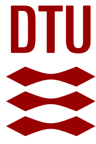
Downloaded from orbit.dtu.dk on: May 15, 2024



DTU Library

Algorithms for Strings and Modern Data Models

Stordalen, Tord Joakim

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Stordalen, T. J. (2023). *Algorithms for Strings and Modern Data Models.* Technical University of Denmark.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Algorithms for Strings and Modern Data Models

Tord Joakim Stordalen

December 2023

Preface

The research presented in this dissertation was done while I was enrolled as a PhD student at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, from January 1st, 2021 to December 31st, 2023. It was funded by the DTU Compute PhD scholarship. My advisors are Professor Philip Bille and Professor Inge Li Gørtz. During my studies I spent three months at Universidad de Chile, visiting Professor Gonzalo Navarro.

Acknowledgments I am deeply grateful to my advisors Philip Bille and Inge Li Gørtz for introducing me to field of algorithms, for all the time they spent teaching me how to think, write, and teach, for being patient, understanding, and supportive, for setting standards I had to work to achieve, for challenging my ideas and opinions, and for being open to being challenged by me in return. They are truly excellent advisors. I would also like to thank all my co-authors for being inspiring collaborators; this includes both my advisors, Johannes Fischer, and Max Rishøj Pedersen. Thank you to Gonzalo Navarro for hosting me during my external stay in Chile. Finally, I would like to thank all my colleagues and friends at DTU for all the wonderful coffee breaks, banter, and interesting conversations, and my family and friends for their support throughout my studies.

Abstract

We study problems related to pattern matching in strings, and problems in modern data models. Specifically, we study the predecessor problem in a model of computation that captures modern vector processor architectures, the problem of compactly and efficiently counting the co-occurrences of a set of characters in a string, pattern matching in a sliding window over a stream, and the problem of supporting rank and select on degenerate strings.

Predecessor on the Ultra-Wide Word RAM We consider the predecessor problem on the ultra-wide word RAM model of computation, which extends the word RAM model with *ultrawords* consisting of w^2 bits [TAMC, 2015]. The model supports arithmetic and boolean operations on ultrawords, in addition to *scattered* memory operations that access or modify w (potentially non-contiguous) memory addresses simultaneously. The ultra-wide word RAM model captures (and idealizes) modern vector processor architectures. Our main result is a simple, linear space data structure that supports predecessor in constant time and updates in amortized, expected constant time. This improves the space of the previous constant time solution that uses space in the order of the size of the universe. Our result holds even in a weaker model where ultrawords consist of $w^{1+\epsilon}$ bits for any $\epsilon > 0$. It is based on a new implementation of the classic x-fast trie data structure of Willard [Inform. Process. Lett. 17(2), 1983] combined with a new dictionary data structure that supports fast parallel lookups.

The Complexity of the Co-Occurrence Problem Let S be a string of length n over an alphabet Σ and let Q be a subset of Σ of size $q \geq 2$. The *co-occurrence problem* is to construct a compact data structure that supports the following query: given an integer w return the number of length-w substrings of S that contain each character of Q at least once. This is a natural string problem with applications to, e.g., data mining, natural language processing, and DNA analysis. The state of the art is an $O(\sqrt{nq})$ space data structure that — with some minor additions — supports queries in $O(\log\log n)$ time [CPM 2021].

Our contributions are as follows. Firstly, we analyze the problem in terms of a new, natural parameter d, giving a simple data structure that uses $O(d)$ space and supports queries in $O(\log\log n)$ time. The preprocessing algorithm does a single pass over S, runs in expected $O(n)$ time, and uses $O(d + q)$ space in addition to the input. Furthermore, we show that $O(d)$ space is optimal and that $O(\log\log n)$-time queries are optimal given optimal space. Secondly, we bound $d = O(\sqrt{nq})$, giving clean bounds in terms of n and q that match the state of the art. Furthermore, we prove that $\Omega(\sqrt{nq})$ bits of space is necessary in the worst case, meaning that the $O(\sqrt{nq})$ upper bound is tight to within polylogarithmic factors. All of our results are based on simple and intuitive combinatorial ideas that simplify the state of the art.

Sliding Window String Indexing in Streams Given a string S over an alphabet Σ, the *string indexing problem* is to preprocess S to subsequently support efficient pattern matching queries, that is, given a pattern string P report all the occurrences of P in S. In this paper we study the *streaming sliding window string indexing problem.* Here the string S arrives as a stream, one character at a time, and the goal is to maintain an index of the last w characters, called the *window*, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and all occurrences of P within the current window must be returned. The streaming sliding window string indexing problem

naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching.

Our main result is a simple $O(w)$ space data structure that uses $O(\log w)$ time with high probability to process each character from both the input string S and any pattern string P. Reporting each occurrence of P uses additional constant time per reported occurrence. Compared to previous work in similar scenarios this result is the first to achieve an efficient worst-case time per character from the input stream with high probability. We also consider a delayed variant of the problem, where a query may be answered at any point within the next δ characters that arrive from either stream. We present an $O(w+\delta)$ space data structure for this problem that improves the above time bounds to $O(\log(w/\delta))$. In particular, for a delay of $\delta = \epsilon w$ we obtain an $O(w)$ space data structure with constant time processing per character. The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees of independent interest, inspired by the classic log-structured merge trees.

Rank and Select on Degenerate Strings 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.

Contents

Preface **i**

Contents **v**

1 Introduction **3**
1.1 Models of Computation . . . 3
1.2 Predecessor on the Ultra-Wide Word RAM . . . 4
1.3 The Complexity of the Co-Occurrence Problem . . . 6
1.4 Sliding Window String Indexing in Streams . . . 7
1.5 Rank and Select . . . 9

2 Predecessor on the Ultra-Wide Word RAM **11**
2.1 Introduction . . . 12
2.2 The Ultra-Wide Word RAM Model . . . 15
2.3 Computing Multiply-Shift in Parallel . . . 16
2.4 The w^ϵ-Parallel Dictionary . . . 17
2.5 The $x$tra-fast Trie . . . 20
2.6 The $x$tra-fast Trie With Smaller Ultrawords . . . 24
2.7 Conclusions and Open Problems . . . 25
2.A Blend and $2w$-bit Multiplication . . . 25

3 The Complexity of the Co-Occurrence Problem **29**
3.1 Introduction . . . 30
3.2 The Left-Minimal Co-Occurrence Problem . . . 33
3.3 The Co-Occurrence Problem . . . 35
3.4 Lower Bounds . . . 36
3.A Preprocessing . . . 39
3.B Lower Bound on Time . . . 40

4 Sliding Window String Indexing in Streams **41**
4.1 Introduction . . . 42
4.2 Preliminaries . . . 45
4.3 The Timely SSWSI Problem . . . 46
4.4 The Delayed SSWSI Problem . . . 51
4.5 Obtaining High Probability . . . 53
4.6 Conclusion and Future Work . . . 54

5 Rank and Select on Degenerate Strings **55**
5.1 Introduction . . . 56
5.2 Our Results . . . 57
5.3 Reductions . . . 58
5.4 Lower Bound . . . 59
5.5 Experimental Setup . . . 60
5.6 Results . . . 62
5.A Additional Data . . . 62
5.B Results for all Data Structures . . . 63

Bibliography **67**

Chapter 1

Introduction

This dissertation is based on the (revised) full versions of the following papers.

Chapter 2 Predecessor on the Ultra-Wide Word RAM. Philip Bille, Inge Li Gørtz, Tord Joakim Stordalen. Full version: *Algorithmica* (accepted without reservations as of December 2023; not yet published). Conference version: *Proc. 18th SWAT*, 2022, pages 18:1 - 18:15.

Chapter 3 The Complexity of the Co-Occurrence Problem. Philip Bille, Inge Li Gørtz, Tord Joakim Stordalen. *Proc. 29th SPIRE*, 2022, pages 38-52.

Chapter 4 Sliding Window String Indexing in Streams. Philip Bille, Johannes Fischer, Inge Li Gørtz, Max Rishøj Pedersen, Tord Joakim Stordalen. *Proc. 34th CPM*, 2023, pages 4:1 - 4:18.

Chapter 5 Rank and Select on Degenerate Strings. Philip Bille, Inge Li Gørtz, Tord Stordalen. Accepted for publication at DCC 2024 (Data Compression Conference); not yet published.

The rest of this section briefly introduces the history of each problem, our results, and our techniques.

1.1 Models of Computation

Word RAM The word RAM model of computation was proposed by Hagerup [Hag98]. The memory is an infinite array consisting of w-bit cells (*words*) for some positive integer parameter w. The model supports reads and writes on the memory, standard arithmetic operations on w-bit integers (addition, subtraction, multiplication, division), standard bitwise operations (or, and, exclusive or, negation), and other standard bit manipulation operations such as left and right shifts. Each of these instructions take constant time, time complexity is measured in the number of instructions run by an algorithm, and the space complexity of a data structure is measured in the number of words stored. We make the common assumption that $w \geq \log n$ (where n is the input size) such that a pointer or an index into our input can be stored in a single word. We allow our algorithms to use a constant number of w-bit constants.

Ultra-Wide Word RAM The Ultra-Wide Word RAM (UWRAM) model is a variant of the word RAM proposed by Farzan, Lopés-Ortiz, Nicholson, and Salinger [FLNS15]. It extends the word RAM with w^2-bit words named *ultrawords*. Ultrawords are stored in memory as length-w arrays, can be written and read in constant time, and use $O(w)$ space. The model supports constant time addition, subtraction, bitwise operations, and bit shifts on ultrawords, in addition to the standard word RAM operations on regular w-bit words. Furthermore, the model also supports *scattered* memory operations, allowing reads and writes of w non-contiguous words in constant time.

Farzan et al. introduce both the *restricted* and the *multiplication* variants of the model, which respectively do not and do support multiplication on ultrawords. This distinction highlights a practical concern, as the

size of the smallest known multiplication circuit for two w-bit integers is $\Theta(w \log w)$ [HVDH21]. Therefore, assuming a multiplication circuit for w^2-bit integers may not be realistic. In Chapter 2 we compromise by allowing *component-wise* multiplication; an ultraword can be seen as consisting of w regular words (or *components*) and we allow multiplying two ultrawords component by component, i.e., w parallel multiplications. This is consistent with the type of multiplication supported by modern vector hardware. Furthermore, we show all our results also in a weaker model where ultrawords consist of only $w^{1+\epsilon}$ bits for any $\epsilon > 0$, i.e., only ϵ components.

Streaming In the streaming model, we assume that the input arrives one element at a time, as opposed to being available up-front, and we must process each element as it arrives. When designing algorithms, the goal is to minimize both the cost per processed element, and the number of words stored by the algorithm. Motivations for this type of model include, e.g., computing statistics of network activity in routers (which have limited memory and must be fast), processing data that is stored on slow, external storage[1], and constructing small-space summaries of large data sets. One type of streaming model is the *sliding window* model where the goal is to maintain some data structure or computation over a fixed-width segment that is being slid across the input, one element at a time. We use the (sliding window) streaming model in Chapters 3 and 4.

1.2 Predecessor on the Ultra-Wide Word RAM

Problem Definition and History Given a subset S of a totally ordered universe U, the *static predecessor problem* is to preprocess S to support the operation $\mathsf{predecessor}(x) = \max\{y \in S \mid y \le x\}$ (or NONE if x has no predecessor). The dynamic variant also supports updates on S. This is a fundamental problem in computer science that has been studied extensively [PT06, PT14, vEBKZ77, Wil83, FW93, And96, BF02, vEB77, BBV10, BBPV09, Ajt88, BF02, Mil94, MNSW98, SV08, PT06, PT07] with applications to, for instance, integer sorting [And96, AHNR98, FW93, Han04], string sorting [AFGV97, BFK06, Far97], and string searching [Bel12, BBV10, BEGV18, BGS17, BLR+15]. See Navarro and Rojas-Ledesma [NR20] for a recent survey.

Arguably, the most well-known type of solutions to the predecessor problem are balanced binary search trees such as, e.g., red-black trees [GS78] and splay trees [ST83]. These data structures work for any universe U whose elements can be compared, and each operation takes $O(\log n)$ comparisons where $n = |S|$ (for splay trees the bound is amortized).

In the word RAM model and for integer universes $U = \{0, 1, \ldots, u\}$, it is possible to do much better using techniques that go beyond comparisons (e.g., hashing) and by exploiting the representation of the elements of S. For instance, van Emde Boas [vEB77] gave a now famous data structure using $O(u)$ space and supporting predecessor in $O(\log \log u)$ time by dividing the universe into smaller universes of size $\sqrt{u}$, storing each element in the corresponding sub-universe, and recursively solving predecessor in each subproblem. Other famous data structures include the x-fast and y-fast tries by Willard [Wil83]; the y-fast trie uses $O(n)$ space and supports all operations in $O(\log \log u)$ expected time (updates are amortized). The main idea is to store all prefixes of all keys in S in a hash table and, given a query x, binary search over the binary representation of x to find the longest prefix of x that occurs as a prefix in S. Fredman and Willard [FW93] presented the *fusion tree*, which supports predecessor queries on sets of size $w^{O(1)}$ in constant time (recall that w is the word width). Pătraşcu and Thorup [PT14] presented the *dynamic fusion tree* which also supports updates in constant time on sets of size $w^{O(1)}$. Andersson [And96] presented the *exponential search tree* supporting predecessor in $O(\sqrt{\log n})$. There are many more results for the predecessor problem; see [NR20] for a comprehensive survey.

Pătraşcu and Thorup presented new lower bounds, new upper bounds, and modifications of existing bounds to show the following tight upper and lower bound [PT06, PT07, PT14] for the dynamic predecessor

[1] Other models, such as the external memory model, deal with this issue explicitly.

problem.

$$\Theta\left(\max\left[1, \min\left\{\log_w n, \frac{\log\frac{w}{\log w}}{\log\left(\log\frac{w}{\log w} / \log\frac{\log n}{\log w}\right)}, \log\frac{\log(2^w - n)}{\log w}\right\}\right]\right)$$

From the upper bound perspective, the first branch matches dynamic fusion trees [PT14], the second branch is based on an extension of the techniques from Beame and Fich [BF02], and the last branch is based on an extension of dynamic van Emde Boas trees [vEBKZ77].

Modern Models of Computation The lower bound implies that — in the word RAM model — we cannot support operations in constant time for general n and w. Hence, a natural question is if practical models of computation capturing modern hardware can allow us to overcome the superconstant lower bound. One such model is the *RAM with byte overlap* (RAMBO) by Brodnik et al. [BCF+05]. This model extends the word RAM model by adding a set of special words that share bits; flipping a bit in one word will also affect all the other words that share that bit. Prototypes for this type of model have been built [LMu+99]. Brodnik et al. give a data structure using constant time per operation with $O(2^w/w)$ space (counting both regular words and shared words). More recently, Farzan et al. [FLNS15] introduced the UWRAM model, as described above. They show how to simulate the result of Brodnik et al [BCF+05] at the cost of using a polylogarithmic factor more space; they use $O(w2^w)$ space and support both predecessor queries and updates in worst case constant time.

Our Results Let S be a subset of the universe $U = \{0, \dots, 2^w - 1\}$ and define $n = |S|$. We revisit the predecessor problem on the UWRAM and give a data structure that uses $O(n)$ space, supports predecessor queries on S in constant time, and updates on S in amortized expected constant time. Our result also holds when ultrawords consist only of $w^{1+\epsilon}$ bits for any fixed $\epsilon > 0$. Compared to the previous result of Farzan et al. [FLNS15] we reduce the space from $O(w2^w)$ (i.e., superlinear in $|U|$) to linear in S while supporting all operations in constant time. By restricting ourselves to only $w^{1+\epsilon}$-bit ultraword we limit our reliance on the powerful scattered memory operations by allowing them to access only w^ϵ words in memory in parallel.

A key component in our solution is a new dictionary data structure of independent interest that supports fast parallel lookups on the UWRAM. Let D be a size-n subset of U. We give an $O(n)$ space data structure that supports insertions to and deletions from D in amortized expected constant time. It also supports a constant time parallel query that, when given w (or w^ϵ) elements in an ultraword, returns an ultraword indicating which of the inputs are present in D.

Techniques Our parallel dictionary is based on the dynamic perfect hashing structure of Dietzfelbinger et al. [DKM+94], which is a two-level structure similar to the classic FKS static perfect hashing structure [FKS84]. The first level divides the set D into smaller, disjoint subsets using a hash function, and these subsets are each stored in a separate array — without collisions — using another hash function. We carefully lay out the data structure in memory, show how to evaluate a class of universal hash functions in parallel using the ultraword operations, and use the scattered memory operations to navigate the structure in parallel for each of the queried elements.

For w^2-bit ultrawords, our predecessor structure is based on the x-fast trie by Willard [Wil83], combined with the parallel dictionary structure above. The core idea of the x-fast trie is to store all prefixes of all elements in S in a hash table, using $O(nw)$ space. Given an element x it is then possible to find the longest prefix of x that also occurs in S in $O(\log w) = O(\log\log u)$ time by binary searching over the length of the prefix. Once the longest prefix is found, finding the predecessor is straight forward. We use the parallel dictionary to query all prefixes simultaneously, achieving constant time. This also allows us to store the compact trie of S instead of the uncompacted trie reducing the space to $O(n)$ (since we are checking all possibilities we do not need a monotonic property to decide which direction to search). To extend the result to $w^{1+\epsilon}$-bit ultrawords we run a w^ϵ-ary search instead of a binary search, taking $O(\log_{w^\epsilon} w) = O(1)$ time. In this case we cannot use the compacted trie; we use the top-bottom-type decomposition from Willard [Wil83] to reduce the space from $O(nw)$ to $O(n)$.

1.3 The Complexity of the Co-Occurrence Problem

Problem Definition and History Let S be a length-n string over an alphabet Σ and let Q be a subset of Σ of size $q \geq 2$. Given integers $1 \leq i \leq j \leq n$, the interval $[i, j] = \{i, i+1, \ldots, j\}$ is a *co-occurrence* of Q if the substring $S[i, j]$ contains each character in Q at least once. The *co-occurrence problem* is to preprocess S and Q into a data structure that supports the query

$\mathsf{co}_{S,Q}(w)$: return the number of co-occurrences of Q in S that have length w.

For example, let $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{-}\}$, $Q = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ and

$$S = \underset{1}{\texttt{-}}\ \underset{2}{\texttt{-}}\ \underset{3}{\texttt{-}}\ \underset{4}{\texttt{-}}\ \underset{5}{\texttt{B}}\ \underset{6}{\texttt{C}}\ \underset{7}{\texttt{-}}\ \underset{8}{\texttt{A}}\ \underset{9}{\texttt{C}}\ \underset{10}{\texttt{C}}\ \underset{11}{\texttt{B}}\ \underset{12}{\texttt{-}}\ \underset{13}{\texttt{-}}$$

Then

- $\mathsf{co}_{S,Q}(3) = 0$, because no length-three substring contains all three characters A, B, and C.
- $\mathsf{co}_{S,Q}(4) = 2$, because both $[5, 8]$ and $[8, 11]$ are co-occurrences of Q.
- $\mathsf{co}_{S,Q}(8) = 6$, because all six of the length-eight substrings of S are co-occurrences of Q.

Only data structures using sublinear space are interesting, since we otherwise can precompute $\mathsf{co}_{S,Q}(i)$ for each i. This problem is related to the sliding window streaming model mentioned above; the answer to $\mathsf{co}_{S,Q}(i)$ is equivalent to sliding a window of size i over S, character by character, and counting the number times the window contains all characters in Q. Furthermore, we also aim to construct this data structure using a streaming algorithm that processes S only once.

The co-occurrence problem, introduced by Sobel, Bertram, Ding, Nargesian and Gildea [SBD+21], is a new take on standard data mining problems. For instance, a large amount of work has gone towards related problems such as finding frequent items in streams [DLM02, GDD+03, KSP03, LCK14] and finding frequent sets of items in streams [AH18, CL06, DP13, LL09, LCWC05, MTZ08, YYL+15], i.e., extracting a set Q. In the co-occurrence problem, however, the goal is to extract information about a predetermined set of elements. Analyzing a given query set over all window lengths may reveal domain specific properties of the set. For instance, Sobel et al. motivate the problem by listing potential applications such as training models for natural language processing (short and long co-occurrences of a set of words tend to represent respectively syntactic and semantic information), automatically organizing the memory of a computer program for good cache behaviour (variables that are used close to each other should be near each other in memory), and analyzing DNA sequences (co-occurrences of nucleotides in DNA provide insight into the evolution of viruses). See [SBD+21] for a more detailed discussion of these applications.

Our work is inspired by [SBD+21]. They do not consider fast, individual queries, but instead they give an $O(\sqrt{nq})$ space data structure from which they can determine $\mathsf{co}_{S,Q}(i)$ for each $i = 1, \ldots, n$ in $O(n)$ time. Supporting fast queries is a natural extension to their problem, and we note that their solution can be extended to support individual queries in $O(\log \log n)$ time using the techniques presented below.

We also introduce the following related problem. A co-occurrence $[i, j]$ is *left-minimal* if $[i + 1, j]$ is not a co-occurrence. The *left-minimal co-occurrence problem* is to preprocess S and Q to support

$\mathsf{lmco}_{S,Q}(w)$: return the number of left-minimal co-occurrences of Q in S that have length w.

We use our solution to this simpler problem as a building block for our solution to the co-occurrence problem, showing that the following equality holds

$$\mathsf{co}_{S,Q}(x) = \left(\sum_{i=2}^{x} \mathsf{lmco}_{S,Q}(i) \right) - \max(x - r_1, 0) \tag{1.1}$$

where r_1 is the first index in S such that $S[1, r_1]$ is a co-occurrence. This makes sense intuitively; the sum counts all the positions where the the left-minimal co-occurrence ending at that position has length at most x, and from these positions the subtraction excludes the positions where no length-x co-occurrence may end (for instance, $S[1, r_1]$ is not a length-x co-occurrence if $r_1 < x$). To our knowledge, the left-minimal co-occurrence problem has not been studied before.

Our Results Our two main contributions are as follows. Firstly, we give an upper bound that matches and simplifies the state of the art. Secondly, we provide lower bounds that show that our solution has optimal space, and that our query time is optimal for optimal-space data structures. As in previous work, all our results work on the word RAM model with logarithmic word size.

To do so we use the following parametrization. Define $\delta(i) = \mathsf{lmco}(i) - \mathsf{lmco}(i-1)$ for each $i \in [2, n]$ and let d be the number of indices i such that $\delta(i) \neq 0$ (here we are omitting the subscript on δ, d, and lmco). Our results are as follows.

- We give an $O(d)$ space data structure that supports (left-minimal) co-occurrence queries in $O(\log\log n)$ time. It can be constructed by a single-pass streaming algorithm over S that uses $O(n)$ expected time and $O(d+q)$ space.
- Any data structure supporting (left-minimal) co-occurrence queries needs $\Omega(d)$ space in the worst case, and structures with $d\log^{O(1)} d$ space cannot support queries faster than $\Omega(\log\log n)$.
- We bound $d = O(\sqrt{nq})$ and show that $\Omega(\sqrt{nq})$ *bits* of space is necessary in the worst case for any data structure supporting (left-minimal) co-occurrence queries.

Thus, we prove that our data structure has optimal space and is as fast as possible within optimal space. Compared to the previous work, we match their $O(\sqrt{nq})$ space and $O(\log\log n)$ time solution, and show that this space bound is optimal to within a logarithmic factor. All of our results are based on simple and intuitive combinatorial ideas that simplify the state of the art.

Techniques The key technical insights that lead to our results stem mainly from the structure of δ.

For the upper bound, we store a predecessor structure over the set $\{(i, \mathsf{lmco}(i)) \mid \delta(i) \neq 0\}$, i.e., for each i where $\mathsf{lmco}(i) \neq \mathsf{lmco}(i-1)$. Then we answer left-minimal co-occurrence queries using a single predecessor query. There are linear space predecessor structures that support queries in $O(\log\log |U|)$ time [Wil83], which in this case is $O(\log\log n)$. For co-occurrennce queries, we store an additional piece of information for each entry allowing us to compute Equation 1.1 in constant time. We bound $d = O(\sqrt{nq})$ observing that each non-zero $\delta(i)$ essentially corresponds to a *minimal* co-occurrence (i.e., both left- and right-minimal) of length i, and we bound the cumulative lengths of minimal matches to $O(nq)$. Thus, $d = \omega(\sqrt{nq})$ is not possible because the cumulative length would be at least $1 + 2 + \ldots + d = \Omega(d^2) = \omega(nq)$.

For the lower bounds on space we carefully design lmco instances that encode sets (or sequences) in the δ-function. We can retrieve $\delta(i) = \mathsf{lmco}(i) - \mathsf{lmco}(i-1)$, so the data structure thus represents these sets. If there are L possible sets, the data structure needs to use $\log L$ bits in the worst case in order to distinguish them all. For the lower bounds on query time we reduce from the static predecessor problem to the left-minimal co-occurrence problem and apply lower bounds by Pătraşcu and Thorup [PT07].

1.4 Sliding Window String Indexing in Streams

Problem Definition and History The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. In this paper, we introduce a basic variant of string indexing called the *streaming sliding window string indexing (SSWSI) problem*, where the goal is to support pattern matching queries in the sliding window streaming model, as described in Section 1.1.

Specifically, the string S arrives as a stream, one character at a time, and the goal is to maintain an index over the w most recent characters, i.e., the *window* (this paper consistently uses w as window width as opposed to the word width in the word RAM model). At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and we need to report the occurrences of P within the current window. The goal is to use $O(w)$ space and to minimize the time spent per character from both S and P. We consider two variants of the problem: a *timely* variant where each query must

be answered immediately, and a *delayed* variant where it may be answered at any point within the next δ characters arriving from either stream, for a specified parameter δ.

The SSWSI problem has not been explicitly studied before in our precise formulation, but closely related work on maintaining the suffix tree over a sliding window [FG89,Lar99,Sen05,BJ18,NAIP03] can be extended to solve the timely version. For constant-sized alphabets, the best of these solutions [BJ18] maintains the sliding window suffix tree in constant *amortized* time per character while supporting efficient pattern matching queries. The worst-case time for updates is $\Omega(w)$. The amortization is unavoidable because a new character may cause $\Omega(w)$ changes to the suffix tree.

Another closely related problem is *online string indexing* [AKLL05, Kop12, BI13, Kos94, AN08, KN17, FG05, AFG+14], i.e., incrementally building an index over S as it arrives one character at a time. The best of these solutions update the index in either constant time per character for constant-sized alphabets [KN17] or $O(\log\log n + \log\log|\Sigma|)$ [Kop12] time for a general alphabet Σ. However, they heavily rely on processing S in reverse order, which we cannot do in our model. We could pretend that S is arriving in reverse and reverse the patterns prior to querying, but this would lead to worst-case $\Omega(|P|)$ time per character in the pattern as they are also being streamed. Furthermore, it is not clear how to adapt these results to index a sliding window as opposed to a growing prefix or suffix of the string.

Another line of work shows how to maintain a fully dynamic suffix array under insertions and deletions [AB20, AB21, SLLM10, KK22]. These can be used to solve SSWSI but are more general and lead to polylogarithmically slower bounds than our results while being more complicated.

Our Results For the timely variant, we present a simple $O(w)$ space data structure that spends $O(\log w)$ time per character in both S and each pattern P. This bound holds with high probability in w, that is, it fails to hold only with probability $1/w^d$ for any positive constant d. In the delayed variant, where the answer to queries may be given at any point within the next δ characters to arrive from any stream, the space is $O(w + \delta)$ while the time bound is improved to $O(\log(w/\delta))$ with high probability for both updates and queries. In particular, if $\delta = \epsilon w$ for any constant $\epsilon > 0$, we achieve linear time and optimal constant time processing per character. Compared to previous suffix tree based approaches for indexing a sliding window, we improve the worst-case time bounds per character in the stream from $\Omega(w)$ to $O(\log w)$ with high probability. For both the timely and delayed variants, reporting the occurrences of a pattern incurs an additional constant time per reported occurrence. The result hold on the word RAM with logarithmic word size, and for any alphabet where each character fits into a constant number of machine words.

Techniques The core idea of our data structure is to maintain at most $\log w$ suffix trees[2] that do not overlap and together cover the window. The trees are organized by the *log-structured merge technique* [OCGO96] such that the trees grow exponentially towards the left. We build a new suffix tree over each character that arrives from S and append it to the data structure. Whenever two equal-sized suffix trees are adjacent we "merge" them by constructing a new tree covering them both. This process is deamortized by merging in the background, resulting in $O(\log w)$ time per update in the timely variant (since each element is included in $\log w$ suffix trees before it leaves the window). We use powerful dictionaries with high-probability guarantees to store the edges of the suffix tree (for fast navigation), and a new alphabet reduction hashing scheme to reduce large alphabets to small ranges (this is necessary for linear time construction of suffix trees).

To support queries we query each individual suffix tree, and use different methods for querying across the boundaries between adjacent suffix trees; the exact method depends on the exact model, see the paper for details. This takes $O(\log w)$ time per character as we spend constant time per tree and boundary.

To extend the result to the delayed variant of the problem we store only the $O(\log(w/\delta))$ largest trees and leave a suffix of size $\Theta(\delta)$ of the window uncovered by suffix trees. We answer queries as follows. If $|P| > \delta/4$ we say that P is *long*, and otherwise it is *short*. Long patterns are long enough that we have the time build a suffix tree over the uncovered suffix *at query time*, so we may apply the techniques from the timely variant more or less directly. For short patterns we utilize that they are smaller than the delay

[2] A *suffix tree* is a fundamental data structure for string processing [Wei73]. It is a static data structure that uses linear space and supports pattern matching queries in linear time in the length of the pattern.

to temporarily buffer the queries and later batch process them, at which point we can also afford to build a suffix tree over the uncovered suffix. In both cases, we spend constant time per character for each tree, boundary, and the uncovered suffix, spending $O(\log(w/\delta))$ time in total per character.

1.5 Rank and Select

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

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

The rank-select problem is a fundamental string problem due to its wide applicability, e.g., [BN15, OS07, GMR06, RRS07, PNB17, BCPT15, MN07, FMMN07, BHMS11, BCG+14, NS14, HM10, NN14, GRSV13], references therein, and surveys [Gag16].

A *degenerate string* is a sequence $X = X_1, \dots X_n$ where each X_i is a subset of $[1, \sigma]$. It encodes the set of strings that can be obtained by selecting one character from each X_i, from left to right. 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 $X_1, \dots, X_n$. Degenerate strings have been studied since the 80s [Abr87] and the literature contains papers on problems such as degenerate string comparison [AAB+20], finding string covers for degenerate strings [CIK+17], and pattern matching with degenerate patterns, degenerate texts, or both [Abr87, IMR08].

Alanko, Biagi, Puglisi, and Vuohtoniemi [ABPV23] 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 $X_1, \dots, X_i$ that contain c
- $\mathsf{subset\text{-}select}_X(i, c)$: return the index of the ith set that contains c

Their motivation for studying is problems in pangenomics; specifically, in another work they show how to preprocess a string S such that they can answer *k-mer queries* (i.e., "does this length-k string appear in S?") using $2k$ subset-rank queries [APV23]. Their implementation outperforms the previous state of the art by one to two orders of magnitude, while improving or matching space usage. Their theoretical result [ABPV23] supports both queries in $O(\log \sigma)$ time and uses $2N \log \sigma + 2n_0 + o(N \log \sigma + n_0)$ bits of space[3].

Our Results We introduce the natural parameter N, and reanalyze a number of reductions from subset rank-select to regular rank-select, based on reductions from [APV23]. By plugging in standard rank-select data structures we improve the existing theoretical query times exponentially to $O(\log \log \sigma)$, while improving, or essentially matching, the space bounds. We provide a lower bound showing that any solution to the problem must use $N \log \sigma - o(N \log \sigma)$ bits in the worst case, and show that the reductions match this bound up to lower order terms in many cases. Finally, we provide implementations and compare them to the implementations provided by [APV23, ABPV23] for their results. Our most compact structure matches the space of their most compact structure while answering queries twice as fast. We also provide a data structure utilizing modern vector hardware, which matches the space of the most compact structure, improves query time a factor four to seven, and remains competitive with the remaining fast structures.

Techniques Most of the techniques are straightforward. For the lower bound we use similar techniques to those described in Section 1.3. For the upper bounds we analyze the reductions by carefully considering different cases for n, N, and n_0. For our implementation using vector hardware, one interesting component is the operation `vpternlogq`, which — given three vectors A, B, and C — evaluates *any* three-variable boolean function f bitwise for each of the vectors, i.e., the ith bit of the output is the result of applying f

[3]The space bound they state is slightly different as they do not explicitly use N as a parameter; the bound we state here is the result of our reanalysis of their result.

to the ith bits of A, B, and C. We use this during rank queries to quickly indicate the character that we are performing a rank query for, before counting the number of occurrences using an operation that counts the number of 1-bits.

Chapter 2

Predecessor on the Ultra-Wide Word RAM

Predecessor on the Ultra-Wide Word RAM

Philip Bille*
DTU Compute
phbi@dtu.dk

Inge Li Gørtz*
DTU Compute
inge@dtu.dk

Tord Joakim Stordalen
DTU Compute
tjost@dtu.dk

Abstract

We consider the predecessor problem on the ultra-wide word RAM model of computation, which extends the word RAM model with *ultrawords* consisting of w^2 bits [TAMC, 2015]. The model supports arithmetic and boolean operations on ultrawords, in addition to *scattered* memory operations that access or modify w (potentially non-contiguous) memory addresses simultaneously. The ultra-wide word RAM model captures (and idealizes) modern vector processor architectures. Our main result is a simple, linear space data structure that supports predecessor in constant time and updates in amortized, expected constant time. This improves the space of the previous constant time solution that uses space in the order of the size of the universe. Our result holds even in a weaker model where ultrawords consist of $w^{1+\epsilon}$ bits for any $\epsilon > 0$. It is based on a new implementation of the classic x-fast trie data structure of Willard [Inform. Process. Lett. 17(2), 1983] combined with a new dictionary data structure that supports fast parallel lookups.

2.1 Introduction

Let S be a set of n w-bit integers. The *predecessor problem* is to maintain S under the following operations.

- $\mathsf{predecessor}(x)$: return the largest $y \in S$ such that $y \leq x$.
- $\mathsf{insert}(x)$: add x to S.
- $\mathsf{delete}(x)$: remove x from S.

The predecessor problem is a fundamental and well-studied data structure problem, both from the perspective of upper bounds [PT06, PT14, vEBKZ77, Wil83, FW93, And96, BF02, vEB77, BBV10, BBPV09] and lower bounds [Ajt88, BF02, Mil94, MNSW98, SV08, PT06, PT07]. The problem has many applications, for instance integer sorting [And96, AHNR98, FW93, Han04], string sorting [AFGV97, BFK06, Far97], and string searching [Bel12, BBV10, BEGV18, BGS17, BLR+15]. See Navarro and Rojas-Ledesma [NR20] for a recent survey.

On the word RAM model of computation, the complexity of the problem is well-understood with the following tight upper and lower bound on the time for operations given by Pătraşcu and Thorup [PT14].

$$\Theta\left(\max\left[1, \min\left\{\log_w n, \frac{\log \frac{w}{\log w}}{\log\left(\log \frac{w}{\log w} / \log \frac{\log n}{\log w}\right)}, \log \frac{\log(2^w - n)}{\log w}\right\}\right]\right)$$

From the upper bound perspective, the first branch matches dynamic fusion trees [PT14], the second branch is based on an extension of the techniques from Beame and Fich [BF02], and the last branch is based on an

*Supported by Danish Research Council grant DFF-8021-002498

extension of dynamic van Emde Boas trees [vEBKZ77]. Note that the lower bound implies that we cannot support operations in constant time for general n and w. Hence, a natural question is if practical models of computation capturing modern hardware can allow us to overcome the superconstant lower bound.

One such model is the *RAM with byte overlap* (RAMBO) by Brodnik et al. [BCF+05]. This model extends the word RAM model by adding a set of special words that share bits; flipping a bit in one word will also affect all the other words that share that bit. The precise model is determined by the layout of the shared bits. It is feasible to make hardware based on this model, and prototypes have been built [LMu+99]. In the RAMBO model, Brodnik et al. [BCF+05] gave a predecessor data structure using constant time per operation with $O(2^w/w)$ space (counting both regular words and shared words). They also gave a randomized version of the solution that uses constant time with high probability and reduces the regular space to $O(n)$ (but still needs $\Omega(2^w/w)$ space for the shared words). In both cases, the total space is near-linear in the size of the universe.

More recently, Farzan et al. [FLNS15] introduced the *ultra-wide word RAM model* (UWRAM). The UWRAM extends the word RAM model by adding special *ultrawords* of w^2 bits. The model supports standard boolean and arithmetic operations on ultrawords, as well as *scattered* memory operations that access w words in memory in parallel. The UWRAM model captures (and idealizes) modern vector processing architectures [Rei13, SBB+17, CRDI07] (see Section 2.2 for details of the model). Farzan et al. [FLNS15] showed how to simulate algorithms for the RAMBO model on the UWRAM at the cost of increasing the space by a polylogarithmic factor. Simulating the above RAMBO solution for the predecessor problem, they gave a solution to the predecessor problem on the UWRAM using worst case constant time for all operations and $O(w2^w)$ space.

2.1.1 Our Results

We revisit the predecessor problem on the UWRAM and show the following main result.

Theorem 1. *Given a set of n w-bit integers, we can construct an $O(n)$ space data structure on a UWRAM that supports* predecessor *in constant time and* insert *and* delete *in amortized expected constant time. The result holds even when ultrawords consist of $w^{1+\epsilon}$ bits for any fixed $\epsilon > 0$.*

Compared to the previous result of Farzan et al. [FLNS15], Theorem 1 significantly reduces the space from $O(w2^w)$ to linear while maintaining constant time for operations (note that query time is worst-case, while updates are amortized expected). Furthermore, our result works in a weaker model were ultrawords consist of only $w^{1+\epsilon}$ bits for any arbitrarily small $\epsilon > 0$. In this restricted model we limit our reliance on the powerful scattered memory operations by allowing them to access only w^ϵ words in memory in parallel.

A key component in our solution is a new dictionary data structure of independent interest that supports fast parallel lookups on the UWRAM. We define the problem as follows. Recall that an ultraword X consists of w^2 (or $w^{1+\epsilon}$) bits. We view X as divided into w (or w^ϵ) words of w consecutive bits each, numbered from right to left starting from 0. The ith word in X is denoted $X\langle i\rangle$ (we discuss the model in detail in Section 2.2). Given a set S of n w-bit integers, the *w^ϵ-parallel dictionary problem* is to maintain S under the following operations.

- pMember(X): return an ultraword I where $I\langle i\rangle = 1$ if $X\langle i\rangle \in S$ and $I\langle i\rangle = 0$ otherwise.
- insert(x): Add x to S.
- delete(x): Remove x from S.

Thus, pMember takes an ultraword X of w^ϵ integers and returns an ultraword encoding which of these integers are in S. To the best of our knowledge, the w^ϵ-parallel dictionary problem has not been studied before. We show the following result.

Theorem 2. *Given a set of n w-bit integers on a UWRAM with $w^{1+\epsilon}$-bit ultrawords for any fixed $\epsilon > 0$, we can construct an $O(n + w^\epsilon)$-space data structure that supports* pMember *queries in worst case constant time and* insert *and* delete *in amortized expected constant time.*

Note that the queries are worst-case constant time, while the updates are amortized expected constant time. The time bounds of Theorem 2 thus match the well-known dynamic perfect hashing structure of Dietzfelbinger et al. [DKM+94] (which is also the basis of our solution), except that the queries are parallel. The space is linear except for the additive w^ϵ term, which is needed even for storing the input to the pMember query.

In our data structures we only need to store a constant number of ultrawords during the computation. This is important since modern vector processor architectures only have a limited number of ultraword registers.

An extended abstract of this paper appeared at the *18th Scandinavian Symposium and Workshops on Algorithm Theory* [BGS22b]. The current paper improves that result by extending it to ultrawords consisting of only $w^{1+\epsilon}$ bits, and also provides complete proofs for all the claims.

2.1.2 Techniques

Our results are achieved by novel and efficient parallel implementations of well-known sequential data structures.

Our parallel dictionary structure of Theorem 2 is based on the dynamic perfect hashing structure of Dietzfelbinger et al. [DKM+94]. This is a two-level data structure similar to the classic static perfect hashing structure of Fredman et al. [FKS84]. At the first level, a universal hash function partitions the input into smaller subsets, each of which is then resolved at the second level using another universal hash function mapping the elements into sufficiently large tables. The structure supports (sequential) membership queries in worst-case constant time by evaluating the hash functions and navigating the structure accordingly. Updates are supported in amortized expected constant time by carefully rebuilding and rehashing the structure during execution. At any point in time the structure never uses more than $O(n)$ space. We show how to parallelize the evaluation of a universal hash function (the simple and practically efficient *multiply-shift* hash function). Then, using the scattered memory access operations, we show how to access the corresponding entries in the structure in parallel. Our technique requires only small changes to the structure of Dietzfelbinger et al. [DKM+94] and we can directly apply their update operations to our solution. Thus, we are able to parallelize the worst-case constant time sequential membership query while maintaining the amortized expected constant update time bound of Dietzfelbinger et al. [DKM+94], leading to the bounds of Theorem 2.

We first show Theorem 1 for the simpler case $\epsilon = 1$ that corresponds to the original UWRAM model by [FLNS15]. Our data structure is based on the *x-fast trie* of Willard [Wil83] combined with our parallel dictionary structure of Theorem 2. The x-fast trie consists of the trie T of the binary representation of the input set. Also, at each level i, the structure stores a dictionary containing the length-i prefixes of the input set. In total, this uses $O(nw)$ space. The x-fast trie supports predecessor queries in $O(\log w)$ time by binary searching the levels (with the help of the dictionaries) to find the longest common prefix of the query and the input set. Though not designed for it, we can implement updates on the x-fast trie in $O(w)$ time by directly updating each level of the dictionary accordingly. Our new predecessor structure, which we call the *xtra-fast trie*, instead stores the compact trie of the binary representation of the input set (i.e., the trie where paths of nodes with a single child are merged into a single edge). We store a dictionary representing the prefixes (similar to the x-fast trie) using our parallel dictionary structure of Theorem 2, but now only for the branching nodes in the compact trie. This reduces the space to $O(n)$. To support predecessor queries for an integer x, we generate all w prefixes of x and apply a parallel membership query on these in the dictionary. We show how to identify the longest match in parallel which in turn allows us to identify the predecessor. In total this takes worst-case constant time for the predecessor query. To handle updates, we show how to modify the trie efficiently using scattered memory access operations and a constant number of dictionary updates, leading to the expected amortized constant time bound of Theorem 1.

We generalize our result for Theorem 1 to arbitrary $\epsilon > 0$ as follows. The main challenge is that pMember now supports only w^ϵ member queries in parallel, so we cannot search for all prefixes of x simultaneously. Instead, we adapt ideas from the y-fast trie by Willard [Wil83] to our $x$tra-fast trie. The y-fast trie works as follows. Partition the input set S into $O(n/w)$ sets $S_1, \ldots, S_t$ where each S_i consists of w consecutive values

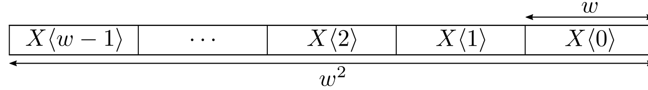


Figure 2.1: The layout of an ultraword X

from S, i.e., where $\max(S_i) < \min(S_{i+1})$ for each i. Build an x-fast trie over the set $S' = \{\max(S_i) \mid i = 1, \ldots, t-1\}$ — which takes $O(n)$ space since $|S'| = O(n/w)$ — and a balanced binary search tree over each S_i. To determine $\mathsf{predecessor}(x)$, do a predecessor query in the x-fast trie to determine the set S_i containing the predecessor of x and do a predecessor query in S_i, both of which takes $O(\log w)$ time. Insertions are supported by instead inserting x in S_i. If S_i subsequently becomes too large (e.g., larger than $2w$), split S_i into two and add an additional element to S' in the x-fast trie. This takes $O(w)$ time, which is constant when amortized over the $\Omega(w)$ insertions necessary for S_i to grow too large. Deletions are supported similarly. In our data structure we use dynamic fusion trees by Pătraşcu and Thorup [PT14] for each S_i, which solves the predecessor problem on sets of size $w^{O(1)}$ in linear space and constant time per operation. We build an *uncompacted* $x$tra-fast trie over S', i.e. the $x$tra-fast trie that also includes non-branching nodes. To support fast queries and updates for an integer x, we use the scattered memory operations to simulate a w^ϵ-way search (as opposed to a binary search) to find the longest common prefix between x and S'. This eliminates a factor $1/w^\epsilon$ of the remaining possibilities per round, leading to a running time of $O(\log_{w^\epsilon} w) = O(1/\epsilon)$, i.e., constant for any fixed ϵ.

2.1.3 Outline

In Section 2.2 we describe the UWRAM model of computation and some useful procedures. In Sections 2.3 and 2.4 we show how to do parallel hash function evaluation and w^ϵ-parallel dictionaries, proving Theorem 2. Finally, in Section 2.5 we prove Theorem 1 for $\epsilon = 1$, which we generalize to arbitrary $\epsilon > 0$ in Section 2.6.

2.2 The Ultra-Wide Word RAM Model

The *word RAM* model of computation [Hag98] consists of an unbounded memory of w-bit words and a standard instruction set including arithmetic, boolean, and bitwise operations (denoted '&', '|' and '∼' for *and*, *or* and *not*) and shifts (denoted '≫' and '≪') such as those available in standard programming languages (e.g., C). Furthermore, we assume for simplicity that there is a constant-time operation for finding the index of the leftmost 1-bit in a word. This operation is supported by most modern computers; otherwise there is a constant-time implementation using standard bitwise and arithmetic operations [FW93]. We make the standard assumption that we can store a pointer into the input in a single word and hence $w \geq \log n$, where n is the size of the input, and for simplicity we assume that w is even. We denote the address of x in memory as $\mathsf{addr}(x)$, and the address of an array is the address of its first index. The time complexity of a word RAM algorithm is the number of instructions and the space is the number of words stored by the algorithm.

The *ultra-wide word RAM* (UWRAM) model of computation [FLNS15] extends the word RAM model with special *ultrawords* of w^2 bits (in Section 2.6 we consider the case where ultrawords have $w^{1+\epsilon}$ bits for any fixed $\epsilon > 0$). As in [FLNS15], we distinguish between the *restricted UWRAM* that supports a minimal set of instructions on ultrawords consisting of addition, subtraction, shifts, and bitwise boolean operations, and the *multiplication UWRAM* that additionally supports multiplications. We extend the notation for bitwise operations and shifts to ultrawords. The UWRAM (both restricted and multiplication) also supports contiguous and scattered memory access operations, as described below. The time complexity is the number of instructions (on standard words or ultrawords) and the space complexity is the number of words used by the algorithms, where each ultraword is counted as w words. The UWRAM model captures (and idealizes) modern vector processing architectures [Rei13, SBB+17, CRDI07]. See also Farzan et al. [FLNS15] for a detailed discussion of the applicability of the UWRAM model.

2.2.1 Instructions and Componentwise Operations

Recall that an ultraword consists of w^2 bits. We often view an ultraword X as divided into w words of w consecutive bits each, which we call the *components* of X. We number the components in X from right-to-left starting from 0 and use the notation $X\langle i\rangle$ to denote the ith word in X (see Fig. 2.1). We will also use the notation $X = \langle x_{w-1}, \dots, x_0\rangle$, denoting that $X\langle i\rangle = x_i$.

We define a number of useful componentwise operations on ultrawords that we will need for our algorithms in the following. Let X and Y be ultrawords. The *componentwise addition* of X and Y, denoted $X + Y$, is the ultraword Z such that $Z\langle i\rangle = X\langle i\rangle + Y\langle i\rangle \mod 2^w$. We define *componentwise subtraction*, denoted $X - Y$, and *componentwise multiplication*, denoted XY, similarly. The *componentwise comparison* of X and Y is the ultraword Z such that $Z\langle i\rangle = 1$ if $X\langle i\rangle < Y\langle i\rangle$ and 0 otherwise. Given another ultraword I where each component is either 0 or 1, we define the *componentwise blend* of X, Y, and I to be the ultraword Z such that $Z\langle i\rangle = X\langle i\rangle$ if $I\langle i\rangle = 0$ and $Z\langle i\rangle = Y\langle i\rangle$ if $I\langle i\rangle = 1$.

Except for componentwise multiplication, all of the above componentwise operations can be implemented in constant time on the restricted UWRAM using standard word-level parallelism techniques [Hag98,BGS22a] (see Appendix 2.A for details on blend). For our purposes, we will need componentwise multiplication as an instruction (for evaluating hash functions in parallel) and thus we include this in the instruction set of the UWRAM. This is the UWRAM model that we will use throughout the rest of the paper. Note that all of the componentwise operations are widely supported directly in modern vector processing architectures. For instance, a componentwise multiplication (e.g., the `vpmullw` operation) is defined in Intel's AVX2 vector extension [Cor11].

We will need componentwise operations on components that are small constant multiples of w. In particular, we will need a *$2w$-bit componentwise multiplication* that multiplies $w/2$ components of w bits and returns the $w/2$ resulting components of $2w$ bits. Specifically, let $X = \langle 0, x_{w-2}, \dots, 0, x_2, 0, x_0\rangle$ and $Y = \langle 0, y_{w-2}, \dots, 0, y_2, 0, y_0\rangle$, i.e., X and Y store $w/2$ components aligned at the even positions. The $2w$-bit componentwise multiplication is the ultraword $Z = \langle z^+_{w-2}, z^-_{w-2}, \dots, z^+_2, z^-_2, z^+_0, z^-_0\rangle$ where z^+_i and z^-_i is the leftmost and rightmost w bits, respectively, of the $2w$-bit product of x_i and y_i. We can implement $2w$-bit componentwise multiplication using standard techniques in constant time on the UWRAM. See Appendix 2.A for details.

Finally, the UWRAM model supports the compress operation that, given X, returns the word that results from concatenating the rightmost bit of each component of X. We do not need the corresponding inverse spread operation, defined by Farzan et al. [FLNS15].

2.2.2 Memory Access

The UWRAM supports standard memory access operations that read or write a single word or a sequence of w contiguous words. More interestingly, the UWRAM also supports *scattered* access operations that access w memory locations (not necessarily contiguous) in parallel. Given an ultraword A containing w memory addresses, a *scattered read* loads the contents of the addresses into an ultraword X, such that $X\langle i\rangle$ contains the contents of memory location $A\langle i\rangle$. Given ultrawords X and A a *scattered write* sets the contents of memory location $A\langle i\rangle$ to be $X\langle i\rangle$. Scattered memory accesses captures the memory model used in IBM's *Cell* architecture [CRDI07]. They also appear (e.g., `vpgatherdd`) in Intel's AVX2 vector extension [Cor11]. Scattered memory access operations were also proposed by Larsen and Pagh [LP12] in the context of the I/O model of computation. Note that while the addresses for scattered writes must be distinct, we can read simultaneously from the same address. We can use this to efficiently copy x into all w components of an ultraword X. To do so, create the ultraword $\langle 0, \dots, 0\rangle$ by left-shifting any ultraword by w^2 bits, write x to address 0, and do a scattered read on $\langle 0, \dots, 0\rangle$. We say that we *load* x into X.

2.3 Computing Multiply-Shift in Parallel

We show how to efficiently compute a universal hash function in parallel. The *multiply-shift* hashing scheme is a standard and practically efficient family of universal hash functions due to Dietzfelbinger et al. [DHKP97].

For some integer $1 \le c \le w$, define the class $H_c = \{h_a \mid 0 < a < 2^w \text{ and } a \text{ is odd}\}$ of hash functions where $h_a(x) = (ax \mod 2^w) \gg (w-c)$. Each function in H_c maps from w-bit to c-bit integers. The class H_c is *universal* in the sense that for any $x \neq y$ and for $h_a \in H_c$ selected uniformly at random, it holds that $P[h_a(x) = h_a(y)] \le 2/2^c$.

We will show how to evaluate w such functions in constant time. Given $X\langle i\rangle = x_i$, $A\langle i\rangle = a_i$ and $C\langle i\rangle = 2^{c_i}$ where $h_i(x) = (a_i x \mod 2^w) \gg (w - c_i)$ the goal is to compute $H\langle i\rangle = h_i(x_i)$. To do so we first evaluate the functions in two rounds of $w/2$ functions each, and then combine the results.

Step 1: Evaluate the hash function on the even indices. We construct an ultraword H_{even} containing all the values of $h_i(x_i)$ at all even indices i. First construct the ultrawords

$$C' = \langle 0, 2^{c_{w-2}}, 0, 2^{c_{w-4}}, \ldots, 0, 2^{c_0}\rangle$$
$$T' = \langle 0, p_{w-2}, 0, p_{w-4} \ldots, 0, p_0\rangle.$$

where p_i is the product $a_i x_i \mod 2^w$.

To do so, we do componentwise multiplication of C with the constant $M = \langle 0, 1, \ldots, 0, 1\rangle$ and componentwise multiplications of A, X, and M. Then, we do a $2w$-bit multiplication of C' and T' and right shift the result by w. This produces the ultraword

$$H_{\text{even}} = \langle \star, p_{w-2} \gg (w - c_{w-2}), \star, p_{w-4} \gg (w - c_{w-4}), \ldots, \star, p_0 \gg (w - c_0)\rangle.$$

Note that $p_i \gg (w - c_i) = (a_i x_i \mod 2^w) \gg (w - c_i) = h_i(x_i)$. Thus, all even indices in H_{even} store the resulting hash values of the integers at the even indices in the input. We will not need the values in the odd indices (resulting from the $2w$-bit multiplication and the right shift) and therefore these are marked with a wildcard symbol $\star$.

Step 2: Evaluate the hash function on the odd indices. Symmetrically, we now construct the ultraword H_{odd} containing $h_i(x_i)$ at all odd indices i. To do so, repeat step 1 and modify the shifting to align the computation for the odd indices. More precisely, right shift X, C and A by w and repeat step 1, then left shift the result by w to align the results back to the odd positions. This produces the ultraword

$$H_{\text{odd}} = \langle p_{w-1} \gg (w - c_{w-1}), \star, p_{w-3} \gg (w - c_{w-3})\star, \ldots, p_1 \gg (w - c_1), \star\rangle$$

Step 3: Combine the results. Finally, we combine the results by blending H_{even} and H_{odd} using $I = \langle 1, \ldots, 1\rangle - M$, producing the ultraword H of the even indices of H_{even} and the odd indices of H_{odd}.

This takes constant time since componentwise multiplication, $2w$-bit multiplication, shifting, blending, loading 1 into $\langle 1, \ldots, 1\rangle$, and componentwise subtraction all run in constant time. Hence, we can evaluate each case of $w/2$ hash functions in constant time and combine the results in constant time. In summary, we have the following result.

Lemma 1. *Given $X\langle i\rangle = x_i$, $A\langle i\rangle = a_i$, $C\langle i\rangle = 2^{c_i}$, and the constant $M = \langle 0, 1, \ldots, 0, 1\rangle$ we can evaluate each of the w multiply-shift hash functions $h_i(x) = (a_i x \mod 2^w) \gg (w - c_i)$ by computing the ultraword $H = \langle h_{w-1}(x_{w-1}), \ldots, h_0(x_0)\rangle$ in constant time on a UWRAM.*

2.4 The w^ϵ-Parallel Dictionary

We now show how to construct the w^ϵ-parallel dictionary of Theorem 2. Throughout the section we assume that $\epsilon = 1$, but the result generalizes to any $\epsilon > 0$ in a straightforward manner. Our data structure is based on a dictionary by Dietzfelbinger et al. that implements a dynamic perfect hashing strategy [DKM+94]. Their dictionary already supports insert and delete in amortized expected constant time. Furthermore, it

supports sequential member queries (i.e. "is $x \in S$") in worst case constant time. We will show that we can use scattered memory operations to run w member queries simultaneously, thus implementing $\mathsf{pMember}$ in constant time.

2.4.1 Dynamic Perfect Hashing

In this section we briefly describe the contents of the data structure of Dietzfelbinger et al. [DKM+94]. Note that we use the multiply-shift hashing scheme, while they use another class of universal hash functions. Multiply-shift satisfies all the necessary constraints and the analysis from [DKM+94] still works. It does however incur a multiplicative, constant space overhead for our arrays since the range of a multiply-shift function is a power of two.

The main idea of the data structure is as follows. Let S be a set of w-bit integers. Choose $h \in H_c$ and partition S into $2^c = \Theta(n)$ sets $S_0, \ldots, S_{2^c-1}$ where $S_i = \{x \mid x \in S \text{ and } h(x) = i\}$. Each set S_i is stored in a separate array using a hash function $h_i(x) = (a_i x \mod 2^w) \gg (w - c_i)$. Dietzfelbinger et al. show how to implement the operations insert and delete such that they maintain that h_i has no collisions on S_i. The values of c and each c_i vary as as the size of S changes. However, in general 2^c is smaller than, e.g., $3|S|$ and 2^{c_i} is approximately $2\binom{|S_i|}{2}$. See details in Dietzfelbinger et al. [DKM+94].

The data structure consists of the following.

- For each S_i, store an array T_i of size 2^{c_i}. For each $x \in S_i$ let $T_i[h_i(x)] = x$, i.e. the position that x hashes to stores x. If there is no $x \in S_i$ that hashes to j, then $T_i[j] = 2^{w-1}$ if $j = 0$ and $T_i[j] = 0$ otherwise. We claim that $h_i(0)$ is always zero and $h_i(2^{w-1})$ is never zero; it follows that $x \in S_i$ if and only if $T_i[h_i(x)] = x$ since each unused entry $T_i[j]$ stores a value whose hash cannot be j. We have that $h_i(2^{w-1})$ is not zero because

  $$h_i(2^{w-1}) \;=\; (a_i 2^{w-1} \mod 2^w) \gg (w - c_i) \;=\; 2^{w-1} \gg (w - c_i) \;\geq\; 1.$$

  The second step follows since a_i is odd; then $a_i 2^{w-1} = 2^{w-1} + (a_i - 1)2^{w-1}$, and the latter term is 0 modulo 2^w since $a_i - 1$ is even. The last step follows because $c_i \geq 1$.

- An array T of size 2^c. At index $T[i]$ we store the 5-tuple $(\mathsf{addr}(T_i), 2^{c_i}, a_i, \star, \star)$ where $\star$ are book-keeping values used by insert and delete. Note that 2^{c_i} and a_i encode h_i.

- The integers a and 2^c representing the top-level hash function $h(x) = (ax \mod 2^w) \gg (w - c)$, as well as $\mathsf{addr}(T)$.

It follows from this construction that $x \in S$ if and only if $T_i[h_i(x)] = x$ where $i = h(x)$. Dietzfelbinger et al. show that the data structure uses linear space, that member runs in worst-case constant time, and that insert and delete run in amortized expected constant time [DKM+94].

Extending the Data Structure. We extend this data structure by storing the constant $M = \langle 0, 1, \ldots, 0, 1, 0, 1 \rangle$ from Section 2.3 used to evaluate multiply-shift functions in parallel. This increases the space of the data structure to $O(n + w)$. Note that linear space in w is needed even to store the input to a $\mathsf{pMember}$ query.

2.4.2 Parallel Queries

In this section, we begin by describing a single member query, before we show how to run w copies of the member query in parallel to support $\mathsf{pMember}$. We compute $\mathsf{member}(x)$ as follows.

1. Using a and 2^c, compute $j = h(x)$.

2. Let $q = \mathsf{addr}(T) + 5j = \mathsf{addr}(T[j])$ (recall that each index in T stores five words). Read the values stored at q, $q+1$ and $q+2$ to get respectively $\mathsf{addr}(T_j)$, 2^{c_j} and a_j, the first three words stored at $T[j]$. Compute $k = h_j(x)$.

3. Check whether the value stored at $\mathsf{addr}(T_j) + k = \mathsf{addr}(T_j[k])$ is equal to x.

The parallel algorithm runs this algorithm for all w inputs simultaneously. Given $X = \langle x_{w-1}, \ldots, x_0 \rangle$ we implement $\mathsf{pMember}(X)$ as follows. Each of the steps below executes the corresponding step above in parallel for each of the w inputs.

Step 1: Evaluate the top-level hash function. Load the two ultrawords $A = \langle a, \ldots, a \rangle$ and $C = \langle 2^c, \ldots, 2^c \rangle$. Compute the ultraword $J = \langle h(x_{w-1}), \ldots, h(x_0) \rangle$ using the multiply-shift algorithm of Lemma 1.

Step 2: Evaluate each of the second-level hash functions. Load $F = \langle 5, \ldots, 5 \rangle$ and $P = \langle \mathsf{addr}(T), \ldots, \mathsf{addr}(T) \rangle$. Compute $Q = P + FJ$. Then $Q\langle i \rangle = \mathsf{addr}(T) + 5J\langle i \rangle = \mathsf{addr}(T[J\langle i \rangle])$. Do scattered reads of Q, $Q + \langle 1, \ldots, 1 \rangle$, and $Q + \langle 2, \ldots, 2 \rangle$ to produce the ultrawords P', C', and A'. We have that

$$
\begin{aligned}
P' &= \langle \mathsf{addr}(T_{J\langle w-1 \rangle}), \ldots, \mathsf{addr}(T_{J\langle 0 \rangle}) \rangle \\
C' &= \langle 2^{c_{J\langle w-1 \rangle}}, \ldots, 2^{c_{J\langle 0 \rangle}} \rangle \\
A' &= \langle a_{J\langle w-1 \rangle}, \ldots, a_{J\langle 0 \rangle} \rangle
\end{aligned}
$$

Compute the ultraword $K = \langle h_{J\langle w-1 \rangle}(x_{w-1}), \ldots, h_{J\langle 0 \rangle}(x_0) \rangle$ using the multiply-shift algorithm of Lemma 1.

Step 3: Check whether the inputs are present in the dictionary. Do a scattered read of $P' + K$ and name the result R. Then $R\langle i \rangle = T_j[h_j(x_i)]$ where $j = h(x_i)$. Return the result I of componentwise equality between X and R. That is

$$
I\langle i \rangle = \begin{cases} 1 & \text{if } X\langle i \rangle = R\langle i \rangle \\ 0 & \text{otherwise} \end{cases}
$$

Evaluating the hash functions in steps 1 and 2 takes constant time according to Lemma 1. The remaining operations are scattered reads, loads and componentwise operations, all of which run in constant time. Since there is only a constant number of operations, pMember runs in constant time. This concludes the proof of Theorem 2.

Note that both the algorithm for parallel hashing and the dictionary generalizes to the case with $w^{1+\epsilon}$-bit ultrawords and w^ϵ inputs in a straight forward manner. In this case, the space is $O(n + w^\epsilon)$ since the ultraword constants use only w^ϵ space.

2.4.3 Satellite Data

Suppose we associate some value $\mathsf{data}(x)$ with each $x \in S$. We extend the data structure to support the following operation, where $X = \langle x_{w-1}, \ldots, x_0 \rangle$ as above.

- $\mathsf{pRetrieve}(X)$: returns a pair (I, D) where I is the result of $\mathsf{pMember}(X)$ and

$$
D\langle i \rangle = \begin{cases} \mathsf{addr}(\mathsf{data}(x_i)) & \text{if } I\langle i \rangle = 1 \text{, i.e if } x_i \in S \\ \text{undefined} & \text{otherwise} \end{cases}
$$

We return $\mathsf{addr}(\mathsf{data}(x))$ instead of $\mathsf{data}(x)$ since the data would not fit into an ultraword if $\mathsf{data}(x)$ requires more than one word to store.

We extend the data structure as follows to support pRetrieve. Store two words for each index in T_i. For each $x \in S_i$, the first word in $T_i[h_i(x)]$ stores x and the second stores $\mathsf{addr}(\mathsf{data}(x))$. The remaining entries store either 0 or 2^{w-1}, as above.

To do the retrieval, first compute $I = \mathsf{pMember}(X)$. However, in step 3, multiply K by $\langle 2, \ldots, 2 \rangle$ before the scattered read since each index in T_i now stores two words. Also, add $\langle 1, \ldots, 1 \rangle$ to $P' + \langle 2, \ldots 2 \rangle K$ and do a scattered read to compute the ultraword D. The space of the data structure remains $O(n + w)$ (assuming that $\mathsf{data}(x)$ uses constant space), and pRetrieve runs in constant time.

2.5 The $x$tra-fast Trie

In this section we prove Theorem 1 for the special case where $\epsilon = 1$, i.e. where ultrawords consist of w^2 bits. We generalize our result to arbitrary $\epsilon > 0$ in Section 2.6. Our data structure, the $x$tra-fast trie, supports predecessor in worst case constant time and insert and delete in amortized expected constant time. In our description we assume that we have keys of $w-1$ bits each and we give a solution that uses $O(n+w)$ space. At the end of this section we will reduce the space to $O(n)$ and extend the solution to w-bit keys, proving Theorem 1 for $\epsilon = 1$.

2.5.1 Data Structure

Consider the compacted trie T over the binary representation of the elements in S. For each node $v \in T$ define $\mathsf{str}(v)$ to be the bitstring encoded by the path from the root to v in T. Also let $\mathsf{min}(v)$ and $\mathsf{max}(v)$ be the smallest and largest leaves in the subtree of v, respectively. By $\mathsf{min}(v)$ and $\mathsf{max}(v)$ we refer both to a leaf and to the value the leaf represents.

For each edge $(u, v) \in T$, let $\mathsf{label}(u, v)$ be $\mathsf{str}(u)$ followed by the first bit on the edge (u, v). Define $\mathsf{key}(u, v)$ to be $\mathsf{label}(u, v)$ followed by a single 1-bit and $w - |\mathsf{label}(u, v)| - 1$ zeroes. Note that $|\mathsf{key}(u, v)| = w$ and that the keys of two distinct edges in T always differ. See Fig. 2.2 for an example.

We define the *exit edge* for an integer x to be the edge in T where the match of x ends. In other words, it is the edge $(u, v) \in T$ such that $\mathsf{label}(u, v)$ is a prefix of x and $|\mathsf{label}(u, v)|$ is maximum. See Fig. 2.2 for an example. It is possible that x has no exit edge if the root has fewer than two children.

Our data structure consists of the following:

- A sorted, doubly linked list L of the leaves of T, i.e., the elements of S.
- A dictionary D supporting parallel queries using Theorem 2. For each edge $(u, v) \in T$ we store an entry in D with the key $\mathsf{key}(u, v)$ and $\mathsf{data}(u, v) = (\mathsf{addr}(\mathsf{min}(v)), \mathsf{addr}(\mathsf{max}(v)))$. Here, $\mathsf{addr}(\mathsf{min}(v))$ and $\mathsf{addr}(\mathsf{max}(v))$ are the addresses to the corresponding elements in L, and we denote the addresses to $\mathsf{min}(v)$ and $\mathsf{max}(v)$ as the *min-* and *max-pointer* of (u, v).
- The two ultraword constants M' and H described in the next section.

Storing L and the ultraword constants takes $O(n+w)$ space combined. Since T is compacted there are $O(n)$ entries in D, so by Theorem 2 the dictionary also uses $O(n + w)$ space.

2.5.2 Predecessor Queries

The main idea of the predecessor query for x is to first find the exit edge of x by simultaneously searching for all prefixes of x in D. Then we use the min- and max-pointer of the exit edge to find the predecessor of x. If x has no exit edge, then the root does not have an outgoing edge matching the leftmost bit of x. If the leftmost bit of x is 1, the predecessor of x is the largest leaf in the left subtree of the root, and otherwise x has no predecessor. Assuming that x has an exit edge, the procedure has three steps.

Step 1: Compute all prefixes of x. Let $b_{w-2}b_{w-3}\cdots b_0$ be the binary representation of x of length $w-1$. We compute the ultraword

$$\overline{X} = \langle b_{w-2}b_{w-3}\cdots b_0 1\ ,\ b_{w-2}b_{w-3}\cdots b_1 10\ ,\ \ldots\ ,\ 10\cdots 0\rangle.$$

That is, $\overline{X}\langle i\rangle$ contains the prefix of x of length i followed by a 1-bit and $w - i - 1$ zeroes. Thus, for any edge $(u, v) \in T$ such that $\mathsf{label}(u, v)$ is the length-i prefix of x, we have $\overline{X}\langle i\rangle = \mathsf{key}(u, v)$. We compute $\overline{X}$ as follows.

Let M' be the constant such that $M'\langle i\rangle$ consists of i consecutive 1-bits followed by $w - i$ consecutive 0-bits. Let H be the constant where the $(i+1)$th leftmost bit in $H\langle i\rangle$ is 1 and the remaining bits are zeroes. First load x into X such that $X = \langle x, x, \ldots, x\rangle$. Then compute $\overline{X} = (X \ \&\ M') \mid H$.

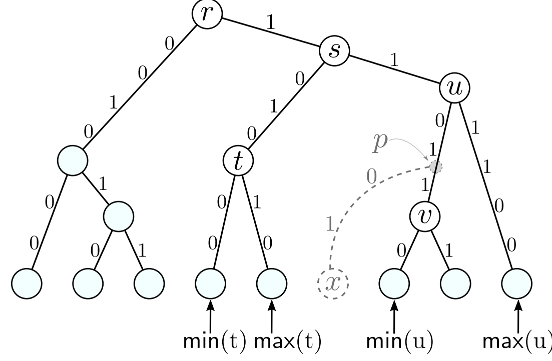


Figure 2.2: An $x$tra-fast trie for $S = \{001000, 001010, 001011, 101000, 101010, 110110, 110111, 111100\}$. The dashed edge and nodes illustrate how the trie would change if $x = 110101$ were inserted. The exit edge for x is (u, v) since we match the bitstring 1101 but do not match the next 1 on (u, v). Similarly, the exit edge for 100100 is (s, t). We have that $\mathsf{key}(u, v) = \mathsf{label}(u, v)\underline{1000} = 110\underline{1000}$ where the underlined part is what we append to the labels to disambiguate the keys. Similarly, $\mathsf{key}(r, s) = 1\underline{100000}$ and $\mathsf{key}(s, t) = 10\underline{10000}$. The dictionary entry of (s, u) has $\mathsf{key}(s, u) = 11\underline{10000}$, and the min- and max-pointer of (s, u) are $\mathsf{addr}(\mathsf{min}(u))$ and $\mathsf{addr}(\mathsf{max}(u))$. Similarly, the min-pointer of (r, s) is to $\mathsf{min}(s) = \mathsf{min}(t)$ and the max-pointer is to $\mathsf{max}(s) = \mathsf{max}(u)$. Note that if we insert x we would have to update the min-pointer of (s, u), since $x < \mathsf{min}(v)$. However, the min-pointer of (r, s) remains unchanged since $\mathsf{min}(t) < x$

Step 2: Find the exit edge (u, v) of x. First do $(I, P) = \mathsf{pRetrieve}(\overline{X})$ on D. Then compute $c = \mathsf{compress}(I)$ such that the ith rightmost bit in c is 1 if $I\langle i\rangle = 1$ and zero otherwise. Note that x has no exit edge if $c = 0$. Find the index k of the leftmost bit in c that is 1. Then $\overline{X}\langle k\rangle = \mathsf{key}(u, v)$ where (u, v) is the exit edge of x. Furthermore, the values stored at the addresses $P\langle k\rangle$ and $P\langle k\rangle + 1$ are the min- and max-pointers of (u, v), respectively.

Step 3: Find the predecessor of x. Use the min- and max-pointer of (u, v) found in step 2 to retrieve $\mathsf{min}(v)$ and $\mathsf{max}(v)$. If $x \geq \mathsf{max}(v)$ then return $\mathsf{max}(v)$; otherwise return the element immediately left of $\mathsf{min}(v)$ in L. Note that there might not be an element immediately left of $\mathsf{min}(v)$ if x is smaller than than everything in S, in which case x has no predecessor.

Since we search for all prefixes of x and take the edge corresponding to the longest prefix found, we find the exit edge (u, v) of x. If $x \in S$, then $x = v = \mathsf{max}(v)$ and we correctly return that x is the predecessor of itself. If $x \notin S$ then the path to where x would have been located if it were in T branches off (u, v) either to the left (if $x < \mathsf{min}(v)$) or right (if $x > \mathsf{max}(v)$). In the first case, $\mathsf{predecessor}(x)$ is the element located immediately left of $\mathsf{min}(v)$ in T, and in the second case $\mathsf{predecessor}(x)$ is $\mathsf{max}(v)$.

By Theorem 2 the parallel dictionary query in step 2 takes worst case constant time. The remaining operations are standard operations available in the model, so the procedure runs in constant time.

2.5.3 Insertions

The main idea of the insertion procedure is as follows. Since T is compacted, inserting a new leaf x will cause only a constant number of edges to be inserted and removed, so we can make these changes sequentially. Furthermore, some of the at most $w - 1$ edges on the path from the root to x might have their min- or max-pointers changed, and we will update these edges in parallel.

Consider inserting $x = 110101$ in the trie in Fig. 2.2. When x is inserted we add a new leaf for x, as well as a new node p at the location where the path to x branches off the exit edge (u, v) of x. This removes the edge (u, v), but adds the three new edges (u, p), (p, x) and (p, v). Furthermore, we must update the min-pointer of (s, u), because $\mathsf{min}(v)$ was replaced by x as the smallest leaf under u. On the other hand,

we do not update the min-pointer of (r, s) because $\mathsf{min}(t)$ is smaller than x. Note that we do not explicitly store internal nodes and therefore do not add p anywhere in the data structure.

We now describe the insertion procedure. First we note that if x does not have an exit edge it is because the root does not have an outgoing edge which shares the same leftmost bit as x. This case is easily solved by adding an edge from the root to the new leaf x and adding x to either the start or end of L. We will now assume that x has an exit edge, and also that x branches off its exit edge to the left; the other case is symmetric.

Step 1: Find the predecessor of x. Do a predecessor query as described in Section 2.5.2, which determines

- The predecessor of x in L.
- The exit edge (u, v) of x, $\mathsf{label}(u, v)$ and $\mathsf{data}(u, v) = (\mathsf{addr}(\mathsf{min}(v)), \mathsf{addr}(\mathsf{max}(v)))$.
- The result (I, P) of $\mathsf{pRetrieve}(\,\overline{X}\,)$ on D.

Step 2: Insert x in L. Insert x immediately to the right of its predecessor in L.

Step 3: Update edges. Let p be the new node that is added on (u, v) when we insert x. We insert (u, p), (p, x) and (p, v) and remove (u, v) from D. We find the labels of the three edges to insert as follows. We have that $\mathsf{label}(u, p) = \mathsf{label}(u, v)$ since (u, p) is the edge (u, v) shortened by adding the node p and since only the first character of the edge affects the label. By definition, $\mathsf{label}(p, x)$ and $\mathsf{label}(p, v)$ consist of $\mathsf{str}(p)$ with a zero and a one appended, respectively. We compute $\mathsf{str}(p)$ by finding the longest common prefix $\hat{p}$ of x and $\mathsf{min}(v)$. To do so, do bitwise XOR between x and $\mathsf{min}(v)$ and find the index k of the leftmost bit that is 1 in the result. Now k indicates the leftmost bit where x and $\mathsf{min}(v)$ differ. To extract the longest common prefix compute $\hat{p} = x \;\&\; \sim((1 \ll (k + 1)) - 1)$. Given the labels we can easily construct the keys for the edges.

We now construct the satellite data for the edges. Both the min- and max-pointer for (p, x) are $\mathsf{addr}(x)$ since x is a leaf. For (p, v) they are $\mathsf{addr}(\mathsf{min}(v))$ and $\mathsf{addr}(\mathsf{max}(v))$, which were determined during the predecessor query. Finally, the min-pointer for (u, p) is $\mathsf{addr}(x)$ and the max-pointer is $\mathsf{addr}(\mathsf{max}(v))$.

Step 4: Update min-pointers. We update the min-pointers for the edges on the path from the root to u that are incorrect after inserting x. Note that inserting x cannot invalidate any max-pointers since we assumed that x branched off its exit edge to the left. The edges that must be updated are exactly those that have a min-pointer to $\mathsf{min}(v)$, since x has replaced $\mathsf{min}(v)$ as the smallest leaf under u.

Consider the result (I, P) from the $\mathsf{pRetrieve}$ query. We begin by setting $I\langle k'\rangle = 0$ for the index k' corresponding to the exit edge (u, v) of x (we know k' from the predecessor query). The indices in I that now contain 1 indicate the edges on the path from the root to u.

Next we identify the edges that needs to be updated by creating I' where $I'\langle i\rangle = 1$ if and only if both $I\langle i\rangle = 1$ and what is stored at address $P\langle i\rangle$ is the address of $\mathsf{min}(v)$. To do so, first do a scattered read of P and store the result in M. Now M contains $\mathsf{addr}(\mathsf{min}(b))$ for each edge (a, b) on the path to u.[1] Note the value of $P\langle i\rangle$ is arbitrary if $I\langle i\rangle = 0$, i.e. if no edge has the length-i prefix of x as its label. Load $\mathsf{addr}(\mathsf{min}(v))$ into the ultraword V. Let E be the result of componentwise equality between M and V. Then $E\langle i\rangle = 1$ if and only if what is stored at address $P\langle i\rangle$ is $\mathsf{addr}(\mathsf{min}(v))$. Finally compute $I' = I \;\&\; E$.

Now we use P and I' to update the incorrect min-pointers. First, load the address of the node for x into U. Then compute B by blending M (the result of the scattered read of P) and U conditioned on I' such that

$$B\langle i\rangle = \begin{cases} M\langle i\rangle & \text{if } I'\langle i\rangle = 0 \quad \text{(i.e. the value already at the address } P\langle i\rangle\text{)} \\ U\langle i\rangle & \text{if } I'\langle i\rangle = 1 \quad \text{(i.e. the address of } x\text{)} \end{cases}$$

[1] If x branched off to the right of its exit edge, we would do a scattered read of $P + \langle 1, \dots, 1\rangle$ to load the max-pointers instead of min-pointers.

Finally, do a scattered write of B to the addresses in P. Hence, what is stored at the address $P\langle i\rangle$ remains the same if $I'\langle i\rangle = 0$ and is replaced by the address of x otherwise.

The predecessor query in step 1 takes constant time. The operations in step 2 and step 4 are all standard RAM or UWRAM operations. The dictionary updates in step 3 run in amortized expected constant time by Theorem 2. Since the rest of step 3 consists of standard operations, the running time for insertions is amortized expected constant.

2.5.4 Deletions

The deletion procedure is essentially the inverse of the insertion procedure. We assume that x is the left child of its parent p; the other case is symmetric.

Step 1: Find x. Do a predecessor query for x. Since $x \in S$, the predecessor of x is itself. This determines

- The position of x in L.
- The exit edge (p, x) for x, along with $\mathsf{label}(p, x)$. Since $x \in S$, this edge must end in the leaf for x.
- The result (I, P) of $\mathsf{pRetrieve}(\overline{X})$ on D.

Step 2: Update min-pointers. If p is the root (i.e. if $|\mathsf{label}(p, x)| = 1$) we remove the edge (p, x) from D and remove x from L which completes the deletion of x. Otherwise p is an internal node and must have another child which we denote by v. Consider the edges on the path to p. Any min-pointer to x should be replaced by the address of $\mathsf{min}(v)$, since $\mathsf{min}(v)$ is the successor of x and also in the subtree of all of these edges. We find $\mathsf{min}(v)$ in the node immediately right of x in L. As we did in Step 4 for insertions, we compute an ultraword I'' where $I''\langle i\rangle = 1$ if and only if both $I\langle i\rangle = 1$ and if what is stored at the address $P\langle i\rangle$ is the address of x. Using I'', P, and the same algorithm as in Step 4 we replace every min-pointer to x by the address of $\min(v)$.

Step 3: Delete edges. We delete (p, x) and (p, v) from D. Determine $\mathsf{label}(p, v)$ by flipping the last bit in $\mathsf{label}(p, x)$. Using the labels we easily find the keys. Note that we do not explicitly delete the edge (u, p) or insert the edge (u, v). These two edges share the same key, and the min-pointer of (u, p) was changed to the address of $\mathsf{min}(v)$ in step 2.

Step 4: Update L. Remove x from L.

Steps 1, 2 and 4 all take constant time (see Sections 2.5.2 and 2.5.3). The two deletions in step 3 take amortized constant time according to Theorem 2. The remainder of step 3 takes constant time, so deletions run in amortized expected constant time.

2.5.5 Reducing to Linear Space and Supporting w-bit Keys

Here, we reduce the space to $O(n)$ and show how to support w-bit keys, concluding the proof of Theorem 1.

The $O(w)$ term in the space bound above is due to the w^ϵ-parallel dictionary D and $O(1)$ ultraword constants. To avoid this when $n = o(w)$, we will initially support $\mathsf{predecessor}$, insert and delete using the *dynamic fusion tree* by Pătraşcu and Thorup [PT14] (based on the fusion tree by Fredman and Willard [FW93]), which uses linear space and supports all three operations in constant time for sets of size $w^{O(1)}$. Simultaneously, we build the ultraword constants we need over the course of $\Theta(w)$ insertions, maintaining linear space. When $n \geq w$, the constants have been built and we move all elements into the trie. If at any point $n \leq w/2$, we move all elements from the trie into a fusion tree and remove the trie and the ultraword constants, leaving us with linear space and $\Theta(w)$ insert operations in which to rebuild the constants. Updates still run in amortized expected constant time since we always do $\Omega(w)$ updates before we move $O(w)$ elements.

To extend the solution to work with w-bit keys, we partition the input set S into S_0 and S_1 where $S_i = \{s \mid s \in S$ and the leftmost bit of s is $i\}$, and store an $x$tra-fast trie for each set. Suppose the leftmost bit of an integer x is i. An insert, delete or predecessor operation on x is performed on the data structure for S_i. Additionally, if $i = 1$ and the predecessor query on S_1 returns that x has no predecessor, we return the largest element in S_0, or report that x has no predecessor if S_0 is empty.

2.6 The $x$tra-fast Trie With Smaller Ultrawords

In this section we show how to match the bounds of Theorem 1 when ultrawords consist of only $w^{1+\epsilon}$ bits (i.e. w^ϵ components) for any fixed $\epsilon > 0$. The model is otherwise exactly as described in Section 2.2. For simplicity, we assume that w^ϵ is an integer to avoid writing $\lfloor w^\epsilon \rfloor$ throughout the section.

As mentioned, our data structure is based on the y-fast trie by Dan Willard [Wil83], which we briefly sketched in Section 2.1.2. As before, we describe an $O(n + w)$ space data structure for keys of $w - 1$ bits, which can be improved to $O(n)$ space and w-bit keys using the same tricks from Section 2.5.5.

2.6.1 Data Structure

Partition the input set S into $t = \lceil n/w \rceil$ sets $S_1, \ldots, S_t$ such that $|S_i| = w$ and $\max(S_i) < min(S_{i+1})$ for each $i < t$. The last set S_t might be smaller than w. Define $S' = \{\max(S_i) \mid i = 1, \ldots, t-1\}$. The data structure consists of

- The *uncompacted* $x$tra-fast trie T over S', i.e., the $x$tra-fast trie where we also include non-branching nodes. Its definition is identical to that in Section 2.5.1, defined over the *trie* of S' instead of the *compacted trie.*
- A dynamic fusion tree [PT14] over each S_i. This word-RAM data structure by Pătraşcu and Thorup supports insert, delete, and predecessor in constant time and linear space on sets of size $w^{O(1)}$.
- A w^ϵ-ary search tree B over the set $W = \{1, \ldots, w-1\}$ defined as follows. Divide W into $w^\epsilon + 1$ roughly equally-sized consecutive subranges $W_1, \ldots, W_{w^\epsilon+1}$ (their sizes might differ by 1). Recursively construct search trees over each subrange. The root v of B stores a pointer to each subtree and the keys $\min(W_2), \min(W_3), \ldots, \min(W_{w^\epsilon+1})$ in sorted order in an array $\mathcal{I}_v$. The recursion terminates in a leaf when there are less than w^ϵ elements in the range. Each leaf v stores the elements it represents in sorted order in the array $\mathcal{I}_v$, as well as the number of elements $|\mathcal{I}_v|$ that is stores.
- The w^2-bit ultraword constants M' and H from Section 2.5.2 that we used to compute the keys for the parallel dictionary queries. Each constant is stored in a length-w array.

Throughout the operation of the data structure, we maintain that $|S_i| = \Omega(w)$ for each i. Therefore, $|S'| = O(n/w)$. The uncompacted $x$tra-fast trie uses $O(|S'|w + w^\epsilon) = O(n + w^\epsilon)$ space since each root-to-leaf path has w edges. Here the w^ϵ comes from the w^ϵ-parallel dictionary used in the trie. The dynamic fusion trees use linear space in total. The tree B uses $O(w)$ space since the leaves use $O(w)$ space and each internal node is branching. The constants M' and H also use $O(w)$ space, so the total space consumption is $O(n+w)$.

2.6.2 Predecessor Queries

Since each ultraword contains only w^ϵ components, we cannot simultaneously search for all prefixes of a query x to determine the longest prefix present in the tree. Instead, we will use B to do a w^ϵ-ary search. Once we have found the longest prefix, we find the predecessor of x in S' as in Section 2.5.2, and then determine the predecessor of x in the corresponding S_i using the fusion tree.

To determine the longest prefix of x that occurs in S' we do the following. Let v be the root node of B. Read $\mathcal{I}_v$ into an ultraword and use the result to do a scattered read relative to the constants M' and H, denoting the resulting ultrawords by M'_v and H_v. Load x into the ultraword X. As in Section 2.5.2 we

compute $\overline{X}_v = (X \;\&\; M'_v) \mid H_v$ to determine the keys of the prefixes of x of length $\mathcal{I}_v[0], \ldots, \mathcal{I}_v[w^\epsilon - 1]$. Do a parallel query for $\overline{X}_v$ in the parallel dictionary; the result I_v indicates which prefixes of x occur in S'. Compress I_v and find the leftmost 1-bit to determine which subrange contains the length of the longest prefix of x, and recurse in the corresponding subtree. Note that if v is a leaf, $|\mathcal{I}_v| < w^\epsilon$ is possible. To avoid false positives in the dictionary query we load $\mathcal{I}_v$ into the $|\mathcal{I}_v|$ rightmost components of an ultraword and zero out the remaining components by doing bitwise & with $(1 \ll |\mathcal{I}_v|w) - 1$; this does not lead to false positives in the parallel member query since 0 is not a valid key for any edge in S' (see the definition of keys in Section 2.5.1).

Once the longest prefix of x has been found we find the predecessor of x in S' as in Section 2.5.2, and find the predecessor of x in the corresponding S_i in constant time using the fusion tree. The time it takes to determine the longest prefix is the time it takes to search in B. For each node we do a constant number of scattered reads, bitwise operations, and a parallel member query, all of which takes constant time. The height of B is $O(\log_{w^\epsilon} w) = O(1/\epsilon) = O(1)$ since the branching factor of all internal nodes is $w^\epsilon + 1$, concluding the proof.

2.6.3 Insertions and Deletions

To insert a new element x we first determine which set S_i to add x to using a predecessor query. We then insert x in S_i in constant time using the dynamic fusion tree. If S_i becomes too large (e.g., $2w$) we split it by deleting and reinserting half of the elements in another dynamic fusion tree. We also add new separator element to S' by manually inserting every new edge; this takes expected $O(w)$ time in total since each edge can be added to the w^ϵ-parallel dictionary in amortized expected constant time. The time it takes to support insertions is amortized expected constant since there are $\Omega(w)$ updates between splits. Deletions are supported similarly, merging adjacent fusion trees if they become too small.

2.7 Conclusions and Open Problems

We have studied the predecessor problem on the UWRAM model of computation. We have given a linear space data structure that supports predecessor queries in worst case constant time and updates in amortized expected constant time, even when ultrawords consist of only $w^{1+\epsilon}$ bits for any fixed $\epsilon > 0$.

Furthermore, we have shown how to implement a w^ϵ-parallel dictionary on the UWRAM. The dictionary supports w (or w^ϵ) simultaneous membership queries in worst case constant time and individual updates in amortized expected constant time.

We wonder if it is possible to achieve constant time with high probability for all operations in the predecessor problem. The limiting factor for our solution is the time for updates in the w^ϵ-parallel dictionary. There are dictionaries that achieve constant time with high probability for all operations in the word RAM model, e.g. [DadH90]. However, such dictionaries seem to require hash functions that are difficult to evaluate in parallel on the UWRAM. For instance, [DadH90] uses the modulo operator, for which we cannot see an obvious way to make a component-wise version.

Acknowledgments We would like to thank the anonymous reviewers for their comments, which improved the presentation of the paper. In particular, we would like to thank the reviewer who suggested that it might be possible to strengthen the result by restricting the model to $w^{1+\epsilon}$-bit ultrawords.

2.A Blend and $2w$-bit Multiplication

Supporting Blend

Given the ultrawords X, Y and I where each component of I is either 0 or 1, we define the *componentwise blend* of X, Y, and I to be the ultraword Z such that $Z\langle i\rangle = X\langle i\rangle$ if $I\langle i\rangle = 0$ and $Y\langle i\rangle$ if $I\langle i\rangle = 1$. To compute the blend in constant time we do as follows. Compute $I' = \langle 0, \ldots, 0\rangle - I$; then $I'\langle i\rangle$ contains only

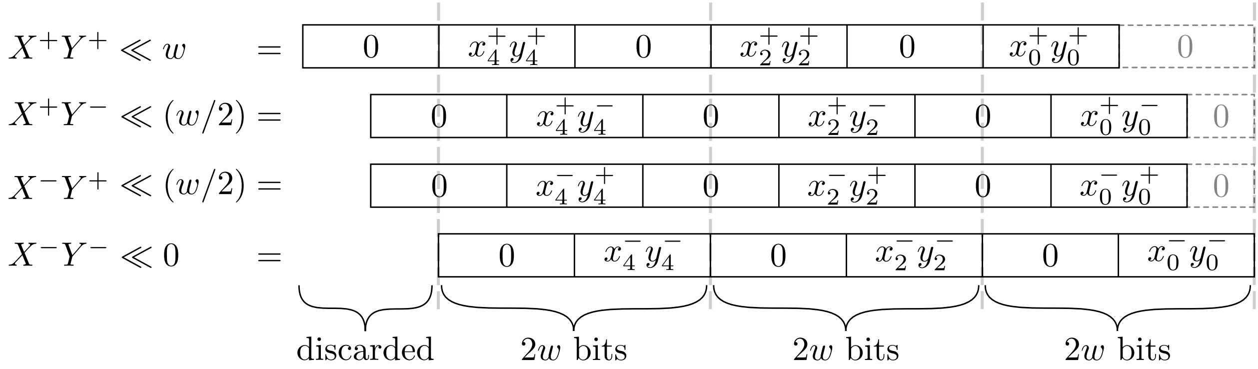


Figure 2.3: Illustrates step 3 of $2w$-bit multiplication. Each of the products X^+Y^+, X^+Y^-, X^-Y^+ and X^-Y^- are left-shifted by respectively w, $w/2$, $w/2$ and 0 by shifting in zeroes from the right. Then they are added together using componentwise addition for $2w$-bit components. Since what we sum up in a $2w$-bit component adds up to the product of two w-bit integers, we only need $2w$ bits to store the result. Hence the addition will not overflow

1-bits if $I\langle i\rangle = 1$ and only 0-bits otherwise, since $0 - 1 \mod 2^w = 2^w - 1$. Then the blend of X and Y can be computed by $(X \ \& \ \sim I') \ | \ (Y \ \& \ I')$.

Supporting $2w$-Bit Componentwise Multiplication

We show how to implement $2w$-bit componentwise multiplication in constant time. Let x^+ and x^- denote the leftmost and rightmost half of the binary representation of x, respectively. Then, if x is a $2k$-bit integer we have that $x = x^+2^k + x^-$ where $x^+ = x/2^k$ and $x^- = x \mod 2^k$. Given $X = \langle 0, x_{w-2}, \ldots, 0, x_2, 0, x_0\rangle$ and $Y = \langle 0, y_{w-2}, \ldots, 0, y_2, 0, y_0\rangle$, recall that the $2w$-bit componentwise multiplication of X and Y is the ultraword $Z = \langle z^+_{w-2}, z^-_{w-2}, \ldots, z^+_2, z^-_2, z^+_0, z^-_0\rangle$ where z_i is the $2w$-bit product of x_i and y_i.

The main idea for computing Z is to use the identity

$$\begin{aligned} xy &= (x^+2^{w/2} + x^-)(y^+2^{w/2} + y^-) \\ &= x^+y^+2^w + (x^+y^- + x^-y^+)2^{w/2} + x^-y^- \end{aligned} \tag{2.1}$$

where x and y are w-bit integers. We simulate this in parallel as follows.

Step 1: Compute x_i^+, x_i^-, y_i^+ and y_i^- for all even i. We first construct X^+ and X^- such that $X^+\langle i\rangle = x_i^+$ and $X^-\langle i\rangle = x_i^-$ for even i and zero otherwise, and similarly for Y. Compute the integer $m = 2^{w/2} - 1$ which consists of $w/2$ zeroes followed by $w/2$ ones. Load m into M. Compute $X^- = X \ \& \ M$ and $X^+ = (X \gg w/2) \ \& \ M$. Compute Y^+ and Y^- in the same way.

Step 2: Compute the products of the $w/2$-bit integers. Use componentwise multiplication to compute each of the ultrawords X^+Y^+, X^+Y^-, X^-Y^+ and X^-Y^-. Since each component of X^+, X^-, Y^+ and Y^- is a $(w/2)$-bit integer, no overflow occurs. The odd components still store 0.

Step 3: Align and add the products. Align the products by left-shifting them the amount specified in Equation 2.1, i.e.

$$X^+Y^+ \ll w \qquad X^+Y^- \ll w/2 \qquad X^-Y^+ \ll w/2 \qquad X^-Y^- \ll 0$$

Add the aligned ultrawords using componentwise addition for $2w$-bit components (see e.g. Hagerup [Hag98]) and return the result. See Fig. 2.3 for an illustration. Since the sum of the terms added together in a $2w$-bit component exactly correspond to the multiplication of two w-bit integers, the addition will not overflow.

Bitwise &, left- and right-shifts, componentwise multiplication and componentwise additions for arbitrary component sizes all run in constant time. Each step uses a constant number of these operations, so the procedure runs in constant time.

Chapter 3

The Complexity of the Co-Occurrence Problem

The Complexity of the Co-Occurrence Problem

Philip Bille* DTU Compute phbi@dtu.dk

Inge Li Gørtz* DTU Compute inge@dtu.dk

Tord Joakim Stordalen DTU Compute tjost@dtu.dk

Abstract

Let S be a string of length n over an alphabet Σ and let Q be a subset of Σ of size $q \geq 2$. The *co-occurrence problem* is to construct a compact data structure that supports the following query: given an integer w return the number of length-w substrings of S that contain each character of Q at least once. This is a natural string problem with applications to, e.g., data mining, natural language processing, and DNA analysis. The state of the art is an $O(\sqrt{nq})$ space data structure that — with some minor additions — supports queries in $O(\log\log n)$ time [CPM 2021].

Our contributions are as follows. Firstly, we analyze the problem in terms of a new, natural parameter d, giving a simple data structure that uses $O(d)$ space and supports queries in $O(\log\log n)$ time. The preprocessing algorithm does a single pass over S, runs in expected $O(n)$ time, and uses $O(d+q)$ space in addition to the input. Furthermore, we show that $O(d)$ space is optimal and that $O(\log\log n)$-time queries are optimal given optimal space. Secondly, we bound $d = O(\sqrt{nq})$, giving clean bounds in terms of n and q that match the state of the art. Furthermore, we prove that $\Omega(\sqrt{nq})$ bits of space is necessary in the worst case, meaning that the $O(\sqrt{nq})$ upper bound is tight to within polylogarithmic factors. All of our results are based on simple and intuitive combinatorial ideas that simplify the state of the art.

3.1 Introduction

We consider the *co-occurrence problem* which is defined as follows. Let S be a string of length n over an alphabet Σ and let Q be a subset of Σ of size $q \geq 2$. For two integers i and j where $1 \leq i \leq j \leq n$, let $[i,j]$ denote the discrete interval $\{i, i+1, \ldots, j\}$, and let $S[i,j]$ denote the substring of S starting at $S[i]$ and ending at $S[j]$. The interval $[i,j]$ is a *co-occurrence* of Q in S if $S[i,j]$ contains each character in Q at least once. The goal is to preprocess S and Q into a data structure that supports the query

- $\texttt{co}_{S,Q}(w)$: return the number of co-occurrences of Q in S that have length w, i.e., the number of length-w substrings of S that contain each character in Q at least once.

For example, let $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{-}\}$, $Q = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ and

$$S = \underset{1}{\texttt{-}}\ \underset{2}{\texttt{-}}\ \underset{3}{\texttt{-}}\ \underset{4}{\texttt{-}}\ \underset{5}{\texttt{B}}\ \underset{6}{\texttt{C}}\ \underset{7}{\texttt{-}}\ \underset{8}{\texttt{A}}\ \underset{9}{\texttt{C}}\ \underset{10}{\texttt{C}}\ \underset{11}{\texttt{B}}\ \underset{12}{\texttt{-}}\ \underset{13}{\texttt{-}}\ .$$

Then

- $\texttt{co}_{S,Q}(3) = 0$, because no length-three substring contains all three characters A, B, and C.
- $\texttt{co}_{S,Q}(4) = 2$, because both $[5,8]$ and $[8,11]$ are co-occurrences of Q.
- $\texttt{co}_{S,Q}(8) = 6$, because all six of the length-eight substrings of S are co-occurrences of Q.

*Supported by Danish Research Council grant DFF-8021-002498

Note that only sublinear-space data structures are interesting. With linear space we can simply precompute the answer to $\mathsf{co}_{S,Q}(i)$ for each $i \in [0, n]$ and support queries in constant time.

This is a natural string problem with applications to, e.g., data mining, and a large amount of work has gone towards related problems such as finding frequent items in streams [DLM02, GDD+03, KSP03, LCK14] and finding frequent sets of items in streams [AH18, CL06, DP13, LL09, LCWC05, MTZ08, YYL+15]. Furthermore, it is similar to certain string problems, such as episode matching [DFG+97] where the goal is to determine all the substrings of S that occur a certain number of times within a given distance from each other. Whereas previous work is mostly concerned with identifying frequent patterns either in the whole string or in a sliding window of fixed length, Sobel, Bertram, Ding, Nargesian and Gildea [SBD+21] introduced the problem of studying a given pattern across all window lengths (i.e., determining $\mathsf{co}_{S,Q}(i)$ for all i). They motivate the problem by listing potential applications such as training models for natural language processing (short and long co-occurrences of a set of words tend to represent respectively syntactic and semantic information), automatically organizing the memory of a computer program for good cache behaviour (variables that are used close to each other should be near each other in memory), and analyzing DNA sequences (co-occurrences of nucleotides in DNA provide insight into the evolution of viruses). See [SBD+21] for a more detailed discussion of these applications.

Our work is inspired by [SBD+21]. They do not consider fast, individual queries, but instead they give an $O(\sqrt{nq})$ space data structure from which they can determine $\mathsf{co}_{S,Q}(i)$ for each $i = 1, \dots, n$ in $O(n)$ time. Supporting fast queries is a natural extension to their problem, and we note that their solution can be extended to support individual queries in $O(\log \log n)$ time using the techniques presented below.

A key component of our result is a solution to the following simplified problem. A co-occurrence $[i, j]$ is *left-minimal* if $[i + 1, j]$ is not a co-occurrence. The *left-minimal co-occurrence problem* is to preprocess S and Q into a data structure that supports the query

- $\mathsf{lmco}_{S,Q}(w)$: return the number of left-minimal co-occurrences of Q in S that have length w.

We first solve this more restricted problem, and then we solve the co-occurrence problem by a reduction to the left-minimal co-occurrence problem. To our knowledge this problem has not been studied before.

3.1.1 Our Results

Our two main contributions are as follows. Firstly, we give an upper bound that matches and simplifies the state of the art. Secondly, we provide lower bounds that show that our solution has optimal space, and that our query time is optimal for optimal-space data structures. As in previous work, all our results work on the word RAM model with logarithmic word size.

To do so we use the following parametrization. Let $\delta_{S,Q}$ be the difference encoding of the sequence $\mathsf{lmco}_{S,Q}(1), \dots, \mathsf{lmco}_{S,Q}(n)$. That is, $\delta_{S,Q}(i) = \mathsf{lmco}_{S,Q}(i) - \mathsf{lmco}_{S,Q}(i-1)$ for each $i \in [2, n]$ (note that $\mathsf{lmco}(1) = 0$ since $|Q| \geq 2$). Let $Z_{S,Q} = \{i \in [2, n] \mid \delta(i) \neq 0\}$ and let $d_{S,Q} = |Z_{S,Q}|$. For the remainder of the paper we will omit the subscript on lmco, co, Z, and d whenever S and Q are clear from the context. Note that d is a parameter of the problem since it is determined exclusively by the input S and Q. We prove the following theorem.

Theorem 3. *Let S be a string of length n over an alphabet Σ, let Q be a subset of Σ of size $q \geq 2$, and let d be defined as above.*

(a) There is an $O(d)$ space data structure that supports both $\mathsf{lmco}_{S,Q}$- and $\mathsf{co}_{S,Q}$-queries in $O(\log \log n)$ time. The preprocessing algorithm does a single pass over S, runs in expected $O(n)$ time and uses $O(d + q)$ space in addition to the input.

(b) Any data structure supporting either $\mathsf{lmco}_{S,Q}$- or $\mathsf{co}_{S,Q}$-queries needs $\Omega(d)$ space in the worst case, and any $d \log^{O(1)} d$ space data structure cannot support queries faster than $\Omega(\log \log n)$ time.

(c) The parameter d is bounded by $O(\sqrt{nq})$, and any data structure supporting either $\mathsf{lmco}_{S,Q}$- or $\mathsf{co}_{S,Q}$-queries needs $\Omega(\sqrt{nq})$ bits *of space in the worst case.*

Theorem 3(a) and 3(b) together prove that our data structure has optimal space, and that with optimal space we cannot hope to support queries faster than $O(\log\log n)$ time. In comparison to the state of the art by Sobel et al. [SBD+21], Theorem 3(c) proves that we match their $O(\sqrt{nq})$ space and $O(\log\log n)$ time solution, and also that the $O(\sqrt{nq})$ space bound is tight to within polylogarithmic factors. All of our results are based on simple and intuitive combinatorial ideas that simplify the state of the art.

Given a set X of m integers from a universe U, the *static predecessor problem* is to represent X such that we can efficiently answer the query $\mathsf{predecessor}(x) = \max\{y \in X \mid y \leq x\}$. Tight bounds by Pătraşcu and Thorup [PT07] imply that $O(\log\log|U|)$-time queries are optimal with $m\log^{O(1)} m$ space when $|U| = m^c$ for any constant $c > 1$. The lower bound on query time in Theorem 3(b) follows from the following theorem, which in turn follows from a reduction from the predecessor problem to the (left-minimal) co-occurrence problem.

Theorem 4. *Let $X \subseteq \{2,\ldots,u\}$ for some u and let $|X| = m$. Let n, q, and d be the parameters of the (left-minimal) co-occurrence problem as above. Given a data structure that supports* lmco- *or* co-*queries in $f_t(n,q,d)$ time using $f_s(n,q,d)$ space, we obtain a data structure that supports predecessor queries on X in $O(f_t(2u^2,2,8m))$ time using $O(f_s(2u^2,2,8m))$ space.*

In particular, if $f_s(n,q,d) = d\log^{O(1)} d$ then we obtain an $m\log^{O(1)} m$-space predecessor data structure on X. If also $u = m^c$, then it follows from the lower bound on predecessor queries that $f_t(2u^2,2,8m) = \Omega(\log\log u)$, which in turn implies that $f_t(n,q,d) = \Omega(\log\log n)$, proving the lower bound in Theorem 3(b).

The preprocessing algorithm and the proof of Theorem 4 can be found in Appendices 3.A and 3.B, respectively.

3.1.2 Techniques

The key technical insights that lead to our results stem mainly from the structure of δ.

To achieve the upper bound for lmco-queries we use the following very simple data structure. By definition, $\mathsf{lmco}(w) = \sum_{i=2}^{w} \delta(i)$. Furthermore, by the definition of Z it follows that for any $w \in [2,n]$ we have that $\mathsf{lmco}(w) = \mathsf{lmco}(w_p)$ where w_p is the predecessor of w in Z. Our data structure is a predecessor structure over the set of key-value pairs $\{(i, \mathsf{lmco}(i)) \mid i \in Z\}$ and answers lmco-queries with a single predecessor query. There are linear space predecessor structures that support queries in $O(\log\log|U|)$ time [Wil83]. Here the universe U is $[2,n]$ so we match the $O(d)$ space and $O(\log\log n)$ time bound in Theorem 3(a).

Furthermore, we prove the $O(\sqrt{nq})$ upper bound on space by bounding $d = O(\sqrt{nq})$ using the following idea. In essence, each $\delta(z)$ for $z \in Z$ corresponds to some length-z *minimal co-occurrence*, which is a co-occurrence $[i,j]$ such that neither $[i+1,j]$ nor $[i,j-1]$ are co-occurrences (see below for the full details on δ). We bound the cumulative length of all the minimal co-occurrences to be $O(nq)$; then there are at most $d = O(\sqrt{nq})$ distinct lengths of minimal co-occurrences since $d = \omega(\sqrt{nq})$ implies that the cumulative length of the minimal co-occurrences is at least $1 + \ldots + d = \Omega(d^2) = \omega(nq)$.

To also support co-queries and complete the upper bound we give a straight-forward reduction from the co-occurrence problem to the left-minimal co-occurrence problem. We show that by extending the above data structure to also store $\sum_{i=2}^{z} \mathsf{lmco}(i)$ for each $z \in Z$, we can support co-queries with the same bounds as for lmco-queries.

On the lower bounds side, we give all the lower bounds for the left-minimal co-occurrence problem and show that they extend to the co-occurrence problem. To prove the lower bounds we exploit that we can carefully design lmco-instances that result in a particular difference encoding δ by including minimal co-occurrences of certain lengths and spacing. Our lower bounds on space in Theorem 3(b) and 3(c) are the results of encoding a given permutation or set in δ, respectively.

Finally, as mentioned above, we prove Theorem 4 (and, by extension, the lower bound on query time in Theorem 3(b)) by encoding a given instance of the static predecessor problem in an lmco-instance such that the predecessor of an element x equals $\mathsf{lmco}(x)$.

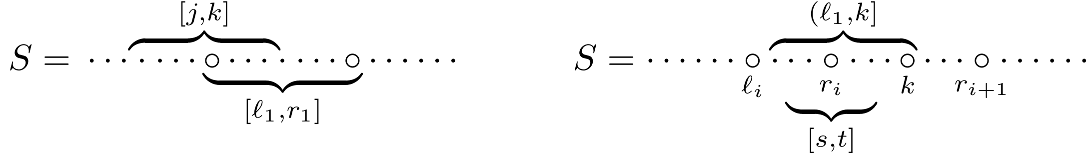


Figure 3.1: *Left (Lemma 2(a)):* Any left-minimal co-occurrence $[j,k]$ must contain a minimal co-occurrence ending at or before k. If $k < r_1$ this contradicts that r_1 is the smallest endpoint of a minimal co-occurrence. *Right (Lemma 2(b)):* $[\ell_i,k]$ is a co-occurrence because it contains $[\ell_i,r_i]$. However, $[\ell_i+1,k]$ is not a co-occurrence; if it were it would contain a minimal co-occurrence $[s,t]$ that ends between r_i and k, leading to a contradiction since $r_i < t < r_{i+1}$.

3.2 The Left-Minimal Co-Occurrence Problem

3.2.1 Main Idea

Let $[i,j] \subseteq [1,n]$ be a co-occurrence of Q in S. Recall that then each character from Q occurs in $S[i,j]$ and that $[i,j]$ is left-minimal if $[i+1,j]$ is not a co-occurrence. The goal is to preprocess S and Q to support the query $\mathsf{lmco}(w)$, which returns the number of left-minimal co-occurrences of length w.

We say that $[i,j]$ is a *minimal co-occurrence* if neither $[i+1,j]$ nor $[i,j-1]$ are co-occurrences and we denote the μ minimal co-occurrences of Q in S by $[\ell_1,r_1],\ldots,[\ell_\mu,r_\mu]$ where $r_1 < \ldots < r_\mu$. This ordering is unique since at most one minimal co-occurrence ends at a given index. To simplify the presentation we define $r_{\mu+1} = n+1$. Note that also $\ell_1 < \ldots < \ell_\mu$ due to the following property.

Property 1. *Let $[a,b]$ and $[a',b']$ be two minimal co-occurrences. Either both $a < a'$ and $b < b'$, or both $a' < a$ and $b' < b$.*

Proof. If $a < a'$ and $b \geq b'$ then $[a,b]$ strictly contains another minimal co-occurrence $[a',b']$ and can therefore not be minimal itself. The other cases are analogous.

□

We now show that given all the minimal co-occurrences we can determine all the left-minimal co-occurrences.

Lemma 2. *Let $[\ell_1,r_1],\ldots,[\ell_\mu,r_\mu]$ be the minimal co-occurrences of Q in S where $r_1 < \ldots < r_\mu$ and let $r_{\mu+1} = n+1$. Then*

(a) there are no left-minimal co-occurrences that end before r_1, i.e., at an index $k < r_1$, and

(b) for each index k where $r_i \leq k < r_{i+1}$ for some i, the left-minimal co-occurrence ending at k starts at ℓ_i.

Proof. See Fig. 3.1 for an illustration of the proof.

(a) If $[j,k]$ is a left-minimal co-occurrence where $k < r_1$, it must contain some minimal co-occurrence that ends before r_1 — obtainable by shrinking $[j,k]$ maximally — leading to a contradiction.

(b) Let $r_i \leq k < r_{i+1}$. Then $[\ell_i,k]$ is a co-occurrence since it contains $[\ell_i,r_i]$. We show that it is left-minimal by showing that $[\ell_i+1,k]$ is not a co-occurrence. If it were, it would contain a minimal co-occurrence $[s,t]$ where $\ell_i < s$ and $t < r_{i+1}$. By Property 1, $\ell_i < s$ implies that $r_i < t$. However, then $r_i < t < r_{i+1}$, leading to a contradiction.

□

Let $\mathsf{len}(i,j) = j - i + 1$ denote the length of the interval $[i,j]$. Lemma 2 implies that each minimal co-occurrence $[\ell_i, r_i]$ gives rise to one additional left-minimal co-occurrence of length k for $k = \mathsf{len}(\ell_i, r_i), \ldots, \mathsf{len}(\ell_i, r_{i+1}) - 1$. Also, note that each left-minimal co-occurrence is determined by a minimal co-occurrence in this manner. Therefore, $\mathsf{lmco}(w)$ equals the number of minimal co-occurrences $[\ell_i, r_i]$ where $\mathsf{len}(\ell_i, r_i) \le w < \mathsf{len}(\ell_i, r_{i+1})$. Recall that $\delta(i) = \mathsf{lmco}(i) - \mathsf{lmco}(i-1)$ for $i \in [2,n]$. Since $\mathsf{lmco}(1) = 0$ (because $|Q| \ge 2$) we have $\mathsf{lmco}(w) = \sum_{i=2}^{w} \delta(i)$. It follows that

$$\delta(w) = \sum_{i=1}^{\mu} \begin{cases} 1 & \text{if } \mathsf{len}(\ell_i, r_i) = w \\ -1 & \text{if } \mathsf{len}(\ell_i, r_{i+1}) = w \\ 0 & \text{otherwise} \end{cases}$$

since the contribution of each $[\ell_i, r_i]$ to the sum $\sum_{i=2}^{w} \delta(i)$ is one if $\mathsf{len}(\ell_i, r_i) \le w < \mathsf{len}(\ell_i, r_{i+1})$ and zero otherwise. We say that $[\ell_i, r_i]$ *contributes* plus one and minus one to $\delta(\mathsf{len}(\ell_i, r_i))$ and $\delta(\mathsf{len}(\ell_i, r_{i+1}))$, respectively.

However, note that only the non-zero $\delta(\cdot)$-entries affect the result of lmco-queries. Denote the elements of Z by $z_1 < z_2 < \ldots < z_d$ and define $\mathsf{pred}(w)$ such that $z_{\mathsf{pred}(w)}$ is the predecessor of w in Z, or $\mathsf{pred}(w) = 0$ if w has no predecessor. We get the following lemma.

Lemma 3. *For any $w \in [z_1, n]$ we have that*

$$\mathsf{lmco}(w) = \sum_{i=1}^{\mathsf{pred}(w)} \delta(z_i) = \mathsf{lmco}(z_{\mathsf{pred}(w)}).$$

For $w \in [0, z_1)$, w has no predecessor in Z and $\mathsf{lmco}(w) = 0$.

Proof. The proof follows from the fact that $\mathsf{lmco}(w) = \sum_{i=2}^{w} \delta(i)$ and $\delta(i) = 0$ for each $i \notin Z$. □

3.2.2 Data Structure

The contents of the data structure are as follows. Store the linear space predecessor structure from [Wil83] over the set Z, and for each key $z_i \in Z$ store the data $\mathsf{lmco}(z_i)$. To answer $\mathsf{lmco}(w)$, find the predecessor $z_{\mathsf{pred}(w)}$ of w and return $\mathsf{lmco}(z_{\mathsf{pred}(w)})$. Return 0 if w has no predecessor.

The correctness of the query follows from Lemma 3. The query time is $O(\log\log|U|)$ [Wil83] which is $O(\log\log n)$ since the universe is $[2,n]$, i.e., the domain of δ. The predecessor structure uses $O(|Z|) = O(d)$ space, which we now show is $O(\sqrt{nq})$. We begin by bounding the cumulative length of the minimal co-occurrences.

Lemma 4. *Let $[\ell_1, r_1], \ldots, [\ell_\mu, r_\mu]$ be the minimal co-occurrences of Q in S. Then*

$$\sum_{i=1}^{\mu} \mathsf{len}(\ell_i, r_i) = O(nq).$$

Proof. We prove that for each $k \in [1,n]$ there are at most q minimal co-occurrences $[\ell_i, r_i]$ where $k \in [\ell_i, r_i]$; the statement in the lemma follows directly. Suppose that there are $q' > q$ minimal co-occurrences $[s_1, t_1], \ldots, [s_{q'}, t_{q'}]$ that contain k and let $t_1 < \ldots < t_{q'}$. By Property 1, and because each minimal occurrence contains k, we have

$$s_1 < \ldots < s_{q'} \le \; k \; \le t_1 < \ldots < t_{q'}$$

Furthermore, for each s_i we have that $S[s_i] = p$ for some $p \in Q$; otherwise $[s_i + 1, t_i]$ would be a co-occurrence and $[s_i, t_i]$ would not be minimal. Since $q' > q$ there is some $p \in Q$ that occurs twice as the first character, i.e., such that $S[s_i] = S[s_j] = p$ for some $i < j$. However, then $[s_i + 1, t_i]$ is a co-occurrence because it still contains $S[s_j] = p$, contradicting that $[s_i, t_i]$ is minimal. □

By the definition of δ, we have that $\delta(k) \neq 0$ only if there is some minimal co-occurrence $[\ell_i, r_i]$ such that either $\mathsf{len}(\ell_i, r_i) = k$ or $\mathsf{len}(\ell_i, r_{i+1}) = k$. Using this fact in conjunction with Lemma 4 we bound the sum of the elements in Z.

$$\begin{aligned}\sum_{z \in Z} z = \sum_{k \text{ where } \delta(k) \neq 0} k \quad &\leq \sum_{i=1}^{\mu} \mathsf{len}(\ell_i, r_i) + \mathsf{len}(\ell_i, r_{i+1}) \\ &= \sum_{i=1}^{\mu} \mathsf{len}(\ell_i, r_i) + \Big(\mathsf{len}(\ell_i, r_i) + \mathsf{len}(r_i + 1, r_{i+1})\Big) \\ &= \sum_{i=1}^{\mu} 2 \cdot \mathsf{len}(\ell_i, r_i) + \sum_{i=1}^{\mu} \mathsf{len}(r_i + 1, r_{i+1}) \\ &= O(nq) + O(n)\end{aligned}$$

Since the sum over Z is at most $O(nq)$ we must have $d = O(\sqrt{nq})$, because with $d = \omega(\sqrt{nq})$ distinct elements in Z we have

$$\sum_{z \in Z} z \geq 1 + 2 + \ldots + d = \Omega(d^2) = \omega(nq).$$

3.3 The Co-Occurrence Problem

Recall that $\mathsf{co}(w)$ is the number of co-occurrences of length w, as opposed to the number of left-minimal co-occurrences of length w. That is, $\mathsf{co}(w)$ counts the number of co-occurrences among the intervals $[1, w], [2, w+1], \ldots, [n - w + 1, n]$. We reduce the co-occurrence problem to the left-minimal co-occurrence problem as follows.

Lemma 5. *Let S be a string over an alphabet Σ, let $Q \subseteq \Sigma$ and let lmco be defined as above. Then*

$$\mathsf{co}(w) = \left(\sum_{i=2}^{w} \mathsf{lmco}(i) \right) - \max(w - r_1, 0).$$

Proof. For any index $k \geq w$, the length-w interval $[k - w + 1, k]$ ending at k is a co-occurrence if and only if the length of the left-minimal co-occurrence ending at index k is at most w. The sum $\sum_{i=2}^{w} \mathsf{lmco}(i)$ counts the number of indices $j \in [1, n]$ such that the left-minimal co-occurrence ending at j has length at most w. However, this also includes the left-minimal co-occurrences that end at any index $j \in [r_1, w-1]$. While all of these have length at most $w - 1$, none of the length-w intervals that end in the range $[r_1, w - 1]$ correspond to substrings of S, so they are not co-occurrences. Therefore, the sum $\sum_{i=2}^{w} \mathsf{lmco}(i)$ overestimates $\mathsf{co}(w)$ by $w - r_1$ if $r_1 < w$ and by 0 otherwise.

□

We show how to represent the sequence $\sum_{i=2}^{2} \mathsf{lmco}(i), \ldots, \sum_{i=2}^{n} \mathsf{lmco}(i)$ compactly, in a similar way to what we did for lmco-queries. Recall that the elements of Z are denoted by $z_1 < \ldots < z_d$, that $z_{\mathsf{pred}(x)}$ is the predecessor of x in Z, and that $\mathsf{pred}(x) = 0$ if x has no predecessor. Then, for any $w \geq 2$ we get that

$$\begin{aligned}\sum_{i=2}^{w} \mathsf{lmco}(i) &= \sum_{i=2}^{w} \sum_{j=1}^{\mathsf{pred}(i)} \delta(z_j) \\ &= \sum_{k=1}^{\mathsf{pred}(w)} \delta(z_k)(w - z_k + 1) \\ &= (w+1) \underbrace{\sum_{k=1}^{\mathsf{pred}(w)} \delta(z_k)}_{\mathsf{lmco}(w)} - \sum_{k=1}^{\mathsf{pred}(w)} z_k \delta(z_k)\end{aligned} \tag{3.1}$$

The first step follows by Lemma 3 and the second step follows because $\delta(z_k)$ occurs in $\sum_{j=1}^{\mathsf{pred}(i)} \delta(z_j)$ for each of the $w - z_k + 1$ choices of $i \in [z_k, w]$.

To also support co-queries we extend our data structure from before as follows. For each z_k in the predecessor structure we store $\sum_{i=1}^{k} z_i \delta(z_i)$ in addition to $\mathsf{lmco}(z_k)$. We also store r_1. Using Lemma 5 and Equation 3.1 we can then answer co-queries with a single predecessor query and a constant amount of extra work, taking $O(\log\log n)$ time. The space remains $O(d) = O(\sqrt{nq})$. This completes the proof of Theorem 3(a), as well as the upper bound on space from Theorem 3(c).

3.4 Lower Bounds

In this section we show lower bounds on the space complexity of data structures that support lmco- or co-queries. In Section 3.4.1 we introduce a gadget that we use in Section 3.4.2 to prove that any data structure supporting lmco- or co-queries needs $\Omega(d)$ space (we use the same gadget in Appendix 3.B to prove Theorem 4). In Section 3.4.3 we prove that any solution to the (left-minimal) co-occurrence problem requires $\Omega(\sqrt{nq})$ bits of space in the worst case.

All the lower bounds are proven by reduction to the left-minimal co-occurrence problem. However, they extend to data structures that support co-queries by the following argument. Store r_1 and any data structure that supports co on S and Q in time t per query. Then this data structure supports lmco-queries in $O(t)$ time, because by Lemma 5 we have that

$$\begin{aligned}
&\mathsf{co}(w) \; - \; \mathsf{co}(w-1) \\
&\quad = \left(\sum_{i=2}^{w} \mathsf{lmco}(i) \; - \; \max(w - r_1, 0) \right) - \left(\sum_{i=2}^{w-1} \mathsf{lmco}(i) \; - \; \max(w - 1 - r_1, 0) \right) \\
&\quad = \mathsf{lmco}(w) - \max(w - r_1, 0) + \max(w - 1 - r_1, 0)
\end{aligned}$$

3.4.1 The Increment Gadget

Let $Q = \{\mathtt{A}, \mathtt{B}\}$ and $U = \{2, \ldots, u\}$. For each $i \in U$ we define the *increment gadget*

$$G_i = \underbrace{\mathtt{A}\,\$\cdots\$\,\mathtt{B}}_{i}\;\underbrace{\$\cdots\$}_{u}$$

where $\$\cdots\$$ denotes a sequence of characters that are not in Q.

Lemma 6. *Let $Q = \{A, B\}$, $U = \{2, \ldots, u\}$, and let G_i be defined as above. Furthermore, for some $E = \{e_1, e_2, \ldots, e_m\} \subseteq U$ let S be the concatenation of $c_1 > 0$ copies of G_{e_1}, with $c_2 > 0$ copies of G_{e_2}, and so on. That is,*

$$S = \underbrace{G_{e_1} \cdots G_{e_1}}_{c_1} \;\cdots\cdots\cdots\; \underbrace{G_{e_m} \cdots G_{e_m}}_{c_m}$$

Then $\delta(e_i) = c_i$ for each $e_i \in E$ and $\delta(e) = 0$ for any $e \in U \setminus E$. Furthermore, $m \leq d \leq 8m$ and $n \leq 2uC$ where $C = \sum_{i=1}^{m} c_i$ is the number of gadgets in S.

Proof. Firstly, $|G_j| = j + u \leq 2u$ since $j \in U$, so the combined length of the C gadgets is at most $2uC$.

Now we prove that $\delta(e_i) = c_i$ for each $e_i \in E$ and $\delta(e) = 0$ for each $e \in U \setminus E$. Consider two gadgets G_j and G_k that occur next to each other in S.

$$\overbrace{\underbrace{\mathtt{A}\,\$\cdots\$\,\mathtt{B}}_{j}\;\underbrace{\$\cdots\$}_{u}}^{G_j}\;\overbrace{\underbrace{\mathtt{A}\,\$\cdots\$\,\mathtt{B}}_{k}\;\underbrace{\$\cdots\$}_{u}}^{G_k}$$

Three of the minimal co-occurrences in S occur in these two gadgets. Denote them by $[s_1, t_1]$,$[s_2, t_2]$ and $[s_3, t_3]$.

$$\underbrace{\texttt{A}\,\$\cdots\$\,\texttt{B}}_{[s_1,t_1]}\overbrace{\,\$\cdots\$\,\texttt{A}}^{[s_2,t_2]}\underbrace{\,\$\cdots\$\,\texttt{B}}_{[s_3,t_3]}\,\$\cdots\$$$

The two first minimal co-occurrences start in G_j. They contribute

- plus one to $\delta(x)$ for $x \in \{\mathsf{len}(s_1, t_1), \mathsf{len}(s_2, t_2)\} = \{j, u + 2\}$.
- minus one to $\delta(x)$ for $x \in \{\mathsf{len}(s_1, t_2), \mathsf{len}(s_2, t_3)\} = \{j + u + 1, k + u + 1\}$.

Hence, each occurrence of G_j contributes plus one to $\delta(j)$, and the remaining contributions are to $\delta(x)$ where $x \notin U$. The argument is similar also for the last gadget in S that has no other gadget following it. For each $e_i \in E$ there are c_i occurrences of G_{e_i} so $\delta(e_i) = c_i$. For each $e \in U \setminus E$ there are no occurrences of G_e so $\delta(e) = 0$.

Finally, note that each occurrence of $G_j G_k$ at different positions in S contributes to the same four $\delta(\cdot)$-entries. Therefore the number of distinct non-zero $\delta(\cdot)$-entries is linear in the number of distinct pairs (j, k) such that G_j and G_k occur next to each other in S. Here we have no more than $2m$ distinct paris since G_{e_i} is followed either by G_{e_i} or $G_{e_{i+1}}$. Each distinct pair contributes to at most four $\delta(\cdot)$-entries so $d \leq 8m$. Finally each G_{e_i} contributes at least to $\delta(e_i)$ so $m \leq d$, concluding the proof.

□

3.4.2 Lower Bound on Space

We prove that any data structure supporting lmco-queries needs $\Omega(d)$ space in the worst case. Let U and Q be defined as in the increment gadget and let $P = p_2, \ldots, p_m$ be a sequence of length $m - 1$ where each $p_i \in U$ (the first element is named p_2 for simplicity). We let S be the concatenation of p_2 occurrences of G_2, with p_3 occurrences of G_3, and so on. That is,

$$S = \underbrace{G_2 \ldots G_2}_{p_2} \cdots\cdots\cdots \underbrace{G_m \ldots G_m}_{p_m}$$

Then any data structure supporting lmco on S and Q is a representation of P; by Lemma 6 we have that $\delta(i) = p_i$ for $i \in [2, m]$ and by definition we have $\delta(i) = \mathsf{lmco}(i) - \mathsf{lmco}(i - 1)$.

The sequence P can be any one of $(u - 1)^{m-1}$ distinct sequences, so any representation of P requires

$$\log((u - 1)^{m-1}) = (m - 1)\log(u - 1) = \Omega(m \log u)$$

bits — or $\Omega(m)$ words — in the worst case. By Lemma 6 this is $\Omega(d)$.

3.4.3 Lower Bound on Space in Terms of n and q

Here we prove that any data structure supporting lmco needs $\Omega(\sqrt{nq})$ bits of space in the worst case.

The main idea is as follows. Given an integer α and some $k \in \{2, \ldots, \alpha\}$, let V be the set of *even* integers from $\{k + 1, \ldots, k\alpha\}$, and let T be some subset of V. We will construct an instance S and Q where

- the size of Q is $q = k$
- the length of S is $n = O(k\alpha^2)$
- for each $i \in V$ we have $\delta(i) = 1$ if and only if $i \in T$.

Then, as above, any data structure supporting lmco-queries on S and Q is a representation of T since $\delta(i) = \mathsf{lmco}(i) - \mathsf{lmco}(i - 1)$. There are $2^{\Omega(k\alpha)}$ choices for T, so any representation of T requires

$$\log 2^{\Omega(k\alpha)} = \Omega(k\alpha) = \Omega(\sqrt{k^2\alpha^2}) = \Omega(\sqrt{nq})$$

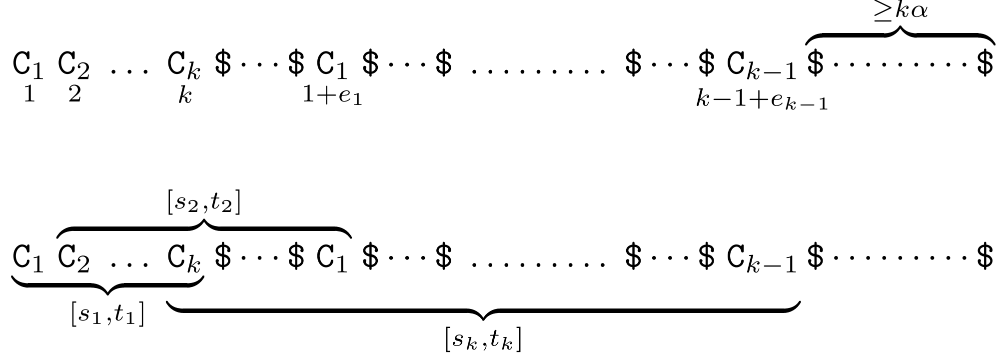


Figure 3.2: *Top:* Shows the layout of the gadget R_j, where $\texttt{\$}\cdots\texttt{\$}$ denotes a sequence of characters not in Q. The first k characters are $\texttt{C}_1 \ldots \texttt{C}_k$. For $i \in [1, k-1]$ there is another occurrence of $\texttt{C}_i$ at index $i + e_i$. Note that the second occurrence of $\texttt{C}_1$ occurs before the second occurrence of $\texttt{C}_2$, and so on. All other characters are $. Since $|R_j| = 3k\alpha$ and $k - 1 + e_{k-1} \leq 2k\alpha$, R_j ends with at least $k\alpha$ characters that are not in Q. *Bottom:* Shows the k minimal co-occurrences in R_j denoted by $[s_1, t_1], \ldots, [s_k, t_k]$. Each of the $k - 3$ minimal co-occurrences that are not depicted start at the first occurrence of some $\texttt{C}_i$ and ends at the second occurrence of $\texttt{C}_{i-1}$.

bits in the worst case.

The reduction is as follows. Let $Q = \{\texttt{C}_1, \ldots, \texttt{C}_k\}$ and let $ be a character not in Q. Assume for now that $|T|$ is a multiple of $k - 1$ and partition T arbitrarily into $t = O(\alpha)$ sets $T_1, \ldots, T_t$, each of size $k - 1$. Consider $T_j = \{e_1, \ldots, e_{k-1}\}$ where $e_1 < e_2 < \ldots < e_{k-1}$. We encode T_j in the gadget R_j where

- the length of R_j is $3k\alpha$.
- $R_j[1, k] = \texttt{C}_1\texttt{C}_2 \ldots \texttt{C}_k$.
- $R_j[i + e_i] = \texttt{C}_i$ for each $\texttt{C}_i$ except $\texttt{C}_k$. This is always possible since $i + e_i < (i + 1) + e_{i+1}$.
- all other characters are $.

See Fig. 3.2 for an illustration both of the layout of R_j and of the minimal co-occurrences contained within it. There are k minimal co-occurrences contained in R_j which we denote by $[s_1, t_1], \ldots, [s_k, t_k]$.

- The first one, $[s_1, t_1] = [1, k]$, starts and ends at the first occurrence of $\texttt{C}_1$ and $\texttt{C}_k$, respectively.
- For each $i \in [2, k]$, the minimal co-occurrence $[s_i, t_i]$ starts at the first occurrence of $\texttt{C}_i$ and ends at the second occurrence of $\texttt{C}_{i-1}$, i.e., $[s_i, t_i] = [i, i - 1 + e_{i-1}]$.

Consider how these minimal co-occurrences contribute to δ. Each $[s_i, t_i]$ contributes plus one to $\delta(\mathsf{len}(s_i, t_i))$. For the first co-occurrence, $\mathsf{len}(s_1, t_1) = k$ (which is not a part of the universe V). Each of the other co-occurrences $[s_i, t_i]$ has length

$$\mathsf{len}(s_i, t_i) \; = \; t_i - s_i + 1 \; = \; (i - 1 + e_{i-1}) - i + 1 \; = \; e_{i-1}$$

Therefore, the remaining minimal co-occurrences contribute plus one to of each the $\delta(\cdot)$-entries $e_1, \ldots, e_{k-1}$.

Furthermore, each $[s_i, t_i]$ contributes negative one to $\delta(\mathsf{len}(s_i, t_{i+1}))$, where we define $t_{k+1} = |R_j| + 1$. For $i < k$ we have that $\mathsf{len}(s_i, t_{i+1}) = 1 + \mathsf{len}(s_{i+1}, t_{i+1}) = 1 + e_i$ since $s_i + 1 = s_{i+1}$. Note that $1 + e_i$ is odd since e_i is even, and therefore not in V. For $i = k$, we get that $t_{i+1} = t_{k+1} = |R_j| + 1$. The last $k\alpha$ (at least) characters of R_j are not in Q, so $\mathsf{len}(s_k, |R_j| + 1) > k\alpha$ and therefore not in V.

Hence, R_j contributes plus one to $\delta(e_1), \ldots, \delta(e_{k-1})$ and does not contribute anything to $\delta(i)$ for any other $i \in V \setminus T_j$. To construct S, concatenate $R_1, \ldots, R_t$. Note that any minimal co-occurrence that crosses

the boundary between two gadgets will only contribute to $\delta(i)$ for $i > k\alpha$ due to the trailing characters of each gadget that are not in Q. Since S consists of $t = O(\alpha)$ gadgets that each have length $O(k\alpha)$, we have $n = O(k\alpha^2)$ as stated above.

Finally, note that the assumption that $|T|$ is a multiple of $k-1$ is not necessary. We ensure that the size is a multiple of $k-1$ by adding at most $k-2$ *even* integers from $\{k\alpha+1, \dots, 2k\alpha\}$ and adjusting the size of the gadgets accordingly. The reduction still works because we add even integers, the size of S is asymptotically unchanged, and any minimal co-occurrence due to the extra elements will have length greater than $k\alpha$ and will not contribute to any relevant δ-entries.

Acknowledgements We would like to thank the anonymous reviewers for their comments, which improved the presentation of the paper.

3.A Preprocessing

Finding Minimal Co-Occurrences To build the data structure, we need to find all the minimal co-occurrences in order to determine δ. For $j \geq r_1$, let $\mathsf{lm}(j)$ denote the length of the left-minimal co-occurrence ending at index j. By Lemma 2, $\mathsf{lm}(r_i) = \mathsf{len}(\ell_i, r_i)$ for each $i \in [1, \mu]$. Furthermore, for $j \in [r_i + 1, r_{i+1} - 1]$ we have $\mathsf{lm}(j) = \mathsf{lm}(j-1) + 1$ since both of the left-minimal co-occurrences ending at these two indices start at ℓ_i. However, $\mathsf{lm}(r_i) \leq \mathsf{lm}(r_i - 1)$ for each $i \in [2, \mu]$; the left-minimal co-occurrence ending at r_i starts at least one index further to the right than the left-minimal co-occurrence ending at $r_i - 1$ because $\ell_{i-1} < \ell_i$, so it cannot be strictly longer.

We determine $\ell_1, \dots, \ell_\mu$ and $r_1, \dots, r_\mu$ using the following algorithm. Traverse S and maintain $\mathsf{lm}(j)$ for the current index j. Whenever $\mathsf{lm}(j) \neq \mathsf{lm}(j-1) + 1$ the interval $[j - \mathsf{lm}(j) + 1,\ j]$ is one of the minimal co-occurrences. Note that this algorithm finds the minimal co-occurrences in order by their rightmost endpoint. We maintain $\mathsf{lm}(j)$ as follows. For each character $p \in Q$ let $\mathsf{dist}(j, p)$ be the distance to the closest occurrence of p on the left of j. Then $\mathsf{lm}(j)$ is the maximum $\mathsf{dist}(j, \cdot)$-value. As in [SBD+21], we maintain the $\mathsf{dist}(j, \cdot)$-values in a linked list that is dynamically reordered according to the well-known *move-to-front* rule. The algorithm works as follows. Maintain a linked list over the elements in Q, ordered by increasing dist-values. Whenever you see some $p \in Q$, access its node in expected constant time through a dictionary and move it to the front of the list. The least recently seen $p \in Q$ (i.e., the p with the largest $\mathsf{dist}(j, \cdot)$-value) is found at the back of the list in constant time. The algorithm uses $O(q)$ space and expected constant time per character in S, thus it runs in expected $O(n)$ time.

Building the Data Structure We build the data structure as follows. Traverse S and maintain the two most recently seen minimal co-occurrences using the algorithm above. We maintain the non-zero $\delta(\cdot)$-values in a dictionary D that is implemented using chained hashing in conjunction with universal hashing [CW79]. When we find a new minimal co-occurrence $[\ell_{i+1}, r_{i+1}]$ we increment $D[\mathsf{len}(\ell_i, r_i)]$ and decrement $D[\mathsf{len}(\ell_i, r_{i+1})]$. Recall that $Z = \{z_1, \dots, z_d\}$ where $z_j < z_{j+1}$ is defined such that $\delta(i) \neq 0$ if and only if $i \in Z$. After processing S the dictionary D encodes the set $\{(z_1, \delta(z_1)), \dots, (z_d, \delta(z_d))\}$. Sort the set to obtain the array $E[j] = \delta(z_j)$. Compute the partial sum array over E, i.e the array

$$F[j] = \sum_{i=1}^{j} E[i] = \sum_{i=1}^{j} \delta(z_i) = \mathsf{lmco}(z_j). \qquad \text{(we use 1-indexing)}$$

Build the predecessor data structure over Z and associate $\mathsf{lmco}(z_j)$ with each key z_j.

The algorithm for finding the minimal co-occurrences uses $O(q)$ space and the remaining data structures all use $O(d)$ space, for a total of $O(d + q)$ space. Finding the minimal co-occurrences and maintaining D takes $O(n)$ expected time, and so does building the predecessor structure from the sorted input.

Furthermore, we use the following sorting algorithm to sort the d entries in D with $O(d)$ extra space in expected $O(n)$ time. If $d < n/\log n$ we use merge sort which uses $O(d)$ extra space and runs in $O(d \log d) =$

$O(n)$ time. If $d \geq n/\log n$ we use radix sort with base $\sqrt{n}$, which uses $O(\sqrt{n})$ extra space and $O(n)$ time. To elaborate, assume without loss of generality that $2k$ bits are necessary to represent n. We first distribute the elements into $2^k = O(\sqrt{n})$ buckets according to the most significant k bits of their binary representation, partially sorting the input. We then sort each bucket by distributing the elements in that bucket according to the *least* significant k bits of their binary representation, fully sorting the input. The algorithm runs in $O(n)$ time and uses $O(\sqrt{n}) = O(n/\log n) = O(d)$ extra space.

3.B Lower Bound on Time

We now prove Theorem 4 by the following reduction from the predecessor problem. Let U, Q and G_i be as defined in Section 3.4.1 and let $X = \{x_1, x_2, \ldots, x_m\} \subseteq U$ where $x_1 < \ldots < x_m$. Define

$$S = \underbrace{G_{x_1} \cdots G_{x_1}}_{x_1} \; \underbrace{G_{x_2} \cdots G_{x_2}}_{x_2 - x_1} \; \cdots\cdots\cdots \; \underbrace{G_{x_m} \cdots G_{x_m}}_{x_m - x_{m-1}}$$

By Lemma 6 we have that $\delta(x_1) = x_1$, $\delta(x_i) = x_i - x_{i-1}$ for $i \in [2, m]$ and $\delta(i) = 0$ for $i \in U \setminus X$. Then, if the predecessor of some $x \in U$ is x_p, we have

$$\mathsf{lmco}(x) = \sum_{i=2}^{x} \delta(i) = x_1 + (x_2 - x_1) + \ldots + (x_p - x_{p-1}) = x_p$$

On the other hand, if $x < x_1$ then $\sum_{i=0}^{x} \delta(i) = 0$, unambiguously identifying that x has no predecessor.

Applying Lemma 6 again, we have $d \leq 8m$. Furthermore, there are $x_1 + (x_2 - x_1) + \ldots + (x_m - x_{m-1}) = x_m \leq u$ gadgets in total so $n \leq 2u^2$. Hence, given a data structure that supports lmco in $f_t(n, q, d)$ time using $f_s(n, q, d)$ space, we get a data structure supporting predecessor queries on X in $O(f_t(2u^2, 2, 8m))$ time and $O(f_s(2u^2, 2, 8m))$ space, proving Theorem 4.

Chapter 4

Sliding Window String Indexing in Streams

Sliding Window String Indexing in Streams

Philip Bille[*,†] phbi@dtu.dk
Johannes Fischer[‡] johannes.fischer@cs.tu-dortmund.de
Inge Li Gørtz[*,†] inge@dtu.dk
Max Rishøj Pedersen[*,†] mhrpe@dtu.dk
Tord Joakim Stordalen[†] tjost@dtu.dk

† : Technical University of Denmark, DTU Compute, Kgs. Lyngby, Denmark
‡ : Technische Universität Dortmund, Department of Computer Science, Germany

Abstract

Given a string S over an alphabet Σ, the *string indexing problem* is to preprocess S to subsequently support efficient pattern matching queries, that is, given a pattern string P report all the occurrences of P in S. In this paper we study the *streaming sliding window string indexing problem*. Here the string S arrives as a stream, one character at a time, and the goal is to maintain an index of the last w characters, called the *window*, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and all occurrences of P within the current window must be returned. The streaming sliding window string indexing problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching.

Our main result is a simple $O(w)$ space data structure that uses $O(\log w)$ time with high probability to process each character from both the input string S and any pattern string P. Reporting each occurrence of P uses additional constant time per reported occurrence. Compared to previous work in similar scenarios this result is the first to achieve an efficient worst-case time per character from the input stream with high probability. We also consider a delayed variant of the problem, where a query may be answered at any point within the next δ characters that arrive from either stream. We present an $O(w + \delta)$ space data structure for this problem that improves the above time bounds to $O(\log(w/\delta))$. In particular, for a delay of $\delta = \epsilon w$ we obtain an $O(w)$ space data structure with constant time processing per character. The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees of independent interest, inspired by the classic log-structured merge trees.

4.1 Introduction

The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. In this paper, we introduce a basic variant of string indexing called the *streaming sliding window string indexing (SSWSI) problem*. Here, the string S arrives as a stream one character at a time, and the goal is to maintain an index of a *window* of the last w characters, for a specified parameter w. At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and we need to report the occurrences of P within the current window. The goal is to compactly maintain the index while processing the characters arriving in either stream efficiently. We consider two variants of the problem: a *timely* variant where each query must be answered immediately, and a *delayed* variant where it may be answered at any point within the next δ characters arriving from either stream, for a specified parameter δ. See Section 4.1.1 for precise definitions.

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

The SSWSI problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching. For instance, monitoring a high-rate data stream system where we cannot feasibly index the entire stream but still want to support efficient queries. Depending on the specific system we may require immediate answers to queries, or we may be able to afford a delay that allows for more efficient queries and updates.

The SSWSI problem has not been explicitly studied before in our precise formulation, but for the timely variant several closely related problem are well-studied. In particular, the *sliding window suffix tree problem* [FG89, Lar99, Sen05, BJ18, NAIP03] is to maintain the *suffix tree* of the current window (i.e., the compact trie of the suffixes of the window) as each character arrives. With appropriate augmentation the suffix tree can be used to process pattern matching queries efficiently, leading to a solution to the timely SSWSI problem. For constant-sized alphabets, the best of these solutions [BJ18] maintains the sliding window suffix tree in constant *amortized* time per character while supporting efficient pattern matching queries. The worst-case time for updates is $\Omega(w)$. The other solutions achieve similar amortized time bounds. This amortization cannot be avoided since explicitly maintaining the suffix tree after the arrival of a new character may incur $\Omega(w)$ changes.

Another closely related problem is the *online string indexing problem* [AKLL05, Kop12, BI13, Kos94, AN08, KN17, FG05, AFG$^+$14]. Here the goal is to process S one character at a time (in either left-to-right or right-to-left order), while incrementally building an index of the string read so far. The best of these solutions update the index in either constant time per character for constant-sized alphabets [KN17] or $O(\log\log n + \log\log|\Sigma|)$ time for any alphabet where each character fits in a constant number of machine words [Kop12]. These solutions all heavily rely on processing the string in right-to-left order to avoid the inherent linear time suffix tree updates due to appending, as mentioned above. Therefore they cannot be applied in our left-to-right streaming setting. Alternatively, we can instead apply these solutions on the reverse of the string S, but then each pattern must be processed in reverse order, which also cannot be done in our setting. Also, note that these solutions index the entire string read so far. It is not clear if they can be adapted to efficiently index a sliding window.

Another line of work shows how to maintain a fully dynamic suffix array under insertions and deletions [AB20, AB21, SLLM10, KK22]. These can also be used to solve SSWSI but are more general and lead to polylogarithmically slower bounds than our results while being more complicated.

Our main result is an efficient and simple solution to the SSWSI problem in both the timely and delayed variant. Let w denote the size of the window. For the timely variant, we present a string index that uses $O(w)$ space and processes a character from the stream S in $O(\log w)$ time. Each pattern matching query P is also supported in $O(\log w)$ time per character with additional $O(\mathrm{occ})$ time incurred after receiving the last character of P, where occ is the number of occurrences of P in the current window. The index is randomized and both time bounds hold with high probability. Compared to previous suffix tree based approaches for indexing a sliding window, we improve the worst-case time bounds per character in the stream from $\Omega(w)$ to $O(\log w)$ with high probability. This is particularly important in the above mentioned applications, such as high-rate data stream systems. Our solution generalizes to the delayed variant of the problem. If we allow a delay of δ before answering each query we achieve $O(w+\delta)$ space while improving the above time bounds to $O(\log(w/\delta))$. In particular, if we allow a delay of $\delta = \epsilon w$ for any constant $\epsilon > 0$, we achieve linear space and optimal constant time (reporting the occurrences still takes $O(\mathrm{occ})$ time, and we do not count the reporting time towards the delay). Note that $\delta \leq w$ is sufficient delay for optimal time bounds and we can assume $O(w+\delta) = O(w)$. The results hold on a word RAM and for any alphabet size, assuming that each character fits into a constant number of machine words.

The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees inspired by log-structured merge trees [OCGO96]. Instead of maintaining a single suffix tree on the window we maintain a collection of suffix trees of exponentially increasing sizes that cover the current window. We show how to efficiently maintain the structure as new characters from the stream arrive by incrementally "merging" suffix trees, while supporting efficient pattern matching queries within the window.

Our solution uses randomization to construct suffix trees in linear time with high probability. Plugging in a deterministic construction algorithm such as the one by Ukkonen [Ukk95], we obtain a solution

using $O(\log w \log|\Sigma|)$ time for both queries and updates. With more recent deterministic suffix tree solutions [FG05, CKL15, BGS17] we can improve this to obtain $O(\log w \log\log n)$ time per character for both queries and updates. Note that the $O(\log\log|\Sigma|)$ in the time bounds of [FG05] has been replaced by $O(\log\log n)$ here due to an additional sorting step using [Han02].

4.1.1 Setup and Results

We formally define the problem as follows. Let S be a stream over any alphabet Σ where each character fits in a constant number of machine words. For given integer parameters $w \geq 1$ and $\delta \geq 0$, the *δ-delayed streaming sliding window string indexing ((w,δ)-SSWSI)* problem is to maintain a data structure that, after receiving the first i characters of S, supports

- Report(P): report all the occurrences of P in $S[i-w+1,i]$ before an additional δ characters have arrived from either stream.
- Update(): process the next character in the stream S.

In the Report(P) query the pattern string P is also streamed. When P is streamed it interrupts the stream S, arrives one character at a time, and all characters of P arrive before the streaming of S resumes. Furthermore, we do not assume that we know the length of P before the arrival of its last character. Although P is streamed we assume random access to its characters after they arrive, as any pattern that fits in the window is at most w characters long and we can afford to store it. The delay is counted from after the last character of P arrives. Characters from S and from new patterns count towards the delay, while reported occurrences do not (otherwise it would be impossible to answer the query in time if there are more than δ occurrences).

We define the *timely streaming sliding window string indexing (w-SSWSI) problem* to be $(w,0)$-SSWSI, that is, queries must be answered immediately as the last character of the pattern arrives.

We show the following general main result.

Theorem 5. *Let S be a stream and let $w \geq 1$ and $\delta \geq 0$ be integers. We can solve the (w,δ)-SSWSI problem on S with an $O(w+\delta)$ space data structure that supports* Update *and* Report *in $O(\log \frac{w}{\delta+1})$ time per character with high probability. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

Here, with high probability means with probability at least $1-\frac{1}{w^d}$ for any constant d. Theorem 5 provides a trade-off in the delay parameter δ. In particular, plugging in $\delta = 0$ in Theorem 5 we obtain a solution to the timely SSWSI problem that uses $O(w)$ space and $O(\log w)$ time per character for both Update and Report. Compared to the previous work on sliding window stream indexing [FG89, Lar99, Sen05, BJ18, NAIP03, ISTA04, SD08] this improves the worst-case bounds on the Update operation from $\Omega(w)$ to $O(\log w)$ with high probability and also removes the restriction on the alphabet. At the other extreme, plugging in $\delta = \epsilon w$ for constant $\epsilon > 0$ in Theorem 5 we obtain a solution to the delayed SSWSI problem that uses $O(w)$ space and optimal constant time per character with high probability. All our results hold on a word RAM where each machine word has at least $\log w$ bits, and where each character of the alphabet fits into a constant number of machine words.

4.1.2 Techniques

We obtain our result for the timely variant, but without high probability guarantees, as follows. At all times we maintain at most $\log w$ suffix trees that do not overlap and together cover the window. The trees are organized by the *log-structured merge technique* [OCGO96], where the rightmost tree is the smallest and their sizes increase exponentially towards the left. For each new character that arrives we append its suffix tree to the right side of our data structure. Whenever there are two trees of the same size next to each other we "merge" them by constructing a new suffix tree covering them both. Each character from S is involved in at most $\log w$ merges and each merge takes expected linear time, so we spend expected amortized $O(\log w)$ time per character in S. We deamortize the updates by temporarily keeping both trees while merging them

in the background. Note that for each adjacent pair of suffix trees we also store a suffix tree approximately covering them both, referred to as *boundary trees* (see details below).

We find the occurrences of a pattern P in the window by querying each of these trees, which takes $O(\log w)$ time per character in P. For adjacent pairs of trees larger than $|P|$ we find the occurrences of P crossing from one into the other using the boundary trees. The remaining trees cover a suffix of the window of length $O(|P|)$, and we grow a suffix tree to answer queries in this suffix *at query time*. Our data structure has some "overhang" on the left side of the window, and we use range maximum queries to report only the occurrences that start inside the window.

This solution is generalized to incorporate a delay of δ as follows. We store the $O(\log(w/\delta))$ largest trees from the timely solution and leave a suffix of size $\Theta(\delta)$ of the window uncovered by suffix trees. We answer queries as follows. If $|P| > \delta/4$ we say that P is *long*, and otherwise it is *short*. For long patterns we do as in the timely case; the suffix tree we grow at query time now must also contain the uncovered suffix, but it still has size $O(|P|)$ since the uncovered part of the window has length $O(\delta) = O(|P|)$. We show how to do this in $O(\log(w/\delta))$ time per character in P. For short patterns we utilize that they are smaller than the delay to temporarily buffer the queries and later batch process them. We buffer up to $O(\delta \log(w/\delta))$ work and deamortize it over $\Theta(\delta)$ characters, obtaining the same bound as for long patterns. Updates run in the same bound since each character from S is involved in at most $O(\log(w/\delta))$ merges before it leaves the window.

Finally, we improve the time bounds by proving that for any substring S' of our window, we can construct the suffix tree over S' in $O(|S'|)$ time with probability $1 - w^{-d}$ for any constant $d > 1$. We do so by reducing the alphabet $\Sigma' = \{c \in S'\}$ of S' to rank-space $\{1, 2, \ldots, |\Sigma'|\}$ from which the algorithm by Farach-Colton et al. [FFM00] can construct the suffix tree in worst-case linear time. For large strings ($|S'| > w^{1/5}$) we pick a hash function from $\Sigma \to [0, w^c]$ that with high probability is injective on S', and then we use radix sort to reduce to rank-space in linear time. For small strings ($|S'| \leq w^{1/5}$) we pick a hash function from $\Sigma \to [0, w/\log w]$ that is injective with (almost) high probability, and use this to manually construct a mapping into rank space in $O(S')$ time. This mapping algorithm uses additional $O(w/\log w)$ space, but we construct at most $O(\log w)$ suffix trees at any time so the total space is linear.

4.1.3 Outline

In Section 4.2 we cover the preliminaries, including some useful facts about suffix trees. In Section 4.3 we give a solution to the timely SSWSI problem that supports each operation in expected logarithmic time per character. In Section 4.4 we show how to generalize this to incorporate delay, and in Section 4.5 we show how to get good probability guarantees, proving Theorem 5.

4.2 Preliminaries

Given a string X of length n over an alphabet Σ, the ith character is denoted $X[i]$ and the substring starting at $X[i]$ and ending at $X[j]$ is denoted $X[i, j]$. The substrings of the form $X[i, n]$ are the *suffixes* of X.

A *segment* of X is an interval $[i, j] = \{i, i+1, \ldots, j\}$ for $1 \leq i \leq j \leq n$. We will sometimes refer to segments as strings, i.e., the segment $[i, j]$ refers to the string $X[i, j]$. The definition differs from "substring" by being specific about position; even if $X[1, 2] = X[3, 4]$ we have $[1, 2] \neq [3, 4]$. A *segmentation* of X is a decomposition of X into disjoint segments that cover it. For instance, $x_1 = [1, i]$ and $x_2 = [i+1, n]$ is a segmentation of X into two parts. The two segments x_1 and x_2 are *adjacent* since x_2 starts immediately after x_1 ends, and for a pair of adjacent segments we define the *boundary* (x_1, x_2) to be the implicit position between i and $i+1$.

The *suffix tree* [Wei73] T over X is the compact trie of all suffixes of $X\$$, where $\$ \notin \Sigma$ is lexicographically smaller than any letter in the alphabet. Each leaf corresponds to a suffix of X, and the leaves are ordered from left to right in lexicographically increasing order. The suffix tree uses $O(n)$ space by implicitly representing the string associated with each edge using two indices into X. Farach-Colton et al. [FFM00] show that the optimal construction time for T is $\text{sort}(n, |\Sigma|)$, i.e., the time it takes to sort n elements from the universe Σ. For alphabets of the form $\Sigma = \{0, \ldots, n^c\}$ for constant $c \geq 1$ this implies that T can be built in worst-case

$O(n)$ time using radix sort. For larger alphabets we can reduce to the polynomial case in expected linear time using hashing, building T in expected linear time (see Section 4.5 for details).

The *suffix array* L of X is the array where $L[i]$ is the starting position of the ith lexicographically smallest suffix of X. Note that $L[i]$ corresponds to the ith leaf of T in left-to-right order. Furthermore, let v be an internal node in T and let s_v be the string spelled out by the root-to-v path. The descendant leaves of v exactly correspond to the suffixes of X that start with s_v, and these leaves correspond to a consecutive range $[\alpha, \beta]_v$ in L.

We augment the suffix tree to support efficient pattern matching queries as follows. First, we use the well-known FKS perfect hashing scheme [FKS84] to store the edges of the suffix tree, so we can for any node determine if there is an outgoing edge matching a character $a \in \Sigma$ in worst-case constant time. Note that this construction takes *expected* linear time. Furthermore, we also build a *range maximum query* data structure over L. This data structure supports range maximum queries, i.e., given a range $[\alpha, \beta]$ return the $j \in [\alpha, \beta]$ maximizing $L[j]$. It also supports range minimum queries, defined analogously. The data structure can be built in linear time and supports queries in constant time [GBT84]. Finally, we preprocess the suffix tree in linear time such that each internal node v stores the range $[\alpha, \beta]_v$ into L corresponding to the occurrences of s_v.

We can use this structure to efficiently find all the occurrences of P in $O(|P| + \text{occ})$ time, where occ is the number of occurrences, or the leftmost and rightmost occurrence of P in $O(|P|)$ time. The *locus* of a string P is the minimum depth node v such that P is a prefix of s_v. First we find the locus by walking downwards in the suffix tree, matching each character in P in worst-case constant time using the dictionary. Once we have found v we can report all the occurrences in $[\alpha, \beta]_v$ in $O(\text{occ})$ time. Alternatively, we can find the rightmost occurrence of P in constant time by doing a range maximum query on the range $[\alpha, \beta]_v$ in L, which returns the $j \in [\alpha, \beta]_v$ maximizing the *string position* $L[j]$. We can also find the leftmost occurrence by doing a range minimum query.

Finally, note that it is possible to deamortize algorithms with *expected* running time using the standard technique of distributing the work evenly. Specifically, if an algorithm runs in expected λn time we can do λ work for $n - 1$ steps; by linearity of expectation only expected λ work remains for the last step.

4.3 The Timely SSWSI Problem

Here we present a solution for the timely variant that matches the bounds in Theorem 5 in expectation. Section 4.5 shows how to get the bounds with high probability. Throughout this section we assume without loss of generality that w is a power of two. Section 4.3.3 briefly mentions how to generalize to arbitrary w.

The main idea is as follows. We maintain a suffix of S of length at least w. This suffix is segmented into at most $\log w$ segments whose sizes are distinct powers of two, in increasing order from right to left. The length of the suffix we store is at most $2^0 + \ldots + 2^{\log w} = 2w - 1$. When a new character arrives, we append a new size-one segment to our data structure and merge equally-sized segments until they all have distinct sizes again. We also discard the largest segment when it no longer intersects the window. For each segment we store a suffix tree, and for every pair of adjacent segments we store a *boundary tree* approximately covering them both (see below). To support queries we query the suffix tree for each individual segment, and also each boundary tree. For the segments larger than the pattern, the boundary trees are sufficient to find the occurrences crossing the respective boundary. The remaining trees cover a suffix of S that is $O(|P|)$ long, and we grow a suffix tree at query time to find the remaining occurrences in this suffix.

4.3.1 Data Structure

At any point, the data structure contains a suffix s of S of length $w \leq |s| \leq 2w - 1$ and a segmentation of s into at most $\log w$ segments. Specifically, if $|s| = 2^{b_1} + \ldots + 2^{b_k}$ for integers $b_1 < \ldots < b_k$ then we have the segmentation $s_1, \ldots, s_k$ where $|s_i| = 2^{b_i}$, and s is the concatenation of the strings $s_k, s_{k-1}, \ldots, s_1$, in that order. The set $\{b_1, \ldots, b_k\}$ is unique and corresponds to the 1-bits in the binary encoding of $|s|$. Three different configurations can be seen in Figure 4.1.

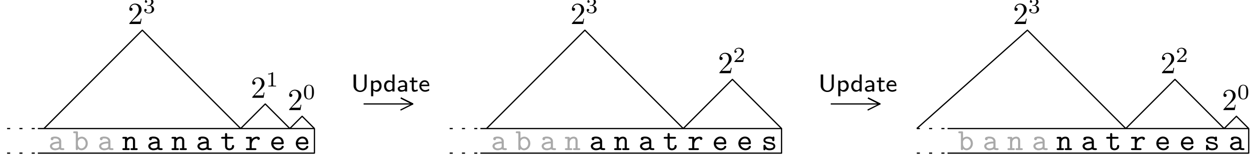


Figure 4.1: Example of updating the data structure with a window size of $w = 8$. Here we illustrate the segments by the suffix trees built over them. Characters outside of the window are gray. As the character `s` arrives we construct a new suffix tree of size one, which is then immediately merged with the existing size-one suffix tree over `e` into a size-two suffix tree over `es`, which is then merged to into the final size-four suffix tree over `rees`. After receiving `a` we again have a size-one suffix tree. Note that after three more updates the suffix tree of size eight will no longer overlap the window and will be discarded.

For each segment s_i we store the suffix tree T_i over s_i, along with a range maximum query data structure over the suffix array of s_i. For each boundary (s_{i+1}, s_i) we store the *boundary tree* B_i, which is the suffix tree over the substring centered at the boundary and extending $|s_i|$ characters in both directions. We augment B_i with an additional data structure that we will use for reporting occurrences across the boundary. Let BL_i be the suffix array corresponding to B_i. We define the *modified suffix array* BL'_i as

$$BL'_i[j] = \begin{cases} BL_i[j] & \text{if } BL_i[j] \text{ corresponds to a suffix starting in } s_{i+1} \\ -\infty & \text{if } BL_i[j] \text{ corresponds to a suffix starting in in } s_i \end{cases}$$

We store a range maximum query data structure over BL'_i. Each of the data structures use $O(s_i)$ space, so the whole data structures uses $O(s) = O(w)$ space.

We note a few properties of the data structure. Let $S[n]$ be the most recent character to arrive and let $W_n = S[n - w + 1, n]$ be the current window. Then W_n is a suffix of s since $|s| \geq w$. The largest, and leftmost, segment s_k always has size $2^{\log w} = w$; it is not larger since $\log w$ bits are sufficient to represent $|s| \leq 2w - 1$, and it is always there since $|s| \geq w$ cannot be represented with $\log w - 1$ bits. For the same reason, s_k always intersects at least partially with W_n, and each of $s_1, \ldots, s_{k-1}$ are fully contained in W_n.

4.3.2 Queries

The idea is as follows, as exemplified in Figure 4.2. Any occurrence of a pattern P that is fully contained in a segment is found using the suffix tree over that segment. Similarly, any occurrence that only crosses a single boundary far enough away from the end of the window is found in the respective boundary tree. Note that in the leftmost segment we must be careful to not report any occurrences that start before the left window boundary. The remaining occurrences are not contained in any of the trees in the data structure (either because they cross multiple boundaries or because they cross a single boundary (s_{i+1}, s_i) but start more than $|s_i|$ characters to the left of the boundary). However, these occurrences are all located within a substring of size $O(m)$ ending at position $S[n]$, so we build, at query time, a suffix tree to find these occurrences.

Let P be the length-m pattern being queried, $S[n]$ be the most recent character to arrive, and let W_n, the suffix s, the segmentation $s_1, \ldots, s_k$, and the indices $b_1 < \ldots < b_k$ be defined as above. As mentioned, any occurrence of P in W_n must either be fully contained within one of the segments, or it must cross the boundary between two adjacent segments. We will show how to handle each of these cases separately.

Fully Contained in a Segment Fix a specific segment s_i. As each character of P arrives we match it in T_i. When the last character arrives we have a (possibly empty) range $[\alpha, \beta]$ into the suffix array of s_i corresponding to the occurrences of P. If s_i is not the leftmost segment then it is fully contained in W_n and we report all the occurrences. Otherwise, $s_i = s_k$ is the leftmost segment, which might overlap only partially with W_n, and it may contain occurrences of P that are not contained in the window. However, note that the intersection between W_n and s_k is a suffix of s_k. Therefore, if an occurrence of P in s_k *starts* inside W_n it

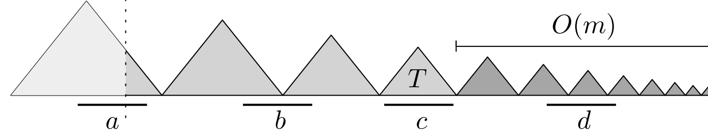


Figure 4.2: Illustration of how we answer queries for a pattern P of length m. The lines denoted a, b, c, and d indicate occurrences of P. The segmentation is illustrated by the trees over the segments. The leftmost window boundary is marked with a vertical dashed line. Note that the leftmost segment intersects only partially with the window. The tree T marks the smallest segment of size at least m. The segments to the right of T are all smaller than m, so they cover at most $m + m/2 + \ldots + 1 = O(m)$ characters. To answer the query we match P in the tree over each segment and in each boundary tree, and we also build a suffix tree over the segments smaller than m at query time. We find b because the respective boundary tree is sufficiently large. We find c because it is fully contained in a segment. We find d in the suffix tree that we build at query time. Note that a is not contained in the window; we avoid reporting it by recursively using range maximum queries to find the *rightmost* occurrence of P in the leftmost segment.

also ends inside W_n. We find all such occurrences as follows. Let L_k be the suffix array of s_k. As described in Section 4.2 we find the index j of the rightmost occurrence of P by doing a range maximum query on the range $[\alpha, \beta]$ in L_k. If $L_k[j]$ is not inside W_n then none of the occurrences are, and we are done. Otherwise we recurse on $[\alpha, j-1]$ and $[\beta, j-1]$. Matching P in the trees of all the segments takes $O(\log w)$ overall time per character of P. Reporting each occurrence takes constant time since range maximum queries run in constant time.

Crossing a Boundary We now show how to report the occurrences of P that span a boundary. The main idea is as follows, as illustrated in Figure 4.3. Let s_i be the smallest segment where $|s_i| \geq m$. Consider any boundary (s_{j+1}, s_j) to the left of s_i, i.e., where $j \geq i$. Since both of these segments have size at least $|s_i| \geq m$, the boundary tree B_j extends at least m characters in both directions from the boundary. Therefore, all the occurrences of P crossing the boundary are contained in B_j, and none of them can cross another boundary as well. Now consider the suffix $\mathcal{R}$ of s containing the $m-1$ last characters of s_i and extending to the end of s. This substring contains all the other boundary-crossing occurrences. Furthermore, all the occurrences in $\mathcal{R}$ cross at least one boundary since the longest consecutive part of a single segment in $\mathcal{R}$ is the $m-1$ characters in s_i. Note that the length of $\mathcal{R}$ is at most $m-1+|s_{i-1}|+|s_{i-1}|/2+\ldots+1 < m-1+2|s_{i-1}| < 3m$ since $|s_{i-1}| < m$. Thus, the number of boundary-crossing occurrences of P equals the number of occurrences in $\mathcal{R}$ plus the number of occurrences crossing the boundaries $(s_k, s_{k-1}), (s_{k-1}, s_{k-2}), \ldots (s_{i+1}, s_i)$.

The algorithm for finding the occurrences in the sufficiently large boundary trees is as follows. Fix a boundary (s_{x+1}, s_x). We match each character of P in B_x as it arrives. When the last character arrives we know if $|s_x| \geq m$, and also the range $[\alpha, \beta]$ corresponding to the occurrences of P in the boundary tree. If $|s_x| \geq m$ (hence $x \geq i$) we report the occurrences as follows. As above we do a range maximum query to find the j maximizing $BL'_x[j]$. If $BL'_x[j] = -\infty$ then all occurrences of P start in s_x, and there are no occurrences crossing the boundary. Otherwise, $BL'_x[j]$ corresponds to the starting position of the rightmost occurrence of P in s_{x+1}. Since all of P has arrived and we now know m, we know that this occurrence crosses the boundary if and only if $BL'_x[j] \geq |s_x| - m + 2$ (recall that B_x extends $|s_x|$ characters in both directions from the boundary). If it does not cross the boundary, then none of the other occurrences do either. Otherwise we report $BL'_x[j]$ and recurse on $[\alpha, j-1]$ and $[j+1, \beta]$ to find the remaining occurrences. Matching P in all boundary trees takes $O(\log w)$ overall time per character, and reporting each occurrence with range maximum queries takes constant time.

We now show how to find the occurrences of P in $\mathcal{R}$ with the same bounds. Assume that we know that $2^\ell \leq m < 2^{\ell+1}$ for some integer ℓ. We build the suffix tree over the last $3 \cdot 2^{\ell+1}$ characters of s, deamortized over receiving the first $2^{\ell-1}$ characters of P. Over the next $2^{\ell-1}$ characters we match P in the tree, at a rate of two characters per new character from P. Then, when the 2^ℓth character arrives, we have caught up to

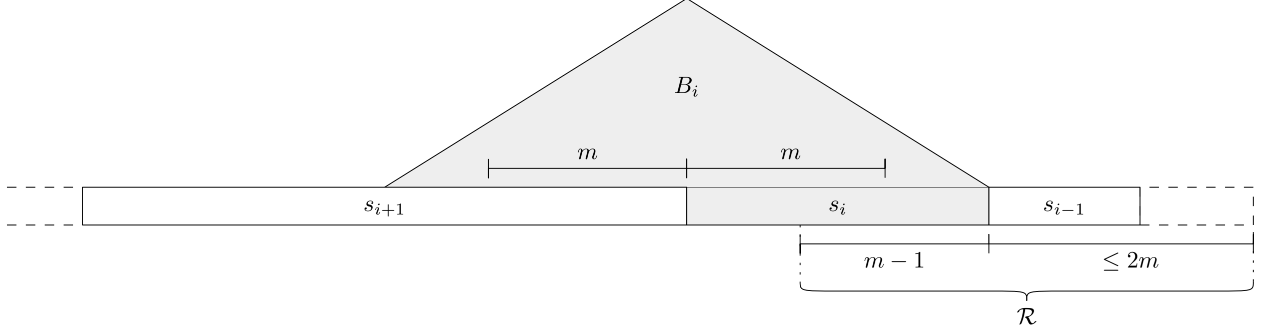


Figure 4.3: The segment s_i is the smallest segment where $|s_i| \geq m$. For each boundary (s_{j+1}, s_j) where $j \geq i$, the tree B_j is large enough to find all occurrences of P across the boundary. All other occurrences of P that cross a boundary must be in $\mathcal{R}$, the string covering the $m-1$ rightmost characters of s_i and extending to the end of the window. The length of $\mathcal{R}$ is no more than $m - 1 + |s_{i-1}| + |s_{i-1}|/2 + \ldots + 1 < 3m$.

the stream P, and we match the remaining $m - 2^\ell$ characters as they arrive. When the last character arrives we have matched P in a tree of size at least $3m$, and we can start reporting occurrences. Note that we are overestimating the size of the tree, and it potentially includes some occurrences of P that are contained in s_i. To avoid reporting these, we also build a range maximum query data structure over the suffix array such that we can use recursive range maximum queries. When deamortized, we construct the tree in expected constant time per character of P. Matching P also takes constant time per character. We know that $m \leq w$, so we run this algorithm simultaneously for each of the $\log w$ different choices for ℓ, using expected $O(\log w)$ time per character in P. Note that the trees use $O(w)$ space in total since the sum of the space is a geometric sum where the largest term is $O(w)$.

4.3.3 Amortized Updates

We show how to support updates in amortized $O(\log w)$ time. Let $S[n]$ be the last character to arrive and as in the description of the data structure let $b_1 < b_2 < \ldots < b_k$ be the positions of the 1-indices in the binary encoding of $|s|$. When the new character $c = S[n+1]$ arrives, we update s and the segmentation $s_1 \ldots s_k$ to create the new suffix s' with the new segmentation $s'_1, \ldots s'_{k'}$. See Figure 4.1 for an example.

If $|s| < 2w - 1$ then we set $s' = sc$. The segmentation of s' corresponds to the unique binary encoding of $|s'| = |s| + 1$, so we update the segmentation analogously to a "binary increment". One way to do so is as follows. We create a new segment of size one over c. If there was not already a segment of size one, then we add the new segment and we are done. Otherwise we *merge* (see below) the two size-one segments to create a segment of size two. The process cascades until we reach a size 2^b that does not exist in the segmentation of s (i.e., the smallest index $b \notin \{b_1, \ldots, b_k\}$). At this point we replace all of the segments $s_{b-1}, \ldots, s_1$ with s'_1 covering the last 2^b characters of s'. The remaining segments for s' are the same as the segments $s_{b+1}, \ldots, s_k$. If $|s| = 2w - 1$ then there is a segment of each size $2^0, 2^1, \ldots, 2^{\log w}$. Since the segments have decreasing size from left to right, the $\log w - 1$ rightmost segments cover the last $2^0 + \ldots + 2^{\log w - 1} = w - 1$ characters of s. Thus, after c arrives, the leftmost segment of size $2^{\log w} = w$ no longer intersects the window. We remove it by setting $s' = s[w+1, |s|]c$, and update the segmentation as above.

Let s_a, s_b and s_c be three adjacent segments, in that order. To *merge* s_b and s_c we combine them into a new segment s_d that spans them both, construct the suffix tree over s_d, and construct a range maximum query data structure on the suffix array of s_d. Furthermore, since s_a and s_d are now adjacent we also construct the boundary-spanning suffix tree for the boundary (s_a, s_d) that extends $|s_d|$ characters in each direction. The construction of all of these data structures takes expected $O(|s_d|)$ time (see Section 4.2). Thus, it takes expected constant time per character every time it moves into a new, larger segment. Each character is contained in at most $\log w$ segments before it leaves the window, so the amortized update time is expected $O(\log w)$ per character.

Note that all but the last merge are unnecessary to actually compute s_1'; in the amortized setting we can simply determine where the cascade will end and immediately construct the suffix tree over the corresponding segment. However, the cascading merges will come into play in the deamortized variant.

Also note that if w is not a power of two we can use a similar scheme where we allow either two simultaneous trees of size $2^{\lfloor \log w \rfloor}$, or one tree of size $2^{\lceil \log w \rceil}$. In both cases, there are some straightforward edge cases for when to remove the leftmost segment.

4.3.4 Deamortized Updates

We now show how to deamortize the updates. Unfortunately the previous construction cannot be directly deamortized since the suffix tree construction algorithm by Farach-Colton et al. [FFM00] requires access to the whole string. Therefore, if a new character c causes a cascade of merges resulting in a new segment of size 2^i we have to build the suffix tree over that segment when c arrives.

Instead, we modify the structure slightly. When two segments of size 2^i become adjacent we temporarily keep both while deamortizing the cost of merging them over the *next* 2^i characters of S, doing expected constant work per character. Note that queries are unaffected, with one exception for reporting occurrences across the boundaries; there might now be two adjacent segments s_{i+1} and s_i of the same size that are both the smallest segment at least as large as $|P|$. In this case the suffix $\mathcal{R}$ extends only $m-1$ characters into the rightmost segment s_i. The boundary tree for (s_{i+1}, s_i) is large enough to report all occurrence crossing that boundary since both segments have size at least $|P|$. Furthermore, $\mathcal{R}$ potentially becomes twice as long, so we adjust the constants of the trees that we grow at query time.

To bound the time for updates we show that we are constructing at most $\log w$ suffix trees at any point, from which it follows that the update time is expected $O(\log w)$. To do so we show the following lemma.

Lemma 7. *When the construction of a segment of size 2^i finishes there is exactly one segment of each size $2^{i-1}, \ldots, 2^0$.*

Proof. The proof is by induction on i. For $i = 1$, when two size-one segments become adjacent we merge them when the next character c from S arrives. This results in a segment of size two, as well as a size-one segment containing c, proving the base case.

Inductively, consider the first time two segments of size 2^i become adjacent. By the induction hypothesis, there is one segment of each size $2^0, 2^1, \ldots, 2^{i-1}$ to the right of these two segments. For another segment of size 2^i to be constructed, we must first receive one more character, which triggers a merge that eventually cascades through all $i-1$ of these segments. For this to happen, $1+(2^0+2^1+\ldots+2^{i-1}) = 2^i$ more characters from S must arrive, where the 1 is for the next character to arrive, and 2^j is the amount of characters the jth merge is deamortized over. However, at this point the merge of the two segments of size 2^i is complete, so we constructed two new segments, one of size 2^{i+1} and one of size 2^i. By the induction hypothesis, there is also one segment of each size $2^0, \ldots, 2^{i-1}$, concluding the proof. □

Lemma 7 implies that there are never more than two segments of the same size adjacent to each other, and therefore at most one merging process for each segment size $2^0, 2^1, \ldots, 2^{\log w}$. To see this, consider the first time two segments a and b of size 2^i are adjacent. At this point, there are $2^0 + 2^1 + \ldots + 2^{i-1} = 2^i - 1$ characters to the right of b. When the next segment c of size 2^i arrives there are $2^i - 1$ characters to the right of that, too. But then there are $|c| + 2^i - 1 = 2^i + 2^i - 1$ characters to the right of b. Thus 2^i new characters must have arrived in the meanwhile, and the merging of a and b is done.

We obtain the following theorem.

Theorem 6. *Let S be a stream and let $w \geq 1$ be an integer. We can solve the w-SSWSI problem on S with an $O(w)$ space data structure that supports* Update *and* Report *in expected $O(\log w)$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

4.4 The Delayed SSWSI Problem

In this section we show how to improve the result from Section 4.3 if we are allowed a delay of δ. The main idea is as follows. As before, we maintain suffix trees of exponentially increasing sizes, although only the $O(\log(w/\delta))$ largest of them. As a result there are fewer trees to query, but also an *uncovered* suffix of size $\Theta(\delta)$ of the window for which we do not have any suffix trees. As in Section 4.3 we denote the part of S covered by suffix trees by s and we denote the uncovered suffix by t. As above, s is segmented into $s_1, \ldots, s_k$.

We will first explain how to solve the problem when all patterns are *long*, that is, $|P| > \delta/4$, and then when all patterns are *short*, that is, $|P| \leq \delta/4$. Finally we show how to combine these solutions. When all the patterns are long we can afford to construct, at query time, a suffix tree covering t. On the other hand, when all the patterns are short we can do both updates and queries in an offline fashion; we buffer queries and updates until we have approximately $\delta/2$ operations to do, at which point we can afford to construct a suffix tree over t in a deamortized manner. See Figure 4.4 for an example.

Throughout this section we assume without loss of generality that δ is a power of two. Otherwise we instead use a more restrictive delay of $\delta' = 2^{\lfloor \log \delta \rfloor}$ and achieve the same asymptotic bounds.

4.4.1 Long Patterns

We first show how to support queries if all patterns have a length $m > \delta/4$. We modify the data structure from Section 4.3 slightly. The smallest tree now has size $\delta/2$ as opposed to 1, so there are $\Theta(\log w - \log(\delta/2)) = O(\log(w/\delta))$ segments and boundary trees. The uncovered suffix t has length at most δ.

We answer queries the same way as in Section 4.3.2, with only small modifications. Let P be a pattern of length $m > \delta/4$. As before, let s_i be the smallest and rightmost segment with $|s_i| \geq m$. We find any occurrence within a segment or crossing a single boundary by using the suffix trees over each segment and the boundary trees to the left of s_i, as before. The remaining occurrences we again find by growing suffix trees of exponentially increasing sizes from the right window boundary. The only change is that we now grow the trees faster, as we must also cover t, and we can afford to let the smallest tree have size δ since we have $m > \delta/4$ characters in the pattern to deamortize the work over. As above, let $\mathcal{R}$ be the string covering the $m-1$ last characters of s_i and extending to the right window boundary, which now also includes t. As $|t| < \delta$ the length of $\mathcal{R}$ is $|\mathcal{R}| < 3m + \delta < 7m$. Assuming $2^\ell \leq m < 2^{\ell+1}$, we build the suffix tree of size $7 \cdot 2^{\ell+1}$ and match P in it, amortized over the characters of P. As we have $m > \delta/4$ characters to deamortize the work over, we only do this for each choice of ℓ where $2^{\ell+1} \geq \delta$, which results in $O(\log w - \log \delta) = O(\log(w/\delta))$ work per character in P. As in Section 4.3.2 we use recursive range maximum queries to avoid double reporting any occurrences of P that are also in s. As there are also only $O(\log(w/\delta))$ segments and boundary trees we spend $O(\log(w/\delta))$ time per character in P. Note that we answer these queries without delay.

Updates are performed as follows. For each segment of $\delta/2$ characters that arrives we construct the suffix tree over it, deamortized over the next $\delta/2$ characters of S. We merge suffix trees as before, also deamortized over new characters of S. The induction proof from Section 4.3.4 still works by modifying the base case; the merging of two trees of size $\delta/2$ takes $\delta/2$ characters, at which point another tree of size $\delta/2$ is constructed. The inductive step follows from the fact that δ is a power of two. Thus, we spend expected $O(\log(w/\delta))$ time per update.

4.4.2 Short Patterns

We now show how to support queries if all patterns have a length $m \leq \delta/4$. We extend the data structure with a buffer of size δ. This buffer will contain queries that we have not yet answered and characters for S that we have not yet processed. The total space is still $O(w + \delta) = O(w)$.

Whenever a character from S arrives we append it to both t and to the buffer. When a pattern arrives we append the full pattern to the buffer, and along with it we store the current position of the right window boundary. Once the buffer has more than $\delta/2$ characters (patterns and text combined) we immediately allocate a new buffer of size δ and *flush* the old buffer as follows. Note that at this point there are strictly less $\frac{3}{4}\delta$ characters in the buffer since each pattern is short.

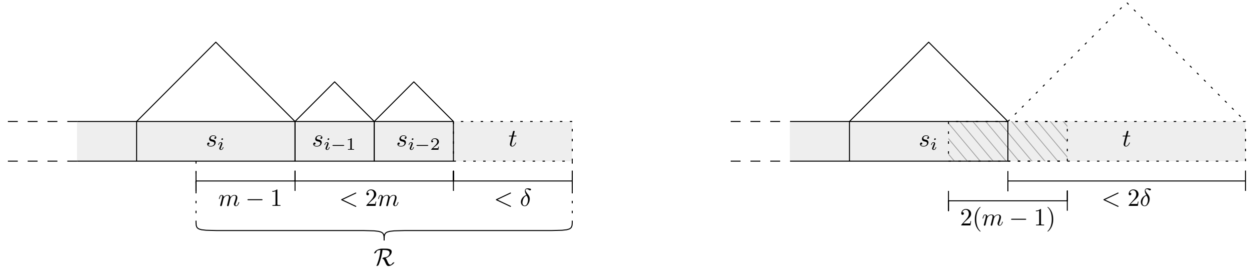


Figure 4.4: *Left:* Example of a query with a long pattern. Here s_i is the smallest and rightmost segment with $|s_i| \geq m$. Note that the non-indexed suffix t is less than $\delta < 4m$ characters long. *Right:* Example of a query with a short pattern. Note that for short patterns, s_i is always the rightmost segment. Any occurrence in s cross at most a single boundary and is found using the constructed trees. Any occurrence in t is found by the suffix tree over t that we construct when we flush the buffer. Any occurrence that cross the boundary (s, t) is found by the KMP automaton we build over the substring the extends $m-1$ characters in both directions from the boundary, which is hatched in the figure.

When we flush the buffer, we first answer all the buffered queries, and then we process all the buffered updates. We deamortize this work over the next $\delta/4$ characters that arrive from either stream. To answer the buffered queries we do as follows. Let $P_1, \ldots, P_\ell$ be the patterns in the buffer, let $m_i = |P_i|$, and let $M = \sum_{1 \leq i \leq \ell} m_i$. We have $M < \delta$. We start by building a suffix tree over t, along with a range maximum query data structure over the suffix array of t. This takes expected $O(\delta)$ time. An occurrence of P_i is either contained in s, or it crosses the boundary (s, t), or it is contained in t. Since P_i is smaller than each segment s_j we can find all the occurrences within s using the suffix trees over the segments and the boundary trees in $O(m_i \log(w/\delta))$ time. To find the occurrences crossing the boundary we build the KMP matching automaton [KJP77] for P_i. In it we match the string that is centered at the boundary (s, t) and extends $m_i - 1$ characters in each direction. This takes $O(m_i)$ time. To find the occurrences in t we match P_i in the suffix tree over t in $O(m_i)$ time. In total, this takes $O(M \log(w/\delta)) = O(\delta \log(w/\delta))$ time for all the patterns, or expected $O(\log(w/\delta))$ time per character when deamortized. Note however, that after P_i arrived more characters from S could have arrived and been appended to t. We must therefore take care not to report any occurrences of P_i that extend past what *was* the right window boundary when P_i arrived. The KMP automaton finds the occurrences in left-to-right order, and in t we avoid reporting too far right using recursive range minimum queries.

Finally, we process each update in the buffer in the order they arrived, using the same procedure as for long patterns. This takes $O(\log(w/\delta))$ time per update and $O(\delta \log(w/\delta))$ time in total. Thus flushing the buffer takes expected $O(\log(w/\delta))$ time per character since we deamortize the expected $O(\delta \log(w/\delta))$ work over $\delta/4$ characters. Since we allocate a new buffer immediately when we begin flushing, we will complete the flush before the next flush begins.

4.4.3 Both Long and Short Patterns

We now show how to combine the solutions for short and long patterns, to obtain a solution that handles patterns of any length. The data structure is the same as for small patterns above. As above, we append each new character to the buffer. However, whenever we start streaming a pattern we also proceed as if P were long. If P turns out to fit in the buffer without triggering a flush (which might also happen if P is long), we simply discard the work we did for the long-pattern case. However, if adding P to the buffer results in more than $\frac{3}{4}\delta$ characters being in the buffer, then P must be long. We immediately start flushing the buffer (ignoring the characters related to P) and also continue processing P as a long pattern. Note that since we are potentially streaming a long pattern while batch processing the updates in the buffer, the data structure might change while we are matching in it. However, it only changes when a merge finishes, replacing a pair of suffix trees by a larger tree. If this happens we keep the old trees in memory until we are done processing

the pattern, at which point we discard them.

We obtain the following theorem.

Theorem 7. *Let S be a stream and let $w \geq 1$ and $\delta \geq 1$ be integers. We can solve the (w, δ)-SSWSI problem on S with an $O(w)$ space data structure that supports* Update *and* Report *in expected $O(\log(w/\delta))$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

4.5 Obtaining High Probability

In this section we show how to improve the time bounds to $O(\log(w/\delta))$ with probability $1 - w^{-d}$ for any constant $d \geq 1$.

The expectation in the time bounds in Section 4.4 comes from the construction of suffix trees (recall that we also build suffix trees at query time). Below, in Lemma 8, we prove that given a string $\mathcal{K}$ of length $k = O(w)$ we can construct the suffix tree over $\mathcal{K}$ in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$, using additional $O(w/\log w)$ space. We use this algorithm to construct suffix trees during updates and queries, deamortizing them as before and doing $O(\log(w/\delta))$ work per character that arrives. When a new character arrives from S or P, at most $O(\log(w/\delta)) = O(\log w)$ suffix tree constructions will finish. At this point, we finish constructing those trees that did not finish in time, that is, used more more time than what was allotted to them. By the union bound, the probability that any of them fail to finish in time (and thus incurring extra construction cost) is no more than $c \log w/w^{1+\epsilon}$ for some constant c which is no more than $1/w$ for large w. Thus, for each character from S or P we spend $O(\log(w/\delta))$ time with high probability in w. We obtain the $1 - 1/w^d$ probability bound by probability boosting, running $d = O(1)$ independent copies of the construction algorithm simultaneously. The algorithm from Lemma 8 uses additional $O(w/\log w)$ space, but we are never constructing more than $O(\log w)$ suffix trees, so the space usage is $O(w)$ in total.

Furthermore, as mentioned in Section 4.2, we previously used an FKS dictionary [FKS84] to store the edges to support reporting queries in worst-case constant time per character in the pattern. The construction time of this dictionary is expected linear, so it can no longer be used. Instead we use a dictionary by Dietzfelbinger and Meyer auf der Heide [DadH90]. If there are n elements in the dictionary it supports searches in worst-case constant time and any sequence of $\frac{1}{2}n$ updates takes constant time per update with probability $1 - 1/n^{d'}$ for any constant $d' \geq 1$. We store all the edges of all the suffix trees in one such dictionary. At all times, we keep $\Theta(w)$ dummy-elements in the dictionary to ensure that we get good probability bounds in terms of w, and we choose d' large enough that any sequence of $O(w)$ operations (e.g., the construction of any one of our suffix trees) runs in $O(w)$ time with probability $1 - 1/w^{d+\epsilon}$.

Universal Hashing Before we prove Lemma 8 we restate some basic facts about universal hashing, introduced by Carter and Wegman [CW79]. Let $M, m > 0$ be integers, $\mathcal{H}$ be a set of functions $[0, M] \to [0, m]$, and $h \in \mathcal{H}$ be selected uniformly at random. Then $\mathcal{H}$ is *universal* if $P[h(x) = h(y) \mid x \neq y] \leq 1/m$. Let $R \subseteq [0, M]$ and $|R| = r$. It follows from the union bound that h has a *collision* on R with probability at most

$$P[h(x) = h(y) \text{ for some } x \neq y] \leq \sum_{x \neq y \in R} P[h(x) = h(y)] = \frac{r(r-1)}{2} \cdot \frac{1}{m} < \frac{r^2}{m} . \tag{4.1}$$

In particular, if $m = r^c$ for constant $c \geq 1$ then h is *injective* (i.e., has no collisions) on R with probability at least $1 - 1/r^{c-2}$. Carter and Wegman gave several classes of universal hash functions from which we can sample a function uniformly at random in constant time.

Fast Suffix Tree Construction We now prove Lemma 8, showing how to construct our suffix trees in linear time with high probability.

Lemma 8. *Given a string $\mathcal{K}$ of length $k \leq 2w$ there is an algorithm that uses $O(k + w/\log w)$ space and constructs the suffix tree over $\mathcal{K}$ in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$ for some $\epsilon > 0$.*

Proof. Let $\sigma = \{\mathcal{K}[i] \mid i \in [1,k]\} \subseteq \Sigma$ be the alphabet of $\mathcal{K}$. We show how to, in $O(k)$ time, find a function $h : \Sigma \to [1, k^{O(1)}]$ such that h is injective on σ with probability at least $1 - 1/w^{1+\epsilon}$. If h is injective on σ, we can construct the suffix tree over $\mathcal{K}'$ where $\mathcal{K}'[i] = h(\mathcal{K}[i])$ in time $O(\text{sort}(k, k^{O(1)})) = O(k)$ using radix sort. After the tree is constructed we can substitute for the original alphabet in linear time. Therefore, the construction algorithm finishes in $O(k)$ time with probability at least $1 - 1/w^{1+\epsilon}$ (otherwise we make no guarantee on the construction time and we can build the suffix tree in any way).

For some m to be determined later, let $f : \Sigma \to [1, m]$ be chosen uniformly at random from a class of universal hash functions. By Equation 4.1, the probability that f has a collision on σ is

$$P[f \text{ has collisions on } \sigma] < \frac{|\sigma|^2}{m} \leq \frac{k^2}{m}\,.$$

We divide into the cases of large trees ($k \geq w^{1/5}$) and small trees ($k < w^{1/5}$). If k is large then $w^{1/5} \leq k \leq 2w$, and we set $m = w^4$ so the probability that f has a collision is at most

$$\frac{k^2}{m} \leq \frac{(2w)^2}{w^4} = \frac{4}{w^2} \leq \frac{1}{w^{1+\epsilon}}$$

for some $\epsilon > 0$. We check whether f is injective by sorting the set $\{(x, f(x)) \mid x \in \sigma\}$ with respect to the $f(\cdot)$-values and checking if two consecutive elements $(x, f(x))$ and $(y, f(y))$ have $x \neq y$ and $f(x) = f(y)$. This takes time $O(\text{sort}(k, w^4)) = O(k)$ using radix sort since $k \geq w^{1/5}$. If f is injective we set $h = f$, concluding the proof of the large case.

If k is small then we allocate an array A of length $w/\log w$ in constant time. For simplicity we assume that A is initialized such that $A[i] = 0$ for all i. This can be avoided using standard constant-time initialization schemes; assume each entry in A contains an arbitrary value initially. We maintain two other arrays B and C such that if we have written a value to $A[i]$ at least once then $A[i]$ is a pointer to some $B[j]$, $B[j]$ is a pointer to $A[i]$, and $C[j]$ stores the value most recently written to $A[i]$. From this we can determine if $A[i]$ has been initialized (check if the pointers match), and if it has not we can initialize it in constant time.

Then we set $m = w/\log w$ such that the probability that f has a collision is no more than

$$\frac{k^2}{m} < \frac{w^{2/5}}{w/\log w} = \frac{\log w}{w^{3/5}} = \frac{\log w}{w^{1/2}} \cdot \frac{1}{w^{1/10}} \leq \frac{1}{w^{1/10}}$$

for $w \geq 16$. We check if f is injective on σ by for each character x in $\mathcal{K}$ setting $A[f(x)] = x$ and seeing if two distinct characters hash to the same index. If f is injective we then arbitrarily assign the values $1, \ldots, |\sigma|$ to the now non-zero indices of A and let $h(x) = A[f(x)]$ (at this point we know σ since it is equal to the number of entries in A that we modified). To boost the probability of success we run this algorithm up to eleven times with independent choices for f. The probability that all of them fail is at most $1/w^{11/10} \leq 1/w^{1+\epsilon}$ concluding the proof for the small case. ☐

In conjunction with Theorems 6 and 7, this proves Theorem 5.

4.6 Conclusion and Future Work

We have studied two variants of the streaming sliding window string indexing problem; the timely variant, where queries must be answered immediately, and the delayed variant where a query may be answered at any point within the next δ characters received, for a specified parameter δ. For a sliding window of size w we have given an $O(w)$ space data structure that, in the timely variant, supports updates in $O(\log w)$ time with high probability and queries in $O(\log w)$ time with high probability per character in the pattern; each occurrence is reported in additional constant time. For the delayed variant we improved these bounds to $O(\log(w/\delta))$, where each occurrence is still reported in constant time.

One open problem is whether these bounds can be improved. Another is to find efficient solutions when queries may be interleaved with new updates to the stream. That is, while you are streaming a pattern, new characters of S might arrive that move the current window.

Chapter 5

Rank and Select on Degenerate Strings

Rank and Select on Degenerate Strings

Philip Bille*
DTU Compute
phbi@dtu.dk

Inge Li Gørtz*
DTU Compute
inge@dtu.dk

Tord Joakim Stordalen
DTU Compute
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.

5.1 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., [BN15, OS07, GMR06, RRS07, PNB17, BCPT15, MN07, FMMN07, BHMS11, BCG+14, NS14, HM10, NN14, GRSV13], references therein, and surveys [Gag16].

A *degenerate string* is a sequence $X = X_1, \dots X_n$ 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 $X_1, \dots, X_n$. Degenerate strings have been studied since the 80s [Abr87] and the literature contains papers on problems such as degenerate string comparison [AAB+20], finding string covers for degenerate strings [CIK+17], and pattern matching with degenerate patterns, degenerate texts, or both [Abr87, IMR08].

Alanko, Biagi, Puglisi, and Vuohtoniemi [ABPV23] 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

*Supported by Danish Research Council grant DFF-8021-002498

- $\textsf{subset-rank}_X(i, c)$: return the number of sets in $X_1, \ldots, X_i$ that contain c
- $\textsf{subset-select}_X(i, c)$: 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 [ABPV23, APV23] for details and further references). Specifically, in another work by some of the authors [APV23], 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 [ABPV23]. They introduce the *Subset Wavelet Tree*, a generalization of the well-known wavelet tree (see [GGV03]) 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 [APV23]) 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 [APV23], 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.

5.2 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 [APV23]. 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 8). 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 9). By plugging a standard rank-select data structure into Theorem 8 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 [ABPV23], and the implementations of the reductions provided in [APV23]. 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 8. *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 8(i) and (ii) are based on the reduction from [APV23, Sec. 4.3], and Theorem 8(iii) is a variation of Theorem 8(ii) that handles empty sets using a natural, alternative strategy. By plugging a standard rank-select structure into Theorem 8 we exponentially improve query times while essentially matching, or improving, space usage compared to Alanko et al. [ABPV23]. For example, consider the rank-select structure by Golynski, Munro, and Rao [GMR06] 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. [BN15].

For $n_0 = 0$, plugging this structure into Theorem 8(i) yields an $N\log\sigma + N + o(N\log\sigma + N)$ bit data structure supporting $\mathsf{subset\text{-}rank}$ in $O(\log\log\sigma)$ time and $\mathsf{subset\text{-}select}$ in constant time. Compared to the previous result by Alanko et al. [ABPV23], 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 $\mathsf{subset\text{-}rank}$ and constant for $\mathsf{subset\text{-}select}$. 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 8(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 8(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 [CM96, Jac89], 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 $\mathsf{select}(i, 1)$ 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 $\mathsf{select}(i, 0)$ 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 [OS07, GORR14].

We also show the following lower bound on the space required to support either $\mathsf{subset\text{-}rank}$ or $\mathsf{subset\text{-}select}$ on a degenerate string.

Theorem 9. *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 8 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 [ABPV23] and the reductions [APV23] implemented by Alanko et al. Specifically, we apply the test framework from [ABPV23] 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 $\mathsf{subset\text{-}rank}$ queries are tested in isolation. We implement Theorem 8(iii) and plug in efficient off-the-shelf rank-select structures from the *Succinct Data Structure Library (SDSL)*[1] [GBMP14]. We also implement a variation of another reduction from [APV23, 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.

5.3 Reductions

We now present the reductions from Theorem 8. Let X, $\mathcal{D}$, and $\mathcal{B}$ be defined as in Theorem 8. Furthermore, let $\mathcal{V}$ be the data structure from [Jac89], which for a length-ℓ bitstring uses $\ell + o(\ell)$ bits and supports rank

[1] `https://github.com/simongog/sdsl-lite`

$$X = \left\{\begin{matrix}\texttt{A}\\\texttt{C}\\\texttt{G}\end{matrix}\right\} \left\{\begin{matrix}\texttt{A}\\\texttt{T}\end{matrix}\right\} \left\{\texttt{C}\right\} \left\{\begin{matrix}\texttt{T}\\\texttt{G}\end{matrix}\right\}$$

$X_1 \quad X_2 \quad X_3 \quad X_4$

$S =$ ACG AT C TG
$R =$ 100 10 1 10 1

$S_1 \quad S_2 \quad S_3 \quad S_4$

Figure 5.1: *Left:* A degenerate string X over the alphabet $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ where $n = 4$ and $N = 8$. *Right:* The reduction from Theorem 8(i) on X. White space is for illustration purposes only. To compute $\textsf{subset-rank}(2, \texttt{A})$, we first compute $\textsf{select}_R(3, \texttt{1}) = 6$. Now we know that S_2 ends at position 5, so we return $\textsf{rank}_S(5, \texttt{A}) = 2$. To compute $\textsf{subset-select}(2, \texttt{G})$ we compute $\textsf{select}_S(2, \texttt{G}) = 8$, and compute $\textsf{rank}_R(8, \texttt{1}) = 4$ to determine that position 8 corresponds to X_4.

and select in constant time.

5.3.1 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 1 followed by $|X_i| - 1$ 0s. 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 1 appended after R_n. The lengths of S and R are respectively N and $N+1$. See Figure 5.1 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 $\textsf{subset-rank}(i, c)$, compute the starting position $k = \textsf{select}_R(i + 1, \texttt{1})$ of S_{i+1} and return $\textsf{rank}_S(k - 1, c)$. To support $\textsf{subset-select}(i, c)$, find the index $k = \textsf{select}_S(i, c)$ of the ith occurrence of c, and return $\textsf{rank}_R(k, \texttt{1})$ to determine which set k is in. Since rank and select queries on R take constant time, subset-rank and subset-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 8(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 8(ii).

5.3.2 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 $\textsf{subset-rank}_X(i, c)$ first compute $k = i - \textsf{rank}_E(i, \texttt{1})$, mapping X_i to its corresponding set X''_k. Then return $\textsf{subset-rank}_{X''}(k, c)$. This takes $\mathcal{B}_r(n, n_0) + \mathcal{D}_r(N, \sigma) + O(1)$ time. To support $\textsf{subset-select}_X(i, c)$, find $k = \textsf{subset-select}_{X''}(i, c)$ and return $\textsf{select}_E(k, \texttt{0})$, 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.

5.4 Lower Bound

In this section we prove Theorem 9. The strategy is as follows. Any structure supporting subset-rank or subset-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

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

bits, concluding the proof.

5.5 Experimental Setup

5.5.1 Setup and Data

The code to replicate our results is available on GitHub[2]. Our tests are based on the test framework by Alanko et al. [ABPV23], also available on GitHub[3]. Like them, we used the following data sets.

1. A pangenome of 3682 E. coli genomes, available on Zenodo[4]. According to [ABPV23], 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 $\approx$ 17 million length-502 sequence reads sampled from the human gut from a study on irritable bowel syndrome and bile acid malabsorption [JDO+20].

We applied two tests. Firstly, we plugged our data structures into the k-mer query test from [ABPV23]; 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 [ABPV23], 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`.

5.5.2 Data Structures

This section summarizes a representative subset of the data structures we tested; see appendix 5.B for a description of, and results for, the remaining data structures. We implement both Theorem 8(iii) as well as variation of the reduction *split representation* from [APV23, 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 8(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

[2] `https://github.com/tstordalen/subset-rank-select`
[3] `https://github.com/jnalanko/SubsetWT-Experiments/`
[4] `https://zenodo.org/record/6577997`

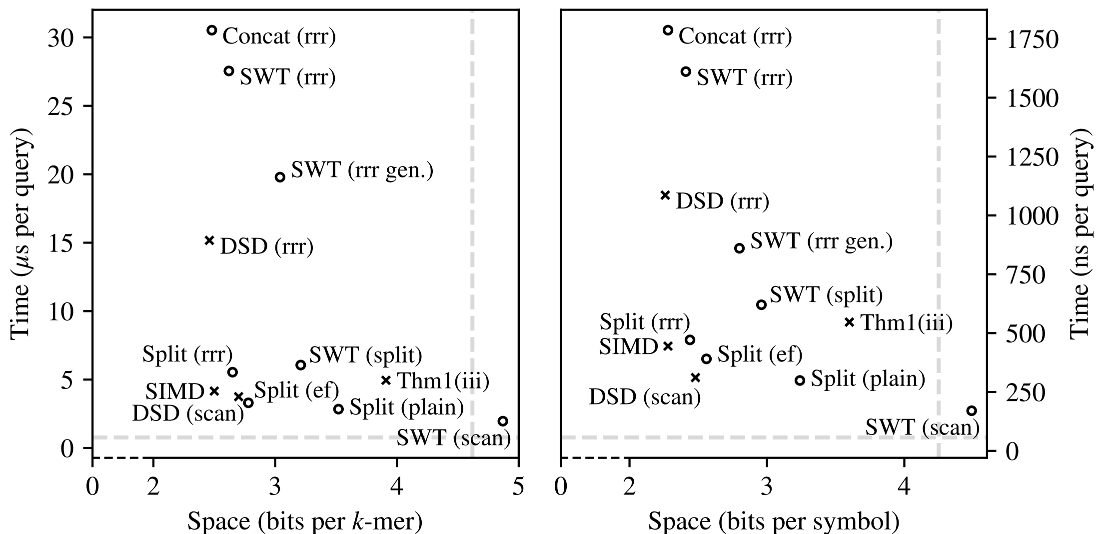


Figure 5.2: Note that the x-axis is truncated in both plots. The two gray lines represent the performance of the benchmark solution "Matrix". The crosses indicate our data structures and the circles indicate the data structures from [ABPV23, APV23]. *Left:* Results of the k-mer query test on the metagenome 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.

in the regular string, and one in the sparse bitvector. In the split representation by [APV23], 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 [ABPV23], consisting of one bitvector per character (i.e., a $4 \times n$ matrix). **Thm 8(iii)** is the reduction from Theorem 8(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. [ABPV23, 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 [ABPV23, 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 [RRS07], *(4)* **rrr gen.**, a generalization of RRR to size-four alphabets [ABPV23, Sec. 5.4], *(5)* **ef**, an efficient implementation of rank queries on a bitvector stored using Elias-Fano encoding from [MPRZ21], and *(6)* **plain**, a standard SDSL bitvectors supporting rank in constant time.

Furthermore, [APV23] 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 also an essential idea in the 'scan' structure by [ABPV23]). We use *SIMD (single instruction, multiple data)* instructions to speed up the partial in-block rank queries, which allows for large blocks and a reduction in space (see Appendix 5.B). 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.

5.6 Results

The test results for the metagenome data set can be seen in Figure 5.2; the results for the E. Coli data set are similar. See appendix 5.A for the data belonging to Figure 5.2, and appendix 5.B for the results of the data structures omitted from this figure. The fastest structure is SWT (scan), but it is large and is outperformed by the benchmark solution on both parameters. Our unoptimized reduction Thm8(iii) uses $20-60\%$ more space than the remaining structures of [ABPV23, APV23] 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 [ABPV23]), 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.

5.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
Thm8(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 5.1: 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 of data structures, separated by horizontal lines; the benchmark structure, our reductions, our structure using SIMD, the Subset Wavelet Trees from [ABPV23], and the reductions from [APV23]. Each group is ordered from fastest to slowest and largest to smallest, except for Thm8(iii) which breaks space order. Each value in the table is the average of five measurements.

5.B Results for all Data Structures

This section elaborates on the data structures that were omitted from Sections 5.5.2 and 5.6. From [APV23], we omitted the structure **Concat (plain)**, which is the structure **Concat (ef)** explained in Section 5.5.2 parameterized with standard SDSL bitvcectors instead. We also omitted **Matrix (ef)** and **Matrix (rrr)**, which is the benchmark solution parameterized with different types of bitvectors.

Furthermore, this section includes more parameterizations of our SIMD enhnaced data structure. We elaborate on the SIMD structure from Section 5.5.2. **DSD (SIMD (i))** (which we refer to as SIMD (i) for simplicity) is the DSD parameterized with the SIMD (i) structure, which supports rank queries on strings over the alphabet $\{0, 1, 2, 3\}$. As described in Section 5.5.2, the main idea is to divide the string into blocks, precompute the answer to rank queries up to the start of each block, and compute partial rank queries internally in blocks as needed using SIMD. The width of the block is determined by the parameter i; there are $512i$ characters stored per block (recall that the width of the SIMD registers we used was 512 bits). The blocks are stored as follows. Each character in $\{0, 1, 2, 3\}$ consists of two bits. We separate the string represented by the block into two bitstrings; one consisting of the low bits, and one of the high bits. To compute partial rank queries, we use the operation `vpternlogq`, which given three vectors and an 8-bit integer value evaluates *any* three-variable boolean function bit-wise for the three vectors (the 8-bit integer describes the result column of the three-variable truth table, which has eight rows). The operation also accepts an additional integer, used to mask out results if they are not needed for the computation. To answer rank queries, we traverse the low and high bitstrings, using `vpternlogq` to find occurrences of the queried character and the mask to filter out results occurring after the queried index. The version of the SIMD data structure in Sections 5.5.2 and 5.6 is SIMD (8).

The results for all the structures on the metagenome data set can be seen in Figure 5.3 (k-mer search test) and Figure 5.4 (rank query benchmark). All the data, for all structures, both data sets, and both tests, can be seen in Table 5.2.

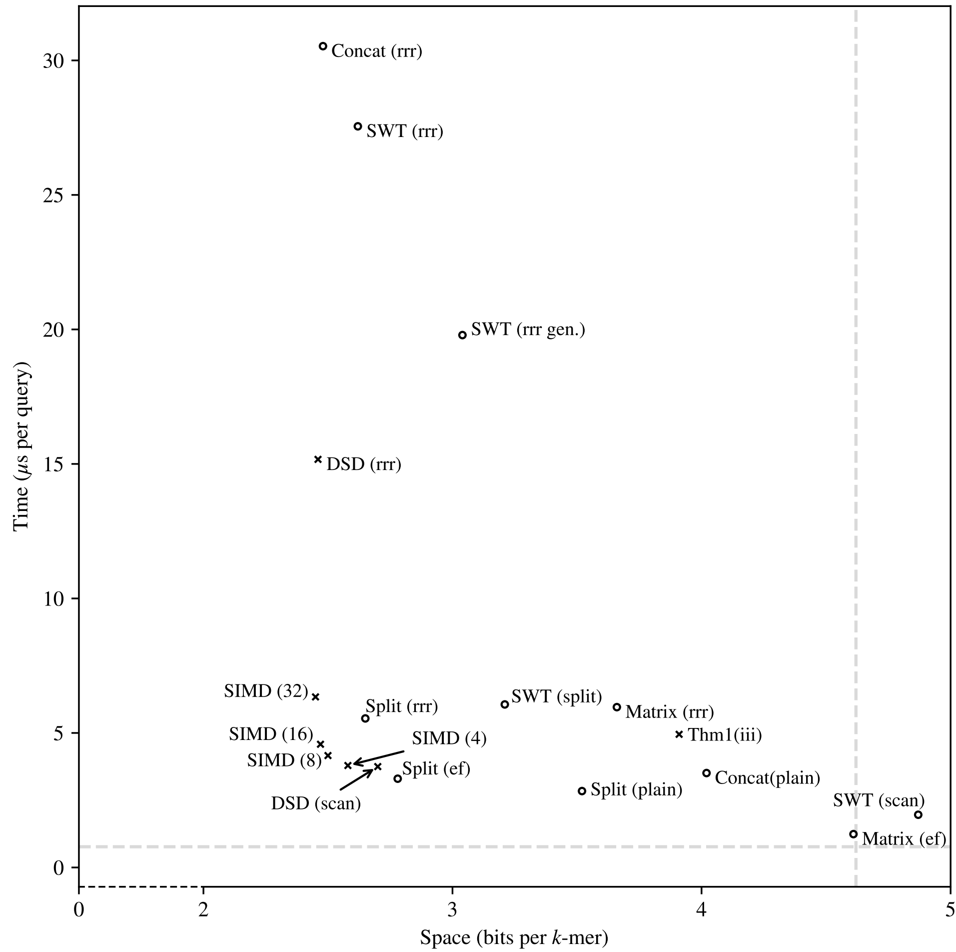


Figure 5.3: Note that the x-axis is truncated. Shows the performance of all data structures for the k-mer search test on the metagenome data set. The two gray lines represent the performance of the benchmark solution "Matrix". The crosses indicate our data structures and the circles indicate the data structures from [ABPV23, APV23].

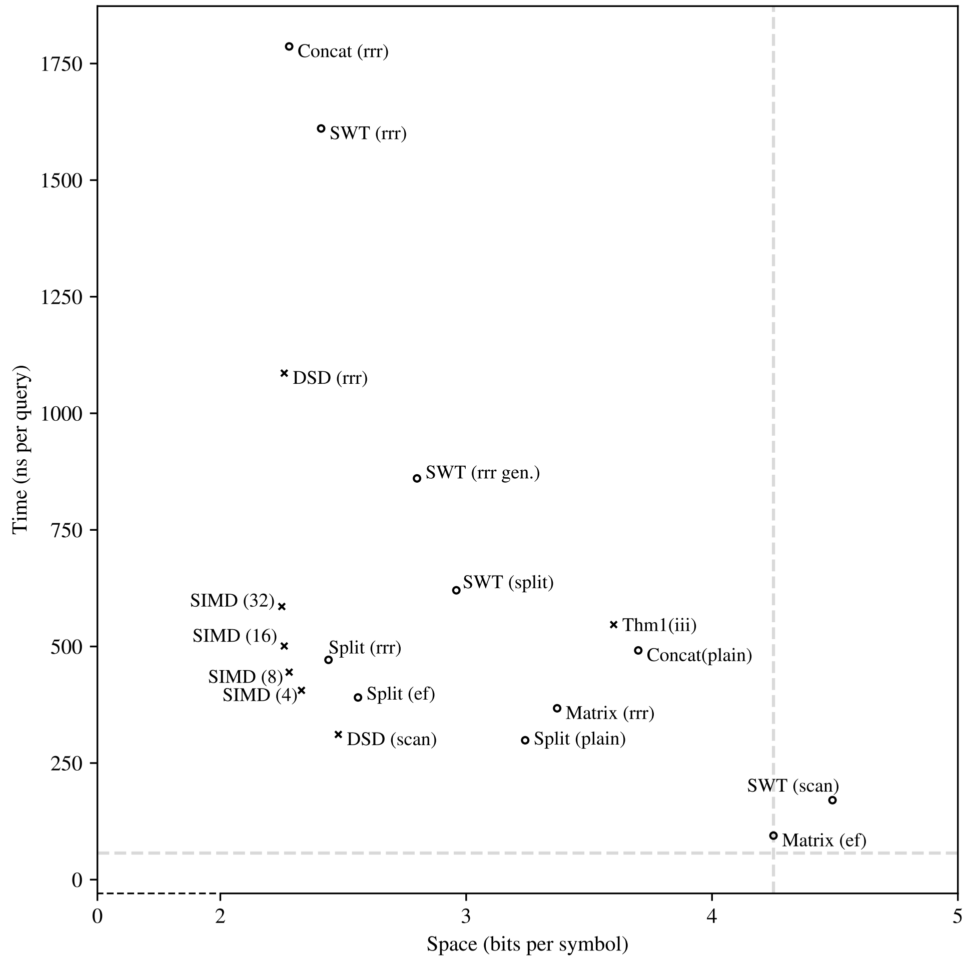


Figure 5.4: Note that the x-axis is truncated. Shows the performance of all data structures for the rank query benchmark. The space is in number of bits per symbol, i.e., bits$/N$. The two gray lines represent the performance of the benchmark solution “Matrix”. The crosses indicate our data structures and the circles indicate the data structures from [ABPV23, APV23].

Data structure	k-mer Queries: E. Coli Query (μs)	E. Coli Space (bpk)	Metagenome Query (μs)	Metagenome Space (bpk)	Subset Rank Queries: E. Coli Query (ns)	E. Coli Space (bps)	Metagenome Query (ns)	Metagenome Space (bps)
Matrix	0.63	4.29	0.77	4.62	38.75	4.26	56.98	4.25
Matrix (ef)	0.92	4.07	1.24	4.61	73.9	4.04	94.4	4.25
Matrix (rrr)	5.28	3.38	5.96	3.66	301.7	3.34	367.45	3.37
DSD (scan)	3.0	2.61	3.75	2.7	210.23	2.57	311.33	2.48
Thm8(iii)	3.87	3.68	4.95	3.91	435.28	3.64	546.89	3.6
DSD (rrr)	13.21	2.38	15.17	2.46	850.99	2.34	1086.11	2.26
SIMD (4)	2.98	2.49	3.79	2.58	289.54	2.42	405.88	2.33
SIMD (8)	3.31	2.42	4.16	2.5	320.53	2.37	444.94	2.28
SIMD (16)	3.67	2.39	4.58	2.47	371.87	2.35	500.97	2.26
SIMD (32)	5.35	2.37	6.34	2.45	447.28	2.34	585.65	2.25
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	2.8
SWT (rrr)	25.33	2.48	27.55	2.62	1384.0	2.45	1610.73	2.41
Concat(plain)	2.53	3.74	3.51	4.02	375.44	3.71	491.55	3.7
Concat(ef)	26.25	2.38	30.53	2.48	1372.2	2.35	1786.65	2.28
Split (plain)	2.28	3.3	2.84	3.52	235.22	3.27	298.87	3.24
Split (ef)	2.71	2.69	3.3	2.78	317.71	2.65	390.65	2.56
Split (rrr)	4.7	2.54	5.54	2.65	393.14	2.51	471.3	2.44

Table 5.2: 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 six groups of data structures, separated by horizontal lines; the variants of the benchmark structure, our reductions, our structure using SIMD, the Subset Wavelet Trees from [ABPV23], the "Concat" reduction from [APV23], and the "Split" reduction from [APV23]. Each group is ordered from fastest to slowest and largest to smallest, except for Thm8(iii) which breaks space order. Each value in the table is the average of five measurements.

Bibliography

[AAB$^+$20] 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*, 175(1-4):41–58, 2020.

[AB20] Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In *Proc. 31st ISAAC*, volume 181, pages 63:1–63:16, 2020.

[AB21] Amihood Amir and Itai Boneh. Dynamic suffix array with sub-linear update time and poly-logarithmic lookup time. *CoRR*, abs/2112.12678, 2021.

[ABPV23] Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi. Subset wavelet trees. In *Proc. 21st SEA*, pages 4:1–4:14, 2023.

[Abr87] Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

[AFG$^+$14] Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing Unbounded-Length Keys in Comparison-Driven Data Structures with Applications to Online Indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014.

[AFGV97] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *Proc. 29th STOC*, pages 540–548, 1997.

[AH18] Daichi Amagata and Takahiro Hara. Mining top-k co-occurrence patterns across multiple streams (extended abstract). In *Proc. 34th ICDE*, pages 1747–1748, 2018.

[AHNR98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *J. Comput. Syst. Sci.*, 57(1):74–93, 1998.

[Ajt88] Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Comb.*, 8(3):235–247, 1988.

[AKLL05] Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th SPIRE*, volume 3772, pages 67–78. Springer, 2005.

[AN08] Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th SODA*, pages 1086–1095, 2008.

[And96] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th FOCS*, pages 135–141, 1996.

[APV23] 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*, pages 225–236, 2023.

[BBPV09] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. 20th SODA*, pages 785–794, 2009.

[BBV10] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *Proc. 17th SPIRE*, pages 159–172, 2010.

[BCF+05] Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. *J. Syst. Softw.*, 78(3):249–256, 2005.

[BCG+14] Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[BCPT15] Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. 23rd ESA*, pages 142–154, 2015.

[BEGV18] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theor. Comput. Sci.*, 713:66–77, 2018.

[Bel12] Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *J. Discrete Algorithms*, 14:91–106, 2012.

[BF02] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[BFK06] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th PODS*, pages 233–242, 2006.

[BGS17] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic indexing for packed strings. In *Proc. 28th CPM*, pages 6:1–6:11, 2017.

[BGS22a] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Partial sums on the ultra-wide word RAM. *Theor. Comput. Sci.*, 905:99–105, 2022. Announced at TAMC 2020.

[BGS22b] Philip Bille, Inge Li Gørtz, and Tord Stordalen. Predecessor on the ultra-wide word RAM. In *Proc. 18th SWAT*, pages 18:1–18:15, 2022.

[BHMS11] Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Trans. Algorithms*, 7(4):52:1–52:27, 2011.

[BI13] Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

[BJ18] Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018.

[BLR+15] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[BN15] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015.

[CIK+17] 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.

[CKL15] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica*, 72(2):450–466, 2015.

[CL06] Joong Hyuk Chang and Won Suk Lee. Finding recently frequent itemsets adaptively over online transactional data streams. *Inf. Syst.*, 31(8):849–869, 2006.

[CM96] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proc. 7th SODA*, pages 383–391, 1996.

[Cor11] Intel Corporation. Intel® advanced vector extensions programming reference. *Intel Corporation*, 2011.

[CRDI07] Thomas Chen, Ram Raghavan, Jason N. Dale, and Eiji Iwata. Cell Broadband engine architecture and its first implementation - A performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007.

[CW79] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Comput. System Sci.*, 18(2):143–154, 1979.

[DadH90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th ICALP*, pages 6–19, 1990.

[DFG+97] Gautam Das, Rudolf Fleischer, Leszek Gasieniec, Dimitrios Gunopulos, and Juha Kärkkäinen. Episode matching. In *Proc. 8th CPM*, pages 12–27, 1997.

[DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[DKM+94] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[DLM02] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Proc. 10th ESA*, pages 348–360, 2002.

[DP13] Michele Dallachiesa and Themis Palpanas. Identifying streaming frequent items in ad hoc time windows. *Data Knowl. Eng.*, 87:66–90, 2013.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS*, pages 137–143, 1997.

[FFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[FG89] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

[FG05] Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. 26th CPM*, pages 160–171, 2005.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FLNS15] Arash Farzan, Alejandro López-Ortiz, Patrick K. Nicholson, and Alejandro Salinger. Algorithms in the ultra-wide word model. In *Proc. 12th TAMC*, pages 335–346, 2015.

[FMMN07] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[Gag16] Travis Gagie. Rank and select operations on sequences. In *Encyclopedia of Algorithms*, pages 1776–1780. 2016.

[GBMP14] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th SEA*, pages 326–337, 2014.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th STOC*, pages 135–143. ACM, 1984.

[GDD+03] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. 3rd ACM IMC*, pages 173–178, 2003.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.

[GMR06] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 15th SODA*, pages 368–373, 2006.

[GORR14] Alexander Golynski, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. Optimal indexes for sparse bit vectors. *Algorithmica*, 69(4):906–924, 2014.

[GRSV13] Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini. Dynamic compressed strings with random access. In *Proc. 40th ICALP*, pages 504–515, 2013.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th FOCS*, pages 8–21, 1978.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proc. 15th STACS*, pages 366–398, 1998.

[Han02] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proc. 34th STOC*, pages 602–608, 2002.

[Han04] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

[HM10] Meng He and J. Ian Munro. Succinct representations of dynamic strings. In *Proc. 17th SPIRE*, pages 334–346, 2010.

[HVDH21] David Harvey and Joris Van Der Hoeven. Integer multiplication in time o(nlog\,n). *Annals of Mathematics*, 193(2):563–617, 2021.

[IMR08] 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.*, 1(4):557–569, 2008.

[ISTA04] Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. *J. Discrete Algorithms*, 2(1):33–51, 2004.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

[JDO+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.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KK22] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. 54th STOC*, pages 1657–1670, 2022.

[KN17] Gregory Kucherov and Yakov Nekrich. Full-Fledged Real-Time Indexing for Constant Size Alphabets. *Algorithmica*, 79(2):387–400, 2017.

[Kop12] Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *53rd FOCS*, pages 283–292, 2012.

[Kos94] S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th STOC*, pages 310–316, 1994.

[KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[Lar99] N. Jesper Larsson. *Structures of String Matching and Data Compression.* PhD thesis, Lund University, Sweden, 1999.

[LCK14] Yongsub Lim, Jihoon Choi, and U Kang. Fast, accurate, and space-efficient tracking of time-weighted frequent items from data streams. In *Proc. 23rd CIKM*, pages 1109–1118, 2014.

[LCWC05] Chih-Hsiang Lin, Ding-Ying Chiu, Yi-Hung Wu, and Arbee L. P. Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *Proc. 5th SDM*, pages 68–79, 2005.

[LL09] Hua-Fu Li and Suh-Yin Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Syst. Appl.*, 36(2):1466–1477, 2009.

[LMu+99] R. Leben, M. Miletic, M. Špegel, A. Torst, A. Brodnik, and K. Karlsson. Design of high performance memory module on PC100. In *Proc. Electrotechnical and Computer Science Conference (ERK)*, pages 75–78, 1999.

[LP12] Kasper Green Larsen and Rasmus Pagh. I/O-efficient data structures for colored range and prefix reporting. In *Proc. 23rd SODA*, pages 583–592, 2012.

[Mil94] Peter Bro Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Proc. 26th STOC*, pages 625–634, 1994.

[MN07] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007.

[MNSW98] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *J. Comput. Syst. Sci.*, 57(1):37–49, 1998.

[MPRZ21] Danyang Ma, Simon J Puglisi, Rajeev Raman, and Bella Zhukova. On elias-fano for rank queries in fm-indexes. In *Proc. DCC, 2021*, pages 223–232, 2021.

[MTZ08] Barzan Mozafari, Hetal Thakkar, and Carlo Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *Proc. 24th ICDE*, pages 179–188, 2008.

[NAIP03] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003.

[NN14] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.

[NR20] Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5):105:1–105:35, 2020.

[NS14] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.

[OCGO96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[OS07] Daisuke Okanohara and Kunihiko Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. 9th ALENEX*, 2007.

[PNB17] Alberto Ordóñez Pereira, Gonzalo Navarro, and Nieves R. Brisaboa. Grammar compressed sequences with rank/select support. *J. Discrete Algorithms*, 43:54–71, 2017.

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th STOC*, pages 232–240, 2006.

[PT07] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proc. 18th SODA*, pages 555–564, 2007.

[PT14] Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175, 2014.

[Rei13] James Reinders. Intel® AVX-512 instructions. Intel® Corporation, 2013.

[RRS07] 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*, 3(4):43, 2007.

[SBB+17] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[SBD+21] Joshua Sobel, Noah Bertram, Chen Ding, Fatemeh Nargesian, and Daniel Gildea. AWLCO: all-window length co-occurrence. In *Proc. 32nd CPM*, LIPIcs, pages 24:1–24:21, 2021.

[SD08] Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *Proc. 15th SPIRE*, pages 109–120, 2008.

[Sen05] M Senft. Suffix tree for a sliding window: An overview. In *Proc. WDS*, volume 5, pages 41–46, 2005.

[SLLM10] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *J. Discrete Algorithms*, 8(2):241–257, 2010.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary trees. In *Proc. 15th ACM*, pages 235–245, 1983.

[SV08] Pranab Sen and Srinivasan Venkatesh. Lower bounds for predecessor searching in the cell probe model. *J. Comput. Syst. Sci.*, 74(3):364–385, 2008.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6(3):80–82, 1977.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.

[YYL+15] Ziqiang Yu, Xiaohui Yu, Yang Liu, Wenzhu Li, and Jian Pei. Mining frequent co-occurrence patterns across multiple data streams. In *Proc. 1th EDBT*, pages 73–84, 2015.