Subset Wavelet Trees

Jarno N. Alanko
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Elena Biagi
Department of Computer Science, University of Helsinki, Finland

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Jaakko Vuohtoniemi
Department of Computer Science, University of Helsinki, Finland

Abstract

Given an alphabet Σ of $\sigma = |\Sigma|$ symbols, a degenerate (or indeterminate) string X is a sequence $X = X[0], X[1] \ldots, X[n-1]$ of n subsets of Σ. Since their introduction in the mid 70s, degenerate strings have been widely studied, with applications driven by their being a natural model for sequences in which there is a degree of uncertainty about the precise symbol at a given position, such as those arising in genomics and proteomics. In this paper we introduce a new data structural tool for degenerate strings, called the subset wavelet tree (SubsetWT). A SubsetWT supports two basic operations on degenerate strings: subset-rank(i, c), which returns the number of subsets up to the i-th subset in the degenerate string that contain the symbol c; and subset-select(i, c), which returns the index in the degenerate string of the i-th subset that contains symbol c. These queries are analogs of rank and select queries that have been widely studied for ordinary strings. Via experiments in a real genomics application in which degenerate strings are fundamental, we show that subset wavelet trees are practical data structures, and in particular offer an attractive space-time tradeoff. Along the way we investigate data structures for supporting (normal) rank queries on base-4 and base-3 sequences, which may be of independent interest. Our C++ implementations of the data structures are available at `https://github.com/jnalanko/SubsetWT`.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Design and analysis of algorithms

Keywords and phrases degenerate strings, compressed data structures, succinct data structures, string processing, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.4

Supplementary Material *Software (Source Code)*: `https://github.com/jnalanko/SubsetWT`
Software (Source Code): `https://github.com/jnalanko/SubsetWT-Experiments`

Funding This work was supported in part by the Academy of Finland via grants 339070 and 351150.

Acknowledgements A brief description of the subset wavelet tree first appeared in a technical report by the authors [3].

1 Introduction

Given an alphabet Σ of σ symbols, a degenerate (or indeterminate) string is a sequence $X = X[0], X[1] \ldots, X[n-1]$ of subsets of Σ. For example, here is a degenerate string of length 15 on the alphabet $\Sigma = A, C, G, T$ (note that empty subsets are allowed):

$$X = \{T\}\{G\}\{A, C, G, T\}\{\}\{\}\{C, G\}\{\}\{A\}\{\}\{A\}\{A, C\}\{\}\{\}\{A\}\{A\}.$$

Since their introduction in a classic paper by Fischer and Paterson [10], degenerate strings have been widely studied in the field of string processing and its application domains. Abrahamson [1] studied online pattern matching for degenerate strings, describing theoretically optimal algorithms. Since then, several authors (see, e.g., [16, 30]) have described practical pattern matching algorithms that are fast in practice. Authors have also considered covering problems [7], data structures for extension queries [17], constrained LCS [18], and the computation of inverted repeats [2] on degenerate strings.

Interest in degenerate strings has been driven by them being a natural model of the uncertainty or flexibility often present in real world sequence data. For example, the IUPAC encoding for biological sequences [19] designates specific symbols, referred to as degenerate, to represent a sequence position corresponding to a set of possible alternative nucleotides. In music sequences, single notes may match chords, or notes separated by an octave may match – properties naturally captured by degenerate strings [5].

In this paper we study two new and seemingly fundamental operations on indeterminate strings - subset rank and select. In particular, for a given degenerate string X, we define (for $i \leq n$ and $c \in \Sigma$):

$$\begin{aligned}\textsf{subset-rank}_X(i, c) &= \text{number of subsets among the first } i \text{ subsets of } X \text{ that contain } c;\\ \textsf{subset-select}_X(i, c) &= \text{index in } X \text{ of the } i\text{th subset that contains } c.\end{aligned}$$

For example, if $X = \{T\}\{G\}\{A, C, G, T\}\{\}\{\}\{C, G\}\{\}\{A\}\{\}\{A\}\{A, C\}\{\}\{\}\{A\}\{A\}$ as before, then we would have $\textsf{subset-rank}_X(8, A) = 2$.

Rank and select queries on ordinary (i.e. non-degenerate) strings are now considered fundamental to the field of succinct and compressed data structures [26, 23]. To our knowledge, however, the literature on degenerate and indeterminate strings has not explicitly considered these queries before.

Our own interest in supporting these types of queries on degenerate strings comes from problems in pangenomics, and in particular the spectral Burrows-Wheeler transform (SBWT), a recently described approach for representing the de Bruijn graph of a set of strings [3]. The de Bruijn graph is a central data structure in computational biology, used for a variety of tasks, including genome assembly [6] and pangenomic read alignment [15, 21]. There is exactly one node in the de Bruijn graph for every distinct k-length substring, or k-mer occurring in the set of input strings, and nodes are labelled with these substrings. A k-mer query on the de Bruijn graph asks if there is a node in the graph labelled with a specified query k-mer. In [3] it is shown that these queries can be reduced to a sequence of $2k$ subset-rank queries on a particular degenerate string L produced by the SBWT that encodes the graph. For brevity, we avoid defining the SBWT here, but we note that L has a special property that its length is also equal to the sum of the sizes of the sets, i.e. $|L| = \sum_i |L[i]|$. The allowance of empty sets means this is possible without the resulting sequence becoming an ordinary string. We call such a degenerate string *balanced*. We return to this special case later, but note here that the data structure we describe applies to all degenerate strings, balanced or not.

Contribution. We describe the subset wavelet tree, a new data structure for subset-rank and subset-select queries on degenerate strings. Our experiments on a real-world application show that subset wavelet trees offer attractive space-time tradeoffs for subset-rank queries in real-world genomics applications. A key subproblem in the navigation of a SubsetWT to answer subset-rank is computing normal rank queries on small alphabets sequences (in particular, base-3 and base-4 sequences). With this in mind, we describe and benchmark several efficient methods for that subproblem, which may be of independent interest.

Roadmap. In the next section we cover basic concepts and related work. We also provide details of our experimental setup and the data sets we use in later sections. In Section 3 we describe a simple data structure for subset-rank and subset-select that acts as a baseline against which the practical performance of our data structure can be gauged. In Section 4 we describe the subset wavelet tree and algorithms for supporting subset-rank and subset-select with it. Section 5 describes methods for computing normal rank queries on small alphabets sequences. Section 6 then reports on experiments using the SubsetWT for k-mer queries using SBWT representation discussed above.

2 Preliminaries

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

$$\mathsf{rank}_X(i,c) = \text{number of } c\text{'s among the first } i \text{ bits of } X$$
$$\mathsf{select}_X(i,c) = \text{position of the } i\text{-th } c \text{ in } X$$

Classical techniques [24] (see also [28]) 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 and select operations. Perhaps the foremost of these, known as "RRR", is due to Raman, Raman, and Rao Satti [29] and takes space $\mathcal{B}(n, n_1) + o(n)$ and answers all queries above in $O(1)$ time. Fast implementations of RRR exist [27, 12, 22].

Rank and select for larger alphabets. There are also solutions for rank and select for sequences on larger alphabets [14, 13, 8, 4]. Perhaps the most versatile and useful of these is the wavelet tree [14, 25], which we now describe.

Consider a (ordinary) string $S = S[0]S[1]\dots S[n]$ over alphabet Σ. The wavelet tree of S is a balanced binary tree, where each leaf represents a symbol of Σ. The root is associated with the complete sequence S. Its left child is associated with a subsequence obtained by concatenating the symbols $S[i]$ of S satisfying $S[i] < |\Sigma|/2$. The right child corresponds to the concatenation of every symbol $S[i]$ satisfying $S[i] \geq |\Sigma|/2$. This relation is maintained recursively up to the leaves, which are associated with the repetitions of a unique symbol. At each node we store only a binary string of the same length of the corresponding sequence, using at each position a 0 to indicate that the corresponding symbol is mapped to the left child, and a 1 to indicate the symbol is mapped to the right child.

If the bit strings of the nodes support constant-time rank and select queries, then the wavelet tree supports fast rank and select on T. Before describing how those queries are carried out, it is instructive to examine a simpler query, namely accessing a given symbol in the input string using only its wavelet tree.

access: In order to obtain the value of $S[i]$ the algorithm begins at the root, and depending on the value of the root bit string B at position i, it moves down to the left or to the right child. If the bit string value is 0 it goes to the left, and replaces $i \leftarrow \mathsf{rank}_B(i, 0)$. If the bit string value is 1 it goes to the right child and replaces $i \leftarrow \mathsf{rank}_B(i, 1)$. When a leaf is reached, the symbol associated with that leaf is the value of a_i.

rank: To obtain the value of $\mathsf{rank}_S(i, c)$ the algorithm is similar. It begins at the root and goes down updating i as in the previous query, but the path is chosen according to the bits of c instead of looking at $B[i]$. When a leaf is reached, the i value is the answer.

Table 1 Statistics on the raw genomic datasets used in experiments. A k-mer is considered equal to its reverse complement in the k-mer counts. We derived a single degenerate string from each of these data sets using the Spectral Burrows-Wheeler transform.

	Number of sequences	**Total length**	**Unique 31-mers**
E. coli	745,409	18,957,578,183	170,648,610
SARS-CoV-2	1,234,695	36,808,137,972	2,407,721
Metagenome	17,336,887	8,703,117,274	2,761,523,935

select: The value of $\mathsf{select}_S(j, c)$ is computed as follows: The algorithm begins in the leaf corresponding to the character c, and then moves upwards until reaching the root. When it moves from a node to its parent, j is updated as $j \leftarrow \mathsf{select}_B(j, 0)$ if the node is a left child, and $j \leftarrow \mathsf{select}_B(j, 1)$ otherwise. When the root is reached, the final j value is the answer.

Experimental Setup. All our experiments were conducted on a machine with four 2.10 GHz Intel Xeon E7-4830 v3 CPUs with 12 cores each for a total of 48 cores, 30 MiB L3 cache, 1.5 TiB of main memory, and a 12 TiB serial ATA hard disk. The OS was Linux (Ubuntu 18.04.5 LTS) running kernel 5.4.0-58-generic. The compiler was `g++` version 10.3.0 and the relevant compiler flags were `-O3 -march=native` and `-DNDEBUG`. All runtimes were recorded by instrumenting the code with calls to the high-resolution clock of `std::chrono` in C++. The sizes of the index structures in memory were calculated by adding together the sizes of each individual component. The code to reproduce the experiments is available at `https://github.com/jnalanko/SubsetWT-Experiments`.

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. The data was downloaded during the year 2020 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" with date before March 22, 2016. The resulting collection is available at `zenodo.org/record/6577997` .
2. A set of 17,336,887 Illumina HiSeq 2500 reads of length 502 sampled from the human gut (SRA identifier ERR5035349) in a study on irritable bowel syndrome and bile acid malabsorption [20].
3. A set of 1,234,695 genomes of the SARS-CoV-2 virus downloaded from NCBI datasets.

Table 1 shows a number of key statistics. The constructed index structures include both forward and reverse DNA strands.

3 Simple Subset Rank and Select

We now describe a straightforward way to support subset-rank and subset-select on a degenerate string X of length n over alphabet Σ in $O(1)$ time and uses $O(n\sigma)$ bits of space. For each symbol $c \in \Sigma$ we store a bit string R_c of length n such that $R_c[i] = 1$ if and only if set $X[i]$ contains symbol c. Each bit string is preprocessed for rank and select queries. To answer $\mathsf{subset\text{-}rank}_X(i, c)$ we simply return $\mathsf{rank}_{R_c}(i, 1)$. Select is answered in a similar way. The approach is fast in practice, and will act as a baseline in our experiments.

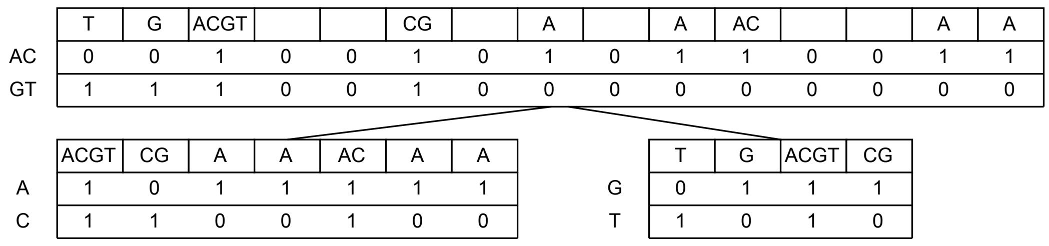

Figure 1 SubsetWT of $X = \{T\}\{G\}\{A,C,G,T\}\{\}\{\}\{C,G\}\{\}\{A\}\{\}\{A\}\{A,C\}\{\}\{\}\{A\}\{A\}$. Conceptually there are two bitvectors at each node of the tree, L_v and R_v which are shown on top of each other in the figure. As described in the text, L_v and R_v can be combined into a base-4 sequence at the root and into a base-3 sequence at other nodes. At the root of this example, the sequence would be 113003020220022, and at the left child of the root would be 3122322 (or 2011211 on a minimal base-3 alphabet).

4 Subset Wavelet Tree

We build a tree with $\log \sigma$ levels[1]. 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 when 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 $R_v[i] = 1$ iff $Q_v[i]$ contains a character from the second half of A_v. Figure 1 illustrates our running example. The bit vectors L_v and R_v can be combined to form a string on the alphabet $\{0, 1, 2, 3\}$, such that the i-th character is defined as $(2 \cdot L_v[i] + R_v[i])$.

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 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, analogous to a regular wavelet tree query. Pseudocode is given in Algorithm 1.

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

Query time for the subset wavelet tree is clearly $O(\log \sigma)$, as constant time is spent at each of the $\log \sigma$ levels. For a general sequence of sets, the data structure requires $2n(\sigma - 1) + o(n\sigma)$ bits of space. The subset wavelet tree can be thought of as a complete binary tree with σ leaves, labeled with the symbols of the alphabet. These are not in Figure 1 since leaves are not actually stored in the subset wavelet tree. If all sets are full, then each set goes both

[1] We assume for simplicity that σ is a power of 2.

Algorithm 1 Subset wavelet tree rank 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($i, c$):
    $v \leftarrow$ root
    $[\ell, r] \leftarrow [1, \sigma]$
    while $\ell \neq r$ do
        if $c < (\ell + r)/2$ then
            $r \leftarrow \lfloor(\ell + r)/2\rfloor$
            $i \leftarrow rank_{L_v}(i, 1)$
            $v \leftarrow$ left child of $v$
        else
            $\ell \leftarrow \lceil(\ell + r)/2\rceil$
            $i \leftarrow rank_{R_v}(i, 1)$
            $v \leftarrow$ right child of $v$
    return $i$.
```

Algorithm 2 Subset wavelet tree select query.

Input: Character c from an alphabet $\Sigma = \{1, \ldots, \sigma\}$ and an index i.
Output: The position of subset X_j such that the i^{th} $c \in X_j$.

```
function SubsetSelect($i, c$):
    $v \leftarrow c$ leaf
    while $v \neq$ root do
        $u \leftarrow$ parent of $v$
        if $v =$ left child of $u$ then
            $i \leftarrow select_{L_v}(i, 1)$
        else
            $i \leftarrow select_{R_v}(i, 1)$
        $v \leftarrow u$
    return $i$.
```

to the left and to the right child at each level. This means that every internal node of the subset wavelet tree stores $2n$ bits. The total size of the subset wavelet tree is then given by the number of (internal) nodes multiplied by the size of each of them, thus $(\sigma - 1)2n$.

For a balanced degenerate string, however, less space is needed. In particular, because each element in each set corresponds to at most one symbol in sequence at a given level of the tree, the total length of the sequences is bound by the total sizes of the sets, making the number of bits over all $\log \sigma$ levels of the tree $2n \log \sigma$. We thus have the following theorem.

▶ **Theorem 1.** *The subset wavelet tree of a balanced degenerate string takes $2n \log \sigma + o(n \log \sigma)$ bits of space and supports subset rank and subset select queries in $O(\log \sigma)$ time.*

5 Rank for Base-3 and Base-4 Sequences

A critical operation in answering subset rank queries with the subset WT is to answer (ordinary) rank queries on the small alphabet sequences stored at the nodes of the tree. The sequence at the root is base-4 (with alphabet $\Sigma = \{0, 1, 2, 3\}$) and the sequence at every other node is base-3 (with alphabet $\Sigma = \{0, 1, 2\}$). Actually, the required operation is more specific than a rank query: we always want to know the sum of $\mathsf{rank}(i, \sigma - 1)$ and either of $\mathsf{rank}(i, \Sigma[0])$ or $\mathsf{rank}(i, \Sigma[1])$. We call these combined queries *rank-pair* queries. In particular:

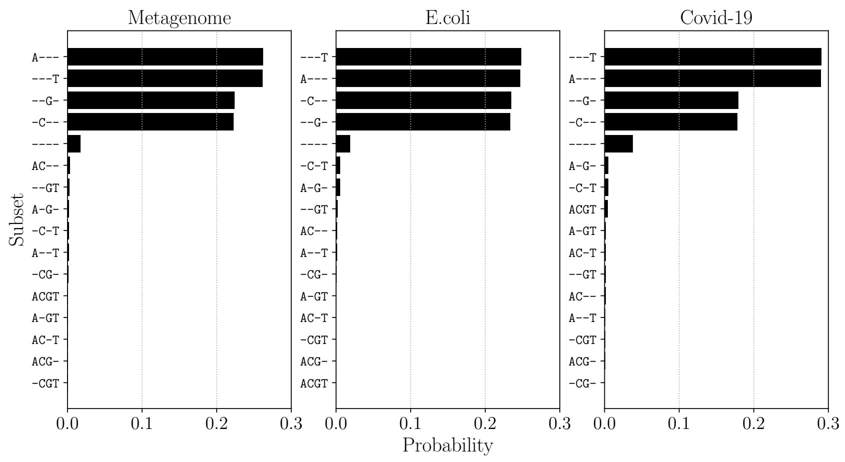

Figure 2 Distributions of subsets in the balanced degenerate string produced by the Spectral Burrows-Wheeler transform on the three genomic data sets described in Section 2. The three plots show that in all cases the set distribution is highly skewed, with the vast majority of the sets being singletons. The entropies for the distributions are: 2.21 (Metagenome), 2.24 (E.coli), and 2.31 (Covid-19).

Base-4

$$\mathsf{rankpair}(i,1) = \mathsf{rank}(i,1) + \mathsf{rank}(i,3)$$
$$\mathsf{rankpair}(i,2) = \mathsf{rank}(i,2) + \mathsf{rank}(i,3)$$

Base-3

$$\mathsf{rankpair}(i,0) = \mathsf{rank}(i,0) + \mathsf{rank}(i,2)$$
$$\mathsf{rankpair}(i,1) = \mathsf{rank}(i,1) + \mathsf{rank}(i,2)$$

In this section we examine methods for supporting rank and rank-pair queries on small alphabet sequences. All the methods we describe support both types of queries, but, importantly, some structures offer more ready support for rank-pair than do others.

We develop the structures with our spectral BWT application in mind. As stated earlier, the SBWT sequence is a balanced degenerate string. Moreover, singleton sets dominate, as the plots in Figure 2 show very clearly. In developing our rank/*rank-pair* data structures in this section, we specifically target degenerate strings with skewed distributions.

5.1 Wavelet Trees

The current de facto standard for rank queries on sequences over non-binary alphabets is the wavelet tree. We, therefore, use WTs as a baseline for the other methods we develop in this section. Using different bitvector implementations inside WTs leads to different space-time tradeoffs. We experimented with both plain bitvectors, which make the WT faster and larger, and RRR bitvectors, which, as discussed in Section 2 take $nH_0 + o(n)$ bits of space for an input bit string of n bits and are generally slower.

We used implementations from the Succinct Data Structures Library (SDSL), which are the fastest wavelet tree implementations we know of. We remark that wavelet trees as implemented in the SDSL offer no ready support for *rank-pair* queries, and so we implement *rank-pair* by issuing two separate rank queries for the appropriate symbols.

5.2 Scanning Rank

This approach is inspired by fast methods for binary rank queries [12], where the task is to count 1s up to a given position. For brevity, we describe only the structure for base-4 sequences here – the structure for base-3 sequences is essentially the same.

The data structure consists of three layers. At the lowest layer is the sequence X itself, packed into words. Assuming 64-bit words, we can pack 32 base-4 symbols into a single word, and so this layer takes $64 \cdot \lceil n/32 \rceil \approx 2n$ bits. At the highest layer, we divide X into superblocks each of size s. For each superblock we store the answer to $\mathsf{rank}(i, c)$ for all $c \in \Sigma$, where i is the start of the superblock. These answers are stored in a table of size $\sigma n/s$ words so that we can access the answers for the superblock containing a given position j in constant time at column j/s of the table. In the middle layer of the structure we divide X into blocks of size $b < s$, where b is a divisor of s. For a block beginning at position i, we precompute and store, for each symbol $c \in \Sigma$, the number of occurrences of c in $X[s\lfloor i/s \rfloor..i)$ – in other words, the count between the start of the block and the start of its enclosing superblock. If we set $s = 2^{32}$, then the block counts need 32 bits each. In our experiments we set $b = 1024$.

A critical optimization is to interleave the counts stored for each block with the part of the sequence covered by the block. In memory, the format of a block is a 2-word header containing four precomputed counts, followed by a data section of $b/64$ words into which the b symbols themselves are packed. Interleaving the header and data sections in this way, the lower and middle layers as a single array A of $(2n/b + n/32)$ words in memory. Thus, accessing the data sections of a block immediately after its header has good memory locality.

Query $\mathsf{rank}(i, c)$ is answered as follows. The header for the block containing position i starts at position $j = i/(b + 2)$ in A. We retrieve the count for c and add it to the relevant count retrieved from the superblock table. We then proceed to scan $A[j + 2..i \mod b)$, counting occurrences of c. In general this involves inspecting zero or more whole words and possibly one partial word, which together contain part of the input sequence relevant to the query. Counting occurrences of bit patterns 00, 01, 10, and 11 in whole (or partial) words can be made fast by the use of bitwise operations. *rank-pair* affords a particularly fast implementation with relevant symbol occurrences counted inside a word via a single bitwise AND (with an appropriate mask) and a single popcount operation.

5.3 Sequence Splitting

Our next structure aims to exploit the skewed distribution in real subset sequences observable in Figure 2. Because the sets in X are mostly singletons, in the base-4 sequence at the root of the SubsetWT symbols 1 and 2 will dominate[2]. With this in mind, to represent a base-4 sequence X of length n, as follows. Let $X_{a,b}$ be the subsequence of X consisting of only symbols $X[i] \in \{a, b\}$. For $X_{1,2}$ we store a bitvector L of $|X_{1,2}|$ bits where $L[i] = 1$ if $X_{1,2}[i] = 2$ and $L[i] = 0$ if $X_{1,2}[i] = 1$. We store a similar bitvector R for $X_{0,3}$: $R[i] = 1$ if $X_{0,3}[i] = 3$ and $R[i] = 0$ if $X_{0,3}[i] = 1$. Finally, we store positions i such that $X[i] \in \{0, 3\}$ in a predecessor data structure, P. If there is skewness of the subset distribution we can expect P and R to be small. Both bitvectors L and R are indexed for rank queries. In summary, the final data structure for a base-4 sequence consists of P, L and R and their rank support structures. For base-3 sequences there is no need to store the bitvector L, since P stores exclusively the indexes i such that $X[i] = 2$, as those are the only non-singleton sets.

On a base-4 sequence X query $\mathsf{rank}_X(i, c)$ is answered with a predecessor query on P for position i, which returns p, the number of elements in P smaller than i (i.e., the rank of the predecessor of i in P), followed by a binary rank query on L or R. Subtracting the result of the predecessor query p from i gives us the appropriate index for a binary rank query on L if $c \in \{1, 2\}$. In particular $\mathsf{rank}_X(i, 1) = \mathsf{rank}_L(i - p, 0)$ and $\mathsf{rank}_X(i, 2) = \mathsf{rank}_L(i - p, 1)$. For answering rank queries with $c \in \{0, 3\}$, we require a binary rank query on the bitvector

[2] In sequences at lower nodes, which are base-3, it will be symbols 0 and 1 that dominate

R at position p, in particular $\mathsf{rank}_X(i,0) = \mathsf{rank}_R(p,0)$ and $\mathsf{rank}_X(i,3) = \mathsf{rank}_R(p,1)$. Rank queries on a base-3 sequence are the same as for base-4 for singletons, $x \in \{0,1\}$, specifically $\mathsf{rank}(i,0) = \mathsf{rank}_L(i-p,0)$ and $\mathsf{rank}(i,1) = \mathsf{rank}_L(i-p,1)$. As no second binary vector is present, the result of the predecessor query gives us directly the rank of $c = 2$.

rank-pair queries with this structure can be answered faster than two separate single rank queries. Indeed, with rank-pair queries we can save a predecessor query as p is computed only once for both symbols in the query.

5.4 Extending RRR to Base-3 and Base-4 Sequences

Our final method is a generalization – to base-3 and base-4 sequences – of the famous entropy compressed bitvector due to Raman, Raman, and Rao-Satti [29], the so-called RRR data structure. RRR represents a bitstring using at most $nH_0 + o(n)$ bits and supports rank and select operations on the bitstring in $O(1)$ time per query, without needed access to the original input after construction. Practical implementations of generalizations of RRR have been proposed before [8], however our approach is different, drawing on ideas by Navarro and Providel [27] for a particular implementation of the binary RRR scheme.

Let X be a sequence of length n from an alphabet with constant size σ. We index X using a three-level structure similar to the basic binary RRR structure. That is, we segment X into blocks of size $b = O(\log n)$ and superblocks of size $B = O(\log^2 n)$, where B is a multiple of b. We precompute the counts of symbols up to the start of each superblock, and the counts of symbols inside each block. The precomputed values are represented using $O(\log n)$ bits each for superblock, and $O(\log \log n)$ bits for the regular blocks, making the total space for those values $O(n\sigma \log \log n / \log n)$, which is $o(n)$, as σ was assumed constant. A rank query $\mathsf{rank}(i,c)$ is answered in three parts: first, look up the count of c up to the superblock containing i, then, add up the counts of c in blocks preceding index i in the superblock, and lastly, add the count of occurrences of c in the prefix of length $p = i \bmod b$ in the block containing index i.

To compute the count of a symbol within a prefix of a block, we encode some extra information to be able to decode the sequence of symbols in a block, and then loop to count the number of occurrences in the prefix of length $i \bmod b$. Consider the equivalence relation that partitions the space of all the σ^b possible distinct blocks into equivalence classes such that two blocks are in the same class if and only if they contain the same multiset of symbols. We store for each block the rank r of the block in the lexicographically sorted list of blocks in its equivalence class. The class and the lexicographic rank within the class completely determine the sequence of symbols inside the block. That is, there exists a function $\mathsf{unrank}(r, d_0, d_1, \ldots d_{\sigma-1})$ that takes the lexicographic rank r and the counts $d_0, d_1, \ldots d_{\sigma-1}$ of symbols inside the block, and returns the sequence of symbols in the block. It remains to show how to implement $\mathsf{unrank}(r, d_0, d_1, \ldots d_{\sigma-1})$.

One way to implement unrank would be to precompute and store the answers to all queries $\mathsf{unrank}(r, d_0, d_1, \ldots d_{\sigma-1})$. This corresponds to the universal tables in the original RRR data structure. This, however, is space consuming for large b, so we describe a way to compute unrank without using any extra space at all. Our method can be seen as a generalization of the scheme used in the practical RRR implementation of Navarro and Providel [27], from a binary alphabet to an integer alphabet.

We denote by $\binom{n}{d_0 d_1 \ldots d_{\sigma-1}}$ the *multinomial coefficient* $\frac{n!}{d_0! d_1! \cdots d_{\sigma-1}!}$, defined so that the value is 0 if any of the $d_0, \ldots, d_{\sigma-1}$ are negative or their sum is greater than n. Let *lexrank*$(c_0, c_1, \ldots c_{b-1})$ be the lexicographic rank of a block $c_0, c_1, \ldots c_{b-1}$ in its equivalence class. Let $D_c(i)$ be the number of occurrences of symbol c in the suffix $c_i, \ldots, c_{b-1}$. Now we can write:

$$\mathsf{lexrank}(c_0, \dots c_{b-1}) = \sum_{i=0}^{b-1} \sum_{j=0}^{c_i-1} \binom{b-1-i}{D_0(i) \quad \dots \quad D_j(i)-1, \quad \dots \quad D_{\sigma-1}(i)},$$

where the -1 in the choices of the multinomial is only applied for choice $D_j(i)$. The formula represents a process that iterates the symbols of the block from left to right, adding up ways to complete the block using the remaining counts such that the completed block is lexicographically smaller than the input block. Computing the unrank function is a matter of inverting the lexrank function. We do this by adding the multinomials in the inner sum until the total would become greater than the target rank r. When this happens, we append the current symbol j to the sequence of the block and proceed to the next round of the outer sum. Algorithm 3 provides the pseudocode of the process for a base-4 sequence.

Algorithm 3 Base-4 block unrank. Prints the sequence of symbols in the block with rank r among the class of blocks with symbol counts d_0, d_1, d_2 and d_3.

```
function BASE4BLOCKUNRANK($r, d_0, d_1, d_2, d_3$):
    $b \leftarrow d_0 + d_1 + d_2 + d_3$                          ▷ Block size
    $s \leftarrow 0$                                               ▷ Blocks counted so far
    for $i = 0, \dots, b-1$ do
        for $j = 0, \dots, 3$ do                                   ▷ 0 to $\sigma - 1$
            $d_j \leftarrow d_j - 1$
            count $\leftarrow \binom{b-1-i}{d_0, d_1, d_2, d_3}$
            $d_j \leftarrow d_j + 1$
            if $s$ + count $> r$ then
                print $j$
                $d_j \leftarrow d_j - 1$
                break
            else
                $s \leftarrow s$ + count
```

5.4.1 Practical considerations

In practice, we use a block size $b = 31$ and superblock size $B = 32b = 992$. With this choice of b, the counts of symbols inside blocks fit into 5 bits each. We omit the count of the last symbol of the alphabet in each block because it can be computed by subtracting the counts of the other symbols from the block size b. This choice of b also guarantees that lexicographic ranks of blocks within their classes always fit in 64-bit integers, assuming that the alphabet size is at most 4. To compute the multinomial coefficients for unrank, we use the formula $\binom{n}{d_0 \dots d_{\sigma-1}} = \binom{n}{d_0}\binom{n-d_0}{d_1}\binom{n-d_0-d_1}{d_2} \dots \binom{n-d_0-\dots-d_{\sigma-2}}{d_{\sigma-1}}$. The expression is evaluated using only $(\sigma - 1)$ multiplications by loading the binomials from a precomputed table and omitting the last term which is always equal to 1. We terminate the block decoding process early after having decoded the prefix of the required length.

These lexicographic ranks within a class are stored compactly using $\lceil \log_2 m \rceil$ bits each, where m is the size of the class of the block. The binary representations of these ranks are concatenated in memory. Since the query algorithm will always access the list of lexicographic ranks of blocks in sequential order starting from a superblock boundary, we do not have to store the widths of all of the binary representations in the concatenation, but instead, we only store the sum of widths up to each superblock, and we can compute the width of the binary representation of the i-th block from the stored symbol counts during query time.

The binomials involved in the computation are again loaded from a precomputed table, and the integer base-2 logarithms are efficiently implemented using a machine instruction to count the number of leading zeroes in a word.

5.5 Microbenchmark

To evaluate the practical performance of the small-alphabet rank data structures developed earlier in this section, we benchmarked 10^7 rank and rank-pair queries at random positions for random characters, in the base-3 and base-4 sequences extracted from the SubsetWT of the SBWT of our metagenomic read dataset.

The smallest data structure was the RRR-based wavelet tree, which was also the slowest. The fastest was the Scanning solution, but it had the largest space. The full results are in Table 2. The WT RRR, Generalized RRR, and Split methods all achieve some level of compression, while WT plain and Scanning methods both expand on the size of the input sequence. Finally, we observe that all methods answer rank − *pair* queries in less than twice the time it takes to answer a single rank query. The most impressive *rank-pair* performance (relative to rank performance) is shown by Generalized RRR and Scanning, both of which can save significant computation when computing *rank-pair*.

Table 2 Microbenchmark results on random queries on base-4 and base-3 sequences derived from the SubsetWT of the Spectral Burrows-Wheeler transform of the metagenomic read dataset.

Sequence	Structure	space (bps)	rank time (ns)	*rank-pair* time (ns)
Base-4	WT plain	2.13	247	404
	WT RRR	1.29	1017	1517
	Generalized RRR	1.55	826	829
	Split	1.69	290	328
	Scanning	2.25	142	106
Base-3	WT plain	2.12	199	369
	WT RRR	1.15	718	1006
	Generalized RRR	1.26	681	679
	Split	1.39	224	248
	Scanning	2.25	148	107

Table 3 SBWT k-mer search queries with different subset rank implementations. The space is given in units of bits per indexed k-mer, where a k-mer is considered distinct from its reverse complement. The time is reported in microseconds per queried k-mer.

Dataset	Subset Rank Structure	Space (bpk)	Query Time (μs)
Metagenomic reads	Simple	4.66	1.49
	SubsetWT<Generalized-RRR>	3.07	44.13
	SubsetWT<WT-RRR>	2.67	71.37
	SubsetWT<Split>	3.36	6.03
E. coli genomes	Simple	4.29	1.04
	SubsetWT<Generalized-RRR>	2.84	41.21
	SubsetWT<WT-RRR>	2.48	67.01
	SubsetWT<Split>	3.17	6.84

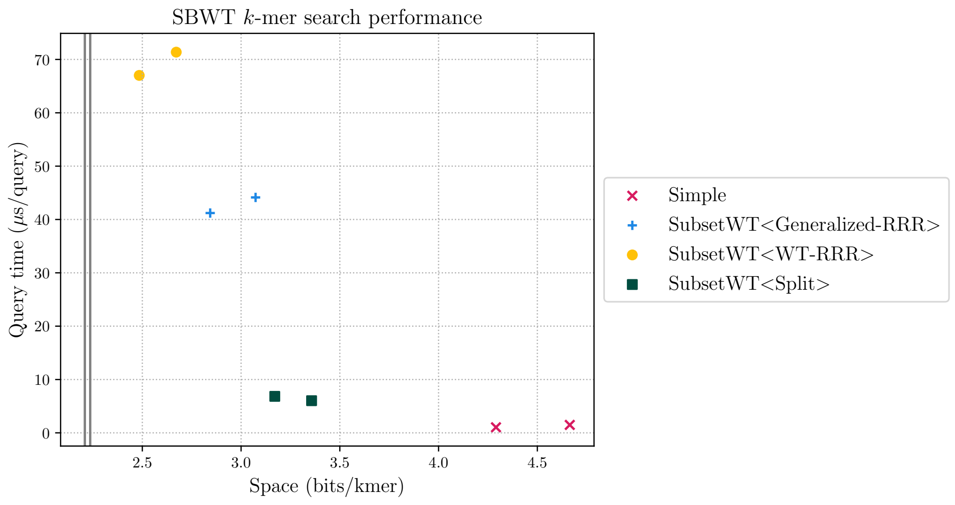

Figure 3 Time and space required for k-mer search on the SBWT using different implementations for subset rank queries. There are two data points per data structure since we have two datasets. For all data structures the metagenome result is the right point and E. coli the left. The two vertical lines mark the entropies of the distribution of subsets in the two datasets.

6 Subset rank query performance on k-mer search of the Spectral BWT

As mentioned at the start of this paper, our main interest in the SubsetWT is for implementing a k-mer search algorithm using the Spectral BWT [3], which reduces a k-mer search query to $2k$ subset rank queries on a degenerate string. In this section, we compare our implementations of the SubsetWT parameterized by different base-3 and base-4 rank structures, to the simple solution of Section 3.

We used the value $k = 31$ in all experiments. The time to load the index into memory was disregarded and the running time includes only the time spent running queries. Table 3 shows the query times against SBWT index structures built for the metagenomic read set and the E. coli genomes. In case of the metagenomic read set, we queried the first 25,000 reads in the dataset, and in case of the E. coli genomes, we queried all k-mers of a single genome in the dataset (assembly id `GCA_000005845`).

The experiments show that the most succinct solution was the SubsetWT with the RRR-encoded wavelet tree for the base-3 and base-4 rank queries, at 2.5 – 2.7 bits per k-mer, but, on the flipside, its query time was the slowest. The generalized RRR was approximately 15% larger, but had approximately 1.6 times faster queries. The next-largest structure was the Split structure, being 26% larger than the RRR wavelet tree, with dramatically improved query time, up to 12 times faster. The plain matrix solution was the largest, being 73% larger than the RRR wavelet tree, with 48 – 64 times faster queries. We omit from the results the SubsetWT parameterized by the scanning solution of Section 5.2 and by the plain bitvector wavelet tree of Section 5.1, since on the DNA alphabet, they are dominated in the time-space plane by the simple solution. They may lead to competitive solutions for degenerate strings on larger alphabets. Figure 3 shows the data points in Table 3 in the time-space plane.

7 Concluding Remarks

We have described the subset wavelet tree – a new data structural tool for degenerate strings. On degenerate strings from a real-world large-scale genomics application, subset wavelet trees offer significant space savings over a non-trivial baseline method, at an acceptable slowdown to query times. Along the way we have described and engineered several rank data structures specialized for ternary and quarternary sequences, which are of independent interest.

The main open problem we leave is to find a tighter analysis of the space required by subset wavelet trees when entropy compression is applied to their node sequences. In particular, can the size of the resulting structure be related in some way to the entropy of the subset sequence. Our experimental results suggest this may well be the case.

Another interesting avenue for future work is to apply the new small alphabet rank data structures we have developed to other settings, for example FM indexes [9] for DNA sequence data, or structures currently of a somewhat esoteric nature, such as multiary wavelet trees [8]. Our results suggest some of our structures (e.g., Scanning) are superior to regular wavelet trees, which until now have been the main practical solution for rank on non-binary sequences and are currently in wide use via the Succinct Data Structures Library [11].

References

1 Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

2 H. Alamro, M. Alzamel, C.S. Iliopoulos, S. P. Pissis, and S. Watts. IUPACpal: efficient identification of inverted repeats in IUPAC-encoded dna sequences. *BMC Bioinformatics*, 22(51), 2021.

3 Jarno N Alanko, Simon J Puglisi, and Jaakko Vuohtoniemi. Succinct k-mer sets using subset rank queries on the spectral Burrows-Wheeler transform. *bioRxiv*, 2022.

4 J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):article 52, 2011.

5 E. Cambouropoulos, T. Crawford, and C.S. Iliopoulos. Pattern processing in melodic sequences: Challenges, caveats and prospects. *Computers and the Humanities*, 35:9–21, 2001.

6 Rayan 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.

7 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Covering problems for partial words and for indeterminate strings. *Theor. Comput. Sci.*, 698:25–39, 2017.

8 P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

9 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, 2000.

10 Michael J. Fischer and Michael S. Paterson. String-matching and other products. *Complexity of Computation*, 7:113–125, 1974.

11 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, LNCS 8504, pages 326–337. Springer, 2014.

12 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014.

13 A. Golynski, I. Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

14 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), !booktitle = "SODA"*, pages 841–850, 2003.

15 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome biology*, 21(1):1–20, 2020.

16 Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008.

17 Costas S. Iliopoulos and Jakub Radoszewski. Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties. In Roberto Grossi and Moshe Lewenstein, editors, *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

18 Costas S. Iliopoulos, M. Sohel Rahman, Michal Vorácek, and Ladislav Vagner. The constrained longest common subsequence problem for degenerate strings. In Jan Holub and Jan Zdárek, editors, *Proc. 12th International Conference on Implementation and Application of Automata (CIAA)*, LNCS 4783, pages 309–311. Springer, 2007.

19 IUPAC-IUB Commission on Biochemical Nomenclature. Abbreviations and symbols for nucleic acids, polynucleotides, and their constituents. *Biochemistry*, 9(20):4022–4027, 1970.

20 Ian B Jeffery, Anubhav Das, Eileen O'Herlihy, Simone Coughlan, Katryna Cisek, Michael Moore, Fintan Bradley, Tom Carty, Meenakshi Pradhan, Chinmay 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.

21 Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *BioRxiv*, 2020.

22 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proceedings of the Data Compression Conference (DCC)*, pages 302–311, 2014.

23 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press, 2015.

24 J. Ian 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. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7354, pages 2–26, 2012.

26 Gonzalo Navarro. *Compact Data Structures – A Practical Approach.* Cambridge University Press, 2016.

27 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Experimental Algorithms: 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings 11*, pages 295–306. Springer, 2012.

28 Rajeev Raman. Rank and select operations on bit strings. In *Encyclopedia of Algorithms*, pages 1772–1775. Springer, 2016. doi:10.1007/978-1-4939-2864-4_332.

29 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

30 Sun Wu and Udi Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Winter 1992 Technical Conference*, pages 153–162, 1992.