Rank and Select on Degenerate Strings

Philip Bille[1], Inge Li Gørtz[1], and Tord Stordalen

DTU Compute, Technical University of Denmark
Lyngby, Denmark
{phbi,inge,tjost}@dtu.dk

Abstract

A *degenerate string* is a sequence of subsets of some alphabet; it represents any string obtainable by selecting one character from each set from left to right. Recently, Alanko et al. generalized the rank-select problem to degenerate strings, where given a character c and position i the goal is to find either the ith set containing c or the number of occurrences of c in the first i sets [SEA 2023]. The problem has applications to pangenomics; in another work by Alanko et al. they use it as the basis for a compact representation of *de Bruijn Graphs* that supports fast membership queries.

In this paper we revisit the rank-select problem on degenerate strings, introducing a new, natural parameter and reanalyzing existing reductions to rank-select on regular strings. Plugging in standard data structures, the time bounds for queries are improved exponentially while essentially matching, or improving, the space bounds. Furthermore, we provide a lower bound on space that shows that the reductions lead to succinct data structures in a wide range of cases. Finally, we provide implementations; our most compact structure matches the space of the most compact structure of Alanko et al. while answering queries twice as fast. We also provide an implementation using modern vector processing features; it uses less than one percent more space than the most compact structure of Alanko et al. while supporting queries four to seven times faster, and has competitive query time with all the remaining structures.

1 Introduction

Given a string S over an alphabet $[1, \sigma]$ the *rank-select problem* is to preprocess S to support, for any $c \in [1, \sigma]$,

- $\mathsf{rank}_S(i, c)$: return the number of occurrences of c in $S[1, i]$
- $\mathsf{select}_S(i, c)$: return the index j of the ith occurrence of c in S

This fundamental string problem has been studied extensively due to its wide applicability, see, e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14], references therein, and survey [15].

A *degenerate string* is a sequence $X = X_1, \ldots X_n$ where each X_i is a subset of $[1, \sigma]$. We define its *length* to be n, its *size* to be $N = \sum_i |X_i|$, and denote by n_0 the number of empty sets among $X_1, \ldots, X_n$. Degenerate strings have been studied since the 80s [16] and the literature contains papers on problems such as degenerate

[1]Supported by Danish Research Council grant DFF-8021-002498.

string comparison [17], finding string covers for degenerate strings [18], and pattern matching with degenerate patterns, degenerate texts, or both [16, 19].

Alanko, Biagi, Puglisi, and Vuohtoniemi [20] recently generalized the rank-select problem to the *subset rank-select problem*, where the goal is to preprocess a given degenerate string X to support

- $\mathsf{subset\text{-}rank}_X(i, c)$: return the number of sets in $X_1, \ldots, X_i$ that contain c
- $\mathsf{subset\text{-}select}_X(i, c)$: return the index of the ith set that contains c

Their motivation for studying this problem is to support fast membership queries on *de Bruijn graphs*, a useful tool in pangenomic problems such as genome assembly and pangenomic read alignment (see [20, 21] for details and further references). Specifically, in another work by some of the authors [21], they show how to represent the de Bruijn graph of all length-k substrings of a given string such that membership queries on the graph can be answered using $2k$ subset-rank queries. They also provide an implementation that, when compared to the previous state of the art, improves query time by one order of magnitude while improving space usage, or by two orders of magnitude with similar space usage.

Their result for subset rank-select is the following [20]. They introduce the *Subset Wavelet Tree*, a generalization of the well-known wavelet tree (see [22]) to degenerate strings. It supports both subset-rank and subset-select queries in $O(\log \sigma)$ time and uses $2(\sigma - 1)n + o(n\sigma)$ bits of space in the general case. In the special case of $n = N$ (which is the case for their representation of de Bruijn Graphs in [21]) they show that their structure uses $2n \log \sigma + o(n \log \sigma)$ bits. We note that their analysis for this special case happens to generalize nicely to also show that their structure uses at most $2N \log \sigma + 2n_0 + o(N \log \sigma + n_0)$ bits for any N.

Furthermore, in [21], Alanko, Puglisi, and Vuohtoniemi present a number of reductions from the subset rank-select to the regular rank-select problem. We will elaborate on these reductions later in the paper.

2 Our Results

Our contributions are threefold. Firstly, we introduce the natural parameter N and revisit the subset rank-select problem to reanalyze a number of simple and elegant reductions to the regular rank-select problem, based on the reductions from [21]. We express the complexities in terms of the performance of a given rank-select structure, achieving flexible bounds that benefit from the rich literature on the rank-select problem (Theorem 1). Secondly, we show that any structure supporting either subset-rank or subset-select must use at least $N \log \sigma - o(N \log \sigma)$ bits in the worst case (Theorem 2). By plugging a standard rank-select data structure into Theorem 1 we, in many cases, match this bound to within lower order terms, while simultaneously matching the query time of the fastest known rank-select data structures (see below). Note that any lower bound for rank-select queries also holds for subset rank-select queries since any string is also a degenerate string. All our results hold on a word

RAM with logarithmic word-size. Finally, we provide implementations of the reductions and compare them to the implementations of the Subset Wavelet Tree provided in [20], and the implementations of the reductions provided in [21]. Our most compact structure matches the space of their most compact structure while answering queries twice as fast. We also provide a structure using vector processing features that matches the space of the most compact structure while imporving query time by a factor four to seven, remaining competitive with the fast structures for queries.

We now elaborate on the points above. The reductions are as follows.

Theorem 1. *Let X be a degenerate string of length n, size N, and with n_0 empty sets over an alphabet $[1, \sigma]$. Let $\mathcal{D}$ be a $\mathcal{D}_b(\ell, \sigma)$-bit data structure for a length-ℓ string over $[1, \sigma]$ that supports* rank *in $\mathcal{D}_r(\ell, \sigma)$ time and* select *in $\mathcal{D}_s(\ell, \sigma)$ time. If $n_0 = 0$ we can solve subset rank-select on X in*
(i) $\mathcal{D}_b(N, \sigma) + N + o(N)$ bits, $\mathcal{D}_r(N, \sigma) + O(1)$ subset-rank*-time, and $\mathcal{D}_s(N, \sigma) + O(1)$* subset-select*-time.*
Otherwise, if $n_0 > 0$ we can solve subset rank-select on X in
(ii) the bounds in (i) where we replace N by $N' = N + n_0$ and σ by $\sigma' = \sigma + 1$.
(iii) the bounds in (i) with additional $\mathcal{B}_b(n, n_0)$ bits of space, additional $\mathcal{B}_r(n, n_0)$ time for subset-rank*, and additional $\mathcal{B}_s(n, n_0)$ time for* subset-select*. Here $\mathcal{B}$ is a data structure on a length-n bitstring that contains n_0 1s, uses $\mathcal{B}_b(n, n_0)$ bits, and supports* rank$(\cdot, 1)$ *in $\mathcal{B}_r(n, n_0)$ time and* select$(\cdot, 0)$ *in $\mathcal{B}_s(n, n_0)$ time.*

Here Theorem 1(i) and (ii) are based on the reduction from [21, Sec. 4.3], and Theorem 1(iii) is a variation of Theorem 1(ii) that handles empty sets using a natural, alternative strategy. By plugging a standard rank-select structure into Theorem 1 we exponentially improve query times while essentially matching, or improving, space usage compared to Alanko et al. [20]. For example, consider the rank-select structure by Golynski, Munro, and Rao [3] which uses $\ell \log \sigma + o(\ell \log \sigma)$ bits, supports rank in $O(\log \log \sigma)$ time, and supports select in constant time. These query times are optimal in succinct space, see e.g. [1].

For $n_0 = 0$, plugging this structure into Theorem 1(i) yields an $N \log \sigma + N + o(N \log \sigma + N)$ bit data structure supporting subset-rank in $O(\log \log \sigma)$ time and subset-select in constant time. Compared to the previous result by Alanko et al. [20], this improves the constant on the space bound from 2 to $1 + 1/\log \sigma$ and improves the query time from $O(\log \sigma)$ for both queries to $O(\log \log \sigma)$ for subset-rank and constant for subset-select. Note that the additional N bits in the space bound are a lower order term when $\sigma = \omega(1)$.

For $n_0 > 0$, plugging their structure into Theorem 1(ii) gives the same time bounds as above and the space bound

$$(N + n_0) \log(\sigma + 1) + (N + n_0) + o(n_0 \log \sigma + N \log \sigma + N + n_0)$$

bits. If $n_0 = o(N)$ and $\sigma = \omega(1)$, the space bound is identical to the one above. In any case, the query time is still improved exponentially.

Alternatively, by plugging it into Theorem 1(iii) the space bound becomes $N \log \sigma + o(N \log \sigma) + \mathcal{B}_s(n, n_0)$ bits. For $n = o(N \log \sigma)$ we can choose $\mathcal{B}$ to be an $(n + o(n))$-bit data structure with constant time rank and select, such as [23,24], again achieving

the same space and time bounds as when $n_0 = 0$. Otherwise, we can plug in any data structure for $\mathcal{B}$ that is sensitive to the number of 1-bits in the bitvector. For example, if $n_0 = O(\log n)$ we can store the positions of the 1-bits in sorted order using $O(n_0 \log n) = O(\log^2 n)$ bits, supporting $\mathsf{select}(i, \mathtt{1})$ in constant time and $\mathsf{rank}(i, \cdot)$ in $O(\log n_0) = O(\log \log n)$ time using binary search. We can also binary search for $\mathsf{select}(i, \mathtt{0})$ in $O(\log n_0) = O(\log \log n)$ time using the fact that — if the ith position of a 1-bit is p_i — there are $p_i - i$ zeroes in the prefix ending at p_i. There are many such sensitive data structures that obtain various time-space trade-offs, e.g [2, 25].

We also show the following lower bound on the space required to support either subset-rank or subset-select on a degenerate string.

Theorem 2. *Let X be a degenerate string of size N over an alphabet $[1, \sigma]$. Any data structure supporting subset-rank or subset-select on X must use at least $N \log \sigma - o(N \log \sigma)$ bits in the worst case.*

Thus, applying Theorem 1 in many cases results in *succinct* data structures, whose space deviates from this lower bound by at most a lower order term. The three examples above each illustrate this when respectively (1) $\sigma = \omega(1)$, (2) $n_0 = o(N)$ and $\sigma = \omega(1)$, and (3) $n = o(N \log \sigma)$.

Finally, we provide implementations and compare them to variants of the Subset Wavelet Tree [20] and the reductions [21] implemented by Alanko et al. Specifically, we apply the test framework from [20] and run two types of tests: one where the subset rank-select structures are used to support k-mer queries on a de Bruijn Graph (the motivation for, and practical application of, the subset rank-select problem), and one where subset-rank queries are tested in isolation. We implement Theorem 1(iii) and plug in efficient off-the-shelf rank-select structures from the *Succinct Data Structure Library (SDSL)* [26] (`https://github.com/simongog/sdsl-lite`). We also implement a variation of another reduction from [21, Sec. 4.2], which is more optimized for genomic test data. The highlight is our most compact structure, which matches the space of their most compact structure while supporting queries twice as fast, as well as our structure using vector processing, which matches the most compact structure while supporting queries four to seven times faster.

3 Reductions

We now present the reductions from Theorem 1. Let X, $\mathcal{D}$, and $\mathcal{B}$ be defined as in Theorem 1. Furthermore, let $\mathcal{V}$ be the data structure from [24], which for a length-ℓ bitstring uses $\ell + o(\ell)$ bits and supports rank and select in constant time.

Reductions (i) and (ii)

First assume that $n_0 = 0$. For each X_i let the string S_i be the concatenation of the characters in X_i in an arbitrary order, and let the string R_i be a single $\mathtt{1}$ followed by $|X_i| - 1$ $\mathtt{0}$s. This is always possible since $|X_i| \geq 1$. Let S (resp. R) be the concatenation of $S_1, \ldots, S_n$ (resp. $R_1, \ldots, R_n$) in that order, with an additional $\mathtt{1}$ appended after R_n. The lengths of S and R are respectively N and $N + 1$. See

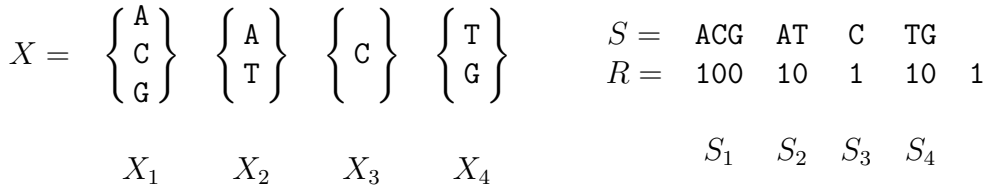

Figure 1: *Left:* A degenerate string X over the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ where $n = 4$ and $N = 8$. *Right:* The reduction from Theorem 1(i) on X. White space is for illustration purposes only. To compute $\mathsf{subset\text{-}rank}(2, \mathtt{A})$, we first compute $\mathsf{select}_R(3, \mathtt{1}) = 6$. Now we know that S_2 ends at position 5, so we return $\mathsf{rank}_S(5, \mathtt{A}) = 2$. To compute $\mathsf{subset\text{-}select}(2, \mathtt{G})$ we compute $\mathsf{select}_S(2, \mathtt{G}) = 8$, and compute $\mathsf{rank}_R(8, \mathtt{1}) = 4$ to determine that position 8 corresponds to X_4.

Figure 1 for an example. The data structure consists of $\mathcal{D}$ built over S and $\mathcal{V}$ built over R, which takes $\mathcal{D}(N, \sigma) + N + o(N)$ bits.

To support $\mathsf{subset\text{-}rank}(i, c)$, compute the starting position $k = \mathsf{select}_R(i + 1, \mathtt{1})$ of S_{i+1} and return $\mathsf{rank}_S(k - 1, c)$. To support $\mathsf{subset\text{-}select}(i, c)$, find the index $k = \mathsf{select}_S(i, c)$ of the ith occurrence of c, and return $\mathsf{rank}_R(k, \mathtt{1})$ to determine which set k is in. Since rank and select queries on R take constant time, $\mathsf{subset\text{-}rank}$ and $\mathsf{subset\text{-}select}$ queries take respectively $\mathcal{D}_r(N, \sigma) + O(1)$ and $\mathcal{D}_s(N, \sigma) + O(1)$ time, achieving the bounds stated in Theorem 1(i).

If $n_0 \neq 0$, add a new character $\sigma+1$ and replace each empty set with the singleton set $\{\sigma+1\}$, and then apply reduction (i). This instance has $N' = N+n_0$ and $\sigma' = \sigma+1$, achieving the bounds in Theorem 1(ii).

Reduction (iii)

Let E denote the length-n bitvector where $E[i] = 1$ if $X_i = \emptyset$ and $E[i] = 0$ otherwise. Let X'' denote the degenerate string obtained by removing all the empty sets from X. The data structure consists of reduction (i) over X'' and $\mathcal{B}$ built over E. This takes $\mathcal{D}_b(N, \sigma) + N + o(N) + \mathcal{B}_b(n, n_0)$ bits. To support $\mathsf{subset\text{-}rank}_X(i, c)$ first compute $k = i - \mathsf{rank}_E(i, \mathtt{1})$, mapping X_i to its corresponding set X''_k. Then return $\mathsf{subset\text{-}rank}_{X''}(k, c)$. This takes $\mathcal{B}_r(n, n_0) + \mathcal{D}_r(N, \sigma) + O(1)$ time. To support $\mathsf{subset\text{-}select}_X(i, c)$, find $k = \mathsf{subset\text{-}select}_{X''}(i, c)$ and return $\mathsf{select}_E(k, \mathtt{0})$, the position of the kth zero in E (i.e., the kth non-empty set). This takes $\mathcal{B}_s(n, n_0) + \mathcal{D}_s(N, \sigma) + O(1)$, matching the stated bounds.

4 Lower Bound

In this section we prove Theorem 2. The strategy is as follows. Any structure supporting $\mathsf{subset\text{-}rank}$ or $\mathsf{subset\text{-}select}$ on X is a representation of X since we can fully recover X by repeatedly using either of these operations. We will lower bound the number L of distinct degenerate strings that can exist for a given N and σ. Any representation of X must be able to distinguish between these instances, so it needs to use at least $\log_2 L$ bits in the worst case. Let sufficiently large N and $\sigma = \omega(\log N)$ be given and assume without loss of generality that $\log N$ and $N/\log N$ are integers.

Consider the class of degenerate strings $X_1, \ldots, X_n$ where each $|X_i| = \log N$ and $n = N/\log N$. There are $\binom{\sigma}{\log N}^{N/\log N}$ such degenerate strings, so any representation must use at least

$$\begin{aligned}
\log \binom{\sigma}{\log N}^{N/\log N} &= \frac{N}{\log N} \log \binom{\sigma}{\log N} \\
&\geq \frac{N}{\log N} \log \left(\frac{\sigma - \log N}{\log N} \right)^{\log N} \\
&= N \log \left(\frac{\sigma - \log N}{\log N} \right) \\
&= N \log \sigma - o(N \log \sigma)
\end{aligned}$$

bits, concluding the proof.

5 Experimental Setup

Setup and Data

The code to replicate our results is available on GitHub (`https://github.com/tstordalen/subset-rank-select`). Our tests are based on the test framework by Alanko et al. [20] (`https://github.com/jnalanko/SubsetWT-Experiments/`). Like them, we used the following data sets.

1. A pangenome of 3682 E. coli genomes, available on Zenodo (`https://zenodo.org/record/6577997`). According to [20], the data was collected by downloading a set of 3682 E. Coli assemblies from the National Center for Biotechnology Information.

2. A human metagenome (SRA identifier ERR5035349) consisting of a set of ≈ 17 million length-502 sequence reads sampled from the human gut from a study on irritable bowel syndrome and bile acid malabsorption [27].

We applied two tests. Firstly, we plugged our data structures into the k-mer query test from [20]; they plug subset rank-select structures into their k-mer index and query a large number of k-mers. Secondly, we tested the subset rank-select structures in isolation by building the k-mer indices, extracting the subset rank-select structures, and performing twenty million randomly generated subset-rank queries. For each measurement we built only the structure under testing, and timed only the execution of the queries. Each value reported below is the average of five such measurements. Note that, like [20], we do not test subset-select queries; only subset-rank queries are necessary for their k-mer index.

All the tests were run on a system with a 3.00GHz i7-1185G7 processor and 32 gigabytes of DDR4 random access memory, running Ubuntu 22.04.3 LTS with kernel version 6.2.0-35-generic. The programs were compiled using g++ version 11.4.0 with compiler flags `-O3`, `-march=native`, and `-DNDEBUG`.

Data Structures

This section summarizes a subset of the data structures we tested (the ones we have omitted do not affect the results; see the full version [28]). We implement both Theorem 1(iii) as well as variation of the reduction *split representation* from [21, Sec 4.2]; this reduction is optimized for their k-mer query structure built over genomic data, in which most of the sets are singletons. We name our variation the *dense-sparse decomposition (DSD)*, which works as follows. The empty sets are handled in the same way as in Theorem 1(iii). Furthermore, we store a sparse bitvector of length n for each character, i.e., `A`, `C`, `G`, and `T`. For each X_i of size at least two we remove $|X_i| - 1$ of the characters and set the ith bit in the corresponding bitvector to 1. What remains are $n - n_0$ singleton sets, i.e., a regular string, for which we store a rank-select structure. A query thus consists of three rank queries; one to eliminate empty sets, one in the regular string, and one in the sparse bitvector. In the split representation by [21], each such set is instead removed and *all* the characters are represented in the additional bitvectors.

The data structures we tested are as follows. **Matrix** is the benchmark structure from [20], consisting of one bitvector per character (i.e., a $4 \times n$ matrix). **Thm 1(iii)** is the reduction from Theorem 1(iii), using a wavelet tree for the string, a bitvector for the length-N indicator string, and a sparse bitvector for the empty sets. **DSD (x)**, **SWT (x)**, and **Split (x)** are the DSD, Subset Wavelet Tree, and split representation parameterized by **x**, respectively, where **x** may be any of the following data structures: *(1)* **scan**, the structure from Alanko et al. [20, Sec. 5.2], inspired by scanning techniques for fast `rank` queries on bitvectors, *(2)* **split**, a rank structure for size-four alphabets optimized for the skewed distribution of singleton to non-singleton sets [20, Sec 5.3] (not to be confused with the split representation) *(3)* **rrr**, an SDSL wavelet tree using H_0-compressed bitvectors, based mainly on the result by Raman, Raman, and Rao Satti [4], *(4)* **rrr gen.**, a generalization of RRR to size-four alphabets [20, Sec. 5.4], *(5)* **ef**, an efficient implementation of rank queries on a bitvector stored using Elias-Fano encoding from [29], or *(6)* **plain**, a standard SDSL bitvectors supporting rank in constant time.

Furthermore, [21] implements Concat (rrr), which is essentially reduction (ii) using a wavelet tree with RRR-compressed bitvectors, and we also implement the structure DSD (SIMD). It is based on a standard idea for compact data structures; we divide the string into blocks, precompute the answer to rank queries up to each block, and compute partial rank queries for blocks as needed using word parallelism. This is the essence of how the 'scan' structure by [20] works; we use *SIMD (single instruction, multiple data)* instructions to speed up the process further, allowing for large blocks, further reducing space. Most computers support SIMD to some extent, allowing the same operation to be performed on many words simultaneously. We used AVX512, which supports 512-bit vector registers.

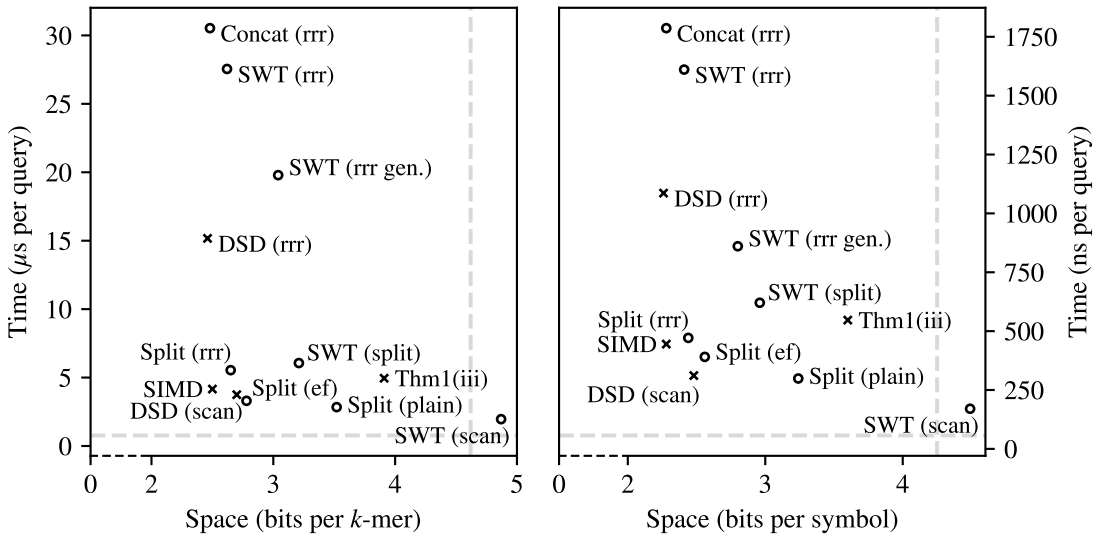

Figure 2: Note that the x-axis is truncated in both plots. The two gray lines represent the performance of the benchmark solution "Matrix". The crosses indicate our data structures and the circles indicate the data structures from [20, 21]. *Left:* Results of the k-mer query test on the metagenome data set. *Right:* The result of the subset-rank test on the metagenome data set. The space is in number of bits per symbol, i.e., bits/N.

6 Results

The test results for the metagenome data set can be seen in Figure 2; the results for the E. Coli data set are similar. See appendix A for all the data. The fastest structure is SWT (scan), but it is large and is outperformed by the benchmark solution on both parameters. Our unoptimized reduction Thm1(iii) uses $20-60\%$ more space than the remaining structures of [20, 21] while remaining within a factor two in query time of most of them. Our fastest structure, DSD (scan), is competitive with both Split (ef) and Split (rrr). Our most compact structure DSD (rrr) matches the space of the previous smallest structure, Concat (ef), while supporting queries twice as fast. Our SIMD-enhanced structure uses less than one percent more space than Concat (ef) while supporting queries four to seven times faster. It is also competitive with the fast and compact structures Split (ef) and Split (rrr). We note that the entropies for the distributions of sets in the Metagenome and E. Coli data sets are respectively 2.21 and 2.24 bits (as seen in [20]), and that reduction from 2.44 bits (Split (rrr), Metagenome) to 2.28 bits (SIMD, Metagenome) reduces the distance to the entropy from approximately 10% to 3%, while simultaneously supporting queries faster.

References

[1] Djamal Belazzougui and Gonzalo Navarro, "Optimal lower and upper bounds for representing sequences," *ACM Trans. Algorithms*, vol. 11, no. 4, pp. 31:1–31:21, 2015.

[2] Daisuke Okanohara and Kunihiko Sadakane, "Practical Entropy-Compressed Rank/Select Dictionary," in *Proc. 9th ALENEX*, 2007.

[3] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao, "Rank/select operations on large alphabets: a tool for text indexing," in *Proc. 15th SODA*, 2006, pp. 368–373.

[4] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, pp. 43, 2007.

[5] Alberto Ordóñez Pereira, Gonzalo Navarro, and Nieves R. Brisaboa, "Grammar compressed sequences with rank/select support," *J. Discrete Algorithms*, vol. 43, pp. 54–71, 2017.

[6] Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei, "Access, rank, and select in grammar-compressed strings," in *Proc. 23rd ESA*, 2015, pp. 142–154.

[7] Veli Mäkinen and Gonzalo Navarro, "Rank and select revisited and extended," *Theor. Comput. Sci.*, vol. 387, no. 3, pp. 332–347, 2007.

[8] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro, "Compressed representations of sequences and full-text indexes," *ACM Trans. Algorithms*, vol. 3, no. 2, pp. 20, 2007.

[9] Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti, "Succinct indexes for strings, binary relations and multilabeled trees," *ACM Trans. Algorithms*, vol. 7, no. 4, pp. 52:1–52:27, 2011.

[10] Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich, "Efficient fully-compressed sequence representations," *Algorithmica*, vol. 69, no. 1, pp. 232–268, 2014.

[11] Gonzalo Navarro and Kunihiko Sadakane, "Fully functional static and dynamic succinct trees," *ACM Trans. Algorithms*, vol. 10, no. 3, pp. 16:1–16:39, 2014.

[12] Meng He and J. Ian Munro, "Succinct representations of dynamic strings," in *Proc. 17th SPIRE*, 2010, pp. 334–346.

[13] Gonzalo Navarro and Yakov Nekrich, "Optimal dynamic sequence representations," *SIAM J. Comput.*, vol. 43, no. 5, pp. 1781–1806, 2014.

[14] Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini, "Dynamic compressed strings with random access," in *Proc. 40th ICALP*, 2013, pp. 504–515.

[15] Travis Gagie, "Rank and select operations on sequences," in *Encyclopedia of Algorithms*, pp. 1776–1780. 2016.

[16] Karl R. Abrahamson, "Generalized string matching," *SIAM J. Comput.*, vol. 16, no. 6, pp. 1039–1051, 1987.

[17] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone, "Comparing degenerate strings," *Fundam. Informaticae*, vol. 175, no. 1-4, pp. 41–58, 2020.

[18] 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.*, vol. 698, pp. 25–39, 2017.

[19] Costas S. Iliopoulos, Laurent Mouchard, and Mohammad Sohel Rahman, "A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching," *Math. Comput. Sci.*, vol. 1, no. 4, pp. 557–569, 2008.

[20] Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi, "Subset wavelet trees," in *Proc. 21st SEA*, 2023, pp. 4:1–4:14.

[21] Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi, "Small searchable κ-spectra via subset rank queries on the spectral burrows-wheeler transform," in *Proc. ACDA, 2023*, 2023, pp. 225–236.

[22] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th SODA*, 2003, pp. 841–850.

[23] David R. Clark and J. Ian Munro, "Efficient suffix trees on secondary storage (extended abstract)," in *Proc. 7th SODA*, 1996, pp. 383–391.

[24] Guy Jacobson, "Space-efficient static trees and graphs," in *Proc. FOCS*, 1989, pp. 549–554.

[25] Alexander Golynski, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao, "Optimal indexes for sparse bit vectors," *Algorithmica*, vol. 69, no. 4, pp. 906–924, 2014.

[26] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri, "From theory to practice: Plug and play with succinct data structures," in *Proc. 13th SEA*, 2014, pp. 326–337.

[27] 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*, vol. 158, no. 4, pp. 1016–1028, 2020.

[28] Philip Bille, Inge Li Gørtz, and Tord Stordalen, "Rank and select on degenerate strings," 2023, *arXiv*, arXiv:2310.19702.

[29] Danyang Ma, Simon J Puglisi, Rajeev Raman, and Bella Zhukova, "On elias-fano for rank queries in fm-indexes," in *Proc. DCC, 2021*, 2021, pp. 223–232.

A Additional Data

	k-mer Queries				Subset Rank Queries			
	E. Coli		Metagenome		E. Coli		Metagenome	
Data structure	Query (μs)	Space (bpk)	Query (μs)	Space (bpk)	Query (ns)	Space (bps)	Query (ns)	Space (bps)
Matrix	0.63	4.29	0.77	4.62	38.75	4.26	56.98	4.25
DSD (scan)	3.00	2.61	3.75	2.70	210.23	2.57	311.33	2.48
Thm1(iii)	3.87	3.68	4.95	3.91	435.28	3.64	546.89	3.60
DSD (rrr)	13.21	2.38	15.17	2.46	850.99	2.34	1086.11	2.26
SIMD	3.31	2.42	4.16	2.50	320.53	2.37	444.94	2.28
SWT (scan)	1.63	4.53	1.96	4.87	129.44	4.49	170.44	4.49
SWT (split)	4.93	3.17	6.06	3.21	436.69	3.13	620.47	2.96
SWT (rrr gen.)	18.97	2.84	19.79	3.04	789.12	2.81	860.4	2.80
SWT (rrr)	25.33	2.48	27.55	2.62	1384.0	2.45	1610.73	2.41
Split (plain)	2.28	3.30	2.84	3.52	235.22	3.27	298.87	3.24
Split (ef)	2.71	2.69	3.30	2.78	317.71	2.65	390.65	2.56
Split (rrr)	4.70	2.54	5.54	2.65	393.14	2.51	471.30	2.44
Concat(ef)	26.25	2.38	30.53	2.48	1372.2	2.35	1786.65	2.28

Table 1: The left half of the table shows the result for the k-mer query test. The times are listed in microseconds per query, and space in the number of bits per represented k-mer. The right half shows the result of the subset-rank query test. Times are listed in nanoseconds per query, and space in bits per symbol (i.e., the number of bits divided by N). There are five groups, separated by horizontal lines; the benchmark structure, our reductions, our structure using SIMD, the Subset Wavelet Trees from [20], and the reductions from [21]. Each group is ordered from fastest to slowest and largest to smallest, except for Thm1(iii) which breaks space order. Each value in the table is the average of five measurements.