Proceedings of the Prague Stringology Conference 2025

Edited by Jan Holub and Jan Žďárek

August 2025

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

ISBN 978-80-01-07461-9

Preface

The proceedings in your hands contain a collection of papers presented in the Prague Stringology Conference 2025 (PSC 2025) held on August 25–26, 2025 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 five papers in this proceedings made the cut and were selected for regular presentation at the conference.

The PSC 2025 was organized in both present and remote form. Speakers we required to present their papers in person. Non-speakers could decide whether to arrive to Prague or to participate remotely.

The Prague Stringology Conference has a long tradition. PSC 2025 is the twenty-eight 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, 2023–24 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 2025 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 2025. 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 2025

Jan Holub and Robert Mercas

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Ferdinando Cicalese	(University of Verona, Italy)
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	(Tokyo Medical and Dental University, Japan)
Thierry Lecroq	(Université de Rouen, France)
Robert Mercas *Co-chair*	(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 Bohuslavová	Josef Erik Sedláček	Jan Trávníček
Jan Holub, *Chair*	Regina Šmídová	Jan Žďárek
Tomáš Pecka		

External Referees

Arnaud Lefebvre	Estéban Gabory	Felipe A. Louza

Table of Contents

Invited Talk

Contributed Talks

On Periodicities in Strings
(*Abstract*)

Frantisek Franek

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

Periodicities in strings, in particular tandem repetitions, have been of interest to researchers from the beginning. The pioneering work of Corchemore in 1981 showed that the optimal bound for the number of maximal repetitions in a string of length n is of $O(n\log(n))$ complexity and attained by Fibonacci strings, followed closely by the seminal work of Apostolico and Preparat. In 1989, Main introduced an $O(n\log(n))$ algorithm for detection of maximal repetitions where the $\log(n)$ factor represented the size of the alphabet, so for a constant size alphabet, it was a linear algorithm and a linear number of repetitions. From these beginnings, two subsequent lines of research crystallized over several years. The first line of research, dealing with the generalization of maximal repetitions in the form of runs, focused on determining and computing the maximum number of runs, the second line of research focused on determining the maximum number of distinct squares in a string. These two lines of research culminated in respective conjectures: the ***runs conjecture:*** *the maximum number of runs in string is bounded by the length of the string*, and the ***distinct squares conjecture:****the maximum number of distinct squares in string is bounded by the length of the string.*

Intense research on these problems were initiated by the pioneering work of Kolpakov & Kutcherov (1999) for runs and by Fraenkel & Simpson (1998) for distinct squares. Before both conjectures were settled, Deza, myself, and our graduate students Jiang and Baker strengthen both conjecture to hypothesize the bound to be the length of the string less the size of the alphabet of the string. The bounds $n-d$ were based on the d-step approach where the d stands for the size of the alphabet and n the length of the string.

The runs conjecture was settled by Banai et al. in 2015 (published in 2017). The d-step conjecture for runs was proven by Deza et al. in 2017. The d-step conjecture for distinct squares (and hence the distinct squares conjecture) was recently proven by Brlek & Li. The Brlek & Li approach shows why the size of the alphabet naturally occurs in the bound. The d-step aspects of both problems investigated by Deza et al. have some additional consequences beyond the size of the bound, and in both cases they are shown to be optimal as strings of length n with $n-d$ runs or distinct squares are shown.

Acknowledgements
This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant program number RGPIN-5504-2018.

Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9

Best Practices in Adaptive Encoding

Igor Zavadskyi and Maksym Kovalchuk

Taras Shevchenko National University of Kyiv
2d Glushkova ave.
Kyiv, Ukraine
ihorzavadskyi@knu.ua, max.koval4uk@knu.ua

Abstract. We discuss and experimentally evaluate a range of classical and recently developed adaptive data compression algorithms. Key features of each method are highlighted, and their effectiveness in real-world data compression scenarios is assessed. By combining these techniques, we construct a family of simple adaptive encoding methods that achieve an excellent space-time trade-off, particularly for text with small alphabets such as ASCII. Compared to classical Vitter's algorithm, our method is several times faster while maintaining a similar compression ratio. Compared to Gagie's worst-case optimal solution, our algorithm generates compressed files that are about 10% smaller, while also delivering faster encoding and decoding speeds.

1 Introduction and Related Work

Adaptive encoding is a well-established data compression technique that enables "online" compression, meaning the encoder does not require prior knowledge of the entire file's statistical characteristics. Instead, it infers this information from the portion of the file it has already encoded and uses it to estimate the properties of the remaining data. Compared to non-adaptive methods, this approach allows the encoder to make a single pass through the text, and often leads to more accurate probability estimates for the symbols. Classical statistical compression algorithms using adaptive techniques were introduced by Faller [1], Gallager [5], Knuth [7] (FGK), and Vitter [11].

Both the FGK and Vitter algorithms adapt the Huffman tree dynamically, updating it after processing each symbol. The FGK method has been shown in [9] to use, in the worst case, $\delta + 2$ bits per symbol more than the zero-order entropy H, where δ represents the overhead of static Huffman coding over the entropy. Vitter's more advanced algorithm reduces this overhead to $\delta + 1$ bits. In practice, however, both the FGK and Vitter methods often outperform the theoretical entropy limit H. In [2], Gagie revised the FGK algorithm to implement adaptive Shannon encoding rather than Huffman coding, which eliminated the δ term and reduced the worst-case space usage to $H + 1$ bits per symbol. This estimation is near-optimal as in [4] Gagie and Nekrich proved the lower bound for the worst-case space, $H + 1 - o(1)$ bits per symbol.

Updating the Huffman tree takes $O(H)$ time per symbol for both encoding and decoding. However, it is not likely that its structure will change with every next symbol, especially if the alphabet is small. To improve time efficiency, the natural idea is to update the code over intervals. With the appropriate relation between the interval length, the length of a text, and the alphabet size, the code may remain space-optimal. This idea was first implemented by Karpinsky and Nekrich [6]. The authors introduced an advanced algorithm using the canonical Shannon code with quantized code updates. Although each update still takes $O(l_{max})$ time (where l_{max} is

Igor Zavadskyi, Maksym Kovalchuk: Best Practices in Adaptive Encoding, pp. 3–12.
Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9

the longest codeword length), updates are spaced, distributing the cost over chunks of symbols and achieving $O(1)$ amortized time per symbol. In addition, their method reduces the decoding time to $O(\log H)$ per symbol. The worst-case space complexity is guaranteed to be $H + 1$ bits per symbol as long as $\sigma = o(n/\log^2 n)$, where σ is the alphabet size, and n is the total length of the text.

Shannon's prefix-free code is convenient for estimating worst-case space, as it guarantees that a symbol having probability p is assigned to a codeword of length at most $\lceil \log 1/p \rceil$ bits (hereinafter log denotes the binary logarithm). In [3], Gagie proposed an algorithm based on Shannon code that maintains a space usage of $H + 1$ bits per symbol and supports constant-time encoding and decoding for the alphabets consisting of $o(\sqrt{n}/\log n)$ symbols. Similarly to [6], Gagie's method performs code updates periodically over intervals of length $\lceil \sigma \log n \rceil$. Before every code update, character probabilities are *smoothed*, i.e., replaced with the weighted average of the character's probability with weight $(\log n - 1)/\log n$, and uniform distribution with weight $1/\log n$. This technique restricts the maximal codeword length, reducing the size of a decoding lookup table and its construction time. Although Gagie's algorithm [3] is the first worst-case optimal method in terms of space and time complexity that can be easily implemented in practice, it was published as a theoretical contribution only. Its experimental testing is one of the goals of our research.

Yet another approach to accelerate adaptive encoding and decoding was proposed in [12]. The core idea is to decouple the update of the codeword set from the update of the mapping between alphabet symbols and codewords. For Huffman codes, updating the codeword set takes $O(H)$ time per symbol, while updating the symbol-to-codeword correspondence can be done in constant time by simply swapping two entries in the (symbol, codeword) map, which is ordered by symbol frequencies. In this method, an $O(1)$-time swap is performed after processing each symbol, whereas the full codeword set is updated at geometrically increasing intervals.

This approach is particularly effective for large alphabets, such as word-based representations of natural language texts. In such cases, the data typically contains many symbols with small frequencies, e.g. 1 or 2, causing frequent changes in the relative order of symbols by frequency. However, the set of codewords itself changes much less often. As a result, the algorithm [12] achieves significant speedups − by orders of magnitude over Vitter's algorithm − at the cost of only about a 1% reduction in compression efficiency.

In this paper, we focus on the compression of text over small, e.g. character-based, alphabets. In this case, neither the frequent swapping of the (symbol, codeword) map elements nor the symbol-after-symbol updating of the codeword set is necessary. In such scenarios, the frequency order of characters changes less frequently, making the method [12] less time-efficient. Nevertheless, the key idea of updating the codeword set at geometrically increasing intervals proves valuable even for small alphabets, as shown by the experiments in Section 3. This strategy results in only a few code updates per file.

We refer to algorithms that require a small number of code updates as *low-adaptive*. The concept of low adaptivity, along with other ideas discussed above, can be explored in various combinations to further improve the trade-off between compression efficiency and encoding/decoding speed. Namely,

- update the code over fixed-length or variable-length intervals;
- use smoothed or non-smoothed symbol probabilities;

– use Shannon or Huffman codes, canonical or non-canonical.

All these options can be implemented on the basis of a very simple lookup table-based approach:

– during the encoding, get codewords from the lookup table indexed by characters;
– during the decoding, use several bits from the encoded bitstream as indices of lookup tables allowing us to get the output character and the bit length of the decoded codeword; shift the bitstream read position by this length;
– update the codeword set together with the lookup tables over some intervals.

In Section 2, we implement this baseline approach in encoding and decoding algorithms featuring some memory tricks that reduce the number of time-consuming bit-level operations. In Section 3, we discuss the results of experiments introducing the above-listed techniques into the baseline algorithms. Gagie's original algorithm is one of the options in this more general schema. Conclusions on the practical applicability of different approaches to adaptive encoding are given in Section 4.

2 Encoding and Decoding Algorithms

The baseline low-adaptive encoding approach is implemented in Algorithm 1. The code itself, as well as the correspondence between characters and codewords, are updated in variable-length intervals (line 20). In homogeneous files, it is reasonable to increase the interval length geometrically with some constant *rate* (line 21). A substring between two code update points is processed in lines $7-19$. If we use fixed-length intervals between code updates, set $rate = 1$.

We store two precomputed lookup tables indexed by characters: $\mathsf{Codewords}[c]$ consists of the codeword corresponding to the character c, while $\mathsf{Lengths}[c]$ stores its bitlength. In lines $8-10$ we read a character, increase its frequency, and move the read position forward. In lines $11-18$, the codeword of the character c is appended to the output bitstream. To accelerate the bit-wise operations, we output 32-bit words, which are constructed inside the 64-bit variable *buffer*, starting from its leftmost bit. Each codeword is shifted by *shift* bits to the left to find its appropriate bit position in the buffer. The variable *shift* initially equals 64 and at each iteration is decreased by the length of a codeword (line 11). A codeword is appended to the buffer in line 12. When the left half of the buffer is full (line 13), we output it (lines $14-15$) and shift the buffer by 32 bits to the left (lines $16-17$). It is assumed that there are no codewords longer than 32 bits, which is realistic for character-based compression.

In the reverse decoding Algorithm 2, the code is updated in the same intervals as during the encoding (lines $20-21$). To reduce the number of time-consuming bit-level operations, we use the 64-bit *buffer*, which is initialized with the first 8 bytes of the code in line 3. The current codeword is aligned with the leftmost bit of the buffer. We decode it and get its length using the function DecodeCodeword in line 8, which can be implemented differently depending on the underlying code. In lines $9-11$, we output the decoded character and increase its frequency. In line 12, we shift the buffer by the length of the codeword to the left to align the next codeword with the leftmost bit of the buffer. In line 13, we update the total shift length in the variable *shift*. When it exceeds 32 bits, we insert the next 32-bit word to the buffer and decrease the shift value by 32 (lines $14-18$).

Algorithm 1: Adaptive encoding

```
input  : - String Text[1..n];
         - Position of the first code update initialIntervalLength;
         - Ratio of interval length between code updates rate;
         - Precomputed codewords array Codewords;
         - Precomputed codeword length array Lengths.
output: The bitstream of codewords Out, indexed by bytes.
1  shift ← 64;
2  buffer ← 0;
3  foreach c ∈ A do Freq[c] ← 0;
4  intervalLength ← initialIntervalLength;
5  inPos ← 1; outPos ← 1;
6  while inPos < n do
7      for i ← 1 to intervalLength do      // Process block of characters
8          c ← Text[inPos];                          // Read the character
9          Freq[c] ← Freq[c] + 1;                    // Increase frequency
10         inPos ← inPos + 1;                  // Shift the read position
11         shift ← shift − Lengths[c];        // Fill the buffer from left
12         buffer ← buffer + (Codewords[c] << shift);
13         if shift ≤ 32 then            // Output 32 bits from the buffer
14             Out[outPos..outPos + 3] ← buffer >> 32;
15             outPos ← outPos + 4;
16             buffer ← buffer << 32;
17             shift ← shift + 32;
18         end
19     end
20     UpdateCode(Freq);          // Update the code and interval length
21     intervalLength ← min{n − inPos, intervalLength · rate};
22 end
```

A possible smoothing of character probabilities before the code update does not affect encoding/decoding algorithms as it is encapsulated in the function UpdateCode.

The decoding of a noncanonical codeword is simple and shown in Algorithm 3. In line 1, we get the $maxLen$-bit value, which consists of the current codeword and is used as the index of lookup tables to get the decoded character in line 2 and the codeword length in line 3.

This approach works well unless $maxLen$ exceeds $12-13$ bits. The tables Decode and DecodeLen then become too large for the cache memory, which dramatically slows down decoding. To address this issue, one can use the *Canonical Huffman Codes* (CHC), first introduced in [10]. For a given source symbol distribution, codewords of CHC have the same lengths as the classical Huffman codes, and thus, the compression ratios of these code classes are the same. In addition, the adaptive encoding and general decoding algorithms are the same as for non-canonical codes (Algorithms 1 and 2); the only difference is the content of the lookup tables, which are filled in the function UpdateCode in line 20. However, the DecodeCodeword function is quite different and is shown in Algorithm 4.

Consider a list of codewords sorted by increasing length, which corresponds to a list of characters sorted by decreasing frequency. To decode a codeword, find its index

ind in this list. The decoded character is then given by Dict[*ind*], where Dict is the dictionary.

Algorithm 2: Adaptive decoding

input : - Encoded bitstream Code, indexed by bytes;
- Decoded string length n;
- Position of the first code update $initialIntervalLength$;
- Ratio of interval length between code updates $rate$.

output: Decoded string Text.

```
1  shift ← 0;
2  foreach c ∈ A do Freq[c] ← 0;
3  buffer ← Code[1..8];
4  intervalLength ← initialIntervalLength;
5  inPos ← 5; outPos ← 1;
6  while outPos < n do
7      for i ← 1 to intervalLength do
8          (c, length) ← DecodeCodeword(buffer);
9          Text[outPos] ← c;                          // Output the character
10         outPos ← outPos + 1;                  // Shift the output position
11         Freq[c] ← Freq[c] + 1;                      // Increase frequency
12         buffer ← buffer << length;          // Remove codeword from buffer
13         shift ← shift + length;
14         if shift ≥ 32 then              // Insert 32-bit word to the buffer
15             shift ← shift − 32;
16             inPos ← inPos + 4;
17             buffer ← buffer + (Code[inPos..inPos + 3] << shift);
18         end
19     end
20     UpdateCode(Freq);              // Update the code and interval length
21     intervalLength ← min{n − outPos, intervalLength · rate};
22 end
```

Algorithm 3: Function DecodeCodeword – Decoding the current codeword and obtaining its length for a noncanonical code.

input : - Maximal codeword length $maxLen$;
- Precomputed decoding table Decode;
- Precomputed array of codeword lengths DecodeLen;
- 64-bit chunk of the input stream $buffer$.

output: Decoded character c and the $length$ of its codeword.

```
1  x ← buffer >> (64 − maxLen);                  // Get the maxLen-bit value
2  c ← Decode[x];
3  length ← DecodeLen[x];
4  Return c, length.
```

The most important property of canonical codes is that codewords of the same length form a contiguous set of integers. Let baseCwd[l] be the smallest value of the codeword of length l, and baseSym[l] be the result of its decoding. Then, if we know that the buffer bit-vector starts from the codeword cwd of length l, the result of its decoding can be calculated as baseSym[l] + cwd − baseCwd[l] (line 6). The codeword cwd can be obtained by shifting the 64-bit buffer by $64 - l$ bits to the right (line 5).

The only problem that remains is to get the length of the codeword given the buffer vector. In the original canonical codes, the desired length is searched linearly, i.e., the length l is incremented, starting from the minimum possible value, until the buffer value is greater or equal to the threshold $\mathsf{Limit}[l]$ – this is the smallest buffer value, which bit representation starts from the smallest codeword of length l. This search can be faster than traversing the Huffman tree; however, it remains too slow. In [8], Liddell and Moffat propose to get the initial length value from the lookup table Start of limited size (line 1). They use p leftmost bits from the buffer as the index x of that table (values p in the range $8 - 10$ are a good choice). $\mathsf{Start}[x]$ is the length of the shortest codeword *cwd*, which is consistent with x. The word 'consistent' here means that either some prefix of x coincides with *cwd* or some prefix of *cwd* coincides with x. Then, the obtained length value l is incremented until the buffer value is not less than the threshold $\mathsf{Limit}[l]$ (lines $2 - 4$).

As a result, we avoid using lookup tables with more than 2^p elements versus 2^{maxLen} elements in noncanonical codes.

Algorithm 4: Function DecodeCodeword – Decoding the current codeword and obtaining its length for canonical Huffman codes.

input : - 64-bit chunk of the input stream *buffer*;
- Lookup table for the initial codeword length values Start;
- Bit length p of the lookup table Start index;
- Threshold for buffer values Limit;
- The smallest value of a codeword of a given length baseCwd;
- The result of the decoding of the smallest codeword of a given length baseSym;
- Dictionary Dict.

output: Decoded character c and the length l of its codeword.

```
1 l ← Start[buffer >> (64 − p)];                  // Get the initial length
2 while buffer ≥ Limit[l] do                      // Find the true length
3 |  l ← l + 1;
4 end
5 cwd ← buffer >> (64 − l);          // Get l-bit codeword from the buffer
6 c ← Dict[baseSym[l] + cwd − baseCwd[l]];          // Decode the codeword
7 Return c, l.
```

3 Experiments

Compression experiments were conducted on 4 texts of different sizes and nature.

1. *Harry Potter, vol. 1.* 439,741 bytes. $H_0 = 251,221$ bytes = 4.57 bits/character.
2. *The Bible, King James version.* 4,047,392 bytes. $H_0 = 2,197,350$ bytes = 4.343 bits/character.
3. *The Bible, King James version*, preprocessed with Burrows-Wheeler + Move-To-Front transforms. 4,047,392 bytes. $H_0 = 1,550,143$ bytes = 3.064 bits/character.
4. *Source program code* from Pizza&Chili corpus. 20,055,515 bytes. $H_0 = 13,915,769$ bytes = 5.551 bits/character.

Table 1 presents the compressed file sizes in bits per character with the percentage of excess over the entropy given in brackets. In each pair of lines in Table 2, the encoding (upper) and decoding (lower) times are shown. They are averaged over 100

runs of each algorithm on a PC with an AMD Athlon 3000G processor, 3.50 GHz, 2 cores, 32 KB L1 data cache per core, L2 cache – 512 KB per core, L3 cache – 4 MB, 16 GB RAM. Algorithms implemented in C++ and compiled using the g++ compiler with full optimization for speed.[1]

Let us compare the results in different dimensions.

- **Code type.** Vitter's algorithm updates the code after processing each character and consistently produces the smallest compressed files. However, our low-adaptive Huffman code-based method performs nearly as well − less efficient by a fraction of a percent − and even surpasses Vitter's algorithm in one case (Text 3). In terms of speed, Vitter's approach is always several times slower than the low-adaptive encoding with variable-length update intervals, regardless of whether they use Shannon or Huffman codes. All adaptive methods based on Shannon codes exhibit significantly lower compression efficiency, producing files approximately 10 % or more above the entropy, compared to about 1 % overhead for Huffman-based methods. Nevertheless, the Shannon-based variant with variable-length update intervals (column 5 in Table 2) achieves the fastest encoding and decoding times. Although its encoding speed is similar to that of Huffman-based methods (columns 8 and 10), its decoding is nearly twice as fast as that of Huffman canonical codes.
- **Update interval.** For the first three texts, variable-length update intervals yield compression ratios comparable to or better than those achieved with fixed-length intervals (as defined in Gagie's algorithm). This is attributed to the relative homogeneity of these texts. In contrast, the fourth text − comprising a mix of source code snippets in different programming languages, large constant arrays, and other heterogeneous components − benefits more from frequent updates, making fixed-length intervals advantageous. However, the improvement in compression ratio is modest (less than 1.5 %), while the encoding and decoding are significantly slower across all texts. This indicates that the fixed-length update strategy, as proposed by Gagie, introduces a major performance bottleneck. The slowdown becomes especially critical when the decoding table is large enough to exceed cache capacity (see columns 4, 7, and 9 of Table 2).
- **Smoothing.** The smoothing technique proposed by Gagie adjusts character probabilities to reduce the size of the decoding table and thereby accelerate both its construction and overall decoding performance. This effect is validated experimentally (see column 3 vs. column 4, and column 5 vs. column 6 in Table 2). The trade-off is a slight loss in compression efficiency − less than 1.5 % in most cases, except for the BWT+MTF preprocessed text, where it approaches 7 %. This exception is due to frequent character repetitions in that text, where small changes in probability distributions over short intervals can substantially alter the codeword set. Notably, smoothing using the formula from [3] does not affect the structure of the Huffman tree in our experiments and therefore does not affect the results for Huffman-based encodings. For this reason, smoothed and non-smoothed variants of Huffman-based methods are not distinguished in Tables 1 and 2.
- **Canonical codes.** Canonical Huffman codes are used to speed up the decoding of long codewords (typically those exceeding 12–13 bits) without affecting compression quality. Long codewords often appear when character frequencies differ significantly, as can happen in large texts or when statistics are gathered over longer intervals. This is supported by the data in columns 8 and 10 of Table 2,

[1] The source code can be found at https://github.com/zavadsky/low-adaptive.

where canonical decoding outperforms its non-canonical counterpart for three of the larger texts and underperforms only on the shortest one. The benefit is especially noticeable in the BWT+MTF preprocessed text, where character frequency differences within blocks are particularly large.

Table 1: Comparison of the compression efficiency, bits/character
(FLI - Fixed Length Intervals, VLI - Variable Length Intervals).

Text	Vitter	Shannon-based				Huffman-based	
		Gagie original	*FLI, non-smoothed*	*VLI*	*VLI, non-smoothed*	*FLI*	*VLI*
1	4.618	5.149	5.099	5.137	5.091	4.629	4.625
	(1.04%)	(12.65%)	(11.57%)	(12.4%)	(11.4%)	(1.29%)	(1.21%)
2	4.385	4.872	4.852	4.87	4.824	4.387	4.39
	(0.97%)	(12.19%)	(11.7%)	(12.12%)	(11.06%)	(1.02%)	(1.08%)
3	3.079	3.625	3.415	3.443	3.418	3.085	3.078
	(0.49%)	(18.31%)	(11.47%)	(12.36%)	(11.54%)	(0.69%)	(0.49%)
4	5.57	6.139	6.064	6.22	6.136	5.572	5.614
	(0.34%)	(10.6%)	(9.24%)	(12.06%)	(10.54%)	(0.38%)	(1.13%)

Table 2: Encoding and decoding time, seconds
(FLI - Fixed Length Intervals, VLI - Variable Length Intervals).

Text	Vitter	Shannon-based				Huffman-based		Canonical Huffman	
		Gagie original	*FLI, non-smoothed*	*VLI*	*VLI, non-smoothed*	*FLI*	*VLI*	*FLI*	*VLI*
1	2	3	4	5	6	7	8	9	10
1	0.063	0.0169	0.017	0.0051	0.0051	0.017	0.0051	0.017	0.0053
	0.176	0.0197	0.234	0.0058	0.04	0.28	0.0084	0.28	0.011
2	0.578	0.11	0.117	0.04	0.041	0.136	0.042	0.131	0.042
	0.582	0.14	15.52	0.049	0.12	25	0.098	24.16	0.093
3	0.423	0.134	0.129	0.045	0.047	0.16	0.051	0.152	0.048
	0.533	0.166	10.55	0.053	0.192	15.92	0.279	15.55	0.102
4	3.44	0.692	0.642	0.22	0.215	0.695	0.222	0.724	0.236
	4.51	0.723	147	0.26	0.787	227	0.675	215	0.48

Let us note that all codes in research satisfy the $H + 1$ compressed size limit, proved in theory for Gagie's algorithm. However, for a character-based alphabet, this one extra bit per character gives more than 20% of the compressed file size. Therefore, the worst-case optimality limit is too weak for practical applicability.

The experimental results are summarized in the plane (compressed size, decoding speed) shown in Figure 1. Each algorithm is represented with four markers that correspond to four tested texts. Compression rate is given as the percentage of excess over the entropy, while the decoding speed is shown in microseconds per character. Among algorithms that update the code over fixed-length intervals, only Gagie's original algorithm is presented, as all the others have extremely high decoding times (columns 4,7, and 9 of Table 2).

All markers in Figure 1 can be divided into two groups: the right group corresponds to algorithms based on Shannon codes, while the left group corresponds to algorithms based on Huffman codes. The wide gap between the two groups highlights the low compression efficiency of Shannon codes in practical applications. However, two series of round markers occupy the most attractive Pareto-optimal areas. The round filled markers correspond to Shannon-based encoding with smoothed character probabilities and variable-length update intervals. This method provides the worst compression ratio (though not significantly worse than other Shannon-based encodings), but the best decoding time. The round empty markers correspond to adaptive encoding with variable-length update intervals based on Huffman canonical codes. It combines the compression rate almost on a level with Vitter's algorithm, with good decoding speed, which is inferior only to the fastest version of Shannon-based codes.

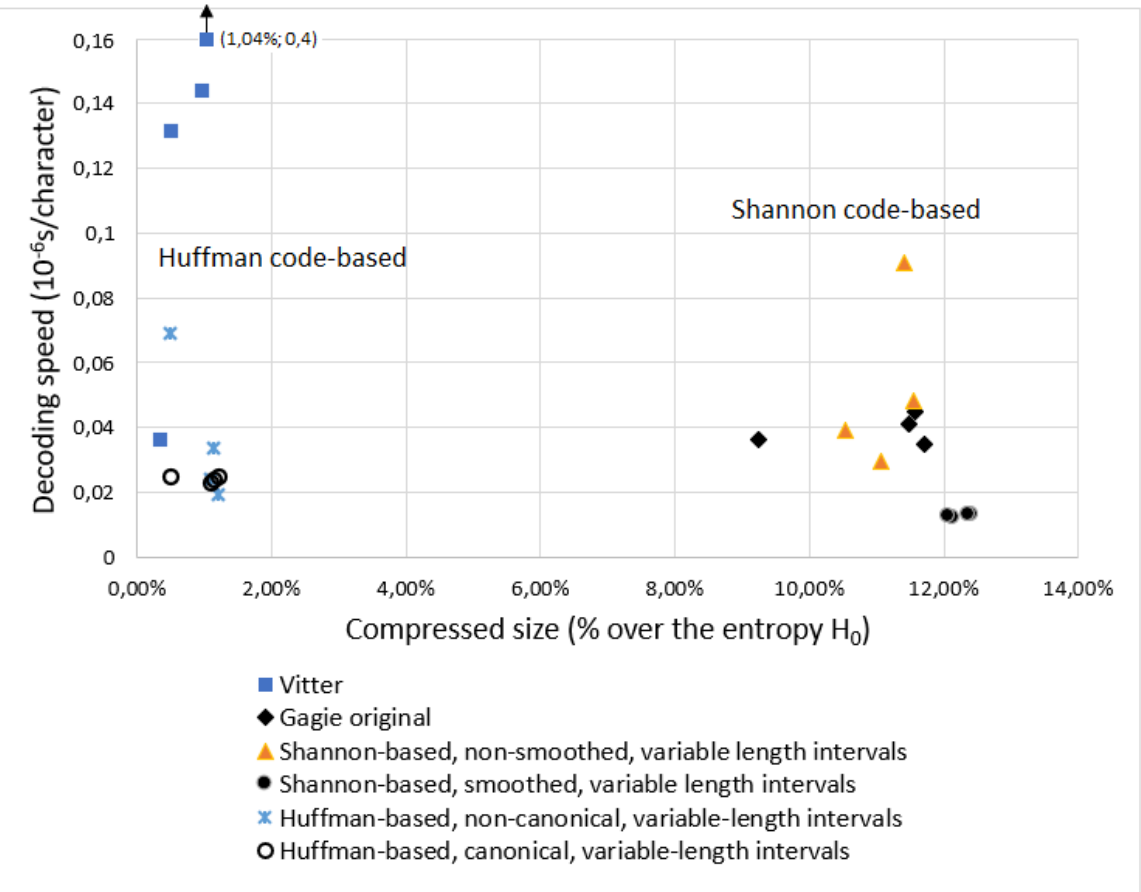


Figure 1: Compression efficiency and decoding speed of tested algorithms

4 Conclusions

We investigate the practical applicability of various classical and recently proposed adaptive encoding algorithms for character-level text compression. Among the evaluated methods, Vitter's classical algorithm consistently achieves the best compression ratio. However, alternative approaches significantly outperform it in terms of encoding and decoding speed. Gagie's algorithm, which is provably optimal in both space and time in the worst case, based on properties of the underlying Shannon code, appears

to be space-inefficient in practice – an issue shared by other Shannon code-based adaptive methods. Nonetheless, when the code is updated at geometrically increasing intervals rather than at the originally proposed $\lceil \sigma \log n \rceil$ intervals, the algorithm attains the highest observed encoding and decoding speeds. Furthermore, replacing the Shannon code with a canonical Huffman code yields a highly practical encoding scheme, achieving a compression ratio comparable to that of Vitter's algorithm while offering significantly superior performance in terms of speed. The theoretical investigation of worst-case optimality for Huffman code-based adaptive compression methods remains an open direction for future research.

References

1. N. Faller: *An adaptive system for data compression*, in Record of the 7th Asilomar Conference on Circuits, Systems and Computers, 1973, pp. 593–597.
2. T. Gagie: *Dynamic Shannon coding*, in Proceedings of the 12th European Symposium on Algorithms (ESA), 2004, p. 359–370.
3. T. Gagie: *Simple worst-case optimal adaptive prefix-free coding*, in Proceedings of 30th Annual European Symposium on Algorithms (ESA), vol. 244, 2022, pp. 57:1–57:5.
4. T. Gagie and Y. Nekrich: *Worst-case optimal adaptive prefix coding*, in Proceedings of the 11th Symposium on Algorithms and Data Structures (WADS), 2009, p. 315–326.
5. R. Gallager: *Variations on a theme by Huffman.* IEEE Transactions on Information Theory, 24(6) 1978, pp. 668–674.
6. M. Karpinski and Y. Nekrich: *A fast algorithm for adaptive prefix coding.* Algorithmica, 55(1) 2009, p. 29–41.
7. D. Knuth: *Dynamic Huffman coding.* Journal of Algorithms, 6(2) 1985, p. 163–180.
8. M. Liddell and A. Moffat: *Decoding prefix codes.* Software Practice and Experience, 36 2006, pp. 1687–1710.
9. R. L. Milidiú, E. S. Laber, and A. A. Pessoa: *Bounding the compression loss of the FGK algorithm.* Journal of Algorithms, 32(2) 1999, p. 195–211.
10. E. S. Schwartz and B. Kallick: *Generating a canonical prefix encoding.* Communications of the ACM, 7 1964, pp. 166–169.
11. J. S. Vitter: *Design and analysis of dynamic Huffman coding*, in Proceedings of the 26th Symposium on Foundations of Computer Science (FOCS), 1985, p. 293–302.
12. I. O. Zavadskyi: *Fast practical adaptive encoding on large alphabets*, in Data Compression Conference, DCC 2025, Snowbird, Utah, USA, 2025, pp. 141–150.

Electronic Alternatives to Micro-Fiches

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

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

Abstract. Micro-fiches have historically been used for distributing and storing large amounts of data, but data access was slow and cumbersome, and searching for a specific item was often involved with hour-long browsing. Today there are of course electronic alternatives, and we study the possibility of recoding *logically* a printed page, instead of *physically* reducing its size. Such coding allows subsequent lossless compression techniques to be applied. The paper describes the suggested process, including a novel approach to compress numerical data, and presents the details of a typical example.

1 Introduction and Background

Long before the disk-on-key USB stick became the medium of choice for distributing and storing large amounts of data, and even before their predecessors, the CD-Roms, did micro-fiches serve this purpose. Newspapers and other publishers of primarily textual information used to archive their products by photographically reducing them to small cards, a small box of which could contain thousands of newspaper pages. The principal goal was indeed storage, and retrieval had much lower priority, requiring special micro-fiche viewing devices. Data access was therefore slow and cumbersome, and searching for a specific item often involved hour-long browsing.

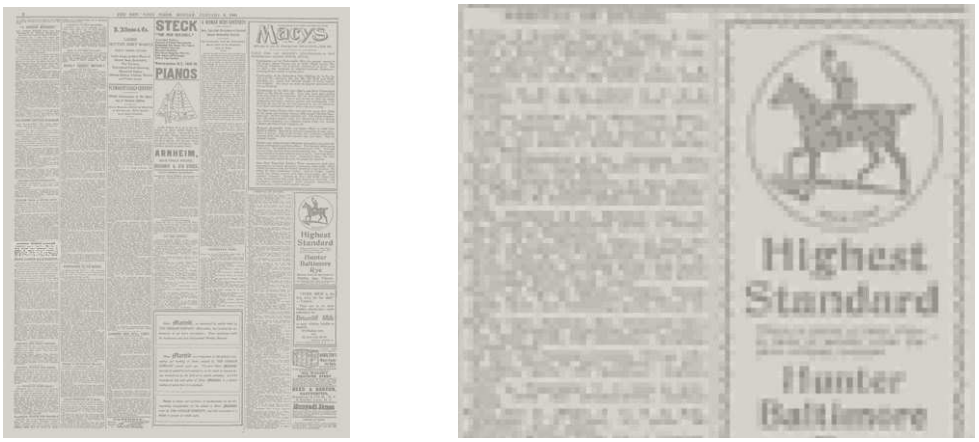
The present suggestion deals with an electronic alternative to the classical micro-fiches. Instead of physically reducing the size of a printed page, it could be *recoded* logically. Such coding allows subsequent lossless compression techniques to be applied and moreover, the output may be stored on electronic or magnetic storage, which is by far superior with respect to the retrieval possibilities vs. the old micro-fiches.

Large information retrieval systems, in which the full texts are stored electronically and which allow direct access to any word or phrase, have been developed since the late sixties. Today, of course, practically all available data are produced electronically, and access is thereby facilitated. The exact layout and pagination is mostly not considered as an important factor, and the same information is often displayed differently on different devices, be it a small portable phone or a large computer screen. Our main concern here, however, is with retrieval systems in which the original layout of the text *is* of importance, as in the following applications:

– *Historical editions.* The study of historical texts may be of interest to humanists, historians, Bible experts and other scholars of classical texts. There are often well accepted standardized and widely referenced editions, which ought to be stored in their original form and pagination. A case in point is the Babylonian Talmud, whose 6000 pages of the Vilna Edition have become so widespread, that references to the Talmud are generally given by page number. Other examples could be original manuscripts of famous authors, poets and composers, and Holy Texts for several religions.

Shmuel T. Klein, Dana Shapira: Electronic Alternatives to Micro-Fiches, pp. 13–25.
Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9

- *Journal pages.* Today, newspapers are designed, prepared and printed with the help of computers, but older journals have not been stored electronically. Nonetheless, reporters, anthropologists, lawyers and many others are often concerned with older editions, and much can be learned from the co-occurrence of certain data items on the same page or in proximity. As an example, consider in Figure 1 a page from the *New York Times* archive from 1900, and an enlarged detail of it; as can be seen, the text is not readable.
- *Technical design.* Manufacturers of complex machines such as cars, aircrafts, etc., used to store repair instructions in thousands of detailed design sheets. These included drawings and texts.



(a) NYT page from 1900 (b) Enlarged detail

Figure 1: *Example of archived NYT page*

2 Related Work

The usual approach to store such data electronically, was to scan the printed page as if it were a picture, and store it as a raster file. This always resulted in a dramatic reduction in picture quality, depending on the resolution chosen for the scanning process. Recently, scanned images, even in color, are of much better quality, but at a price: a color pixel may be coded in three bytes, covering a possible range of $2^{24} = 16$ million color shades; images are scanned at high resolutions, typically 600 dots per inch (dpi) or higher (i.e., 360,000 pixels per square inch), so that the scanned image of a standard $8.5'' \times 11''$ sheet would occupy about 96 MB! Even if modern compression methods may reduce this to merely 1% (using some standard like JPEG [28], or similar methods, all of which are *lossy*, i.e., discard a part of the information), we are still left with about 1 MB for a single sheet, which is much too large for many practical applications.

The state of the art today is the *Portable Document Format*, known by its acronym PDF, which is widely used. It includes many of the features we are interested in, but using PDF may be an overkill, and a typical page at reasonable resolution and containing a large amount of text may span about 3MB and even more per page. The use of the PDF format has recently been criticized in [7].

Our main concern here is with primarily textual data. On the one hand, this simplifies the problem, as there is no need for colors to be coded. A pixel can thus be stored in a single byte instead of three, giving a range of 256 grey levels, or ultimately even in a single bit, for strictly black and white text images. A standard $8.5'' \times 11''$ sheet, still at 600 dpi, would thus need about 4 MB, or about 1 MB at the lower resolution of 300 dpi. On the other hand, lossy compression techniques are no more adequate, and the standard lossless compression methods, when applied to data of this type, rarely reduce their size to less than about 30%. We are thus still left with about 0.3 MB per page. In addition, a text image at 300 dpi is not always readable if small fonts are used.

Current text compression methods achieve a reduction of up to 70%, e.g., [32], but do generally not care about the original page layout. Keeping both the text in textual form as well as a scanned image is generally considered as redundant and is not supported by many available systems.

Many text compression methods have been published in the literature and as patents. Most fast on-the-fly methods are based on algorithms by Lempel and Ziv [33, 34]. A partial list of patents based on these includes [9, 11, 29, 30] and many others. A major problem of these algorithms is to find an efficient way to locate a recurring string in the text. Efficient solutions have been given in [8, 20]. When the speed of the encoding process is not critical, as in applications addressed in the current method, in which the compression may be done off-line in a preprocessing stage, better compression can be achieved by what is called a recompression method, as in [18]. As an alternative, one may use a compression method that does not require any decompression at all and allows the search for a pattern directly in the compressed text. This paradigm is known as *compressed pattern matching* and has been thoroughly investigated, e.g., in [22].

The compression of layouts has not been addressed as a topic in its own right. What might come closest is the compression of structured data such as lists of numbers or bit-vectors, as for example in [1, 6]. The system suggested herein does not claim to invent a new special purpose compression method, to be applied to images of textual data. Rather, it suggests a new approach, encompassing methods known as prior art, but unifying several such methods into a new coherent system.

As to related work, a similar system to the one suggested below, albeit with another application area, was suggested by Wong and Chan [31]. They teach how to store computer display information, which may contain both text and graphics. However, since the information there is supposed to be given already electronically, the basic assumptions are different from ours. In particular, [31] does not address the problem of scanning a page given in printed format, storing its layout and *simulating* its original form.

Reference to the reconstruction of layouts can be found in Morgan [25], who deals with generic forms. He defines the layout as a set of rectangles, similarly to what we do, but it is different in that it allows dynamic alterations of the general layout; for example, columns may grow or shrink, depending on the text they should contain, whereas for the applications we have in mind, the given page is supposed to be fixed and imposed as a part of the input, and its exact original form has to be approximated as closely as possible.

A number of publications that mainly appeared as patents deal with creating, for a given page, the layout which is best under some criterion, out of a set of possible layouts. What these methods have in common with our suggestion is that they scan the page and convert it into a sequence of images and text parts. But the purpose is not to reconstruct a page similar to the original, but to change it and produce some optimal tiling. Some of these works can be found, e.g., in the patents Hart et al. [12], Hayashi and Saito [13], Mason [24] and Kataoka et al. [15].

Many works, among others, Hernandez et al. [14], Fukui et al. [10], Borgendale and Dobkin [3], deal with text editors, in which the page can be changed. They include the treatment of graphics, but are not aimed at displaying a fixed printed page. Finally, Knuth's [23] TEX typesetting software (that has been used for the preparation of this paper) produces a printed page in a Device Independent file format that shows many similarities to the proposed system, but lacks the requirement of producing a layout closely simulating a given original.

3 General Description of the Suggested System

The main idea of our work is a shift of focus: instead of taking high quality pictures of a text page and approximating the original by trying to keep the loss of information due to the application of some lossy compression technique to a minimum, we try to approximate the building blocks at the lowest level, i.e., the images of the alphabet characters in various fonts, record the general layout of the page to be printed, and thereby *simulate* the image of the text page. The following advantages may thus be achieved:

- Even if the simulation is not an absolutely true copy of the original, the difference might be very hard to notice, if at all. Anyway, a scanned image is not a true replica either, because of the limited resolution.
- The picture quality will be by far superior to that of a scanned image.
- The necessary space will at the same time be considerably reduced, actually so far as to enable to storage of millions of images on a single USB stick.
- Contrarily to a scanned image, the simulated one can be the basis of a location sensitive software package, allowing access to and processing of the *words* on the page, not just their image. This opens new opportunities, such as grammatical analysis, automatic translation, various Hypertext functions, etc.

In a first phase, the underlying text has to be stored as such, rather than as a picture. For many applications, this incurs no additional overhead at all, since the text is anyway available. This is true for literary works like those collected in the *Bibliothèque de la Pléiade* [26], which is a part of the French full text retrieval system known as *Trésor de la Langue Française*, described in [4]; other examples might be famous texts such as the Bible or the Talmud. In case the text has not been recorded previously, this may be done with the help of optical character recognition (OCR) packages, which nowadays operate with very high accuracy, especially if printed and not handwritten pages are scanned. The overhead, even in this case, will not be significant. To return to the above example of a standard $8.5'' \times 11''$ sheet, it rarely contains more than about 1000 words or 5000 characters. Using even simple compression methods, this could be stored in less than 3K, and more typically, for less dense

pages and with better compression, in about 1K — just 0.1% of our earlier estimate of the size of a stored image of such a sheet.

The second phase consists of recording the general layout of the page. Practically all printed pages may be decomposed into a set of non-overlapping rectangles. For newspapers these consist usually of several columns. For various literary works, the layout may be more sophisticated, with the main text in several rectangles in the center, surrounded by rectangles of various shapes, holding commentaries written in smaller fonts. This layout can be stored effectively, since all we need for exactly locating a rectangle is the coordinates of two of its diagonally opposed corners, as in [25], or equivalently the coordinates of one of the corners and the height and width of the rectangle, as in [12]; in any case, two number pairs suffice. If two bytes are used for each of these numbers, a granularity of 10^{-3} inches may be achieved, by far high enough so that any deviation below it may not be detected by the human eye. Even if up to a hundred rectangles have to be stored for a single page (and the actual number is usually much smaller), the storage requirements for the layout are still below 1K.

The next step consists of "reconstructing" the original form of the printed page, using the text and the layout. This might be complicated for handwritten manuscripts, for which the form of the lines, as well as the spaces between lines, words and characters, may fluctuate. But for printed pages, the inter-line space is usually constant, and within a line, the words are generally spread out as uniformly as possible.

The amount of space between characters within a word, as well as the height of the characters influencing the space between the lines, are a part of the font specification [23]. In older printed texts, the typesetter may have physically arranged the letters and words and adjusted the spaces manually, yielding unavoidable fluctuations. One of the assumptions of this work is, that it is not worth investing the effort (and the additional bits) allowing the faithful reconstruction of these rather arbitrary deviations from equally distributed inter-word spacing. So even if in the simulated page, the spaces are not exactly matching those of the original one, the differences will mostly remain undetected, but even when they are visible, no important data has been lost.

All one needs to store to enable the reconstruction of the page is therefore: for each rectangle, a pointer to the starting word in the text, the number of lines within the rectangle, and for each line, the number of words it contains. Such lists of numbers may be stored very efficiently (as e.g., in [6]), and the amount of space needed for it per page will generally be of the order of 1–2K.

In a final touch, thought has also to be given to pictorial and any other data that is not textual in nature. Though we assume here that the bulk of the information is text, there might be small ornaments, pictures, drawings, etc., or even characters in special fonts (like an overdimensional starting character of a chapter), that have to be embedded. These will be kept in image form, yet it will not have the disadvantages of storing the full page as a raster file:

1. for mainly textual data, for which this method is intended, the overall proportion of pictorial data should not exceed a few percent; furthermore, lossless techniques can subsequently be applied.
2. for the data that will be kept in image form, standard lossy compression techniques are suitable, since the images will not represent small type characters.

The main purpose of displaying a page is to let the user browse through it. This is the only function supported by conventional systems based on micro-fiches (where this

requires special hardware) or on the representation of the text page as an image. But here, there is also a possibility of using sophisticated search mechanisms, as supported by text retrieval systems. One can therefore easily look for words or phrases, using well-known pattern matching techniques such as [5], or use more advanced tools, including grammatical variants and metrical constraints, requiring dictionaries [21], concordances or signature files [16, Section 8.1.1], that can be compressed by various techniques [17, Section 6.5].

In addition, the system allows also interaction with the page. With the help of a pointing device, which could be a mouse, a lightpen, or anything else supported by touch-sensitive screens like, ultimately, our bare fingers, certain parts of the text image can be "marked" as selected. Given that the system has created the display and the layout, basing itself on the text, which is stored separately, it is possible to evaluate exactly the characters displayed in the region on the screen that has been pointed to. In other words, the system can "understand" which words or signs have been selected. One can therefore enlarge selected parts of the image, as we are all used by handling our smartphones, zooming in and out on request, which is especially useful in the presence of small type fonts.

A zooming function may also be available if the full page has been stored as an image. But in that case, enlarging the display cannot add information that has not been stored beforehand, so that the resolution at which the page has originally been scanned essentially limits the usefulness of such enlargements. If certain characters of the scanned font are of the size of a small number of pixels, they might be impossible to read regardless of the enlargement used, as can be seen in Figure 1(b).

Some of the possible interactions may include, among others: using dictionaries for the automatic translation of selected words into another language; for proper nouns, automatic access to historical, geographical or biograph- ical data; for any word or even character string, giving all possibilities to interpret it (very important for languages with high frequency of homonyms), adding global statistical information, such as the number of times the string occurs in this and other texts; if a full phrase, rather than a single word is selected, access to parsers, taggers or other data of interest to linguists.

4 Compression methods

One of the possible partitions of the vast area of Data Compression is by the nature of the data to be handled. A large amount of the data considered as important enough to be stored is *textual*, generally written in some natural language. Another significant part of the data is pictorial: *images* of all kinds and sizes, often collected into sequences forming videos and films. We wish to concentrate here on a third kind, which is *numerical* data. This differs from the textual form, of which it is a subset, in several ways:

1. There is no inherent redundancy, which is exploited by many of the text compression techniques;
2. The underlying alphabet, generally consisting of just 2, 10 or 16 digits, plus some whitespace, is much smaller than for text, especially if the elements to be encoded are not restricted to be just simple characters, ASCII for example, but may be chosen as the different words in a textual database. If numbers, rather than digits, are to be encoded, the alphabet is potentially infinite;

3. Contrarily to text, errors will be impossible, or much harder, to detect;
4. Depending on the origin of the data, lossless compression may be required, or various degrees of lossy methods could be tolerated.

More specifically, most of the data that needs to be encoded for the layout consists of the description of rectangular bounding boxes into which the various texts have to be placed. Practically all of these rectangles have their edges parallel to the borders, so that two points in the plane, corresponding to diagonally opposed corners, suffice to obtain a well-defined rectangle. The rare cases of rectangles or other forms deviating from this assumption about the orientation are dealt with as *additional images* in the third phase of the suggested system. Figure 2 shows a small sample of the 4-tuples defining the rectangles on our test file to be described below. The measurements are given in points (pt), where 1pt is $\frac{1}{72}$ of an inch, about a third of a millimeter or more precisely 0.353mm. Obviously, the precision given by the software of up to 15 digits after the decimal point is a senseless overkill, measuring distances of the order of attometers (a billionth of a nanometer).

```
107.94351196289062   28.557645797729492   527.6393432617188   581.769775390625
107.93960571289062   58.182525634765625   527.402099609375    717.7293701171875
255.5399169921875    125.14797973632812   379.80010986328125  193.12799072265625
278.819091796875     133.1363067626953    356.4554138183594   167.07965087890625
232.61822509765625   200.4618682861328    527.4641723632812   615.6199951171875
```

Figure 2: *Sample of 4-tuples defining rectangles*

We suggest the following compression method that is especially adapted to deal with such layout data. It could be qualified as being *semi-lossless*, as on the one hand, it is lossy by not restoring accurately the original file bit per bit, yet it is lossless in the sense that no useful information has been removed. This new paradigm could find applications beyond those discussed here in the context of numerical data. In an application to X-rays, for instance, one could tolerate a certain degree of lossy compression, as long as the reconstructed image could be judged as "diagnostically equivalent" to the original.

There are several freely available tools helping to extract information about bounding boxes from PDF pages, and we used PyMuPDF [27] on our test data. Every quadruple of numbers is given in decimal form in about 70–80 bytes, representing four 64-bit floating point numbers.

We shall store each number x by encoding separately the integer and fractional parts, I_x and F_x, respectively. Using just 4 bits for F_x partitions the unit into 16 equi-sized parts, so by choosing the integer k, $0 \le k < 16$, such that $\frac{k}{16}$ is closest to F_x, the potential error is bounded by $\frac{1}{32}$ pt, about one hundredth of a millimeter. This is hardly noticeable to human eyes. Taking the last number in Figure 2 as example, $F_x = 0.619995\cdots$ and will be represented by the integer $k = 10$, standing for $\frac{10}{16} = 0.625$ pt; the error is less than 0.002mm.

As to the integer part I_x, we first note that for the application at hand, the distribution of the occurring values is not uniform. This is due to the fact that the rectangles are not placed arbitrarily within a page, but that the typesetter tried to impose some alignment, so that certain values of the x part in the (x, y) coordinates, indicating the offset from the left edge of the page, show a tendency to re-occur. Of course, the high precision floats will never be matching exactly, but when quantization to the integer part level is used, repetitions will appear. This suggests that applying Huffman coding to the set of possible values might yield some savings.

Instead of preparing codewords for all the possible values (about 800 for a page of height 11"), requiring 10 bits, we evaluated the average length of a Huffman codeword for each of the four components of the two pairs (x_0, y_0) and (x_1, y_1) separately. On our test data, this yielded averages of 3.93, 5.49, 4.59 and 5.92 bits, respectively. As can be seen, the averages for the x components are smaller than for the y components, which can be explained by the fact that there is apparently more regularity in the *width* of the rectangles, depending on the horizontal offset from the left edge of the page, than in their *height*, measured as vertical offset from the top edge. Indeed, many different pages will display rectangles with very similar widths, but the heights are adjusted according to the desired balance between the amounts of the various texts that have to be included.

In a final touch, we mentioned already that a rectangle can be encoded equivalently by (x_0, y_0) and $(\mathit{width}, \mathit{height})$, instead of (x_0, y_0) and (x_1, y_1), where $\mathit{width} = x_1 - x_0$ and $\mathit{height} = y_1 - y_0$. This could be advantageous, because width and height are translation invariant, so one may expect more repetitions, and thus a more skewed distribution, implying better performance of the Huffman code. On our test data, the average number of bits for *height* and *width* are 5.82 and 4.16 bits, instead of 5.92 and 4.59 for y_1 and x_1, a further improvement of 2–10%. The expected total number of bits required for a quadruple is thus 36.4, or 9.1 bits per number.

Figure 3 gives a schematical view of the compressed file, in which variable length color-coded Huffman codewords, standing for the integer parts of the four different components, alternate with lossy fixed length encodings of the fractional parts. Decoding is possible because each of the Huffman codes is prefix-free, and so is therefore also a sequence of alternating elements from different Huffman codes, even if interspersed with fixed-length codewords. A detailed example for this compression method is given below.

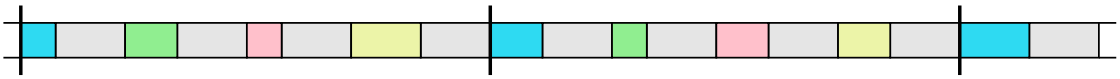
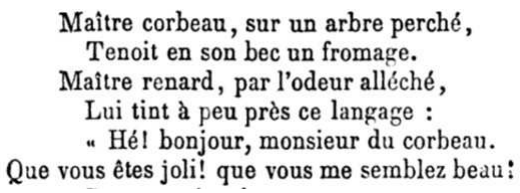


Figure 3: *Schematic view of the compressed form of the coordinate quadruples*

5 Typical Examples

A first example of the usefulness of the new system can be found in Figure 4, displaying the first few verses of the famous fable *Le Corbeau et le Renard* about the Raven and the Fox by *Jean de La Fontaine* (1621–1695). The left part is from a historical edition printed in 1890, and spans more than 303K in its original raster form of 554×187 pixels, which is reduced to about 31.3K by JPEG. Though readable, the quality is very low. The version on the right can be magnified as much as desired, is almost visually identical to the original, yet requires, even in uncompressed form, less than 0.4K bytes. This evaluation includes the LaTeX formatting commands, but excludes the definition of the characters in the various fonts, which are considered negligible overhead when amortized over a large enough text corpus.

The following additional example has already been mentioned in the introduction and will be brought here in more detail. The Babylonian Talmud is one of the major works of Jewish culture. It includes texts relating to all aspects of Jewish life, and describes the discussions about various possible interpretations of the Bible, held by more than three thousand scholars, mainly from Babylon and Israel, who lived


Maître corbeau, sur un arbre perché,
Tenoit en son bec un fromage.
Maître renard, par l'odeur alléché,
Lui tint à peu près ce langage :
« Hé! bonjour, monsieur du corbeau.
Que vous êtes joli! que vous me semblez beau!

(a) From a book printed 1890

Maître corbeau, sur un arbre perché,
Tenoit en son bec un fromage.
Maître renard, par l'odeur alléché,
Lui tint à peu près ce langage :
« Hé ! bonjour, monsieur du corbeau.
Que vous êtes joli ! que vous me semblez beau !

(b) LaTeX version

Figure 4: *Example of a Fable by La Fontaine*

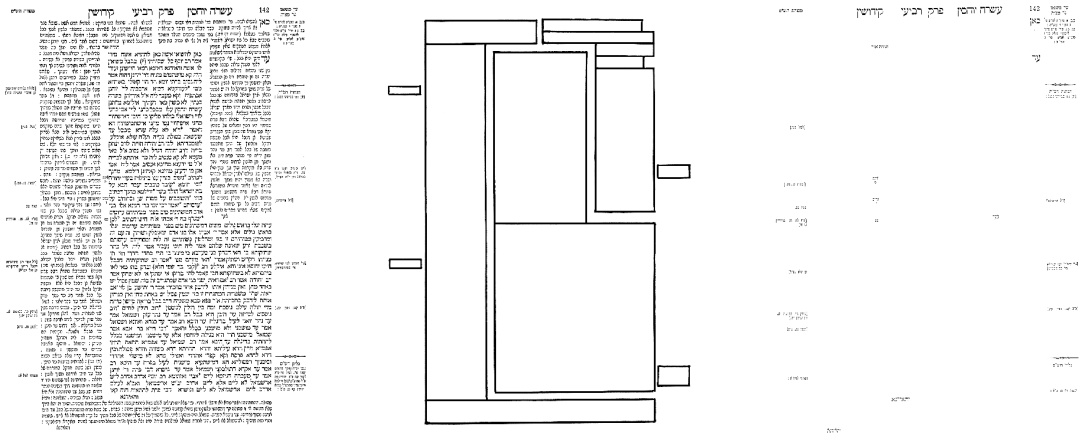
between the first and fifth centuries. The Talmud consists of 36 tractates, and was first printed by Daniel Bomberg in Venice in 1520–23. This edition had a total of 5894 pages, and most posterior editions kept this partition. The talmudic text consists of about 2.5 million words, or about 13 MB.

Figure 5(a) displays a typical page (71b of tractate *Kiddushin*). The text of the Talmud can be found in the center, surrounded by various commentaries: the most important ones are those put immediately adjacent: on the left the commentary by Rashi (Rabbi Shlomo Yitzhaki, 1040–1105), and on the right by the Tossafists (13^{th} century). More annotations are further away from the center, and appear generally in smaller print. The top line is a running title, giving the name of the tractate, and name and number of the current chapter. It is important to note that this layout, corresponding to the edition printed in Vilna by the Romm family in 1880–86, has been practically universally accepted, and thousands of Talmud scholars all over the world read, study and refer to these pages in exactly this form. It is thus not rare that one remembers certain passages not so much by their exact word sequence, but by their location (e.g., close to the upper right corner) on the page. Hence the need to store not only the text, but to store it specifically in this form.

Figure 5(b) brings the general layout of the page of Figure 5(a). Note that a small number of rectangles is sufficient to cover the bulk of the text. The small squares missing in the upper right corner and closely below in the layout correspond to words written in the original page in a special larger font. Note also that not all the text close to the borders has been put into rectangles in this example, to emphasize the possibility of leaving even text parts, like short comments, as additional pictures. In the given example, only texts spanning two or more full lines are recorded. Figure 5(c) then gives whatever has not been covered. This includes here the title line, small geometric ornaments, words in special fonts and shorter comments, but could be significantly reduced by also storing the rest of the text, and leaving in image form only non-textual data, which is rare in the Talmud.

In addition, cross-references are extremely frequent in the Talmud, which therefore is ideally suited to become a Hypertext environment [2]. Having this possibility directly on the displayed page that looks identical to the page the user is familiar with from his books, is a great advantage over other display systems without Hypertext support.

We now proceed to a rough calculation of the required space by the different methods. Even if we assume that one can store a typical page of the Talmud in 200K in compressed form (which corresponds to low resolution and reduced quality), one would need a total of 1.2 GB for the entire Talmud. A simple scanning application like those available on our mobile phones produces low-quality PDF images of about 0.5 MB per page, totaling 3GB for the full text. Moreover, this would only include the pictures of the pages, no text, dictionary or concordance. On the other hand,



(a) Original page (b) Layout (c) Additional pictures

Figure 5: *Example of typical text page*

the text alone of the full Talmud, together with the commentaries by Rashi and the Tossafists, span less than 50 MB, and can easily be compressed to 25 MB. To this we have to add the exact pictures of all the characters in all the fonts that are used in the text, but this is done only once for the full Talmud, and will overall occupy less than 1 MB. If we count 1K per page of layout (6 MB in total), another 1K for storing the number of words in each line (6 MB in total) and even 4K for the compressed form of the additional pictures per page (24 MB in total), we end up with about 60 MB, roughly 2% of the low resolution parallel in PDF. One can thus add other, related, texts, and all the auxiliary files mentioned above for gaining access to the text with the help of an information retrieval system [19].

To show an example of the compression of the layout data, we took the ten first pages of the Talmud, Tractate *Brachot*, pages 2a to 6b, and applied PyMuPDF, which extracted 135 bounding boxes, each defined by 4 high precision floating point numbers, (x_0, y_0) and (x_1, y_1), the first five of which are depicted in Figure 2. As explained above, the actual encoding is performed on

$$(x_1 - x_0, y_1 - y_0) = (\texttt{419.69583129882818, 553.212129592895508}),$$

instead of (x_1, y_1). The four integer parts are therefore 107, 28, 419 and 553, and the corresponding fractional parts, starting with 0.9435, 0.5567, 0.6958 and 0.2121 are encoded by the 4-bit fixed length standard binary representations of the integers 15, 9, 11 and 3, respectively.

Four different Huffman trees are then constructed, one for each of the distributions of the possible values of x_0, y_0, $x_1 - x_0$ and $y_1 - y_0$. For our example, 135 values with repetitions are sampled for each of the trees, yielding the trees shown in Table 1. We here use the notation $\langle n_1, n_2, \ldots, n_k \rangle$, known as a *quantized source*, where n_i is the number of codewords of length i in a given Huffman tree. If *canonical* Huffman codes are used, in which the depths of the leaves are non-decreasing from left to right, the quantized source suffices to uniquely define the layout of the Huffman tree and accordingly, the corresponding codewords, after having ordered the elements by non-increasing frequencies. Once we know the optimal length l_i of the i-th codeword, the codeword itself can be defined as the first l_i bits following the binary point in the infinite expansion of the binary number $\sum_{j=1}^{i-1} 2^{-l_j}$.

Table 1 shows, for each distribution, the corresponding quantized source, the integer to be encoded, its rank in the ordered list of different elements, the depth of the corresponding leaf in the Huffman tree, which is the length of the codeword used to encode the element, and finally the codeword itself.

distribution	quantized source	integer	rank	depth	codeword
x_0	$\langle 0,2,0,2,6,9,6\rangle$	107	8	5	10111
y_0	$\langle 0,0,0,4,10,8,40\rangle$	28	5	5	01000
$x_1 - x_0$	$\langle 0,0,4,4,2,5,14\rangle$	419	4	3	011
$y_1 - y_0$	$\langle 0,0,0,4,4,6,68\rangle$	553	64	7	1101101

Table 1: *Huffman trees and encoding of the integer parts.*

Figure 6 summarizes the example, showing the encoding of the first 4-tuple and using a color-code to match that of Figure 3. The decimal value of the grey binary fixed-length parts appear above the encoding, the Huffman encoded values are listed below their variable-length blue, green, pink or yellow codewords. In this example, four floating point numbers are encoded by $5 + 5 + 3 + 7 + 4 \times 4 = 36$ bits, exactly 9 bits per number. Extrapolating from this (not necessarily representative) example, we get an estimate of less than 3MB for the layout data of the entire Talmud.

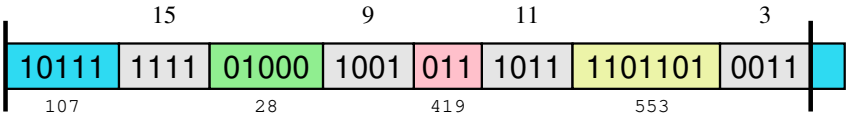


Figure 6: *Encoding of the first 4-tuple*

The Talmud is today commercially available in picture form, with varying degrees of accessibility. The suggested system would make the full collection available on significantly reduced storage space, in much higher quality, and allow many functions to be performed that are not supported by picture-based systems. There are also Information Retrieval Systems on the market that support full-text access but lack the ability of showing the retrieved text in its customary form. By increasing the total size of the system only slightly, the suggested method could also add this desirable feature.

Summarizing, the contribution of this paper is the suggestion of replacing the modern alternatives to micro-fiches, in which we assume that not only the content, but also the exact original layout is of importance, by a system that *simulates* the output page rather than reproducing it as a picture. The loss should be hardly noticeable by the bare eye, but the advantages in terms of compression and possible further processing may be significant. In addition, we suggested also a novel semi-lossless compression scheme for numerical data appearing in the description of layouts.

References

1. Asraf S., Gross Y., Klein S.T., Revivo R., Shapira D., New compression schemes for natural number sequences, *Discrete Applied Mathematics*, **327** (2023) 18–27.

2. Aviad A., HyperTalmud: a hypertext system for the Babylonian Talmud and its commentaries, MSc. Thesis, Department of Mathematics and Computer Science, Bar-Ilan University, Israel (1993).
3. Borgendale K.W., Dobkin D.J., System and method for editing a structured document to preserve the intended appearance of document elements, U.S. Patent 5,276,793, Jan. 4, 1994.
4. Bookstein A., Klein S.T., Ziff D.A., A Systematic Approach to Compressing a Full-Text Retrieval System, *Information Processing and Management* **28**(6) (1992) 795–806.
5. Boyer R.S., Moore J.S., A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.
6. Choueka Y., Fraenkel A.S., Klein S.T., Compression of Concordances in Full-Text Retrieval Systems, *Proc. 11-th ACM-SIGIR Conf.,* Grenoble (1988) 597–612.
7. Ertl, A.M., What is the PDF format good for?, Vienna University of Technology, https://www.complang.tuwien.ac.at/anton/why-not-pdf.html, 2024.
8. Fiala E.R., Greene D.H., Data Compression with Finite Windows, *Communications of the ACM* **32**(4) (1989) 490–505.
9. Fruchtman A., Gross Y., Shapira D., Klein S.T., Systems and methods of data compression, U.S. Patent 11,722,148, Aug. 8, 2023.
10. Fukui M., Iwai I., Doi M., Takebayashi Y., Method and apparatus for editing documents, U.S. Patent 5,179,650, Jan. 12, 1993.
11. Gibson D.K., Graybill M.D., Apparatus and method for very high data rate compression incorporating lossless data compression and expansion utilizing a hashing technique, U.S. Patent 5,049,881, Sep. 17, 1991.
12. Hart P.E., Chilton J.K., Peairs M., Document layout using tiling, U.S. Patent 5,553,217, Sep. 3, 1996.
13. Hayashi N., Saito K., Document layout processing method and device for carrying out the same, U.S. Patent 5,379,373, Jan. 3, 1995.
14. Hernandez I.H., Barker B.A., Machart B.H., Flow attribute for text objects, U.S. Patent 4,723,209, Feb. 2, 1988.
15. Kataoka M., Usami Y., Yamada M., Harada K., Hori C., Layout displaying apparatus for a word processor, U.S. Patent 5,287,445, Feb. 15, 1994.
16. Klein S.T., *Basic Concepts in Data Structures*, Cambridge University Press, Cambridge, 2016.
17. Klein S.T., *Basic Concepts in Algorithms*, World Scientific Publishers, New Jersey, 2021.
18. Klein S.T., Efficient Optimal Recompression, *The Computer Journal* **40**(2/3) (1997) 117–126. Efficient optimal data recompression method and apparatus, U.S. Patent 5,392,036, Feb. 21, 1995.
19. Klein S.T., Bookstein A., Deerwester S., Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** (1989) 230–245.
20. Klein S.T., Shapira D., A New Compression Method for Compressed Matching, *Proc. Data Compression Conference (DCC)* (2000) 400–409.
21. Klein S.T., Shapira D., Compressed Matching in Dictionaries, *Algorithms*, **4**(1) (2011) 61–74.
22. Klein S.T., Shapira D., Pattern matching in Huffman encoded texts, *Information processing & management*, **41**(4) (2005) 829–841.
23. Knuth D.E., *TEX and METAFONT: New directions in typesetting*, American Mathematical Society, Providence, RI (1979).
24. Mason C.A., Document processing method and system, U.S. Patent 5,214,755, May 25, 1993.
25. Morgan M.W., Method and apparatus for representing bordered areas of a generic form with records, U.S. Patent 5,208,906, May 4, 1993.
26. https://en.wikipedia.org/wiki/Bibliothèque_de_la_Pléiade
27. https://pymupdf.readthedocs.io/en/latest/
28. Wallace G.K., The JPEG still picture compression standard, *Comm. of the ACM* **34**,4 (April 1991) 31–44.
29. Welch T.A., A Technique for High-Performance Data Compression, *Computer* **17**(6) (1984) 8–19.
30. Whiting D.L., George G.A., Ivey G.E., Data compression apparatus and method, U.S. Patent 5,016,009, May 14, 1991.
31. Wong H.H., Chan V.W., Compact memory for mixed text in graphics, U.S. Patent 5,457,776, Oct. 10, 1995.

32. Zavadskyi I., Klein S.T., Shapira D., Word-Based Forward Coding, *Proc. Data Compression Conf. (DCC)*, Snowbird, Utah (2024) 351–361.
33. Ziv J., Lempel A., A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT-23** (1977) 337-343.
34. Ziv J., Lempel A., Compression of individual sequences via variable rate coding, *IEEE Trans. on Inf. Th.* **IT-24** (1978) 530–536.

Towards Efficient k-Mer Set Operations via Function-Assigned Masked Superstrings*

Ondřej Sladký[1,2], Pavel Veselý[2], and Karel Břinda[3]

[1] ETH Zürich, Switzerland ondra.sladky@gmail.com
[2] Computer Science Institute of Charles University, Prague, Czech Republic
vesely@iuuk.mff.cuni.cz
[3] Inria, Irisa, Univ. Rennes, Rennes, France karel.brinda@inria.fr

Abstract. The design of efficient dynamic data structures for large k-mer sets belongs to central challenges of sequence bioinformatics. Recent advances in compact k-mer set representations via Spectrum-Preserving String Sets (SPSS), culminating with the masked superstring framework, have provided data structures of remarkable space efficiency for wide ranges of k-mer sets. However, the possibility to perform set operations with the resulting indexes has remained limited due to the static nature of the underlying compact representations. Here, we develop f-masked superstrings, a concept combining masked superstrings with custom demasking functions f to enable k-mer set operations based on index merging. Combined with the FMSI index for masked superstrings, we obtain a memory-efficient k-mer membership index and compressed dictionary supporting set operations via Burrows-Wheeler Transform merging. The framework provides a promising theoretical solution to a pressing bioinformatics problem and highlights the potential of f-masked superstrings to become an elementary data type for k-mer sets.

Keywords: k-mer sets, data structures, set operations, masked superstrings

1 Introduction

To store and analyze the vast volumes of DNA sequencing data [41], modern bioinformatics methods increasingly rely on *k-mers*, k-long substrings of genomic data. As k-mer-based methods bypass the computationally expensive sequence alignment, they become increasingly popular in data-intense applications such as large-scale data search [5,13,23], metagenomic classification [40,11], infectious disease diagnostics [6,12], or transcript abundance quantification [7,30]. All these applications rely on indexing collections of k-mer sets using advanced data structures [26], which often rely at their core on data structures for single k-mer sets [15].

A central challenge in contemporary sequence bioinformatics is to design single-k-mer-set data structures with two qualities: scalability and dynamism. On one hand, as k-mer sets can be large, possibly exceeding billions of distinct k-mers [23], we need data structures that are efficient both in space and time simultaneously, with an internal adaptability to different types of k-mer sets. On the other hand, as modern genomic databases undergo rapid development, due to rapidly growing content as well as databases curation, we also need to perform efficient updates across k-mer indexes

* Research supported by the French National Research Agency (ANR) under Grant ANR-24-CE45-1226 for the REALL project (KB), by the Czech Science Foundation project 22-22997S (OS, PV), by the ERC-CZ project LL2406 of the Ministry of Education of the Czech Republic (PV), by the Center for Foundations of Modern Computer Science (Charles Univ. project UNCE 24/SCI/008, PV), and by Campus France under PHC BARRANDE 2025 grant n° 52374TC for the EFFIMAS project (KB, PV).

Ondřej Sladký, Pavel Veselý, Karel Břinda: Towards Efficient k-Mer Set Operations via Function-Assigned Masked Superstrings, pp. 26–40.
Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9

to avoid repetitive and costly index recomputations. This includes scenarios such as k-mer set operations across sets, and additions or removals of individual k-mers.

A substantial advance in the scalability challenge was achieved using the concept of Spectrum-Preserving String Sets (SPSS) [16,9,10,32,35,36]. k-mers in genomic k-mer sets tend to be highly non-independent [14] – they typically correspond to sets of k-long substrings of a small number of (potentially long) strings, a property known as the *spectrum-like property (SLP)* [15]. This gave rise to textual k-mer set representations that correspond to path covers of de Bruijn graphs, which we will collectively refer to as SPSS [16,9,10,32,35,36] and which are now standard and widely used across data structures (e.g., in [27,2,31]).

Masked superstrings (MS) provided additional space gains and structural adaptability by virtue of a better k-mer set compaction [37]. The core improvement over SPSS lies in modeling the structure of k-mer sets by overlap graphs instead of de Bruijn graphs, thus being able to exploit overlaps of *any* length. MS represent k-mer sets using an approximately shortest superstring of all k-mers and a binary mask to avoid false positive k-mers. MS generalize any existing SPSS representation as these can always be encoded as MS, but provide further compression power, especially for non-SLP data arising from sketching or subsampling. The resulting representation is well indexable using a technique called Masked Burrows Wheeler Transform [39], resulting in a k-mer data structure with $2 + o(1)$ bits per k-mer under the SLP.

However, the lack of dynamism of both SPSS and MS – and of their derived data structures – has been limiting their wider applicability. To the best of our knowledge, the only supported operations were union, either via merging SPSS of several k-mer sets resulting in an SPSS of their union, or by the Cdbg-Tricks [19] to calculate the union unitigs from the unitigs of multiple original k-mer sets. However, other operations besides union, such as intersection or symmetric difference, have never been considered.

Here, we develop a dynamic variant of masked superstrings (and thus also of SPSS) called the *f-masked superstrings (f-MS)*. The key idea is to equip masked superstrings with so-called demasking functions f for more flexible mask interpretation (**Sec. 3**). When complemented with the concatenation and mask recast operations (**Sec. 4.1**) and possibly compaction (**Sec. 4.2**), this provides support for union, intersection, and symmetric difference and thus any set operation with k-mer sets, resulting in a complete algebraic type for k-mer sets (**Sec. 4.3**). We demonstrate the applicability of the concept on the FMSI index for masked superstrings [39] (**Sec. 5**) and provide a proof-of-concept implementation in the FMSI program (**Sec. 6**).

1.1 Problem formulation

We focus on the problem of developing a time- and space-efficient index for k-mer sets with support for set operations.

(i) **Index construction**, with time complexity linear in the number of k-mers.
(ii) **Membership queries**, with low bits-per-k-mer memory requirements, approaching 2 bits per distinct k-mer for datasets following the SLP [15].
(iii) **Set operations**, including union, intersection, difference, and symmetric difference.

1.2 Related Work

Many works have recently focused on data structures for single k-mer sets and their collections; we refer to [15,26] for recent surveys. Here, we primarily focus on those that are exact and offer some kind of dynamicity, i.e., an efficient support for set operations or, at least, insertions/deletions of individual k-mers. The recently introduced Conway-Bromage-Lyndon (CBL) structure [28] builds on the work of Conway and Bromage [17] on sparse bit-vector encodings and combines them with smallest cyclic rotations of k-mers (a.k.a. Lyndon words), which yields a dynamic and exact k-mer index supporting set operations, such as union, intersection, and difference, as well as insertions or deletions of k-mers.

While, to the best of our knowledge, other k-mer indexes do not support efficient set operations such as the intersection or difference, other tools, including BufBoss [1], DynamicBoss [3], and FDBG [18] allow for efficient insertions and deletions of individual k-mers. Bifrost [20], VARI-merge [29], Metagraph [23], or the very recent Cdbg-Tricks [19] support insertions but not deletions. We note that there are many more highly efficient but static data structures for individual or multiple k-mer sets, e.g., [4,25,2]. In particular, one can combine SPSS representations with efficient full-text search [16,32,10] or hashing [31], but this has so far yielded only static indexes.

Another line of work focused on k-mer counting, where we additionally require to compute the k-mer frequencies. Although some counters [22,24,33] support operations such as union, intersection, or difference on k-mer multisets, counting typically requires comparatively larger memory and heavy disk usage compared to efficient k-mer indexes.

Baseline approaches for performing set operations. We note that one can add support for set operations to a static data structure in a straightforward way: One option is extracting the k-mer sets from the input indexes, performing the given operation with the sets, and computing the new index for the resulting set; however, this process requires substantial time and memory. Another option is to keep the indexes for input k-mer sets and process a k-mer query on the set resulting from the operation by asking each index for the presence/absence of the k-mer in each input set, which is, however, time and memory inefficient as all the indexes need to be loaded into memory, as also noted in [21]. Therefore, we seek to perform set operations without the costly operation of recomputing the index or making multiple queries to original indexes.

2 Preliminaries

Strings and k-mers. We use constant-size alphabets Σ, typically the nucleotide alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ (unless stated otherwise). The set Σ^* contains all finite strings over Σ, with ε representing the empty string. A substring of S is a contiguous sequence of characters within S. For a given string $S \in \Sigma^*$, $|S|$ denotes its length, and $|S|_c$ the number of occurrences of the letter c in S. For two strings S and T, $S + T$ denotes their concatenation. A *k-mer* is a k-long string over Σ. For a string S and a fixed length k, the k-mers *generated* by S are all k-long substrings of S, and similarly for a set of strings $\mathcal{R}$, they are those generated by the individual strings $S \in \mathcal{R}$. We present our framework in the uni-directional model, where all k-mers are treated as distinct (unless stated otherwise).

Masked superstrings. Given a k-mer set K, a *masked superstring* (MS) [37] consists of a pair (S, M), where S is an arbitrary superstring of the k-mers in K and M is a binary mask of the same length. An occurrence of a k-mer in an MS is said to be ON if there is `1` at the corresponding position in the mask (i.e., the initial position of the occurrence), and OFF otherwise. The set of k-mers generated by S are referred to as the *appearing k-mers*, and only those that have at least one ON occurrence are *represented k-mers*, i.e. those from the represented set K. All masks M that represent a given K in a combination with a given superstring are called *compatible*.

Occurrence function. Let (S, M) be an MS and suppose that our objective is to determine whether a given k-mer Q is among the MS-represented k-mers. Conceptually, this process consists of two steps: (1) identify the starting positions of occurrences of Q in S, and (2) verify using the mask M whether at least one occurrence of Q is ON. We can formalize this process via a so-called *occurrence function*.

Definition 1. *For a superstring S, a mask M, and a k-mer Q, the* occurrence function $\Lambda(S, M, Q) \to \{0,1\}^*$ *is a function returning a finite binary sequence with the mask symbols of the corresponding occurrences, i.e.,*

$$\Lambda(S, M, Q) := \left(M_i \,\middle|\, S_i \cdots S_{i+k-1} = Q\right) . \tag{1}$$

In this notation, verifying k-mer presence corresponds to evaluating the composite function '$\mathbf{or} \circ \Lambda$'; that is, a k-mer is present if $\Lambda(S, M, Q)$ is non-empty and the logical **or** operation on these values yields 1. Thus, the set of all MS-represented k-mers can be expressed as

$$K = \{Q \in \Sigma^k \mid \mathbf{or}(\Lambda(S, M, Q)) = 1\}. \tag{2}$$

Example 2. Consider the k-mer set $K = \{\texttt{ACG}, \texttt{GGG}\}$. One possible superstring is `ACGGGG`, with three compatible masks: `101100`, `100100`, `101000`, resulting in the mask-cased encodings `AcGGgg`, `AcgGgg`, `AcGggg`, respectively. Given a k-mer `GGG` and masked superstring `AcgGgg`, $\Lambda(S, M, Q) = (0, 1)$ and $\mathbf{or}((0, 1)) = 1$, therefore `GGG` is considered represented.

3 Function-Assigned Masked Superstrings

This work is based on two important observations on masked superstrings in the context of their applications for k-mer-set indexing.

First, **or** is one member of a large class of functions that could be used to demask k-mers in masked superstrings: for instance, MS could have been defined using the **xor** function, with a k-mer considered present if the number of its ON occurrences is odd. In fact, any symmetric Boolean function k-mer demasking can serve the role.

Second, different data structures may impose different constrains on f-masked superstrings, and this can be embedded directly into demasking functions via a special return value, `invalid`. Additionally, we treat the non-appearing k-mers as non-represented by requiring $f(\varepsilon) = 0$.

Definition 3. *We call a symmetric function $f : \{0,1\}^* \to \{0, 1, \texttt{invalid}\}$ with $f(\varepsilon) = 0$ a k-mer* demasking function.

We now generalize the concept of MS into so-called f-masked superstrings (f-MS).

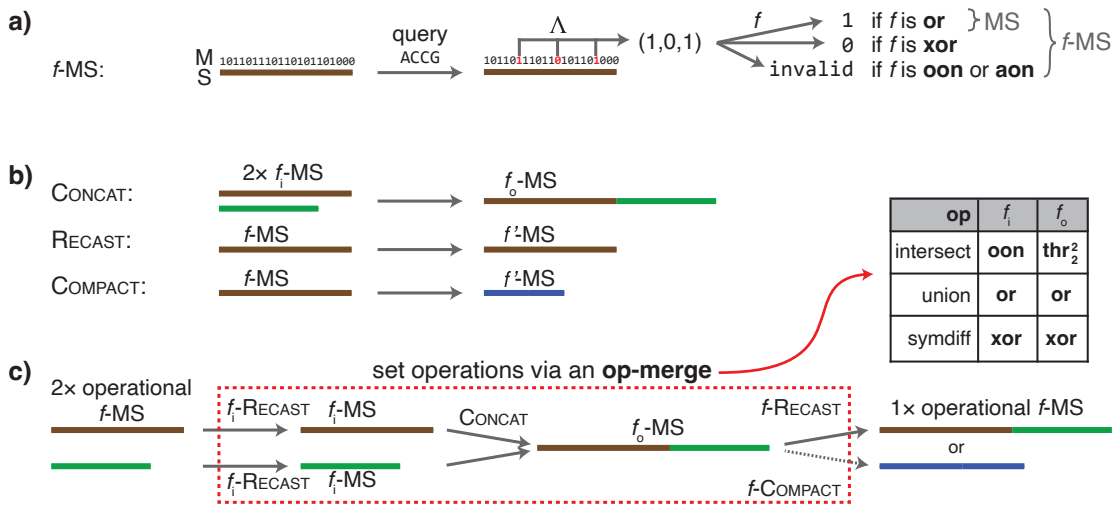


Figure 1: **Overview of the f-MS framework. a) The concept of f-MS.** For a given k-mer, the corresponding mask bits $\Lambda(S, M, Q)$ are evaluated using a demasking function f. **b) Low-level operations.** A $f \to f'$ RECAST changes mask under function f to another mask under function f', while preserving the represented k-mer set. CONCAT merges two superstrings and masks. Both operations may conceptually be performed either on the original f-MS or on their associated indexes. **c) Set operations.** An operation **op** is performed by a sequence of CONCAT and RECAST applied to the input f-MS, with operation-specific input and output functions (see Tab. 1). The RECAST, which serves for keeping the f-MS operational for its data structure, may be replaced by compaction with the same target function.

Definition 4. *Given a demasking function f, a superstring S, and a binary mask M, such that $|M| = |S|$, we call a triplet $\mathcal{S} = (f, S, M)$ a* function-assigned masked superstring *or* f-masked superstring*, and abbreviate it as* f-MS.
If $f(\Lambda(S, M, Q)) =$ `invalid` for any k-mer Q, we call the f-MS invalid.

Now, for a valid f-MS, we generalize Equation (2) for k-mer decoding as

$$K = \{Q \in \Sigma^k \mid f(\Lambda(S, M, Q)) = 1\}. \tag{3}$$

We note that by the validity requirement, f imposes structural constraints on the valid masks, e.g., **oon** requires that only a single occurrence of a represented k-mer is ON and all other occurrences are OFF. Still, there may be many valid masks, leaving room for an application-specific mask optimization. In Table 1 we overview f-MS used throughout the paper. See also Figure 1a for an example evaluation of k-mer presence for sample f-MS.

Example 5. Consider Example 2 with the set of 3-mers $K = \{\texttt{ACG}, \texttt{GGG}\}$ and the masked superstring `AcGGgg`. For the query k-mer $Q =$ `GGG`, the occurrence function gives $\Lambda(S, M, Q) = (1, 1)$. The result of demasking $f(\Lambda(S, M, Q))$ then depends on the specific function f: for **or** it evaluates to 1 and thus `GGG` is represented; however, for **xor** the result would be 0 and `GGG` would not be represented.

Operational f-MS with respect to different applications. Different use cases may require different properties of masks. Since the choice of a demasking function

Fn f	**Definition**	**Comprehensive**	**Use cases**
or	`1` if $\lvert\Lambda\rvert_1 > 0$ `0` if $\lvert\Lambda\rvert_1 = 0$	yes	• f for MS (**Sec. 3**), (r)SPSS (full version) • f_i, f_o for union (**Sec. 4.3**)
xor	`1` if $\lvert\Lambda\rvert_1$ is odd `0` if $\lvert\Lambda\rvert_1$ is even	yes	• f_i, f_o for sym. difference (**Sec. 4.3**)
$\mathbf{thr}_a^b$	`1` if $a \leq \lvert\Lambda\rvert_1 \leq b$ `0` otherwise	iff $a = 1$	• $[a, b]$-threshold • f_o for intersection (**Sec. 4.3**)
oon	`1` if $\lvert\Lambda\rvert_1 = 1$ `0` if $\lvert\Lambda\rvert_1 = 0$ `invalid` otherwise	yes	• One-or-Nothing (corresponds to min-1 masks) • f in FMSI-dict queries (**Sec. 3**) • f_i for intersection (**Sec. 4.3**)
aon	`1` if $\Lambda \neq \epsilon \wedge \lvert\Lambda\rvert_0 = 0$ `0` if $\lvert\Lambda\rvert_1 = 0$ `invalid` otherwise	yes	• All-or-Nothing (corresponds to max-1 masks) • f in FMSI-memb queries (**Sec. 3**)

Table 1: **Overview of demasking functions.** $\Lambda(f, S, M)$ is abbreviated as Λ. Only comprehensive functions can be a target of recasting.

can be also viewed as a restriction of the set of acceptable masks for a given superstring and k-mer set, we can choose different demasking function to enforce certain properties of the masks. We call the demasking functions that enforce the desired properties of masks as *operational*. The concept of operational demasking functions plays a central role in the whole f-MS framework, see Figure 1c.

As an example, considering the FMSI index [39], for its version for membership queries, only **aon** is operational. This is due to a certain query speed optimization. If this optimization is omitted, all demasking functions are operational. For the dictionary version, only **oon** is operational. For more details, see Section 5.

4 Abstract algebraic framework for *k*-mer set operations

The dynamization of a data structure can be conceptually split into two parts – the dynamization of the representation and the dynamization of the data structure internals itself. Here, we give a general approach to dynamize the representation, based on two low-level operations for their manipulation – mask recasting and concatenation, where we assume that the concatenation in the data structure can be done efficiently. In the context of different data structures, these can have very different forms, and we discuss a specific variant for BWT in Section 5.

4.1 Concat and Recast: The Two Essential Low-Level Operations

The elementary operations of concatenation and recasting described next are those used to perform set operations, as described in Section 4.3. While concatenation effectively merges two f-MS, recasting facilitates adjusting f-MS into the desired form, e.g., before executing a set operation or making it operational after performing the operation. See also Figure 1 for conceptual overview of these operations.

Concat For full generality, we define concatenation on f-MS as concatenating the underlying superstrings and masks for all possible input and output functions f.

Definition 6. *Given f-MS (f_i, S_1, M_1) and (f_i, S_2, M_2), we define the (f_i, f_o)-concatenation as the operation taking these two f_i-MS and producing the result $(f_o, S_1 + S_2, M_1 + M_2)$.*

Note that Definition 6 can be easily extended to more than two input f-MS. In the case that all the functions are the same, i.e., $f = f_i = f_o$, we call it *f-concatenation* or just *concatenation* if f is obvious from the context.

Furthermore, note that while the set of appearing k-mers of S_1+S_2 clearly contains the union of appearing k-mers of S_1 and of S_2, additional new occurrences of k-mers may appear at the boundary of the two superstrings. These newly occurring k-mers may not be appearing in any of the superstrings S_1 and S_2. The occurrences of appearing k-mers of $S_1 + S_2$ that overlap both input superstrings are called *boundary occurrences*.

Example 7. Consider **aon**-MS `AcGGgg`, representing 3-mers $K_1 = \{\texttt{ACG}, \texttt{GGG}\}$, and **aon**-MS `Ggg`, representing $K_2 = \{\texttt{GGG}\}$. Their **aon**-concatenation is `AcGGggGgg`, with two boundary occurrences of GGG; causing the result to be an invalid **aon**-MS even though the input **aon**-MS were valid. Similarly, boundary occurrences of k-mers can change the presence in the result for some functions, illustrating the need of a careful treatment of concatenation.

Recast When performing operations with f-MS, recasting is used to change the demasking function f without altering the represented k-mer set and the underlying superstring; here, we note that recomputing the superstring is the most resource-demanding operation [37].

In many cases, recasting triggers recomputation of the mask, which is however a less expensive operation. We first note that recasting to an arbitrary demasking function may not be possible with a fixed superstring as there is no valid mask, e.g., for a function that requires the represented k-mers to have at least two ON occurrences. Nevertheless, recasting is possible for a large class of demasking functions, specifically functions with the property that all appearing k-mers are encodable using a compatible mask as present or absent, irrespective of the number of their occurrences; we call such functions *comprehensive*:

Definition 8. *A demasking function f is* comprehensive *if for every $n > 0$, there exist $x, y \in \{0,1\}^n$ such that $f(x) = 0$ and $f(y) = 1$.*

The definition directly implies the possibility of recasting an f-MS by changing the function f to any comprehensive f'. This also implies an efficient initialization procedure to create a desired operational f-MS for a comprehensive f: First, compute an **or**-MS as described in [37] and then recast it to f.

Lemma 9 (recasting). *Let (f, S, M) be a valid f-MS representing a k-mer set K. Then, for every comprehensive demasking function f', there exists a valid mask M' such that (f', S, M') represents K.*

Proof. Since $f(\varepsilon) = 0$ for any demasking function f, we know that non-appearing k-mers are not represented in the f-MS (f, S, M) and will correctly be non-represented in the resulting f'-MS. For any appearing k-mer Q in S, we find all of its $n_Q > 0$ occurrences and then use Definition 8 to set the mask bits of M' at these occurrences either to x with $f(x) = 0$ if $Q \notin K$, or to y with $f(y) = 1$ otherwise (since f is symmetric by Definition 3, the order does not matter).

The proof gives a general algorithm for recasting to any comprehensive function as for each appearing k-mer we can check whether it is represented in the f-MS of origin, and find the number of ON occurrences for it to be represented (or not) in the new f-MS. Moreover, for all comprehensive functions mentioned in Table 1, recasting can be done either by maximizing the number of 1s in the mask (**aon**), or by minimizing the number of 1s (all other comprehensive functions in Table 1). Both can be achieved in linear time, namely using a two-pass algorithm for max-ones or a single-pass algorithm for min-ones, as described in [37].

4.2 Compact: an Optional Low-Level Operation

After performing a number of set operations, the resulting superstring may be much longer than the size of the k-mer set it represents. Then it is desirable to change also the superstring alongside recasting the mask. We call this operation of changing (f, M, S) into (f', M', S'), while preserving the represented set K, *compaction*, and typically require $|S'| < |S|$. Lemma 9 implies that if f' is comprehensive we can use any superstring of K as our S', and therefore compact can be performed by any algorithm for superstring computation. We give an example of how compaction can be performed on a BWT-based index in Section 5.

4.3 k-Mer Set Operations via f-MS and its Low-Level Operations

Union. As implicitly shown in [37], concatenating MS, which are **or**-MS in our notation, acts as union on the represented sets; that is, the resulting represented set is the union of the original represented sets. Specifically, the boundary occurrences of k-mers do not affect the result by the definition of **or**. This allows **or**-MS to generalize SPSS representations, since any set of k-mers in the SPSS representation can be directly viewed as an **or**-MS by concatenating the individual simplitigs/matchtigs.

We show that **or** is the only comprehensive demasking function that acts as union on the represented sets, up to the interchange of the meaning of 0s and 1s; see Supplementary Materials [38] for details. We further demonstrate this uniqueness even when the concatenated masked superstrings directly correspond to individual matchtigs and, therefore, **or**-MS are the only f-MS that generalize SPSS representations.

Symmetric difference. Next, we observe that **xor** naturally acts as the symmetric difference set operation, i.e., concatenating two **xor**-MS results in an **xor**-MS representing the symmetric difference of the original sets. Indeed, recall that using **xor** implies that a k-mer is represented if and only if there is an odd number of ON occurrences of that k-mer. Observe that the boundary occurrences of k-mers do not affect the resulting represented set as those have zeros in the mask. Thus, if a k-mer is present in both sets, it has an even number of ON occurrences in total and hence, it is not represented in the result. Likewise, if a k-mer belongs to exactly one input set, it has an odd number of ON occurrences in this input set and an even number (possibly zero) in the other; thus, it is represented in the result. As any appearing k-mer is either boundary or appears in one of the input MS, the result corresponds to the symmetric difference.

Intersection. After seeing functions for union and symmetric difference operations, it might seem natural that there exists a comprehensive function f such that f-concatenation yields intersection. This is however not the case as there exists no

comprehensive demasking function acting as intersection, see Supplementary Materials [38]. We can circumvent the non-existence of a single demasking function acting as intersection by using a different function on the output than on the input. To this end, we will need two different demasking functions:

- The **oon** (abbreviation of one-or-nothing) function is a demasking function that returns 1 if there is exactly one 1 in the input, 0 if there are no 1s, and `invalid` if there is more than a single ON occurrence of the k-mer.
- The $\mathbf{thr}_a^b$ function (an abbreviation of threshold), where $0 < a \leq b$, is a demasking function that returns 1 whenever it receives an input of at least a ones and at most b ones and 0 otherwise. Note that unless $a = 1$, $\mathbf{thr}_a^b$ functions are not comprehensive. The corresponding f-MS are denoted $\mathbf{thr}_a^b$ -MS.

We use **oon**-MS to represent the input masked superstrings and treat the result of the concatenation as $\mathbf{thr}_2^2$-MS, that is, we apply (**oon**, $\mathbf{thr}_2^2$)-concatenation. Since the represented k-mers are those that have precisely one ON occurrence in the original **oon**-MS, they are those with two ON occurrences in the result and therefore are correctly recognized by the $\mathbf{thr}_2^2$ function.

Further generalizations and optimization. Other set operations can be performed by chaining intersection, union, and symmetric difference. For example, the asymmetric difference $A - B$ can be obtained as $A \Delta (A \cap B)$. Such an approach is possible for *any set operation* on *any number of input sets*.

Furthermore, the approach with **oon**-MS for intersection can be utilized to obtain more efficient computation of any symmetric operation for any number of input sets. If we represent each input as **oon**-MS, then after concatenation of all of them, the number of ON occurrences counts the number of sets where the k-mer appeared and so the symmetric set operation can be recognized with appropriate demasking function. Furthermore, the same such mask and superstring can be used for different operations, e.g., for intersection and union at once, just with different f.

Finally, there are more demasking functions that could be used as operational in other use cases. For instance, the **and** function, for which an appearing k-mer is represented if all its occurrences are ON, could be used to allow ON occurrences of non-represented k-mers, potentially making the masks of **and**-MS better compressible.

5 Example Application: Dynamization of the FMSI Index

The f-MS framework may be used for the dynamization of many different data structures. Here, we will demonstrate such a procedure on the FMSI index introduced in our concurrent work [39]. The FMSI index [39] for a k-mer set K is constructed from its masked superstring (M, S) either maximizing the number of ones for membership queries (FMSI-memb), or minimizing them for dictionary queries (FMSI-dict). The FMSI index consists of the Burrows-Wheeler transform (BWT) [8] of S with an associated rank data structure, and the *SA-transformed mask* M' which is a bit-vector of the same length as S, where $M'[i] = M[j_i - 1 \mod |M|]$ where j_i is the starting position of the lexicographically i-th suffix of S. FMSI can be constructed in linear time through Masked BWT [39], a tailored variant of the classical BWT [8]. FMSI can index a k-mer Q in $O(k)$ time by first computing the range of occurrences of Q in the suffix-array coordinates. Here, we only consider the most memory-efficient version of FMSI index, which requires $2 + o(1)$ bits of memory per distinct k-mer

under the spectrum-like property and at most $3 + o(1)$ bits per superstring character in the general case [39]. In addition, we consider the rank data structure also for the SA-transformed mask, which does not asymptotically increase complexity, that is, costs only another $o(1)$ bits per superstring character.

5.1 Step 1: Extending the FMSI index to f-MS

The first step to an f-MS dynamization of a data structure is to identify its operational demasking function. For FMSI-memb and FMSI-dict, this is **aon** and **oon**, respectively, but the implementation can be generalized for any demasking function to be operational. Afterwards, we simply implement the evaluation of the operational demasking functions, possibly with some pre- and post-processing which may be specific to the operation and to the internals of the data structure.

Efficient membership queries with arbitrary demasking functions. As FMSI-memb retrieves only the mask symbol at an arbitrary occurrence, it requires all the symbols of a k-mer to be the same. Thus, the only comprehensive operational f-MS is **aon**-MS, which returns 1 if it receives a list of ones, 0 is a list without a one, and `invalid` otherwise (see Table 1). This f-MS corresponds to **or**-MS with maximized number of ones.

However, the search in FMSI can be easily extended to return all occurrences of a k-mer. In such case, all possible demasking functions are operational. In Lemma 10, we demonstrate that if the function can be evaluated efficiently, k-mer presence in FMSI can be determined in constant time after performing backwards search.

Lemma 10. *Consider a query for k-mer Q on an f-MS (f, S, M) representing a k-mer set K, such that f can be computed in $O(1)$ from the number of* ON *and* OFF *occurrences of Q. Let M' be the corresponding SA-transformed mask [39] and assume we know the range $[i, j)$ of sorted rotations of S starting with a k-mer Q. Then the presence or absence of Q in K can be determined in $O(1)$ time.*

Proof. From [39, Lemma 1], $M'[x]$ for $x \in [i, j)$ corresponds to the mask symbol of a particular occurrence of Q. Therefore, $|\Lambda(S, M, Q)|_1 = \mathrm{rank}_1(M', j) - \mathrm{rank}_1(M', i)$, which can be computed in $O(1)$ time using two rank queries on the mask; here, $\mathrm{rank}_1(M', i) = \sum_{a=0}^{i-1} M'[a]$ is the number of ones on coordinates $0, \ldots, i-1$ in M', computed by the rank data structure. Furthermore, $|\Lambda(S, M, Q)|_0 = |\Lambda(S, M, Q)| - |\Lambda(S, M, Q)|_1 = j - i - |\Lambda(S, M, Q)|_1$. Since f is commutative, $f(\Lambda(S, M, Q))$ can be evaluated from the two quantities in constant time.

Dictionary queries. For dictionary queries, FMSI requires the mask to contain only a single ON occurrence of a k-mer. This exactly corresponds to **oon**-MS (introduced in Section 4.3, see Table 1). After a recast to **oon**-MS, dictionary queries work in the same way as in FMSI-dict by retrieving the rank in the SA-transformed mask of the first lexicographic occurrence [39].

5.2 Step 2: Adding Concat and Recast

Recast. Recasting of an f-MS can be done directly in SA coordinates without additional memory. For each position of BWT that was not previously visited, we first retrieve the k-mer by traversing the inverse LF mapping [39, Alg. 3] and then obtain the range of this k-mer. Then from Lemma 9, we find the number of ones needed

to make this k-mer represented if it was represented in f-MS of origin. This takes time $O(|S| + k|K'|)$, where K' are all appearing k-mers. Then, we need to update the auxiliary data structures (rank and select), possibly by recomputing them.

Concat. Performing the operation on indexes boils down to merging two BWTs using any algorithm for BWT merging, for example [21] which runs in linear time. To merge the SA-transformed masks, we attach the mask symbols to the corresponding characters of BWT, in the same way as for FMSI construction [39, Alg. 1] and perform BWT merge; hence, the existing algorithms for BWT merging can be adjusted in a straightforward way to merge the SA-transformed masks. The BWT merging is then followed by recomputation of the ranks necessary for the functioning of FMSI.

5.3 Step 3 (optional): Adding Compact

If an f-MS contains too many redundant copies of individual k-mers, e.g., if an f-MS is obtained by concatenating multiple input f-MS, it might be desirable to *compact* it, i.e., reoptimize its support superstring. This can be done by recasting to **oon**-MS, exporting this **oon**-MS to its superstring form by reversing the LF-mapping, removing all segments of support superstring not containing any ON occurrence of a k-mer, which can be done by single-pass algorithm, and reindexing the result. Although this algorithm requires reindexing of the result, it uses memory linear in the total size of the input superstrings. We leave it as future work to design more efficient algorithms for compaction directly in the FMSI index without the need to export the f-MS.

6 Implementation and Proof-of-Concept Experiments

To demonstrate feasibility of using f-MS for set operations in practice, we developed a prototype implementation of the framework, embedded it into the FMSI k-mer-set index [39], and evaluated it using two selected genomes.

6.1 Implementation in the FMSI tool

We implemented the f-MS framework as an experimental extension of the FMSI program [39] (`https://github.com/OndrejSladky/fmsi`). Our implementation supports any value of k up to 64. In its basic form, FMSI takes an input masked superstring, constructs an FMSI index over it and enables performing membership and dictionary k-mer queries. For membership queries, we extended the original FMSI implementation [39, Alg. 3] to reflect the new Lemma 10. Dictionary queries are supported as in the original implementation from [39].

Set operations follow the workflow described in Figure 1. The CONCAT operation is implemented via FMSI index merging: first, by exporting the individual masked superstrings, then, by concatenating them, and finally by reindexing the newly obtained masked superstring result, with the f_i, f_o functions tracked by the user. The RECAST operation is currently supported for **or**, **xor**, **oon**, and **aon**, and proceeds via three steps: exporting the masked superstring by FMSI, mask re-optimization by KmerCamel [37] (min-1 for **or**, **xor**, **oon**, and max-1 for **aon**), and reindexing the new f-masked superstring by FMSI. The COMPACT operation is supported directly in FMSI, and internally runs the global greedy algorithm [37] to approximate shortest superstring.

6.2 Experiments using selected genomes

We evaluated our framework using the *C. elegans* and *C. briggsae* genomes, with a primary focus on the memory requirements for processing membership queries. First, we indexed the initial **or**-MS obtained by KmerCamel's global greedy algorithm [37]. Then, we constructed the indexes for source genomes with $k = 21$ and tested the union, intersection, and symmetric difference operations of the corresponding k-mer sets; we note that $k = 21$ is already sufficiently large to capture characteristics of genomes and similar values are typically used for minimizer indexing [31]. Finally, we evaluated the associated quantitative characteristics, including the superstring length of each computed f-MS. Additionally, we measured the memory requirements to perform queries on the indexed f-MS before and after concatenation, both without and with compaction (measured by GNU time), and also compared FMSI to CBL [28] (commit `328bcc6`, 28 prefix bits). More details are in Supplementary Materials [38].

The starting memory requirements of membership queries of the FMSI and CBL indexes were approximately 2.7 and 83 bits per distinct k-mer, respectively, for both of the genomes. After computing the union, this decreased slightly to about 2.5 bits-per-distinct k-mer in the union for FMSI, and to about 59 bits for CBL (although still over 20 times more than FMSI). The results for symmetric difference were similar, with FMSI requiring 2.6 bits per distinct k-mer and CBL 59 bits per distinct k-mer.

However, for intersection, which contains only less than 1% of the original k-mers, FMSI's memory consumption increased significantly to 524 bits per distinct k-mer, even underperforming CBL whose performance increased to about 420 bits. This is caused by the resulting superstring length being much larger than the represented set, in which case compaction is necessary; indeed, then the memory requirements decreased to 43 bits per k-mer (almost 10 times less than CBL). Moreover, in this case, the compaction was relatively cheap thanks to the small size of the resulting k-mer sets compared to the original one. We illustrate this in Supplementary Materials [38].

Overall, this experiment demonstrates that if the percentage of k-mers in the result is high, FMSI maintains its superior memory efficiency over CBL even after index merging, without the need for compaction. If the percentage of k-mers in the result is very low, compaction needs to be applied in order for FMSI to maintain low memory footprint.

7 Conclusion and Outlook

We have proposed f-masked superstrings (f-MS) as an algebraic data type for k-mer sets that allows for seamless execution of set operations. It is primarily based on equipping masked superstrings (MS) from [37] with a demasking function f, and we have thoroughly investigated several natural demasking functions, demonstrating that set operations on k-mer sets can be carried out simply by masked superstring concatenation or, if indexed, by merging their masked Burrows-Wheeler transform from [39]. This leads to a simple data structure that simultaneously allows for beyond worst-case compressibility, answering exact membership queries, and efficiently performing set operations on the k-mer sets, without the costly operation of recomputing the underlying representation. Another major advantage is the versality of our concept as it can in fact be combined with (repetitive) Spectrum Preserving String Sets [10,32,36] instead of (more general) masked superstrings. For instance, our approach can be applied to SPSS-based indexes, such as SSHash – simplitigs of k-mers

split based on minimizers can be treated as f-MS and the same Concat and Recast operations can be performed on those to achieve dynamicity.

The main practical limitation of our work is the current implementation of the index merging, which is very slow and not using the state-of-the-art algorithms for BWT merging [21]. Furthermore, our proof-of-concept experiment is only meant to demonstrate feasibility, and we leave a more thorough evaluation, using various datasets and including a thorough comparison to other tools for set operations, such as CBL [28], to future work.

Our work opens up several research directions for future theoretical investigation. On the algorithmic level, our work relies on efficient algorithms for merging the Burrows-Wheeler transform. Additionally, we seek an algorithm for mask recasting that is not only very space-efficient, but also has faster running time, which can possibly be achieved by using additional data structures for the index, such as the kLCP array [34]. Moreover, it is open how to directly perform compaction with f-masked superstrings indexed with the masked Burrows-Wheeler transform [39], that does not necessitate to reverse the BWT and export the masked superstring, but rather work locally with the BWT of the superstring and the SA-transformed mask. Furthermore, as our work deals only with set operations, we open the question of performing single insertions and deletions in a more efficient way than performing these through set operations; note that for comprehensive demasking functions, deletions can be handled efficiently by changing the corresponding mask bits.

In conclusion, while the primary contributions of this paper are conceptual, they pave the path towards a space- and time-efficient library for k-mer sets that would include all of these features. In the light of advances in efficient superstring approximation algorithms and BWT-based indexing, we believe that the f-masked superstring framework is a useful step towards designing appropriate data structures for this library, which is now mainly an engineering challenge.

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. J. N. Alanko, S. J. Puglisi, and J. Vuohtoniemi: *Small searchable κ-spectra via subset rank queries on the spectral Burrows-Wheeler transform*, in SIAM 2023 (ACDA23), 2023, pp. 225–236.
3. B. Alipanahi, A. Kuhnle, S. J. Puglisi, L. Salmela, and C. Boucher: *Succinct dynamic de Bruijn graphs.* Bioinformatics, 37(14) 2021, pp. 1946–1952.
4. A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya: *Succinct de Bruijn graphs*, in WABI 2012, vol. 7534 of Lecture Notes in Computer Science, Springer, 2012, pp. 225–235.
5. P. Bradley, H. C. Den Bakker, E. P. Rocha, G. McVean, and Z. Iqbal: *Ultrafast search of all deposited bacterial and viral genomic data.* Nature Biotechnology, 37(2) 2019, pp. 152–159.
6. P. Bradley et al.: *Rapid antibiotic-resistance predictions from genome sequence data for staphylococcus aureus and mycobacterium tuberculosis.* Nature Communications, 6(1) 2015, p. 10063.
7. N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter: *Near-optimal probabilistic RNA-seq quantification.* Nature Biotechnology, 34(5) 2016, pp. 525–527.
8. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm.* Technical Report 124, Digital Equipment Corporation, 1994.
9. K. Břinda: *Novel computational techniques for mapping and classification of Next-Generation Sequencing data.* PhD thesis, Université Paris-Est, 2016.

10. K. Břinda, M. Baym, and G. Kucherov: *Simplitigs as an efficient and scalable representation of de Bruijn graphs.* Genome Biology, 22(96) 2021.
11. K. Břinda, K. Salikhov, S. Pignotti, and G. Kucherov: *Prophyle 0.3.1.0.* Zenodo, 5281 2017.
12. K. Břinda et al.: *Rapid inference of antibiotic resistance and susceptibility by genomic neighbour typing.* Nature Microbiology, 5(3) 2020, pp. 455–464.
13. K. Břinda et al.: *Efficient and robust search of microbial genomes via phylogenetic compression.* Nature Methods, 22 2025, pp. 692–697.
14. R. Chikhi: *K-mer data structures in sequence bioinformatics.* HDR thesis, Institut Pasteur Ecole Doctorale "EDITE", 2021.
15. R. Chikhi, J. Holub, and P. Medvedev: *Data structures to represent a set of* k*-long DNA sequences.* ACM Computing Surveys, 54(1) 2022, pp. 17:1–17:22.
16. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev: *On the representation of de Bruijn graphs*, in RECOMB 2014, Cham, 2014, Springer International Publishing, pp. 35–55.
17. T. C. Conway and A. J. Bromage: *Succinct data structures for assembling large genomes.* Bioinformatics, 27(4) 2011, pp. 479–486.
18. V. G. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie: *Practical dynamic de Bruijn graphs.* Bioinformatics, 34(24) 2018, pp. 4189–4195.
19. K. Hannoush, C. Marchet, and P. Peterlongo: *Cdbgtricks: Strategies to update a compacted de Bruijn graph*, in Proceedings of the Prague Stringology Conference 2024, 2024, pp. 86–103.
20. G. Holley and P. Melsted: *Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs.* Genome Biology, 21(1) 2020, pp. 1–20.
21. J. Holt and L. McMillan: *Merging of multi-string bwts with applications.* Bioinformatics, 30(24) 2014, p. 3524–3531.
22. L. Kaplinski, M. Lepamets, and M. Remm: *GenomeTester4: a toolkit for performing basic set operations - union, intersection and complement on k-mer lists.* GigaScience, 4(1) 2015, pp. s13742–015–0097–y.
23. M. Karasikov, H. Mustafa, D. Danciu, M. Zimmermann, C. Barber, G. Rätsch, and A. Kahles: *Indexing all life's known biological sequences.* bioRxiv, 2020.10.01.322164 2024.
24. M. Kokot, M. Długosz, and S. Deorowicz: *KMC 3: counting and manipulating k-mer statistics.* Bioinformatics, 33(17) 2017, pp. 2759–2761.
25. A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo: *Fast and scalable minimal perfect hashing for massive key sets*, in SEA 2017, vol. 75 of LIPIcs, 2017, pp. 25:1–25:16.
26. C. Marchet, C. Boucher, S. J. Puglisi, P. Medvedev, M. Salson, and R. Chikhi: *Data structures based on k-mers for querying large collections of sequencing data sets.* Genome Research, 31(1) 2021, pp. 1–12.
27. C. Marchet, M. Kerbiriou, and Antoine: *Blight: efficient exact associative structure for k-mers.* Bioinformatics, 37(18) 2021, pp. 2858–2865.
28. I. Martayan, B. Cazaux, A. Limasset, and C. Marchet: *Conway–Bromage–Lyndon (CBL): an exact, dynamic representation of k-mer sets.* Bioinformatics, 40(Supplement_1) 2024, pp. i48–i57.
29. M. D. Muggli, B. Alipanahi, and C. Boucher: *Building large updatable colored de Bruijn graphs via merging.* Bioinformatics, 35(14) 2019, pp. i51–i60.
30. R. Patro, G. Duggal, M. I. Love, R. A. Irizarry, and C. Kingsford: *Salmon provides fast and bias-aware quantification of transcript expression.* Nature Methods, 14(4) 2017, pp. 417–419.
31. G. E. Pibiri: *Sparse and skew hashing of K-mers.* Bioinformatics, 38(Supplement_1) 2022, pp. i185–i194.
32. A. Rahman and P. Medevedev: *Representation of k-mer sets using spectrum-preserving string sets.* Journal of Computational Biology, 28(4) 2021, pp. 381–394, PMID: 33290137.
33. A. Rhie, B. P. Walenz, S. Koren, and A. M. Phillippy: *Merqury: reference-free quality, completeness, and phasing assessment for genome assemblies.* Genome biology, 21 2020, pp. 1–27.
34. K. Salikhov: *Efficient algorithms and data structures for indexing dna sequence data.* PhD thesis, Université Paris-Est, 2017.

35. S. Schmidt and J. N. Alanko: *Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time.* Algorithms for Molecular Biology, 18(1) 2023, p. 5.
36. S. Schmidt, S. Khan, J. N. Alanko, G. E. Pibiri, and A. I. Tomescu: *Matchtigs: minimum plain text representation of k-mer sets.* Genome Biology, 24(1) 2023, p. 136.
37. O. Sladký, P. Veselý, and K. Břinda: *Masked superstrings as a unified framework for textual k-mer set representations.* bioRxiv, 2023.02.01.526717 2023.
38. O. Sladký, P. Veselý, and K. Břinda: *Towards efficient k-mer set operations via function-assigned masked superstrings supplementary material.* Zenodo, 2025, available from https://github.com/OndrejSladky/f-masked-superstrings-supplement and https://doi.org/10.5281/zenodo.15804734.
39. O. Sladký, P. Veselý, and K. Břinda: *FroM Superstring to Indexing: a space-efficient index for unconstrainedk-mer sets using the masked burrows-wheeler transform (MBWT).* bioRxiv, 2024.10.30.621029 2024.
40. D. E. Wood and S. L. Salzberg: *Kraken: ultrafast metagenomic sequence classification using exact alignments.* Genome Biology, 15(3) 2014, pp. 1–12.
41. Z. D. Stephens et al.: *Big data: Astronomical or genomical?* PLoS Biology, 13(7) 2015, p. e1002195.

String Partition for Building Long Burrows-Wheeler Transforms

Enno Adler, Stefan Böttcher, and Rita Hartel

Paderborn University,
Warburger Straße 100, Paderborn, Germany
{enno.adler, stefan.boettcher, rita.hartel}@uni-paderborn.de

Abstract. Constructing the Burrows-Wheeler transform (BWT) for long strings poses significant challenges regarding construction time and memory usage. We use a prefix of the suffix array to partition a long string into shorter substrings, thereby enabling the use of multi-string BWT construction algorithms to process these partitions fast. We provide an implementation, partDNA, for DNA sequences. Through comparison with state-of-the-art BWT construction algorithms, we show that partDNA with the BWT construction algorithm IBB by Adler et al.[1] offers a novel trade-off for construction time and memory usage for BWT construction on real genome datasets. Beyond this, the proposed partitioning strategy is applicable to strings of any alphabet.

Keywords: Burrows-Wheeler transform, string partition

1 Introduction

The Burrows-Wheeler transform (BWT) [4] is a widely used reversible string transformation with applications in text compression [8], indexing [8], and short-read alignment [10]. The BWT reduces the number of equal-symbol runs for data compressed with run-length encoding and allows pattern search in time proportional to the pattern length [8]. Because of these advantages, and the property that the BWT of a string S can be constructed and reverted in $\mathcal{O}(|S|)$ time and space, the BWT is important in computational biology.

In computational biology, there are at least two major BWT construction cases: The BWT can be generated either for a single long genome, for example, the reference genome [10], or for a collection of comparatively short reads [2]. Like BCR [2] and ropebwt2 [11], the standard way [5] to define and compute a text index for a collection W of strings W_i is to concatenate the W_i with different end-marker symbols $\#_i$ between the W_i: $W' = W_0\#_0W_1\#_1 \dots W_k\#_k$.[1] The multi-string BWT [7] we use is also called BCR BWT [6], or mdolBWT [5].

In this paper, we show how to partition a string S into a collection W of short strings W_i and order the W_i in such a way that the BWT of W is similar to the BWT of S. We say 'similar' here because the BWT of $W_0, \dots, W_k$ contains $k+1$ # symbols that we need to remove from the BWT after construction. For example, compare the BWT of $S = CAAAACAAACCGTAAAACAAACCGGAACAA\$$ to the BWT of the collection $W = \{W_0, \dots, W_8\}$ of words

$$W_0 = A, W_1 = A, W_2 = AAACCGGAAC, W_3 = AAACCGT,$$
$$W_4 = \$C, W_5 = A, W_6 = A, W_7 = AAAC, W_8 = AAAC.$$

[1] The $\#_i$ symbol at the end of the strings W_i and the \$ symbol in S are only different to explain the concept and break ties; implementations like ropebwt2 [11] and our approach, partDNA, use the same symbol for each end-marker.

Enno Adler, Stefan Böttcher, Rita Hartel: String Partition for Building Long Burrows-Wheeler Transforms, pp. 41–55.
Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9

Because $W_4W_6W_8W_3W_5W_7W_2W_1W_0 \cdot \$ = \$ \cdot S$, W is a partition of the first right rotation[2] of S. The BWT of W is constructed using $W_0, \ldots, W_k$ in the order of their indices:

$$BWT(W) = AACTCAACC\#\#\#\#\#\#\#\#GAAAAAAAAAA\$AAAACCGCCG$$
$$BWT(S) = AACTCAACC \qquad\qquad GAAAAAAAAAA\$AAAACCGCCG$$

If we remove the run of # symbols, the BWTs of S and W are identical. In Figure 1, we show the connection between single-string and multi-string BWT construction algorithms and the contribution of our paper.

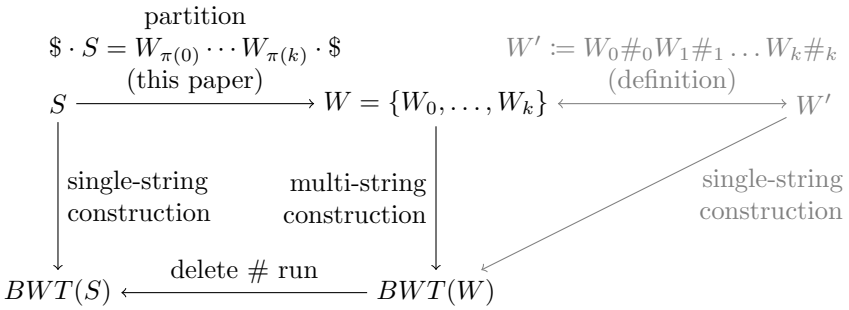


Figure 1. Summary of the relationship of single-string and multi-string BWT construction algorithms and the contribution of this paper. The BWT for S can also be obtained by partitioning S, using a multi-string construction algorithm on the sorted partition, and removing the #-run at the end. The output of the multi-string BWT construction algorithm is equal to the BWT for W'.

Herein, our main contributions are as follows:

- The proof of the correctness of computing the $BWT(S)$ as shown in Figure 1.
- An implementation, partDNA, to partition long DNA sequences for BWT construction.[3]
- A comparison of state-of-the-art BWT construction algorithms regarding the construction time and memory usage of using the partition.

2 Related Work

The BWT [4] is fundamental to many applications in bioinformatics such as short-read alignment. Bauer et al. [2] designed BCR for a collection of short DNA reads. BCR inserts all reads in parallel starting at the end of each sequence at the same time. The position to insert the next symbol of each sequence is calculated from the current position and the constructed part of the BWT. The BWT is partitioned into buckets where one bucket contains all BWT symbols of suffixes starting with the same character. The buckets are saved on disk. Ropebwt, ropebwt2 by Li [11], and IBB by Adler et al. [1] are similar to BCR, but use different trees instead of linear saved buckets.

The BWT can be obtained from the suffix array by taking the characters at the position before the suffix. Thereby, suffix array construction algorithms (SACAs) like divsufsort [9], SA-IS by Nong et al. [15], libsais, gSACA-K by Louza et al. [12], or

[2] Using the substring notation of Section 3, the first right rotation of S is $\$ \cdot S[0, |S| - 1]$.
[3] The implementation is available at `https://github.com/adlerenno/partDNA`.

gsufsort by Louza et al. [13] can be used to obtain the BWT. Many SACAs rely on induced suffix sorting. Induced suffix sorting derives the order of the previous positions of a sorted set of suffix positions if these point to equal characters. An example is shown in Figure 4.

The grlBWT method by Díaz-Domínguez et al. [6] uses induced suffix sorting, but additionally uses run-length encoding and grammar compression to store intermediate results and to speed up computations required for BWT construction.

eGap by Egidi et al. [7] divides the input collection into small subcollections and uses gSACA-K [12] to compute the BWT for the subcollections. Thereafter, eGap merges the subcollection BWTs into one BWT.

In a prefix-free set, no two words from the set are prefixes of each other; thus, their order is based on a different character rather than on word length. Consequently, the order of two suffixes starting with words from the prefix-free set is determined by these words. BigBWT by Boucher et al. [3] builds the BWT using prefix-free sets. BigBWT replaces the input with a dictionary D and a parse P using the rolling Karp-Rabin hash. P is the list of entries in D according to the input string; D forms a prefix-free set. r-pfbwt by Oliva et al. [16] recursively uses prefix-free parsing on the parse P to further reduce the space needed to represent the input string.

SA-IS [15], libsais, gSACA-K [12], BigBWT [3], and r-pfbwt [16] partition the input at LMS-Positions or by using a dictionary. The difference of our partition to all these approaches is that we do not create a BWT as a result. Instead, we translate the problem into a multi-string BWT construction problem and compute the BWT using such a construction method.

3 Preliminaries[4]

We define a string S of length $|S| = n$ over Σ by $S = a_0 a_1 \cdots a_{n-1}$ with $a_i \in \Sigma$ for $i < n$ and always append $a_n = \$$ to S. We write $S[i] = a_i$, $S[i,j] = a_i a_{i+1} \cdots a_j$ for a substring of S, and $S[i..] = S[i,n]$ for the suffix starting at position i. For simplicity, we assume that $S[-1] = S[n] = \$$, and also allow $S[-1,j] = a_n a_0 \cdots a_j$ as a valid interval.

The suffix array $SA(S)$ [14] of a string S is a permutation of $\{0,\ldots,n\}$ such that the i-th smallest suffix of S is $S[SA(S)[i]..]$. The suffix array $SA(W)$ and document array $DA(W)$ for a collection W of strings W_i are arrays of numbers such that the j-th smallest suffix of all W_i is $W_{DA(W)[j]}[SA(W)[j]..]$.

The Burrows–Wheeler transform $BWT(S)$ of a string S [4] can either be obtained by $BWT(S)[i] = S[SA(S)[i]-1]$ or by taking the last column of the sorted rotations of S.

4 Partition Theorem

A single string S has only unique suffixes because the suffixes differ in their length. However, if we partition S into $W = \{W_0,\ldots,W_k\}$, $W_0,\ldots,W_k$ can have several equal suffixes. For example, $W_2 = AAACCGGAAC$ and $W_7 = AAAC$ both have the suffix AAC. Given only the suffixes, we cannot decide how to order the characters G and A in $BWT(W)$ that occur before the suffixes AAC in W_2 and W_7. We break the tie by using the word order of W_2 and W_7. For that purpose, we choose the word

[4] There is a list of symbols available in the full version on `https://arxiv.org/abs/2406.10610`.

order of the W_i to be the order of the suffixes occurring in S after these words W_i. In other words, the index i of word W_i is the number of strictly smaller suffixes within the set of all suffixes of S that start behind words from the collection W. In Figure 2, we visualized this concept of obtaining the word indices.

We transfer this argumentation from suffixes to suffix arrays: The index i of word W_i is the index i within a filtered suffix array that contains only those positions in S that are after the words of W. Computing the full suffix array and thereafter filtering it for the positions that follow the word ends yields the correct order of the words. However, if we would construct the full suffix array to partition S, this would be inefficient because we can obtain the BWT from the suffix array directly. But this trivial solution shows that we can obtain the word indices in $\mathcal{O}(n)$ time.

In multi-string construction of $BWT(W)$, the last characters $c_0, \ldots, c_k$ of each word $W_0, \ldots, W_k$ are inserted at the first positions of the BWT of W; thus, for our approach, $c_0, \ldots, c_k$ must be the first $k+1$ symbols of $BWT(S)$. Because $BWT(S)[i] = S[SA(S)[i]-1]$, the positions in S following the words $W_0, \ldots, W_k$ form a continuous subarray at the lowest positions of the suffix array of S: A prefix of the suffix array.

We define $PSA(S) = SA(S)[0 \ldots k]$ as a prefix of length $k+1$ of the suffix array of S. In the following, we assume $k < n$ to be a fixed value. We write $t \in PSA(S)$, if there exists i, $0 \leq i \leq k$, such that $t = PSA(S)[i]$.

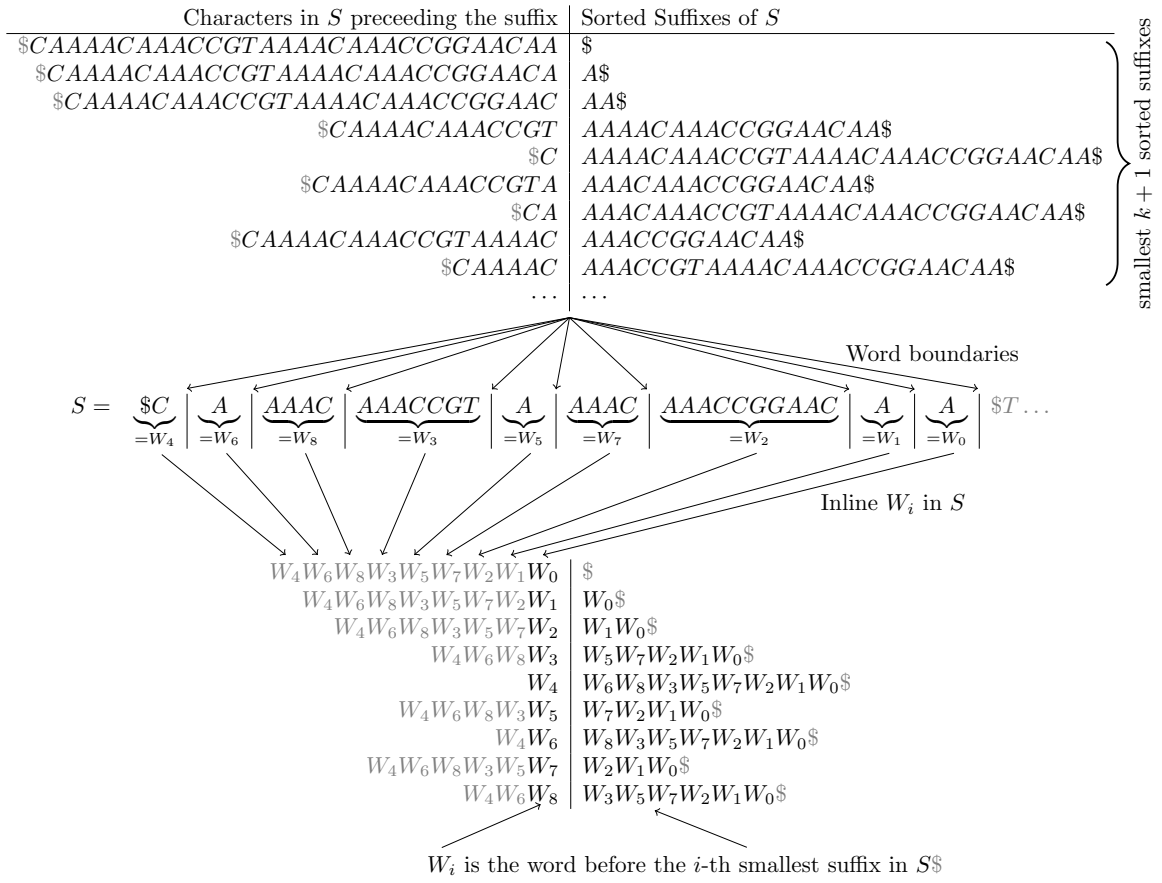


Figure 2. Concept of partitioning S using $PSA(S)$ with $k = 8$ to obtain the collection W. The suffix of a word W_i in S could either be expressed by characters of S or by words from W in the order of their appearance in S

For each suffix entry $PSA(S)[j]$, let $\Omega(j) = \max(\{-1\} \cup \{t \in PSA(S) : t \leq PSA(S)[j]-1\})$ be the next smaller suffix array entry in $PSA(S)$ or -1, if $PSA(S)[j]$ is already the smallest value in $PSA(S)$. We define W as the collection of words $W_i = S[\Omega(i), PSA(S)[i] - 1]$ for $0 \leq i \leq k$, which is a partition of the first right rotation of S into substrings.

As W is a partition of the first right rotation of S, each character of S is mapped to exactly one character in one of the words of W. Therefore, we define two functions *word* and *position* to map a position q from S to the word $W_{word(q)}$ in W and position $position(q)$ of that character $S[q]$ in $W_{word(q)}$: For each q with $-1 \leq q < |S|$ exists exactly one $j \leq k$ such that $q \in [\Omega(j), PSA(S)[j] - 1]$, because each position in the partition belongs to exactly one interval. We set $word(q) = j$ and $position(q) = q - \Omega(j)$. We also set $word(n) = word(-1)$ and $position(n) = position(-1)$.

In the initial example, the C at position 0 in S corresponds to the C in word W_4 at position 1, so $word(0) = 4$ and $position(0) = 1$.

Theorem 1. *Let $l(= k + 1)$ be the size of W and $m(= n + l)$ be the total length of $BWT(W)$. Then, for all $i < m$:*

$$BWT(W)[i] = \begin{cases} BWT(S)[i] & 0 \leq i < l \\ \# & l \leq i < 2l \\ BWT(S)[i - l] & 2l \leq i < m \end{cases}$$

Thus, if we know $BWT(W)$, we get by removing the #-run

$$BWT(S) = BWT(W)[0, l - 1] + BWT(W)[2l, m - 1].$$

The full proof is in Appendix A. Proof sketch: Using the functions *position* and *word*, we prove the statement: if two suffixes $S[i..] < S[j..]$, then the order of the suffixes is $W_{word(i)}[position(i)..] < W_{word(j)}[position(j)..]$. This can be proven by using c, the smallest value for which either $S[i + c] \neq S[j + c]$, or $i + c \in PSA(S)$. Using this statement, we express $SA(W)$ and $DA(W)$ with $SA(S)$ and the *position* and *word* functions. Then, we use $BWT(W)[i] = W_{DA(W)[i]}[SA(W)[i] - 1]$ for the most cases to retrieve the $BWT(W)$ from $SA(W)$ and $DA(W)$ except for the additional # symbols.

5 Partition DNA Sequences: partDNA

Next, we partition a DNA sequence because genomes are long strings over $ACGT$. To partition a DNA sequence S, we first find the words having the smallest suffixes in S, and second, we order the words according to their suffixes in S.

We do not allow every k as the length of the prefix $PSA(S)$ of the suffix array and k is not given explicitly. Instead, the exact value of k will be part of the result of the partition. In particular, we use the chosen length h for the minimal length of an A run as a parameter to partition S. As a longer A run contains a shorter A run at its end, a smaller value of h results in an equal or larger value of k.

We partition $S = CAAAACAAACCGTAAAACAAACCGGAACAA\$$ with $h = 3$ within our following continuous example. The example is visualized in Figure 3. Before we go into the details, we give a short high-level description: In Step 1, we scan S to find an initial collection of words. This collection is smaller than the final collection W, because we can avoid sorting the larger collection W by using induced sorting. This will be Step 6. To sort the subset of suffixes behind the words in the

collection, we sort the initial collection of words in Steps 2 and 3 and if at least two words are equal (Step 4), we will break their tie by their suffixes. As these suffixes start with words again, see Figure 2 for an example, and we have already sorted the words in Step 2 and 3, we can use this to assign a unique name to every different word (Step 4) and compute a suffix array (Step 5) that breaks all remaining ties. With the induced sorting in Step 6 and the final polish in Step 7 we get W.

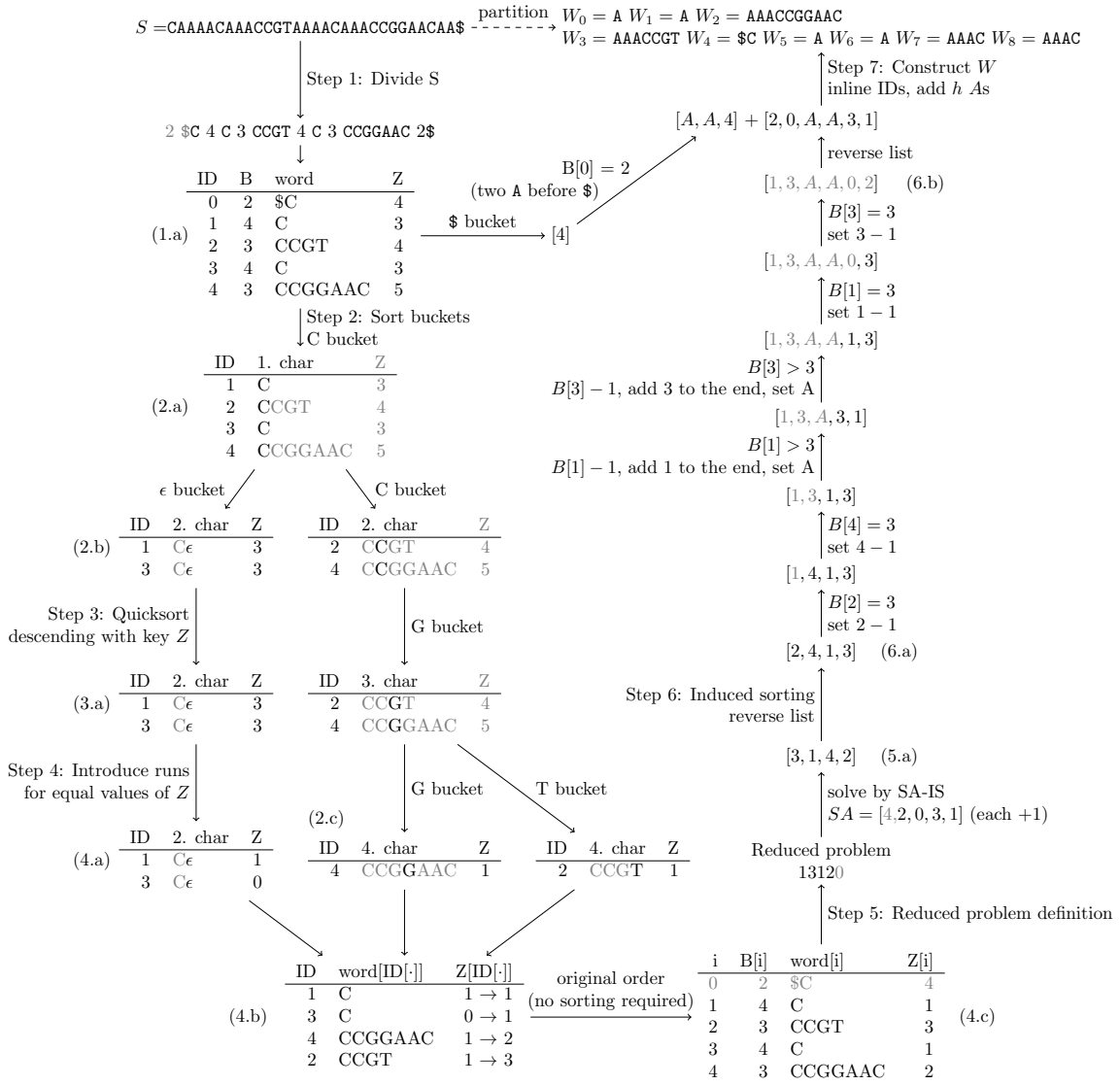


Figure 3. Example calculation of partitioning using $h = 3$ on the word S. In Steps 2 and 3, we only reorder the array ID, the reordering of columns of the other elements is only shown for illustration. Steps 2 and 3 are done in place, so there is no action needed to go from (4.a) and (2.c) to (4.b).

In Step 1, we divide S either before each run of at least h A symbols or before $A^*\$$ and assign IDs to the words in order of their appearance. Note that these word IDs are different from the word incides that are assigned to order their appearance in W. Additionally, we avoid writing runs having h or more As, we only save their run-length. This results in fewer words than the final partition W has; we obtain the other words in Step 6 by induced suffix sorting. We keep the number of As before a word in an array B and the number of As after the word in an array Z. When we reach the \$ symbol, we keep the length of the A-run preceding \$ in $B[0]$, so in the

example of Figure 3, $B[0] = 2$. We store the maximum length of an A run plus 1 in the last element of Z regardless of how long the current A run is; so, in the example, we store $Z[4] = 5$. We use these values of Z later to sort the words descendingly because we designed the values in Z in such a way that a higher value implies that a smaller suffix follows. Because S does not start with at least h As, S is not divided at position 0, which is also the position after the \$ regarding rotations of S. Therefore, we place the \$ at the beginning of the word with the ID 0.

In Step 2, we sort the IDs with a value of 1 and greater (Figure 3 (1.a)), which form our initial bucket. We sort the IDs by recursively refining buckets: At recursion depth i, we group the IDs of one bucket into subbuckets by the i-th character of their words. Therefore, the call structure is a tree (Figure 3 (2.a) to (2.c)), that has a maximum number of 5 branches on the next level, one for each character A, C, G, and T and one branch named ϵ for words with no more characters. In particular, to order the IDs into the subbuckets, we first count the number of symbols to get the size of the subbuckets, then create 5 auxiliary arrays, one for each bucket, copy in a single scan the IDs into the auxiliary array and finally copy all auxiliary arrays back in order. The arrays B and Z and the words are only accessed indirectly and changed by using $B[ID[i]]$ and $Z[ID[i]]$, so we only sort the IDs in Step 2.

If a word w has no further characters, we reached an ϵ-leaf of the sort tree for the word w. Then, the suffix of S starting with w is smaller than the suffixes of S of the words that have a character at that position. The reason is that the pattern A^{y+h} or the lexicographically even smaller suffix $A^y\$$ with $y \geq 0$ occurs after w in S, because we used these patters to divide S into the words. Because we used different patterns to divide S, we sort the words according to the order of these patterns with quicksort as Step 3. In Figure 3 (1.a), the patterns A^3, A^4, and $A^2\$$ are encoded by the values 3, 4, and 5 in the array Z. Therefore, the quicksort sorts the IDs in descending order by using the values in Z as keys.

After the quicksort in the leaves of the sort tree, we might have identical words as in Figure 3 (3.a). In the leaf, two words are equal if they appear consecutively and their value in Z is equal. In Step 4, we encode equal words as a run, so the first of the equal words gets a 1 and each other word gets a 0 in Z. This allows an easier assignment of names in the following step. All words in leaves containing only one word get a 1 in Z, like in Figure 3 (2.c).

To solve the case that we found at least two equal words in Step 4, we define a reduced suffix array construction problem to sort them. In Figure 3 (4.a), the words with IDs 1 and 3 are equal. The reduced problem is defined as in SA-IS [15] and similar SACAs. In a single pass over $Z[ID[i]]$, we give integer names to the words, like in Figure 3 (4.b): We add the current value of Z, which is 0 or 1, to the last name and then assign the sum to Z. The smallest assigned name is 1, because we use 0 as a global end marker for a valid suffix array problem definition. We obtain the reduced problem by reading the names from $Z[i]$ omitting the first word with ID 0.

In Figure 3 after Step 5, the reduced problem is $R = 13120$, with the global end marker 0. We obtain $SA(R) = [4,2,0,3,1]$. We omit the first value 4 because the 4 points towards the end marker 0. We need to add 1 to the $SA(R)$ values because we skipped the word with ID 0 in the recursive problem. We obtain $[3,1,4,2]$ (Figure 3 (6.a)).

In Step 6, we obtain all words from the sorted IDs by induced suffix sorting. Figure 4 shows the possible induced suffix sorting steps. Let L be the list of IDs in inverted order, so $L = [2,4,1,3]$. We iterate from the start until we reach the end of

L. For the ID j, if $B[j]$ is greater than h: We reduce $B[j]$ by 1, write an A to the current position of the list, and append the current ID j to the end of the list again. If $B[j]$ is h: We reduce j by 1 at the current position. Thereby, we change over from the word at the start of a suffix to the word before the suffix. We invert the list again, so $L = [2, 0, A, A, 3, 1]$.

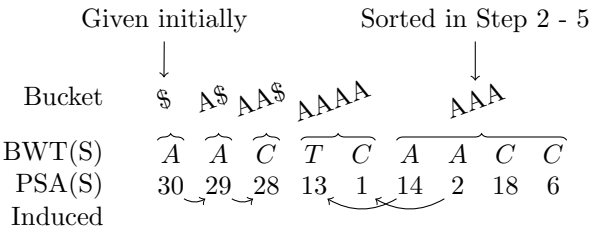


Figure 4. The buckets and the steps of induced suffix sorting that start and end inside the displayed interval. The name of a bucket is a shared prefix of the suffixes. For the A^v buckets with $v \in \mathbb{N}$, it is required that the next symbol is not an A; otherwise, the AAA and $AAAA$ bucket would also belong to the AA bucket. We induce the positions from the \$ bucket to the next bucket on the right and reduce the suffix array entry by 1 if and only if the symbol at the position before the entry is an A, which is the symbol in the BWT row. In the same way, we can fill the A^v buckets right to left.

For the \$-bucket, we also use induced sorting: The starting list is $Q = [4]$ because the word with ID 4 in Figure 3 (1.a) of the divided words has $A^{B[0]}\$$ as a suffix and we do $B[0]$ steps each inserting one A. The result is $Q = [A, A, 4]$ and the combined result is $Q + L$.

In Step 7, each A in $Q + L$ yields an own word $W_i = A$. Each ID in $Q + L$ yields the corresponding word in Figure 3 (1.a). Additionally, in front of each word with an ID that is 1 or greater in Figure 3 (1.a), we prepend h As. After Step 7, we get the partition:

$$W_0 = A, W_1 = A, W_2 = AAACCGGAAC, W_3 = AAACCGT,$$
$$W_4 = \$C, W_5 = A, W_6 = A, W_7 = AAAC, W_8 = AAAC.$$

To summarize: By partitioning S into the collection $W = \{W_0, \ldots, W_8\}$, we have transformed the task of constructing the BWT of a long string S into the task of constructing the BWT of a collection W of smaller words.

6 Runtime Complexity

We claim that partDNA runs in $\mathcal{O}(n)$ time. First, the number of words of the scan in Step 1 is upper bounded by $\frac{n}{h} + 1$, because there is always at least one word, which is produced by the \$-symbol, and there needs to be an A-run of length h or longer between two words to divide the words. Thus, in all next steps, we will work on at most $\frac{n}{h}$ words, because we process the word with ID 0 differently.

We simplify the recursion to ease the analysis of the Steps 2, 3, and 4. First, instead of using quicksort on the Z-values, we extend the word with ID i by a $Z[i]$-long A-run. Note that after h consecutive A-buckets, the sort order changes in the sense that the A bucket is smaller than the ϵ-bucket and that there are only those two buckets anymore, however this does not change the number of steps to perform the sort. Additionally, we stop the recursive bucket sort only if an ϵ-bucket is reached,

we do not stop if it is the last element in a bucket. These adjustments let us remove the quicksort at the higher cost of a deeper recursion. Using this simplification, each letter of S is exactly used once to put an ID into a subbucket, because each letter of S belongs to exactly one word and the i-th letter is used only at recursion depth i in Step 2. Putting an ID into a bucket needs only $\mathcal{O}(1)$ steps. Thereby, Step 2 has $\mathcal{O}(n)$ steps. Step 4 uses $\mathcal{O}(1)$ steps per word in the ϵ-leaf, thus Step 4 needs $\mathcal{O}\left(\frac{n}{h}\right)$ steps.

For Step 5, the reduced problem R consists of one letter per sorted word plus the end marker, thus R has size $\frac{n}{h}+1$. The suffix array of R needs $\mathcal{O}(\frac{n}{h})$ steps using SA-IS. All induced sorting steps together, including those of Step 6, perform $\frac{n}{h}+1$ times an insertion of a word ID and up to n times an insertion of an A, because $|S| = n$ limits the number of As in S. Hence, $\mathcal{O}(n) + \mathcal{O}\left(\frac{n}{h}+1\right) = \mathcal{O}(n)$.

As all steps are in $\mathcal{O}(n)$, partDNA has $\mathcal{O}(n)$ runtime complexity.

7 Experimental Results

We compare the BWT construction algorithms on the datasets listed in Table 1 regarding construction time and RAM usage. We obtain time and maximum resident set size (max-rss) by the `/usr/time` command.[5] Each input file is a concatenation of all bases within the reference file because we want to test a single long input string. Ambiguous bases are omitted. In Table 1, if a partition length h is provided, the dataset was partitioned using h from the dataset with the row $h = -$. We performed all tests on a Debian 5.10.209-2 machine with 128GB RAM and 32 Cores Intel(R) Xeon(R) Platinum 8462Y+ @ 2.80GHz.

ropebwt3, which uses a SA-IS implementation, BigBWT, r-pfbwt, divsufsort, libsais, grlBWT, and gsufsort compute the BWT of a collection by concatenating the strings. Hence, they are only tested on the single-string dataset (called original), because partitioning the input adds extra symbols in form of the end-markers. ropebwt, ropebwt2, IBB, and BCR are tested on the partitioned datasets. Because partDNA and the following construction algorithm are sequential, we took the sum of their runtimes, and we used the maximum of their max-rss values. In most cases, partDNA had a lower max-rss than the following BWT construction algorithm. eGap was tested on both, but we omit the results for eGap on the partitioned datasets because they were slower than on the single-string dataset, which is internally constructed by gSACA-K only.

In Figure 5, our test series suggest that $h \in \{3, 4, 5\}$ works best regarding construction time as shown for chromosome 1 of GRCh38. The scatter plots in Figure 5 visualize the needed time and RAM usage given the file, the approach, and the length h. An algorithm A is pareto optimal if there is no other algorithm B that uses both, less or equal time, which means the point of B is left or equal to the point of A, and less or equal RAM, so the point of B is equal or lower than the point of A. For example, eGap is not pareto optimal on the files GRCm39 and GRCh38, because divsufsort uses less time and less RAM compared to eGap, so divsufsorts point is in the left lower direction from the point of eGap.

Our test shows that libsais is the fastest construction algorithm for this class of BWT construction problems followed by divsufsort and ropebwt3. grlBWT uses the lowest amount of RAM. Our results show, that only libsais, divsufsort, ropebwt3, grlBWT and partDNA with IBB achieve pareto optimal results. partDNA and IBB

[5] The test is available at `https://github.com/adlerenno/partDNAtest`.

Table 1. Used BWT construction algorithms and datasets from NCBI.

approach	paper	implementation
ropebwt	–	github.com/lh3/ropebwt
ropebwt2	[11]	github.com/lh3/ropebwt2
ropebwt3	–	github.com/lh3/ropebwt3
IBB	[1]	github.com/adlerenno/ibb
BigBWT	[3]	gitlab.com/manzai/Big-BWT
r-pfbwt	[16]	github.com/marco-oliva/r-pfbwt
divsufsort	[9]	github.com/y-256/libdivsufsort
libsais	–	github.com/IlyaGrebnov/libsais
grlBWT	[6]	github.com/ddiazdom/grlBWT
eGap	[7]	github.com/felipelouza/egap
gsufsort	[13]	github.com/felipelouza/gsufsort
BCR	[2]	github.com/giovannarosone/BCR_LCP_GSA

dataset	h	average length	collection size l
GRCm39	–	2,654,621,783	1
GRCm39	3	30	88,252,043
GRCm39	4	83	31,890,467
GRCm39	5	215	12,350,256
GRCh38	–	3,049,315,783	1
GRCh38	3	26	116,219,956
GRCh38	4	67	46,529,667
GRCh38	5	151	20,189,969
JAGHKL01	–	14,314,496,836	1
JAGHKL01	3	40	356,650,569
JAGHKL01	4	121	118,749,573
JAGHKL01	5	364	39,284,785

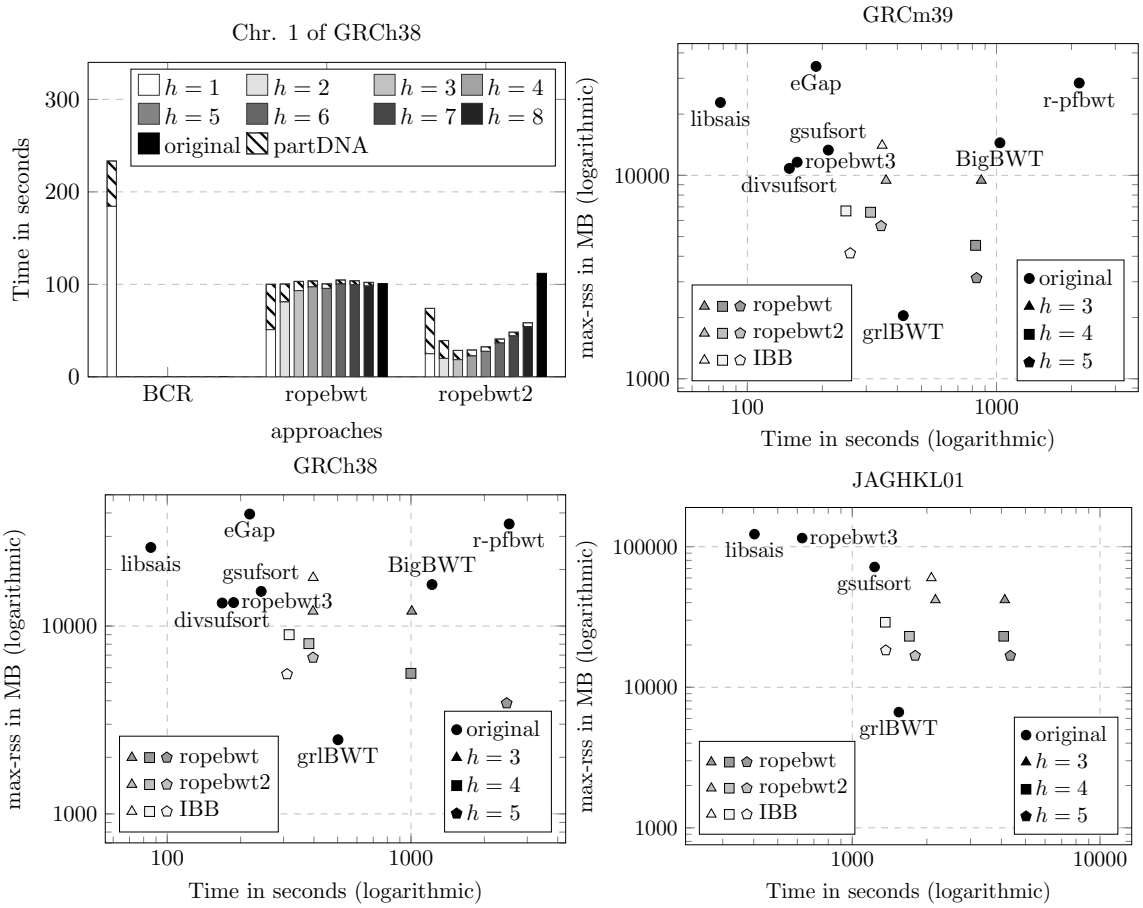


Figure 5. BWT construction times and maximum resident set sizes (max-rss). Grey polygons in scatter plots belong to a partioned dataset: the grey tone determines the BWT construction algorithm and the number of edges the used parameter h, as the legends explain. Missing points mean that the construction algorithms abort or do not create an output file.

together, especially for $h \in \{4, 5\}$, offer a new, and balanced trade-off between speed and space consumption.

8 Conclusion

We have presented an approach that partitions long strings of any alphabet to transform a long single-string BWT construction problem into a multi-string BWT construction problem. We have shown that our implementation partDNA designed for DNA sequences in conjunction with IBB provides a new pareto-optimum within the time-space trade-off for BWT construction.

References

1. E. Adler, S. Böttcher, R. Hartel, and C. A. Steininger: *IBB: Fast Burrows-Wheeler Transform Construction for Length-Diverse DNA Data.* CoRR, abs/2502.01327 2025.
2. M. J. Bauer, A. J. Cox, and G. Rosone: *Lightweight algorithms for constructing and inverting the BWT of string collections.* Theor. Comput. Sci., 483 2013, pp. 134–148.
3. C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun: *Prefix-free parsing for building big BWTs.* Algorithms Mol. Biol., 14(1) 2019, pp. 13:1–13:15.
4. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm*, in Digital SRC Research Report, Citeseer, 1994.
5. D. Cenzato and Z. Lipták: *A Theoretical and Experimental Analysis of BWT Variants for String Collections*, in 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic, 2022, pp. 25:1–25:18.
6. D. Díaz-Domínguez and G. Navarro: *Efficient construction of the BWT for repetitive text using string compression.* Inf. Comput., 294 2023, p. 105088.
7. L. Egidi, F. A. Louza, G. Manzini, and G. P. Telles: *External memory BWT and LCP computation for sequence collections with applications.* Algorithms Mol. Biol., 14(1) 2019, pp. 6:1–6:15.
8. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) 2005, pp. 552–581.
9. J. Fischer and F. Kurpicz: *Dismantling DivSufSort*, in Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017, 2017, pp. 62–76.
10. B. Langmead and S. L. Salzberg: *Fast gapped-read alignment with Bowtie 2.* Nature methods, 9(4) 2012, pp. 357–359.
11. H. Li: *Fast construction of FM-index for long sequence reads.* Bioinform., 30(22) 2014, pp. 3274–3275.
12. F. A. Louza, S. Gog, and G. P. Telles: *Induced Suffix Sorting for String Collections*, in 2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016, 2016, pp. 43–52.
13. F. A. Louza, G. P. Telles, S. Gog, N. Prezza, and G. Rosone: *gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections.* Algorithms Mol. Biol., 15(1) 2020, p. 18.
14. U. Manber and G. Myers: *Suffix Arrays: A New Method for On-Line String Searches*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA, 1990, pp. 319–327.
15. 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.
16. M. Oliva, T. Gagie, and C. Boucher: *Recursive Prefix-Free Parsing for Building Big BWTs*, in Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023, 2023, pp. 62–70.

A Proof of the Partition Theorem

Proposition 2. *For any q with $-1 \leq q < n$: if $q \in PSA(S)$ or $q = -1$, then*

$$position(q) = 0.$$

Proof. $SA[0] = n$, because $S[n] = \$$ is by definition the smallest symbol and unique, thereby, n is the index of the smallest suffix. Let p be the $\min(\{t \in PSA(S) : t > q\})$ which is well defined, because $q < n$ and $n \in PSA(S)$ for any $k \geq 0$. Then $q = \Omega(p)$ and $q \in [\Omega(p), PSA(S)[p] - 1]$, so $position(q) = q - \Omega(p) = q - q = 0$.

Proposition 3. *For any q with $0 \leq q < n$: if $q \notin PSA(S)$, then*

$$word(q) = word(q-1) \text{ and } position(i-1) = position(i) - 1.$$

Proof. If $position(q) = 0 = q - \Omega(j)$ for a j, it follows $\Omega(j) = q$, so $q \in \{-1\} \cup PSA(A)$, which contradicts the assumptions. Thereby, $\Omega(j) < q < PSA(j) - 1$, from this we get $\Omega(j) \leq q - 1 < PSA(j) - 1$, so $word(q-1) = word(q)$. $position(q-1) = q - 1 - \Omega(j) = q - \Omega(j) - 1 = position(q) - 1$.

We define $W'_i = W_i + \#_i$ and use the W'_i in the proofs, because $BWT(W)$ contains k # symbols that the W_i do not contain. The lexicographical order is $\#_0 < \#_1 < \cdots < \#_k < \$ < a$ for all $a \in \Sigma$. The advantage of using the W'_i is, that their total length is equal to the length of $BWT(W)$. Thus, we can write down the suffix array for the W'_i that fits to $BWT(W)$, which is not possible for the W_i. The advantages of using the W_i in the previous part of the paper are the easier presentation of the partition of S and the definition of a multi-string problem as in Figure 1. We add the index i to the # symbols in order to break the tie of equal suffixes at the # symbols.

Proposition 4. *If $S[i..] < S[j..]$, then*

$$W'_{word(i)}[position(i)..] < W'_{word(j)}[position(j)..].$$

Proof. Let $c \geq 0$ be the smallest value for which either $S[i+c] \neq S[j+c]$, or $i + c \in PSA(S)$, so $S[i, i+c-1] = S[j, j+c-1]$. The idea here is, that c is the distance to the positions where the tie between the suffixes starting at i and j breaks. First, from $S[i, i+c-1] = S[j, j+c-1]$ and $S[i..] < S[j..]$, we conclude that $S[i+b..] < S[j+b..]$ for all b with $0 \leq b < c$. As $PSA(S)$ contains the positions of the smallest suffixes of S, we conclude from $i, i+1, \ldots, i+c-1 \notin PSA(S)$ and $S[i+b..] < S[j+b..]$ for all b with $0 \leq b < c$ that $j, j+1, \ldots, j+c-1 \notin PSA(S)$.

Second, by Proposition 3, we now get

$$\begin{aligned}
&word(i) = word(i+1) = \cdots = word(i+c-1),\\
&word(j) = word(j+1) = \cdots = word(j+c-1),\\
&position(i+c-1) = position(i+c-2) + 1 = \cdots = position(i) + c - 1,\\
&position(j+c-1) = position(j+c-2) + 1 = \cdots = position(j) + c - 1.
\end{aligned}$$

This shows, that the tie of $position(i)$ in $W'_{word(i)}$ in comparison to $position(j)$ in word $W'_{word(j)}$ is not decided before the distance c. In other words, if the order of the suffixes is $W'_{word(i+c-1)+1}[position(i+c-1)+1..] < W'_{word(j+c-1)+1}[position(j+c-1)+1..]$, we get $W'_{word(i)}[position(i)..] < W'_{word(j)}[position(j)..]$.

Now, we distinguish three cases.

First case, if $S[i+c] \neq S[j+c]$ and $i+c \notin PSA(S)$: Like above, $j+c \notin PSA(S)$, so we get

$$W'_{word(i+c)}[position(i+c)] = S[i+c] < S[j+c] = W'_{word(j+c)}[position(j+c)].$$

This is the case, when the comparison of two suffixes in W can be decided without getting to the end of a word in W.

Second case, if $i+c = PSA(S)[v]$ and $j+c \notin PSA(S)$: Then

$$W'_{word(i+c-1)}[position(i+c-1)+1] = \#_v < S[j+c] = W'_{word(j+c)}[position(j+c)].$$

Note that $position(i+c-1)+1 \geq 1$ and $position(i+c) = 0$ by Proposition 2 because $word(i+c-1) \neq word(i+c)$.

Third case, if $i+c = PSA(S)[v]$ and $j+c = PSA(S)[w]$: From $S[i..] < S[j..]$, we get $v < w$ by the definition of the suffix array. Then

$$\begin{aligned} W'_{word(i+c-1)}[position(i+c-1)+1] &= \#_v \\ &< \#_w \\ &= W'_{word(j+c-1)}[position(j+c-1)+1] \end{aligned}$$

Theorem 5. *Let S be a string of length n over the alphabet Σ. Let W be the collection of partitioned words $W'_i = S[\Omega(i), PSA(S)[i]-1]+\#_i$ obtained from S. Let $l(=k+1)$ be the size of W, $m(=n+l)$ be the total length of W, and let $SA(S)$ and $SA(W)$ be the suffix arrays of S and W, respectively. Then, the suffix array and document array are (for all $i < m$):*

$$SA(W)[i] = \begin{cases} |W_i| & 0 \leq i < l \\ position(SA(S)[i-l]) & l \leq i < m \end{cases}$$

$$DA(W)[i] = \begin{cases} i & 0 \leq i < l \\ word(SA(S)[i-l]) & l \leq i < m \end{cases}$$

Proof. By construction of the words W'_i, the smallest $k+1 = l$ characters in W are $\#_i$ and each $\#_I$ occurs only once. Thereby, we get $SA(W)[i] = |W'_i|-1 = |W_i|$, which is the position of the $\#_i$ in W'_i, together with $DA(W)[i] = i$ for $0 \leq i < m$ by the order of the $\#_i$ symbols.

Next, there is a continuous block of length n left in the $SA(W)$ and $DA(W)$ arrays to prove. By definition of the suffix array, we get for the string S

$$S[SA(S)[0]..] < \cdots < S[SA(S)[n-1]..].$$

By Proposition 4, we get the following order of the remaining n suffixes of W:

$$\begin{aligned} & W'_{word(SA(S)[0])}[position(SA(S)[0])..] \\ <& W'_{word(SA(S)[1])}[position(SA(S)[1])..] \\ <& \cdots \\ <& W'_{word(SA(S)[n-1])}[position(SA(S)[n-1])..]. \end{aligned}$$

The inequations show the order of the remaining n suffixes. For example, we get that $W'_{word(SA(S)[0])}[position(SA(S)[0])..]$ is the l-th lowest suffix of W, so $SA(W)[l] = position(SA(S)[l-l])$ and $DA(W)[l] = word(SA(S)[l-l])$. The additional $-l$ within the terms $SA(S)[i-l]$ come from the fact that this order starts at position l in $SA(W)$ instead of at position 0.

Theorem 1. *Let S be a string of length n over the alphabet Σ. Let W be the collection of partitioned words $W'_i = S[\Omega(i), PSA(S)[i]-1]+\#_i$ obtained from S. Let $l(=k+1)$ be the size of W, $m(=n+l)$ be the total length of $BWT(W)$, and let $BWT(S)$ and $BWT(W)$ be the BWTs of S and W, respectively. Then, for all $i < m$:*

$$BWT(W)[i] = \begin{cases} BWT(S)[i] & 0 \leq i < l \\ \# & l \leq i < 2l \\ BWT(S)[i-l] & 2l \leq i < m \end{cases}$$

Proof. We calculate $BWT(W)$ from $SA(W)$. For any $i < m$:

$$BWT(W)[i] = W'_{DA(W)[i]}[SA(W)[i]-1].$$

In the case that $i < l$, we get

$$W'_{DA(W)[i]}[SA(W)[i]-1] = W'_i[|W_i|-1]$$

and with the definition of W'_i, we get

$$W'_i[|W_i|-1] = S[PSA(i)-1] = S[SA(S)[i]-1] = BWT(S)[i].$$

In the case $i \geq l$, we get

$$W'_{DA(W)[i]}[SA(W)[i]-1] = W'_{word(SA(S)[i-l])}[position(SA(S)[i-l])-1].$$

Next, if $i < 2l$, we have $i-l < l$, so $SA(S)[i-l] \in PSA(S)$. We can use Proposition 2 now: $position(SA(S)[i-l]) = 0$, hence

$$W'_{DA(W)[i]}[SA(W)[i]-1] = W'_{word(SA(S)[i-l])}[0-1] = \#_{word(SA(S)[i-l])}$$

Last, if $i \geq 2l$, so $SA(S)[i-l] \notin PSA(S)$. There is exactly one $j < l$, such that $SA(S)[i-l] \in [\Omega(j), PSA(j)-1]$. Then, $position(SA(S)[i-l]) = \Omega(j) - SA(S)[i-l]$ and $position(SA(S)[i-l]-1) = \Omega(j) - SA(S)[i-l]-1$ due to Proposition 3.

$$\begin{aligned} BWT(W)[i] &= W'_{word(SA(S)[i-l])}[position(SA(S)[i-l])-1] \\ &= S[\Omega(j), PSA(j)-1][SA(S)[i-l]-1-\Omega(j)] \\ &= S[\Omega(j)+SA(S)[i-l]-1-\Omega(j)] \\ &= S[SA(S)[i-l]-1] \\ &= BWT(S)[i-l]. \end{aligned}$$

In the proofs of Theorem 1 and 5, we have only shown the correctness of the partition using a single word S, but the proofs did not use the limitation that only one string S was given. The presented partitioning can also transform a collection of strings $S = \{S_0, \ldots, S_n\}$ into a larger collection of shorter words. The necessary changes for partitioning a collection of strings is to use a suffix array and a document array of S instead of using only the suffix array of S and they include a set of \$ symbols to terminate the strings and a set of # symbols for partitioning the strings into words. Hereby, each \$ symbol is lexicographically larger than each # symbol. There is no change necessary in the proof steps. Note that our partDNA implementation can partition a collection S of strings as input.

B On the Size of the Reduced Problem compared to SA-IS

Theorem 2. *Each position $p > 0$ with either $S[p] = A = S[p+1] = \cdots = S[p+d-1]$ and $S[p+d] \notin \{\$, A\}$ and $S[p-1] \neq A$, or $S[p] = \$$ is a left-most S-type (LMS) position (according to the definition in SA-IS [15]).*

Proof. If $S[p] = \$$: If $p = 0$ then $S = \$$. If $p > 0$, then $S[p-1] \neq \$$ and thereby, $S[p-1] > S[p]$. Then $p-1$ is L-Type and p is S-Type by definition, which means that p is a LMS position.

Next, $S[p] = A$. $\$ \neq S[p-1] \neq A$, so $S[p-1] \in \{C, G, T\}$. We get $S[p-1] > S[p]$, so $p-1$ is L-Type. Because $S[p+d] \notin \{\$, A\}$, $p+d-1$ is S-Type. Finally, $S[p] = A = S[p+1] = \cdots = S[p+d-1]$ implies that the type of p is equal to $p+1$ is equal to ... is equal to $p+d-1$, so p is S-Type and LMS-position.

We conclude that our reduced problem is smaller or equal to the recursive problem of SA-IS because our reduced problem contains only one character for each position p with $d > h$.

Approximate Longest Common Substring of Multiple Strings: Experimental Evaluation

Hamed Hasibi*, Neerja Mhaskar*, and William F. Smyth

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

Abstract. Determining an *Approximate Longest Common Substring* (ALCS) among a collection of strings $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$, $m \geq 2$, is especially significant in computational biology — for instance, when tracking related mutations across several genomic sequences. To address this, the Restricted k-t Longest Common Substring (*Rkt*-LCS) problem was introduced by Hasibi *et al.* in [12], along with several efficient solutions under various distance metrics. In this paper, we describe and evaluate two parallel strategies for computing the *Rkt*-LCS of a set $\boldsymbol{S}$ of strings under the Hamming distance metric. The first strategy parallelizes the computation of the core data structure introduced in [12], while the second makes use of a Graphics Processing Unit (GPU) that we demonstrate executes over a hundred times faster than its CPU counterpart.

Keywords: Approximate longest common substring, Hamming distance, parallel computation, longest common prefix, biological sequences.

1 Introduction

The Longest Common Substring (LCS) problem has been extensively studied for decades, with applications ranging from genome sequence comparison and plagiarism detection to compression algorithms. However, evolutionary variation necessitates algorithms that can tolerate mismatches. Hence, the approximate longest common substring (ALCS) problem — which seeks the longest substring appearing in strings while permitting a fixed number of operations such as mismatches, insertions, or deletions — is well-suited to comparative genomics. The distance metric d_δ measures the minimum number of allowed operations required to transform a string u to another string u', and denoted as $d_\delta(u, u')$. Recently, Hasibi *et al.* in [12] introduced several new variants of the ALCS problem for a set of strings $\boldsymbol{S}$. One of the variants is stated as follows:

Problem 1. [Restricted k-t Longest Common Substring (*Rkt*-LCS) [12]] Given integers $k, t, m \in \mathbb{N}$ with $1 \leq t \leq m$ and a set $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$ of strings, find a longest substring u taken from any string in $\boldsymbol{S}$ such that there exist t distinct strings $s'_1, \ldots, s'_t \in \boldsymbol{S}$ with corresponding substrings $u_1, \ldots, u_t$ satisfying $d_\delta(u, u_j) \leq k$ for every $j = 1, \ldots, t$.

In Problem 1, the distance metric $\delta = \{H, L, E\}$ denotes the Hamming, Levenshtein, and edit distances, respectively, which are defined in the next section. The authors present two $\mathcal{O}(N^2)$ and $\mathcal{O}(mN \log^k \ell)$ time sequential algorithms for $\delta = H$

* Corresponding author

Hamed Hasibi, Neerja Mhaskar, William F. Smyth: Approximate Longest Common Substring of Multiple Strings: Experimental Evaluation, pp. 56–68.
Proceedings of PSC 2025, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07461-9

and two $\mathcal{O}(k\ell N^2)$ and $\mathcal{O}(mN \log^k \ell)$ time sequential algorithms for $\delta \in \{L, E\}$, where ℓ is the length of each string and N is the total length of all strings.

In this paper, we present two parallel implementations for the Rkt-LCS problem under the Hamming distance metric ($\delta = H$), each designed for speed and scalability on large datasets. The first implementation uses p processors to lower the sequential $\mathcal{O}(N^2)$ runtime by a factor of p, whereas the second accelerates the computation still further via GPU[1] threads[2].

In Section 2, we introduce the relevant terminology. Section 3 reviews the LCS literature. Section 4 presents detailed descriptions of each implementation, while Section 5 provides a comparative analysis of their run times. Finally, we summarize in Section 6.

2 Preliminaries

A ***string*** s is a sequence of $n \geq 0$ ***letters*** drawn from a finite ordered ***alphabet*** Σ of size $\sigma = |\Sigma|$. A string s of ***length*** n is represented as an array $s[1..n]$ with elements from Σ. The length of a string s is denoted by $|s| = n$. A ***substring*** of s is defined as a contiguous sequence of characters within s. Specifically, given integers $1 \leq i \leq j \leq n$, a substring of s is denoted as $s[i..j] = s[i]s[i+1] \dots s[j]$. We say string s_1 ***occurs*** in string s_2 if there exists a substring $s_2[i..j]$ such that $s_2[i..j] = s_1$. A ***prefix*** of a string s is a substring that starts at position 1, i.e., $s[1..j]$ for some $1 \leq j \leq n$. Similarly, a ***suffix*** of a string s is a substring that ends at position n, i.e., $s[i..n]$ for some $1 \leq i \leq n$. Given two strings s_1 and s_2, each of length n, the **Hamming distance** $d_H(s_1, s_2)$ is the number of positions at which s_1 and s_2 differ. For any two strings s_1 and s_2 of arbitrary lengths, the ***edit distance*** $d_E(s_1, s_2)$ is the minimum cost over all sequences of edit operations that transform s_1 to s_2. In the case that each edit operation has unit cost, the edit distance is called ***Levenshtein distance*** and denoted $d_L(s_1, s_2)$. For $1 \leq i' \leq |s_1|$ and $1 \leq j' \leq |s_2|$, $LCP^{H,k}_{(s_1,s_2)}[i', j']$ is defined as the length of the longest common prefix (LCP) between the suffixes $s_1[i'..|s_1|]$ and $s_2[j'..|s_2|]$, allowing for at most k mismatches under Hamming distance. $MaxLCP^{H,k}_{(s_i,s_j)}$ is defined as an array of length $|s_i|$, where each entry $MaxLCP^{H,k}_{(s_i,s_j)}[i']$ stores the maximum value of $LCP^{H,k}_{(s_i,s_j)}[i', j']$ over all $1 \leq j' \leq |s_j|$. In other words, $MaxLCP^{H,k}_{(s_i,s_j)}$ stores the maximum value in each row of the $LCP^{H,k}_{(s_i,s_j)}$ table. The values of $LCP^{H,k}_{(s_1,s_2)}$ and $MaxLCP^{H,k}_{(s_i,s_j)}[i']$ for $s_1 =$ `ACGTA` and $s_2 =$ `ACGACA` with $k = 1$ are shown in Table 1.

3 Literature Review

The LCS problem has been investigated under four categories:

1. Exact LCS (ELCS) of two strings
2. Exact LCS of multiple strings

[1] A Graphics Processing Unit (GPU) is a high-throughput, multi-core processor designed for parallel computation, ideal for tasks like machine learning, data-parallel algorithms, and high performance computing [8].

[2] A GPU thread is a lightweight execution unit that runs a single instance of a kernel, working in parallel with thousands of other threads to process different pieces of data simultaneously [8].

	A	C	G	A	C	A
A	4	1	1	3	1	1
C	1	3	1	1	2	1
G	1	1	2	1	1	1
T	1	1	2	1	2	1
A	1	1	1	1	1	1

$MaxLCP^{H,1}_{(s_1,s_2)}$
4
3
2
2
1

Table 1: $LCP^{H,1}_{(s_1,s_2)}$ for $s_1 =$ `ACGTA` (rows) and $s_2 =$ `ACGACA` (columns). For example, element [2,2]=3 is the length of the longest common prefix of suffixes $s_1[2..] =$ `CGTA` and $s_2[2..] =$ `CGACA` with up to 1 mismatch. The array to the right is $MaxLCP^{H,1}_{(s_1,s_2)}$.

3. Approximate LCS (ALCS) of two strings
4. Approximate LCS of multiple strings

The LCS investigation started with the ELCS problem for two or more strings [2,5,7,15,19]. The ELCS for two strings is defined as follows: given two strings s_1 and s_2 of length n, locate the longest substring that appears in both strings. For a constant-size alphabet, Weiner obtained a linear-time solution [19]. Farach later proved that, even when the alphabet is unbounded, a suffix tree can be built in overall $\mathcal{O}(n)$ time plus the time to sort the characters, furnishing an $\mathcal{O}(n)$ algorithm on the word-RAM for polynomially bounded integer alphabets [7]. Building on Farach's construction, Charalampopoulos *et al.* showed that the problem over an alphabet $[0, \sigma)$ can be solved in $\mathcal{O}(n \log \sigma / \sqrt{\log n})$ time and $\mathcal{O}(n/\log_\sigma n)$ space whenever $\log \sigma = o(\sqrt{\log n})$ [5].

The ELCS for multiple strings is defined as follows: given a string set $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$, m positive integers $x_1, x_2, .., x_m$, and t where $1 \leq t \leq m$, find a longest substring u of any string in $\boldsymbol{S}$ for which there are at least t strings $s_{i_1}, s_{i_2}, \ldots, s_{i_t}$ $(1 \leq i_1 < i_2 < \cdots < i_t \leq m)$ such that u occurs at least x_{i_j} times in s_{i_j} for each j with $1 \leq j \leq t$ [2]. Lee and Pinzon solved the problem in $\mathcal{O}(N)$ time, where $N = \sum_{i=1}^{m} |s_i|$ [15]. Later, Arnold and Ohlebusch showed an $\mathcal{O}(N)$ time solution for this problem for all $1 \leq t \leq m$ [2]. Recently, the decision version of the ELCS problem for two strings has been studied in the quantum model by Jin and Nogler [13]. Given two strings, they propose a quantum algorithm that decides whether there exists an ELCS of length d in $\tilde{\mathcal{O}}\big(n^{2/3}/d^{1/6-o(1)}\big)$[3] time complexity.

The *ALCS under Hamming distance for two strings* is defined as follows: look for the longest substring that appears in both s_1 and s_2 while permitting at most k mismatches. For $k = 1$, Babenko and Starikovskaya achieved an $\mathcal{O}(n^2)$-time, $\mathcal{O}(n)$-space algorithm (2011) [3]; Flouri *et al.* improved this to $\mathcal{O}(n \log n)$ time with the same space (2015) [9]. Leimeister and Morgenstern (2014) were the first to study the case $k > 1$, proposing a greedy heuristic solution [16]. Subsequent work yielded faster worst-case bounds: Flouri *et al.* (2015) gave a concise $\mathcal{O}(n^2)$ algorithm using only constant extra space; Grabowski (2015) produced two output-dependent algorithms running in $\mathcal{O}\big(n\big((k+1)(\ell_0+1)\big)^k\big)$ and $\mathcal{O}\big(n^2k/\ell_k\big)$ time, where ℓ_0 is the ELCS length and ℓ_k is the ALCS length [11]. Abboud *et al.* (2015) developed a randomized algorithm whose running time is $k^{1.5}n^2/2^{\Omega(\sqrt{(\log n)/k})}$ [1]. Thankachan *et al.* (2016) achieved $\mathcal{O}(n \log^k n)$ time and $\mathcal{O}(n)$ space [18]. Charalampopoulos *et al.* (2018) showed that if the answer

[3] $\tilde{\mathcal{O}}(\cdot)$ hides a polylog(n) factor.

length is $\Omega(\log^{2k+2} n)$, the problem can be solved in linear time and space [4]. Kociumaka *et al.* (2019) proved, under the Strong Exponential Time Hypothesis, that no strongly subquadratic algorithm exists for $k = \Omega(\log n)$ [14]. Finally, Charalampopoulos *et al.* (2021) obtained an $\mathcal{O}(n \log^{k-1/2} n)$-time, $\mathcal{O}(n)$-space algorithm [5].

The *ALCS under edit distance* problem for two strings resembles the Hamming distance variant, but also permits insertions and deletions in addition to substitutions. Abboud *et al.* (2015) gave a randomized solution with run time $k^{1.5}n^2/2^{\Omega(\sqrt{(\log n)/k})}$ [1]. Thankachan *et al.* (2018) later proposed an $\mathcal{O}(n \log^k n)$-time, $\mathcal{O}(n)$-space algorithm for this edit-distance variant [17].

4 Parallel Implementations of *Rkt*-LCS

In this section, we first give a brief summary of the data structure computed to solve the *Rkt*-LCS problem in [12], and then present the details of our two new parallel implementations. Finally, we review the feasibility of one of the existing relevant implementations for computing *Rkt*-LCS.

The sequential algorithm proposed in [12] compares the longest common prefix of every pair of suffixes across all strings by using a data structure called *LengthStat* (see Definition 2), solving the *Rkt*-LCS problem under Hamming distance in $\mathcal{O}(N^2)$ time, where N is the total length of the input strings (see [12], Theorem 3).

LengthStat is a lightweight, statistical data structure that summarizes the distribution of k-approximate substring matches at different lengths, enabling rapid filtering of candidate substrings for *Rkt*-LCS [12]. It is defined to facilitate computing the answer to the *Rkt*-LCS problem: each entry records whether a length-l prefix of a suffix of s_i occurs in another string s_j with up to k mismatches.

Definition 2 (LengthStat [12]). *Let $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$ be a set of strings. For every (i, x) pair with $1 \le i \le m$ and $1 \le x \le |s_i|$, define the $LengthStat^k_{(i,x)}$ table as follows:*

$$LengthStat^k_{(i,x)}[\,l, j\,] \;=\; \begin{cases} 1 & \text{if for some } y\ (1 \le y \le |s_j|),\ LCP^{H,k}_{(s_i,s_j)}[x, y] \;\ge l \\ 0 & \text{otherwise} \end{cases}$$

where $1 \le j \le m$ indexes the strings $\boldsymbol{S}$ and $1 \le l \le |s_i| - x + 1$ is the prefix length.

The matrix is augmented with a final column $LengthStat^k_{(i,x)}[\,l, m+1\,]$ storing, for each row l, the sum of its first m entries, i.e. the number of strings in $\boldsymbol{S}$ that share with $s_i[x..]$ a prefix of length at least l under k-mismatch Hamming distance.

To reduce the storage requirements of the LCP_k tables, in both our implementations, we derive *LengthStat* from the $MaxLCP^{H,k}_{(s_i,s_j)}$ arrays as follows:

$$LengthStat^k_{(i,x)}[l, j] = \begin{cases} 1, & \text{if } MaxLCP^{H,k}_{(s_i,s_j)}[x] \ge l \\ 0, & \text{otherwise} \end{cases}$$

In this section, we present two parallel implementaions for computing the *Rkt*-LCS of the string set $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$. Each algorithm returns the *Rkt*-LCS when its length is at least the user-specified threshold τ; otherwise it outputs `Null`. Although the *LengthStat* definition is for strings of fixed length ℓ, our implementations allow for different length strings in $\boldsymbol{S}$. For ease of explanation, however, we provide results here only for the fixed length case.

1. **Parallel CPU algorithm.** We parallelize the $\mathcal{O}(N^2)$ sequential algorithm proposed by [12] across p processors, achieving an expected runtime of $\mathcal{O}(\frac{N^2}{p})$. Compared with the sequential algorithm, doubling the number of processors approximately yields a two-fold speed-up; furthermore, unlike the implementation of Chockalingam *et al.* [6] discussed later in this section, our runtime is not exponential in k, the number of permissible mismatches.
2. **Parallel GPU-accelerated algorithm.** This GPU implementation further reduces the effective run time of CPU implementation and scales well for large N. By offloading the computation of multiple $MaxLCP^{H,k}$ arrays to the GPU, this implementation achieves over hundred times improvement in runtime over the parallel CPU version.

4.1 Parallel CPU algorithm

In this parallel implementation, the *Rkt*-LCS computation for the string set $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$, where each string has length ℓ, is distributed across p processors. We assume, for simplicity, that the m input strings are evenly divisible among the p processors (i.e., $p \mid m$), though this is not strictly required. Each processor P_r $(1 \leq r \leq p)$ is assigned exactly $\frac{m}{p}$ consecutive strings of $\boldsymbol{S}$; that is, the set of strings $\boldsymbol{S}_r = \{s_i | s_i \in \boldsymbol{S} \wedge i = (r-1) \cdot \frac{m}{p} + t', 1 \leq t' \leq \frac{m}{p}\}$ are assigned to processor P_r.

Processor P_r computes all the longest substrings in any $s_i \in \boldsymbol{S}_r$ that occur with at most k mismatches in at least t strings in entire $\boldsymbol{S}$. For each $s_i \in \boldsymbol{S}_r$, this is accomplished implicitly via the computation of the $LengthStat^k_{(p,i)}[l, j]$ tables (see Algorithm 1). Then, for each $s_i \in \boldsymbol{S}_r$, processor P_r computes the longest substring of s_i that satisfies the *Rkt*-LCS criteria (i.e., occurring in at least t other strings with at most k mismatches), identifying it as a *candidate* *Rkt*-LCS, denoted as C_i (as shown in Algorithm 2). Thus, each processor computes $\frac{m}{p}$ candidates, and across all processors, a total of m candidates for *Rkt*-LCS, $C_1, C_2, \ldots, C_m$, are generated. The candidate with the greatest length among them is reported as the global *Rkt*-LCS. Under ideal load balancing and negligible communication overhead, this strategy reduces the sequential $\mathcal{O}(N^2)$ running time to $\mathcal{O}(N^2/p)$.

By Definition 2, the last column of the $LengthStat^k_{(i,p)}[l, m+1]$ table stores the count of strings in $\boldsymbol{S}$ that contain a k-mismatch occurrence of $s_i[p, p+l-1]$. As shown in Algorithm 1, for efficiency, we do not store the entire *LengthStat* table. Instead, the last column of this table is stored as a key-value pair data structure named `LS`, where the tuple (i, p, l) is the key and `count` is the value. The elements of the tuple i, p, and l refer to the string index of the string $s_i \in \boldsymbol{S}$, the starting position of s_i, and the prefix length of the *p-th* suffix of s_i, respectively. For instance, the entry `((1,2,4),5)` in `LS` states that the substring $s_1[2\,..\,2+4-1]$ occurs with at most k mismatches in five strings of the set $\boldsymbol{S}$. Storing `LS` for all strings requires $\mathcal{O}(m\ell^2)$ space, whereas retaining the entire *LengthStat* tables in memory requires $\mathcal{O}(m^2\ell^2)$ space, where ℓ is the length of each string in $\boldsymbol{S}$.

Line #5 of Algorithm 1 calls the `COMPUTE_MAXLCP_K`(s_i, s_j, k, τ) function that computes $MaxLCP^{H,k}_{(s_i,s_j)}[0..|s_i| - \tau]$. The implementation of this function is adapted from Flouri *et al.* [9] with a single modification: once the entire matrix $LCP^{H,k}_{(s_i,s_j)}$ has been computed, the $MaxLCP^{H,k}_{(s_i,s_j)}$ array is obtained by taking the maximum value in

each row of $LCP^{H,k}_{(s_i,s_j)}$. This reduces the peak memory usage by freeing each $LCP^{H,k}$ table after computing its corresponding $MaxLCP^{H,k}$.

Algorithm 1 COMPUTE_LS_KEYVALUES($s_i, \{s_1, \ldots, s_m\}, k, \tau$)

```
1: Define LS: (key : i, position, length, value : count)
2: Initialize empty array MaxLCP of length |s_i| − τ + 1
3: for j ← 1 to m do
4:     if j = i then skip                          ▷ Skip comparing s_i to itself
5:     MaxLCP ← COMPUTE_MAXLCP_K(s_i, s_j, k, τ)
6:     for p ← 0 to |s_i| − τ do
7:         ℓ ← MaxLCP[p]
8:         for l ← τ to ℓ do
9:             entry.key ← (i, p, l)
10:            found ← false
11:            if entry.key exists in LS then
12:                entry.count++
13:                found ← true
14:                break
15:            if not found then
16:                Add (entry.key,1) to LS
17: return LS
```

To determine the global *Rkt*-LCS, each processor P_r examines LS for each string $s_i \in \boldsymbol{S}_r$ and extracts the pairs (i, p, l) with the largest l such that the associated count satisfies count $\geq t$. As mentioned above, we refer to such a pair as a *candidate* *Rkt*-LCS originating from s_i (C_i). This procedure is detailed in Algorithm 2.

As noted earlier, to enhance scalability and performance, the candidate computation procedure (Algorithm 2) is parallelized. Each processor is assigned a subset of indices i — that is, a consecutive range of strings — and independently computes the *Rkt*-LCS candidates for each strings is $\boldsymbol{S}_r$; that is, processor P_r invokes Algorithm 2 exactly $|\boldsymbol{S}_r|$ times, once for each string in $\boldsymbol{S}_r$. Finally, all candidates $C_1, C_2, \ldots, C_m$ are examined, and the longest of them is returned as the final *Rkt*-LCS. Since these computations are fully independent, the overall parallel time complexity is:

$$\mathcal{O}\left(\frac{m}{p} \cdot m\ell^2\right) = \mathcal{O}\left(\frac{N^2}{p}\right),$$

where N is the total length of the strings in $\boldsymbol{S}$ and ℓ is the length of each string.

Algorithm 2 COMPUTE_CANDIDATE_RKT_LCS($s_i, \{s_1, s_2, \ldots, s_m\}, k, t, \tau$)

```
1: results ← COMPUTE_LS_KEYVALUES(s_i, {s_1, s_2, ..., s_m}, k, τ)
2: if results = null then
3:     return (i = −1, position = −1, length = −1)
4: C_i ← (−1, −1, −1)
5: max_length ← 0
6: for each entry in results do
7:     if entry.count ≥ t and entry.key.length > max_length then
8:         C_i ← entry.key
9:         max_length ← entry.key.length
10: return C_i
```

4.2 Parallel GPU-accelerated algorithm

This implementation focuses on improving the computation of the $MaxLCP^{H,k}$ tables using GPU threads. Algorithms 1 and 2 are then used to compute the Rkt-LCS solution for the set $\boldsymbol{S}$.

In our parallel scheme, the strings in $\boldsymbol{S}$ are evenly divided among the p processors. Let $\boldsymbol{S}_r$ be the set of substrings assigned to processor P_r (as in Section 4.1). Then, each processor P_r off-loads every string in its assigned subset of strings $\boldsymbol{S}_r$ to the GPU for further processing. We compute $MaxLCP^{H,k}(s_i, S_{\text{buffer}})$ for every $s_i \in \boldsymbol{S}_r$, where $S_{\text{buffer}} = s_1 \cdot s_2 \cdots s_{i-1} \cdot s_{i+1} \cdots s_m$ is the flattened concatenation of all strings in $\boldsymbol{S}$ except s_i. In other words, $MaxLCP^{H,k}_{(s_i,S_{\text{buffer}})}$ is the concatenation of the $MaxLCP^{H,k}_{(s_i,s_j)}$ arrays for all $1 \le j \le m$ with $j \neq i$. The workload distribution, together with the launch time of every GPU task, is illustrated below. We explain below why $\mathcal{O}(m\ell)$ threads are created for each s_i.

$$\begin{cases} P_1 \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_1,S_{buffer})} \text{ at } t^1_1 \\ P_1 \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_2,S_{buffer})} \text{ at } t^1_2 \\ \quad \vdots \\ P_1 \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_{m/p},S_{buffer})} \text{ at } t^1_{m/p} \end{cases}$$

$$\vdots$$

$$\begin{cases} P_p \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_{m-m/p+1},S_{buffer})} \text{ at } t^p_1 \\ P_p \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_{m-m/p+2},S_{buffer})} \text{ at } t^p_2 \\ \quad \vdots \\ P_p \text{ invokes } \mathcal{O}(m\ell) \text{ threads for } MaxLCP^{H,k}_{(s_m,S_{buffer})} \text{ at } t^p_{m/p} \end{cases}$$

Each processor P_r invokes $\mathcal{O}(m\ell)$ GPU threads for each $s_i \in \boldsymbol{S}_r$ in order to compute $MaxLCP^{H,k}_{(s_i,S_{\text{buffer}})}$. These threads are created for a specific string $s_i \in \boldsymbol{S}_r$ and execute in parallel. However, the set of GPU threads created by P_r for the next string $s_{i+1} \in \boldsymbol{S}_r$ can only be created after the GPU threads created for the previous string s_i finish their execution. Let t^r_c denote the launch time of the c-th GPU task issued by processor P_r ($1 \le r \le p$ and $1 \le c \le m/p$); these times then satisfy

$$t^1_1 < t^1_2 < \cdots < t^1_{m/p},\ \ldots,\ t^p_1 < t^p_2 < \cdots < t^p_{m/p}.$$

Recall that in Section 4.1 the array $MaxLCP^{H,k}$ was computed pairwise for two strings s_i and s_j ($1 \le j \le m$). In the GPU version, we replace the second string by the entire S_{buffer}, processing each s_i against all strings in $\boldsymbol{S}$ except s_i. This choice maximizes GPU throughput on a large, contiguous workload and avoids repeated host–device transfers.

In Algorithm 3, each GPU thread is determined by the string index j — where $j \in \{1..m\} \setminus \{i\}$ denotes the index of a string in S_{buffer} — and the starting position $start$ in s_i. Thread $(j, start)$ computes $MaxLCP^{H,k}_{(s_i,s_j)}[start]$. In lines 8-15, the kernel first compares substrings of length τ, allowing at most k mismatches. In lines 18-24,

if this initial match is valid, the comparison continues beyond τ, extending as far as possible while ensuring that the mismatch count does not exceed k, and the substring bounds are not crossed. The longest valid match found by the thread is tracked in variable *longest*. If the matched length is at least τ, the result is stored as a tuple $(j, start, longest)$. This tuple is identical to (i, p, l) tuple stored in the LS key-value data structure in the CPU implementation (See Section 4.1).

Each processor P_r calls Algorithm 3 $|\boldsymbol{S}_r|$ times, and each algorithm call creates $c(m-1)(\ell-\tau+1) = \mathcal{O}(m\ell)$ GPU threads[4]. In other words, in order to compute $MaxLCP^{H,k}_{(s_i,S_{buffer})}$, we spawn $\mathcal{O}(m\ell)$ threads for each string s_i $(1 \le i \le m)$.

Over the entire computation, $\mathcal{O}(m^2\ell)$ GPU threads are created to obtain the *Rkt*-LCS. After computing all $MaxLCP^{H,k}$ arrays, we compute the candidate *Rkt*-LCS set $\{C_1, C_2, \ldots, C_m\}$, as shown in Algorithm 2 (which begins by executing Algorithm 1).

Algorithm 3 compute_MaxLCP$_k$_Kernel (CUDA)

```
1: Input: s_i, S_buffer, k, τ
2: Create GPU thread for each (j, start) pair
3: for each GPU thread (j, start) do
4:     longest ← 0
5:     compute S_offset                          ▷ Starting position of string j in S_buffer
6:     for q1 = 0 to ℓ − 1 do
7:         mismatch_count ← 0
8:         for q2 = 0 to τ − 1 do
9:             if start + q2 ≥ ℓ or q1 + q2 ≥ ℓ then
10:                break
11:            if s_i[start + q2] ≠ S_buffer[S_offset + q1 + q2] then
12:                if mismatch_count = k then
13:                    break
14:                if mismatch_count < k then
15:                    mismatch_count ← mismatch_count + 1
16:        if q2 < τ then
17:            continue
18:        while start + q2 < ℓ and q1 + q2 < ℓ do
19:            if s_i[start + q2] ≠ S_buffer[S_offset + q1 + q2] then
20:                if mismatch_count = k then
21:                    break
22:                if mismatch_count < k then
23:                    mismatch_count ← mismatch_count + 1
24:            q2 ← q2 + 1
25:        if q2 > longest then
26:            longest ← q2
27:    if longest ≥ τ then
28:        store (j, start, longest)
```

Table 2 shows the threads created for the computation of $MaxLCP^{H,k}_{(s_i,S_{buffer})}$ where $s_i = \texttt{ATTTCG}$ and $S_{buffer} = \{\texttt{GTGGA}, \texttt{AGGGGAT}, \texttt{AGCGGA}\}$.

Finally, we review a closely related implementation by Chockalingam *et al.* [6], discuss its potential applicability to the *Rkt*-LCS problem and and explain why our parallel approaches are significantly more effective for solving the *Rkt*-LCS problem. Chockalingam *et al.* in [6], provide a $\mathcal{O}((\frac{N}{p}\log N + occ)\log^k N)$ running time parallel

[4] The constant parameter c is the thread-amplification factor: each task is fanned out into c parallel GPU threads that run together in a single group.

Thread	j	start	S_{buffer} string	s_i substring
0	0	0	GTGGA	ATTTCG
1	0	1	GTGGA	TTTCG
2	0	2	GTGGA	TTCG
3	0	3	GTGGA	TCG
4	1	0	AGGGGAT	ATTTCG
5	1	1	AGGGGAT	TTTCG
6	1	2	AGGGGAT	TTCG
7	1	3	AGGGGAT	TCG
8	2	0	AGCGGA	ATTTCG
9	2	1	AGCGGA	TTTCG
10	2	2	AGCGGA	TTCG
11	2	3	AGCGGA	TCG

Table 2: Thread assignment for `compute_MaxLCP_k_kernel` with s_i = `ATTTCG` and $S_{buffer} = \{s_1, s_2, s_3\} = \{$`GTGGA`, `AGGGGAT`, `AGCGGA`$\}$ where $\tau = 3$. For example, Thread 2 computes $MaxLCP^{H,k}_{(s_i,s_1)}[3]$.

algorithm, where p is the number of processors and *occ* is the number of occurrences reported for the following problem:

Problem 3. (All-pair k-Mismatch Maximal Common Substrings). Given a collection $\boldsymbol{S} = \{s_1, s_2, \ldots, s_m\}$ of m strings with total length N, a length threshold τ, and a mismatch threshold $k \geq 0$, report all k-mismatch maximal common substrings of length $\geq \tau$ between all pairs of strings in $\boldsymbol{S}$.

A pair of two equal-length substrings, $s_i[x..(x+\phi-1)]$ and $s_j[y..(y+\phi-1)]$, $1 \leq i, j \leq m$, form k-mismatch common substrings if the Hamming distance between them is at most k. Also, they are *maximal* if neither $s_i[(x-1)..(x+\phi-1)]$ and $s_j[(y-1)..(y+\phi-1)]$, nor $s_i[x..(x+\phi)]$ and $s_j[y..(y+\phi)]$, are a k-mismatch common substring pair. Coincidentally, the "longest" k-mismatch maximal common substring pair computed by this solution is an Rkt-LCS where $t = 2$. Unfortunately, using the approach and implementation of [6] to compute the Rkt-LCS is impractical for the following reasons:

1. For $k > 2$ and large N, their solution's running time grows exponentially with k.
2. In addition, we need all k-mismatch common substrings (not only *maximal* ones) to compute Rk-LCS. Therefore, every substring of all maximal k-mismatch common substring needs to be evaluated (see Lemma 4).
3. The number of k-mismatch common substrings becomes extremely large even for a small number of relatively repetitive sequences over a small alphabet (see Lemma 5). On the other hand, $LCP^{H,k}_{(s_1,s_2)}$ needs $\mathcal{O}(\ell^2)$ space. Therefore, computing the k-mismatch common substrings is not a viable option for large strings.

Lemma 4. *Assume that substring u is an Rkt-LCS of the set $\boldsymbol{S}$, and substrings $u'_1, \ldots, u'_t$ are the corresponding k-mismatch occurrences of u in t strings of $\boldsymbol{S}$. Then at least one pair (u, u'_i) is a k-mismatch maximal common substring, $1 \leq i \leq t$.*

Proof. Assume, for the sake of contradiction, that all (u, u'_i) pairs are non-maximal. Then, each pair (u, u'_i) can be extended by at least one character to the left or right, yielding a longer Rkt-LCS. This contradicts the assumption that u is an Rkt-LCS. □

Lemma 5. *The number of all k-mismatch common substrings between s_1 and s_2, each of length ℓ, is $\mathcal{O}(\ell^3)$.*

Proof. Consider $s_1 = s_2 = a^\ell$, where $\ell > 0$. Let $\tau = 1$ and $k = 0$. Because every position $1..\ell$ in both strings contains the same symbol a, every pair of equal-length substrings is a 0-mismatch common substring and hence a k-mismatch common substring. For a fixed length ℓ' $(1 \leq \ell' \leq \ell)$, both s_1 and s_2 have $\ell - \ell' + 1$ substrings of length ℓ'. Therefore, the number of ordered pairs of length-ℓ' substrings is $(\ell-\ell'+1)^2$. Summing over all values of ℓ', we get $\sum_{\ell'=1}^{\ell}(\ell-\ell'+1)^2 = \frac{\ell(\ell+1)(2\ell+1)}{6} = \mathcal{O}(\ell^3)$. Hence, the total number of k-mismatch common substring pairs between two strings s_1 and s_2 is $\mathcal{O}(\ell^3)$. □

In contrast, the practical time complexity of our implementations is independent of k and, while quadratic in N, scales robustly even for very large inputs with available hardware resources.

5 Experimental Evaluation

In this section, we evaluate the relative speedup of proposed implementations. Our experiments were conducted on a server equipped with two 4.1 GHz 16-core Intel Xeon Gold 6426Y processors, with 250 GB of main memory running on the REHL 9 operating system. The server also features four NVIDIA H100 GPUs, each with 80 GB of memory. The dataset consists of two files, each containing 1,077,820 nucleotide sequences ($\Sigma = \{A, T, C, G\}$) of uniform length 51, formatted in `FASTQ`. Sequences are taken from soil samples from unidentified taxa[5] from an unpublished study. Our implementation is available online[6]. This repository is written using OpenMPI[7] 5.0.6 and CUDA[8] 12.8 libraries.

Table 3 shows the CPU and GPU runtime comparison for 5000 sequences; that is, for $m = 5000$. GPU implementation shows a 179× runtime improvement over CPU implementaion for the setting $(p, k, t, \tau) = (4, 10, 100, 30)$.

In Table 3a and Table 3b, CPU implementation shows the expected behavior of improved runtime with more processors. CPU processors operate independently without competing for shared resources. Additionally, there is no memory transfer overhead, unlike in the GPU implementation. Finally, the workload is purely computational and scales well across multiple cores. This implementation uses a queue-based approach to compute the $MaxLCP^{H,k}$ table [9]. As k increases, the runtime of the CPU implementation also increases due to several reasons. One key factor is the overhead of queue operations. Each mismatch during sequence comparison results in an enqueue operation. Since the queue size scales with k, larger values of k lead to more frequent queue operations and increased memory handling costs. Another reason is the extended comparison length permitted by higher k values. While the theoretical time complexity might not directly depend on k, practical runtime is affected by more frequent queue operations, longer comparison sequences, and heavier memory

[5] A taxonomic group of any rank, such as a species, family, or class.
[6] https://github.com/neerjamhaskar/Rkt-LCS
[7] OpenMPI is an open-source implementation of the MPI standard that enables parallel computing by coordinating communication between processes across distributed or shared memory systems [10].
[8] CUDA is NVIDIA's parallel-computing platform for general-purpose GPU programming [8].

Table 3: Runtime (in seconds) and Relative SpeedUp (RSU — relative to $Cores = 4$) comparison for $m = 5000$

(a) Parallel CPU, $t = 1000$, $\tau = 15$

Cores	$k = 1$		$k = 3$		$k = 10$	
	Time	RSU	Time	RSU	Time	RSU
4	138	1.00×	242	1.00×	705	1.00×
8	71	1.94×	122	1.98×	352	2.00×
16	40	3.45×	63	3.84×	181	3.89×
32	19	7.26×	42	5.76×	112	6.29×

(b) Parallel CPU, $t = 100$, $\tau = 30$

Cores	$k = 1$		$k = 3$		$k = 10$	
	Time	RSU	Time	RSU	Time	RSU
4	82	1.00×	146	1.00×	358	1.00×
8	44	1.86×	75	1.94×	182	1.96×
16	30	2.73×	39	3.74×	94	3.80×
32	17	4.94×	26	5.61×	66	5.42×

(c) Parallel GPU, $t = 1000$, $\tau = 15$

Cores	$k = 1$		$k = 3$		$k = 10$	
	Time	RSU	Time	RSU	Time	RSU
4	4	1.00×	3	1.00×	72	1.00×
8	5	0.80×	4	0.75×	37	1.94×
16	6	0.66×	6	0.50×	22	3.27×
32	12	0.33×	11	0.27×	21	3.42×

(d) Parallel GPU, $t = 100$, $\tau = 30$

Cores	$k = 1$		$k = 3$		$k = 10$	
	Time	RSU	Time	RSU	Time	RSU
4	2	1.00×	2	1.00×	2	1.00×
8	3	0.66×	3	0.66×	2	1.00×
16	4	0.50×	5	0.40×	5	0.40×
32	9	0.22×	9	0.22×	9	0.22×

management demands. These effects together reduce cache efficiency and increase the number of memory accesses.

As shown in Table 3c (except for $k = 10$) and Table 3d, the observed increase in runtime with more processors in the GPU implementation can be attributed to several key factors. Because each processor P_r launches its GPU tasks for each string in $\boldsymbol{S}_r$ sequentially, at most $\mathcal{O}(pm\ell)$ GPU threads are executed concurrently. Increasing the number of processors (p) increases GPU utilization but also increases the cost of coordination. Therefore, increasing p involves a trade-off between better GPU use and higher *work distribution overhead*. For a relatively small number of sequences and a large number of processors, work distribution overhead arises due to the use of OpenMPI for distributing work across processes. As the number of processes increases, each process handles fewer sequences, but the OpenMPI communication and coordination overhead grows proportionally. In addition, *memory transfer overhead* increases with more processes. Since each processor must transfer its data to the GPU, it results in greater cumulative overhead. However, as shown in Figure 2, increasing the number of processors for a larger number of sequences leads to lower runtime. In Table 3c, for $k = 10$, the runtime decreases as the number of processors increases, reflecting that larger k values incur greater computational work and thus benefit more from parallelization.

As shown in Figure 1, the GPU-accelerated implementation exhibits nearly constant runtime (for a relatively small number of sequences, perhaps 5k to 20k), while the CPU implementation shows a quadratic increase in runtime. As illustrated in Figure 2, by doubling the number of sequences, the quadratic runtime increase becomes more pronounced, and especially in a higher number of sequences (75k to 500k).

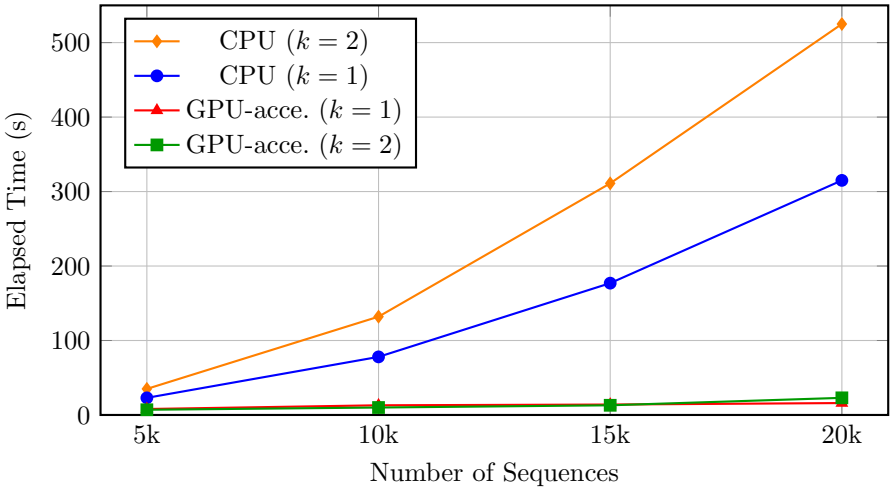


Figure 1: Runtime comparison for CPU and GPU-accelerated implementations with varying k on different sequence set sizes, $t = 1000$, $\tau = 15$, and $p = 32$

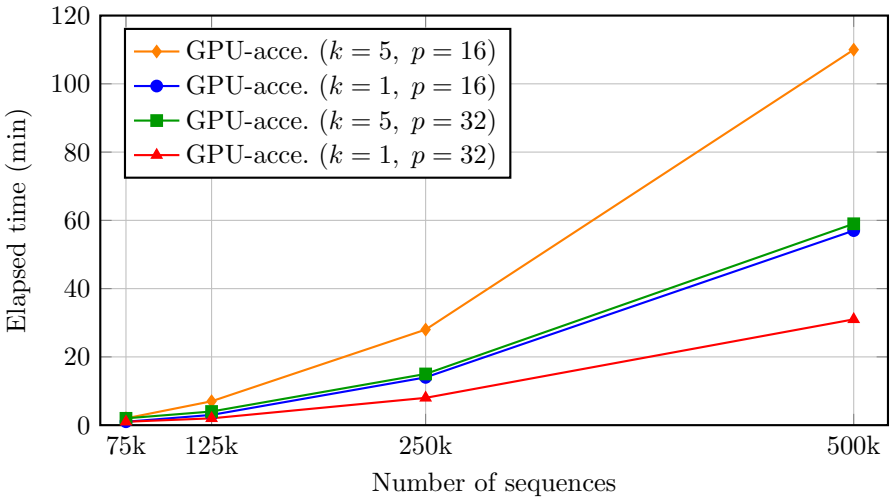


Figure 2: GPU-accelerated implementation runtime (in whole minutes) for different (k, p) settings, $t = 1000$, and $\tau = 15$

6 Conclusion

We introduced two implementations of the *Rkt*-LCS algorithm under Hamming distance: a CPU version with expected running time $\mathcal{O}(N^2/p)$ and a highly parallel GPU version, where both runtimes are independent of the mismatch parameter k. In our evaluations, we observed that off-loading the core computation of the $MaxLCP^{H,k}$ tables to GPU threads yields up to 179× speed-up over the CPU implementation. This improvement is expected to be higher for larger number of sequences.

As future work, the input sequences could first be clustered — e.g., with machine-learning techniques — and then losslessly compressed before computing the *Rkt*-LCS. Conversely, the *Rkt*-LCS itself can help detect similar fragments, which may then be compressed by differential similarity encoding. Another promising direction is to adapt the GPU version to incorporate the $LCP^{H,k}$ computation scheme of Flouri *et al.* [9], combined with FFT (Fast Fourier Transform) computation techniques.

Acknowledgments

The authors wish to thank Shutong Wu for useful discussions of parallel CPU implementation. The second and third authors are supported by Grant Nos. RGPIN-2024-06915 and RGPIN-2024-05921, respectively, of the Natural Sciences & Engineering Research Council of Canada (NSERC).

References

1. A. Abboud, R. R. Williams, and H. Yu: *More applications of the polynomial method to algorithm design*, in Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, 2015, pp. 218–230.
2. M. Arnold and E. Ohlebusch: *Linear time algorithms for generalizations of the longest common substring problem.* Algorithmica, 60(4) 2011, pp. 806–818.
3. M. A. Babenko and T. Starikovskaya: *Computing the longest common substring with one mismatch.* Probl. Inf. Transm., 47(1) 2011, pp. 28–33.
4. P. Charalampopoulos, M. Crochemore, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen: *Linear-time algorithm for long LCF with k mismatches*, in Annual Symposium on Combinatorial Pattern Matching, CPM 2018, vol. 105 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 23:1–23:16.
5. P. Charalampopoulos, T. Kociumaka, S. P. Pissis, and J. Radoszewski: *Faster algorithms for longest common substring*, in 29th Annual European Symposium on Algorithms, ESA 2021, vol. 204 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 30:1–30:17.
6. S. P. Chockalingam, S. V. Thankachan, and S. Aluru: *Sequential and parallel algorithms for all-pair* k*-mismatch maximal common substrings.* J. Parallel Distributed Comput., 144 2020, pp. 68–79.
7. M. Farach: *Optimal suffix tree construction with large alphabets*, in 38th Annual Symposium on Foundations of Computer Science, FOCS '97, IEEE Computer Society, 1997, pp. 137–143.
8. R. Farber: *CUDA Application Design and Development*, Elsevier, 2011.
9. T. Flouri, E. Giaquinta, K. Kobert, and E. Ukkonen: *Longest common substrings with k mismatches.* Inf. Process. Lett., 115(6-8) 2015, pp. 643–647.
10. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall: *Open MPI: goals, concept, and design of a next generation MPI implementation*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, vol. 3241 of Lecture Notes in Computer Science, Springer, 2004, pp. 97–104.
11. S. Grabowski: *A note on the longest common substring with* k*-mismatches problem.* Inf. Process. Lett., 115(6-8) 2015, pp. 640–642.
12. H. Hasibi, N. Mhaskar, and W. F. Smyth: *On the complexity of finding approximate LCS of multiple strings*, 2025, `https://arxiv.org/abs/2505.15992`.
13. C. Jin and J. Nogler: *Quantum speed-ups for string synchronizing sets, longest common substring, and* k*-mismatch matching.* ACM Trans. Algorithms, 20(4) 2024, pp. 32:1–32:36.
14. T. Kociumaka, J. Radoszewski, and T. Starikovskaya: *Longest common substring with approximately k mismatches.* Algorithmica, 81(6) 2019, pp. 2633–2652.
15. I. Lee and Y. J. Pinzón: *A simple algorithm for finding exact common repeats.* IEICE Trans. Inf. Syst., 90-D(12) 2007, pp. 2096–2099.
16. C. Leimeister and B. Morgenstern: *kmacs: the* k*-mismatch average common substring approach to alignment-free sequence comparison.* Bioinform., 30(14) 2014, pp. 2000–2008.
17. S. V. Thankachan, C. Aluru, S. P. Chockalingam, and S. Aluru: *Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis*, in Research in Computational Molecular Biology - 22nd Annual International Conference, vol. 10812 of Lecture Notes in Computer Science, Springer, 2018, pp. 211–224.
18. S. V. Thankachan, A. Apostolico, and S. Aluru: *A provably efficient algorithm for the* k*-mismatch average common substring problem.* J. Comput. Biol., 23(6) 2016, pp. 472–482.
19. P. Weiner: *Linear pattern matching algorithms*, in 14th Annual Symposium on Switching and Automata Theory, IEEE Computer Society, 1973, pp. 1–11.

Author Index

Proceedings of the Prague Stringology Conference 2025
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.
78 pages, first edition.

ISBN 978-80-01-07461-9

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