Space-Efficient Indexes for Uncertain Strings

Esteban Gabory[1], Chang Liu[2], Grigorios Loukides[3], Solon P. Pissis[1], and Wiktor Zuba[1]

[1]CWI, Amsterdam, The Netherlands
[2]Zhejiang University, Zhejiang, China
[3]King's College London, London, UK

***Abstract*—Strings in the real world are often encoded with some level of uncertainty, for example, due to: unreliable data measurements; flexible sequence modeling; or noise introduced for privacy protection. In the *character-level uncertainty model*, an *uncertain string* X of length n on an alphabet Σ is a sequence of n probability distributions over Σ. Given an uncertain string X and a weight threshold $\frac{1}{z} \in (0,1]$, we say that pattern P occurs in X at position i, if the product of probabilities of the letters of P at positions $i, \ldots, i+|P|-1$ is at least $\frac{1}{z}$. While indexing standard strings for online pattern searches can be performed in linear time and space, indexing uncertain strings is much more challenging. Specifically, the state-of-the-art index for uncertain strings has $\Theta(nz)$ size, requires $\Theta(nz)$ time and space to be constructed, and answers pattern matching queries in the optimal $\mathcal{O}(m+\textsc{Occ})$ time, where m is the length of P and $\textsc{Occ}$ is the total number of occurrences of P in X. For large n and (moderate) z values, this index is completely impractical to construct, which outweighs the benefit of the supported optimal pattern matching queries. We were thus motivated to design a space-efficient index at the expense of slower yet competitive pattern matching queries. We show that when we have at hand a lower bound ℓ on the length of the supported pattern queries, as is often the case in real-world applications, we can slash the index size *and* the construction space roughly by ℓ. In particular, we propose an index of $\mathcal{O}(n+\frac{nz}{\ell}\log z)$ expected size, which can be constructed using $\mathcal{O}(n+\frac{nz}{\ell}\log z)$ expected space, and supports very fast pattern matching queries in expectation, for patterns of length $m \geq \ell$. We have implemented and experimentally evaluated several versions of our index. The best-performing version of our index is up to *two orders of magnitude smaller* than the state of the art in terms of *both* index size and construction space, while offering very competitive query times and construction times.**

I. Introduction

A large portion of the data feeding real-world database systems, including bioinformatics systems [80], Enterprise Resource Planning (ERP) systems [73], or Business Intelligence (BI) systems [93], is textual; namely, these data are finite *sequences* of letters over some alphabet (also known as *strings*). This happens because strings can easily encode data arising from different sources such as: sequences of nucleotides read by DNA sequencers (e.g., short or long DNA reads); natural language text generated by humans (e.g., description or comment fields); identifiers generated by software (e.g., URLs, email addresses, or IP addresses); or discretized measurements generated by sensors (e.g., EEG or EMG data).

Given the ever increasing size of string data, it is crucial to represent them in a concise form but also to *simultaneously* allow efficient pattern searches. This goal is formalized by the classic *text indexing* problem [29]: preprocess a string T of length n over an alphabet Σ of size σ, known as the *text*, into a data structure that supports pattern matching queries; i.e., report the set of all $\textsc{Occ}$ positions in T where an occurrence of a string P, known as the *pattern*, begins.

In text indexing we are usually interested in four measures of efficiency [16], [56]: **(i)** How much space does the final index occupy for a string T of length n (the *index size*)? **(ii)** How fast can we answer a query P of length m (the *query time*)? **(iii)** How much working space do we need to construct the index (the *construction space*)? **(iv)** How fast can we construct the index (the *construction time*)? For example, the classic indexing solution of *suffix tree* [94] has index size $\mathcal{O}(n)$, optimal query time $\mathcal{O}(m+\textsc{Occ})$, where $\textsc{Occ}$ is the size of the output, construction space $\mathcal{O}(n)$, and construction time $\mathcal{O}(n)$ [33]. Nowadays, as the data volume grows rapidly, a lot of works are devoted to obtaining space-query time trade-offs [42], [34]. Such works propose data structures that occupy $\mathcal{O}(n\log\sigma)$ bits of space, instead of the $\Theta(n\log n)$ bits occupied by suffix trees in any case, at the expense of a factor of $\mathcal{O}(\log^{\epsilon} n)$ penalty in the query time, where $\epsilon > 0$ is an arbitrary predefined constant.

In the real world, strings are often encoded with some level of uncertainty; for example, due to: (i) imprecise, incomplete or unreliable data measurements, such as sensor measurements, RFID measurements or trajectory measurements [10]; (ii) deliberate flexible sequence modeling, such as the representation of a *pangenome*, that is, a collection of closely-related genomes to be analyzed together [89]; or (iii) the existence of confidential information in a dataset which has been distorted deliberately for privacy protection [9].

While there are many practical solutions for text indexing [70], [34], [40], [37], [16] and also for answering different types of queries on various types of uncertain data (see Section VI), *practical indexing schemes* for uncertain strings are rather undeveloped. In response, our work makes an important step towards developing such practical space-efficient indexes.

A. Our Data Model and Motivation

We use the standard *character-level uncertainty model* [48]. An *uncertain string* (or *weighted string*) X of length n on an alphabet Σ is a sequence of n sets. Every $X[i]$, for all $1 \leq i \leq n$, is a set of $|\Sigma|$ ordered pairs $(\alpha, p_i(\alpha))$, where $\alpha \in \Sigma$ and $p_i(\alpha)$ is the probability of having the letter α at position i. Table I shows an example of a weighted string X for $n = 6$ and $\Sigma = \{\texttt{A}, \texttt{B}\}$, represented as a $|\Sigma| \times n$ matrix.

TABLE I: Example of a weighted string X.

	1	2	3	4	5	6
A	1	1/2	3/4	4/5	1/2	1/4
B	0	1/2	1/4	1/5	1/2	3/4

The data model of [48] has been employed by many works [12], [59], [76], [38], [65], [57]. In bioinformatics, for example, weighted strings are known as *position weight matrices* [54]. As in these works, we define the *occurrence probability* of pattern $P = \texttt{ABA}$ at position 3 in X of Table I as $3/4 \cdot 1/5 \cdot 1/2 = 3/40$ (shown in Table I in red).

The indexing problem on weighted strings can thus be naturally defined as follows [13]: Given a weighted string X of length n on an alphabet Σ of size σ and a weight threshold $\frac{1}{z} \in (0, 1]$, preprocess X into a data structure (an index) that supports pattern matching queries; i.e., report the set of all positions in X where P occurs with probability at least $\frac{1}{z}$.

The indexing problem on weighted strings has attracted a lot of attention by the theory community [46], [13], [20], [18], [20], [17], [22], culminating in the following result:

Theorem 1 ([18], [17]). *For any weighted string of length n and any weight threshold $\frac{1}{z}$, we can construct an index of size $\Theta(nz)$ in $\mathcal{O}(nz)$ time and space supporting pattern matching queries in $\mathcal{O}(m + \text{OCC})$ time, for any pattern of length m.*[1]

Although Theorem 1 is very appealing from a theoretical perspective—due to the linear dependency on z and the linear dependency on n—from a practical perspective, $\Theta(nz)$ size and construction space are *prohibitive* for large strings. Imagine that we have an input weighted string of length $n = 10^9$, that $z = 100$, and that the constant in $\Theta(nz)$ is something small, like 20, which is in line with the state of the art [22]. Then we need 2TBs of RAM to store the index for an input of 1GB! *We were thus motivated to seek space-query time trade-offs for indexing weighted strings.* In particular, we seek conditions under which we can have indexes of size smaller than $\Theta(nz)$. Ideally, we would also like to construct these indexes using smaller than $\Theta(nz)$ space. We show that this is possible, both in theory and in practice, when a lower bound ℓ on the length m of any queried pattern is known in advance, which is arguably a reasonable assumption in applications. For instance, in bioinformatics [47], [95], [66], the length of sequencing reads (patterns) ranges from a few hundreds to 30,000 [66]. Even when at most k errors must be accommodated for matching, at least one out of $k+1$ fragments must be matched exactly. In natural language processing, the queried patterns can also be long [92]. Examples of such patterns are queries in question answering systems [43], *description queries* in TREC datasets [19], [15], and representative phrases in documents [71]. Similarly, a query pattern can be long when it encodes an entire document (e.g., a webpage in the context of deduplication [44]), or machine-generated messages [49].

B. Our Techniques and Results

In [18], Barton et al. showed that for any weighted string X of length n, and any weight threshold $\frac{1}{z}$, one can construct a family $\mathcal{S}$ of $\lfloor z \rfloor$ standard strings, each of length n, so that a pattern P occurs in X at position i with probability p only if P occurs at position i in $\lfloor p \cdot z \rfloor$ strings from $\mathcal{S}$. The authors have then shown that by indexing $\mathcal{S}$ using a modified version of suffix trees [94], we can arrive at an index of total size $\Theta(nz)$ supporting queries in the optimal time $\mathcal{O}(m + \text{OCC})$. The resulting index is known as the *weighted suffix tree* (WST). An array-based, more space-efficient, version was also presented in [22]; it is known as the *weighted suffix array* (WSA). WST and WSA are the state of the art for indexing weighted strings.

Here, we first show how to slash the size of both WST and WSA roughly by ℓ, while still supporting very fast queries in expectation for any pattern P of length $m \geq \ell$, by combining the *minimizers* sampling mechanism [79], [82], several combinatorial and probabilistic arguments, and a geometric data structure (2D grid) [69]. Our technique still requires us to first construct the family $\mathcal{S}$ of the $\lfloor z \rfloor$ strings, which in any case gives an index with $\Theta(nz)$ construction space. To circumvent this, we develop a highly non-trivial algorithm for constructing the *same* index *without generating $\mathcal{S}$ explicitly*. The algorithm samples an implicit representation of $\mathcal{S}$ using minimizers outputting the final index directly.

Our main contributions are summarized as follows:

1) We show that for any weighted string X of length n over an alphabet Σ, a weight threshold $\frac{1}{z}$, and any integer $\ell > 0$, after $\mathcal{O}(nz)$ construction time using $\Theta(nz)$ construction space, we can construct an index of $\mathcal{O}(n + \frac{nz}{\ell}\log z)$ expected size to report all occurrences of a pattern P of length $m \geq \ell$ in $\mathcal{O}(m + \frac{nz}{|\Sigma|^m}\log\frac{nz}{\ell})$ expected query time. In particular, when $m \geq \log_\sigma nz$ (recall that $m \geq \ell$), we get $\mathcal{O}(m + \log\frac{nz}{\ell})$ query time and an index of size less than $\Theta(nz)$. The bounds we prove are in expectation because minimizers are usually analyzed in the average-case model [79], [82]. Similar notions with worst-case guarantees exist [55] but, as they are not practical [16], we resort to employing minimizers.
2) Although the above-mentioned index has very desirable size and query time, we still need $\Theta(nz)$ space to construct it. We were thus motivated to design a space-efficient construction algorithm for this index. In fact, we show that this index can be constructed in expected time $\mathcal{O}(nz\log\ell + \frac{nz}{\ell}\log\frac{nz}{\ell}\log z)$ and space $\mathcal{O}(n + \frac{nz}{\ell}\log z)$. This is the most technically involved result of the paper.
3) Following the different paradigms of suffix trees [94] and suffix arrays [51] in the classic setting of standard strings, we have implemented tree and array-based versions of our index underlying Contributions 1 and 2. The results show that our indexes are up to *two orders of magnitude smaller* than the state of the art in terms of *both* index size

[1] We can safely make the assumption that $\sigma \leq z$. If $\sigma > z$, we construct a new string X_z from X of total size $\lfloor z \rfloor n$, because there can be no more than $\lfloor z \rfloor$ letters in $X[i], i \in [1, n]$, with an occurrence probability at least $\frac{1}{z}$. We then index $X[i]$ using a linked-list or a hash table for a sparse representation.

and construction space. They also show that our array-based indexes outperform the tree-based ones, offering *very competitive query times and construction times* to those of the state-of-the-art indexes. For example, for indexing a collection of $1,432$ bacterial samples, with $\ell = 256$ and $z = 1024$, which are reasonable in applications, our space-efficient index has size 640MBs and needs only 772MBs of memory to be constructed, while WSA has size 7GBs and needs 32GBs of memory to be constructed! Furthermore, compared to WSA, our space-efficient index takes 44% less time to be constructed and its query time is 80 microseconds on average (over about 1.9M queries), while that for WSA is 81 microseconds.

C. Paper Organization

In Section II, we provide the necessary definitions and notation as well as some previous results that we make use of. In Section III, we provide the full description of our new index. In Section IV, we present the space-efficient algorithm for constructing our index. In Section V, we present a simple, more practical algorithm for querying our index. In Section VI, we discuss related work. Finally, in Section VII, we provide an extensive experimental evaluation of our algorithms.

II. Preliminaries and Problem Definition

Strings. An *alphabet* Σ of size $\sigma = |\Sigma|$ is a nonempty set of elements called *letters*. By Σ^k we denote the set of all length-k strings over Σ. By ε we denote the *empty string* of length 0. For a *string* $S = S[1] \cdots S[n]$ over Σ, by $n = |S|$ we denote its *length*. The fragment $S[i\,..\,j]$ of S is an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We also say that P occurs at *position* i in S. A *prefix* of S is a substring of S of the form $S[1\,..\,j]$ and a *suffix* of S is a substring of S of the form $S[i\,..\,n]$. Given a string $S = S[1] \cdots S[n]$, its *reverse* is the string $S^r = S[n] \cdots S[1]$. For any two strings S_1 and S_2 of the same length, we define their *Hamming distance* $d_H(S_1, S_2)$ as their total number of mismatching positions.

Sampling. Given a fixed pair of positive integers ℓ, k s.t. $\ell \geq k$, we call a function $f : \Sigma^\ell \to [1, \ell - k + 1]$ that selects the starting position of a length-k fragment, for any string of length ℓ, an *(ℓ, k)-local scheme*. We call the set $\mathcal{M}_f(S) = \{i + f(S([i\,..\,i + \ell - 1])) - 1 \mid 1 \leq i \leq |S| - \ell + 1\}$, for an *$(\ell, k)$-local scheme* f on a string S, *the set of selected indices*. The *specific density* of f on S is the value $|\mathcal{M}_f(S)|/|S|$, and the *density* of f is the expected specific density on a sufficiently long random string (with letters chosen independently at random). An *(ℓ, k)-minimizer scheme* is an (ℓ, k)-local scheme that selects the position of the leftmost occurrence of the smallest length-k substring, for a fixed k and a fixed order on Σ^k. In that case, we call *minimizers* the selected indices [79], [82]. The minimizer scheme can be based on a lexicographic order on Σ^k. As an example of this scheme, let $S =$ ABAABB, $\ell = 4$, and $k = 2$. We obtain $\mathcal{M}_f(S) = \{3\}$ as $S[3\,..\,4] =$ AA is the lexiographically smallest length-2 substring in every window of S of length $\ell = 4$. The minimizer scheme can also be specified by a hash function, e.g. Karp-Rabin fingerprints [52].

Lemma 2 ([100]). *The density of an (ℓ, k)-minimizer scheme over an alphabet Σ with $k \geq \log_{|\Sigma|} \ell + c$ is $\mathcal{O}(\frac{1}{\ell})$, for some $c = \mathcal{O}(1)$.*

Weighted Strings. A *weighted string* X of length n on an alphabet Σ is a sequence of n sets. Every $X[i]$, for all $1 \leq i \leq n$, is a set of $|\Sigma|$ ordered pairs $(\alpha, p_i(\alpha))$, where $\alpha \in \Sigma$ is a letter and $p_i(\alpha)$ is the probability of having α at position i of X. Formally, $X[i] = \{(\alpha, p_i(\alpha)) \mid \forall\alpha \in \Sigma,\ p_i(\alpha) \in [0, 1],$ and $\sum_{\alpha\in\Sigma} p_i(\alpha) = 1\}$. A letter α *occurs* at position i of a weighted string X if and only if the *occurrence probability* of α at position i, $p_i(\alpha)$, is greater than 0. A string U of length m is a *factor* of a weighted string X if and only if it occurs at some starting position i with *occurrence probability* $\mathbb{P}(X[i\,..\,i + m - 1] = U) = \Pi_{j=1}^{m} p_{j+i-1}(U[j]) > 0$. Given a *weight threshold* $p \in (0, 1]$, factor U is *valid* or equivalently U has a valid occurrence in X, if it occurs at starting position i and if $\mathbb{P}(X[i\,..\,i + m - 1] = U) \geq p$. For a weighted string X, a pattern P, and a weight threshold $p \in (0, 1]$, $\text{Occ}_p(P, X)$ is the set of starting positions of valid occurrences of P in X. String U is a *solid factor* of X if it has a valid occurrence in X for some p; U is *maximal* at position i of X if U is a solid factor of X starting at position i and no string $U' = U\alpha$, for any $\alpha \in \Sigma$, is a solid factor starting at i. Given a weighted string X, we call a *heavy string* H_X of X a string defined such that $H_X[i]$ is the letter having a largest probability in X at position i (ties are broken arbitrarily). For example, let X be the weighted string of Table I; a heavy string of X is $H_X =$ ABAAAB (the tie at position 2 is broken for B and the tie at position 5 is broken for A).

A *property* Π of a string S is a hereditary collection of integer intervals [2] contained in $[1, n]$. For simplicity, we represent every property Π with an array $\pi[1\,..\,|S|]$ such that the longest interval $I \in \Pi$ starting at position i is $[i, \pi[i]]$. Observe that π can be an arbitrary array satisfying $\pi[i] \in [i - 1, n]$, and $\pi[1] \leq \pi[2] \leq \cdots \leq \pi[n]$. For a string P, by $\text{Occ}_\pi(P, S)$ we denote the set of occurrences i of P in S such that $i + |P| - 1 \leq \pi[i]$. For example, let (S_2, π_2) from Table II be a string-property pair. Then $P =$ AAA occurs at position $i = 3$ because $i + |P| - 1 = 3 + 3 - 1 \leq \pi_2[3] = 5$.

Let us consider an indexed family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{k}$ of strings S_j with properties π_j. For a string P and an index i, by $\text{Count}_{\mathcal{S}}(P, i) = |\{j \mid i \in \text{Occ}_{\pi_j}(P, S_j)\}|$ we denote the total number of occurrences of P at position i in the strings $S_1, \ldots, S_k$ of $\mathcal{S}$ that respect the properties. We say that an indexed family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{z}$ is a *z-estimation* of a weighted string X of length n if and only if, for every string P and position $i \in [1, n]$, $\text{Count}_{\mathcal{S}}(P, i) = \lfloor \mathbb{P}(X[i\,..\,i+|P|-1] = P) \cdot z \rfloor$. The following result has been shown by Barton et al.:

Theorem 3 ([17]). *For any weighted string X of length n and any weight threshold $\frac{1}{z}$, X admits a z-estimation of total size $\Theta(nz)$ that can be constructed in $\mathcal{O}(nz)$ time and space.*

[2] A collection that contains all the subintervals of its elements.

TABLE II: A 4-estimation of X from Table I.

i	1	2	3	4	5	6
S_1	A	A	A	A	A	A
π_1	2	2	3	4	5	6
S_2	$\underline{\texttt{A}}$	$\underline{\texttt{A}}$	$\underline{\texttt{A}}$	$\underline{\texttt{A}}$	A	B
π_2	4	4	5	6	6	6
S_3	$\underline{\texttt{A}}$	B	$\underline{\texttt{A}}$	$\underline{\texttt{A}}$	B	B
π_3	4	4	5	6	6	6
S_4	A	B	B	B	B	B
π_4	2	2	3	3	5	6

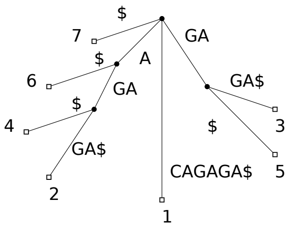


Fig. 1: Suffix tree of $S =$ CAGAGA\$.

Example 1. For $\frac{1}{z} = \frac{1}{4}$, the weighted string X in Table I admits the 4-estimation $\mathcal{S}$ in Table II, given by Theorem 3.

For pattern $P =$ AB and S_3, we have that $\text{Occ}_{\pi_3}(P, S_3) = \{1, 4\}$ because P occurs at position 1, with $1 + |P| - 1 \leq \pi_3[1] = 4$, and at position 4, with $4 + |P| - 1 \leq \pi_3[4] = 6$.

For pattern $P =$ AB and $i = 1$, we have that $\mathbb{P}(X[i\,..\,i + |P| - 1] = P) = 1 \cdot 1/2 = 1/2$ and so P occurs in $\text{Count}_{\mathcal{S}}(P, i) = \lfloor \mathbb{P}(X[i\,..\,i+|P|-1] = P) \cdot z \rfloor = \lfloor (1/2) \cdot 4 \rfloor = 2$ strings of the z-estimation at position 1.

We construct the set of (lexicographic) minimizers that respect the property, for $\ell = 3$ and $k = 2$, for every S_j, $j \in [1, 4]$, from $\mathcal{S}$. We underline the positions of the minimizers. Note that we have selected no minimizer in S_1 or S_4 as they have no solid factor of length 3. $\square$

Problem Definition. We study the following problem:

> ℓ-WEIGHTED INDEXING
>
> **Input:** A weighted string X of length n over an alphabet Σ, a weight threshold $\frac{1}{z} \in (0, 1]$, and an integer $\ell > 0$.
>
> **Query:** For any string P of length $m \geq \ell$, report all elements of $\text{Occ}_{\frac{1}{z}}(P, X)$.

Suffix Tree. The classic indexing solution for standard (not weighted) strings is the suffix tree. Given a set $\mathcal{F}$ of strings, the *compacted trie* of these strings is the trie obtained by compressing each path of nodes of degree one in the trie of the strings in $\mathcal{F}$, which takes $\mathcal{O}(|\mathcal{F}|)$ space [72]. Each edge in the compacted trie has a label represented as a fragment of a string in $\mathcal{F}$. The *suffix tree* $\mathsf{ST}(S)$ is the compacted trie of the suffixes of S [94]. Assuming S ends with a unique terminating letter \$, every suffix $S[i\,..\,n]$ is represented by a leaf decorated by index i; see Fig. 1. The set of indices stored at the leaf nodes in the subtree rooted at node v is the *leaf-list* of v, and we denote it by $LL(v)$. Each edge in $\mathsf{ST}(S)$ is labeled with a nonempty substring of S such that the path from the root to the leaf annotated with index i spells the suffix $S[i\,..\,n]$. The substring of S spelled by the path from the root to node v is the *path-label* of v, and we denote it by $L(v)$. Given any pattern $P[1\,..\,m]$, $\mathsf{ST}(S)$ allows us to report all OCC occurrences of P in S using only $\mathcal{O}(m \log \sigma + \text{OCC})$ operations. We simply spell P from the root of $\mathsf{ST}(S)$ (to access edges by the first letter of their label, we use binary search) until we arrive (if possible) at the first node v such that P is a prefix of $L(v)$. Then all OCC occurrences (starting positions) of P in S are $LL(v)$. The suffix tree occupies $\Theta(n)$ space and it can be constructed in $\mathcal{O}(n)$ time for an integer alphabet [33]. To improve the query time to the optimal $\mathcal{O}(m + \text{OCC})$ we use randomization to construct a perfect hash table in linear time [35] for accessing edges by the first letter of their label in constant time.

III. THE NEW INDEX: MINIMIZER-BASED WST

In this section, we describe our new index for solving ℓ-WEIGHTED INDEXING and the underlying data structures that we employ to construct it. We assume read-only access to X but we can also discard X at the end of this construction.

Main Idea. We start the index construction by building the z-estimation of X, whose total size is $\Theta(nz)$. We then use minimizers sampling to select $\mathcal{O}(\frac{nz}{\ell})$ positions of the z-estimation, where ℓ is a predetermined lower bound on the length of the supported queries. Next, we construct two trees, called *minimizer solid factor trees*: (1) the compacted trie *of all suffixes* of the solid factors in the z-estimation starting at the minimizer positions, and (2) the compacted trie *of all the reversed prefixes* of the solid factors in the z-estimation ending at the minimizer positions. After that, we pair up the leaf nodes corresponding to the same minimizer position, from one of these trees to the other, using a 2D grid for *range reporting* [21]. To reduce the index size, we discard the z-estimation using a combinatorial observation that allows us to store only $\mathcal{O}(\log z)$ information per minimizer position. This results in an index of expected total size $\mathcal{O}(n + \frac{nz \log z}{\ell})$.

Finally, we show how to query the index efficiently, for any pattern of length $m \geq \ell$, by using a probabilistic argument on the number of expected points returned by the 2D grid.

Minimizer Solid Factor Trees. Let us fix a weighted string X of length n over an alphabet Σ and a weight threshold $\frac{1}{z}$. We first define a *forward solid factor tree* (resp. *backward solid factor tree*) for X as the suffix tree for the set of maximal solid factors (resp. the set of reversed solid factors) in X. By Theorem 3, we know that each such solid factor appears in a z-estimation of size $\mathcal{O}(nz)$, and therefore both the solid factor trees have size $\mathcal{O}(nz)$ as well. This argument also gives a method to construct the solid factor trees [17].

We adapt the solid factor trees to make them more space-efficient for ℓ-WEIGHTED INDEXING by employing minimizer schemes. Let us fix ℓ, k and an (ℓ, k)-minimizer scheme f by employing Lemma 2. In particular, we assume throughout that ℓ and k are chosen so that f has density $\mathcal{O}(\frac{1}{\ell})$. We then construct a z-estimation $\mathcal{S} = (S_j, \pi_i)_{j=1}^{\lfloor z \rfloor}$ of X using Theorem 3 and compute the set $\mathcal{M}_X$ of minimizers from $\mathcal{S}$ respecting the property; namely for $S_j \in \mathcal{S}$ we compute $f(S_j[i\,..\,i + \ell - 1])$ if and only if $i + \ell - 1 \leq \pi_j[i]$.

We represent each minimizer in $\mathcal{M}_X$ by a pair (i, j), where i is the minimizer position in the string $S_j \in \mathcal{S}$. In the following, we consider $\mathcal{M}_X$ fixed with $|\mathcal{M}_X| = \mathcal{O}(\frac{nz}{\ell})$, as by Lemma 2 there are in expectation $\mathcal{O}(\frac{nz}{\ell})$ minimizers in $\mathcal{S}$.

Based on $\mathcal{M}_X$, we define a *minimizer* forward (resp. backward) solid factor tree as a compacted trie containing suffixes of solid factors (resp. of reversed solid factors) *starting* at position i from a string $S_j \in \mathcal{S}$ with $(i,j) \in \mathcal{M}_X$. Each leaf has a label $(i,j) \in \mathcal{M}_X$ associated to the corresponding suffix. If one same suffix corresponds to several such labels (it occurs at several minimizers from $\mathcal{S}$), we add one copy of the leaf for each such label. Since $|\mathcal{M}_X| = \mathcal{O}(\frac{nz}{\ell})$, the minimizer solid factor trees contain $\mathcal{O}(\frac{nz}{\ell})$ leaves, and therefore nodes.

Still the size of the z-estimation $\mathcal{S}$ is, by definition, always $\Theta(nz)$, which makes the total size of the index $\mathcal{O}(\frac{nz}{\ell}) + \Theta(nz) = \Theta(nz)$. We avoid this by employing the following simple yet crucial combinatorial observation [58]:

Lemma 4 ([58])**.** *Let H_X be a heavy string of X. For a weight threshold $\frac{1}{z}$ and any solid factor U starting at position i and ending at position j of X, $d_H(U, H_X[i\,..\,j]) \leq \log_2 z$ holds.*

Indeed, we directly get the following result, which allows us to avoid storing the z-estimation $\mathcal{S}$ explicitly.

Corollary 5. *Every solid factor of a weighted string X for a weight threshold of $\frac{1}{z}$ can be characterized by an interval of the heavy string H_X plus the information of at most $\log_2 z$ single mismatches. The minimizer solid factor tree can be implemented as a compacted trie whose edges store only that information, which takes $\mathcal{O}(\log z)$ extra space per edge.*

We apply Corollary 5 to obtain Lemma 6. Based on this lemma, we construct the minimizer solid factor trees for X.

Lemma 6. *The minimizer solid factor trees can be constructed in $\mathcal{O}(nz)$ time using $\mathcal{O}(nz)$ space. Each tree has $\mathcal{O}(\frac{nz}{\ell})$ expected nodes and its expected total size is $\mathcal{O}(\frac{nz}{\ell}\log z)$.* [3]

Proof. We apply Theorem 3 to construct a z-estimation for X in $\mathcal{O}(nz)$ time and space. The set of minimizers of any string can be computed in linear time [67]. Thus, computing $\mathcal{M}_X$ for the z-estimation takes $\mathcal{O}(nz)$ time. The compacted trie of any collection of substrings of a string can be constructed in linear time in the length of the string [22], [53], and thus the minimizer solid factor trees can be constructed in $\mathcal{O}(nz)$ time using $\mathcal{O}(nz)$ space. The number of nodes and the total size of the trees follow from Lemma 2 and Corollary 5. □

Exploiting 2D Range Reporting. We explain how to employ a geometric data structure to pair up the leaf nodes corresponding to the same minimizer position from one of the minimizer solid factor tree we constructed above to the other.

Let us write $\mathcal{T}_{\text{suff}}$ (resp. $\mathcal{T}_{\text{pref}}$) for the forward (resp. backward) minimizer solid factor tree. We fix an order on the leaves of both $\mathcal{T}_{\text{pref}}$ and $\mathcal{T}_{\text{suff}}$, such that for any node in one of the trees, the set of its descendant leaves forms an interval. This is possible, for example, by sorting the strings corresponding to the leaves in lexicographical order. Via this ordering, we can consider a pair of leaves from $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$ as a point of a 2D data structure, which we call *the grid*.

We start by some definitions: (1) Given a string P, we denote by $\mathcal{I}_{\text{suff}}(P)$ (resp. $\mathcal{I}_{\text{pref}}(P)$) the (possibly empty) interval of leaves in the subtree obtained by reading P in $\mathcal{T}_{\text{suff}}$ (resp. $\mathcal{T}_{\text{pref}}$). (2) We denote by $\mathcal{N}$ the set of all those points corresponding to pairs of leaves from $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$ with identical labels. Each point in $\mathcal{N}$ corresponds to a given minimizer $(i,j) \in \mathcal{M}_X$, and a pair of maximal solid factors in X that can be read from i, both right-wise and left-wise. (3) Given a pair P_1, P_2 of strings, we denote by $\mathcal{N}(P_1, P_2)$ the intersection of the set $\mathcal{N}$ with the rectangle $\mathcal{I}_{\text{suff}}(P_1) \times \mathcal{I}_{\text{pref}}(P_2)$. (4) Given a string P of length $m \geq \ell$, such that $f(P[1\,..\,\ell]) = \mu$, we denote $\mathcal{N}(P[\mu\,..\,m], (P[1\,..\,\mu])^r)$ by $\mathcal{N}(P)$. We prove the following.

Lemma 7. *For any pattern P of length m, with $n \geq m \geq \ell$, if P is a solid factor in X, then $\mathcal{N}(P)$ is nonempty. In particular, if P has a valid occurrence in X starting at position k then $\mathcal{N}(P)$ contains at least one point having label $(k - 1 + f(P[1\,..\,\ell]), j)$ for some $j \in [1, z]$.*

Proof. Let P be such a pattern, which is a solid factor in X at position k. By definition of a z-estimation, we know that P occurs at position k in some $S_j \in \mathcal{S}$. The minimizer computed for position k of S is $i = k - 1 + \mu$ with $\mu = f(P[1\,..\,\ell])$, since $S_j[k\,..\,k+\ell-1] = P[1\,..\,\ell]$. Therefore, the tree $\mathcal{T}_{\text{suff}}$ (resp. $\mathcal{T}_{\text{pref}}$) contains a leaf k_1 (resp. k_2) corresponding to the longest substring of S_j starting at position i respecting the property Π, which starts with $P[\mu\,..\,m]$ (resp. the longest reversed substring of S_j ending at position i respecting the property, which starts with $P[1\,..\,\mu]^r$). Those leaf nodes both have a label $(i,j) = (k-1+\mu, j)$, hence the corresponding point is in $\mathcal{N}(P)$, which proves the result. □

Our index (i.e., $\mathcal{T}_{\text{suff}}$, $\mathcal{T}_{\text{pref}}$, and the grid) solves ℓ-WEIGHTED INDEXING by answering 2D range reporting queries [21]. In the *2D range reporting* problem, we are given a set $\mathcal{N}$ of N points to be preprocessed, so that when one gives an axis-aligned rectangle as a query, we report the subset $\mathcal{K}$ of $\mathcal{N}$ such that point $p \in \mathcal{K}$ if and only if the rectangle encloses p. We consider a special case of this problem which suffices for our purposes. In particular, we use the following result.

Lemma 8 (Section 2 of [69])**.** *Let $\mathcal{N}$ be a set of N points coming from pairing two permutations of $[1, N]$. With $\mathcal{O}(N)$ construction time, we can answer 2D range reporting queries in $\mathcal{O}((1+k)\log N)$ time using $\mathcal{O}(N)$ space, where k is the number of points from $\mathcal{N}$ enclosed by the query rectangle.*

Note that, if each occurrence of a pattern can be detected with 2D range reporting queries, the converse is not true: if a pattern U has a minimizer at position μ and both $U[1\,..\,\mu]$ and $U[\mu\,..\,|U|]$ are solid factors occurring at respective positions k and $k+\mu-1$, then a corresponding point will be detected with the 2D range reporting queries, *even* if U is not a solid factor itself. In that case, U is by definition a substring of some $S_j \in \mathcal{S}$, but does *not* respect the property. We can simply compute all the points by 2D range reporting, and check naively for false positives by comparing the pattern with X at the positions corresponding to these points. Conversely, one can have several points corresponding to a single occurrence, if the pattern appears in multiple strings in $\mathcal{S}$ at a same position,

[3] We claim $\mathcal{O}(nz)$ time and space during our construction because if $\log z > \ell$, we can abort the construction and resort to $\mathcal{O}(nz)$ size.

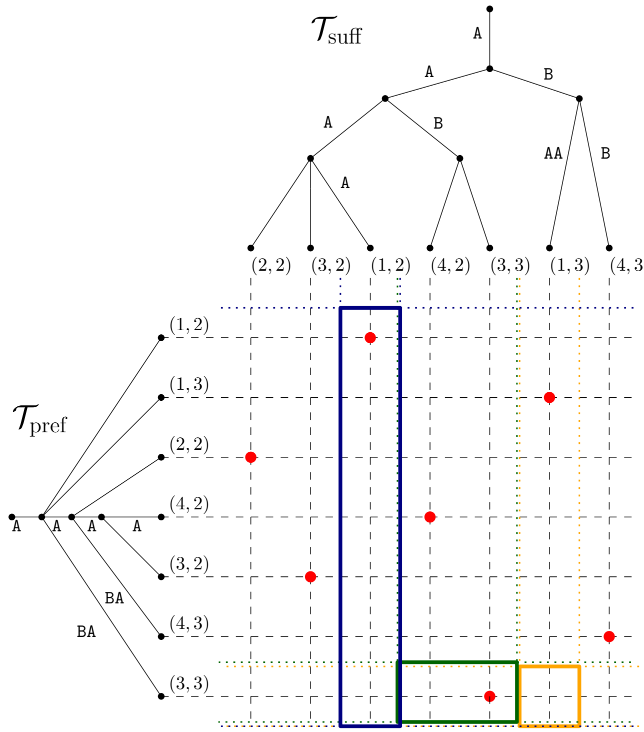


Fig. 2: Our index for the weighted string from Table I, $\frac{1}{z} = \frac{1}{4}$, and the minimizers from Example 2. $\mathcal{T}_{\text{suff}}$ is the forward minimizer solid factor tree and $\mathcal{T}_{\text{pref}}$ is the backward one. Edges without labels are constructed for readability and mean that the parent and the children nodes correspond to the same string. Each leaf node corresponds to a unique pair (i, j) such that the string is the minimizer appearing at position i in string S_j from the 4-estimation. The points from $\mathcal{N}$ are marked in red. The blue rectangle corresponds to $\mathcal{N}(P)$ for the pattern $P = \underline{\mathtt{A}}\mathtt{AAA}$, the green rectangle to $\mathcal{N}(P')$ for $P' = \mathtt{B}\underline{\mathtt{A}}\mathtt{AB}$, and the orange rectangle to $\mathcal{N}(P'')$ for $P'' = \mathtt{B}\underline{\mathtt{A}}\mathtt{BA}$ (the underlined positions are the minimizer positions).

which could also increase the running time. To control the number of such additional checks (both for false positives and duplicate ones), we give a bound on the expected number of occurrences of a given pattern in the z-estimation $\mathcal{S}$:

Lemma 9. *For any string P chosen uniformly at random from Σ^m, there are $\mathcal{O}(nz/\sigma^m)$ points expected in $\mathcal{N}(P)$.*

Proof. $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$ are constructed from a z-estimation $\mathcal{S}$, therefore each point in $\mathcal{N}(P)$ corresponds to an occurrence of P in $\mathcal{S}$ (it might not respect property Π however). Since $\mathcal{S}$ has $(n - m + 1)\lfloor z \rfloor \le nz$ substrings of length m, we have $\sum_{P \in \Sigma^m} |\mathcal{N}(P)| \le nz$, and hence if P is chosen uniformly at random we obtain no more than $\frac{nz}{\sigma^m}$ points in expectation. □

Main Result. We arrive at the main result of the section:

Theorem 10. *Let X be a weighted string of length n, $\frac{1}{z}$ be a weight threshold, and $\ell > 0$ be an integer. With $\mathcal{O}(nz)$ construction time and space, we can construct an index of $\mathcal{O}(n + \frac{nz}{\ell} \log z)$ expected size answering ℓ-*WEIGHTED INDEXING *queries of length $m \ge \ell$ in $\mathcal{O}(m + (1 + \frac{nz}{|\Sigma|^m}) \log \frac{nz}{\ell})$ expected time.*

Proof. We first construct the minimizer solid factor trees of X in $\mathcal{O}(nz)$ time and space; the trees have size $\mathcal{O}(\frac{nz}{\ell} \log z)$ (Lemma 6). We preprocess the pairs of leaves for 2D range reporting queries in $\mathcal{O}(\frac{nz}{\ell})$ time (Lemma 8). When a pattern P of length $m \ge \ell$ is given, we compute its leftmost minimizer μ in $\mathcal{O}(\ell)$ time [67], compute the sides of the rectangle in $\mathcal{O}(m)$ time by spelling $P[\mu \mathinner{.\,.} m]$ in $\mathcal{T}_{\text{suff}}$ and $(P[1 \mathinner{.\,.} \mu])^r$ in $\mathcal{T}_{\text{pref}}$, and answer a 2D range reporting query in $\mathcal{O}(\log \frac{nz}{\ell}(1 + |\mathcal{N}(P)|))$ time (Lemma 8). Finally, we must check, for every output point (i, j), for a valid occurrence around the ith minimizer of the jth string of the z-estimation. To do this efficiently (i.e., in $\mathcal{O}(\log z)$ time per point) *without storing the z-estimation of X*, we store only the $\log_2 z$ closest mismatching positions to the left and to the right of every minimizer in $\mathcal{M}_X$ (Lemma 4). The total verification time is thus $\mathcal{O}((\log z + \log(nz/\ell))(|\mathcal{N}(P)|) + 1) = \mathcal{O}(\log \frac{nz}{\ell}(|\mathcal{N}(P)| + 1))$. By Lemma 9, we know that in expectation we have $|\mathcal{N}(P)| = \frac{nz}{|\Sigma|^m}$. We obtain an expected query time of $\mathcal{O}(m + (1 + \frac{nz}{|\Sigma|^m}) \log \frac{nz}{\ell})$. The total size is $\mathcal{O}(n + \frac{nz}{\ell} \log z)$, to store H_X plus the index. □

Example 2. Let X be the weighted string from Table I and $\frac{1}{z} = \frac{1}{4}$. The construction of our index is detailed in Figure 2, and query rectangles $\mathcal{N}(P)$ (resp. $\mathcal{N}(P')$ and $\mathcal{N}(P'')$) are constructed for patterns $P = \underline{\mathtt{A}}\mathtt{AAA}$ (resp. $P' = \mathtt{B}\underline{\mathtt{A}}\mathtt{AB}$ and $P'' = \mathtt{B}\underline{\mathtt{A}}\mathtt{BA}$) whose minimizer positions are underlined. The blue rectangle $\mathcal{N}(P)$ contains exactly one point which corresponds to a substring $\mathtt{AAAA}$ in S_2. This substring corresponds to an occurrence of P at position 1 with probability $1 \cdot \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{4}{5} = 0.3$ in X. The green rectangle $\mathcal{N}(P')$ contains one point, which corresponds to an occurrence of P' in S_3. However, this occurrence does not respect the property Π (because $i + |P'| - 1 = 2 + 4 - 1 = 5 > \pi_3[2] = 4$) and therefore is a false positive in X. Finally, the orange rectangle $\mathcal{N}(P'')$ does not contain any point, because the pattern does not occur in the z-estimation. □

IV. SPACE-EFFICIENT CONSTRUCTION OF THE INDEX

Recall that to construct the index in Section III, we first construct a z-estimation, which temporarily takes $\Theta(nz)$ space during construction. In this section, we improve the space required for the construction of the index by designing a space-efficient algorithm for constructing a minimizer solid factor tree with only a moderate increase in the construction time.

Main Idea. We start the construction by *simulating* the construction of an *extended solid factor tree*. In particular, we maintain the subtree induced by the solid factors *starting at minimizer positions* but discard the nodes that we do not need upon returning to their parents. We do this via traversing the full tree in a depth-first search (DFS) order. Thus, even though the full tree size is $\Theta(nz)$, we store *only* the current leaf-to-root path plus the actual output. Therefore, we use only $\mathcal{O}(n + \frac{nz}{\ell} \log z)$ expected space at a cost of $\mathcal{O}(nz \log \ell)$ time.

We next reverse this tree (the solid factors are read from leaf to root there, while in the minimizer solid factor tree those are read from root to leaf), in $\mathcal{O}(\frac{nz}{\ell}\log\frac{nz}{\ell}\log z)$ expected time, using $\mathcal{O}(n+\frac{nz}{\ell}\log z)$ expected space. In total, this adds up to $\mathcal{O}(nz\log\ell+\frac{nz}{\ell}\log\frac{nz}{\ell}\log z)$ expected construction time (instead of $\mathcal{O}(nz)$) and $\mathcal{O}(n+\frac{nz}{\ell}\log z)$ expected construction space (instead of $\Theta(nz)$).

Key Concepts. The string $U\cdot H_X[j+1..n]$ (resp. $(H_X[1..i-1]\cdot U)^r$) is called the *right extension of the solid factor U* (resp. *left extension of the solid factor U*), if U is a solid factor of X starting at position i and ending at position $j\geq i-1$ [4].

For such a U, we define a *forward extended solid factor tree* of X as a trie of all the reversals of $U\cdot H_X[j+1..n]$, and a *backward extended solid factor tree* of X as a trie of all $H_X[1..i-1]\cdot U$. We employ the following results.

Lemma 11 ([18])**.** *The set of extensions of every solid factor for a weighted string X is equal to the set of extensions of its* maximal *solid factors.*

Lemma 12 ([18])**.** *The extended solid factor tree of a weighted string X has $\mathcal{O}(nz)$ nodes.*

Construction. We start by constructing the minimizer versions of the extended solid factor trees – that is for the solid factors trimmed to their parts starting (resp. ending) at the position of their minimizer; see Algorithm 1. In particular, we show how to construct the forward extended solid factor tree (see Fig. 4) – the backward one can be constructed by simply doing the same operations on the reversed string, except that the minimizers will be computed on the reversed substrings.

Initialization. (see Algorithm 1). We construct the tree with a DFS traversal of the full (non-minimizer) extended solid factor tree, starting from the root, which corresponds to an empty string. Each node corresponds to the right extension of a solid factor U of X starting at a position i (recall that U can be empty, in which case its right extension is $H_X[i..n]$).

First Visit to a Node. (see Algorithm 2). When a node u that corresponds to a solid factor U starting at position i of X and ending at position j is created, we keep a pair of labels (i, Diff), where Diff is the sequence (list) of mismatches between U and $H_X[i..j]$. By Lemma 4, the label of a given node has size $\mathcal{O}(\log z)$. Note, also, that for any ancestor of a node u its list of mismatches will be a suffix of Diff.

A single node and equivalently such a pair of labels can still represent multiple solid factors (for different values of j – if the suffix of the solid factor matches the heavy string); henceforth, by U we mean the shortest such solid factor: j is the largest element of Diff or $j=i-1$ if Diff is empty.

Additionally, for each node u, we check if the longest represented heavy factor has length at least ℓ [5] in which case we ask for the minimizer μ of this solid factor, and mark the $(\mu-1)$-th ancestor of u as a minimizer node. Such minimizer can be found in $\mathcal{O}(1)$ time using a heap data structure [28], which stores information about the length-k substrings of the length-ℓ prefix of $S=U\cdot H_X[j+1..n]$ and is updated in $\mathcal{O}(\log\ell)$ time in each step of the traversal.

Stepping Down to a Child Node. (see Algorithm 3). If U is empty, then the node u corresponds to a string $H_X[i..n]$. In this case, p is not updated when creating its child v corresponding to $H_X[i-1..n]$. This way, we ensure by induction that $p=1$ at the creation of each such a node, and only for such a node, so that this can be checked in $\mathcal{O}(1)$ time. We now assume that U is nonempty (Diff is nonempty). To create a child v of the node u, corresponding to the right extension of the string $\alpha\cdot U$ for some letter $\alpha\in\Sigma$, one needs to check if $\alpha\cdot U$ is valid by computing its probability. This is done using p, which we multiply by $p_{i-1}[\alpha]$.

In any case, the labels of v are computed from the labels of u by decreasing the starting position and potentially adding a new label to Diff via Algorithm 2.

Returning to the Parent Node. (see Algorithm 4). In the DFS we traverse the full extended solid factor tree, but we are only interested in the strings that start in those minimizer nodes. Thus, we remove the nodes which correspond to letters of the solid factors that appear before the position of the first minimizer and recursively compactify the tree during the traversal to save computation space.

After all descendants of u are created, we keep u explicit if it is a minimizer node or if it has more than one (not removed) child. Otherwise, the node u is made implicit by merging it with its parent. Finally, upon returning to the parent of u we update p by dividing it by $p_i[U[1]]$ (if $p<1$) and the list of differences by removing position i if $i\in$ Diff.

Complexity Analysis. By Lemma 6 the final tree has $\mathcal{O}(\frac{nz}{\ell})$ nodes in expectation. As for the construction space, observe that while a node u is being processed, only the path between u and the root is uncompacted, and contains at most n nodes. All the other global variables also have size $\mathcal{O}(n)$, therefore the total expected work space needed is $\mathcal{O}(\frac{nz}{\ell}\log z+n)$.

As for the construction time, note that the set of created nodes is exactly the set of nodes from the original extended solid factor tree (namely, without minimizers), which has size $\mathcal{O}(nz)$ by Lemma 12. During the construction, all the operations cost $\mathcal{O}(1)$ with the exception of updating the minimizer heap which takes $\mathcal{O}(\log\ell)$ time and storing a copy of the list of differences for each minimizer node in $\mathcal{O}(\log z)$ time. Note that the last type of the operation does not influence the worst case running time as we can abandon the computation upon learning that the total size of those lists reaches nz – in which case the classic (non-minimizer) data structure is more efficient. We have thus proved the following lemma.

Lemma 13. *For any weighted string of length n, any weight threshold $\frac{1}{z}$, and any integer $\ell>0$, we can construct a representation of the minimizer extended solid factor trees in $\mathcal{O}(nz\log\ell)$ time using $\mathcal{O}(n+\frac{nz\log z}{\ell})$ expected space.*

Main result. By Lemma 13, we show below that the minimizer solid factor trees can be constructed in expected time

[4] If $j=i-1$, then $U[i..i-1]=\varepsilon$, the empty string.

[5] We check this in $\mathcal{O}(1)$ time using value p – a global variable that denotes the probability of the current node – that is the weight of U and the precomputed array PP_H of prefix products of H_X for the heavy part.

Algorithm 1 Construct-$\mathcal{T}(X)$

```
Global variables: Weighted string $X$, heavy string $H_X$, $j = n$, $p = 1.0$, string $S = \varepsilon$, set Diff $= \emptyset$, set Minimizers $= \emptyset$.
create a node $root$
run Augment-$\mathcal{T}(n+1, root)$
return $root$                ▷ The minimizer extended solid factor tree
```

Algorithm 3 Augment-$\mathcal{T}(i, u)$

```
for $\alpha \in \Sigma$ do
    if $p = 1$ and $\alpha = H_X[i-1]$ then                ▷ If $U$ is empty
        $j \leftarrow j - 1$
        run DOWN$(i-1, u, \alpha)$
        $j \leftarrow j + 1$
    else if $p \cdot p_{i-1}[\alpha] \geq \frac{1}{z}$ then
        $p \leftarrow p \cdot p_{i-1}[\alpha]$
        run DOWN$(i-1, u, \alpha)$
if $i < n+1$ then run UP$(i, u)$
```

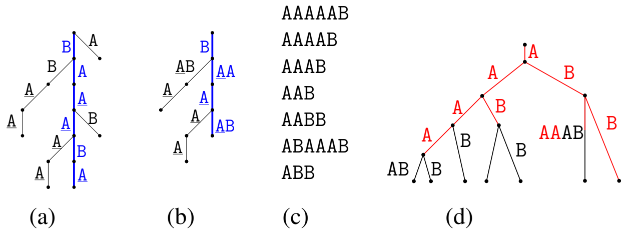
Algorithm 2 DOWN(i, u, α)

```
$S \leftarrow \alpha S$
if $\alpha \neq H_X[i]$ then add $(i, \alpha)$ to Diff
if $|S| \geq \ell$ and $p \cdot PP_H[i-1+\ell]/PP_H[j] \geq \frac{1}{z}$ then                ▷ $S[1\,..\,\ell]$ is solid
    Minimizers $\leftarrow$ Minimizers $\cup \{i + f(S[1\,..\,\ell]) - 1\}$
add a node $v$ as a child of $u$
run Augment-$\mathcal{T}(i, v)$
```

Algorithm 4 UP(i, u):

```
if $i \in$ Minimizers then
    remove $i$ from Minimizers
    set label of $u$ to $(i, \text{Diff})$
else if $u$ has at most one child then merge $u$ with PARENT$(u)$
$p \leftarrow \min(1, p \cdot p_i[S[1]]^{-1})$
if $(i, S[1]) \in$ Diff then remove $(i, S[1])$ from Diff
remove the first letter from $S$
```

Fig. 3: The space-efficient algorithm for constructing the minimizer extended solid factor tree of a weighted string X.



AAAAAB
AAAAB
AAAB
AAB
AABB
ABAAAB
ABB

(a) (b) (c) (d)

Fig. 4: (a) The forward extended solid factor tree with X from Table I. The blue edges correspond to the heavy string H_X = ABAAAB (reversed). The minimizer positions are underlined. (b) The minimizer extended solid factor tree. The edges without any minimizer descendant nodes are pruned and the non-minimizer nodes are made implicit. (c) The lexicographically sorted strings corresponding to each path from a minimizer node to a root. (d) The minimizer solid factor tree constructed by Theorem 14. It contains the forward (top) tree from Figure 2 as a red subtree. The edge with no label is added in the figure to stress that AAB also has a corresponding leaf. In the algorithm, we simply make the corresponding internal node explicit and treat it as a leaf node.

$\mathcal{O}(nz \log \ell + \frac{nz}{\ell} \log \frac{nz}{\ell} \log z)$ and space $\mathcal{O}(n + \frac{nz}{\ell} \log z)$:

Theorem 14. *For any weighted string X of length n, any weight threshold $\frac{1}{z}$, and any integer $\ell > 0$, we can construct the minimizer solid factor tree from the minimizer extended solid factor tree in $\mathcal{O}(n + \frac{nz}{\ell} \log \frac{nz}{\ell} \log z)$ expected time and $\mathcal{O}(n + \frac{nz}{\ell} \log z)$ expected space.*

Proof. We need to reverse the tree: create the trie of all the strings from the minimizer extended solid factor tree read from leaf to root (corresponding to strings $U \cdot H_X[j+1\,..\,n]$). Note that for two such strings we can find their longest common prefix (LCP), and hence also compare them in $\mathcal{O}(\log z)$ time with a use of an LCP data structure for H_X [62] (comparison of $\mathcal{O}(\log z)$ intervals of H_X and $\mathcal{O}(\log z)$ differences).

We first sort those strings in lexicographic order. Since there are in expectation $\mathcal{O}(\frac{nz}{\ell})$ of them, and a single comparison takes $\mathcal{O}(\log z)$ time, this takes $\mathcal{O}(\frac{nz}{\ell} \log \frac{nz}{\ell} \log z)$ time in total using any optimal comparison-based sorting algorithm [28]. Now we construct the compacted trie of those strings node by node in the order of a DFS. Each edge will be labeled with an interval of H_X and a list of at most $\log_2 z$ differences.

We start from creating a single edge from root to a leaf representing the first string. Now we iterate over all remaining strings in lexicographic order – we first compute the length of the LCP of this string, and the previous one, next starting from the leaf representing the previous string we move up the tree node by node to find its ancestor at depth equal to the length of this LCP. If this node turns out to be an implicit one, then we make it explicit by dividing the edge (and hence also the interval of H_X and the list of differences). We finish by creating a new child of the reached node – this child becomes the leaf representing the new string.

Unlike the construction from [18] we do not need to trim the H_X parts after constructing the tree, as in our query algorithm we must verify the weight for each match anyway. □

V. Practically Fast Querying Without a Grid

In this section, we describe a simple and fast querying algorithm that does not require the grid to be constructed on top of the trees. While this querying algorithm has worse guarantees than Theorem 10, it performs much better in practice, due to its simplicity, as we show later in the experimental evaluation.

Like in the previous construction let $\mu = f(P[1\,..\,\ell])$. Without loss of generality we assume that $\mu \leq \frac{m}{2}$ (otherwise we swap the roles of the parts of P and of trees $\mathcal{T}_{\text{suff}}$ and $\mathcal{T}_{\text{pref}}$). Let u be the node reached by reading $P[\mu\,..\,m]$ in

$\mathcal{T}_{\text{suff}}$. We can separately check each leaf in the subtree of u as a potential candidate in $\mathcal{O}(m)$ time: we can do this assuming we have random access to X. This time we cannot use the $\frac{nz}{|\Sigma|^m}$ bound on the expected number of candidates from Lemma 9. However, $P[\mu \mathinner{.\,.} m]$ has length at least $m/2$, and hence the expected number of candidates can still be bounded by $\sum_{k=\lceil m/2 \rceil}^{m} \frac{nz}{|\Sigma|^k} \leq 2\frac{nz}{|\Sigma|^{m/2}}$ using a similar argument. Thus we can answer a query in $\mathcal{O}(m \cdot (1 + \frac{nz}{|\Sigma|^{m/2}}))$ expected time.

VI. Related Work

As mentioned in Introduction, there are no practical indexing schemes for uncertain strings due to their prohibitive space requirements (we refer to [17] for a survey of theoretical solutions). However, there is substantial work on practical indexing schemes for probabilistic/uncertain data. There have been proposed indexes for various types of queries, including range queries [87], [23], [88], [26], [8], top-k queries [98], [45], [99], [78], nearest neighbor queries [7], [25], [24], sql-like queries [77], inference queries [50], and probabilistic equality threshold queries [84]. These indexes were developed for different uncertainty data models, such as tuple uncertainty and attribute uncertainty [85]. Under tuple uncertainty, the presence of a tuple in a relation is probabilistic, while under attribute uncertainty a tuple is certainly present in a database but one or more of its attributes are not known with certainty. Several indexes are built on R-trees or inverted indexes (e.g., [84], [30]), while others are built on R*-trees [87]. There are also specialized indexes, e.g., for probabilistic XML queries [39] or uncertain graphs [97], [86]. Our work differs substantially from these indexes in the type of supported data (uncertain string) and query type (pattern matching query).

Many topics beyond indexing have also been studied on probabilistic, uncertain, incomplete, and/or fuzzy data; see [64], [63], [11], [68], [81] for surveys. These topics range from the theoretical development of data models (e.g., [36], [61], [41], [83], [75]) to query languages (e.g., [14], [60], [31]) and to systems (e.g., [96], [74], [91]).

VII. Experimental Evaluation

A. Data and Setup

Data. We used three real weighted strings which model variations found in the DNA ($\sigma = 4$) of different samples of the same species. The chromosomal or genomic location of a gene or any other genetic element is called a *locus* and alternative DNA sequences at a locus are called *alleles*. Allele frequency, or gene frequency, is the relative frequency of an allele at a particular locus in a population, expressed as a fraction or percentage. Thus, alleles have a natural representation as weighted strings. In particular, we model the probability $p_i(\alpha)$ in these strings as the relative frequency of letter α at position i among the different samples.

We next describe the datasets we used (see also Table III):

- SARS: The full genome of *SARS-CoV-2* (isolate Wuhan-Hu-1) [1] combined with a set of single nucleotide polymorphisms (SNPs) [2] taken from $1,181$ samples [90].

TABLE III: Characteristics of the real datasets we used.

Dataset	**# of samples**	**Length** n	Δ **as percentage of** n	**Size of** z**-estimation for the default** z **(MBs)**
SARS	$1,181$	$29,903$	3.6%	31
EFM	$1,432$	$2,955,294$	6%	378
HUMAN	$2,504$	$35,194,566$	3.2%	282

- EFM: The full chromosome of *Enterococcus faecium* Aus0004 strain (CP003351) [3] combined with a set of SNPs [4] taken from $1,432$ samples [27].
- HUMAN: The full chromosome 22 of the *Homo sapiens* genome (v. GRCh37) [5] combined with a set of SNPs [6] taken from the final phase of the 1000 Genomes Project (phase 3) representing $2,504$ samples on GRCh37 [32].

The percentage of positions where more than one letter has a probability of occurrence larger than 0 is denoted by Δ.

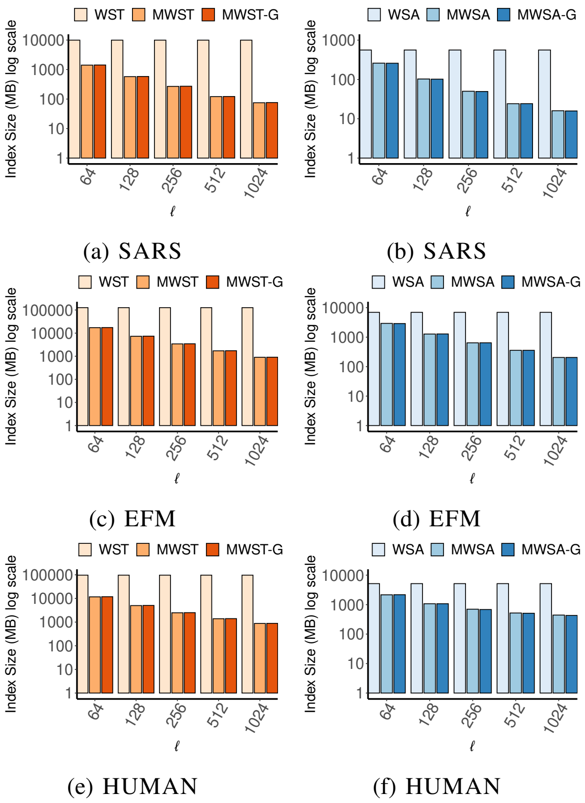
We also used a synthetic weighted string of length $n = 10,000$ over an alphabet of size $\sigma = 20$. For this dataset, $\Delta = 5.37\%$. The results were analogous to those for the real datasets (omitted for space). Indeed, note from our theoretical results (e.g., Theorem 10) that increasing σ does not negatively affect our index in any measure of efficiency.

Parameters. For every weighted string of length n, every pattern length $m \in \{64, 128, 256, 512, 1024\}$, and every z we used, we selected $\lfloor nz/200 \rfloor$ patterns, occurring with probability at least $\frac{1}{z}$, uniformly at random from the weighted string, to account for the different n and z values in our datasets. For example, for HUMAN, which is of length $n = 35,194,566$, and for $z = 32$, we have selected $5,631,130$ valid patterns uniformly at random. The default z for SARS, EFM, and HUMAN was 1024, 128, and 8, respectively, and led to z-estimations with sizes of several MBs; see Table III. The parameter ℓ was set to m, and the default m value was 256.

Implementations. We used the implementations of the state-of-the-art indexes WST and WSA from [17] and [22], respectively. We implemented: (1) MWST-G, the algorithm underlying our Theorem 10. (2) MWST, a simplified version of MWST-G that drops the 2D grid and performs pattern matching as described in Section V. (3) MWSA and MWSA-G, the *array-based* versions of MWST and MWST-G, respectively. A standard in-order DFS traversal of MWST gives MWSA. (4) MWST-SE, the space-efficient construction of MWST underlying Theorem 14. In all implementations, we used Karp-Rabin fingerprints [52] to compute the minimizers.

Measures. We used all four relevant measures of efficiency (see Introduction): index size; query time; construction space; and construction time. To measure the query and construction time, we used the `chrono C++` library. To measure the index size, we used the `malloc2 C++` function. To measure the construction space, we recorded the maximum resident set size using the `/usr/bin/time -v` command.

Environment. All experiments ran using a single AMD EPYC 7282 CPU at 2.8GHz with 252GB RAM under GNU/Linux. All methods were implemented in `C++` and compiled with `g++` (v. 12.2.1) at optimization level `-O3`.

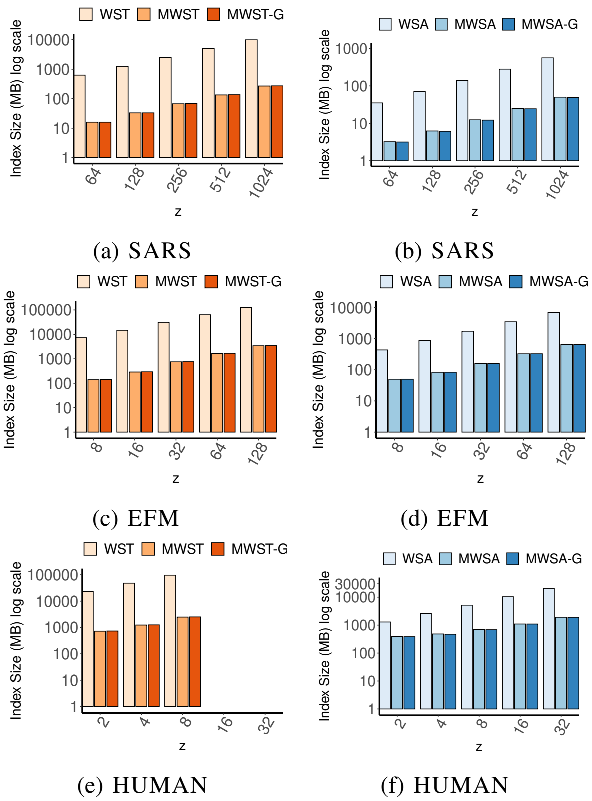


(a) SARS (b) SARS

(c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Fig. 5: Index size (log scale, MB) vs. ℓ.

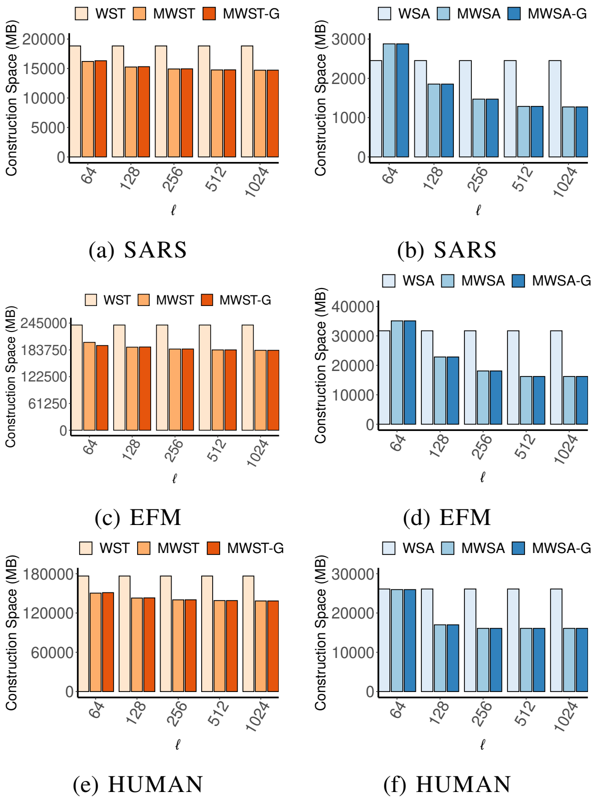


(a) SARS (b) SARS

(c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Fig. 6: Index size (log scale, MB) vs. z. The tree-based indexes for HUMAN (Fig. 6e) needed $>$ 252GB of space when $z \geq$ 16 and hence could not be constructed.

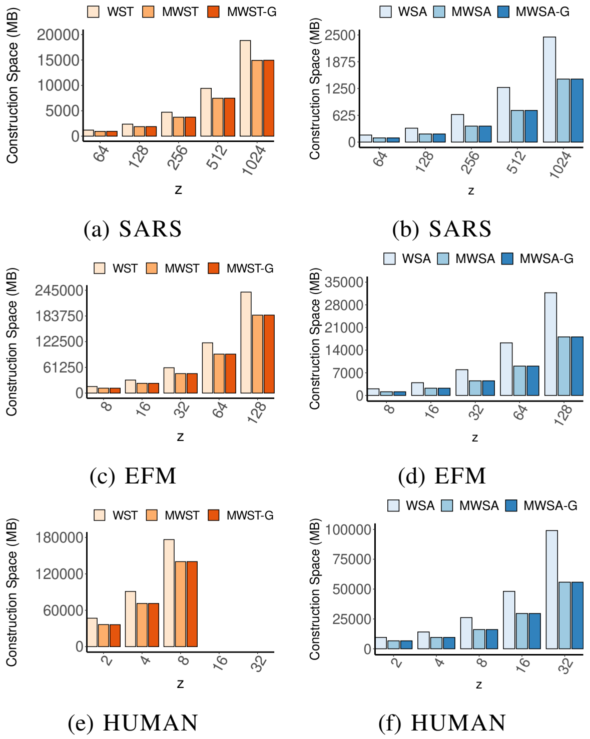


(a) SARS (b) SARS

(c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Fig. 7: Construction space (MB) vs. ℓ.

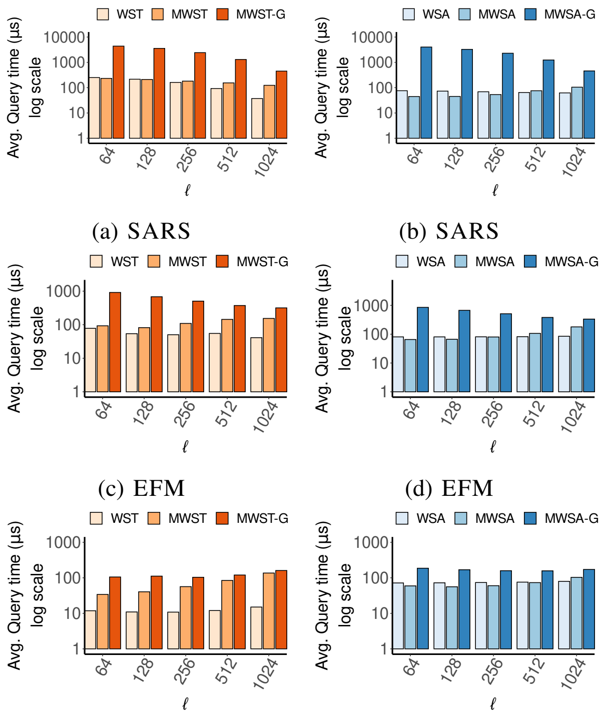


(a) SARS (b) SARS

(c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Fig. 8: Construction space (MB) vs. z. The tree-based indexes for HUMAN (Fig. 8e) needed $>$ 252GB when $z \geq$ 16 and hence could not be constructed.

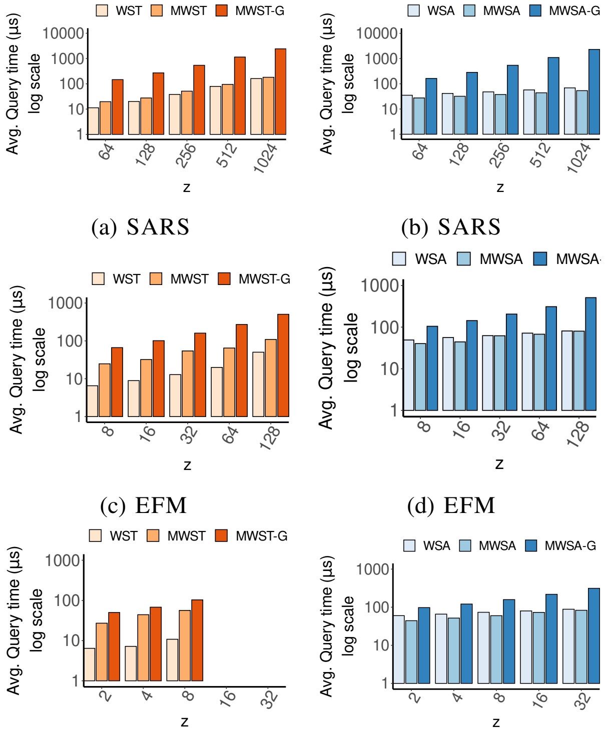


(a) SARS (b) SARS

(c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Fig. 9: Average query time (log scale, μs) vs. ℓ.

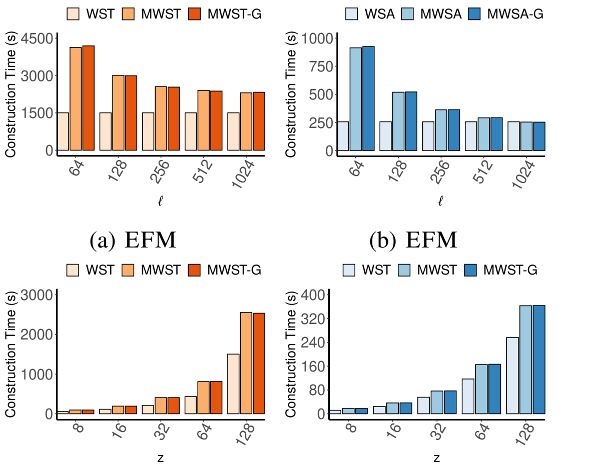


(a) SARS (b) SARS

(c) EFM (d) EFM

(e) HUMAN (f) HUMAN

Fig. 10: Average query time (log scale, μs) vs. z. The tree-based indexes for HUMAN (Fig. 10e) needed > 252GB when $z \geq 16$ and hence could not be constructed.



(a) EFM (b) EFM

(c) EFM (d) EFM

Fig. 11: (a, b) Construction time (s) vs. ℓ for EFM. (c, d) Construction time (s) vs. z for EFM. The results for SARS and HUMAN were analogous (omitted for space).

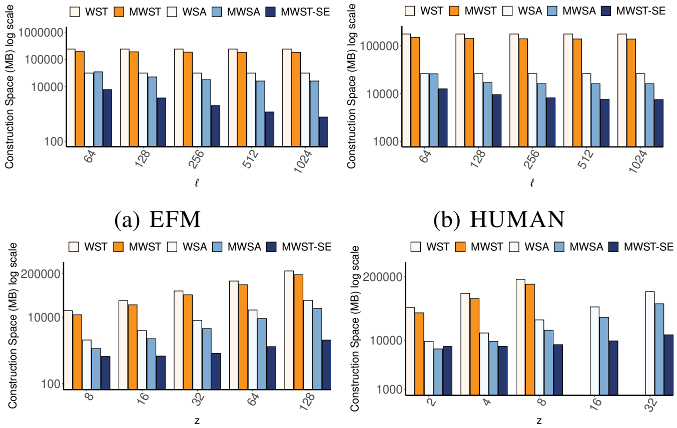
Code and Datasets. The code and all datasets are available at https://github.com/solonas13/ius under GNU GPL v3.0.

B. Evaluating our Minimizer-based Indexes

This section shows that: (1) our indexes are up to two orders of magnitude smaller than the state-of-the-art indexes and can be constructed in much less space; (2) our indexes have query and construction times that are competitive to that of the state of the art; and (3) our simplified indexes allow faster queries than the grid-based ones despite having weaker guarantees.

Index Size. Figs. 5 and 6 show that our tree-based (resp. array-based) indexes occupy *up to two orders of magnitude less space* than WST (resp. WSA). The size of our indexes decreases with ℓ and increases with z (see Theorem 10). Furthermore, the array-based indexes occupy several times less space than the tree-based ones, as it is widely known [51]. For example, note from Figs. 5c and 5d that for $\ell = 1024$, WST occupied 126GB of space, whereas our MWST 900MB and MWSA only 204MB! As expected, our grid-based indexes MWST-G and MWSA-G occupy a slight amount of extra space compared to MWST and MWSA, respectively.

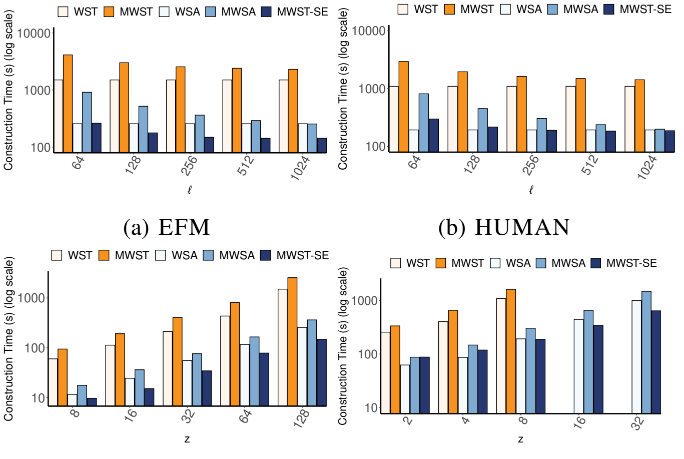
Construction Space. Figs. 7 and 8 show that our tree-based (resp. array-based) indexes outperform WST (resp. WSA) *by* 27% *(resp.* 61%*)* on average. Although our construction algorithm (see Theorem 10) takes $\Theta(nz)$ space in any case, it carries lower constant factors than that of WST. That is, in practice, the index construction space for our tree-based indexes decreases as ℓ increases and increases with z – see Lemmas 6 and 8, which show a clear dependency on the number $\mathcal{O}(\frac{nz}{\ell})$ of sampled minimizers. The same explanation holds for WSA and our array-based indexes. Again, as it is widely known [51], the array-based indexes outperform the tree-based ones in terms of space; and, as expected, MWST-G and MWSA-G need a very slightly larger construction space than MWST and MWSA, respectively.



(a) EFM (b) HUMAN

(c) EFM (d) HUMAN

Fig. 12: Construction space (log scale, MB) vs: (a, b) ℓ. (c, d) z. WST and MWST for HUMAN (Fig. 12d) needed $>$ 252GB when $z \geq 16$ and hence could not be constructed. The results for SARS were analogous (omitted for space).



(a) EFM (b) HUMAN

(c) EFM (d) HUMAN

Fig. 13: Construction time (log scale, s) vs: (a, b) ℓ. (c, d) z. WST and MWST for HUMAN (Fig. 13d) needed $>$ 252GB when $z \geq 16$ and hence could not be constructed. The results for SARS were analogous (omitted for space).

Query Time. Figs. 9 and 10 show that MWST is generally slower than WST because its search operation is more costly than that of WST (see Theorem 10). However, MWSA is competitive to WSA since, due to the smaller size of the former, the binary search operation used in query answering (pattern matching) [70] becomes faster. This is *very encouraging* given its substantially smaller index size and index construction space across all z and ℓ values. Furthermore, MWST and MWSA outperform MWST-G and MWSA-G, respectively. This is in line with the findings of [16], [67], which show that simple verification schemes like the one developed by us in Section V, are faster than grid approaches, even if the theoretical guarantees provided by the former are weaker. The query time of the grid-based indexes is not negatively affected by increasing ℓ, unlike MWST and MWSA, which highlights the benefit of Theorem 10. The query time of all indexes increases with z, as expected by their time complexities. The query time of WST and WSA does not depend on ℓ, as expected by their time complexities.

Construction Time. Fig. 11 shows that WST and WSA can be constructed in less time than our tree-based and array-based indexes, respectively. This is expected as our construction is much more complex than that of WST and WSA [17], [22]. In particular, although our construction algorithm (see Theorem 10) takes $\Theta(nz)$ time in any case, it carries higher constant factors than those of WST and WSA. This is expected as, in some sense, our construction largely follows the one of WST and WSA but it does additional work implied by the sampling mechanism. In practice, the construction time decreases as ℓ increases and increases with z – see Lemmas 6 and 8, which show a clear dependency on the number $\mathcal{O}(\frac{nz}{\ell})$ of sampled minimizers. On average, MWST requires 70% (resp. MWSA requires 41%) more time to be constructed than WST (resp. WSA). MWST-G and MWSA-G has similar construction time to MWST and MWSA, respectively.

C. Evaluating our Space-efficient Index Construction

Construction space. Fig. 12 shows that the construction space of MWST-SE is *one order of magnitude smaller* than that of WSA and 52 *times smaller* than that of MWST on average. The construction space of MWST-SE decreases with ℓ and increases with z, as expected by Theorem 14. For example, in Fig. 12a MWST-SE needs only 772MB of memory to be constructed when $\ell = 1024$, while WSA and MWST need over 32GBs and 183GBs, respectively. Even for $\ell = 64$, the construction space of MWST-SE is 4 times smaller than that of WSA and more than 25 times smaller than that of MWST.

Construction Time. Fig. 13 shows that the construction time of MWST-SE is on average 13% *smaller* than that of WSA, the next fastest index. This is *very encouraging*, as MWST-SE is quite complex. The construction time of MWST-SE decreases with ℓ and increases with z, as expected by Theorem 14. For example, for $\ell = 1024$ and $z = 128$ in Fig. 13a, the construction time of MWST-SE is smaller by 44% (resp. 16 times smaller) compared to that of WSA (resp. MWST). This faster construction is a consequence of WST being always of $\Theta(nz)$ size (producing copies of solid factors), while in the extended solid factor trees each solid factor is considered only once. The $\mathcal{O}(\log \ell)$ cost for heap operations is very optimized and in practice comparable with the large constants of the other constructions for reasonable ℓ values.

D. Conclusion of our Experimental Evaluation

To conclude, the most practical solution to ℓ-WEIGHTED INDEXING is to use the MWST-SE algorithm to construct MWST, which requires the smallest construction space and time (see Figs. 12 and 13), and then infer MWSA, the array-based version of MWST via a standard in-order DFS traversal on MWST [70], as MWSA has the smallest index size and a competitive query time to WSA (see Figs. 5, 6, 9, and 10).

REFERENCES

[1] https://www.ncbi.nlm.nih.gov/nuccore/MN908947.3.

[2] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8363274/bin/elife-66857-supp2.txt.

[3] https://www.ncbi.nlm.nih.gov/nuccore/CP003351.

[4] https://github.com/francesccoll/powerbacgwas/blob/main/data/efm_clade_all.vcf.gz.

[5] https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.13/.

[6] https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL.chr22.phase3_shapeit2_mvncall_integrated_v5b.20130502.genotypes.vcf.gz.

[7] Pankaj K. Agarwal, Boris Aronov, Sariel Har-Peled, Jeff M. Phillips, Ke Yi, and Wuzhou Zhang. Nearest-neighbor searching under uncertainty II. *ACM Trans. Algorithms*, 13(1):3:1–3:25, 2016.

[8] Pankaj K. Agarwal, Siu-Wing Cheng, Yufei Tao, and Ke Yi. Indexing uncertain data. In Jan Paredaens and Jianwen Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 137–146. ACM, 2009.

[9] Charu C. Aggarwal. On unifying privacy and uncertain data models. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 386–395. IEEE Computer Society, 2008.

[10] Charu C. Aggarwal. *Managing and Mining Uncertain Data*, volume 35 of *Advances in Database Systems*. Kluwer, 2009.

[11] Charu C. Aggarwal and Philip S. Yu. A survey of uncertain data algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 21(5):609–623, 2009.

[12] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. In Moshe Lewenstein and Gabriel Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 188–199. Springer, 2006.

[13] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, 2008.

[14] Lyublena Antova, Christoph Koch, and Dan Olteanu. Query language support for incomplete information in the maybms system. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1422–1425. ACM, 2007.

[15] Mozhdeh Ariannezhad, Ali Montazeralghaem, Hamed Zamani, and Azadeh Shakery. Improving retrieval performance for verbose queries via axiomatic analysis of term discrimination heuristic. In Noriko Kando, Tetsuya Sakai, Hideo Joho, Hang Li, Arjen P. de Vries, and Ryen W. White, editors, *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*, pages 1201–1204. ACM, 2017.

[16] Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns: Anchors are all you need. *Proc. VLDB Endow.*, 16(9):2117–2131, 2023.

[17] Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Inf. Comput.*, 270, 2020.

[18] Carl Barton, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Efficient index for weighted sequences. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 4:1–4:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[19] Michael Bendersky and W. Bruce Croft. Discovering key concepts in verbose queries. In Sung-Hyon Myaeng, Douglas W. Oard, Fabrizio Sebastiani, Tat-Seng Chua, and Mun-Kew Leong, editors, *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Singapore, July 20-24, 2008*, pages 491–498. ACM, 2008.

[20] Sudip Biswas, Manish Patil, Sharma V. Thankachan, and Rahul Shah. Probabilistic threshold indexing for uncertain strings. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 401–412. OpenProceedings.org, 2016.

[21] Timothy M. Chan, Kasper Green Larsen, and Mihai Puatracscu. Orthogonal range searching on the RAM, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011.

[22] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications in indexing weighted sequences. *ACM J. Exp. Algorithmics*, 25:1–16, 2020.

[23] Lu Chen, Yunjun Gao, Aoxiao Zhong, Christian S. Jensen, Gang Chen, and Baihua Zheng. Indexing metric uncertain data for range queries and range joins. *VLDB J.*, 26(4):585–610, 2017.

[24] Reynold Cheng, Lei Chen, Jinchuan Chen, and Xike Xie. Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 672–683. ACM, 2009.

[25] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Querying imprecise data in moving object environments. *IEEE Trans. Knowl. Data Eng.*, 16(9):1112–1127, 2004.

[26] Reynold Cheng, Yuni Xia, Sunil Prabhakar, Rahul Shah, and Jeffrey Scott Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 876–887. Morgan Kaufmann, 2004.

[27] Francesc Coll, Theodore Gouliouris, Sebastian Bruchmann, Jody Phelan, Kathy E. Raven, Taane G. Clark, Julian Parkhill, and Sharon J. Peacock. PowerBacGWAS: a computational pipeline to perform power calculations for bacterial genome-wide association studies. *Communications Biology*, 5(266), 2022.

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[29] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[30] Xiangyuan Dai, Man Lung Yiu, Nikos Mamoulis, Yufei Tao, and Michail Vaitis. Probabilistic spatial queries on existentially uncertain data. In Claudia Bauzer Medeiros, Max J. Egenhofer, and Elisa Bertino, editors, *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, volume 3633 of *Lecture Notes in Computer Science*, pages 400–417. Springer, 2005.

[31] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 864–875. Morgan Kaufmann, 2004.

[32] Susan Fairley, Ernesto Lowy-Gallego, Emily Perry, and Paul Flicek. The International Genome Sample Resource (IGSR) collection of open human genomic variation resources. *Nucleic Acids Research*, 48(D1):D941–D947, 10 2019.

[33] Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143, 1997.

[34] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[35] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[36] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[37] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

[38] Tingjian Ge and Zheng Li. Approximate substring matching over uncertain strings. *Proc. VLDB Endow.*, 4(11):772–782, 2011.

[39] Jian Gong, Reynold Cheng, and David W. Cheung. Efficient management of uncertainty in XML schema matching. *VLDB J.*, 21(3):385–409, 2012.

[40] Szymon Grabowski and Marcin Raniszewski. Sampled suffix array with minimizers. *Softw. Pract. Exp.*, 47(11):1755–1771, 2017.

[41] Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. *IEEE Data Eng. Bull.*, 29(1):17–24, 2006.

[42] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[43] Manish Gupta and Michael Bendersky. Information retrieval with verbose queries. In Ricardo Baeza-Yates, Mounia Lalmas, Alistair Moffat, and Berthier A. Ribeiro-Neto, editors, *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015*, pages 1121–1124. ACM, 2015.

[44] Monika Rauch Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In Efthimis N. Efthimiadis, Susan T. Dumais, David Hawking, and Kalervo Järvelin, editors, *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*, pages 284–291. ACM, 2006.

[45] Ming Hua, Jian Pei, and Xuemin Lin. Ranking queries on uncertain data. *VLDB J.*, 20(1):129–153, 2011.

[46] Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios K. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Informaticae*, 71(2-3):259–277, 2006.

[47] Chirag Jain, Arang Rhie, Nancy Hansen, Sergey Koren, and Adam M. Phillippy. Long-read mapping to repetitive reference sequences using winnowmap2. *Nat Methods*, 19:705–710, 2022.

[48] Jeffrey Jestes, Feifei Li, Zhepeng Yan, and Ke Yi. Probabilistic string similarity joins. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 327–338. ACM, 2010.

[49] Jiaojiao Jiang, Steve Versteeg, Jun Han, Md. Arafat Hossain, Jean-Guy Schneider, Christopher Leckie, and Zeinab Farahmandpour. P-gram: Positional n-gram for the clustering of machine-generated messages. *IEEE Access*, 7:88504–88516, 2019.

[50] Bhargav Kanagal and Amol Deshpande. Indexing correlated probabilistic databases. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 455–468. ACM, 2009.

[51] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[52] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[53] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, pages 181–192, 2001.

[54] A.E. Kel, E. Gössling, I. Reuter, E. Cheremushkin, O.V. Kel-Margoulis, and E. Wingender. MATCHTM: a tool for searching transcription factor binding sites in DNA sequences. *Nucleic Acids Research*, 31(13):3576–3579, 07 2003.

[55] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019.

[56] Dominik Kempa and Tomasz Kociumaka. Breaking the $o(n)$-barrier in the construction of compressed suffix arrays and suffix trees. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 5122–5202. SIAM, 2023.

[57] Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. In Seok-Hee Hong, editor, *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*, volume 64 of *LIPIcs*, pages 46:1–46:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[58] Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. *Theory Comput. Syst.*, 63(3):506–542, 2019.

[59] Janne H. Korhonen, Petri Martinmäki, Cinzia Pizzi, Pasi Rastas, and Esko Ukkonen. MOODS: fast search for position weight matrix matches in DNA sequences. *Bioinform.*, 25(23):3181–3182, 2009.

[60] Bart Kuijpers and Walied Othman. Trajectory databases: Data models, uncertainty and complete query languages. *J. Comput. Syst. Sci.*, 76(7):538–560, 2010.

[61] Laks V. S. Lakshmanan, Nicola Leone, Robert B. Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3):419–469, 1997.

[62] Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986.

[63] Lingli Li, Hongzhi Wang, Jianzhong Li, and Hong Gao. A survey of uncertain data management. *Frontiers Comput. Sci.*, 14(1):162–190, 2020.

[64] Yiping Li, Jianwen Chen, and Ling Feng. Dealing with uncertainty: A survey of theories and practices. *IEEE Trans. Knowl. Data Eng.*, 25(11):2463–2482, 2013.

[65] Yuxuan Li, James Bailey, Lars Kulik, and Jian Pei. Efficient matching of substrings in uncertain sequences. In Mohammed Javeed Zaki, Zoran Obradovic, Pang-Ning Tan, Arindam Banerjee, Chandrika Kamath, and Srinivasan Parthasarathy, editors, *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pages 767–775. SIAM, 2014.

[66] Glennis A. Logsdon, Mitchell R. Vollger, and Evan E. Eichler. Long-read human genome sequencing and its applications. *Nat. Rev. Genet.*, 21(10):597–614, 2020.

[67] Grigorios Loukides and Solon P. Pissis. Bidirectional string anchors: A new string sampling mechanism. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 64:1–64:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[68] Zong Min Ma and Li Yan. A literature overview of fuzzy database models. *J. Inf. Sci. Eng.*, 24(1):189–202, 2008.

[69] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006.

[70] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[71] Olena Medelyan and Ian H. Witten. Thesaurus based automatic keyphrase indexing. In Gary Marchionini, Michael L. Nelson, and Catherine C. Marshall, editors, *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2006, Chapel Hill, NC, USA, June 11-15, 2006, Proceedings*, pages 296–297. ACM, 2006.

[72] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[73] Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive string dictionary compression in in-memory column-store database systems. In Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy, editors, *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, pages 283–294. OpenProceedings.org, 2014.

[74] Dan Olteanu, Jiewen Huang, and Christoph Koch. SPROUT: lazy vs. eager query plans for tuple-independent probabilistic databases. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 640–651. IEEE Computer Society, 2009.

[75] Olivier Pivert and Henri Prade. A certainty-based model for uncertain databases. *IEEE Trans. Fuzzy Syst.*, 23(4):1181–1196, 2015.

[76] Cinzia Pizzi, Pasi Rastas, and Esko Ukkonen. Finding significant matches of position weight matrices in linear time. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 8(1):69–79, 2011.

[77] Yinian Qi, Rohit Jain, Sarvjeet Singh, and Sunil Prabhakar. Threshold query optimization for uncertain data. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 315–326. ACM, 2010.

[78] Niranjan Rai and Xiang Lian. Distributed probabilistic top-k dominating queries over uncertain databases. *Knowl. Inf. Syst.*, 65(11):4939–4965, 2023.

[79] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinform.*, 20(18):3363–3369, 2004.

[80] Patricia Rodriguez-Tomé, Peter Stoehr, Graham Cameron, and Tomas P. Flores. The european bioinformatics institute (EBI) databases. *Nucleic Acids Res.*, 24(1):6–12, 1996.

[81] Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, Shubha U. Nabar, and Jennifer Widom. Representing uncertain data: models, properties, and algorithms. *VLDB J.*, 18(5):989–1019, 2009.

[82] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85. ACM, 2003.

[83] Prithviraj Sen, Amol Deshpande, and Lise Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.

[84] Sarvjeet Singh, Chris Mayfield, Sunil Prabhakar, Rahul Shah, and Susanne E. Hambrusch. Indexing uncertain categorical data. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 616–625. IEEE Computer Society, 2007.

[85] Sarvjeet Singh, Chris Mayfield, Rahul Shah, Sunil Prabhakar, Susanne E. Hambrusch, Jennifer Neville, and Reynold Cheng. Database support for probabilistic attributes and tuples. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 1053–1061. IEEE Computer Society, 2008.

[86] Zitan Sun, Xin Huang, Jianliang Xu, and Francesco Bonchi. Efficient probabilistic truss indexing on uncertain graphs. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 354–366. ACM / IW3C2, 2021.

[87] Yufei Tao, Reynold Cheng, Xiaokui Xiao, Wang Kay Ngai, Ben Kao, and Sunil Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 922–933. ACM, 2005.

[88] Yufei Tao, Xiaokui Xiao, and Reynold Cheng. Range search on multidimensional uncertain data. *ACM Trans. Database Syst.*, 32(3):15, 2007.

[89] The Computational Pan-Genomics Consortium. Computational pangenomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018.

[90] Gerry Tonkin-Hill, Inigo Martincorena, Roberto Amato, Andrew RJ Lawson, Moritz Gerstung, Ian Johnston, David K Jackson, Naomi Park, Stefanie V Lensing, Michael A Quail, Sónia Gonçalves, Cristina Ariani, Michael Spencer Chapman, William L Hamilton, Luke W Meredith, Grant Hall, Aminu S Jahun, Yasmin Chaudhry, Myra Hosmillo, Malte L Pinckert, Iliana Georgana, Anna Yakovleva, Laura G Caller, Sarah L Caddy, Theresa Feltwell, Fahad A Khokhar, Charlotte J Houldcroft, Martin D Curran, Surendra Parmar, The COVID-19 Genomics UK (COG-UK) Consortium, Alex Alderton, Rachel Nelson, Ewan M Harrison, John Sillitoe, Stephen D Bentley, Jeffrey C Barrett, M Estee Torok, Ian G Goodfellow, Cordelia Langford, Dominic Kwiatkowski, and Wellcome Sanger Institute COVID-19 Surveillance Team. Patterns of within-host genetic diversity in SARS-CoV-2. *eLife*, 10:e66857, aug 2021.

[91] Thanh T. L. Tran, Liping Peng, Yanlei Diao, Andrew McGregor, and Anna Liu. CLARO: modeling and processing uncertain data streams. *VLDB J.*, 21(5):651–676, 2012.

[92] Kazutoshi Umemoto, Ruihua Song, Jian-Yun Nie, Xing Xie, Katsumi Tanaka, and Yong Rui. Search by screenshots for universal article clipping in mobile apps. *ACM Trans. Inf. Syst.*, 35(4):34:1–34:29, 2017.

[93] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In Alexander Böhm and Tilmann Rabl, editors, *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 1:1–1:6. ACM, 2018.

[94] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.

[95] Aaron M. Wenger et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nat. Biotechnol.*, 37:1155–1162, 2019.

[96] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 262–276. www.cidrdb.org, 2005.

[97] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. Index-based optimal algorithm for computing k-cores in large uncertain graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 64–75. IEEE, 2019.

[98] Ke Yi, Feifei Li, George Kollios, and Divesh Srivastava. Efficient processing of top-k queries in uncertain databases with x-relations. *IEEE Trans. Knowl. Data Eng.*, 20(12):1669–1682, 2008.

[99] Liming Zhan, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Identifying top k dominating objects over uncertain data. In Sourav S. Bhowmick, Curtis E. Dyreson, Christian S. Jensen, Mong-Li Lee, Agus Muliantara, and Bernhard Thalheim, editors, *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part I*, volume 8421 of *Lecture Notes in Computer Science*, pages 388–405. Springer, 2014.

[100] Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement_1):i119–i127, 07 2020.