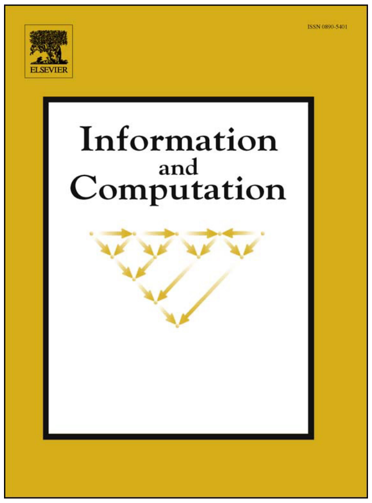
Journal Pre-proof



Indexing weighted sequences: Neat and efficient

Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, Jakub Radoszewski

PII: S0890-5401(19)30078-1
DOI: https://doi.org/10.1016/j.ic.2019.104462
Reference: YINCO 104462

To appear in: *Information and Computation*

Received date: 15 May 2018
Revised date: 15 March 2019
Accepted date: 27 March 2019

Please cite this article as: C. Barton et al., Indexing weighted sequences: Neat and efficient, *Inf. Comput.* (2019), 104462, doi: https://doi.org/10.1016/j.ic.2019.104462.

Indexing Weighted Sequences: Neat and Efficient[☆]

Carl Barton[a], Tomasz Kociumaka[b,c,1], Chang Liu[d], Solon P. Pissis[e], Jakub Radoszewski[c,2]

[a]*European Bioinformatics Institute, Wellcome Genome Campus, Hinxton, Cambridge CB10 1SD, UK*
[b]*Department of Computer Science, Bar Ilan University, 5290002 Ramat Gan, Israel*
[c]*Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland*
[d]*Comprehensive Cancer Centre, King's College London, London SE1 9RT, UK*
[e]*Centrum Wiskunde & Informatica, 1098 XG Amsterdam, The Netherlands*

Abstract

In a *weighted sequence*, for every position of the sequence and every letter of the alphabet a probability of occurrence of this letter at this position is specified. Weighted sequences are commonly used to represent imprecise or uncertain data, for example in molecular biology, where they are known under the name of Position Weight Matrices. Given a probability threshold $\frac{1}{z}$, we say that a string P of length m occurs in a weighted sequence X at position i if the product of probabilities of the letters of P at positions $i, \ldots, i+m-1$ in X is at least $\frac{1}{z}$. In this article, we consider an *indexing* variant of the problem, in which we are to pre-process a weighted sequence to answer multiple pattern matching queries. We present an $\mathcal{O}(nz)$-time construction of an $\mathcal{O}(nz)$-sized index for a weighted sequence of length n that answers pattern matching queries in the optimal $\mathcal{O}(m + Occ)$ time, where Occ is the number of occurrences reported. The cornerstone of our data structure is a novel construction of a family of $\lfloor z \rfloor$ strings that carries the information about all the strings that occur in the weighted sequence with a sufficient probability. We thus improve the most efficient previously known index by Amir et al. (*Theor. Comput. Sci.*, 2008) with size and construction time $\mathcal{O}(nz^2 \log z)$, preserving optimal query time. On the way we develop a new, more straightforward index for the so-called property matching problem. We provide an open-source implementation of our data structure and present experimental results using both synthetic and real data. Our construction allows us also to obtain a significant improvement over the complexities of the approximate variant of the weighted index presented by Biswas et al. at EDBT 2016 and an improvement of the space complexity of their general index. We also present applications of our index.

Keywords: weighted sequence, position weight matrix (PWM), text indexing, suffix tree, property indexing

1. Introduction

We consider a type of uncertain sequence called a *weighted sequence*. In a weighted sequence, every position contains a subset of the alphabet and every letter of the alphabet is associated with a probability of occurrence such that the sum of probabilities at each position equals 1.

[☆]This is a simplified and extended version of an article presented at the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016) without Chang Liu [1].

Email addresses: `carl@ebi.ac.uk` (Carl Barton), `kociumaka@mimuw.edu.pl` (Tomasz Kociumaka), `chang.2.liu@kcl.ac.uk` (Chang Liu), `solon.pissis@cwi.nl` (Solon P. Pissis), `jrad@mimuw.edu.pl` (Jakub Radoszewski)

[1]Supported by ISF grants no. 824/17 and 1278/16 and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

[2]Supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

Weighted sequences are common in a wide range of applications: (i) data measurements with imprecise sensor measurements; (ii) flexible sequence modelling, such as binding profiles of DNA sequences; (iii) observations that are private and thus sequences of observations may have artificial uncertainty introduced deliberately (see [2] for a survey). Pattern matching (or substring matching) is a core operation in a wide variety of applications including genome assembly, computer virus detection, database search, and short read alignment. Many of the applications of pattern matching generalise immediately to the weighted case as much of this data is more commonly uncertain (e.g. reads with quality scores) than certain.

In the *Weighted Pattern Matching* (WPM) problem, we are given a string P called a pattern, a weighted sequence X called a text, both over an alphabet Σ, and a *threshold probability* $\frac{1}{z}$. The task is to find all positions i in X where the product of probabilities of the letters of P at positions $i, \dots, i+|P|-1$ in X is at least $\frac{1}{z}$. Each such position is called an *occurrence* of the pattern; we also say that the fragment of X and the pattern *match*.

In this article, we consider the indexing (or off-line) version of the WPM problem, called *Weighted Indexing*. Here we are given a text being a weighted sequence and we are asked to construct a data structure (called an *index*) to provide efficient operations for answering WPM queries related to the text. We also consider other variants of the indexing problem. In the *Approximate Weighted Indexing* problem, given a pattern and a threshold z', we are to report all occurrences of the pattern with probability at least $\frac{1}{z'}$ but we may also report additional occurrences with probability $\frac{1}{z'} - \epsilon$, for a pre-selected value of $\epsilon > 0$. Note that in this problem there is no pre-selected threshold z. In the *Generalised Weighted Indexing* problem, we are to construct a data structure that allows for WPM queries to be answered for any threshold z' with $z' \le z$.

A problem that is known to be closely related to the Weighted Indexing problem is *Property Indexing*, first introduced in [3]. In this problem, we are given a string S called the text and a *hereditary property* Π, which is a family of integer intervals contained in $\{1, \dots, |S|\}$ (hereditary means that it is closed under subintervals). Our goal is to pre-process the text so that, for a query string P, we can report all occurrences of P in S which, interpreted as intervals, belong to Π. The property Π can be represented in $\mathcal{O}(|S|)$ space using an array $\pi[1\,..\,|S|]$ such that the longest interval starting at position i is $\{i, \dots, \pi[i]\}$.

In each of the indexing problems, we denote the length of the text by n, the length of a query pattern by m, and the number of occurrences of the pattern in the text by Occ.

1.1. Previous Results

An $\mathcal{O}(n \log m)$-time solution for the WPM problem over constant-sized alphabet based on the Fast Fourier Transform was proposed in [4, 5]. Recently, an $\mathcal{O}(n \log z)$-time solution using the suffix array and lookahead scoring was presented in [6]; on-line algorithms for this problem were also proposed [7]. The average case complexity of the WPM problem has also been studied and a number of fast algorithms have been presented for certain values of *weight ratio* $\frac{z}{m}$ [8, 9].

The Weighted Indexing problem was first considered by Iliopoulos et al. [10], who introduced a data structure called a *weighted suffix tree* allowing optimal $\mathcal{O}(m + Occ)$-time queries. The construction time and size of that data structure was, however, $\mathcal{O}(n|\Sigma|^{z \log z})$. Their data structure is a compact trie of all of the factors with the probability of occurrence greater than or equal to $\frac{1}{z}$.

Amir et al. [3] reduced the Weighted Indexing problem to the Property Indexing problem in a text of length $\mathcal{O}(nz^2 \log z)$. For the latter, they proposed a solution with $\mathcal{O}(n \log \log n)$ preprocessing time and optimal $\mathcal{O}(m + Occ)$ query time. Later, it was shown that Property Indexing can be solved in linear time; see [11, 12, 13] (see also [14]). This led to a solution to Weighted Indexing with index size and construction time $\mathcal{O}(nz^2 \log z)$, preserving optimal query time.

Biswas et al. [15] presented a data structure that solves the Approximate Weighted Indexing problem in $\mathcal{O}(\frac{1}{\epsilon} nz^2)$ space (with $\Omega(\frac{1}{\epsilon} n^2 z^2)$ construction time) with $\mathcal{O}(m + Occ)$-time queries; here Occ denotes the number of occurrences reported. They also proposed a data structure for the Generalised Weighted Indexing problem with $\mathcal{O}(nz^2 \log z)$ space and $\mathcal{O}(m + m \cdot Occ)$ query time. The construction time is not mentioned, but a direct construction of their index works in $\Omega(n^2 z^2)$ time. Moreover, they also consider the problem of document listing for weighted sequences.

1.2. Our Contributions

In a preliminary version of this article [1], we presented an $\mathcal{O}(nz)$-sized data structure for the Weighted Indexing problem that can be constructed in $\mathcal{O}(nz)$ time. The query time is still $O(m + Occ)$. The construction works for integer alphabets; however, for non-constant-sized alphabets the construction algorithm becomes randomized.

In this article, we present a new $\mathcal{O}(nz)$-time construction of an $\mathcal{O}(nz)$-sized data structure for the Weighted Indexing problem that answers queries in the optimal $\mathcal{O}(m + Occ)$ time. Our index is based on a novel observation that one can always construct a family of $\lfloor z \rfloor$ strings of length n that carries all the information about all the strings that occur in the weighted sequence. This yields a significantly simpler construction than in the previous index [1] preserving all of its applications. The construction works for integer alphabets and the construction time is worst-case. As a by-product, we obtain an optimal solution to the Property Indexing problem that avoids complex tools used in the previous solutions [3, 11, 12, 14].

We also discuss an even simpler randomised construction with worse space complexity and construction time of the weighted index. We provide an open-source implementation of our data structure and present experimental results using both synthetic and real data.

Our approach lets us significantly improve upon the variants of the weighted index proposed in [15]. In the Approximate Weighted Indexing problem, we obtain $\mathcal{O}(\frac{n}{\epsilon})$ space and $\mathcal{O}(\frac{n}{\epsilon} \log \frac{n}{\epsilon})$ construction time, preserving the query time. We improve the space usage in the Generalised Weighted Indexing problem to $\mathcal{O}(nz)$, also in the document listing variant.

We also present two applications of our index: computation of a weighted prefix table (and, more generally, answering longest common prefix queries on a weighted text) and computation of covers of a weighted text. In both cases, we obtain $\mathcal{O}(nz)$-time algorithms which improve upon the complexities of previously known algorithms (see [16] and [10, 3], respectively) by a factor of $z \log z$.

1.3. Comparison of Our Techniques with the Previous Work

Two main building blocks of our weighted index are a construction of a family of $\lfloor z \rfloor$ solid (standard) strings with properties and a solution to the Property Indexing problem.

The family of strings that we construct has the same set of patterns occurring at each position (respecting the properties) as the weighted text X and, moreover, the number of occurrences of each pattern at each position is a good estimate of the probability of its occurrence at this position in X. The former is used in the construction of a weighted index and the latter in the construction of an approximate weighted index. The existence of such a family is not immediate. However, our proof is not involved and we design an $\mathcal{O}(nz)$-time elementary construction algorithm of such a family based on *tries* (also known as radix trees). In the end, we show that a simple generation of a number of strings according to the probability distribution implied by the weighted text with high probability yields a family of strings that also well describes the set of patterns in X. However, the number of strings that one needs to generate is much larger. Excluding the exponential-size index of Iliopoulos et al. [10], previous work includes the $\mathcal{O}(nz^2 \log z)$-space index of Amir et al. [3] and our $\mathcal{O}(nz)$-space index [1]. Amir et al. [3] show that, after a small modification of the weighted text, the set of maximal string patterns that occur in it has a total length $\mathcal{O}(nz^2 \log z)$. In [1] we show a representation of this set as a trie and apply Shibuya's algorithm for suffix tree of a trie construction [17].

In our solution to the Property Indexing problem, we construct a data structure called *property suffix tree* being the suffix tree in which the nodes corresponding to factors that do not belong to the property are trimmed. The algorithm makes only several traversals of the suffix tree and uses an amortisation argument similar to the one from Ukkonen's suffix tree construction [18]. Very similar data structures were constructed by Amir et al. [3] and Kopelowitz [14]. Amir et al. [3] use a heavy machinery of weighted ancestor queries and a fancy algorithm to mark the properties on edges of the suffix tree. Kopelowitz [14] designs an algorithm for a dynamic setting, but also mentions its static application. He uses amortisation ideas similar to ours, but his construction is more involved due to its generality and also utilises less basic longest common extension queries (i.e., range minimum queries). The solution to the Property Indexing problem that was developed by Iliopoulos et al. [11] and clarified by Juan et al. [12] constructs a different data structure that, in a sense, shifts the hardness of the problem from the construction to the queries. It also requires range minimum queries. Charalampopoulos et al. have recently introduced the *property suffix array* [13].

Our techniques let us immediately answer decision queries of a weighted index. To answer counting and reporting queries in optimal time, we require coloured range counting and reporting data structures in the property suffix tree that were already used for this purpose in our previous construction [1]. In our solution to the Approximate Weighted Indexing, we need to augment the property suffix tree with a data structure for top-k document retrieval queries. The same type of queries was used in the previous solution by Biswas et al. [15], however, not as a black box. Moreover, they also use the less efficient reduction of [3] which caused their data structure to use $\mathcal{O}(\frac{1}{\epsilon}nz^2)$ space, assuming that $z' \le z$ in each query. Finally, we improve the space complexity of the generalised weighted index of Biswas et al. [15] by plugging in our novel construction of $\lfloor z \rfloor$ strings with properties.

1.4. Structure of the Article

In Section 3 we present a combinatorial construction of the special family of $\lfloor z \rfloor$ strings. An efficient implementation of the construction of this family based on tries is proposed in Section 4. In Section 5 a new optimal solution for the Property Indexing problem is presented. Using the construction and the property index, we obtain our weighted index in Section 6. Some applications of our index are presented in Section 7. With the aid of an auxiliary tool, an approximate weighted index is obtained in Section 8. Alternative randomised constructions of the two indexes with worse parameters are discussed in Section 9. In Section 10, we present an experimental evaluation of our weighted index construction. Our improvement to the generalised weighted index is briefly discussed in Section 11.

2. Preliminaries

A *string* S over an alphabet Σ is a finite sequence of letters from Σ. By $n = |S|$ we denote the length of S and by $S[i]$, for $1 \le i \le n$, we denote the i-th letter of S. By $S[i\,..\,j]$ we denote the string $S[i]\cdots S[j]$ called a *factor* of S (if $i > j$, then the factor is the empty string ε). A factor is called a *prefix* if $i = 1$ and a *suffix* if $j = n$. We say that a string P *occurs* at position i in S if $P = S[i\,..\,i + |P| - 1]$.

A *property* Π of S is a hereditary collection of integer intervals contained in $\{1, \dots, n\}$. 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 $\{i, \dots, \pi[i]\}$. Observe that π can be an arbitrary array satisfying $\pi[i] \in \{i - 1, \dots, n\}$ and $\pi[1] \le \pi[2] \le \cdots \le \pi[n]$. For a string P, by $Occ_\pi(P, S)$ we denote the set of occurrences i of P in S such that $i + |P| - 1 \le \pi[i]$. These notions lead us to the statement of the Property Indexing problem.

Problem (Property Indexing)**.**
Input: A string S of length n over an alphabet Σ and an array π representing a property Π.
Queries: For a given pattern string P of length m, compute $|Occ_\pi(P, S)|$ or report all elements of $Occ_\pi(P, S)$.

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

$$Count_{\mathcal{S}}(P, i) = |\{j : i \in Occ_{\pi_j}(P, S_j)\}|$$

we denote the total number of occurrences of P at position i in the strings $S_1, \dots, S_k$ that respect the properties.

A weighted sequence $X = x_1, x_2, \dots, x_n$ of length $|X| = n$ over an alphabet Σ is a sequence of sets of pairs of the form $x_i = \{(c,\ p_i^{(X)}(c)) \ :\ c \in \Sigma\}$. Here, $p_i^{(X)}(c)$ is the occurrence probability of the letter c at the position $i \in \{1, \dots, n\}$. These values are non-negative and sum up to 1 for a given i. An example of a weighted sequence is shown in Table 1 (running example).

The *probability of matching* of a string P at position i of a weighted sequence X equals

$$Prob_X(P, i) = \prod_{j=1}^{|P|} p_{i+j-1}^{(X)}(P[j]).$$

i	1	2	3	4	5	6
$p_i^{(X)}(\mathtt{A})$	1	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{4}{5}$	$\frac{1}{2}$	$\frac{1}{4}$
$p_i^{(X)}(\mathtt{B})$	0	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{2}$	$\frac{3}{4}$

Table 1: A weighted sequence X of length 6 over $\Sigma = \{\mathtt{A}, \mathtt{B}\}$.

We say that a string P *occurs* in X at position i if $Prob_X(P, i) \geq \frac{1}{z}$. We also say that P is a *solid factor* of X (starting, occurring) at position i. By $Occ_{\frac{1}{z}}(P, X)$ we denote the set of all positions where P occurs in X. The main problem in this scope can be formulated as follows.

Problem (Weighted Indexing)**.**
Input: A weighted sequence X of length n over an alphabet Σ and a threshold $\frac{1}{z}$.
Queries: For a given pattern string P of length m, check if $Occ_{\frac{1}{z}}(P, X) \neq \emptyset$ (*decision* query), compute $|Occ_{\frac{1}{z}}(P, X)|$ (*counting* query), or report all elements of $Occ_{\frac{1}{z}}(P, X)$ (*reporting* query).

Our model of computations. We assume the word-RAM model with word size $w = \Omega(\log(nz))$. We consider the log-probability model of representations of weighted sequences in which probabilities can be multiplied exactly in $\mathcal{O}(1)$ time. Without loss of generality, we further assume that each position contains at most $\lceil z \rceil$ characters with non-zero probability. This is because characters c with $p_i^{(X)}(c) < \frac{1}{z}$ can be merged into a dummy character $\$_i$. Consequently, a weighted sequence of length n has a representation using $\mathcal{O}(nz)$ space. We further assume that $\Sigma \subseteq \{1, \ldots, \mathcal{O}(nz)\}$ consists of positive integers.

3. Existence of an Equivalent Family of Strings

In the definition below, we formally characterise a string family that we aim to construct.

Definition 1. We say that an indexed family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ containing strings S_j of length n is a *z-estimation* of a weighted sequence X of length n if and only if, for every string P and position $i \in \{1, \ldots, n\}$, $Count_{\mathcal{S}}(P, i) = \lfloor Prob_X(P, i) z \rfloor$.

Note that a z-estimation $\mathcal{S}$ of a weighted sequence X carries the information about all solid factors of X: a string P occurs in X at position i if and only if it occurs at position i in at least one of the strings S_j respecting its property π_j. This observation will be used in the construction of our weighted index. Moreover, the value $Count_{\mathcal{S}}(P, i)$ provides a good estimation of the probability $Prob_X(P, i)$:

$$\tfrac{1}{z} Count_{\mathcal{S}}(P, i) \leq Prob_X(P, i) < \tfrac{1}{z} Count_{\mathcal{S}}(P, i) + \tfrac{1}{z}.$$

This will let us design an approximate weighted index. An example of a z-estimation is shown in Table 2 (running example). Below, we prove the existence of a z-estimation. An efficient construction is deferred to the next section.

For a fixed weighted sequence X of length n and a threshold z, we can use compact notation:

$$t_i(P) = \lfloor Prob_X(P, i) z \rfloor \quad \text{and} \quad m_i(P) = t_i(P) - \sum_{c \in \Sigma} t_i(Pc)$$

for $i = 1, \ldots, n$. (Note that Pc denotes the concatenation of P and the letter c.) We start with an equivalent characterisation of z-estimations of X.

Observation 3.1. *A family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ is a z-estimation of X if and only if for each position i, every string P is a prefix of exactly $t_i(P)$ strings $S_j[i \mathinner{.\,.} \pi_j[i]]$.*

Next, we prove that this condition uniquely defines the multiset

$$\{S_j[i \mathinner{.\,.} \pi_j[i]] : 1 \leq j \leq \lfloor z \rfloor\}.$$

i	1	2	3	4	5	6
$S_1[i]$	A	A	A	A	A	A
$\pi_1[i]$	2	2	3	4	5	6
$S_2[i]$	A	A	A	A	A	B
$\pi_2[i]$	4	4	5	6	6	6
$S_3[i]$	A	B	A	A	B	B
$\pi_3[i]$	4	4	5	6	6	6
$S_4[i]$	A	B	B	B	B	B
$\pi_4[i]$	2	2	3	3	5	6

string P	$Prob_X(P,3)$	$\{j : 3 \in Occ_{\pi_j}(P, S_j)\}$
ε	1	1, 2, 3, 4
A	0.75	1, 2, 3
AA	0.6	2, 3
AAA	0.3	2
AAB	0.3	3
B	0.25	4

Table 2: To the left: a 4-estimation $\mathcal{S}$ of the weighted sequence X from Table 1. To the right: all the strings that occur at position $i = 3$ in X together with the probabilities of occurrence in X and occurrences in $\mathcal{S}$.

Lemma 3.2. *There exists a unique multiset $\mathcal{M}_i$ such that each string P is a prefix of exactly $t_i(P)$ strings in $\mathcal{M}_i$.*

Proof. Consider a multiset $\mathcal{M}_i$ satisfying the required condition and an arbitrary string P. For each $c \in \Sigma$, there are $t_i(Pc)$ strings in $\mathcal{M}_i$ with the prefix P followed by a letter c. In the remaining $t_i(P) - \sum_{c\in\Sigma} t_i(Pc)$ strings in $\mathcal{M}_i$, the prefix P it is not followed by any letter. Thus, the multiplicity of P in $\mathcal{M}_i$ must be $m_i(P)$. This implies uniqueness of $\mathcal{M}_i$.

Note that $t_i(P) \geq \sum_{c\in\Sigma} t_i(Pc)$ because $Prob_X(P,i) \geq \sum_{c\in\Sigma} Prob_X(Pc,i)$ and the function $x \mapsto \lfloor p \cdot x \rfloor$ is superadditive[3]. Consequently, we may define a multiset $\mathcal{M}_i$ using values $m_i(P)$ as multiplicities. It remains to prove that this multiset satisfies the required condition. For this, we consider strings P in the order of decreasing lengths. The base case is trivial because strings P longer than X satisfy $Prob_X(P,i) = 0$. The inductive hypothesis yields that, for each $c \in \Sigma$, the string Pc is a prefix of $t_i(Pc)$ strings in $\mathcal{M}_i$. Consequently, the string P is a prefix of $m_i(P) + \sum_{c\in\Sigma} t_i(Pc) = t_i(P)$ strings in $\mathcal{M}_i$, as claimed. $\square$

Observe that in a z-estimation, $S_j[i\,..\,\pi_j[i]]$ can be obtained from $S_j[i+1\,..\,\pi_j[i+1]]$ by inserting a leading letter and dropping some number of trailing letters. (This includes dropping the newly inserted letter if $S_j[i\,..\,\pi_j[i]] = \varepsilon$.) The relation between these strings can be formalised as follows.

Definition 2. We say that $P \in \mathcal{M}_i$ is *compatible* with $Q \in \mathcal{M}_{i+1}$ if $P = \varepsilon$ or $P = cQ'$ for some letter $c \in \Sigma$ and a prefix Q' of Q.

Thus, if a z-estimation exists, it yields a perfect matching between $\mathcal{M}_{i+1}$ and $\mathcal{M}_i$ such that the matched strings are compatible. We prove that such a matching exists unconditionally. For an example, see Table 3 (running example).

[3]A function $f(x)$ is *superadditive* if $f(x+y) \geq f(x) + f(y)$ for all arguments x and y.

index	$\mathcal{M}_1$		$\mathcal{M}_2$		$\mathcal{M}_3$		$\mathcal{M}_4$		$\mathcal{M}_5$		$\mathcal{M}_6$
1	AA	—	A	—	A	—	A	—	A	—	A
2	AAAA	—	AAA	—	AAA	—	AAB	—	AB	—	B
3	ABAA	—	BAA	—	AAB	—	ABB	—	BB	—	B
4	AB	—	B	—	B	—	ε	—	B	—	B

Table 3: The sets $\mathcal{M}_i$ for the weighted sequence X from Table 1 with $z = 4$. Perfect matchings of compatible strings between $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$ are marked. The first letters of the strings form the 4-estimation from Table 2 and the length of the j-th string in $\mathcal{M}_i$ corresponds to $\pi_j[i] - i + 1$.

Lemma 3.3. *For every $1 \le i \le n-1$, there exists a one-to-one correspondence from $\mathcal{M}_{i+1}$ into $\mathcal{M}_i$ such that each $Q \in \mathcal{M}_{i+1}$ is matched with a compatible $P \in \mathcal{M}_i$.*

Proof. We greedily transform each $Q \in \mathcal{M}_{i+1}$ into the longest compatible $P \in \mathcal{M}_i$ which is still unmatched. If no compatible $P \in \mathcal{M}_i$ is available, we leave Q unmatched. We will show that all strings $Q \in \mathcal{M}_{i+1}$ are actually matched at the end of this process. Since $|\mathcal{M}_i| = t_i(\varepsilon) = \lfloor z \rfloor = t_{i+1}(\varepsilon) = |\mathcal{M}_{i+1}|$, it suffices to prove that no $P \in \mathcal{M}_i$ is left unmatched.

An empty string $P \in \mathcal{M}_i$ is compatible with every $Q \in \mathcal{M}_{i+1}$, so it cannot be left unmatched. Thus, suppose that $P = cQ' \in \mathcal{M}_i$, for some $c \in \Sigma$ and string Q', is left unmatched. Let us denote by $\mathcal{R}$ the multiset containing all strings $Q \in \mathcal{M}_{i+1}$ compatible with P, i.e., starting with Q'. We further define $\mathcal{L}$ as the multiset containing all strings $P' \in \mathcal{M}_i$ that start with $c'Q'$ for some $c' \in \Sigma$. The construction procedure guarantees that each $Q \in \mathcal{R}$ has been matched to a compatible P' satisfying $|P'| \ge |P|$; such P' must belong to the multiset $\mathcal{L}$.

Observe that $|\mathcal{L}| = \sum_{c' \in \Sigma} t_i(c'Q') \le t_{i+1}(Q') = |\mathcal{R}|$ because $\mathit{Prob}_X(Q', i+1) = \sum_{c' \in \Sigma} \mathit{Prob}_X(c'Q', i)$ and the function $x \mapsto \lfloor xz \rfloor$ is superadditive. Consequently, each $P' \in \mathcal{L}$ must be matched to some $Q \in \mathcal{R}$. Since $P \in \mathcal{L}$ is unmatched, we obtain a contradiction. □

Due to Lemma 3.3, we can index the strings $\mathcal{M}_i = \{P_{j,i} : 1 \le j \le \lfloor z \rfloor\}$ so that we have $\lfloor z \rfloor$ chains $P_{j,1}, \ldots, P_{j,n}, P_{j,n+1} = \varepsilon$ with compatible subsequent strings. It is easy to transform each such chain to a string S_j with property π_j so that $S_j[i \mathinner{.\,.} \pi_j[i]] = P_{j,i}$. The value $S_j[i]$ is not specified if $P_{j,i} = \varepsilon$; in this case, we may set $S_j[i]$ to an arbitrary letter. The resulting family $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ clearly satisfies the characterisation of Observation 3.1, which completes the proof of the following result.

Theorem 3.4. *Each weighted sequence has a z-estimation.*

4. Efficient Implementation

In this section, we describe an algorithm which, given a weighted sequence X of length n and a threshold z, constructs a z-estimation of X in $\mathcal{O}(nz)$ time.

At a high level, we follow the existential construction of Section 3. We start with $\mathcal{M}_{n+1}$, which consists of $\lfloor z \rfloor$ copies of ε, and we iterate over positions $i = n, \ldots, 1$ transforming $\mathcal{M}_{i+1}$ to $\mathcal{M}_i$ so that each $P_{j,i+1} \in \mathcal{M}_{i+1}$ is replaced with a compatible string $P_{j,i} \in \mathcal{M}_i$. We simultaneously build the z-estimation $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$. More precisely, we set $\pi_j[i]$ to $i + |P_{j,i}| - 1$ and $S_j[i]$ to the leading letter of $P_{j,i}$, or an arbitrary letter if $P_{j,i} = \varepsilon$.

Each transformation follows the lines of the procedure described in the proof of Lemma 3.3. However, our implementation uses *solid factor tries* in order to achieve $\mathcal{O}(z)$ amortised running time. We introduce this auxiliary data structure in Section 4.1. Next, in Sections 4.2 and 4.3, we provide efficient implementations for constant-sized and integer alphabets, respectively.

4.1. Solid Factor Tries

Recall that a trie is a rooted tree in which each node represents a string; the string corresponding to node u, called the *label* of u, is denoted $\mathsf{L}(u)$. The root has label ε, and the parent of a node u with $\mathsf{L}(u) = Pc$ for $c \in \Sigma$ is the node v with $\mathsf{L}(v) = P$; the edge from P to Pc is *labelled* with c. Observe that the family of solid factors occurring at position i (i.e., strings P such that $t_i(P) > 0$) is closed with respect to prefixes. Thus, we can define a *solid factor trie* T_i whose nodes represent these factors.

We store $\mathcal{M}_i$ using *tokens* in T_i: each $P_{j,i} \in \mathcal{M}_i$ is represented by a token (with identifier j) located at the node $u \in T_i$ with $\mathsf{L}(u) = P_{j,i}$. For each token j, we store the node $u \in T_i$ with $\mathsf{L}(u) = P_{j,i}$ and the probability $\mathit{Prob}_X(P_{j,i}, i)$. Observe that the number of tokens at the node u is $m_i(\mathsf{L}(u))$ and the number of tokens in the subtree rooted at u is $t_i(\mathsf{L}(u))$. To simplify notation, we denote $m_i(u) = m_i(\mathsf{L}(u))$ and $t_i(u) = t_i(\mathsf{L}(u))$. The observation below follows from the properties of m_i and t_i; for an example, see Figure 1 (running example).

Observation 4.1. *The trie T_i contains $\lfloor z \rfloor$ tokens in total and every leaf contains a token.*

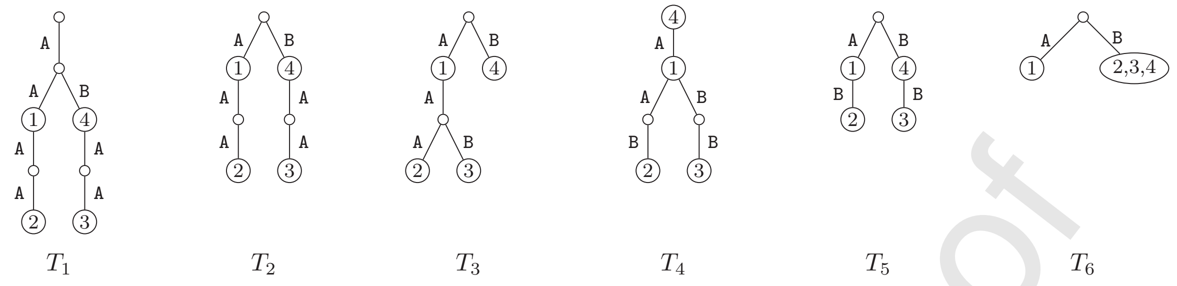


Figure 1: The solid factor tries for the weighted sequence X from Table 1 with $z = 4$. Tokens in the nodes are numbered according to the order from Table 3.

4.2. Transformation Algorithm for Alphabets of Constant Size

For each index i, we transform the solid factor trie T_{i+1} to T_i and move the tokens so that $\mathcal{M}_{i+1}$ is transformed to $\mathcal{M}_i$. Before we describe the implementation, let us formulate a relation between T_i and T_{i+1}.

Observation 4.2. *If $u \in T_i$ has a non-empty label, $\mathsf{L}(u) = cP$, for some $c \in \Sigma$, then T_{i+1} contains a node v with label $\mathsf{L}(v) = P$.*

Consequently, each non-root node $u \in T_i$ has a corresponding node $v \in T_{i+1}$. In our construction algorithm, we sometimes reuse v as u; otherwise, we create u as a copy of v. More precisely, at position i we distinguish a *heavy letter* $h \in \Sigma$ maximising probability $p_i^{(X)}(c)$ over $c \in \Sigma$. We reuse v if $\mathsf{L}(u)$ starts with h and create a copy of v otherwise.

This approach is implemented as follows. First, we create the root of T_i and attach T_{i+1} to the new root using an edge with label h. The resulting subtree, denoted $T_{i,h}$, contains all tokens present in T_{i+1} and may contain nodes v with $t_i(v) = 0$ (we piggyback trimming them to the last phase when tokens are moved). Next, we consider all the remaining letters $c \in \Sigma \setminus \{h\}$. For each such letter, we shall build a subtree $T_{i,c}$ representing solid factors occurring at position i and starting with letter c. We simultaneously build and traverse $T_{i,c}$: we construct the children of a node u while visiting u for the first time. While at node u with $\mathsf{L}(u) = cP$, we maintain the probability $\mathit{Prob}_X(cP, i)$ and a pointer to the corresponding node $v \in T_{i,h}$ such that $\mathsf{L}(v) = hP$. To construct the children of u, we simply compute $t_i(cPc')$ for each $c' \in \Sigma$. Moreover, we determine $m_i(cP)$ and place $m_i(cP)$ *token requests* at node v, announcing that $m_i(cP)$ tokens are needed at u.

Finally, we move the tokens and trim the redundant nodes of $T_{i,h}$. We process the tokens in an arbitrary order. Consider a token located at node v of $T_{i,h}$ with $\mathsf{L}(v) = hQ$ (the token used to represent $Q \in \mathcal{M}_{i+1}$). Recall that we store $\mathit{Prob}_X(Q, i + 1)$ and $\mathit{Prob}_X(\mathsf{L}(v), i)$ is that number multiplied by $p_i^{(X)}(h)$. We traverse the path from v towards the root of T_i maintaining the probability $\mathit{Prob}_X(\mathsf{L}(v'), i)$ at the currently visited node v'. First, we check if there is any token request at v'. If so, we comply with the request, remove it, and terminate the traversal. Otherwise, we compute $m_i(v')$ using the probability. If v' contains less than $m_i(v')$ already processed tokens, we place our token at v' and terminate the traversal. Otherwise, we proceed to the parent of v'. If v' is a leaf and does not contain any (processed or unprocessed) tokens, we remove v' from $T_{i,h}$. If the traversal reaches the root of T_i, we place the token unconditionally at the root. For an example, see Figure 2 (running example).

4.2.1. Correctness

We shall prove that the procedure described above correctly computes T_i and $\mathcal{M}_i$. Due to Observation 4.2, each node of the trie T_i except for the root is uniquely determined by the corresponding node of the trie T_{i+1} and a letter c. The root of T_i is created explicitly. Each node of T_{i+1} is reused as its corresponding

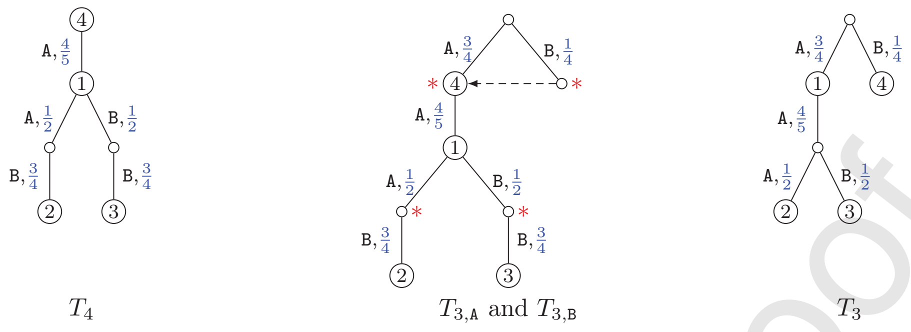


Figure 2: Transformation between T_4 to T_3 from the example in Figure 1. To the left: the trie T_4 with letter probabilities (in blue). In the middle: the trie T_4 is copied as $T_{3,\mathtt{A}}$, whereas $T_{3,\mathtt{B}}$ is created using a backtracking algorithm (in this case, it has only one node). Asterisks denote nodes that require tokens. The token request is shown with an arrow. To the right: the final T_3 created after the tokens are moved up and redundant nodes are removed. Note that the tokens number 1 and 4 could have been interchanged depending on the order of processing.

node in T_i for the heavy letter h (provided that such a node exists in T_i), and for each of the remaining letters we create its copies in a recursive descent. Hence, T_i contains all the necessary nodes.

We only need to prove that no redundant nodes v (with $t_i(v) = 0$) are left in $T_{i,h}$. Suppose that v is the deepest such node; clearly, it must be a leaf of $T_{i,h}$. We did not place the token at v because $m_i(v) \le t_i(v) = 0$. On the other hand, tokens were present in all leaves of T_{i+1}, so the subtree of v in $T_{i,h}$ initially contained a token. Let us consider the moment of moving the last token in this subtree. If the token travelled further to the parent of v, we would have removed v. Thus, the token must have been placed at a node u complying with a token request at v or its descendant. However, in that case we have $t_i(v) \ge t_i(u) \ge m_i(u) > 0$, because h is the heavy letter. This contradiction concludes the proof.

We proceed to prove that the final configuration of tokens represents $\mathcal{M}_i$. For this, we observe that our algorithm simulates the greedy procedure in the proof of Lemma 3.3. In other words, we shall prove that we transformed $P_{j,i+1} \in \mathcal{M}_{i+1}$ to the longest compatible element of $\mathcal{M}_i$ which was still unmatched when we processed token j. Suppose that there was an unmatched string $P' \in \mathcal{M}_i$ longer than $P_{j,i}$. Let $P' = cQ'$ and observe that, when processing token j, we visited the node v' with $\mathsf{L}(v') = hQ'$. If $c = h$, then we would have less than $m_i(v')$ processed tokens at v'. Otherwise, there must have been a token request at v'. In either event, we would not have proceeded to the parent of v'. This contradiction concludes the proof.

4.2.2. Running Time Analysis

It remains to show that the total running time of the n transformations is $\mathcal{O}(nz)$. In a single iteration, processing the j-th token, i.e., transforming $P_{j,i+1}$ to $P_{j,i}$, we visited at most $1+|P_{j,i+1}|-|P_{j,i}|$ nodes of $T_{i,h}$ and deleted some of them. Across all iterations this is $\mathcal{O}(n)$ per token and $\mathcal{O}(nz)$ in total. The remaining operations (construction of subtrees $T_{i,c}$) take $\mathcal{O}(|\Sigma|) = \mathcal{O}(1)$ time per created node. The final tree T_1 has $\mathcal{O}(nz)$ nodes and the overall number of deleted nodes is $\mathcal{O}(nz)$. Hence, the total number of created nodes is also $\mathcal{O}(nz)$.

This concludes the proof that the running time is $\mathcal{O}(nz)$. Hence, we achieve the main goal of this section.

Proposition 4.3. *For a weighted sequence of length n over a constant-sized alphabet, one can construct a z-estimation in $\mathcal{O}(nz)$ time.*

4.3. Transformation Algorithm for Integer Alphabets

For integer alphabets, the running time of the procedure above may increase to $\mathcal{O}(nz|\Sigma|)$ for the following two reasons:

1. Having created a node u with $\mathsf{L}(u) = cP$, we compute $t_i(cPc')$ for $c' \in \Sigma$.

2. Moving the token through a node v' with $\mathsf{L}(v') = hP$, we compute $m_i(v')$.

Below, we modify the algorithm so that it avoids these bottlenecks.

Before we run the algorithm, we preprocess the input weighted sequence.

Fact 4.4 ([3, Section 4.2]). *The weighted sequence X can be preprocessed in $\mathcal{O}(nz)$ time so that for any position i and probability $p \geq \frac{1}{z}$, the characters $\{c : p_i^{(X)}(c) \geq p\}$ can be listed with $\mathcal{O}(1)$ amortised time delay.*

Now, having created a node u of $T_{i,c}$ with $\mathsf{L}(u) = cP$, we only process characters c' for which a child u' with $\mathsf{L}(u') = cPc'$ needs to be created. In other words, these are characters c' such that $\mathit{Prob}_X(cPc', i) \geq \frac{1}{z}$ or, equivalently, $p^{(X)}_{i+|cP|}(c') \geq \frac{1}{z\mathit{Prob}_X(cP,i)}$. This way, the construction of the subtrees $T_{i,c}$ and the underlying token requests takes $\mathcal{O}(1)$ time per node created.

On the other hand, to facilitate moving tokens, we explicitly store $t_i(v)$ for each node v of T_i. These values are easily initialised while we build $T_{i,c}$, but for $v \in T_{i,h}$ with $\mathsf{L}(v) = hP$, some effort is needed to update the stored value from $t_{i+1}(P)$ to $t_i(v) = t_i(hP)$. This operation generalises trimming $T_{i,h}$ (i.e., identifying nodes with $t_i(v) = 0$), and we implement it while moving tokens.

In contrast to the case of constant-size alphabet, we move all tokens simultaneously, processing $T_{i,h}$ in a bottom-up fashion. Having processed a node v with $\mathsf{L}(v) = hP$, we report the probability $\mathit{Prob}_X(hP, i)$, the change $t_{i+1}(P) - t_i(hP)$, as well as a collection of tokens from the subtree of v that need to be moved to the parent of v (or above). We say that the output is *trivial* if the change is 0 and the collection is empty.

Observe that if a node v does not contain any token and if processing its children results in trivial outputs, then processing v also yields a trivial output. Hence, we do not need to process v in that case. In other words, we process a node only if it contains a token or processing one of its children reported a non-trivial output.

The operation of processing a node v with $\mathsf{L}(v) = hP$ is implemented as follows. First, we retrieve $\mathit{Prob}_X(hP, i)$: either directly from a token at v or based on $\mathit{Prob}_X(hPc, i)$ (if processing a child v' with $\mathsf{L}(v') = hPc$ resulted in a non-trivial output). Next, we use this probability to determine $t_i(v) = t_i(hP)$, and we store this value at v, keeping the change so that it can be reported. Then, we compute $m_i(v) = m_i(hP)$: we start from $m_{i+1}(P)$ (equal to the number of tokens at v) and update this value based on the changes $t_{i+1}(P) - t_i(hP)$ and $t_{i+1}(Pc) - t_i(hPc)$ for children v' of v with $\mathsf{L}(v') = hPc$. As the last step, we actually process the tokens: we make a single collection C out of the tokens stored at v and reported by the children of v, handle the token requests at v, and place $m_i(v)$ tokens at v. The remaining tokens are forwarded to the parent of v.

For a single node v, the running time of the procedure above is proportional to $|C|$ plus the number of children of v with non-trivial output. The latter can be accounted to processing these children, so the amortised time is $\mathcal{O}(\max(1, |C|))$. If $|C| = 0$, then t_i must have decreased at v due to a token request in the subtree of v. Hence, processing such a node can be accounted to the creation of a copy of v during the construction of the trees $T_{i,c}$. On the other hand, the total size of collections C is proportional to the total distance traversed by tokens.

At this point we can finally use the amortisation designed for $|\Sigma| = \mathcal{O}(1)$ to prove that overall running time across all iterations i is $\mathcal{O}(nz)$.

Theorem 4.5. *For a weighted sequence of length n over an integer alphabet, one can construct a z-estimation in $\mathcal{O}(nz)$ time.*

5. Property Indexing Made Simple

Most known solutions to the Property Indexing problem make use of suffix trees; our solution makes use of suffix trees as well. Thus, below we recall the basics on suffix trees.

5.1. Suffix Trees

The *suffix tree* T of a non-empty string S of length n is a compact trie representing all suffixes of S. The nodes of the trie which become nodes of the suffix tree (i.e., branching nodes, terminal nodes, and the root) are called *explicit* nodes, while the other nodes are called *implicit*. The edges out-going from a node are numbered with their first letters and can be stored, e.g., in a list.

Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Then, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We use $\mathsf{L}(v)$ to denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v, and denote $depth(v) = |\mathsf{L}(v)|$. The terminal node corresponding to suffix $S[i \mathinner{.\,.} n]$ is marked with the index i. Each string P occurring in S is uniquely represented by either an explicit or an implicit node of T, called the *locus* of P. The *suffix link* of a node v with path-label $\mathsf{L}(v) = cP$ is a pointer to the node path-labelled P, where $c \in \Sigma$ is a single letter and P is a string. The suffix link of every non-root explicit node v leads to an explicit node of T.

The suffix tree of a string of length n over an integer alphabet Σ can be constructed in $\mathcal{O}(n)$ time [19, 20].

5.2. Property Suffix Tree Construction

In analogy to the suffix tree, given a string S with property Π represented by an array π, we define the *property suffix tree* of (S, π) as the compact trie representing strings $S[i \mathinner{.\,.} \pi[i]]$. Each terminal node v stores a list L_v containing all indices i such that $S[i \mathinner{.\,.} \pi[i]]$ is the path-label of v. This way, $Occ_\pi(P, S)$ can be retrieved by locating the locus of P and writing down indices in lists L_v for all descendants v of the locus.

For a given string S, we construct the property suffix tree with respect to property Π from the suffix tree of S. This process is implemented in three steps. First, for each index i we determine the locus v_i of $S[i \mathinner{.\,.} \pi[i]]$. Next, we make all these loci explicit to create new terminal nodes. Finally, we remove nodes which should no longer exist in the tree or no longer be explicit.

Our approach to the first phase is similar to Ukkonen's suffix tree construction [18]. We are to determine the locus v_i of $S[i \mathinner{.\,.} \pi[i]]$. For this, we shall traverse the suffix tree starting from an explicit node u_i guaranteed to be an ancestor of v_i. We obtain u_i by following the suffix link of the nearest explicit ancestor of v_{i-1} (v_{i-1} itself if it is explicit). If $i = 1$ or the explicit ancestor of v_{i-1} is the root, we simply set u_i as the root. Since $\pi[i] \geq \pi[i-1]$ for $i > 1$, u_i is indeed an ancestor of v_i. Therefore, we can progress down the edges in the suffix tree from u_i, keeping track of the current depth until the desired depth is reached. We know that v_i exists in the tree, so it suffices to inspect only the first letter of each traversed edge.

This procedure results in the sequence of loci $(v_i)_{i=1}^{n}$. Let us analyse its time complexity. In the i-th iteration we traverse: one edge to reach u_i, then several edges to a node whose suffix link is u_{i+1}, and finally at most one edge to reach v_i. Hence, the number of edges traversed in this iteration is at most $2 + |\mathsf{L}(u')| - |\mathsf{L}(u_i)| \leq 3 + |\mathsf{L}(u_{i+1})| - |\mathsf{L}(u_i)|$, which gives $\mathcal{O}(n)$ overall.

The remaining steps of the algorithm are performed as follows. We sort the loci v_i by the path label length $\pi[i] - i + 1$ and group them based on the edge where they are located using bucket sort. This lets us appropriately subdivide each edge and compute the lists L_v for the new terminal nodes. Finally, we trim the tree: we traverse the tree bottom-up and remove or dissolve nodes which should no longer be explicit. These steps clearly work in $\mathcal{O}(n)$ time.

5.3. The Case of Integer Alphabet

The above construction also works for an integer alphabet, however, it requires traversing the suffix tree of S downwards. For integer alphabets, one would need to use perfect hashing [21] to efficiently index children of a node and, thus, introduce randomisation. This can be avoided with a different trimming algorithm based on the *weighted ancestor* problem. In this problem, introduced by Farach and Muthukrishnan [22] (see also [23]), we consider a rooted tree T with an integer weight function μ defined on the nodes. We require that the weight of the root is zero and the weight of any other node is strictly larger than the weight of its parent. A weighted ancestor query, given a node v and an integer value $\ell \leq \mu(v)$, asks for the highest ancestor u of v such that $\mu(u) \geq \ell$, i.e., such an ancestor u that $\mu(u) \geq \ell$ and $\mu(u)$ is smallest possible.

We define the weight of a node of the suffix tree of S as the length of the string it represents. Thus a weighted ancestor query can be used for the terminal node corresponding to $S[i\,.\,.\,n]$ to and mark the node that corresponds to $S[i\,.\,.\,\pi[i]]$. The remaining steps of the algorithm stay the same.

In the online setting, one can answer weighted ancestor queries in $\mathcal{O}(\log\log n)$ time after $\mathcal{O}(n)$-time and space preprocessing [24] (see also [22, 25]). In the special case of the weighted tree being a suffix tree of a string, they can be answered in $\mathcal{O}(1)$ time with a data structure of $\mathcal{O}(n)$ space [23] (however, the preprocessing time of this data structure is not stated in the paper). We use an algorithm from [26] where it is shown that in an offline setting $\mathcal{O}(n)$ such queries can be answered in $\mathcal{O}(n)$ time.

Counting and reporting queries on the property suffix tree are answered as on a suffix tree. We arrive at the following theorem.

Theorem 5.1. *For a string S and property Π represented with a table π, the property suffix tree can be computed in $\mathcal{O}(n)$ time. Moreover, this data structure can answer property indexing queries in $\mathcal{O}(m)$ time (counting) or $\mathcal{O}(m + |Occ_\pi(P,S)|)$ time (reporting).*

6. Weighted Index

Let us first describe our data structure for the Weighted Indexing problem. For a weighted sequence X and a threshold z, we construct a z-estimation $\mathcal{S} = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ of X, concatenate all the strings and shift the properties so that a single string S with property π is obtained. Our weighted index is the property suffix tree of S and π. In the property suffix tree, each terminal node is labelled by the list of all the occurrences of the corresponding string in S respecting its property. We shift these indices so that they describe the indices within the respective strings S_j.

We assume left-to-right orientation of the children of each node (e.g., lexicographic). A global occurrence list OL is stored being a concatenation of the lists of occurrences in all the terminal nodes in pre-order. Each node v stores, as $OL(v)$, the occurrence list of terminal nodes in its subtree represented as a pair of pointers to elements of the global list OL. We enhance the occurrence list OL by a data structure for the following coloured range listing problem.

Problem (Coloured range listing)**.** Preprocess a sequence $A[1\,.\,.\,N]$ of elements from $[1\,.\,.\,M]$ so that, given a range $A[i\,.\,.\,j]$, one can list all the distinct elements in that range.

Fact 6.1 (Muthukrishnan [27])**.** *A data structure for the coloured range listing problem of $\mathcal{O}(N)$ size can be constructed in $\mathcal{O}(N + M)$ time and answers queries in $\mathcal{O}(k + 1)$ time where k is the number of distinct elements reported.*

For all nodes we also compute the following values (for the purpose of this computation we replace each leaf v with $|OL(v)|$ bogus leaves with single occurrences).

Fact 6.2 (Colour set size; Hui [28])**.** *Given a rooted tree with N nodes whose L leaves have colours from $[1\,.\,.\,M]$, in $\mathcal{O}(N + M)$ time one can find for each node u the number of distinct leaf colours in the subtree of u.*

The space complexity of the index is obviously $\mathcal{O}(nz)$, where n is the length of X. Theorems 4.5 and 5.1 show that the data structure can be constructed in $\mathcal{O}(nz)$ time. The resulting weighted index is very similar to the one constructed in [1], even though the construction algorithm is very different.

By Definition 1, a string P occurs at position i in X if and only if it occurs at this position in at least one of the strings. Thus, to check if $Occ_{\frac{1}{z}}(P, X) \neq \emptyset$, it suffices to traverse down the property suffix tree and check if it contains an explicit or implicit node v corresponding to P. This search takes $\mathcal{O}(m)$ time, where $m = |P|$. We can use Fact 6.2 to equip each explicit node with the number of distinct positions where the string represented by the node occurs. This way, $|Occ_{\frac{1}{z}}(P, X)|$ can also be determined in $\mathcal{O}(m)$ time. With the aid of the data structure for coloured range listing, we can also report $Occ_{\frac{1}{z}}(P, X)$ in time proportional to the number of reported elements. We thus obtain the following result.

Theorem 6.3. *For a weighted sequence X of length n over an integer alphabet and a threshold z, there is a weighted index of $\mathcal{O}(nz)$ size that can be constructed in $\mathcal{O}(nz)$ time and answers decision and counting queries in $\mathcal{O}(m)$ time and reporting queries in $\mathcal{O}(m + |Occ_{\frac{1}{z}}(P, X)|)$ time.*

7. Applications of the Weighted Index

In this section we present two applications of the weighted index presented in Section 6. In both cases we improve the time complexity of the previously known results by a factor of $z \log z$. Let us denote our weighted index over X by $\mathcal{I}$.

7.1. Weighted Longest Common Prefixes and Weighted Prefix Table

For a weighted sequence X of length n and a pair of indices i, j, $1 \le i, j \le n$, by $wlcp(i, j)$ we denote the length of the longest solid factor that occurs in X at both positions i and j. After some preprocessing our weighted index allows to answer such queries in $\mathcal{O}(z)$ time.

Theorem 7.1. *Given a weighted sequence X of length n, after $\mathcal{O}(nz)$-time preprocessing we can answer $wlcp(i, j)$ queries for any $1 \le i, j \le n$ in $\mathcal{O}(z)$ time.*

Proof. For each position i in X we precompute the list of terminal nodes $L(i)$ of the weighted index $\mathcal{I}$ that contain i in their occurrence lists. Prior to that, all terminal nodes are numbered in pre-order, and the elements of $L(i)$ are stored in this order. By construction through a z-estimation, $|L(i)| \le z$ for each i.

Observe that $wlcp(i, j)$ is the maximum depth of a lowest common ancestor (lca) of a terminal node in $L(i)$ and a terminal node in $L(j)$. To determine this value, we merge the lists $L(i)$ and $L(j)$ according to the pre-order. The claim below (Lemma 4.6 in [20]) implies that, computing $wlcp(i, j)$, it suffices to consider pairs of terminal nodes that are adjacent in the resulting list.

Claim. If l_1, l_2 and l_3 are three nodes of a (compacted) trie such that l_2 follows l_1 and l_3 follows l_2 in pre-order, then

$$depth(lca(l_1, l_3)) = \min(depth(lca(l_1, l_2)), depth(lca(l_2, l_3))).$$

Merging two sorted lists, each of length at most z, takes $\mathcal{O}(z)$ time. Finally let us recall that lca-queries in a tree can be answered in $\mathcal{O}(1)$ time after linear-time preprocessing [29, 30]. $\square$

The weighted prefix table $WPT[1\..n]$ of X is defined as $WPT[i] = wlcp(1, i)$; see [16]. As a consequence of Theorem 7.1 we obtain an $\mathcal{O}(nz)$-time algorithm for computing this table. It outperforms the algorithm of [16], which works in $\mathcal{O}(nz^2 \log z)$ time.

Theorem 7.2. *The weighted prefix table WPT of a given weighted sequence of length n can be computed in $\mathcal{O}(nz)$ time.*

7.2. Efficient Computation of Covers

A *cover* of a weighted sequence X is a string P whose occurrences as solid factors of X cover all positions in X; see [10]. More formally, we define $maxgap$ of an ordered set of integers $A = \{a_1, \dots, a_k\}$ (with $a_1 < \dots < a_k$) as

$$maxgap(A) = \max\{a_i - a_{i-1} : i = 2, \dots, k\}.$$

(Note that this definition extends directly to multisets.) Then P is a cover of X if and only if

$$1 \in Occ_{\frac{1}{z}}(P, X) \quad \text{and} \quad maxgap(Occ_{\frac{1}{z}}(P, X) \cup \{n + 1\}) \le |P|.$$

Note that the former condition means exactly that P is a solid prefix of X. An $\mathcal{O}(n)$-time algorithm computing a representation of all the covers of a weighted sequence under the assumption that $z = \mathcal{O}(1)$ was presented in [10]. Using the results of [3] it could be improved to $\mathcal{O}(nz^2 \log z)$ time. Here we show an algorithm that works in $\mathcal{O}(nz)$ time.

The algorithm of [10] uses a data structure (which we denote here by $\mathcal{D}$) to store a multiset of elements A from the set $\{2, \dots, n\}$ allowing three operations:

1. initialisation with a given multiset of elements A;
2. computing $\mathit{maxgap}(\mathcal{D}) = \mathit{maxgap}(A \cup \{1, n+1\})$ for the currently stored multiset A;
3. removing a specified element from the currently stored multiset A.

The data structure has $\mathcal{O}(n)$ size, executes operation 1. in $\mathcal{O}(|A| + n)$ time and supports operations 2. and 3. in constant time. It consists of: (1) an array $C[2\,..\,n]$ that counts the multiplicity of each element in A; (2) a list L that stores all distinct elements of $A \cup \{1, n+1\}$ in ascending order and retains its maxgap; and (3) an array $P[1\,..\,n+1]$ that stores, for each distinct element of $A \cup \{1, n+1\}$, a pointer to its occurrence in L.

The algorithm of [10], formulated in terms of our index $\mathcal{I}$, works as follows. For a node v let $\mathcal{D}(v)$ be the $\mathcal{D}$-data structure storing the multiset $OL(v) \setminus \{1\}$. The path from the root to each terminal node that represents a *maximal solid prefix* of X is traversed, and at each explicit node v the data structure $\mathcal{D}(v)$ is computed. To this end, when we move from an explicit node v to its explicit child w on the path, from $\mathcal{D}(v)$ we remove all elements from $OL(w')$ for w' being explicit children of v other than w. Afterwards for the node w we perform the following check, which we call *cover-check*(w): if $\mathit{maxgap}(\mathcal{D}(w)) \le \mathit{depth}(w)$, report the covers being prefixes of $\mathsf{L}(w)$ of length $[\max(\mathit{maxgap}(\mathcal{D}(w)), \mathit{depth}(v)+1)..\mathit{depth}(w)]$. The whole procedure works in $\mathcal{O}(nz^2)$ time, as a single traversal works in linear time w.r.t. the size of the index and there are at most z maximal solid prefixes of X.

Let us show how this algorithm can be implemented to run in $\mathcal{O}(nz)$ time. We will call an explicit node of $\mathcal{I}$ a *prefix node* if it corresponds to a solid prefix of X. To implement the solution, it suffices for each prefix node to compute the $\mathcal{D}$-data structure and apply the *cover-check* routine. A prefix node will be called *branching* if it has more than one child being a prefix node, and *starting* if it is the root or its parent is branching. A maximal path going down the tree from a starting prefix node and passing only through non-starting prefix nodes will be called a *covering* path. Such a path ends in the first branching node or in a node corresponding to a maximal solid prefix. Considering the prefix node subtree of $\mathcal{I}$, which contains at most z leaves and, consequently, at most $z-1$ branching nodes, we make the following easy but important observation.

Observation 7.3. *There are $\mathcal{O}(z)$ starting prefix nodes. There are $\mathcal{O}(z)$ covering paths and each prefix node belongs to exactly one of them.*

In the algorithm we compute the $\mathcal{D}$-data structures for all starting prefix nodes (by first computing the C-arrays) and then update the data structure efficiently along each covering path.

Lemma 7.4. *$\mathcal{D}(v)$ for all starting prefix nodes v can be computed in $\mathcal{O}(nz)$ time.*

Proof. We compute the C-arrays of the data structures for all the prefix nodes v from the bottom to the top, storing them for starting prefix nodes only. In the end we shall construct the L-lists and P-arrays for the starting prefix nodes, which takes $\mathcal{O}(nz)$ time as there are at most z starting prefix nodes (Observation 7.3).

For a terminal node v we initialise its C-array with $OL(v)$ in $\mathcal{O}(|OL(v)| + n)$ time.

For a non-branching prefix node v with prefix node child w, we take the C-array of the latter and insert all elements of $OL(w')$ for the remaining explicit children w' of v. The time complexity is proportional to the sum of lengths of those lists $OL(w')$. Note that we do not need to store the C-array of w any more.

Finally, for a branching prefix node v we add up the counters of the C-arrays of all its prefix node children and insert all elements of $OL(w')$ for the remaining explicit children w' of v. This time the complexity is proportional to the sum of lengths of the lists $OL(w')$ and n times the number of prefix node children.

Let us analyse the time complexity of the algorithm. Note that each element of the global list OL will be added to some C-array at most once. Also when two or more C-arrays are added component-wise at a branching prefix node, all of them correspond to starting prefix nodes and each starting prefix node participates in such operation at most once. Moreover, note that we maintain exactly one array for each covering path. This way, the total time and space complexity is $\mathcal{O}(nz)$. □

Lemma 7.5. *The values $\mathit{maxgap}(\mathcal{D}(v))$ for all prefix nodes v can be computed in $\mathcal{O}(nz)$ time.*

Proof. We first perform the computations of Lemma 7.4. Let us fix a starting prefix node. We compute its maxgap and go along the covering path that it belongs to, maintaining the $\mathcal{D}$-data structure. Every node on the path (except for the last one) has exactly one prefix node child. Hence, when going from a prefix node v to its child, prefix node w, we simply remove from $\mathcal{D}(v)$ the elements of $OL(w')$ for all the remaining children w' of v. This allows us to compute $\mathit{maxgap}(\mathcal{D}(w))$.

In total each element of the global occurrence list OL will be removed from a $\mathcal{D}$-data structure at most once, which yields time complexity $\mathcal{O}(nz)$. □

Theorem 7.6. *A representation of size $\mathcal{O}(nz)$ of all covers of a weighted sequence X of length n can be computed in $\mathcal{O}(nz)$ time. In particular, all shortest covers of X can be determined in $\mathcal{O}(nz)$ time.*

Proof. To annotate all the covers on the edges of the index, we compute the maxgaps for all the prefix nodes using Lemma 7.5 and then apply the constant-time *cover-check* routine for each of the nodes. As for the shortest covers, there are at most z of them (as there are at most z different solid prefixes of X of a specified length, each with probability of occurrence at least $\frac{1}{z}$), so they can all be listed explicitly in $\mathcal{O}(nz)$ time and space. □

8. Approximate Weighted Index

Let us proceed to the solution of the Approximate Weighted Indexing problem. We are to answer queries for a pattern P and a probability threshold $\frac{1}{z'}$ and are allowed to report occurrences with probability $\geq \frac{1}{z'}-\epsilon$, for a given value of $\epsilon > 0$. Let us recall that for constant-sized alphabets [15] solves this problem in $\mathcal{O}(\frac{1}{\epsilon}nz^2)$ space (with $\Omega(\frac{1}{\epsilon}n^2z^2)$ construction time) with $\mathcal{O}(m + |Occ_{\frac{1}{z'}-\epsilon}(P, X)|)$-time queries, assuming that $z' \leq z$ holds in all queries. Our techniques lead to a substantial improvement over the complexities of this index.

Assume that the query is for a pattern P and a threshold $\frac{1}{z'}$. If $\frac{1}{z'} \leq \epsilon$, then the query is trivial as all the positions in X can be reported. Henceforth, we assume that $\frac{1}{z'} > \epsilon$.

Let us consider a z-estimation $\mathcal{S}$ for the weighted sequence with $z = \frac{1}{\epsilon}$. Let $\ell = \lfloor \frac{z}{z'} \rfloor$. By Definition 1, we can return position i as an occurrence of P based on whether $\mathit{Count}_{\mathcal{S}}(P, i) \geq \ell$; this is shown in the following lemma.

Lemma 8.1. *If $\mathit{Count}_{\mathcal{S}}(P, i) \geq \ell$, then $\mathit{Prob}_X(P, i) \geq \frac{1}{z'} - \epsilon$. If $\mathit{Count}_{\mathcal{S}}(P, i) < \ell$, then $\mathit{Prob}_X(P, i) < \frac{1}{z'}$.*

Proof. Assume that $\mathit{Count}_{\mathcal{S}}(P, i) \geq \ell$. Then

$$\mathit{Prob}_X(P, i) \geq \tfrac{1}{z}\mathit{Count}_{\mathcal{S}}(P, i) \geq \tfrac{1}{z}\left\lfloor \tfrac{z}{z'} \right\rfloor \geq \tfrac{1}{z}(\tfrac{z}{z'} - 1) = \tfrac{1}{z'} - \epsilon.$$

Now assume that $\mathit{Count}_{\mathcal{S}}(P, i) < \ell$. As $\mathit{Count}_{\mathcal{S}}(P, i) = \lfloor \mathit{Prob}_X(P, i)z \rfloor$, this concludes that $\mathit{Prob}_X(P, i)z < \ell$, which is equivalent to $\mathit{Prob}_X(P, i) < \frac{\ell}{z} = \frac{1}{z}\left\lfloor \frac{z}{z'} \right\rfloor \leq \frac{1}{z'}$. □

Thus our approximate weighted index for X is the weighted index for X constructed for $z = \frac{1}{\epsilon}$. To obtain the desired accuracy, it suffices to find the node v in the property suffix tree that corresponds to P and report all positions i in X such that there are at least $\lfloor \frac{z}{z'} \rfloor$ leaves in the subtree of v labelled with the position i. Let us show that this can be done by augmenting the weighted index by a data structure for *(top-k) document retrieval*.

A version of the document retrieval problem (see [31, Section 4.4]) can be stated operationally as follows. We are given a compact trie T with N leaves, each leaf labelled with a document number being a positive integer up to N. (Usually, T is a suffix tree of a collection of documents.) Given a pattern P, let v be the locus of P. Our goal is to report subsequent documents whose numbers occur most frequently in the leaves of the subtree of v until the process of reporting is interrupted. In [31] a data structure of size $\mathcal{O}(N)$ is shown that, given the node v, reports k top-scoring documents in $\mathcal{O}(k)$ time. The construction time of the data structure is $\mathcal{O}(N \log N)$.

We can augment our property suffix tree with this data structure with the document numbers being the labels of terminals (we can create a separate leaf for each label). This gives $N = \mathcal{O}(nz) = \mathcal{O}(\frac{n}{\epsilon})$. To find the documents with at least ℓ occurrences, we compute by doubling the smallest k such that the last of the top

k documents reported has less than ℓ occurrences. The number of documents reported in the last step of the doubling search will be at most $2|Occ_{\frac{1}{z'}-\epsilon}(P,X)|$ and the total number will not exceed $4|Occ_{\frac{1}{z'}-\epsilon}(P,X)|$.

Theorem 8.2. *For a weighted sequence of length n over an integer alphabet and parameter $\epsilon > 0$, the Approximate Weighted Indexing problem can be solved in $\mathcal{O}(\frac{n}{\epsilon})$ space with $\mathcal{O}(m + |Occ_{\frac{1}{z'}-\epsilon}(P,X)|)$-time queries. The construction time is $\mathcal{O}(\frac{n}{\epsilon}\log\frac{n}{\epsilon})$.*

9. Randomised Construction with Greater Space Usage

A symbol $X[i]$ of a weighted sequence X can be interpreted as a probability distribution on Σ, and the whole sequence X can be interpreted as a product distribution on strings of length n over Σ. In this setting, if $S \sim X$, i.e., S is a random string with distribution X, then, for any position i and string P, we have $\mathbb{P}[S[i\mathinner{..}i+|P|-1] = P] = Prob_X(P,i)$. This interpretation can be used to provide a randomised construction of families $\mathcal{S}$ of strings with properties equivalent to the weighted sequence X in a certain sense, weaker than the one used in Definition 1.

Lemma 9.1. *There is a randomised algorithm which, given a weighted sequence X of length n and a threshold parameter z, in $\mathcal{O}(nz\log(nz))$ time constructs a family $\mathcal{S}$ of $k = \mathcal{O}(z\log(nz))$ strings S_j with properties π_j such that $Count_{\mathcal{S}}(P,i) > 0$ if and only if $Prob_X(P,i) \ge \frac{1}{z}$. It succeeds with high probability ($1 - \frac{1}{(nz)^c}$ for arbitrarily large constant c).*

Proof. We randomly sample $k = \lceil (c+2)z\ln(nz)\rceil$ strings $S_1,\dots,S_k$. Formally, these are independent random variables with distribution X. The properties π_j are specified so that $S_j[i\mathinner{..}\pi_j[i]]$ is the longest prefix of $S_j[i\mathinner{..}n]$ with $Prob_X(S_j[i\mathinner{..}\pi_j[i]], i) \ge \frac{1}{z}$.

This way, $Count_{\mathcal{S}}(P,i) > 0$ implies $Prob_X(P,i) \ge \frac{1}{z}$. On the other hand, if $Prob_X(P,i) \ge \frac{1}{z}$, then, since $\mathbb{P}[S_j[i\mathinner{..}i+|P|-1] \ne P] = 1 - Prob_X(P,i)$, we have:

$$\mathbb{P}[Count_{\mathcal{S}}(P,i) = 0] = (1 - Prob_X(P,i))^k \le \exp(-kProb_X(P,i)) \le \exp(-(c+2)\ln(nz)) = \tfrac{1}{(nz)^{c+2}}.$$

There are at most n^2z pairs (P,i) satisfying $Prob_X(P,i) \ge \frac{1}{z}$ (this is the bound for the sum of lengths of all strings in the sets $\mathcal{M}_i$ from Section 3). Consequently, the resulting family has the required property with probability at least $1 - \frac{n^2z}{(nz)^{c+2}} \ge 1 - \frac{1}{(nz)^c}$. □

We can directly use the same methods as in Section 6 to construct a weighted index from the family of strings constructed in Lemma 9.1. The space complexity of the resulting index is worse than the one in Theorem 6.3 by a factor of $\log(nz)$ and the construction is (Monte-Carlo) randomised.

Corollary 9.2. *There is a data structure of size $\mathcal{O}(nz\log(nz))$ for the Weighted Indexing problem which answers queries in optimal time. It can be constructed using a randomised $\mathcal{O}(nz\log(nz))$-time algorithm which returns a valid weighted index with high probability.*

The same type of construction can be applied to obtain an approximate weighted index. To this end, we need a stronger equivalence property of a string family and a greater number of sampled strings to satisfy this property.

Lemma 9.3. *There is a randomised algorithm which, given a weighted sequence X of length n and a parameter $\epsilon > 0$, in $\mathcal{O}(\frac{n}{\epsilon^2}\log(\frac{n}{\epsilon}))$ time constructs a family $\mathcal{S}$ of $k = \mathcal{O}(\frac{1}{\epsilon^2}\log(\frac{n}{\epsilon}))$ strings S_j with properties π_j such that $|Prob_X(P,i) - \frac{1}{k}Count_{\mathcal{S}}(P,i)| < \epsilon$ for every position i and string P. It succeeds with high probability ($1 - (\frac{\epsilon}{n})^c$ for arbitrarily large constant c).*

Proof. We randomly sample $k = \lceil (c+2)\frac{1}{\epsilon^2}\ln\frac{n}{\epsilon}\rceil$ strings $S_1,\dots,S_k$. The properties π_j are so that $S_j[i\mathinner{..}\pi_j[i]]$ is the longest prefix of $S_j[i\mathinner{..}n]$ such that $Prob_X(S_j[i\mathinner{..}\pi_j[i]], i) \ge \epsilon$.

Observe that if $Prob_X(P,i) < \epsilon$, then $Count_{\mathcal{S}}(P,i) = 0$. On the other hand, if $Prob_X(P,i) \geq \epsilon$, then $Count_{\mathcal{S}}(P,i) \sim \mathrm{Bin}(k, Prob_X(P,i))$. Consequently, Hoeffding's inequality [32] implies

$$\mathbb{P}[|Prob_X(P,i) - \tfrac{1}{k}Count_{\mathcal{S}}(P,i)| > \epsilon] \leq 2e^{-\epsilon^2 k} = 2e^{-(c+2)\ln\frac{n}{\epsilon}} \leq 2(\tfrac{\epsilon}{n})^{c+2}.$$

There are at most $\frac{n^2}{\epsilon}$ such pairs (P,i), so the family $\mathcal{S}$ satisfies the required condition with probability at least $1-(\frac{\epsilon}{n})^c$, as claimed. $\square$

We can use this family of strings to construct an approximate weighted index using top-k document retrieval just as in Section 8. We arrive at the following construction with space complexity greater than the one from Theorem 8.2 by a factor of $\frac{1}{\epsilon}\log\frac{n}{\epsilon}$ (and the construction is randomised).

Corollary 9.4. *There is a data structure of size $\mathcal{O}(\frac{n}{\epsilon^2}\log\frac{n}{\epsilon})$ which solves the Approximate Weighted Indexing problem with $\mathcal{O}(m+|Occ_{\frac{1}{z'}-\epsilon}(P,X)|)$ query time. It can be constructed using a randomised $\mathcal{O}(\frac{n}{\epsilon^2}\log^2\frac{n}{\epsilon})$-time algorithm which returns a valid approximate weighted index with high probability.*

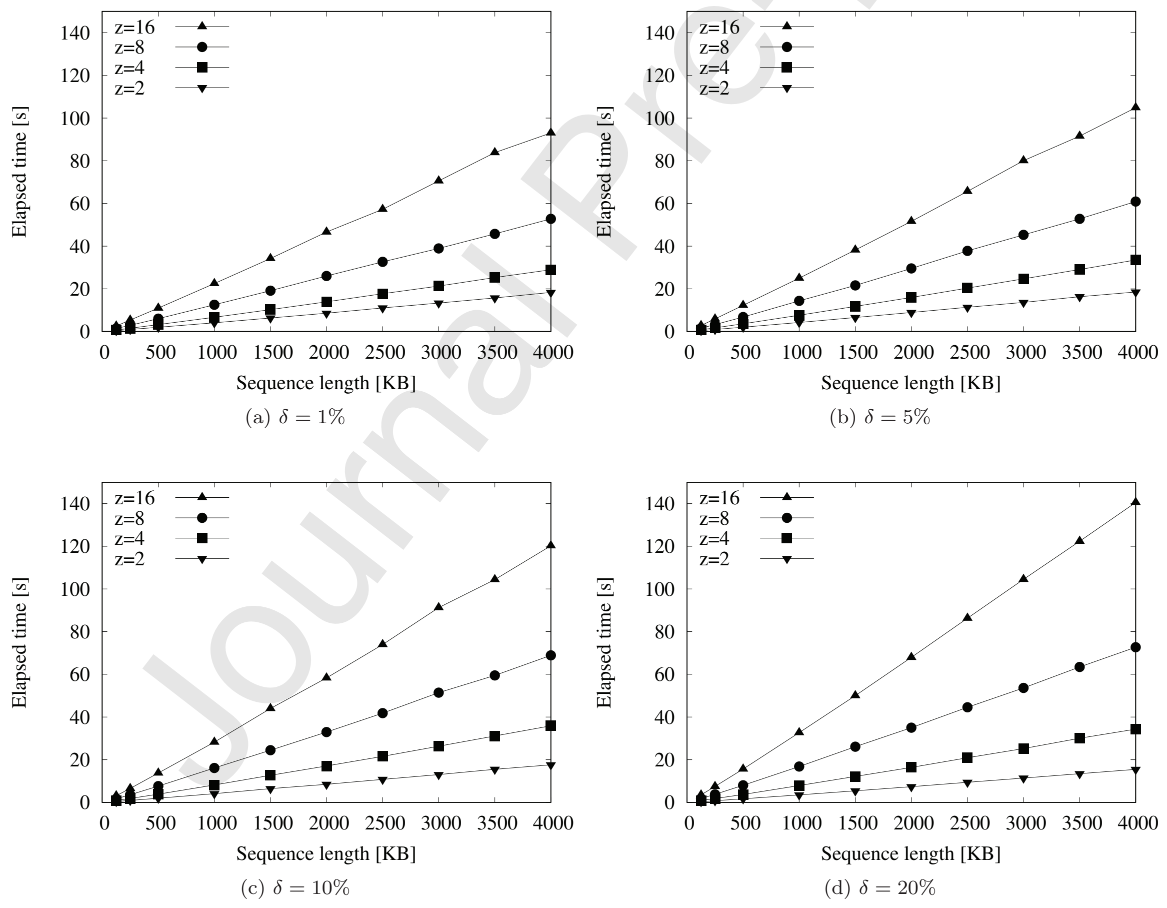
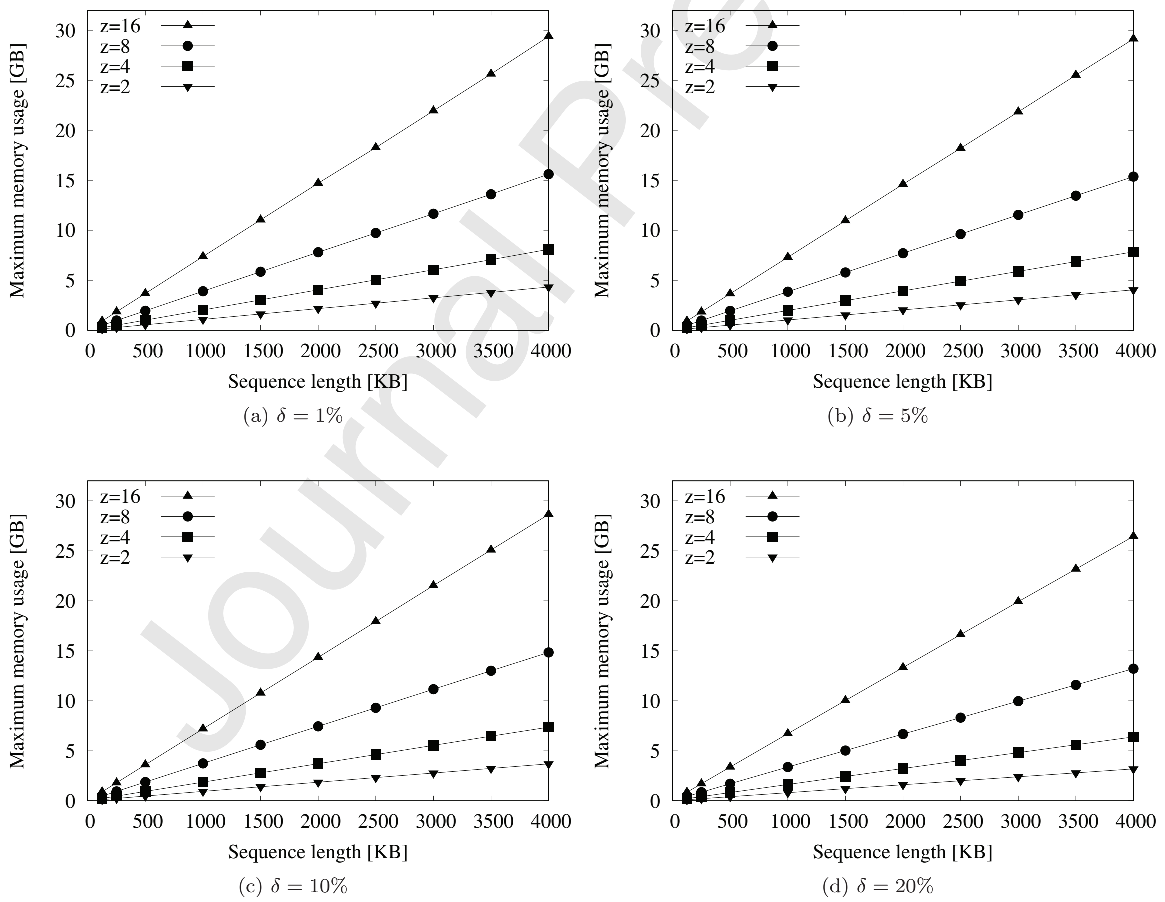


Figure 3: Elapsed time on synthetic weighted DNA sequences of length ranging from 125,000 to 4,000,000 with different degeneracy percentages.

10. Experimental Results

We have implemented the algorithm described in Section 6 for constructing the Weighted Index. The program has been implemented in the `C++` programming language and developed under the GNU/Linux operating system. The input parameters are a weighted sequence of length n over an alphabet of size σ, represented by an $n \times \sigma$ matrix, and a probability threshold $\frac{1}{z}$. The output of our program is the Weighted Index. Our implementation supports decision, counting, and reporting variants of queries; however, only decision operations were implemented in worst-case optimal time. We have validated our implementation for correctness against several known weighted pattern matching algorithms [6, 8, 9]. The source code is distributed at `https://bitbucket.org/kociumaka/weighted_index` under the GNU General Public License. All experiments have been conducted on a Desktop PC using one core of Intel Xeon CPU E5-2640 at 2.60GHz and 64GB of RAM. Our program has been compiled with `g++` version 6.2.0 at optimisation level 3 (-O3).

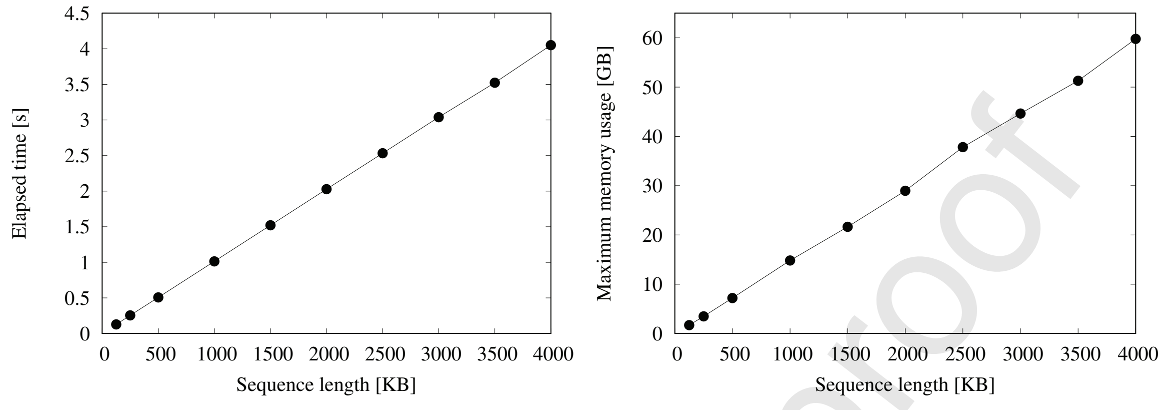
Synthetic weighted DNA sequences. To evaluate the time and space performance of our implementation, we first used *synthetic* weighted sequences over the DNA alphabet ($\sigma = 4$). (In most real-world applications, weighted sequences are over the DNA alphabet.) The weighted sequences were of length ranging from



(a) $\delta = 1\%$ (b) $\delta = 5\%$

(c) $\delta = 10\%$ (d) $\delta = 20\%$

Figure 4: Peak memory usage on synthetic weighted DNA sequences of length ranging from 125,000 to 4,000,000 with different degeneracy percentages.



(a) Elapsed time (b) Peak memory usage

Figure 5: Elapsed time and peak memory usage on real weighted DNA sequences of length ranging from 125,000 to 4,000,000 obtained from human chromosome 21.

125,000 to 4,000,000. For each length, four different degeneracy percentages, denoted by δ, were used: 1%, 5%, 10% and 20%: percentage of positions where at least two letters with positive probability exist. The probability threshold was set to $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ and $\frac{1}{16}$. The results are plotted in Figure 3, for elapsed time, and in Figure 4, for peak memory usage. It is evident that, for fixed z, the elapsed time and peak memory usage grow linearly in n. It should not be difficult to see that, for fixed n, the elapsed time and peak memory usage grow linearly in z. These results thus confirm fully our theoretical findings (see Theorem 6.3).

Real weighted DNA sequences. To further evaluate the time and space performance of our implementation, we created real weighted DNA sequences by combining the Genome Reference Consortium Human Build 37 (GRCh37) with the variants obtained from the 1000 Genomes Project (October 2011 Integrated Variant Set release) [33]. Specifically we made use of human chromosome 21 data. We randomly extracted fragments of length ranging from 125,000 to 4,000,000 from the generated weighted sequence. The probability threshold was set to $\frac{1}{8}$. The results are plotted in Figure 5. In both plots (elapsed time and peak memory usage) we observe that the performance is analogous to the performance with the synthetic data; in particular to the dataset with $\delta = 1\%$. This is because δ is found to be 0.7% in the weighted sequence of chromosome 21.

11. Final Remarks

In this article we presented an efficient index for Weighted Pattern Matching along with new combinatorial insights into the nature of weighted sequences.

Our ideas can also be used to improve the solution for the Generalised Weighted Indexing problem from [15]. The authors use a notion of *special weighted sequences* in which each position contains at most one letter with a positive probability. (In this case, the assumption that the probabilities sum up to 1 at each position is waived.) In [15] the input weighted sequence is transformed using the reduction of [3] into a special weighted sequence of length $\mathcal{O}(nz^2 \log z)$ that preserves the set of maximal solid factors. In the special weighted sequence, a query for a pattern P under the probability threshold $\frac{1}{z'}$ is answered in $\mathcal{O}(m + m \cdot |Occ_{\frac{1}{z'}}(P, X)|)$ time.

Our z-estimation $\mathcal{S}$ can be transformed into a special weighted sequence of length $\mathcal{O}(nz)$ that also preserves the set of solid factors. We simply concatenate the strings, taking the letter probabilities from the respective positions in X, and split the concatenated parts with a zero-probability position. This gives a more space-efficient reduction that can be used in the data structure of [15].

Corollary 11.1. *For a weighted sequence of length n over an integer alphabet, the Generalised Weighted Indexing problem can be solved with $\mathcal{O}(m + m \cdot |Occ_{\frac{1}{z'}}(P, X)|)$-time queries with an index of size $\mathcal{O}(nz)$.*

Acknowledgements

We thank an anonymous referee of a previous version of this article for the idea of a simple randomised construction. We also thank Tsvi Kopelowitz for bringing our attention to the multitude of existing solutions to the Property Indexing problem.

[1] C. Barton, T. Kociumaka, S. P. Pissis, J. Radoszewski, Efficient index for weighted sequences, in: R. Grossi, M. Lewenstein (Eds.), 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, Vol. 54 of LIPIcs, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 4:1–4:13. doi:10.4230/LIPIcs.CPM.2016.4.

[2] C. C. Aggarwal, P. S. Yu, A survey of uncertain data algorithms and applications, IEEE Trans. Knowl. Data Eng. 21 (5) (2009) 609–623. doi:10.1109/TKDE.2008.190.

[3] A. Amir, E. Chencinski, C. S. Iliopoulos, T. Kopelowitz, H. Zhang, Property matching and weighted matching, Theor. Comput. Sci. 395 (2–3) (2008) 298–310. doi:10.1016/j.tcs.2008.01.006.

[4] M. Christodoulakis, C. S. Iliopoulos, L. Mouchard, K. Tsichlas, Pattern matching on weighted sequences, in: Algorithms and Computational Methods for Biochemical and Evolutionary Networks, CompBioNets 2004, KCL publications, 2004.

[5] S. Rajasekaran, X. Jin, J. L. Spouge, The efficient computation of position-specific match scores with the fast Fourier transform, J. Comput. Biol. 9 (1) (2002) 23–33. doi:10.1089/10665270252833172.

[6] T. Kociumaka, S. P. Pissis, J. Radoszewski, Pattern matching and consensus problems on weighted sequences and profiles, Theory Comput. Syst. 63 (3) (2019) 506–542. doi:10.1007/s00224-018-9881-2.

[7] P. Charalampopoulos, C. S. Iliopoulos, S. P. Pissis, J. Radoszewski, On-line weighted pattern matching, Inf. Comput. 266 (2019) 49–59. doi:10.1016/j.ic.2019.01.001.

[8] C. Barton, C. Liu, S. P. Pissis, On-line pattern matching on uncertain sequences and applications, in: T. H. Chan, M. Li, L. Wang (Eds.), 10th International Conference on Combinatorial Optimization and Applications, COCOA 2016, Vol. 10043 of LNCS, Springer, 2016, pp. 547–562. doi:10.1007/978-3-319-48749-6_40.

[9] C. Barton, C. Liu, S. P. Pissis, Fast average-case pattern matching on weighted sequences, Int. J. Found. Comput. Sci. 29 (8) (2018) 1331–1343. doi:10.1142/S0129054118430062.

[10] C. S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, A. K. Tsakalidis, The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications, Fundam. Inform. 71 (2–3) (2006) 259–277.
URL http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07

[11] C. S. Iliopoulos, M. S. Rahman, Faster index for property matching, Inf. Process. Lett. 105 (6) (2008) 218–223. doi:10.1016/j.ipl.2007.09.004.

[12] M.-T. Juan, J.-J. Liu, Y.-L. Wang, Errata for "Faster index for property matching", Inf. Process. Lett. 109 (18) (2009) 1027–1029. doi:10.1016/j.ipl.2009.06.009.

[13] P. Charalampopoulos, C. S. Iliopoulos, C. Liu, S. P. Pissis, Property suffix array with applications, in: M. A. Bender, M. Farach-Colton, M. A. Mosteiro (Eds.), 13th Latin American Symposium on Theoretical Informatics, LATIN 2018, Vol. 10807 of LNCS, Springer, 2018, pp. 290–302. doi:10.1007/978-3-319-77404-6_22.

[14] T. Kopelowitz, The property suffix tree with dynamic properties, Theor. Comput. Sci. 638 (2016) 44–51. doi:10.1016/j.tcs.2016.02.033.

[15] S. Biswas, M. Patil, S. V. Thankachan, R. Shah, Probabilistic threshold indexing for uncertain strings, in: E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, K. Stefanidis (Eds.), 19th International Conference on Extending Database Technology, EDBT 2016, OpenProceedings.org, 2016, pp. 401–412. doi:10.5441/002/edbt.2016.37.

[16] C. Barton, C. Liu, S. P. Pissis, Linear-time computation of prefix table for weighted strings & applications, Theor. Comput. Sci. 656 (2016) 160–172. doi:10.1016/j.tcs.2016.04.029.

[17] T. Shibuya, Constructing the suffix tree of a tree with a large alphabet, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A (5) (2003) 1061–1066.
URL https://search.ieice.org/bin/summary.php?id=e86-a_5_1061

[18] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260. doi:10.1007/BF01206331.

[19] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, J. ACM 47 (6) (2000) 987–1011. doi:10.1145/355541.355547.

[20] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on strings, Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.

[21] M. L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with $O(1)$ worst case access time, J. ACM 31 (3) (1984) 538–544. doi:10.1145/828.1884.

[22] M. Farach, S. Muthukrishnan, Perfect hashing for strings: Formalization and algorithms, in: D. S. Hirschberg, E. W. Myers (Eds.), 7th Annual Symposium on Combinatorial Pattern Matching, Vol. 1075 of LNCS, Springer, 1996, pp. 130–140. doi:10.1007/3-540-61258-0_11.

[23] P. Gawrychowski, M. Lewenstein, P. K. Nicholson, Weighted ancestors in suffix trees, in: A. S. Schulz, D. Wagner (Eds.), 22th Annual European Symposium on Algorithms, ESA 2014, Vol. 8737 of LNCS, Springer, 2014, pp. 455–466. doi:10.1007/978-3-662-44777-2_38.

[24] A. Amir, G. M. Landau, M. Lewenstein, D. Sokol, Dynamic text and static pattern matching, ACM Trans. Algorithms 3 (2) (2007) 19. doi:10.1145/1240233.1240242.

[25] D. E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, Inf. Process. Lett. 17 (2) (1983) 81–84. doi:10.1016/0020-0190(83)90075-3.

[26] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, A linear time algorithm for seeds computation (2019). arXiv:1107.2422v2.

[27] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: D. Eppstein (Ed.), 13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002, ACM/SIAM, 2002, pp. 657–666.
URL http://dl.acm.org/citation.cfm?id=545381.545469

[28] L. C. K. Hui, Color set size problem with application to string matching, in: A. Apostolico, M. Crochemore, Z. Galil, U. Manber (Eds.), 3rd Annual Symposium on Combinatorial Pattern Matching, CPM 1992, Vol. 644 of LNCS, Springer, 1992, pp. 230–243. doi:10.1007/3-540-56024-6_19.

[29] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, J. Algorithms 57 (2) (2005) 75–94. doi:10.1016/j.jalgor.2005.08.001.

[30] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355. doi:10.1137/0213024.

[31] G. Navarro, Y. Nekrich, Time-optimal top-k document retrieval, SIAM J. Comput. 46 (1) (2017) 80–113. doi:10.1137/140998949.

[32] W. Hoeffding, Probability inequalities for sums of bounded random variables, J. Am. Stat. Assoc. 58 (301) (1963) 13–30. doi:10.1080/01621459.1963.10500830.

[33] 1000 Genomes Project Consortium, A. Auton, L. D. Brooks, R. M. Durbin, E. P. Garrison, H. M. Kang, J. O. Korbel, J. L. Marchini, S. McCarthy, G. A. McVean, G. R. Abecasis, A global reference for human genetic variation, Nature 526 (7571) (2015) 68–74. doi:10.1038/nature15393.