Succinct k-mer Set Representations Using Subset Rank Queries on the Spectral Burrows-Wheeler Transform (SBWT)

Jarno N. Alanko, Simon J. Puglisi, Jaakko Vuohtoniemi

May 19, 2022

Abstract

The k-spectrum of a string is the set of all distinct substrings of length k occurring in the string. This is a lossy but computationally convenient representation of the information in the string, with many applications in high-throughput bioinformatics. In this work, we define the notion of the *Spectral Burrows-Wheeler Transform* (SBWT), which is a sequence of subsets of the alphabet of the string encoding the k-spectrum of the string. The SBWT is a distillation of the ideas found in the BOSS and Wheeler graph data structures. We explore multiple different approaches to index the SBWT for membership queries on the underlying k-spectrum. We identify *subset rank queries* as the essential subproblem, and propose four succinct index structures to solve it. One of the approaches essentially leads to the known BOSS data structure, while the other three offer attractive time-space trade-offs and support simpler query algorithms that rely only on fast rank queries. The most general approach involves a novel data structure we dub the *subset wavelet tree*, which we find to be of independent interest. All of the approaches are also amendable to entropy compression, which leads to good space bounds on the sizes of the data structures. Using entropy compression, we show that the SBWT can support membership queries on the k-spectrum of a single string in $O(k)$ time and $(n+k)(\log\sigma + 1/\ln 2) + o((n+k)\sigma)$ bits of space, where n is the number of distinct substrings of length k in the input and σ is the size of the alphabet. This improves from the time $O(k\log\sigma)$ achieved by the BOSS data structure, while maintaining the same asymptotic space complexity of $O(n\log\sigma)$, albeit with smaller constant factors. We show, via experiments on a range of genomic data sets, that the simplicity of our new indexes translates into large performance gains in practice over prior art.

1 Introduction

The set of substrings of a given length k of a string S is called the *k-spectrum* of S. Indexing such spectra has been an important topic in bioinformatics in the past decade. For example, the k-spectrum and the associated de Bruijn graph is a central tool in genome assembly [7]. In metagenomics, k-spectra have found their place as a useful approximation of the sequence content of the sample, allowing rapid similarity estimation between data collected from sequencing estimates [19, 27]. In applications, typical values for k are in the range from 20 to 100.

There are multiple design goals for efficient representations of k-mer spectra. In general, the index should be small enough to fit in the main memory of a server machine, while offering fast support for *membership queries*, that is, queries asking whether a given k-mer (string of length k) is part of the spectrum. Additional query support may include querying for the neighbors of a k-mer, that is, k-mers that share a suffix or a prefix of length $k-1$ with the current k–mer. This allows fast simulation of the de Bruijn graph of the spectrum. The BOSS data structure [4] is a popular solution that meets all the above requirements. Other methods include hashing [22], Bloom filters [28] and the FM-index-based DBGFM structure [6]. Some more recent solutions emphasize the need for dynamic operations, allowing insertion or deletion of data on the index after it has been built [2, 8, 3, 1]. It is also desirable to be able to support attaching some satellite data to each k-mer, like is done in, e.g., colored de Bruijn graphs [16, 23, 15, 22, 20], which are now in widespread use. We refer the reader to the recent surveys of Chikhi [5] and Marchet et al. [21] for comprehensive surveys on existing methods.

In this work, we define a static representation of k-mer spectra which we call the *Spectral Burrows-Wheeler Transform*, or SBWT for short. The SBWT is an evolution of the BOSS data structure [4], which is an indexed representation of the edge-centric de Bruijn graph, based on a version of the Burrows-Wheeler transform. The SBWT differs from the BOSS in that it is node-centric, and more general – the

BOSS data structure can be seen as a particular implementation of the SBWT. The SBWT can also be seen as a specialization of the Wheeler graph framework [11] into k-spectra, taking full advantage of the properties of the special case.

The SBWT, which we define in Section 3, is a particular sequence of subsets from the alphabet of the input string. To implement k-mer membership queries on the SBWT, a form of *rank queries* on subset sequences is required. A subset rank query takes in a character c and an index i, and returns the count of how many of the first i subsets in the sequence of subsets contain c. We propose four possible index data structures for subset rank queries, leading to four different SBWT index structures, which we call ConcatSBWT, MatrixSBWT, SplitSBWT and SubsetWTSBWT. ConcatSBWT is a simplified version of the original BOSS representation, whereas the other three are novel variants offering different time-space tradeoffs. SubsetWTSBWT is based on a new data structure we call the *subset wavelet tree*, which is of independent interest. SplitSBWT uses a practical version of subset wavelet tree that is tailored for an SBWT of an input string with a small alphabet, such as the DNA alphabet. MatrixSBWT is a simple variant suitable for small alphabets, that is only slightly larger than the others in practice, but offers extremely fast subset rank queries and k-mer search operations.

We then show that it is possible to use entropy coding methods to compress the space of these data structures while retaining query support. In particular, we show that MatrixSBWT implemented with bit vectors compressed to the zeroth order entropy leads to a data structure taking 3.25 bits per k-mer on the DNA alphabet, matching the navigational lower bound of Chikhi et al. [6].

An important caveat is that the lower bound of Chikhi et al. is for an arbitrary set of k-mers, not for the spectrum of a single string. The space on a general alphabet of size σ is $(n+k)(\log\sigma+1/\ln 2)+o((n+k)\sigma)$, where n is the number of k-mers in the spectrum. The data structure can answer k-mer membership queries in $O(k)$ time, improving on the original BOSS data structure, which occupies the same asymptotic space, but takes $O(k\log\sigma)$ time for membership queries.

The index structures SplitSBWT and SubsetWTSBWT are aimed at occupying space that is *lower* than the navigational lower bound. This is achieved by exploiting the uneven distribution of the subsets in the subset sequence of the SBWT. We aim to compress the size of the data structures down to the zeroth order entropy of the *subset sequence*, where each subset is considered as a symbol. We show that this method allows us to get down to 2.44 bits per k-mer on an E. coli pangenome.

We also propose a compression-boosting algorithm that aims to minimize the entropy of the subset sequence by adjusting the sets without changing the underlying k–spectrum. We find that in practice the method quickly converges to a local minimum that has 0.25% lower entropy than the initial SBWT. We leave it as an open question whether this local minimum is optimal.

In practice, our methods lead to a radically new level of performance for succinct de Bruijn graphs, significantly outperforming the best previous approach [23] when space-usage is equated and simultaneously offering a range of attractive space-time tradeoffs. Two highlights are (1) an index that takes only 4 to 5 bits per k-mer on our genomic datasets, and is 90 to 112 times faster than the BOSS implementation of VARI and (2) an index taking only 2.6 to 3.35 bits per k-mer while being 19 to 26 times faster than VARI.

2 Preliminaries

Throughout we will consider a *string* $S = S[1..n] = S[1]S[2]\dots S[n]$ on an integer alphabet Σ of σ symbols. The *colexicographic order* of two strings is the same as the lexicographic order of their reverse strings. The *substring* of S that starts at position i and ends at position j, $j \geq i$, denoted $S[i..j]$, is the string $S[i]S[i+1]\dots S[j]$. If $i > j$, then $S[i..j]$ is the empty string ε. A suffix of S is a substring with ending position $j = n$, and a prefix is a substring with starting position $i = 1$. We use the term k-mer to refer to a (sub)string of length k.

A de Bruijn graph (dBG) of order k is directed labelled graph built from a set of k-mers S. There are two prevailing views of a dBG, and we define both here. In the *node-centric* dBG, the node set is given by S and there is an edge from node u to v iff the last $k-1$ symbols of u are equal to the first $k-1$ symbols of v. In an *edge-centric* dBG, the node set is given by the set of $(k-1)$-mers present in S, and, for every $x \in S$, there is an edge from $x[1..k-1]$ to $x[2..k]$. In other words, the k-mers of S are nodes in the node-centric dBG and edges in the edge-centric dBG. See Figure 1. Node-centric and edge-centric dBGs represent equivalent information, however, as we shall see, the ease of representing and navigating them can differ significantly.

A key tool in the design of succinct data structures is the support for the *query* operations rank, select, and access, on a bit string X of length n defined as follows (for $i \leq n$ and $x \in \{0,1\}$):

$$\begin{aligned} \mathsf{rank}_x(i) &= \text{number of } x\text{'s among the first } i \text{ bits of } X \\ \mathsf{access}(i) &= \text{value of the } i\text{th bit of } X \\ \mathsf{select}_x(i) &= \text{position of the } i\text{th } x \text{ in } X \end{aligned}$$

Classical techniques (see, e.g., [24]) require $n + o(n)$ bits to support each of the above queries in $O(1)$ time. However, the information theoretic lower bound on space usage for a bit string of length n having n_1 1s, is $\mathcal{B}(n, n_1) = \log \binom{n}{n_1} = n_1 \log \frac{n}{n_1}$ bits.

There are data structures that come within a lower order term of this lower bound while still supporting fast rank, select, and access operations. Perhaps the foremost of these, known as "RRR", is due to Raman, Raman, and Rao Satti [29], which takes space $\mathcal{B}(n, n_1) + o(n)$ and answers all queries above in $O(1)$ time. Fast implementations of RRR have been studied by several authors [25, 13].

Another notable compressed data structure for bit strings is the so-called "Elias-Fano" (or EF) scheme [31, 9, 10], which occupies $2n_1 + n_1\lceil\log(n/n_1)\rceil$ bits and supports rank in $O(\log(n/n_1))$ time and $\mathsf{select}_1(i)$ in $O(1)$ time, and tends to be faster than RRR in practice when applied to very sparse bit strings. Like RRR, the efficient implementation of EF has also received considerable practical attention [18, 26].

The rank, access and select queries are also sometimes needed on strings with an alphabet larger than 2. The wavelet tree data structure [14] supports these queries in $O(n \log \sigma)$ bits of space and $O(\log \sigma)$ time, where n is the length of the string and σ is the size of the alphabet.

In our analysis later in the paper, we will also make use of the entropy of a probability distribution p, which is denoted $H(p)$ and is defined as:

$$H(p) = -\sum_x p(x) \log p(x) \ , \tag{1}$$

where the sum is over the domain of p.

A useful form of the entropy for strings is the so-called zeroth-order empirical entropy, denoted $H_0(S)$ for a string S, or just H_0 when the context is clear. In particular,

$$H_0 = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}, \tag{2}$$

where $n = |S|$ is the length of S and n_c is the number of occurrences of the symbol c in S. We remark that $\mathcal{B}(n, n_1)$ is bounded above by nH_0.

3 The Spectral Burrows-Wheeler Transform

In this section we define the Spectral Burrows-Wheeler transform, and the spectrum membership query algorithm based on it. We begin with two basic definitions:

Definition 1. *(k-spectrum). The k-spectrum of a string T, denoted with $S_k(T)$, is the set of all k-mers of the string T.*

Definition 2. *(k-prefix set). The k-prefix set of a string T is defined as the left-padded set of prefixes $P_k(T) = \{\$^{k-i}T[1..i] \mid i = 0, \ldots, k-1\}$, where \$ is a special character not found in the alphabet, that is smaller than all characters of the alphabet.*

For example, for string T = TAGCAAGCACAGCATACAGA, we have:

$$S_3(T) = \{\mathrm{AAG}, \mathrm{ACA}, \mathrm{AGA}, \mathrm{AGC}, \mathrm{ATA}, \mathrm{CAA}, \mathrm{CAC}, \mathrm{CAG}, \mathrm{CAT}, \mathrm{GCA}, \mathrm{TAC}, \mathrm{TAG}\},$$

and

$$P_3(T) = \{\$\$\$, \$\$\mathrm{T}, \$\mathrm{TA}\}.$$

We are now ready to define the Spectral BWT.

Definition 3. *(Spectral BWT, SBWT). Let T be a string from an alphabet Σ of size σ. The spectral BWT of order k of T is a mapping from $S_k(T)$ to a sequence $X_1, X_2, \ldots X_n$ of subsets of Σ. The set X_i is defined as follows. Let x_i be the colexicographically i-th k-mer in $S_k(T) \cup P_k(T)$. If $x_i[2..k]$ is the colexicographically smallest k-mer in $S_k(T) \cup P_k(T)$ that has $x_i[2..k]$ as a suffix, then X_i is the set of last characters of k-mers $y \in S_k(T) \cup P_k(T)$ that have $x_i[2..k]$ as a prefix. Otherwise, x_i is an empty set.*

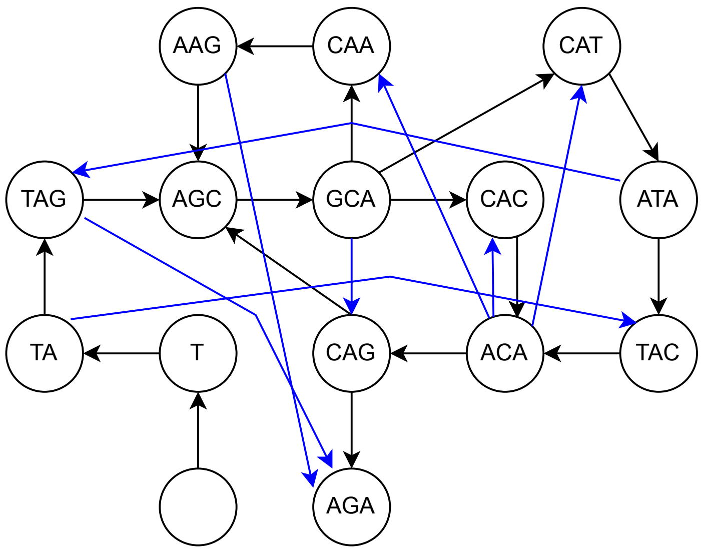


Figure 1: The de Bruijn graph for the string TAGCAAGCACAGCATACAGA. Black edges are in the edge-centric graph. Blue edges are only in the node centric graph

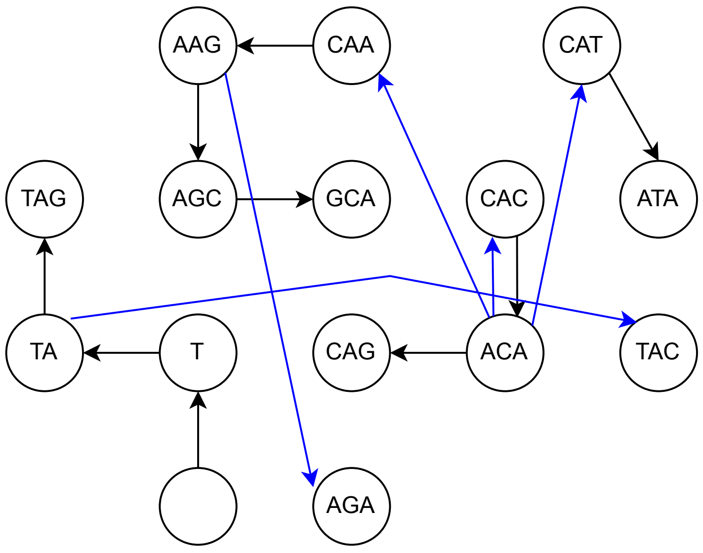


Figure 2: The de Bruijn graph of TAGCAAGCACAGCATACAGA after edge pruning. Every node except for the node of the empty string has exactly one incoming edge, such that the incoming path spells the colexicographically smallest incoming path in the graph prior to pruning.

Continuing for the previous example, the colexicographically ordered list of $S_3(T) \cup P_3(T)$ is:

$$\$\$\$, CAA, ACA, GCA, AGA, \$TA, ATA, CAC, TAC, AGC, AAG, CAG, TAG, \$\$T, CAT$$

and the SBWT is the sequence:

$$\{T\}, \{G\}, \{ACGT\}, \emptyset, \emptyset, \{CG\}, \emptyset, \{A\}, \emptyset, \{A\}, \{AC\}, \emptyset, \emptyset, \{A\}\{A\}$$

The sets in the SBWT represent the labels of outgoing edges in the node-centric de Bruijn graph, such that we only include outgoing edges from k-mers that have a different suffix of length $k-1$ than the preceeding k-mer in the colexicographically sorted list. See Figure 2. The padding of dollar-symbols in Definition 3 is a technical detail that is required to make the SBWT work. Alternatively, the sequence T could be made cyclic, and the need for $P_k(T)$ avoided.

We now describe how to implement efficient k-mer membership queries on the spectrum of the input string, using only the information encoded in the SBWT. Here it is beneficial to view the spectrum as a de Bruijn graph. We can think of the nodes as being ordered by the colexicographic order of the corresponding k-mers. We define an order for the edges such that the edges are sorted primarily by the edge label, and secondarily by the order of the origins of the edges. We denote with $R(e)$ the rank of edge e in this order. Because the graph is a de Bruijn graph, the indices of the destination nodes of the edges are in the same order as the ranks $R(e)$, that is, if $R(e_1) < R(e_2)$, then the destination of edge e_1 is larger than the destination of edge e_2.

Due to Definition 3, every node has exactly one incoming edge, except for the node corresponding to k-mer $\$^k$, which has no incoming edge. This is because when there are multiple nodes that could have an edge to the same node, we always use the node corresponding to the colexicographically smallest k-mer choice, and since the graph is built from a spectrum of a single string, every node has at least one candidate incoming edge (except for the node of k-mer $\$^k$). This means that the destination of edge e is the node with index $R(e) + 1$ in the sorted order.

Thus, the entire graph can be extracted from just the SBWT alone. The k-mer label of a node is spelled by any incoming path of length k to the node (the properties of de Bruijn graphs ensure that every incoming path of length k has the same label). This shows that the SBWT is invertible in the sense that is it possible to extract the original spectrum back from the SBWT.

This graph is also a Wheeler graph [11], which means that the generic Wheeler graph index could be used to index the graph. The Wheeler graph index however requires the storage of the sequence of indegrees and outdegrees of the nodes in the graph, whereas in the SBWT this is not required because every node apart from the node of the empty string has in-degree of exactly 1, and the sequence of outdegrees is already included in the sizes of the sets of the outgoing edge label sets. The most important conceptual advance over the Wheeler graph index, however, is a reformulation of the k-mer search problem in terms of *subset rank queries*.

Subset rank query: Let $X_1, \dots X_n$ be a sequence of subsets of an alphabet $\Sigma = \{1, \dots, \sigma\}$. A subset rank query takes as an input an index i and a character $c \in \Sigma$, and returns the number of subsets X_j with $j \leq i$ such that $c \in X_j$.

We now describe a k-mer search routine that uses subset rank queries as the only subroutine. The search routine works by searching the k-mer character by character from left to right, maintaining the interval of nodes that are suffixed by the prefix that has been processed so far. Suppose we have an interval $[i, j]$ of nodes suffixed by prefix α of the k-mer, and we want to find the interval $[i', j']$ of nodes suffixed by αc, where c is the next character in the k-mer. This is equivalent to following all edges labeled with c from the nodes in $[i, j]$. Due to the way the edges are defined, the end points of these edges are a contiguous range $[i', j']$ such that i' is the destination of the first outgoing edge labeled c from $[i..j]$, and j' is the destination of the last one. Let $C[c]$ be the number of edge labels with label smaller than c in the graph. Now we have the following formulas:

$$\begin{aligned} i' &= 1 + C[c] + \mathit{subsetrank}_c(i-1) + 1 \\ j' &= 1 + C[c] + \mathit{subsetrank}_c(j)\ , \end{aligned} \tag{3}$$

where $\mathit{subsetrank}_c$ is a subset rank query on the sequence of subsets in the SBWT. The "+1" at the start of the formulas is to skip over the node of $\$^k$. The values $C[c]$ can be precomputed for all characters $c \in \Sigma$. By iterating these formulas k times, we have the k-mer search routine. See Algorithm 1. This establishes the result below.

Algorithm 1 SBWT k-mer search query.
Input: k-mer S.
Output: The colexicographic rank of k-mer S in the underlying spectrum of the SBWT, or 0 if S is not in the spectrum.

```
function SEARCH($S$):
    $[\ell, r] \leftarrow [1, n]$
    for $i = 1, \ldots, k$ do
        $c \leftarrow S[i]$
        $1 + \ell \leftarrow C[c] + subsetrank_c(\ell - 1) + 1$
        $1 + r \leftarrow C[c] + subsetrank_c(r)$
        if $\ell > r$ then
            return 0
    return $\ell$.
```

Lemma 1. *The SBWT supports k-mer membership queries in $O(kt)$ time, where t is the time for a subset rank query.*

□

3.1 Extension to Multiple Strings

In this subsection, we define an extension of the SBWT to multiple input strings $T_1, \ldots T_m$. The easiest way to extend the SBWT to multiple strings would be to just replace the the spectrum $S_k(T)$ and the k-prefix set $P_k(T)$ in Definition 3 with the unions $S_k(T_1) \cup \ldots \cup S_k(T_m)$ and $P_k(T_1) \cup \ldots \cup P_k(T_m)$ respectively. However, if the input consists of a large number of short strings, such as DNA sequence reads, this method can introduce a lot of space overhead because every string T_i adds k padded prefixes to the union. On the other hand, the prefixes are only there to ensure that every k-mer has at least one incoming edge. Actually, we only need to include the k-prefix sets of those strings T_i where the leftmost $(k-1)$-mer of T_i does not appear as a suffix of any k-mer in $T_1, \ldots, T_m$. Let $R(T_1, \ldots, T_m)$ be the set of indices i such that T_i have this property, and let

$$P'_k(T_1, \ldots, T_m) = \left(\bigcup_{i \in R(T_1, \ldots, T_m)} P_k(T_i) \right) \cup \{\$^k\}$$

be the modified prefix set. The k-mer $\$^k$ is always added for convenience to match the property in the regular SBWT where the k-mer $\$^k$ always exists. Finally, let

$$S'_k(T_1, \ldots T_m) = \bigcup_{i=1}^{m} S_k(T_i) \; .$$

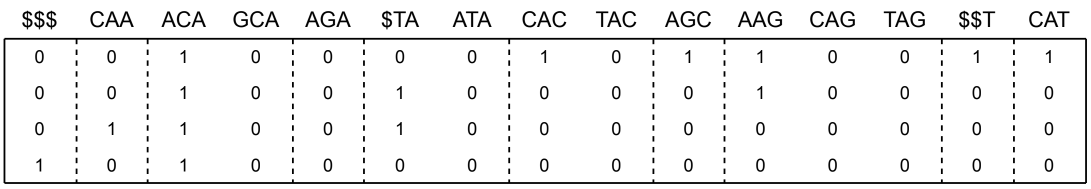
Now, we define the Multi-string Spectral BWT as follows:

Definition 4. *(Multi-SBWT). Let $\{T_1, \ldots T_m\}$ be a set of strings from an alphabet Σ of size σ and let $U = S'_k(T_1, \ldots T_m) \cup P'_k(T_1, \ldots T_m)$. The multi-SBWT of order k of $\{T_1, \ldots T_m\}$ is a sequence $X_1, X_2, \ldots X_n$ of subsets of Σ. The set X_i is defined as follows. Let x_i be the colexicographically i-th k-mer in U. If $x_i[2..k]$ is the colexicographically smallest k-mer in U that is suffixed by $x_i[2..k]$, then X_i is the set of last characters of k-mers $y \in U$ that are prefixed by $x_i[2..k]$. Otherwise, x_i is an empty set.*

All the properties required in the SBWT for the k-mer search in Algorithm 1 to work are preserved in this definition, so the k-mer search routine still works for the multi-SBWT without modifications.

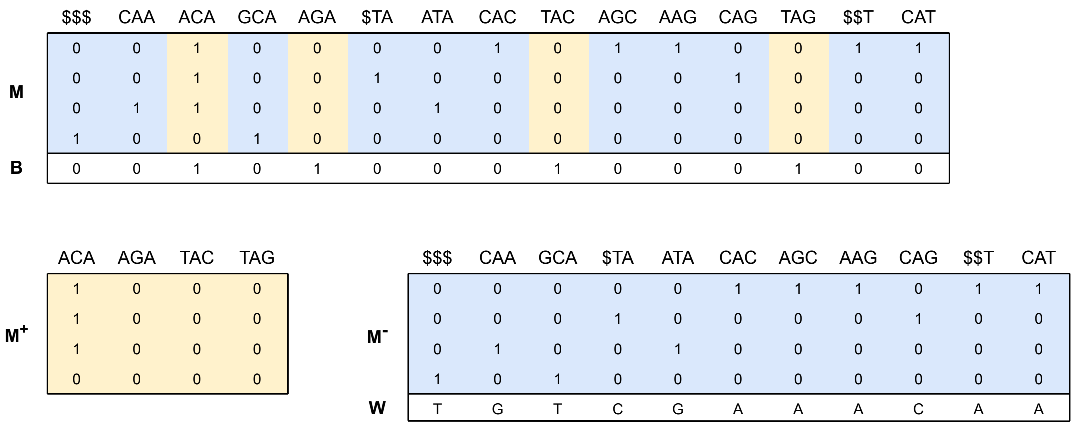
4 Data Structures for Subset Rank Queries

In this section, we propose four different succinct data structures for subset rank queries. Our treatment here is not intended to be exhaustive, but rather to demonstrate that an interesting range of space-time tradeoffs are possible. We use the notation from the previous section, that is, we are indexing a sequence $X_1, \ldots X_n$ of subsets of an alphabet $\Sigma = \{1, \ldots, \sigma\}$. A subset rank query takes as an input an index i and a character $c \in \Sigma$, and returns the number of subsets X_j with $j \leq i$ such that $c \in X_j$.



$$$	CAA	ACA	GCA	AGA	$TA	ATA	CAC	TAC	AGC	AAG	CAG	TAG	$$T	CAT
0	0	1	0	0	0	0	1	0	1	1	0	0	1	1
0	0	1	0	0	1	0	0	0	0	1	0	0	0	0
0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3: MatrixSBWT of TAGCAAGCACAGCATACAGA with $k = 3$. The dashed lines indicate borders of suffix groups. Two adjacent columns are in the same group if they have the same suffix of length $k-1$. Bits may be moved horizontally inside a suffix group without affecting the k-mer set encoded in the matrix.



	$$$	CAA	ACA	GCA	AGA	$TA	ATA	CAC	TAC	AGC	AAG	CAG	TAG	$$T	CAT
M	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1
	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0
	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
B	0	0	1	0	1	0	0	0	1	0	0	0	1	0	0

	ACA	AGA	TAC	TAG
M^+	1	0	0	0
	1	0	0	0
	1	0	0	0
	0	0	0	0

	$$$	CAA	GCA	$TA	ATA	CAC	AGC	AAG	CAG	$$T	CAT
M^-	0	0	0	0	0	1	1	1	0	1	1
	0	0	0	1	0	0	0	0	1	0	0
	0	1	0	0	1	0	0	0	0	0	0
	1	0	1	0	0	0	0	0	0	0	0
W	T	G	T	C	G	A	A	A	C	A	A

Figure 4: SplitSBWT of TAGCAAGCACAGCATACAGA. The bits are spread out inside a suffix group to maximize the number of singleton sets. The index consists of just B, M^+ and W.

4.1 Plain Matrix Representation

This data structure uses a binary matrix M of size $\sigma \times n$, such that $M[i][j] = 1$ iff subset X_j contains the i-th character in the alphabet. See Figure 3. The rows of the matrix are indexed for constant-time succinct rank queries. The subset rank query for the i-th character of the alphabet up to index j is answered in contant time with a rank query on row $M[i]$ up to index j.

4.2 Split Representation

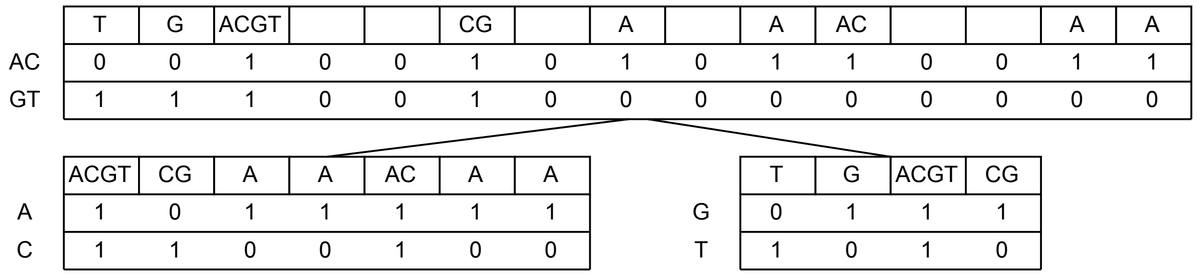
This is a version of the plain matrix representation, tailored for the use case of subset rank queries in the SBWT, exploiting the property that, in many use cases in genomics, most of the sets in the SBWT are singletons. Let M^- be the submatrix of matrix M in the plain matrix representation that contains only the columns of M with exactly one 1-bit set, and let M^+ be the submatrix of M containing the rest of the columns. Let B be a bit vector of length equal to the number of columns in M, marking with 1-bits which columns of M are in M^+.

We index both M^+ and M^- for character rank queries. Matrix M^+ is indexed like in the plain matrix representation. Matrix M^- on the other hand is replaced by a string W that is the concatenation of the labels corresponding to the bits in the columns. The string W is indexed as a wavelet tree. The bit vector B is indexed for rank queries. In summary, the final index consists of just B, M^+ and W and their rank support structures. See Figure 4.

Subset rank query for i-th character up to index j in the original subset sequence can now be answered by using a rank query on B to determine how many of the first j columns of M went to M^+ and M^-, then using the rank structures of M^+ and M^- to count the number of characters in the corresponding prefixes of columns in both matrices, and returning the sum of the two counts. That is, if $r = rank_1(B, j)$, then answer to the query is $rank_1(M^+[i], r) + rank_i(W, j - r)$.

L	T	G	A	C	G	T	$	C	G	A	$	A	A	C	$	A	A
B	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5: ConcatSBWT of TAGCAAGCACAGCATACAGA. The bits are spread out inside a suffix group to maximize the number of singleton sets.



	T	G	ACGT			CG		A		A	AC			A	A
AC	0	0	1	0	0	1	0	1	0	1	1	0	0	1	1
GT	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0

	ACGT	CG	A	A	AC	A	A
A	1	0	1	1	1	1	1
C	1	1	0	0	1	0	0

	T	G	ACGT	CG
G	0	1	1	1
T	1	0	1	0

Figure 6: SubsetWTSBWT of TAGCAAGCACAGCATACAGA. The labels are concentrated to single sets inside a suffix group to create many empty sets. The index consists of just the bit vectors in the figure.

4.3 Concatenated Representation

This data structure uses a concatenation of the contents of the subsets, and an encoding of the sequence of sizes of the subsets. In more detail, let $S(X_i)$ be the concatenation of the characters in subset X_i. If X_i is the empty set, we define $S(X_i) = \$$. We build the string $L = S(X_1)S(X_2)\ldots S(X_n)$ and index it for rank queries. The empty sets are represented as dollars to be able to encode the sizes of the subsets with a bit vector B that is the concatenation $0 \cdot 1^{|S(X_1)|-1} \cdot 0 \cdot 1^{|S(X_2)|-1} \ldots 0 \cdot 1^{|S(X_n)|-1}$. See Figure 5. If the sequence of subsets contains a large number of singleton or empty sets, the bit vector B is sparse, and is efficiently compressed using the Elias-Fano encoding. The bit vector B is indexed for queries for select-0. A subset rank query for a character c up to index i is then answered by computing $rank_c(L, select_0(B, i+1) - 1)$.

4.4 Subset Wavelet Tree

We build a tree with $\log \sigma$ levels (assume for simplicity that σ is a power of 2). Each node of the tree corresponds to a part of the alphabet, defined as follows. We denote with A_v the alphabet of node v. The root node corresponds to the full alphabet. The alphabets of the rest of the nodes are defined recursively such that the left child of a node v corresponds to the first half of A_v, and the right child corresponds to the second half of A_v. Let Q_v be the subsequence of subsets that contain at least one character from A_v. As a special case, the subsequence Q_v also includes the empty sets if v is the root.

Each node v contains two bit vectors L_v and R_v of length $|Q_v|$. We have $L_v[i] = 1$ iff subset $Q_v[i]$ contains a character from the first half of A_v, and correspondingly $R_v[i] = 1$ if $Q_v[i]$ contains a character from the second half of A_v. See Figure 6. The bit vectors L_v and R_v may be entropy-compressed efficiently by considering them together as a string from alphabet $\{0, 1, 2, 3\}$, such that the i-th character is defined as $(2 \cdot L_v[i] + R_v[i])$. This will take advantage of the fact that in the case of the SBWT, it is rare that $L_v[i] = R_v[i]$ because most of the sets in the SBWT tend to be singletons. Rank queries on L_v can then be implemented by summing the ranks of characters 0 and 2, and rank queries on R_v can be implemented by summing the ranks of characters 1 and 3.

To answer our query for a character c and position i, we traverse from the root to the leaf of the tree that where A_v is the singleton subset $\{c\}$. While traversing, we compute for each visited node v the length of the prefix in the current subset sequence Q_v that contains all the subsets of $X_1, \ldots X_i$ that have at least one character from A_v. This is done by using rank queries on the bit vectors L_v and R_v, analogously to the regular wavelet tree query. The pseudocode is given in Algorithm 2.

Query time for the subset wavelet tree is clearly $O(\log \sigma)$, and without compression the data structure consumes $2n \log \sigma + o(n \log \sigma)$ bits of space.

Algorithm 2 Subset wavelet tree query.
Input: Character c from an alphabet $\Sigma = \{1, \ldots, \sigma\}$ and an index i.
Output: The number of subsets X_j such that $j \leq i$ and $c \in X_j$.

```
function SubsetRank($c, i$):
    $v \leftarrow$ root
    $[\ell, r] \leftarrow [1, \sigma]$
    while $\ell \neq r$ do
        if $c < (\ell + r)/2$ then
            $r \leftarrow \lfloor(\ell + r)/2\rfloor$
            $i \leftarrow rank_1(L_v, i)$
            $v \leftarrow$ left child of $v$
        else
            $\ell \leftarrow \lceil(\ell + r)/2\rceil$
            $i \leftarrow rank_1(R_v, i)$
            $v \leftarrow$ right child of $v$
    return $i$.
```

5 Analysis

In this section, we prove bounds on the sizes of entropy-compressed versions of the plain matrix representation of an SBWT. The size bounds are expressed in terms of the number of columns of the matrix, which is equal to $|S_k(T)| + k$, where T is the original string.

Theorem 1. *A plain matrix SBWT representation can be encoded in $n(\log \sigma + 1/\ln 2) + o(n\sigma)$ bits of space, with support for k-mer membership queries in $O(k)$ time, where n is the number of columns in the matrix, and $\sigma > 1$ is the size of the alphabet.*

Proof. The bit matrix has σ rows and n columns, and always has $n - 1$ ones. In other words, the fraction of one-bits in the matrix is $(n - 1)/(n\sigma)$, which is less than $1/\sigma$. Plugging $Pr(1) = 1/\sigma$ and $Pr(0) = 1 - 1/\sigma$ to the entropy formula (Eq. (1)), which is a monotonically increasing function in the interval $[0, 1/\sigma]$, gives

$$H \leq (\log \sigma - (\sigma - 1) \log((\sigma - 1)/\sigma))/\sigma$$

bits of entropy per matrix element, or

$$n\sigma H \leq n(\log \sigma - (\sigma - 1) \log((\sigma - 1)/\sigma))$$

bits of entropy for the whole matrix. The term $-(\sigma - 1) \log((\sigma - 1)/\sigma)$ is upper bounded by $1/\ln 2$, when $\sigma > 1$, where ln is the natural logarithm. This can be shown by plugging in $x = 1/(\sigma - 1)$ to the well-known inequality $\ln(x + 1) \leq x$, which gives $\ln(1/(\sigma - 1) + 1) \leq 1/(\sigma - 1)$, which is equivalent to $-(\sigma - 1) \log((\sigma - 1)/\sigma) \leq 1/\ln 2$. So we have:

$$n\sigma H \leq n(\log \sigma + 1/\ln 2)$$

This shows that entropy compressing the matrix to the zeroth order entropy of the bits results in space $n(\log \sigma + 1/\ln 2)$. In practice this can be done using the RRR bit vector encoding, which also supports constant-time rank queries on the bit vector with an overhead of $o(n\sigma)$. Combining Lemma 1 with the RRR encoding gives the result claimed in the theorem. □

In the case of the DNA alphabet ($\sigma = 4$), the space per k-mer is $(-0.25 \log(0.25) - 0.75 \log(0.75)) \cdot 4 = 3.245$ bits ignoring the lower order term. We note that this exactly matches Chikhi's navigational lower bound of $8 - 3 \log 3 = 3.245$ bits per k-mer [6]. This is remarkable, as our data structure is a *membership structure*, whereas the lower bound of Chikhi et al. is for a weaker navigational structure. On the other hand Chikhi's bound allows an arbitrary k-mer set, whereas ours is restricted to the spectrum of a single string.

A different bit vector representation leads to the following space-time tradeoff:

Theorem 2. *The plain matrix SBWT representation can be encoded in $n\lceil\log \sigma\rceil + 2n$ bits of space, with support for k-mer membership queries in $O(k \log \sigma)$ time, where n is the number of columns in the matrix, and $\sigma > 1$ is the size of the alphabet.*

Proof. We concatenate the rows of the matrix and represent the resulting bit string, which is of length $n\sigma$ and contains at most n 1-bits, using the Elias-Fano scheme. The space required is $n\lceil\log(n\sigma/n)\rceil + 2n = n\lceil\log\sigma\rceil + 2n$ bits. The time for a single rank query is $O(\log(n\sigma/n)) = O(\log\sigma)$, leading to the $O(k\log\sigma)$ time for a k-mer membership query as claimed. □

5.1 Compression Boosting

In this section we use the same notation as before, that is, the SBWT subset sequence is denoted with $X_1, \ldots, X_n$, and the corresponding sequence of k-mers in colexicographic order is denoted with $x_1, \ldots x_n$. We introduce the notion of suffix groups:

Definition 5. *(Suffix groups). Two sets X_i and X_j of SBWT are in the same suffix group iff $x_i[2..k] = x_j[2..k]$.*

All sets in a suffix group are adjacent in the SBWT, and the suffix groups partition the SBWT into contiguous segments. The definition of the SBWT is such that if X_i is in the same suffix group as X_{i-1}, then X_i is the empty set. We now argue that it is possible to move labels in X_i into different sets inside the same suffix group without changing the set of k-mers represented by the SBWT.

This works because moving the labels inside a suffix group changes the source node of the edge, but not the destination. That is, if x_i and x_j are in the same suffix group, then the destinations $x_i[2..k]c$ and $x_j[2..k]c$ are the same for any character c. Moving labels like this does not change the number of incoming edges to a node, so we preserve the critical property that every node apart from the node of the empty string has exactly one incoming edge.

This then raises the question: How should we distribute the labels inside of a suffix group to minimize the entropy of the subset sequence? One obvious strategy would be to concentrate the labels inside a suffix group into a single set in the suffix group, as is done in the definition of the SBWT. This gives the optimal solution sometimes, but not always. We formalize the problem as follows:

The SBWT Entropy Optimization Problem.
Input: A sequence of subsets $X_1, \ldots X_n$ of Σ, and a segmentation of the sequence into t segments. The segmentation is represented by a list of indices $p_1 < p_2 < \ldots < p_{t+1} = n+1$, such that the i-th segment consists of sets $X_{p_i}, \ldots, X_{p_{i+1}-1}$. The sets inside a segment are disjoint.
Output: A sequence of subsets $Y_1, \ldots Y_n$ of Σ with the smallest possible zeroth-order entropy, such that for every segment $Y_i, \ldots, Y_j$, we have $\bigcup_{r=i}^{j} Y_r = \bigcup_{r=i}^{j} X_r$ and the sets inside a segment are disjoint.

We now describe a heuristic optimization algorithm that converges very quickly in practice. Let $C_1, \ldots, C_m$ be the sequence of suffix groups. Each suffix group is described as a pair $C_i = (L_i, w_i)$, where L_i is the union of the sets in the suffix group, and w_i is the number of sets in the group (the "width" of the group). The algorithm starts by counting the number of times each type of a group appears in the sequence. Then, we start to tweak the groups to improve the entropy. We call the distribution of labels L_i into the w_i sets inside a group as the *configuration* of the group. We iterate the group types, and for each group type, we pick the configuration that minimizes the current total entropy. We repeat this until we are unable to improve the entropy by changing the configuration of any single group type.

6 Experiments

In our experiments, we measure query time and the size of the SBWT using the four different subset rank implementations described in Section 4. We used the value $k = 31$ in all experiments. For each implementation, we include a variant that has entropy compression, and a variant that does not. The

	Sequences	Base pairs	Distinct 31-mers	Sets in multi-SBWT
E. coli genomes	745,409	18,917,805,788	257,785,486	258,506,521
SARS-CoV-2	14,641,164	1,464,052,896	69,553,831	82,700,941
Metagenome reads	34,673,774	8,701,078,392	2,771,108,020	2,796,378,115

Table 1: Key statistics on the three experimental datasets used. Note that the number of sequences in the case of E. coli is not the number of genomes, which is 3682, but the number of contigs in assemblies of those genomes.

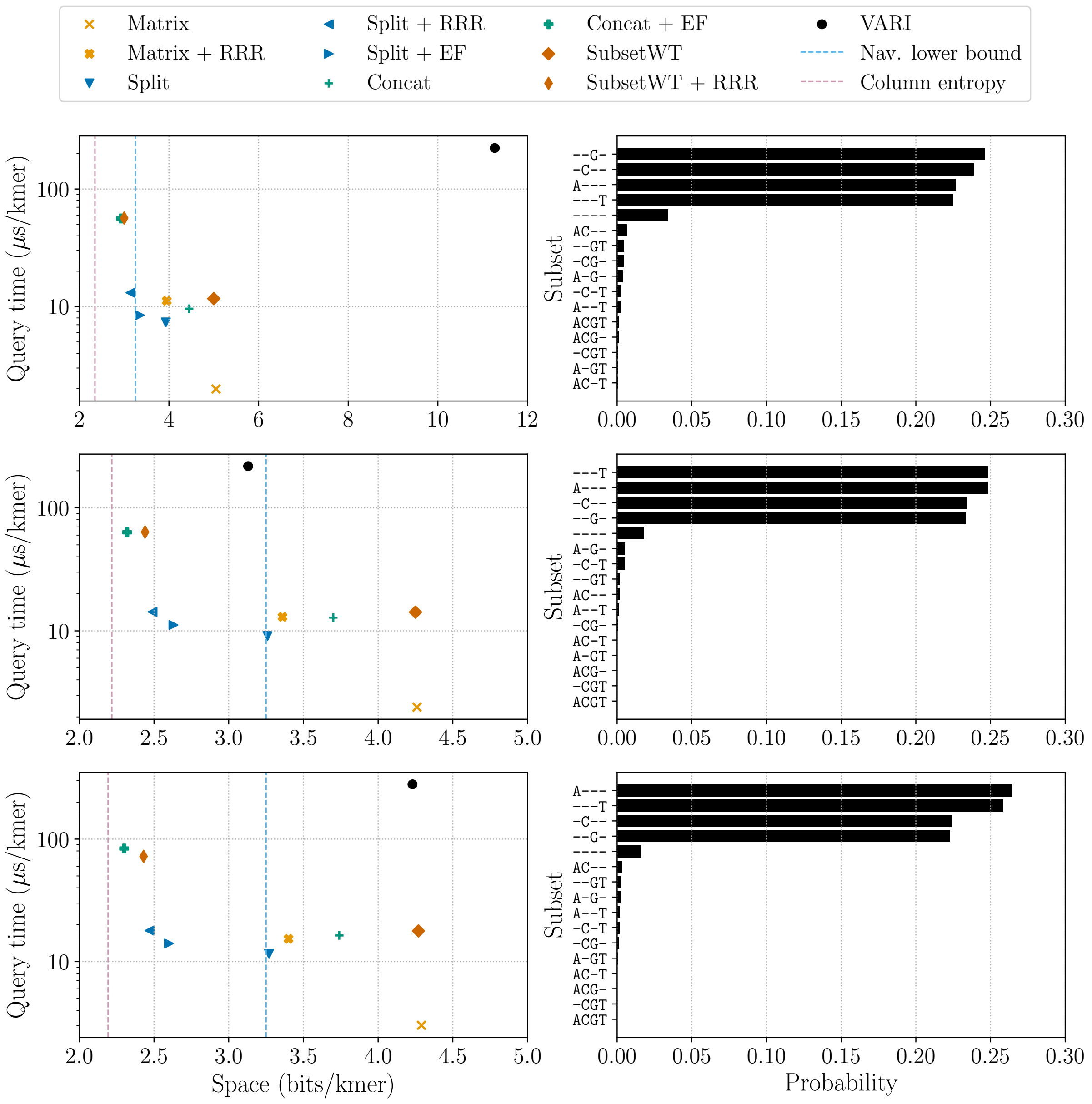


Figure 7: Experimental results. The top two plots are for the SARS-CoV-2 dataset, the middle two for the E. coli dataset, and the bottom two for the metagenomic read dataset. In the left column are the time and space of queries. The plotted times are for the present k-mers. In the right are the subset distributions in the SBWTs of the three datasets.

entropy compression is done either with RRR bit vectors [29], or the Elias-Fano encoding [31, 9, 10]. We make use of the Succinct Data Structures Library (SDSL) [12] for plain and RRR bitvectors, and use the Elias-Fano bitvector implementation from [18]. We also include the VARI data structure [23], which is derived from the original BOSS implementation of Bowe et al. [4], to provide us with an external reference data point for comparison. Specifically, we use the version of VARI found at commit `79693b7` of the branch named VARI-merge in the Github repository `https://github.com/cosmo-team/cosmo/tree/VARI-merge`. We considered including the early BOSS-like index structure of Rødland [30], which is written in Java. Rødland's data structure is somewhat similar to our plain matrix variant, but with a different memory layout and an additional bit vector. However, the implementation was unable to process our datasets, probably due to it's internal use of 32-bit integers limiting the maximum input size.

Experiments were run on Ubuntu 18.04.5 LTS kernel version 4.15.0-147-generic. The compiler was g++ 9.3.0. The CPU was an Intel Core i7-9700K CPU clocked at 3.60 GHz with with L1d, LIi, L2 and L3 caches of size 256KiB, 256KiB, 2MiB and 12MiB, respectively. The system had 64GiB of DDR4 2400 MHz memory. Reported times are the average of 10 repeated runs. The C++ code of the implementations used are available at `https://github.com/algbio/SBWT`.

6.1 Datasets

We experiment on three different data sets that represent typical targets for k-mer indexing in bioinformatics applications:

1. A pangenome of 3682 E. coli genomes, downloaded by selecting a subset of 3682 assemblies listed in `ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt` with the organism name "Escherichia coli", downloaded during the year 2020. To reproduce the dataset, the list of accession numbers of the selected genomes are available at our Github repository. The dataset contains 745,409 contigs and 257,785,486 distinct 31-mers.

2. A set of 17 million Illumina HiSeq 2500 sequence reads of length 502 sampled from the human gut (SRA identifier ERR5035349) in a study on irritable bowel syndrome and bile acid malabsorption [17]. The dataset contains 2,771,108,020 distinct 31-mers.

3. A set of 14,641,164 genomes of the SARS-CoV-2 virus downloaded from NCBI datasets. The dataset contains 69,553,831 distinct 31-mers.

We preprocess the sequences by deleting all characters that are not in the DNA alphabet $\{A,C,G,T\}$. We then construct the multi-string variant of the SBWT defined in Section 3.1 for each of the datasets. For all new index variants, we extract the data structure from the Wheeler BOSS index constructed using the tool Themisto [20]. This is straightforward as the Wheeler BOSS index gives access to the outgoing edge labels from each k-mer of the data in colexicographic order. The VARI index was constructed using the construction algorithm included with VARI. We did not measure index construction time and space since optimizing index construction is out of scope of this work.

Table 1 shows the number of sequences, total number of base pairs, number of distinct 31-mers and the number of sets in the multi-SBWT of each of the datasets. It is notable that in the SARS-CoV-2 data, the number of sets in the multi-SBWT is as much as 19% larger that the number of 31-mers. This shows that in practical datasets, the size of the prefix set $P'_k(T_1, \ldots, T_m)$ in Definition 3.1 can not be neglected. For the other two datasets, the gap was less than 1%.

6.2 Time and Space

For each variant, we measure the size of the index, and the query time. We run two sets of queries on each dataset: a randomly sampled set of 10% of the k-mers that are known to be in the index (so-called *present k-mers*), and a set of uniformly random k-mers from the space of all possible k-mers from the DNA alphabet (*absent k-mers*). This gives rough bounds on the practical query performance. The k-mers that are present in the index are the slower of the two sets to query because the search function has to run for the full k iterations; whereas random k-mers are faster to query because the search function can exit early. We report the average time per query for each query type. The reported times include only the time to query the k-mers, so that the time incurred by disk I/O when reading the index and the queries into memory is not included. The reported space is the size of the data structures serialized to disk, which in the case of all structures here corresponds to the size in memory.

	Bits per k-mer	Present k-mer query	Absent k-mer query
Matrix	5.05	1.99	0.66
Matrix + RRR	3.95	11.19	4.30
Split	3.93	7.33	2.47
Split + RRR	3.14	13.12	4.93
Split + EF	3.35	8.44	2.87
Concat	4.45	9.59	3.22
Concat + EF	2.93	56.14	23.40
SubsetWT	5.00	11.68	4.14
SubsetWT + RRR	3.00	56.64	24.43
VARI	11.27	223.17	100.89

Table 2: Results for the SARS-CoV-2 pangenome.

	Bits per k-mer	Present k-mer query	Absent k-mer query
Matrix	4.26	2.40	0.87
Matrix + RRR	3.36	12.99	5.44
Split	3.26	9.08	3.45
Split + RRR	2.49	14.26	5.94
Split + EF	2.63	11.16	4.27
Concat	3.70	12.82	4.78
Concat + EF	2.32	63.28	28.22
SubsetWT	4.25	14.20	5.66
SubsetWT + RRR	2.44	63.53	29.06
VARI	3.13	218.05	103.72

Table 3: Results for the E. coli pangenome.

	Bits per k-mer	Present k-mer query	Absent k-mer query
Matrix	4.29	3.02	1.34
Matrix + RRR	3.40	15.34	7.45
Split	3.27	11.55	5.34
Split + RRR	2.47	17.91	8.81
Split + EF	2.60	14.05	6.77
Concat	3.74	16.34	7.67
Concat + EF	2.30	83.80	41.96
SubsetWT	4.27	17.80	8.45
SubsetWT + RRR	2.43	72.26	37.59
VARI	4.23	280.05	151.91

Table 4: Results for the metagenomic read set.

The results are listed in Tables 2, 3, 4 and plotted in Figure 7. The index structures for the metagenome were between 15% to 29% larger than corresponding indexes for the other two datasets when the size is measured as bits per k-mer. This is due to the large number of prefixes $P'_k(T_1, \ldots, T_m)$ required for the metagenome in Definition 4. The matrix variant with RRR entropy compression achieved 3.36 bits per k-mer on the E.coli pangenome, with is just 3.5% larger than the theoretical zeroth-order entropy and navigational lower bound of 3.245. The consistently smallest variant was the concatenated representation with Elias-Fano encoding, occupying between 2.32 to 2.93 bits per k-mer.

The plotted times are for the k-mers that are found in the index. The type of dataset used for the index did not affect the query time much, except for the k-mer queries being slightly slower the more k-mers there were in the index. The fastest variant was the plain matrix variant, at 0.66 to 3.02 microseconds per query, and the slowest of our variants was the concatenated variant with Elias-Fano encoding, at 23.40 to 83.80 microseconds per query. The VARI implementation of Muggli et al [23] was slower than all our variants — two orders of magnitude slower than our plain matrix variant. The space was also up to multiple times larger than ours, which is explained by VARI having duplicate dummy prefixes in what corresponds to set $P'_k(T_1, \ldots, T_m)$ of Definition 4.

6.3 Entropy optimization

The distribution character subsets in the SBWTs are also plotted in Figure 7. We ran the column entropy optimization algorithm described in Section 5.1 on the E. coli dataset. The algorithms converged quickly, having to change the configuration of each suffix group at most once. The final column entropy was 2.20974, whereas the entropy of the SBWT was 2.22634, so the entropy was reduced by only approximately 0.25%. While this is a kind of a local optimum, it might not be the global optimum. We leave as an open question whether or not this algorithm is optimal.

7 Conclusion

While our prototypes already handsomely outperform all previous succinct de Bruijn graph implementations, we believe they can be improved in several ways, perhaps most promisingly by the replacement of wavelet trees with rank data structures specialized for small alphabet sequences. We also expect that the simplicity of the new rank-based query algorithms will make them significantly more accessible than the original BOSS, which is notoriously tricky to implement. The plain matrix, split and subset wavelet tree variants are particularly easy to implement, requiring only regular rank queries as the only non-trivial subroutine. This in contrast with the original BOSS, which also requires select and predecessor/successor queries.

As mentioned in the previous section, we have not yet optimized index construction, which currently uses the Themisto tool as an intermediate step to generate de Bruijn graph nodes in colexicographical order. While there is certainly room for improvement, we remark that construction times are already reasonable, taking around six hours on the 19GB E.coli dataset.

Finally, we note that it would be possible to fast add de Bruijn graph edge traversal capability to the SBWT, by marking the starts of suffix groups in a bit vector. An edge labeled with a character c can be traversed iff it occurs as an outgoing character from the current suffix group. In this case, we can run one iteration of the k-mer loop in the search routine at Algorithm 1 to follow the edge. We leave implementing and evaluating this approach to future work.

8 Acknowledgements

We thank Einar Andreas Rødland for the effort in helping us compile and run his succinct k-mer index implementation, and we thank Massimiliano Rossi helpful remarks.

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*, 19:4067–4078, 2021.

[2] B. Alipanahi, A. Kuhnle, S. J. Puglisi, L. Salmela, and C. Boucher. Succinct dynamic de bruijn graphs. *Bioinformatics*, 37(14):1946–1952, 2021.

[3] F. Almodaresi, J. Khan, S. Madaminov, M. Ferdman, R. Johnson, P. Pandey, and R. Patro. An incrementally updatable and scalable system for large-scale sequence search using the bentley-saxe transformation. *Bioinformatics*, 2022.

[4] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012.

[5] R. Chikhi. A tale of optimizing the space taken by de bruijn graphs. In *Proc. 17th Conference on Computability in Europe (CiE)*, volume 12813 of *LNCS*, pages 120–134. Springer, 2021.

[6] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the representation of de bruijn graphs. In *Proc. 18th Annual International Conference Research in Computational Molecular Biology (RECOMB)*, LNCS 8394, pages 35–55. Springer, 2014.

[7] P. E. Compeau, P. A. Pevzner, and G. Tesler. Why are de bruijn graphs useful for genome assembly? *Nature biotechnology*, 29(11):987, 2011.

[8] V. G. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie. Practical dynamic de bruijn graphs. *Bioinformatics*, 34(24):4189–4195, 2018.

[9] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[10] R. Fano. On the number of bits required to implement an associative memory. Technical report, MIT, 1971.

[11] T. Gagie, G. Manzini, and J. Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical computer science*, 698:67–78, 2017.

[12] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*, pages 326–337. Springer, 2014.

[13] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014.

[14] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, 2003.

[15] G. Holley and P. Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome biology*, 21(1):1–20, 2020.

[16] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics*, 44(2):226–232, 2012.

[17] I. B. Jeffery, A. Das, E. O'Herlihy, S. Coughlan, K. Cisek, M. Moore, F. Bradley, T. Carty, M. Pradhan, C. Dwibedi, et al. Differences in fecal microbiomes and metabolomes of people with vs without irritable bowel syndrome and bile acid malabsorption. *Gastroenterology*, 158(4):1016–1028, 2020.

[18] D. Ma, S. J. Puglisi, R. Raman, and B. Zhukova. On Elias-Fano for rank queries in FM-indexes. In *31st Data Compression Conference (DCC)*, pages 223–232. IEEE, 2021.

[19] N. Maillet, C. Lemaitre, R. Chikhi, D. Lavenier, and P. Peterlongo. Compareads: comparing huge metagenomic experiments. In *BMC bioinformatics*, volume 13, pages 1–10. Springer, 2012.

[20] T. Mäklin, T. Kallonen, J. Alanko, Ø. Samuelsen, K. Hegstad, V. Mäkinen, J. Corander, E. Heinz, and A. Honkela. Bacterial genomic epidemiology with mixed samples. *Microbial genomics*, 7(11), 2021.

[21] 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):1–12, 2021.

[22] C. Marchet, M. Kerbiriou, and A. Limasset. BLight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, 04 2021.

[23] M. D. Muggli, A. Bowe, N. R. Noyes, P. S. Morley, K. E. Belk, R. Raymond, T. Gagie, S. J. Puglisi, and C. Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, Oct. 2017.

[24] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180, pages 37–42. Springer, 1996.

[25] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In R. Klasing, editor, *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2012.

[26] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007.

[27] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17(1):1–14, 2016.

[28] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[29] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, Nov. 2007.

[30] E. A. Rødland. Compact representation of k-mer de bruijn graphs for genome read assembly. *BMC bioinformatics*, 14(1):1–19, 2013.

[31] S. Vigna. Quasi-succinct indices. In *Proc. Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92. ACM, 2013.