1(1) 2003, pp. 103–110.
24. S. E. Rasmussen, K. Groenland, R. Gerritsma, K. Schoutens, and N. T. Zinner: *Single-step implementation of high-fidelity n-bit Toffoli gates.* Phys. Rev. A, 101 Feb 2020, p. 022308.
25. P. W. Shor: *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer.* SIAM J. Comp., 26(5) 1997, pp. 1484–1509.
26. M. Thanbichler, S. Wang, and L. Shapiro: *The bacterial nucleoid: A highly organized and dynamic structure.* Journal of cellular biochemistry, 96 10 2005, pp. 506–21.
27. C. Viola and S. Živný: *The Combined Basic LP and Affine IP Relaxation for Promise VCSPs on Infinite Domains.* ACM Trans. Algorithms, 17(3) 2021, pp. 21:1–21:23.
28. R. Weil and J. Vinograd: *The cyclic helix and cyclic coil forms of polyoma viral dna.* Proceedings of the National Academy of Sciences, 50(4) 1963, pp. 730–738.
29. W. K. Wootters, W. K. Wootters, and W. H. Zurek: *A single quantum cannot be cloned.* Nature, 299 1982, pp. 802–803.

A A Qiskit Implementation

Initially, we introduce the fundamental structure of the algorithm and the implementation of the involved operators, as outlined in the reference provided. Subsequently, we delve into more specific algorithmic aspects that enable us to tackle the problem under broader conditions. The code discussed in this study is readily accessible online for consultation and execution via a Colab document.

```
def rot(n, s):
  qc = QuantumCircuit(n)
  for i in range(1, log2(n)-log2(s)+2):
    for j in range(0, n/(s*(2**i))):
      for q in range(j*s*2**i,s*(j*2**i+1)):
        qc.swap(n-1-(q+s), n-1-(q+2**(i-1)*s+s))
  return qc.to_gate(label='ROT'+str(s))

def rot_gate(n,s):
  return rot(n,s).to_gate(label='Rot'+str(s))
```

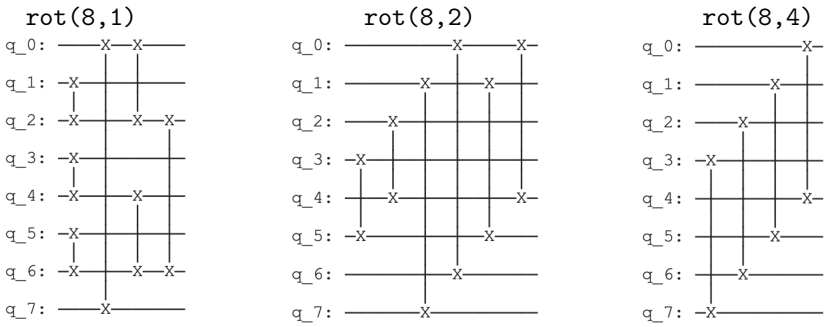

Figure 7. On top: the Qiskit procedure `rot(n,s)` for implementing the left-rotation quantum operator which cyclically rotate a register with size `n` of `s` positions to the right. On bottom: three leftward rotation operators generated by the procedure for a quantum register of 8 qubits and shift amounts equal to 1, 2 and 4 positions, respectively.

```
def parametrized_rot(n):
  logn = int(np.log2(n))
  kr = QuantumRegister(logn,'k')
  yr = QuantumRegister(n,'y')
  qc = QuantumCircuit(kr,yr)
  for i in range(logn):
    qc = qc.compose(rot_gate(n,2**i).control(1), [kr[i]]+yr[:])
  return qc
```

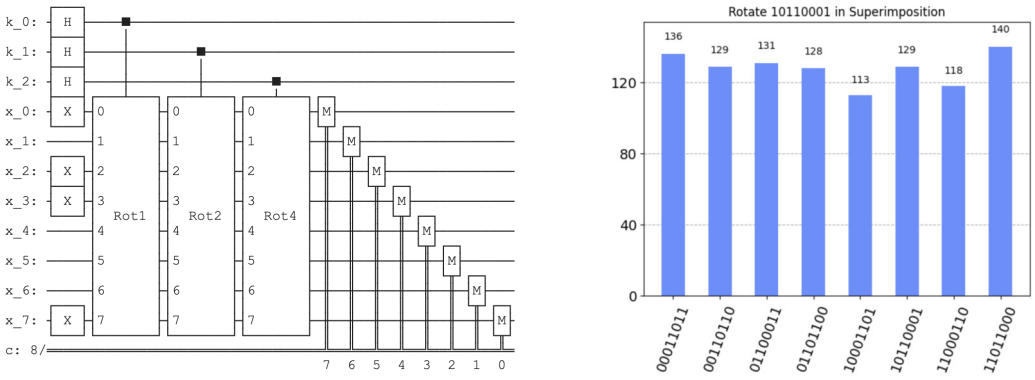

Figure 8. An implementation of the circuit for obtaining a superposition of all the cyclic shifts of an 8-qubit register obtained via the controlled cyclic shift operator.

```
def match(n):
  xr = QuantumRegister(n,'x')
  yr = QuantumRegister(n,'y')
  out = QuantumRegister(1,'out')
  qc = QuantumCircuit(xr,yr,out)
  for i in range(n):
    qc.cx(xr[i],yr[i])
    qc.x(yr[i])
  qc.mcx(yr[:],out)
  for i in range(n):
    qc.x(yr[i])
    qc.cx(xr[i],yr[i])
  return qc

def match_gate(n):
  return match(n).to_gate(label='Match'+str(n))
```

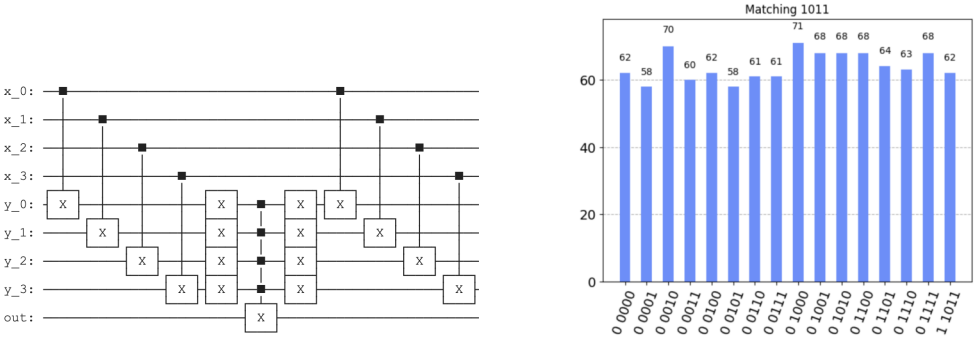

Figure 9. An implementation of the quantum gate checking for a match of two strings, each encoded by a 4-qubit register.

```
def csmo(m,n):
  logm = int(np.log2(m))
  logn = int(np.log2(n))
  ir = QuantumRegister(logm,'i')
  jr = QuantumRegister(logn,'j')
  xr = QuantumRegister(m,'x')
  yr = QuantumRegister(n,'y')
  out = QuantumRegister(1,'out')
  qc = QuantumCircuit(ir,jr,out,xr,yr)
  qc = qc.compose(parametrized_rot(m), ir[:]+xr[:])
  qc = qc.compose(parametrized_rot(n), jr[:]+yr[:])
  qc = qc.compose(match_gate(m), xr[:]+yr[:m]+out[:])
  qc = qc.compose(parametrized_rot(n).inverse(), jr[:]+yr[:])
  qc = qc.compose(parametrized_rot(m).inverse(), ir[:]+xr[:])
  return qc
```

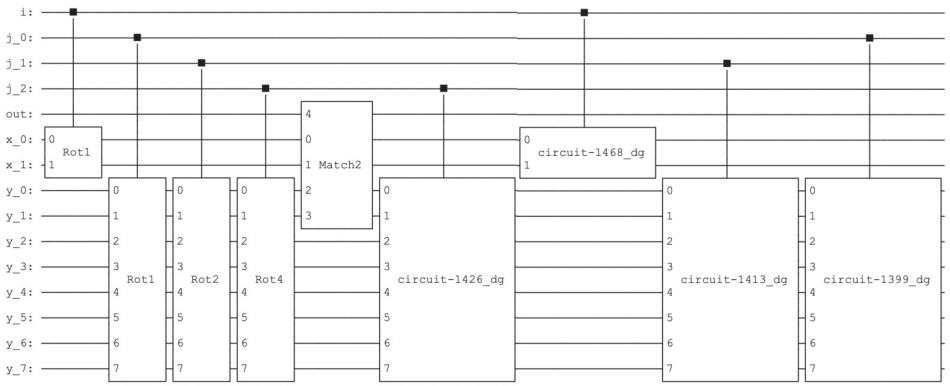

Figure 10. Circuital implementation of Grover's search algorithm for identifying a cyclic occurrence of a pattern `x` of length 2 within a text `y` of length 8.

```
def GroverCSM(x,y,t):
  m = len(x)
  n = len(y)
  logm = int(np.log2(m))
  logn = int(np.log2(n))
  ir = QuantumRegister(logm,'i')
  jr = QuantumRegister(logn,'j')
  xr = QuantumRegister(m,'x')
  yr = QuantumRegister(n,'y')
  out = QuantumRegister(1,'out')
  icr = ClassicalRegister(logm,'ic')
  jcr = ClassicalRegister(logn,'jc')
  qc = QuantumCircuit(ir,jr,out,xr,yr,icr,jcr)
  qc = qc.compose(register_initialize(x), xr[:])
  qc = qc.compose(register_initialize(y), yr[:])
  qc.h(ir)
  qc.h(jr)
  qc.x(out)
  qc.h(out)
  iterations = int((np.pi/4)*(np.sqrt(n*m/t)))
  for i in range(iterations):
    qc = qc.compose(csmo_gate(m,n), ir[:]+jr[:]+out[:]+xr[:]+yr[:])
    qc = qc.compose(diff(logm+logn), ir[:]+jr[:])
  for i in range(logm):
    qc.measure(ir[i], icr[i])
  for i in range(logn):
    qc.measure(jr[i], jcr[i])
  return qc
```

Figure 11. Qiskit implementation of Grover's search algorithm for identifying a cyclic occurrence of a pattern `x` of length `m` within a text `y` of length `n`. The implementation assumes that both strings are binary sequences and that the number `t` of occurrences of `x` in `y` is known.

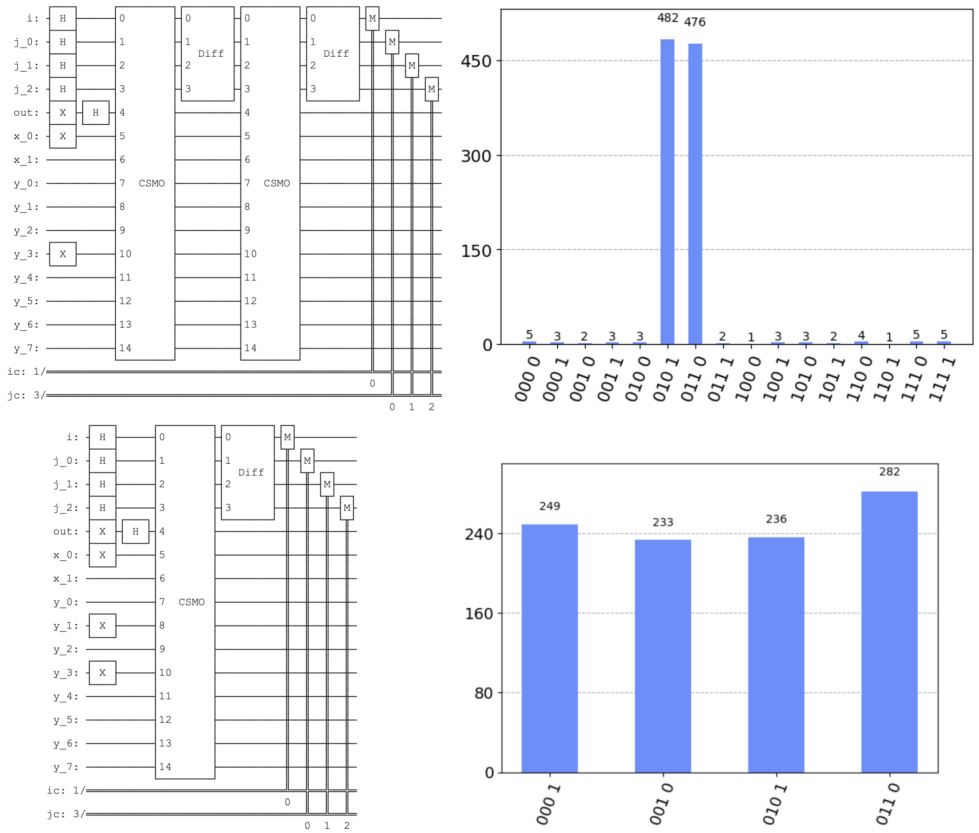

Figure 12. Two runs of the quantum Cyclic String Matching algorithm and the graph of the related results. Above: searching for the string `10` within the text `00010000`. There are 2 occurrences, in position 2 with rotation equal to 1, and in position 3 with rotation equal to 0. Two executions of Grover's algorithm are necessary to identify one of the two occurrences. Bottom: Searching for the string `10` within the text `01010000`. There are 4 occurrences, in positions 0,1,2 and 3 with rotation equal to 1, 0, 1 and 0, respectively. In this case, a single iteration of Grover's algorithm is sufficient to identify one of the 4 solutions.

A Language-Theoretic Approach to the Heapability of Signed Permutations

Gabriel Istrate

Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, 011014, Sector 1, Bucharest, Romania
gabriel.istrate@unibuc.ro

Abstract. We investigate a signed version of the Hammersley process, a discrete process on words related to a property of integer sequences called heapability (Byers et al., ANALCO 2011). The specific version that we investigate corresponds to a version of this property for signed sequences.
We give a characterization of the words that can appear as images of the signed Hammersley process. In particular we show that the language of such words is the intersection of two deterministic one-counter languages.

Keywords: signed Hammersley process, formal languages, heapability

1 Introduction

Consider the following process H_k that generates strings over the alphabet $\Sigma_k = \{0, 1, \ldots, k\}$, $k \geq 1$: start with the empty word $w_0 = \lambda$. Given word w_n, to obtain w_{n+1} insert a k at an arbitrary position of w_n (we will regard the new k as a "particle with k lives"). If w_n contained at least one nonzero letter to the right of the newly inserted k then decrease by 1 the leftmost such letter (that is the new particle takes one life from the leftmost live particle to its right). The process has been introduced in [14], related to a special property of integer sequences called *heapability* (see [9] for a definition, and [14,16,15,4,5,3,7] for subsequent work), and has been called (by analogy to the classical case $k = 1$, which it generalized) *the Hammersley process*. This process has proved essential [1] in the scaling analysis of the longest-increasing subsequence of a random permutation, one of the celebrated problems in theoretical probability [17], and was subsequently rigorously analyzed in [4,5].

In [7] we have undertaken a language-theoretic approach to the study of process H_k by characterizing the language $L(H_k)$ of possible words in the process H_k. It was shown that the language $L(H_k)$ is regular for $k = 1$ and context-free but nonregular for $k \geq 2$. We also gave an algorithm to compute the formal power series associated to the language $L(H_k)$ (where the coefficient of each word w is equal to its multiplicity in the process H_k). We attempted to use this algorithm to study the typical large-scale behavior of process H_k (while this study provided valuable information, the convergence to the limit behavior turned out to be fairly slow).

The purpose of this paper is to study (with language-theoretic tools similar to those in [7]) a variant of the Hammersley process that we will call *the signed Hammersley process*, and we will denote by H_k^{sign}. This is a process over the alphabet $\Gamma_k := \{0^+, 0^-, 1^+, 1^-, 2^+, 2^-, \ldots, k^+, k^-\}$ and differs from process H_k in the following manner:

– The newly inserted letter will always be a *signed* version of k, that is, it will either be a k^+ or a k^-.

Gabriel Istrate: A Language-Theoretic Approach to the Heapability of Signed Permutations, pp. 71–85.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

– When a k^- is inserted it subtracts 1 from the closest (if any) nonzero letter to its right having positive polarity. That is, inserting k^- turns a k^+ into a $(k-1)^+$, a $(k-1)^+$ into a $(k-2)^+$, ..., a 1^+ into a 0^+, but has no effect on letters of type 0^+ or letters with negative polarity.
– On the other hand, inserting a k^+ subtracts a 1 from the closest digit with negative polarity and nonzero value (i.e., one in $\{1^-, \ldots, k^-\}$), if any, to its right.

Example 1. A depiction of the first few possible steps in the evolution of the ordinary Hammersley tree process is presented in Figure 1(a). Similarly for the signed Hammersley tree process, see Figure 1(b).

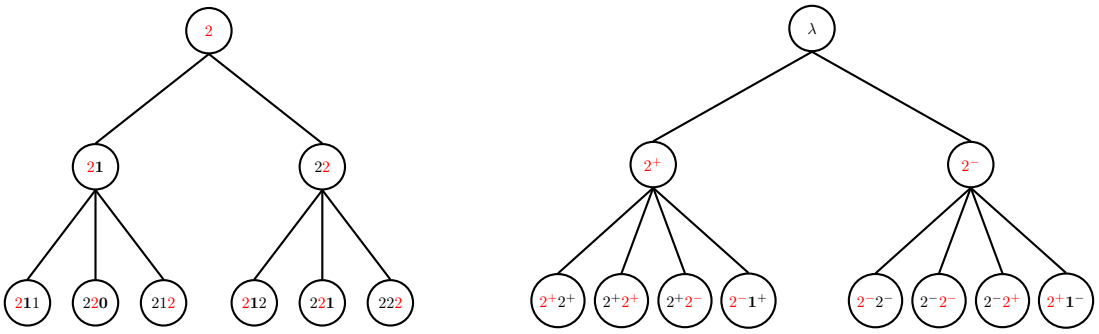

Figure 1. (a). Words in the binary Hammersley process ($k = 2$). The first node λ is omitted. (b). Words in the binary signed Hammersley tree process. Both figures: Insertions are in red. Positions that lost a life at the current stage are bolded.

In this paper we provide (Theorem 7) a complete characterization of the language of words in the signed Hammersley process, they are the words such that all of their prefixes satisfy a certain property called *k-dominance* (see Definition 2 for the precise definition of k-dominant strings). As a consequence, the language of words in the signed Hammersley process is the intersection of two deterministic context-free languages.

Perhaps not surprisingly, the motivation behind our study is a adaptation of the concept of heapability to signed sequences (permutations) $(\sigma(1), \sigma(2), \ldots, \sigma(n))$. Briefly, in the setting we consider, every integer $\sigma(i)$ in the sequence comes with a sign $\tau(i) \in \{\pm 1\}$. An integer $\sigma(i)$ can only become the child of an integer $\sigma(j)$ in a heap-ordered tree when $\sigma(j) < \sigma(i)$ and $\sigma(i)$ has the opposite sign (i.e., $\tau(i) = -\tau(j)$). When the signed permutation cannot be inserted into a single heap-ordered tree we are, instead, concerned with inserting it into the smallest possible number of heap-ordered trees.

The outline of the paper is as follows: In Section 2 we review some preliminary notions we will need in the sequel. Our main result is presented in Section 3. We then give (Section 4) an algorithm for computing the formal power series associated to the signed Hammersley process. Later sections are no longer primarily string-theoretic, and instead attempt to explain the problem on signed integer sequences that motivated our results: in Section 5 we define the heapability of signed permutations and show that a greedy algorithm partitions a signed permutation into a minimal number of heap-ordered trees. This motivates and explains the definition of the signed Hammersley process, that describes the dynamics of "slots" generated by this algorithm.

We refrain in this paper, though, from investigating the scaling in the Ulam-Hammersley problem for signed permutations, as it is not directly connected to the main theme of the conference, and instead leave this topic for future work.

2 Preliminaries

We will assume general acquaintance with formal language theory, as presented in, say, [13], and its extension to (noncommutative) formal power series. For this last topic useful (but by no means complete) references are [18,6]. Denote by Γ_k the alphabet $\{0^+, 0^-, \ldots, k^+, k^-\}$. As usual, for $z \in \Gamma_k^*$ and $p \in \Gamma_k$, denote by $|z|$ the length of z and by $|z|_p$ the number of appearances of letter p in z. Also, for $j \in \{0, \ldots k\}$, define $|z|_j = |z|_{j^+} + |z|_{j^-}$.

Definition 2. *String $z \in \Gamma_k^*$ is called k-dominant iff it starts with a letter from the set $\{k^+, k^-\}$, and the following two conditions are satisfied:*

$$|z|_{k^+} - \sum_{i=1}^{k} i \cdot |z|_{(k-i)^-} + \sum_{i=0}^{k-1} |z|_{i^+} \geq 0 \tag{1}$$

and

$$|z|_{k^-} - \sum_{i=1}^{k} i \cdot |z|_{(k-i)^+} + \sum_{i=0}^{k-1} |z|_{i^-} \geq 0 \tag{2}$$

at least one of the inequalities being strict, namely the one that corresponds to the first letter of z.

Example 3. The only words $z \in \Gamma_k^1$ that are k-dominant are $z = k^+$ and $z = k^-$.

Example 4. For $z \in \{0^+, 0^-, k^+, k^-\}^*$ (or $k = 1$) the two conditions in Definition 2 become $|z|_{k^+} - k \cdot |z|_{0^-} + |z|_{0^+} \geq 0$ and $|z|_{k^-} - k \cdot |z|_{0^+} + |z|_{0^-} \geq 0$.

An example of a k-dominant word is (for $k = 2$) word $z = 2^+2^-$. Indeed, $|z|_{2^+} = 1, |z|_{2^-} = 1, |z|_a = 0$ for $a \notin \{2^-, 2^+\}$. So z satisfies conditions (1) and (2). So does its prefix 2^+.

Definition 5. *A formal power series with nonnegative integer coefficients is a function $f : \Gamma_k^* \to \mathbf{N}$. We will denote by $\mathbf{N}(< \Gamma_k >)$ the set of such formal power series.*

Definition 6. *Given $k \geq 1$,* the signed Hammersley power series of order k *is the formal power series $F_k \in \mathbf{N}(< \Gamma_k >)$ that counts the multiplicity of a given word $w \in \Gamma_k^*$ in the signed Hammersley process of order k.*

The signed Hammersley language of order k, *$L(H_k^{sign})$, is defined to be the support of F_k, i.e., the set of words $w \in \Gamma_k^*$ that are outputs of the signed Hammesley process of order k.*

3 Main Result: The language of the process H_k^{sign}

Our main result is:

Theorem 7. *A word $z \in \Gamma_k^*$ is generated by the signed Hammersley process if and only if z and all its nonempty prefixes are k-dominant.*

Corollary 8. *For every $k \geq 1$ if $L(H_k^{sign})$ is the language of words that appear in the signed Hammersley process there exist two deterministic context-free languages (in fact L_1, L_2 are even deterministic one-counter languages, see [20]) s.t. $L(H_k^{sign}) = L_1 \cap L_2$.*

The proof of Theorem 7 proceeds by first showing that every word generated by the signed Hammersley process satisfies the condition in the theorem, and conversely, every string that satisfies the conditions can be generated by the process. For the first claim we need a couple of simple lemmas:

Lemma 9. *Every word in $L(H_k^{sign})$ starts with a k^+ or a k^-.*

Proof. Digits are only modified by insertions to their left. So the leftmost k^+ or k^- remains unchanged.

□

Lemma 10. *$L(H_k^{sign})$ is closed under prefix.*

Proof. Given $w \in L(H_k^{sign})$ and a position p in w, just ignoring all insertions to the right of p yields a construction for the prefix of w ending at p.

□

Lemma 11. *Every word in $L(H_k^{sign})$ is k-dominant.*

Proof. Let us count the number of particles of type $(k-i)^+$ in z, $i \geq 1$. Such a particle arises from a single k^+ particle through a chain of insertions

$$k^+ \xrightarrow{k^-} (k-1)^+ \xrightarrow{k^-} \ldots \xrightarrow{k^-} (k-i)^+$$

requiring i particles of type k^- and killing the original particle of type k^+. Similarly, a particle of type $(k-i)^-$, $i \geq 1$, arises from a k^- particle through a chain

$$k^- \xrightarrow{k^+} (k-1)^- \xrightarrow{k^+} \ldots \xrightarrow{k^+} (k-i)^-$$

requiring i particles of type k^+ and killing one particle of type k^-.

In addition to the insertions that kill particles, denote by λ^+ the number of particles of type k^+ that, when inserted, don't have any $1^-, \ldots, k^-$ to their right, (hence they don't kill any particle and aren't counted by the above chains) and, similarly, by λ^- particles of type k^+ that don't kill any particle.

Thus the number of particles of type k^+ in word z is

$$|z|_{k^+} = \lambda^+ + \sum_{i \geq 1} i \cdot |z|_{(k-i)^-} - \sum_{i \geq 1} |z|_{(k-i)^+} \tag{3}$$

In the previous equation we have simply tallied up the number of particles k^+ originally inserted, subtracting those that eventually end up as a $(k-i)^+$ for some $i \geq 1$. Similarly

$$|z|_{k^-} = \lambda^- + \sum_{i \geq 1} i \cdot |z|_{(k-i)^+} - \sum_{i \geq 1} |z|_{(k-i)^-} \tag{4}$$

Putting the condition $\lambda^+, \lambda^- \geq 0$ (the appropriate one being strictly > 0) we infer that z is k-dominant. □

From Lemmas 9, 10 and 11 it follows that any word $z \in L(H_k^{sign})$ satisfies the conditions (1) and (2) in the theorem.

We show the converse as follows:

Lemma 12. *Every word z satisfying conditions in Theorem 7 is an output of the signed Hammersley process.*

Proof. Assume otherwise. Consider a z of smallest length that is not the output of the signed Hammersley process. Clearly $|z| > 1$, since $z = k^+$ and $z = k^-$ are outputs.

Consider the last occurrence l of one of the letters k^+, k^- in z. Without loss of generality assume that $l = k^+$ (the other case is similar).

If to the right of l one had only positive letters (if any) then consider the word w_1 obtained by pruning the letter l from z. Its prefixes p are either prefixes of z (hence k-dominant) or are composed of the prefix w_0 of z cropped just before l plus some more positive letters. For such a prefix we have to verify the two conditions (1) and (2) are satisfied: The second one follows directly from condition (2) for the word z, since all letters of z with negative polarity are present in p. As for the first one, it follows from the fact that condition (1) is valid for w_0, since some more positive letters are added. In conclusion, all prefixes of w_1 are k-dominant. As $|w_1| = |z| - 1$, by the minimality of z, it follows that w_1 is an output of the signed Hammersley process. But then the process can output z by simply simulating the construction for w_1 and then inserting the last k^+ of z into w_1 in its proper position.

The remaining case is when $l = k^+$ has some negative letters to its right. Assume that s is the first letter of negative type to the right of l. s cannot be k^-, since we already saw the last of k^+, k^- at position l. Let w_2 be the word obtained by removing l from z and increasing the value of s by 1. We claim that w_2 and all its prefixes are k-dominant.

$$|w_2|_{k^+} - \sum_{i=1}^{k} i \cdot |w_2|_{(k-i)^-} + \sum_{i=0}^{k-1} |w_2|_{i^+} = |z|_{k^+} - \sum_{i=1}^{k} i \cdot |z|_{(k-i)^-} + \sum_{i=0}^{k-1} |z|_{i^+} \quad (5)$$

In fact this is true for every prefix of w_2, since losing a k^- is not counted, and adding one doesn't change the sum.

$$|w_2|_{k^-} - \sum_{i=1}^{k} i \cdot |w_2|_{(k-i)^+} + \sum_{i=0}^{k-1} |w_2|_{i^-} = |z|_{k^-} - \sum_{i=1}^{k} i \cdot |z|_{(k-i)^+} + \sum_{i=0}^{k-1} |z|_{i^-} \geq 0 \quad (6)$$

Again, this is true for every prefix of z containing s, since losing k^- is compensated by increasing s, so the sum doesn't change.

Because of the minimality of z, w_2 is an output of the signed Hammersley process. But then we can obtain z as an output of this process by first simulating the construction of w_2 and then inserting l (which corrects the value of s as well).

□

3.1 Proof of Corollary 8

Proof. The claim that $L(H_k^{sign})$ is a the intersection of two deterministic one-counter languages follows from Theorem 7 as follows: We construct two one-counter PDA's, A_1

and A_2, both having input alphabet Γ_k. Each of them enforces one of the conditions (1) and (2), respectively, including the fact that the appropriate inequality is strict. We describe A_1 in the sequel, the functioning of A_2 is completely analogous.

The stack alphabet of A_1 comprizes two stack symbols, an effective symbol $*$ and the bottom symbol Z. Transitions are informally specified in the following manner:

- A_1 starts with the stack consisting of the symbol Z. If the first symbol it reads is neither k^+ nor k^-, A_1 immediately rejects.
- A_1 has parallel (but disjoint) sets of states corresponding to the situations that the first letter is a k^+, respectively a k^-.
- The first alternative requires that the difference in (1) is strictly positive. A_1 enforces this by first ignoring the first letter and then enforcing the fact that the difference in (1) *corresponding to the suffix obtained by dropping the first letter* is always ≥ 0. This is similar to the behavior in the case when the first letter is a k^-, which we describe below.
- when reading any subsequent positive symbol, A_1 pushes a $*$ on stack.
- if the next symbol of the input is $(k-i)^-$, $i \in 1 \ldots k$, A_1 tries to pop i star symbols from the stack. If this ever becomes impossible (by reaching Z), A_1 immediately rejects.
- When reaching the end of the word A_1 accepts.

□

Observation 1 *A nagging question that would seem to be a simple exercise in formal language theory (but which so far has escaped us) is proving that $L(H_k^{sign})$ is not a context free language.*

This is, we believe, plausible since the words in $L(H_k^{sign})$ have to satisfy not one but two inequalities, (1) and (2) that constrain their Parikh distributions. Verifying such inequalities would seemingly require two stacks.

We have attempted (but failed) to prove this by applying Ogden's lemma. So we leave this as an open question.

4 Computing the formal power series of the signed Hammersley process

In this section we study the following problem: given a word $z \in \Gamma_k^*$, compute the number of copies of z generated by the signed Hammersley process. The problem is, of course, a generalization of the one in the previous section: we are interested not only if a word can/cannot be generated, but in how many ways.

The real motivation for developing such an algorithm is its intended use (similar to the use of the analog algorithm from the unsigned case in [9]) to attempt to analyze the scaling in the Ulam-Hammersley problem for heap decompositions, and is beyond the scope of the current paper. Roughly speaking, by studying the multiplicity of words in the signed Hammersley process we hope to be able to exactly sample from the distribution of words for large n and estimate, using such a sampling process the scaling constant for the decomposition of a random signed permutation into a minimal number of heaps.

The corresponding problem for the unsigned case has been considered in several papers [14,15,4,5], and the application of formal power series techniques to the scaling problem is described in [7].

Multiplicity(k, w) :

Input: $k \geq 1, w \in \Gamma_k^*$
Output: $F_k(w)$
$S = 0.w = w_1 w_2 \dots w_n$

if $w \notin L(H_k^{sign})$ **then**
  return 0
if $w == k^+$ or $w == k^-$: **then**
  return 1
for $i = 1$ to n **do**
  if $w_i == k^+$ and $\exists l : i+1 \leq l \leq n : w_l \in \{0^-, \dots, k^-\}$: **then**
    let $r = min(\{n+1\} \cup \{l \geq i+1 : w_l \in \{1^-, \dots, k^-\})$
    if $r == n+1$ or $w_r \neq k^-$ **then**
      for $j = 1$ to $r-1$ **do**
        if $w_j == 0^-$ **then**
          let $z = w_1 \dots w_{i-1} w_{i+1} \dots w_{j-1} 1^- w_{j+1} \dots w_n$
          let $S := S + Multiplicity(k, z)$
    if $r \leq n$ **then**
      let $z := w_1 \dots w_{i-1} w_{i+1} \dots (w_r + 1)^- w_{r+1} \dots w_n$
      let $S := S + Multiplicity(k, z)$

  if $w_i == k^-$ and $\exists l : i+1 \leq l \leq n : w_l \in \{0^+, \dots, k^+\}$: **then**
    let $r = min(\{n+1\} \cup \{l \geq i+1 : w_l \in \{1^+, \dots, k^+\})$
    if $r == n+1$ or $w_r \neq k^+$ **then**
      for $j = 1$ to $r-1$ **do**
        if $w_j == 0^+$ **then**
          let $z := w_1 \dots w_{i-1} w_{i+1} \dots w_{j-1} 1^+ w_{j+1} \dots w_n$
          let $S := S + Multiplicity(k, z)$
    if $r \leq n$ **then**
      let $z := w_1 \dots w_{i-1} w_{i+1} \dots (w_r + 1)^+ w_{r+1} \dots w_n$
      let $S := S + Multiplicity(k, z)$

  if $w_i == k^+$ or $w_i == k^-$ **then**
    let $Z := w_1 \dots\dots w_{i-1} w_{i+1} \dots w_n$
    let $S := S + Multiplicity(k, z)$

Figure 2. Algorithm Multiplicity(k,w)

Theorem 13. *Algorithm Multiplicity in Figure 2 correctly computes series F_k.*

Proof. The idea of the algorithm is simple, in principle: we simply attempt to "reverse time" and try to identify strings z that can yield the string w in one step of the process. We then add the corresponding multiplicities of all preimages z.

We go from z to w by inserting a k^+ or a k^- and deleting one life from the closest non-zero letter of z that has the correct polarity and is to the right of the newly inserted letter.

Not all letters of k^+, k^- in w can be candidates for the inserted letter, though: a candidate k^+, for instance, cannot have a k^+ as the first letter with positive polarity to its right. That is, if we group the k^+'s in w in blocks of consecutive occurrences, then the candidate k^+'s can only be the right endpoints of a block.

As for the letter l the new k^+ acted upon in z, in w it cannot be a k; also, there cannot be any letters with positive polarity, other than zero, between the new k^+ and l.

So l is either one of the 0^+'s at the right of the new k^+ or the first nonzero positive letter (if not k^+). It is also possible that l does not exist.

Analogous considerations apply if the newly inserted letter is a k^-. □

5 Motivation: The Ulam-Hammersley problem for the heap decomposition of signed permutations

The *Ulam-Hammesley problem* [19,12] can be described as follows: estimate the asymptotic behavior of the expected length of the longest increasing subsequence of a random permutation $\sigma \in S_n$. The correct scaling is $E_{\sigma \in S_n}[LIS[\sigma]] = 2\sqrt{n}(1+o(1))$, however substantially more is known, and the problem has deep connection with several mathematical concepts and areas, including Young tableaux (see, e.g., [17]), random matrix theory [2] and interacting particle systems [1].

The following concept has been defined (for $k = 2$, and can be easily generalized as presented) in [9]: a sequence of integers is *(k)-heapable* if the elements of a sequences can be inserted successively in a k-ary tree, min-heap ordered (i.e., the label of the child is at least as large as that of the parent), not-necessarily complete, so that insertions are always made as a leaf.

Heapable sequences[1] can be viewed as a (loose) generalization of increasing sequences: instead of inserting sequences into an increasing array, so that each node (except the last one) has exactly one successor, we allow a "more relaxed version" of this data structure, in the form of a k-ary tree with a min-heap ordering on the nodes. In other words a node has multiple choices for the insertion (instead of a single one), and using all these leaves can accommodate some limited form of disorder between consecutive elements, although values in such a sequence "tend to increase", simply because the available positions will appear lower and lower in the tree.

It is natural, therefore, to attempt to generalize the Ulam-Hammersley problem to heapable sequences. The direct extension is problematic, though: as discussed in [9], the problem of computing the longest heapable subsequence has unknown complexity (see [10,11] for some related results). A more promissing alternative is the following: by Dilworth's theorem the longest increasing subsequence is equal to the minimal number of *decreasing sequences* in which we can partition the sequence. So the "correct" extension of the longest increasing subsequence is (see [14,3]): **decompose a random permutation $\sigma \in S_n$ into the minimal number of heap-ordered k-ary trees**, and study the scaling of the expected number of trees in such an optimal decomposition.

The longest increasing subsequence problem has also been studied for *signed* or even *colored* permutations [8]. It is reasonable to consider a similar problem for heapability. This is what we do in this paper. Note, though, that the variant we consider here is not the same as the one in [8]. Specifically, in [8] the author requires that the "colors" (i.e., signs) of two adjacent nodes that are in a parent-child relationship are the same. By contrast we require that *the colors of any two adjacent nodes are different.*

Definition 14. *A signed permutation of order n is a pair (σ, τ), with σ being a permutation of size n and $\tau : [n] \to \{\pm 1\}$ being a sign function.*

[1] in this motivating discussion we refer to the original concept, omitting the setting in this paper where letters have polarities

Definition 15. *Given integer $k \geq 1$, signed permutation (σ, τ) is called $\leq k$-heapable if one can successively construct k (min) heap-ordered binary trees (not necessarily complete) $H_0, H_1, \ldots, H_{k-1}$ such that*

- *at time $i = 0$ $H_0, H_1, \ldots H_{k-1}$ are all empty.*
- *for every $1 \leq i \leq n$ element $\sigma(i)$ can be inserted as a new leaf in one of $H_0, H_1, \ldots, H_{k-1}$, such that if $\sigma(j)$, the parent of $\sigma(i)$ exists, $j < i$, then ($\sigma(j) < \sigma(i)$ and) $\tau[i] = -\tau[j]$.*

(σ, τ) will be called k-heapable iff k is the smallest parameter such that (σ, τ) is $\leq k$-heapable. We will write heapable instead of 1-heapable.

First we note that heapability of signed permutations is not a simple extension of heapability of ordinary permutations:

Observation 2 *Heapability of permutation σ is not equivalent to heapability of any fixed signed-version of σ. In particular if $\tau_0[i] = 1$ for all $i = 1, \ldots, n$ then every signed permutation (σ, τ_0) is n-heapable (even though σ might be heapable as a permutation). This is because the sign condition forces all elements of σ to start new heaps, as all values, having the same sign, cannot be inserted as children of any other nodes.*

On the other hand a connection between ordinary heapability and that of signed permutations does exist after all:

Theorem 16. *The following are true:*

- *If (σ, τ) is heapable (as a signed permutation) then σ is heapable (as an ordinary permutation).*
- *There is a polynomial time algorithm that takes as input a permutation $\sigma \in S_n$ and produces a sign $\tau : [n] \to \{\pm 1\}$ such that if σ is heapable (as an ordinary permutation) then (σ, τ) is heapable (as a signed permutation).*

Proof. – Trivial, since ordinary heapability does not care about sign restrictions.

– A simple consequence of the greedy algorithm for (ordinary) heapability [9]: roughly speaking, given a permutation σ there is a canonical way of attempting to construct a min-heap for σ: each element $\sigma(i)$ is added as a child of the largest element $\sigma(j) < \sigma(i)$, $j < i$ that still has a free slot. If no such $\sigma(j)$ is available then σ is not heapable. Further, denote by $p[i]$ the index of $\sigma(j)$ in the permutation σ (i.e., $p[i] = j$).
We use this algorithm to construct signing τ inductively. Specifically, set $\tau(1) = 1$. Given that we have constructed $\tau(j)$ for all $j < i$, define $\tau(i) = -\tau(p[i])$. Then (σ, τ) is heapable as a signed permutation, since the greedy solution for σ is legal for (σ, τ).

□

5.1 A greedy algorithm for the optimal heap decomposition of signed permutations

We now give a greedy algorithm for decomposing signed permutations into a minimal number of heap-ordered k-ary binary trees. The algorithm is presented in Figure 4 and is based on the concept of *slots*. These are positions where a new number can be inserted. So adding one number to a heap removes one slot and creates k new ones.

Each x_i will either be added to an existing heap or will start a new heap. We search for the highest value less than x_i having opposite sign to $\sigma(i)$, and we will create k *slots of value* $\sigma(i)$, allowing the subsequent insertion of values in the interval $[\sigma(i), \infty)$.

Example 17. Consider the sequence of insertions in a heap-ordered tree for a permutation (σ, τ) with initial prefix $\sigma(1) = 1, \tau(1) = -1$, $\sigma(2) = 8, \tau(2) = +1$, $\sigma(3) = 15, \tau(3) = -1$, shortly $\sigma = \{1, 8, 15\}$ and $\tau = \{-, +, -\}$ and $k = 2$. There is essentially a unique way (displayed in Figure 3) to insert these elements successively into a single heap: $\sigma(1)$ at the root, $\sigma(2)$ as a child of $\sigma(1)$, $\sigma(3)$ as a child of $\sigma(2)$. Note that $\sigma(3)$ needs to be a child of $\sigma(2)$ since the condition $\tau(3) = -1$ forbids, e.g., placing $\sigma(3)$ as a child of $\sigma(1)$.

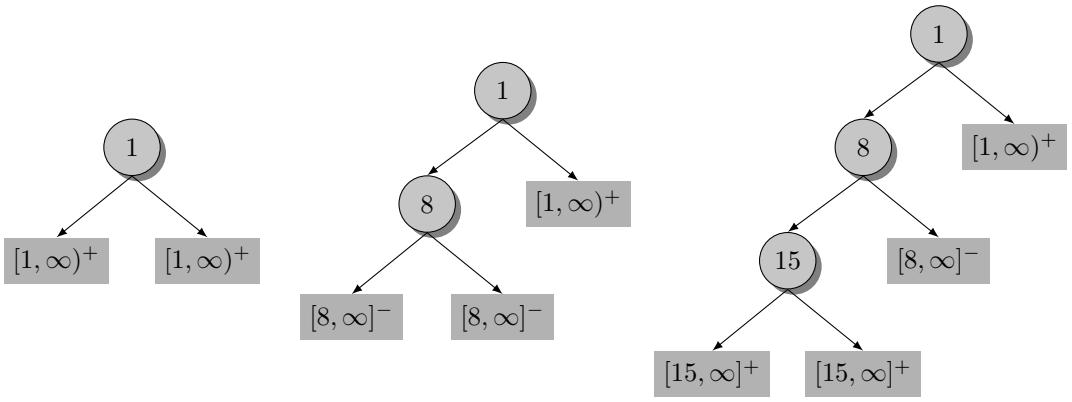

Figure 3. Nodes and slots.

Intuitively the interval of a slot describes the constraints imposed on an integer to be inserted in that position in order to satisfy the heap constraint. For example, after inserting $\sigma(2)$ the unique free slot of the root has value 1, since all children of the root must have values bigger than $\sigma(1) = 1$. After inserting $\sigma(3)$ the unique free slot of node hosting $\sigma(2)$ has value 8, since any descendant of this node must have value at least 8.

Our main result of this section is:

Theorem 18. *The algorithm GREEDY, presented in Fig. 4, decides the heapability of an arbitrary signed permutation and, more generally, constructs an optimal heap decomposition of* (σ, τ).

Proof. The intuition of the proof is that inserting integers into a heap the greedy way makes the collection of available slots grow "as slowly as possible" compared to any other insertion method. We will formalize this by introducing a concept of dominance between multisets and show that dominance is preserved by one insertion step. Since our slots have polarities we have to actually argue separately for the multisets of positive and negative multisets. The importance of dominance is the following: consider the scenario where a newly inserted element (by the greedy algorithm) creates a new heap because no compatible slots are available. Then by dominance no compatible slots are available even in the optimal insertion schedule. Hence greedy creates no more heaps than the optimal solution.

Proving correctness of the algorithm requires the following

$GREEDY(\sigma, \tau)$:

INPUT: $\sigma = (\sigma(1), \sigma(2), \ldots, \sigma(n))$ a permutation in S_n and
$\tau = (\tau(1), \tau(2), \ldots, \tau(n))$ a list of $\{+, -\}$ signs

start with empty heap forest $T = \emptyset$
for $i = 1, \ldots, n$ **do**
    if there exists a slot where $\sigma(i)$ can be inserted, according to the sign $\tau(i)$ **then**
        insert $\sigma(i)$ in the slot with the largest compatible value
    else
        start a new heap consisting of $\sigma(i)$ only.

Figure 4. The greedy algorithm for decomposing a signed permutation into a forest of heaps.

Definition 19. *Given a heap forest T, define the positive signature of T denoted $sig^+(T)$, to be the vector containing the (values of) free slots with positive polarity in T, sorted in non-decreasing order. The negative signature of T, denoted by $sig^-(T)$, is defined analogously.*

Given two binary heap forests T_1, T_2, T_1 dominates T_2 if

- $|sig^+_{T_1}| \leq |sig^+_{T_2}|$ *and inequality* $sig^+_{T_1}[i] \leq sig^+_{T_2}[i]$ *holds for all* $1 \leq i \leq |sig^+_{T_1}|$.
- *Similarly for negative signatures.*

Lemma 20. *Let T_1, T_2 be two heap-order forests such that T_1 dominates T_2. Insert a new element x greedily in T_1 (i.e., at the largest slot with value less or equal to x, or as the root of a new tree, if no such slot exists). Also insert x into an arbitrary compatible slot in T_2. These two insertions yield heap-ordered forests T'_1, T'_2, respectively. Then T'_1 dominates T'_2.*

Proof. We need the following definition

Definition 21. *Given integer $k \geq 1$, a k-multiset is a multiset A such that each element of A has multiplicitly at most k, that is a function $f : A \to \{0, 1, \ldots, k\}$.*

Definition 22. *Given multiset A, a Hammersley insertion of an element x into A is the following process:*

- *x is given multiplicity k in A.*
- *Some element $y \in A$, $y > x$ (if any such y exists), has its multiplicity reduced by 1.*

It is a greedy Hammersley insertion if (when it exists) y is the smallest element of A larger than x having positive multiplicity.

Definition 23. *Given two k-multisets A_1 and A_2 of elements of Γ_k, we define A_i^+, A_i^- the submultisets of A_i, $i = 1, 2$ consisting of elements of A_i with positive (negative) polarity only. We say that A_1 dominates A_2 iff:*

- $|A_1^+| \leq |A_2^+|$ *and inequality* $A_1^+[i] \leq A_2^+[i]$ *holds for all* $1 \leq i \leq |A_1^+|$. *In order for this equation to make sense, we see the multisets A_1^+, A_2^+ as sorted vectors.*
- *Similarly for negative polarities.*

The following result was not explicitly stated in [14] but was implicitly proved, as the basis of the proof of Lemma 1 from [14] (the analog of Lemma 20 for the unsigned case):

Lemma 24. *If A_1 and A_2 are multisets of integers such that A_1 dominates A_2. Consider x an element not present in either set and let A'_1, A'_2 be the result of greedy Hammersley insertion into A_1 and an arbitrary Hammersley insertion into A_2.*

Then A'_1 dominates A'_2.

Rather than repeating the proof, we refer the reader to [14]. To be able to prove Lemma 20 we also need the following:

Lemma 25. *Let A_1 and A_2 be multisets of integers such that A_1 dominates A_2. The following are true:*

- *Let A_1 and A_2 be k-multisets of integers and let x be an integer that does not appear in A_1, A_2. Then, if A'_1, A'_2 represent the result of inserting x* ***with multiplicity*** *k in A_1, A_2, respectively, then A'_1 dominates A'_2.*
- *Let A_1 and A_2 be k-multisets of integers and let x be an integer that appears in both A_1, A_2 with the same multiplicity. Then, if A'_1, A'_2 represent the result of deleting x from A_1, A_2, respectively, then A'_1 dominates A'_2.*

Proof. We prove the two statements as follows:

- It is clear that inserting x adds k to the cardinality of both A_1, A_2. Since $|A_1| \leq |A_2|$, we have $|A'_1| \leq |A'_2|$
  Let i_1 be the smallest index such that $A_1(i_1) > x$ and let i_2 be the smallest index such that $A_2(i_2) > x$. Because A_1 dominates A_2, $i_1 \geq i_2$.
  Recall that we regard multisets as sorted vectors. To prove that domination holds after the insertion of x as well, we have to prove, therefore, that $A'_1[i] \leq A'_2[i]$ for all indices $1 \leq i \leq |A'_1|$. Indeed, let i be such an index:
  - Case 1: $i < i_2$. Then $A'_1[i] = A_1[i]$ and $A'_2[i] = A_2[i]$. Indeed, all these elements are smaller than x. The desired inequality follows from the fact that A_1 dominates A_2.
  - Case 2: $i \geq i_1 + k$. Then $A'_1[i] = A_1[i-k]$ and $A'_2[i] = A_2[i-k]$. Inequality follows from the fact that A_1 dominates A_2.
  - Case 3: $i_2 \leq i < i_1$: Then $A'_1[i] = A_1[i] < x$ and $A'_2[i] = x$. So $A'_1[i] \leq A'_2[i]$.
  - Case 4: $i_1 \leq i < i_1 + k$: Then $A'_1[i] = x$ and $A'_2[i] \geq A'_2[i_1] \geq A'_2[i_2] = x$. So $A'_1[i] \leq A'_2[i]$.
- It is clear that x subtracts k to the cardinality of both A_1, A_2. To prove domination we have to, therefore, prove that $A'_1[i] \leq A'_2[i]$ for all $1 \leq i \leq |A'_1|$. Let i_1 be the smallest index such that $A_1(i_1 + k) > x$ and i_2 be the smallest index such that $A_2(i_2 + k) > x$. i_1, i_2 are well-defined since A_1, A_2 contain k copies of x. Because A_1 dominates A_2, $i_1 \geq i_2$. The first position of x in A_1 is i_1, the last is $i_1 + k - 1$; the first position of x in A_2 is i_2, the last is $i_2 + k - 1$.
  - Case 1: $i < i_2$. Then $A'_1[i] = A_1[i]$ and $A'_2[i] = A_2[i]$. Inequality follows from the fact that A_1 dominates A_2.
  - Case 2: $i \geq i_1$. Then $A'_1[i] = A_1[i-k]$ and $A'_2[i] = A_2[i-k]$. Inequality follows from the fact that A_1 dominates A_2.
  - Case 3: $i_2 \leq i < i_1$: Then $A'_1[i] = A_1[i] < x$ and $A'_2[i] = A_2[i+k] > x$. So $A'_1[i] \leq A'_2[i]$.
  - Case 4: $i_1 \leq i < i_1 + k$: Then $A'_1[i] < x$ and $A'_2[i] \geq A'_2[i_1] \geq A'_2[i_2] = A_2[i_2 + k] > x$. So $A'_1[i] \leq A'_2[i]$.

Now the proof of Lemma 20 follows: the effect of inserting an element with positive polarity x^+ greedily into T_1 can be simulated as follows:

– perform a greedy Hammersley insertion of x^- into $sig^-(T_1)$.
– remove x^- from $sig^-(T_1)$, and insert x^+ into $sig^+(T_1)$.

On the other hand we can insert x^+ into T_2 as follows:

– perform a Hammersley insertion of x^- into $sig^-(T_2)$.
– remove x^- from $sig^-(T_2)$, and insert x^+ into $sig^+(T_2)$.

By applying Lemma 24 and 25 we infer that after the insertion of x^+ T_1' dominates T_2'. The insertion of an element with negative polarity is similar.

□

Using Lemma 20 we can complete the proof of Theorem 18 as follows: by domination, whenever no slot of T_1 can accommodate x (which, thus, starts a new tree) then the same thing happens in T_2 (and thus x starts a new tree in T_2 as well). So the greedy algorithm is at least as good as any sequence of insertions, which means it is optimal.

□

5.2 Connection with the Signed Hammersley process

We are now in a position to explain what role does the signed Hammersley process play in the Ulam-Hammersley problem for the heap decomposition of signed permutations: to each heap-ordered forest F associate a word w_F over Γ_k^* which describes the relative positions of slots in F. Specifically, sort the leaves of F according to their value. If leaf f has at a certain moment $p \leq k$ slots of, say, negative polarity, then encode this into w_F by letter p^-.

Example 26. Let us consider the trees of Example 17. For the first tree the associated word is 2^+. For the second tree the word is 1^+2^-. For the third it is $1^+1^-2^+$.

Observation 3 *Note that when inserting a new element into the heap forest using the greedy algorithm the associated word changes exactly according to the signed Hammersley process*[2]*. In fact we can say more: starting a new heap-ordered tree corresponds to moments when we insert a new k^+ or k^- (whichever is appropriate at the given moment) without subtracting any 1 from the current word. So, if z is the outcome of a sequence of greedy insertions then $trees(z)$, the number of heap-ordered k-ary trees created in the process is equal to $\lambda^+ + \lambda^-$ (in the notation of equations 3 and 4), and is equal to*

$$trees_k(z) = |z|_k - \sum_{i=1}^{k} i \cdot |z|_{k-i} + \sum_{i=1}^{k} |z|_{k-i} = |z|_k - \sum_{i=1}^{k} (i-1)|z|_{k-i}$$

Of course, each word z may arise with a different multiplicity in the signed Hammersley process. So to compute the expected number of heap-ordered k-ary trees generated

[2] or, rather, the signed Hammersley process that removes a one to the left, rather than to the right. We would have to use min-heaps to obtain the signed Hammersley process. But this change is inconsequential.

by using the greedy algorithm with a random signed permutation of length n as input we have to compute amount

$$Z_n^k := \frac{1}{2^n \cdot n!} \sum_{z \in (\Gamma^k)^n} F_k(z) \cdot trees_k(z). \tag{7}$$

The last equation in Observation 3 is what motivated us to give Algorithm 2 for computing the formal power series F_k. We defer, however, the problem of experimentally investigating the scaling behavior of Z_n^k as $n \to \infty$ using Algorithm 2 to subsequent work.

6 Conclusions

The main contribution of this paper is to show that a significant number of analytical tools developed for the analysis of the Ulam-Hammersley problem for heapable sequences [14,7] extend to the case of signed permutations. While going along natural lines, the extension has some moderately interesting features: for instance the languages in the sign case appear to have slightly higher grammatical complexity than those in [7] for the ordinary process.

The obvious continuation of our work is the investigation of the Ulam-Hammer -sley problem for signed (and, more generally, colored) permutations. We do not mean only the kind of experiments alluded to in Observation 3. For instance it is known that the scaling in the case of ordinary permutations is logarithmic [4,5], even though the proportionality constant is not rigorously known. Obtaining similar results for the signed Hammersley (tree) process would be, we believe, interesting.

On the other hand, as we noted, our extension to signed permutations is not a direct version of the one in [8]. Studying a variant of our problem consistent with the model in [8] (or studying the longest increasing subsequence in the model we consider) is equally interesting.

References

1. D. Aldous and P. Diaconis: *Hammersley's interacting particle process and longest increasing subsequences.* Probability theory and related fields, 103(2) 1995, pp. 199–213.
2. J. Baik, P. Deift, and T. Suidan: *Combinatorics and random matrix theory*, vol. 172, American Mathematical Soc., 2016.
3. J. Balogh, C. Bonchiş, D. Diniş, G. Istrate, and I. Todinca: *On the heapability of finite partial orders.* Discrete Mathematics and Theoretical Computer Science, 22(1) 2020, paper # 17.
4. A.-L. Basdevant, L. Gerin, J.-B. Gouéré, and A. Singh: *From Hammersley's lines to Hammersley's trees.* Probability Theory and Related Fields, 2016, pp. 1–51.
5. A.-L. Basdevant and A. Singh: *Almost-sure asymptotic for the number of heaps inside a random sequence.* Electronic Communications in Probability, 23(17) 2018.
6. J. Berstel and C. Reutenauer: *Noncommutative rational series with applications*, vol. 137, Cambridge University Press, 2011.
7. C. Bonchiş, G. Istrate, and V. Rochian: *The language (and series) of Hammersley-type processes*, in Proceedings of the 8th Conference on Machines, Computation and Universality (MCU'18), vol. 10881 of Lecture Notes in Computer Science, 2018.
8. A. Borodin: *Longest increasing subsequences of random colored permutations.* The Electronic Journal of Combinatorics, 6(1) 1999, p. R13.

9. J. Byers, B. Heeringa, M. Mitzenmacher, and G. Zervas: *Heapable sequences and subseqeuences*, in Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics (ANALCO'2011), SIAM Press, 2011, pp. 33–44.
10. K. Chandrasekaran, E. Grigorescu, G. Istrate, S. Kulkarni, Y.-S. Lin, and M. Zhu: *Fixed-parameter algorithms for longest heapable subsequence and maximum binary tree*, in 15th International Symposium on Parameterized and Exact Computation, 2020.
11. K. Chandrasekaran, E. Grigorescu, G. Istrate, S. Kulkarni, Y.-S. Lin, and M. Zhu: *The maximum binary tree problem.* Algorithmica, 83(8) 2021, pp. 2427–2468.
12. J. M. Hammersley et al.: *A few seedlings of research*, in Proceedings of the Sixth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Theory of Statistics, 1972.
13. M. A. Harrison: *Introduction to formal language theory*, Addison-Wesley Longman Publishing Co., Inc., 1978.
14. G. Istrate and C. Bonchiş: *Partition into heapable sequences, heap tableaux and a multiset extension of Hammersley's process*, in Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15), Ischia, Italy, vol. 9133 of Lecture Notes in Computer Science, Springer, 2015, pp. 261–271.
15. G. Istrate and C. Bonchiş: *Heapability, interactive particle systems, partial orders: Results and open problems*, in Proceedings of the 18th International Conference on Descriptional Complexity of Formal Systems (DCFS'2016), Bucharest, Romania, vol. 9777 of Lecture Notes in Computer Science, Springer, 2016, pp. 18–28.
16. J. Porfilio: *A combinatorial characterization of heapability*, Master's thesis, Williams College, May 2015, available from `https://librarysearch.williams.edu/permalink/01WIL_INST/1htsahc/alma991013795933102786`.
Accessed: July 2024.
17. D. Romik: *The surprising mathematics of longest increasing subsequences*, Cambridge University Press, 2015.
18. A. Salomaa and M. Soittola: *Automata-theoretic aspects of formal power series*, Springer Science & Business Media, 2012.
19. S. M. Ulam: *Monte carlo calculations in problems of mathematical physics.* Modern Mathematics for the Engineers, 261 1961, p. 281.
20. L. G. Valiant and M. S. Paterson: *Deterministic one-counter automata.* Journal of Computer and System Sciences, 10(3) 1975, pp. 340–350.

Cdbgtricks: Strategies to update a compacted de Bruijn graph

Khodor Hannoush[1⋆], Camille Marchet[2], and Pierre Peterlongo[1]

[1] Univ. Rennes, Inria, CNRS, IRISA - UMR 6074, Rennes, F-35000 France
khodor.hannoush@inria.fr
pierre.peterlongo@inria.fr
[2] Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
camille.marchet@univ-lille.fr

Abstract. We propose `Cdbgtricks`, a new method for updating a compacted de Bruijn graph when adding novel sequences, such as full genomes. Our method indexes the graph, enabling to identify in constant time the location (unitig and offset) of any k-mer. The update operation that we propose also updates the index. Our results show that `Cdbgtricks` is faster than `Bifrost` and `GGCAT`. We benefit from the index of the graph to provide new functionalities, such as reporting the subgraph that shares a desired percentage of k-mers with a query sequence with the ability to query a set of reads. The open-source `Cdbgtricks` software is available at `https://github.com/khodor14/Cdbgtricks`.

Keywords: compacted de Bruijn graphs, k-mers, data structures, genomics, indexing

1 Introduction

The de Bruijn graph is one of the fundamental data structures that play crucial roles in computational biology. It is tremendously used in various applications, including but not limited to genome assembly [20,23], read error correction [16,13], read alignment [17,4] and read abundance queries [18]. The de Bruijn graph is a data structure in which the nodes represent the distinct substrings of length k of a set of strings, called k-mers, and the edges link the nodes that share an overlap of $(k-1)$. In a compacted de Bruijn graph, nodes along each maximal non-branching path of the de Bruijn graph are compacted into a single node representing a sequence of length $\geq k$, called a unitig. The de Bruijn graph can be constructed from a set of assembled genomes or a set of reads.

Several methods have been proposed in the literature for compacted de Bruijn graph construction, including `PanTools` [24], `Bcalm2` [6], `TwoPaCo` [21], `deGSM` [9], `Bifrost` [11], `Cuttlefish2` [14], `GGCAT` [7], and `FDBG` [8]. As genomic databases grow, there is a demand for dynamic de Bruijn graph data structures supporting sequence additions. `BufBoss` [1], `DynamicBoss` [2], and `FDBG` [8] support the addition and the deletion of nodes and edges in de Bruijn graphs.

Nevertheless, the performance of `BufBoss`, `DynamicBoss` and `FDBG` in sequence addition falls short compared to `Bifrost`. We refer the reader to the results of the `BufBoss` paper [1]. Despite being faster in adding new sequences, `Bifrost` builds a new graph from the sequences to be added, using its construction method, and then it merges the two graphs. The reliance on constructing a graph from the new sequences introduces computational overhead, potentially limiting the scalability and efficiency

⋆ Corresponding author

Khodor Hannoush, Camille Marchet, Pierre Peterlongo: Cdbgtricks: Strategies to update a compacted de Bruijn graph, pp. 86–103.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5

of `Bifrost`, particularly when dealing with large graphs, highlighting the need for a more efficient updating mechanism.

Recent advances have focused on developing memory- and time-efficient indexing structures for k-mers. Some notable methods, such as `BLight` [19], `SSHash` [22], `GGCAT` [7], and `Pufferfish` [3], are recognized for their efficiency in this regard. However, it is important to note that these methods are static, meaning that they cannot easily incorporate new data. This limitation becomes especially problematic with large datasets when construction time must be paid for every addition of new sequences.

Indexing methods are used as well for read queries, read mapping, and read alignment on de Bruijn graph. Read mapping against the de Bruijn graph has been studied extensively. `Bifrost` [11], `GGCAT` [7] and `SSHash` [22] proposed read query methods. The limitation of `Bifrost` and `SSHash` is the need to compare any queried k-mer with all k-mers having the same minimizer, the smallest substring of length m according to some order where $0 < m < k$. Although it is an efficient design to reduce memory usage, it slows down when negative queried k-mers.

In this paper, we propose "`Cdbgtricks`", a novel strategy, and a method to add sequences to an existing uncolored compacted de Bruijn graph. Our method takes advantage of `kmtricks` [15] that finds in a fast way what k-mers are to be added to the graph, and our indexing strategy enables us to determine the part of the graph to be modified while computing the unitigs from these k-mers. The index of `Cdbgtricks` is also able to report exact matches between query reads and the graph. We compared `Cdbgtricks` against `Bifrost` and `GGCAT`. Despite `GGCAT` lacking the update feature on the compacted de Bruijn graph, we included it in our comparison due to its potential efficiency in graph construction, which may outperform an update approach. `Cdbgtricks` is up to 2x faster than `Bifrost` on updating a compacted de Bruijn graph on 100 human genomes datasets, and it shows the competitiveness potential against `GGCAT` on larger human genome datasets for which `GGCAT` may not scale due to the high disk requirement. `Cdbgtricks` is up to 3x faster than `Bifrost` and `GGCAT` on updating a compacted de Bruijn graph on a large *E. coli* genomes dataset.

2 Methods

2.1 Preliminary definitions

A string s is a sequence of characters drawn from an alphabet Σ. In this paper, we use the DNA alphabet $\Sigma = \{A, C, G, T\}$ where every character has its complement in Σ. The complement pairs of Σ are (A, T) and (C, G). The reverse complement $\bar{s}$ of s is found by reversing s and then complementing the characters. The canonical form of a string s is the lexicographically smallest string between s and its reverse complement $\bar{s}$. We denote by $|s|$ the length of s. $s[i]$ denotes the i^{th} symbol of s, starting from zero ($s[0]$ represents the first character of s and $s[|s|-1]$ is its last character). Denote by $\mathrm{s}(i, j)$ the substring of s starting at i and ending at $j-1$. A k-mer is a string of length k. For a given k, in our specific context, we denote by $\mathit{pref}(s) = s(0, k-1)$ the $(k-1)$-prefix of a string s, and by $\mathit{suff}(s) = s(|s|-k+1, |s|)$ the $(k-1)$-suffix of s.

Definition 1. de Bruijn Graph: The de Bruijn graph constructed from a set of sequences S is a directed graph $G = (V, E)$ where V represents the set of distinct k-mers of S. When input sequences S are made of raw sequencing data, before constructing the graph, the k-mers of S are counted, and those whose abundance is smaller than a

fixed threshold are considered to contain sequencing errors, thus they are discarded. Note that a node u represents a k-mer x and its reverse complement $\bar{x}$. Two nodes v and w are connected by an edge $e \in E$ from v to w if one of the following holds:

1. $\mathit{suff}(v) = \overline{\mathit{suff}(w)}$
2. $\mathit{pref}(v) = \overline{\mathit{pref}(w)}$
3. $\mathit{suff}(v) = \mathit{pref}(w)$

In any of these three cases, v is an in-neighbor of w, and w is an out-neighbor of v. It is worth mentioning that within the scope of this paper, the edges are not stored explicitly; rather, they are deduced from the nodes.

Definition 2. Path: A path of a dBG is an ordered set of nodes where every two consecutive nodes are connected by an edge.

Definition 3. Unitig: A maximal non-branching path is a path $p = \{f, v_1, v_2, ..., v_{|p|-2}, l\}$ where every v_i has only one in-neighbor and one out-neighbor and f and l do not have this property. A maximal non branching path can be compacted to form a unitig u. The compaction of two nodes v and w can be achieved as follows:

1. If $\mathit{suff}(v) = \mathit{pref}(w)$ then $\mathit{compaction}(v, w) = v \odot w[|w| - 1]$
2. If $\mathit{suff}(v) = \overline{\mathit{suff}(w)}$ then $\mathit{compaction}(v, w) = v \odot \overline{w[0]}$
3. If $\mathit{pref}(v) = \overline{\mathit{pref}(w)}$ then $\mathit{compaction}(v, w) = \overline{w[0]} \odot v$

Without loss of generality, we suppose in what follows that two nodes v and w are in forward-forward direction i.e., $\mathit{suff}(v) = \mathit{pref}(w)$. It should be noted that the rest of the cases remain valid within the scope of this definition. In what follows we will not emphasize if a k-mer x is canonical or not.

Definition 4. Compacted de Bruijn Graph: Replacing the maximal non-branching paths by their unitigs provides a compacted form of the de Bruijn graph. An illustration of a de Bruijn graph and its compacted version is shown in Figure 1.

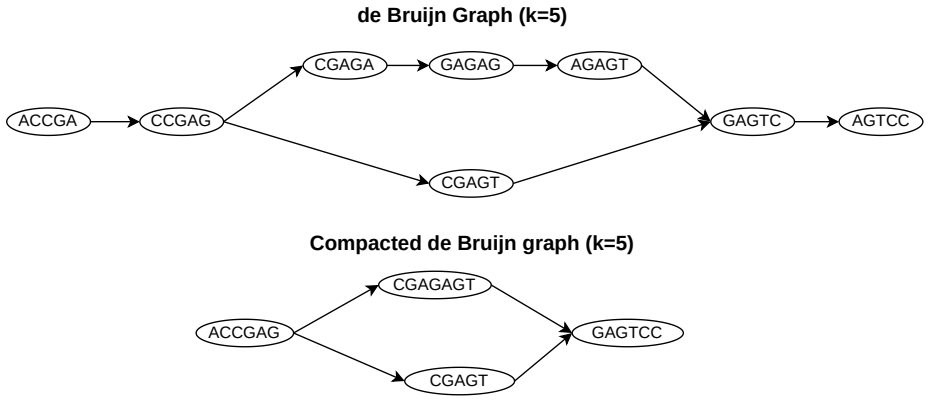

Figure 1. A de Bruijn graph and its compacted de Bruijn graph version.

Definition 5. Minimizer: A minimizer of a string s is a substring q of fixed length m where $m < |s|$ and q is the smallest m-mer of s with respect to some order. In this paper, the order is defined on the basis of a hash function.

Definition 6. Minimal perfect hash function MPHF: Given a set of keys K, a minimal perfect hash function (MPHF) is a function that bijectively maps the elements of K to the elements of the set $I = \{i | 0 \leq i < |K|\}$.

2.2 Overview of the Algorithm

`Cdbgtricks` enables to add a set of new sequences S to a compacted de Bruijn graph G. We denote by K_S the set of k-mers in S and by K_G the set of k-mers in G. The set of k-mers in K_S but not in K_G have to be added to G. We call N this set $K_S \backslash K_G$. To efficiently determine N we use the `kmtricks` [15] tool. To help understand the proposed algorithm, we first describe the process when adding k-mers from N one after another in the compacted de Bruijn graph G. We show later (Section 2.4), how to avoid these $|N|$ individual additions.

When adding a k-mer x from N to the compacted de Bruijn graph G, we distinguish the following cases, as represented in Figure 2:

1. **Add x as a new unitig**. Neither $pref(x)$ nor $suff(x)$ appears in any unitig of G. In this case, the existing unitigs of G are not modified and x is added as a new single unitig in G (Fig 2.a).
2. **Right extension of a unitig**. If $pref(x)$ equals $suff(u)$ for a unitig u of G, and u has no out-neighbor, then u is extended with the last character of x ($u = u \odot x[k-1]$)(Fig 2.b).
3. **Left extension of a unitig**. If $suff(x)$ equals $pref(u)$ for a unitig u of G and u has no in-neighbor, then the first character of x is added to the left of u ($u = x[0] \odot u$) (Fig 2.c).
4. **Merge two unitigs**. If the addition of x leads to the right extension of a unitig u_1 **and** to the left extension of a unitig u_2, then, after the extensions, $suff(u_1) = pref(u_2)$. In this case the two unitigs u_1 and u_2 are merged into a unique unitig $u = u_1 \odot u_2(k, |u_2|)$ (Fig 2.d).
5. **Splitting a unitig**. If $pref(x)$ exists in a unitig u of G, not being the suffix nor the prefix of u, then u is split into two unitigs u_1 and u_2 where $suff(u_1) = pref(u_2) = pref(x)$ and x is added as a single unitig. Respectively, if $suff(x)$ exists in a unitig u of G, not being the suffix nor the prefix of u, then u is split into two unitigs u_1 and u_2 where $suff(u_1) = pref(u_2) = suff(x)$ and x is added as a single unitig (Fig 2.e).

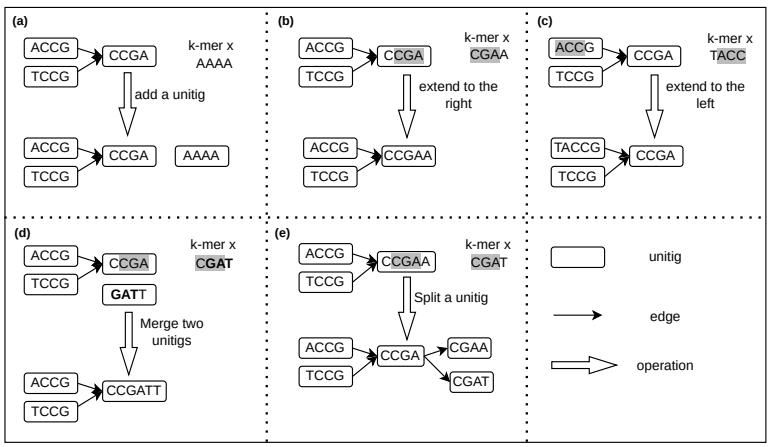

Figure 2. Possible operations when adding a k-mer to a compacted de Bruijn graph with $k = 4$. (a) Adding the k-mer as a new unitig. **(b)** Extending a unitig to the right. **(c)** Extending a unitig to the left. **(d)** Merging two unitigs. **(e)** Split a unitig into two unitigs. Gray and bold sequences represent overlap between the added k-mer and some unitigs of the graph.

These operations rely extensively on the pattern matching of $suff(x)$ and $pref(x)$ in the unitigs of G. In order to rapidly perform these operations, we propose to index the graph, as explained in the next section.

2.3 Indexing the graph

A core operation in `Cdbgtricks` consists in identifying if a $(k-1)$-mer occurs in any unitig of a compacted de Bruijn graph G, and, if this is the case, to determine the couple(s) (unitig id, offset) where it occurs.

This operation is performed twice for each k-mer x of N (for $pref(x)$ and $suff(x)$). Ideally it has to have a $O(1)$ time-complexity and to be fast in practice. This is a very common operation for which existing indexing solutions such as [19,22] are convenient. However, in the context of this work, the specificity is that, when adding sequences to the graph, the indexed data evolve as some unitigs G can be split, merged, extended, and some new unitigs can be added to G. Hence, those static methods are not adapted. We propose the following strategy to cope with this particular situation.

Indexing k-mers for querying $(k-1)$-mers Despite the fact that we query $(k-1)$-mers, we chose to index k-mers instead of $(k-1)$-mers. A $(k-1)$-mer may have up to eight occurrences in G because it can be the suffix of four possible k-mers and the prefix of four possible k-mers. Indexing from one to eight couples (unitig id, offset) per indexed element is not efficient as it requires a structure of undefined and variable size. This leads to heavy data-structures and cache-misses on construction and query times. To cope with this issue, we chose to index k-mers instead of $(k-1)$-mers. Indeed, each k-mer in a compacted de Bruijn graph of order k, occurs at exactly one couple (unitig id, offset).

Given this indexing scheme in which k-mers are indexed, when querying a $(k-1)$-mer x', the eight possible k-mers containing this $(k-1)$-mer (four k-mers in which x' is the prefix, and four k-mers in which x' is the suffix) are queried. If a match is found, the offset of the $(k-1)$-mer is deduced depending on the case (either x' is the prefix or the suffix of a queried k-mer for which a match is found).

As a matter of fact, we only index each k-mer in its canonical form. Then, a queried k-mer is searched in its canonical form.

Partitioning the k-mers of the graph Conceptually, we could use any associative table such as a hash table for mapping each k-mer of G to its couple (unitig id, offset). However, this would require explicitly storing the k-mers which is a waste of space as k-mers are already explicitly existing in unitigs. Alternatively, we use an MPHF f from the k-mers of the graph. Doing so, we need only to store the position of each indexed k-mer. Formally the position of a k-mer x is defined by $p_x =< u_{id}, u_{off}, orientation >$, where u_{id} is the identifier of the unitig u, $0 \leq u_{off} \leq |u| - k$ is the offset of x in u, and $orientation$ is a boolean variable that is true if x is in its canonical form in u, else it is false. The positions of the k-mers in the graph are stored in a vector V. Given a k-mer x, its position is $p_x = V[j]$ where $j = f(canonical(x))$. There are two observations to be made here:

- At query time, f can give valid hash values for alien k-mers, which are k-mers that are not present in the graph. To handle this, we compare the queried

k-mer to the actual k-mer in the graph, whose position is retrieved thanks to $V[f(canonical(x))]$.
- Adding new k-mers requires to recompute f.

This last point is problematic, since for every addition operation, f must be recomputed, which is a linear-time operation in terms of the number of indexed k-mers. To resolve this, we divide the set of k-mers in the graph into multiple subsets called "buckets". Each subset is indexed using its own MPHF. The key idea being that while adding sequences to a graph, only a subset of the buckets are modified, and so only a subset of the MPHFs have to be recomputed. At query time, the bucket of the queried k-mer x is retrieved, and the corresponding MPHF provides the position of x in the graph.

Formally, we define $\{b_0, b_1, ..., b_{n-1}\}$ buckets. For each of these buckets b_i, an MPHF f_i is computed on its k-mers. MPHFs are computed using `PHOBIC` [10] as it provides the fastest lookup compared to the state of the art tools that compute MPHFs.

The k-mers in the graph are separated into buckets based on their minimizers. The k-mers sharing the same minimizer cannot be distributed into different buckets. However, this strategy may result in a well-known problem of non-uniform distribution of the k-mers in the buckets [5]. Some buckets could be orders of magnitudes larger than some others. Also, the small buckets are problematic for the construction of an MPHF using `PHOBIC`, as higher number of $bits/\text{k}{-}mer$ is required for small buckets (see Figure 3).

All in all, we propose a strategy so that all the batches contain a minimum number of k-mers.

- The number of k-mers sharing the same minimizer should be at least equal to a parameter ρ for creating a bucket. Note that the size of a bucket does not have an upper-bound.
- For the remaining k-mers, we process them by groups of k-mers where the k-mers within a group share the same minimizer. From these groups, we create what so-called "super-buckets" which are buckets containing k-mers that share different minimizers. We start with an empty super-bucket S_0 to which we add the groups of k-mers one by one. Once the number of k-mers added to S_0 exceeds $\gamma \times \rho$ k-mers (with γ a user defined multiplicative factor), we create a new super-bucket. We keep creating and filling super-buckets until all groups of k-mers are processed. The rationale behind this strategy is to achieve a balanced distribution of k-mers on the super-buckets. It is important to note that the size of a super-bucket has an upper bound, which we address in section 2.5.

Finally the data-structure is composed of the following components, represented in Figure 4:

1. A hash table T that maps each minimizer to its bucket identifier. The hash table will be used to identify the bucket that may contain a given k-mer x. The identifier of the bucket is then $b_i = T[minimizer(x)]$ where $minimizer(x)$ is the minimizer of x.
2. A hash table F that maps each bucket identifier to its MPHFs. Hence, $F[b_i]$ is the MPHF computed from the set of k-mers in bucket b_i.

3. A hash table U that maps the identifier of each unitig of the graph to its sequence. Hence, $U[u_i]$ is the unitig sequence whose identifier is u_i.

4. The positions of the k-mers in the graph are stored in a 2-D vector P. $P[b_i]$ is the vector of positions for the k-mers in bucket b_i. $P[b_i][F[b_i](x)]$ is the tuple position $< u_{id}, u_{off}, orientation >$ for the k-mer x in the bucket b_i.

Overall given the position of a k-mer x, x can be retrieved by retrieving the unitig $u = U[u_{id}]$ from the hash table U, hence $x = u(u_{off}, u_{off} + k)$ (Figure 4.c). Note that if the minimizer of x is not present in T, then x does not belong to the graph (Figure 4.b).

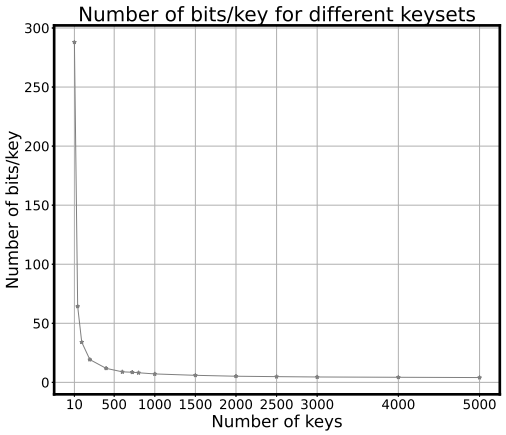

Figure 3. The number of bits/key required for building a MPHF with `PHOBIC`. MPHFs from different sets of keys of sizes ranging from 10 to 5000 random 64-bit keys were computed by `PHOBIC`, and bits/key were then measured.

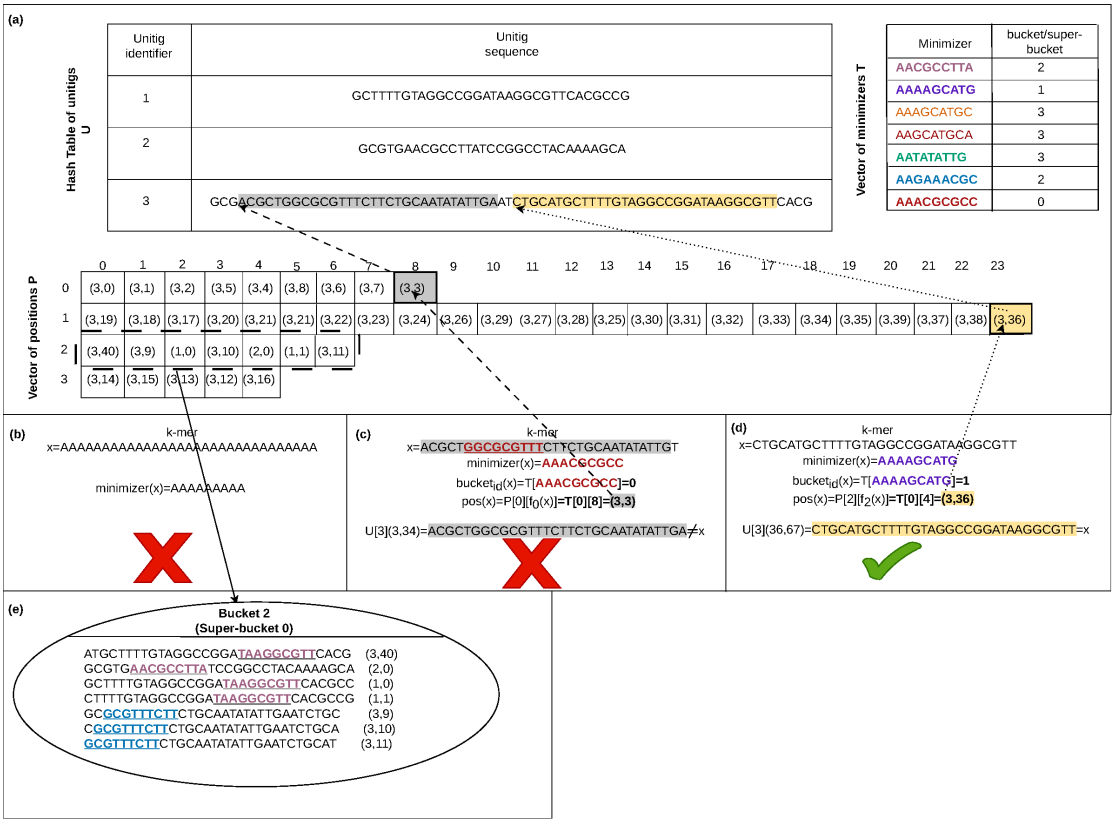

Figure 4. Overview of the data structure. **(a)** the hash table U of the unitigs of a compacted de Bruijn graph with $k = 31$; the hash table T of minimizers that maps a minimizer to its corresponding bucket or super-bucket; and the vector P of positions of the k-mers as a tuple (u_{id}, u_{off}) where u_{id} is the identifier of the unitig in which the k-mer occur at offset u_{off}. Note that the values on top of the vector P represent the hash value of the k-mers computed by the MPHF of their bucket or super-bucket, and the values to the left of P represent the bucket or super-bucket identifier. **(b)** querying an absent k-mer whose minimizer is not in the index. **(c)** querying an absent k-mer whose minimizer is in the index. **(d)** querying a present k-mer. **(e)** the dashed box is converted back to a super-bucket of the k-mers.

2.4 Computing the future unitigs and updating the graph

Recall that N denotes the set of k-mers to be added to a compacted de Bruijn graph G. In Section 2.2 we proposed an overview of algorithms in which k-mers from N are added one after another to G. In practice, for performance reasons, we first compact k-mers from N into what we call "funitigs" (for *future unitigs*).

The funitigs are not simply the unitigs of N as any $(k-1)$-mer of those funitigs that is already in G must be either a prefix or a suffix of a funitig. Doing so, the funitigs are not split latter when added to the graph. The details about the funitig construction are given in Algorithm 3 of supplementary materials.

Once the funitigs are constructed, each of them is added to the graph one after the other. The rules described section 2.2 for adding a k-mer to G exactly apply for adding a funitig to G. The `Cdbgtricks` tool exactly implements those rules, that we do not recall here.

2.5 Updating the index

`Cdbgtricks` enables to update the index of a compacted de Bruijn graph, after the addition of sequences. The updated index can serve for any future update on the graph and for k-mer queries against the graph.

While adding funitigs to the graph, we remember the identifiers of the modified unitigs and of the modified buckets. After all funitigs are added to the graph, the index of the corresponding unitigs and buckets are updated. The update of the index of the graph is divided into three stages:

1. Splitting a unitig or a joining two unitigs or a unitig with a funitig result in changing unitig identifier(s) and the offsets of some k-mers. When we split a unitig u into two unitigs u_1 and u_2, u_1 get the identifier of u, u_2 gets a new identifier and the offsets of its k-mers get recomputed. When we merge a funitig with one or two unitigs, the resultant sequence gets the identifier of one of these unitigs, and the offsets its k-mers get recomputed.
2. The addition of k-mers to super-buckets may lead to doubling their maximum size ($\gamma \times \rho$). In this case, for each concerned super-bucket, it is divided into two new super-buckets.
3. Recompute the MPHFs of the buckets and super-buckets to which new k-mers were added.

When dividing a super-bucket into two super-buckets (case 2), the objective is to balance the size of the two created super-buckets. To address case 2, a straightforward greedy strategy is employed to split the super-bucket into two smaller ones. We propose a simple greedy algorithm (see Algorithm 1) for performing this task.

Algorithm 1 Divide a super-bucket into two super-buckets

Require: A super-bucket of k-mers B
Ensure: Two balanced super-buckets B_1 and B_2
  Initialize four empty sets, B_1, B_2, M_1, M_2
  for each k-mer $x \in B$ **do**
    $m \leftarrow minimizer(x)$
    if $m \in M_1$ **then**
      $B_1 \leftarrow B_1 \cup \{x\}$
    else if $m \in M_2$ **then**
      $B_2 \leftarrow B_2 \cup \{x\}$
    else if $|B_1| < |B_2|$ **then**
      $B_1 \leftarrow B_1 \cup \{x\}$
      $M_1 \leftarrow M_1 \cup \{m\}$
    else
      $B_2 \leftarrow B_2 \cup \{x\}$
      $M_2 \leftarrow M_2 \cup \{m\}$

2.6 Read querying

A compacted de Bruijn graph constructed using `Cdbgtricks` supports sequence queries. In practice, while querying a sequence s on a graph G, `Cdbgtricks` determines if at least α% of the k-mers of s are in the graph (with α a user-defined parameter). If this is the case, `Cdbgtricks` indicates the uni-MEMs, as defined in `deBGA` [17]. Each uni-MEM is a tuple $< u_{id}, u_{start}, u_{end}, s_{start}, s_{end} >$ where u_{start} and u_{end} are the start and end positions of mapping on the unitig whose

identifier is u_{id}, and s_{start} and s_{end} are the start and end positions of mapping k-mers of the queried sequence s. In other words, the k-mers whose offsets in the read are between r_{start} and r_{end} are found in the unitig u_{id} between u_{start} and u_{end}. A uni-MEM is found through the extension of the first common k-mer between the read and a unitig. The extension ends in one of the following cases:

1. A mismatch is encountered.
2. Either the end of the read or the end of the unitig is encountered.

3 Results

All presented results are reproducible using command lines and versions of tested tools, that are given in this repository `https://github.com/khodor14/Cdbgtricks_experiments`. The executions were performed on the GenOuest platform on a node with 4×8 cores Xeon E5-2660 2,20 GHz with 128 GB of memory.

3.1 Genome datasets

We tested `Cdbgtricks` in two frameworks, corresponding to two input datasets of distinct size and complexity. The first one, called "*human*" is composed of 100 assembled human genomes that were used in the `GGCAT` experiments [7]. These genomes are available on zenodo (`10.5281/zenodo.7506049,10.5281/zenodo.7506425`). The second set, called "*coli*" is composed of 7055 *E. coli* genomes downloaded from NCBI (`https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/030/`).

While `Cdbgtricks` is capable of initially constructing a compacted de Bruijn graph from scratch, it is worth noting that there are faster alternatives for creating the initial graph. As such, for each dataset, we created an initial graph in fasta format from one genome (chosen as the first in the alphabetic order of the file names) using `Bifrost`. Once created `Cdbgtricks` can be used for indexing this initial graph. Subsequently, for each dataset, we added one by one the remaining genomes.

3.2 Used Parameters

The used parameters are the same for the two datasets. In all experiments we used $k = 31$ and minimizers of size $m = 11$. During the update experiments the tools were executed using 32 threads, while during the query experiments the tools were executed using a single thread.

For `Cdbgtricks`, the parameters controlling the bucket size were set to default. The bucket lower bound size is $\rho = 5000$ and a the super-bucket multiplicative factor γ is set to 4. This setting of parameters means that the size of a super-bucket is approximately 20000 k-mers, and once it reaches 40000 k-mers, it gets divided into two super-buckets. The values of the parameters were chosen to ensure satisfactory results that will be shown in the subsequent sections.

3.3 Percentage of modified buckets

As explained Section 2.3, one of the key ideas in `Cdbgtricks` is to distribute indexed k-mers into multiple buckets, each bucket being indexed with its own MPHF. Doing

so, we expect that, while adding k-mers from novel sequences, k-mers are added to only a fraction of the buckets, and then only, a fraction of the MPHFs have to be recomputed. More precisely, we expect that the percentage of modified buckets, i.e. $100 \times \frac{\text{number of modified buckets}}{\text{total number of buckets}}$ decreases as the number of genomes in the graph increases. Note that here we do not differentiate buckets and super-buckets and we regroup these two notions in the term "buckets".

In this section we test this expectation on the *human* and *E. coli* datasets. The results about the percentage of modified buckets are shown Figure 5. Results show that, as expected, the percentage of modified buckets decreases with respect to the number of genomes. The shaded cluster of points for the *E. coli* dataset shows that in the majority of cases, the percentage of modified buckets and super-buckets is less than 20%. More specifically, results with, say, more than 5000 *E. coli* genomes show that, except for some outliers, less than 10% of the buckets are modified when adding a new genome.

These results validate the chosen default parameters, and they confirm the expectation that lesser and lesser buckets are modified while increasing the number of genomes of the same species in a compacted de Bruijn graph.

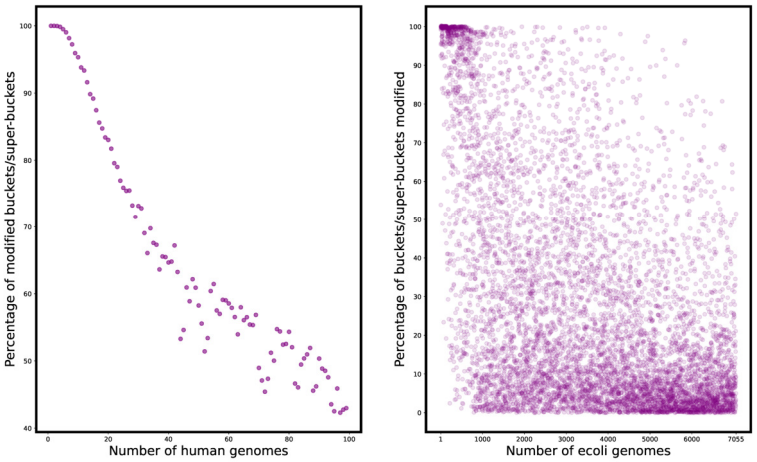

Figure 5. Percentage of modified buckets.

3.4 Scalability

One of the main objective of `Cdbgtricks` is the time performances when updating a compacted de Bruijn graph with new sequences. In this context, we compared the `Cdbgtricks` update time, with the update time obtained thanks to `Bifrost`, also able to update an already created compacted de Bruijn graph. Furthermore, although `GGCAT` does not provide graph updating capabilities, we included it in our comparison due to its efficiency. The memory and disk for the update with `Cdbgtricks` and `Bifrost` and for the construction with `GGCAT` are also reported.

Results for the *human* dataset are shown in Figure 6. Note that, on the *human* dataset, `GGCAT` reached a timeout we set at two days on more than 71 human genomes. Hence, only the results for the first 71 genomes were reported for `GGCAT`. Globally, the results on this dataset show that `Cdbgtricks` is at least 2x faster than `Bifrost` on

graphs composed of 50 genomes or more. Compared to `GGCAT`, `Cdbgtricks` is slower on this small number of genomes. Given that as the number of genomes in the graph increases, the `GGCAT` construction time naturally increases while the `Cdbgtricks` update time decreases, one can expect `Cdbgtricks` to be faster when dealing with more genomes than those tested here. However, given the observe `GGCAT` limitation after 71 genomes, we could not verify this fact in practice, at least for human genomes. The memory used by `Cdbgtricks` and `Bifrost` are slightly the same and are limited to a few dozen gigabytes. `GGCAT` uses much less memory, but needs up to order of magnitude more disk.

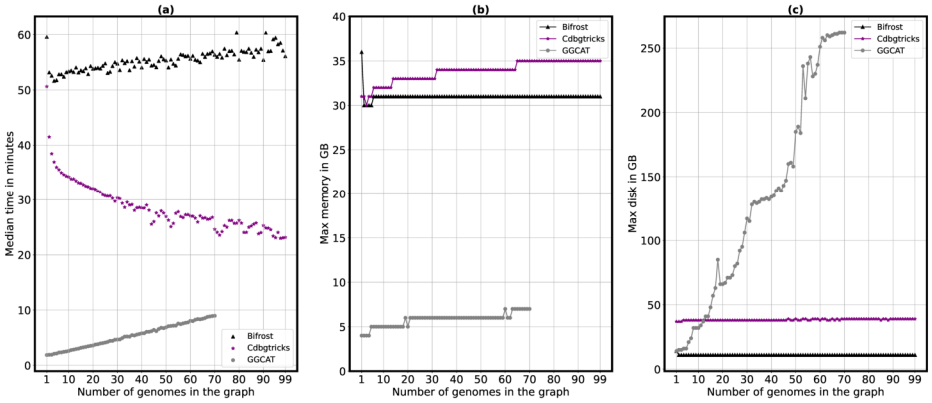

Figure 6. Results on human genomes dataset. Time (a), memory (b) and disk usage (c) are given for updating a graph for for `Cdbgtricks` and `Bifrost` and for constructing a graph from scratch for `GGCAT`.

Results for the *coli* dataset are shown in Figure 7. For the sake of clarity of presenting the results of the *E. coli*, we chose to report the median time over a window of 200 genomes. The detailed presentation of execution time is in supplementary materials. On this dataset, both `Bifrost` and `GGCAT` were faster than `Cdbgtricks` on graphs composed of less than a thousand genomes. However, in the vast majority of cases, when the number of genomes get higher than, say, 2000 genomes, `Cdbgtricks` is 2x to 3x faster than `GGCAT` and `Bifrost`. With `Cdbgtricks`, adding an *E. coli* genome to a compacted de Bruijn graph graph containing already few thousands genomes requires between 30 and 50 seconds. `Cdbgtricks` uses slightly the same amount of memory compared to `GGCAT` and roughly twice the amount of memory compared to `Bifrost`. `Cdbgtricks` uses up to 4x more disk compared to `Bifrost`, while it uses much less disk compared to `GGCAT`.

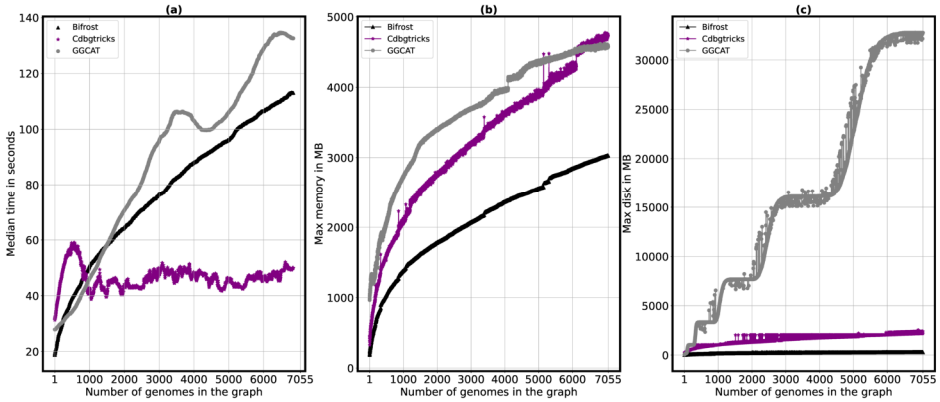

Figure 7. Results on *E. coli* genomes dataset. Time (a), memory (b) and disk usage (c) are given for updating a graph for for `Cdbgtricks` and `Bifrost` and for constructing a graph from scratch for `GGCAT`. The time is given as the median over a window of 200 consecutive points.

3.5 Results querying sequences

We propose some experiments for comparing the query performances of `Cdbgtricks` with those of `Bifrost`, `GGCAT` and `SSHash`. Note that these four tools do not offer the same query features. Although, these results must be considered as rough estimations showing the main tendencies.

Using `Bifrost`, we constructed a graph from 15,806 *E. coli* genomes, and a graph from 10 human genomes. Then we constructed an index for each graph using either `Cdbgtricks`, `Bifrost` or `SSHash`. We have differentiated between results obtained with positive queries (querying sequences present in the graph) and those obtained with negative queries (querying random sequences) with k-mers that are not in the two constructed graphs. The positive queries are a subset of unitigs from each graph. The negative queries are composed of one million random sequences of length between 500 and 1000 base pairs. The querying results are shown Table 1.

Table 1. Performances of sequence queries using a compacted de Bruijn graph for `Cdbgtricks`, `Bifrost`, `SSHash`, and `GGCAT`

Dataset	Query type	Tool	Memory (MB)	Disk (MB)	time (mm:ss)
E. coli	Negative	`Cdbgtricks`	4723	0	10:02
		`Bifrost`	4362	0	06:43
		`SSHash`	725	0	00:07
		`GGCAT`	560	3325	01:32
	Positive	`Cdbgtricks`	4724	0	02:15
		`Bifrost`	4362	0	01:43
		`SSHash`	725	0	01:10
		`GGCAT`	644	2978	01:26
human	Negative	`Cdbgtricks`	25520	0	12:25
		`Bifrost`	27376	0	11:37
		`SSHash`	6090	0	00:07
		`GGCAT`	615	6861	4:55
	Positive	`Cdbgtricks`	25520	0	04:23
		`Bifrost`	27376	0	06:37
		`SSHash`	6090	0	01:14
		`GGCAT`	746	7053	05:04

The results shows that `Cdbgtricks` and `Bifrost` obtained similar results in term of memory, while `Cdbgtricks` is slightly slower. While `GGCAT` is faster than `Bifrost` and `Cdbgtricks`, and it uses the smallest amount of memory in these query experiments, it uses few Gigabytes of disk. Despite `SSHash` being the fastest tool, it consumes an order of magnitude more memory than `GGCAT`. These results shows that `Cdbgtricks` offer queries in a reasonable amount of time, and the performance of `Cdbgtricks` is close to the performance of `Bifrost`.

4 Discussion and future work

In this paper, we presented `Cdbgtricks`, a novel method for updating a compacted de Bruijn graph when adding new sequences such as full genomes. `Cdbgtricks` also indexes the graphs, hence it enables to query sequences and detect the portions of the graph that share k-mers with the query. The dynamicity of the proposed index is achieved thanks to the distribution of k-mers into multiple buckets, each bucket being indexed using a minimal perfect hash function (MPHF). The addition of new k-mers affects only a fraction of the buckets, for which the MPHF has to be recomputed. In practice, when indexing a large number of genomes (dozens of human genomes or thousand of *E. coli* genomes) `Cdbgtricks` outperforms the computation time of state-of-the-art tools dedicated to the creation of the update of compacted de Bruijn graphs.

Of independent interest, exploiting `PTHash`, the `Cdbgtricks` indexing framework offers a theoretical way to bijectively associate each k-mer from a set composed of n distinct k-mers with a unique value in $[0, n[$. In that respect, despite the fact that some engineering work remains to be done to achieve this practical feature, our indexing strategy can be used as an MPHF for this kind of dataset. Furthermore, it will present two main additional advantages when compared to a classical MPHF:

- In essence, an MPHF is static. Adding an element to this kind of data structure requires one to recompute the entire MPHF from scratch to associate the $n + 1$ elements to a unique value in $[0, n + 1[$. As `Cdbgtricks` distributes the k-mer set over numerous "sub-MPHFs", adding an element requires only to recompute one of the sub-MPHFs. This offers a clear advantage when adding a few elements to large MPHFs, composed of, say, billions of elements.
- The MPHF definition does not impose that so-called "alien k-mers" (k-mers not belonging to the indexed set) are detected as aliens at query time. Actually, MPHFs that do not store the indexed set of elements (as this is the case for `BBhash` and `PTHash`), are not able to always discriminate an alien k-mer from an indexed one. In the context of this work, the presence of the indexed k-mers in the stored unitigs enables us to validate that a query k-mer actually belongs to the original set, and thus enables us to detect whether it is an alien k-mer or not.

A future research direction is to devise a smarter bucket clustering approach. One way could be to group the buckets whose minimizers appear in the same unitigs. Doing so, we could expect more data locality, limiting the cache-misses.

The compacted de Bruijn graph computed by `Cdbgtricks` is not colored. This means that the information is lost about the original genome(s) a k-mer belongs to. In recent years, significant attention has been given to the use of colored and compacted de Bruijn graphs in computational biology applications [12]. Hence, another research priorities for the future of this tool is to integrate the color information. This will

necessitate minor yet potentially expensive operations. To include the new colors associated with a set of new sequences S, the color information of all k-mers in S already present in the graph G will have to be updated, while those k-mers are not modified in the current uncolored `Cdbgtricks` version.

There exists no limitation for using `Cdbgtricks` for merging the information of two compacted de Bruijn graphs G_1 and G_2. We can simply consider the k-mers of, say, G_2 to be added into G_1, and apply the exact same algorithm as proposed here. Future work will include validation and scaling tests for this approach.

Finally, we believe that the number of common k-mers but also the number of splits and joins performed when adding a sequence to a graph could be used as metrics to estimate the distance between a sequence and a compacted de Bruijn graph, or even between two compacted de Bruijn graphs.

5 Funding

This project received funding from the European Union's Horizon 2020 research and innovation program 369 under the Marie Skłodowska-Curie grant agreement No 956229.

Acknowledgements

We acknowledge the GenOuest bioinformatics core facility
`https://www.genouest.org` for providing the computing infrastructure.

References

1. J. Alanko, B. Alipanahi, J. Settle, C. Boucher, and T. Gagie: *Buffering updates enables efficient dynamic de bruijn graphs.* Computational and Structural Biotechnology Journal, 19 2021, pp. 4067–4078.
2. B. Alipanahi, A. Kuhnle, S. J. Puglisi, L. Salmela, and C. Boucher: *Succinct dynamic de Bruijn graphs.* Bioinformatics, 37(14) 07 2021, pp. 1946–1952.
3. F. Almodaresi, H. Sarkar, A. Srivastava, and R. Patro: *A space and time-efficient index for the compacted colored de Bruijn graph.* Bioinformatics, 34(13) 06 2018, pp. i169–i177.
4. F. Almodaresi, M. Zakeri, and R. Patro: *PuffAligner: a fast, efficient and accurate aligner based on the Pufferfish index.* Bioinformatics, 37(22) 06 2021, pp. 4048–4055.
5. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev: *On the representation of de bruijn graphs.* Journal of Computational Biology, 22(5) 2015, pp. 336–352, PMID: 25629448.
6. R. Chikhi, A. Limasset, and P. Medvedev: *Compacting de bruijn graphs from sequencing data quickly and in low memory.* Bioinformatics, 32 06 2016, pp. i201–i208.
7. A. Cracco and A. Tomescu: *Extremely-fast construction and querying of compacted and colored de Bruijn graphs with GGCAT.* bioRxiv, 2022.
8. V. G. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie: *Practical dynamic de Bruijn graphs.* Bioinformatics, 34(24) 06 2018, pp. 4189–4195.
9. H. Guo, Y. Fu, Y. Gao, J. Li, Y. Wang, and B. Liu: *degsm: Memory scalable construction of large scale de bruijn graph.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 18(6) 2021, pp. 2157–2166.
10. S. Hermann, H.-P. Lehmann, G. E. Pibiri, P. Sanders, and S. Walzer: *Phobic: Perfect hashing with optimized bucket sizes and interleaved coding.* arXiv, 2024.
11. G. Holley and P. Melsted: *Bifrost - Highly parallel construction and indexing of colored and compacted de Bruijn graphs.* bioRxiv, 2019.

12. S. T. Horsfield, N. J. Croucher, and J. A. Lees: *Accurate and fast graph-based pangenome annotation and clustering with ggcaller*. Genome Research, 2023.
13. B. Hou, R. Wang, and J. Chen: *Long read error correction algorithm based on the de bruijn graph for the third-generation sequencing*, in 2021 4th International Conference on Information Communication and Signal Processing (ICICSP), 2021, pp. 616–620.
14. J. Khan, M. Kokot, S. Deorowicz, and R. Patro: *Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2*. Genome biology, 23(1) 2022, p. 190.
15. T. Lemane, P. Medvedev, R. Chikhi, and P. Peterlongo: *kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections*. Bioinformatics Advances, 2(1) 04 2022.
16. A. Limasset, J.-F. Flot, and P. Peterlongo: *Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs*. Bioinformatics, 36(5) 02 2019, pp. 1374–1381.
17. B. Liu, H. Guo, M. Brudno, and Y. Wang: *deBGA: read alignment with de Bruijn graph-based seed and extension*. Bioinformatics, 32(21) 07 2016, pp. 3224–3232.
18. C. Marchet, Z. Iqbal, D. Gautheret, M. Salson, and R. Chikhi: *REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets*. Bioinformatics, 36(Supplement_1) 07 2020, pp. i177–i185.
19. C. Marchet, M. Kerbiriou, and A. Limasset: *BLight: efficient exact associative structure for k-mers*. Bioinformatics, 37(18) 04 2021, pp. 2858–2865.
20. J. R. Miller, S. Koren, and G. Sutton: *Assembly algorithms for next-generation sequencing data*. Genomics, 95(6) 2010, pp. 315–327.
21. I. Minkin, S. Pham, and P. Medvedev: *TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes*. Bioinformatics, 33(24) 09 2016, pp. 4024–4032.
22. G. E. Pibiri: *Sparse and skew hashing of K-mers*. Bioinformatics, 38(Supplement_1) 06 2022, pp. i185–i194.
23. M. Schatz, A. Delcher, and S. Salzberg: *Assembly of large genomes using second-generation sequencing*. Genome research, 20 09 2010, pp. 1165–73.
24. S. Sheikhizadeh, M. E. Schranz, M. Akdel, D. de Ridder, and S. Smit: *PanTools: representation, storage and exploration of pan-genomic data*. Bioinformatics, 32(17) 08 2016, pp. i487–i493.

Supplementary Materials

Figures

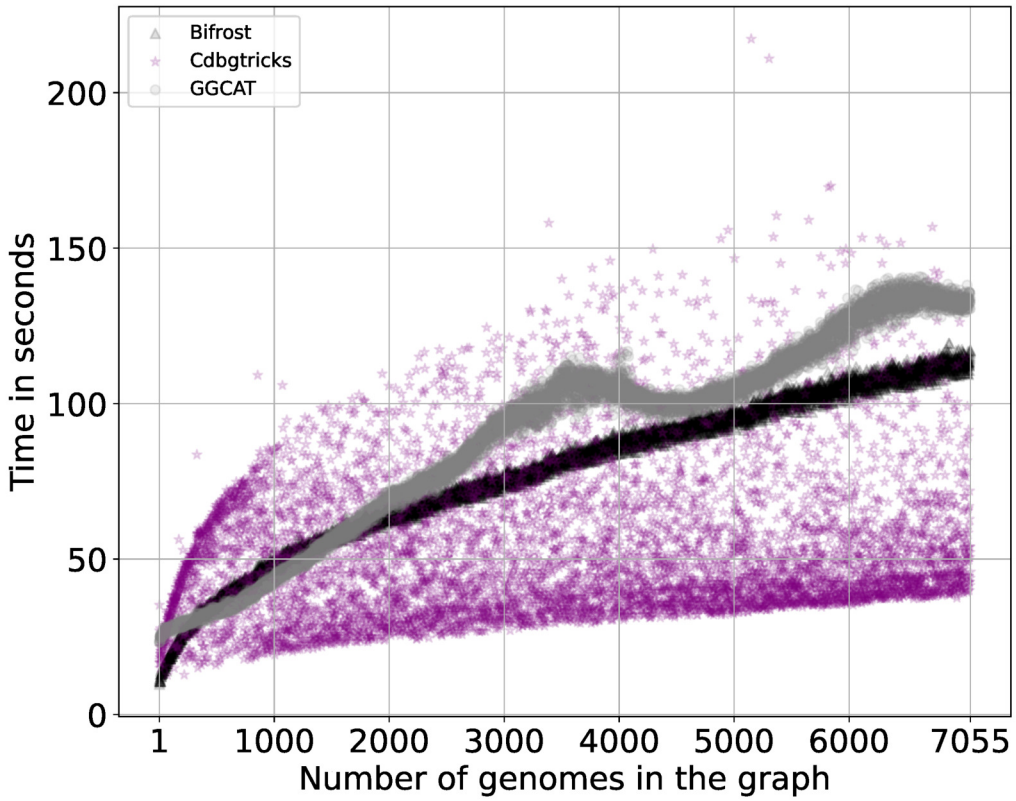

Figure 8. Results on *E. coli* genomes dataset. Time is given for updating a graph for for `Cdbgtricks` and `Bifrost` and for constructing a graph from scratch for `GGCAT`.

Algorithms

Algorithm 2 Test k-mer presence

Require: Hash Table of unitigs U, k-mer s, Table of buckets T, Table of mphfs F
Ensure: s belongs to a unitig

function KmerPresence(H,s,T,F)
- $mini_s \leftarrow canonicalminimizer(s)$ ▷ compute the canonical minimizer of s
- $i \leftarrow T[mini_s]$ ▷ retrieve the identifier of the bucket of $mini_s$
- **if** $i \neq NIL$ **then** ▷ the minimizer is in the index
  - $f_i \leftarrow F[i]$ ▷ retrieve the mphf of the bucket
  - $q \leftarrow canonical(s)$ ▷ compute the canonical form of s
  - $j \leftarrow f_i(q)$ ▷ compute the hash value of q
  - $p_j \leftarrow P[i][j]$ ▷ retrieve the position tuple of q
  - $id \leftarrow p_j.u_{id}$ ▷ retrieve the unitig identifier
  - $o \leftarrow p_j.u_{off}$ ▷ retrieve the offset of q in this unitig
  - $u \leftarrow U[id]$ ▷ retrieve the unitig
  - return $canonical(u(o, o+k)) = q$ ▷ compare the k-mers
- return false ▷ the minimizer is not in the index, so s is not in the graph

Algorithm 3 Construct funitigs

Require: Hash table of new k-mers K,Index of the graph I
Ensure: A set of funitigs X

```
function ConstructFunitig(K)
    X ← ∅
    for each k-mer s ∈ K do
        if s is not marked as used then
            mark s as used
            s_1 ← ExtendRight(s)
            s_2 ← ExtendRight(s̄)
            X ← X ∪ {s̄_2 ⊙ s_1(k − 1, |s_1|)}
    return X
function ExtendRight(s)
    q ← s
    g ← s(1, k)                                   ▷ the (k-1)-suffix of s
    while True do
        a ← FindRightExtensions(g)
        if a ≠ ϵ then
            b ← FindRightExtensions(ḡ)
            if b ≠ ϵ then
                q ← q ⊙ a
                g ← g(1, k − 1) ⊙ a
            else
                break
        else
            break
    return q
function FindRightExtensions(x)                   ▷ x is a (k-1)-mer
    extension ← ϵ
    count ← 0
    for each a ∈ ∑ do
        q ← x ⊙ a
        if q ∈ K then
            extensions ← a
            count ← count + 1
        else if KMERPRESENCE(I.U,s,I.T,I.F) then  ▷ x is found in the graph
            extension ← ϵ
            break
    if count ≠ 1 then
        extension ← ϵ
    return extension
```

Author Index

Proceedings of the Prague Stringology Conference 2024
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 978-80-01-07328-5

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