Indexing Weighted Sequences: Neat and Efficient

Carl Barton\textsuperscript{1}, Tomasz Kociumaka\textsuperscript{2}, Chang Liu\textsuperscript{3}, Solon P. Pissis\textsuperscript{3}, and Jakub Radoszewski\textsuperscript{2,3}

\textsuperscript{1}European Bioinformatics Institute, Wellcome Genome Campus, Hinxton, UK, carl@ebi.ac.uk
\textsuperscript{2}Institute of Informatics, University of Warsaw, Warsaw, Poland, [kociumaka,jrad]@mimuw.edu.pl
\textsuperscript{3}Department of Informatics, King’s College London, London, UK, [chang.2.liu,solon.pissis]@kcl.ac.uk

Abstract

In a \textit{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 \textit{indexing} variant of the problem, in which we are to preprocess a weighted sequence to answer multiple pattern matching queries. We present an $O(nz)$-time construction of an $O(nz)$-sized index for a weighted sequence of length $n$ over a constant-sized alphabet that answers pattern matching queries in optimal, $O(m + \text{Occ})$ time, where $\text{Occ}$ is the number of occurrences reported. The cornerstone of our data structure is a novel construction of a family of $\lfloor z \rfloor$ special strings that carries the information about all the strings that occur in the weighted sequence with a sufficient probability. We obtain a weighted index with the same complexities as in the most efficient previously known index by Barton et al. \cite{Barton2017}, but our construction is significantly simpler. The most complex algorithmic tool required in the basic form of our index is the suffix tree which we use to develop a new, more straightforward index for the so-called property matching problem. We provide an implementation of our data structure. 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. \cite{Biswas2016} and an improvement of the space complexity of their general index.

1 Introduction

We consider a type of uncertain sequence called a \textit{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.

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 \cite{Barton2017} 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 particular probabilistic databases have been a very active area of research in recent years; see e.g. \cite{Biswas2016}. A common assumption in practice is that the alphabet of weighted sequences is constant since the most commonly studied alphabet is $\Sigma = \{A, C, G, T\}$. 

1
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 $\Sigma$, 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 $i, \ldots, i+|P|-1$ in $X$ is at least $\frac{1}{z}$. Each such position is called an occurrence of the pattern; we also say that the fragment 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 $\epsilon > 0$. 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 $z' \leq z$.

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

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 $\text{Occ}$.

### 1.1 Previous Results

An $O(n \log m)$-time solution for the Weighted Pattern Matching problem based on the Fast Fourier Transform was proposed in [1] [19]. Recently, an $O(n \log z)$-time solution using the suffix array and lookahead scoring was presented in [15]. The average case complexity of the WPM problem has also been studied and a number of fast algorithms have been shown with both linear [4] and sub-linear on average algorithms being presented [5].

The Weighted Indexing problem was first considered by Iliopoulos et al. [12], who introduced a data structure called weighted suffix tree allowing optimal $O(m + \text{Occ})$-time queries. The construction time and size of that data structure was, however, $O(n|\Sigma|z \log z)$.

Amir et al. [2] reduced the Weighted Indexing problem to the Property Indexing problem in a text of length $O(nz^2 \log z)$. For the latter, they proposed a solution with $O(n \log n \log z)$ preprocessing time and optimal $O(m + \text{Occ})$ query time. Later it was shown that the Property Indexing problem can be solved in linear time; see [13] [14] (see also [16]). This lead to a solution to the Weighted Indexing problem with index size and construction time $O(nz^2 \log z)$, preserving optimal query time.

These results were recently improved by some of the authors in [4], where they proposed an $O(nz)$-sized data structure for the Weighted Indexing problem that can be constructed also in $O(nz)$ time. The query time is still $O(m + \text{Occ})$. The authors proposed several applications of their index.

Biswas et al. [6] presented a data structure that solves the Approximate Weighted Indexing problem in $O(nz^2 \log z)$ space (with $\Omega(nz^2 \log z)$ construction time) with $O(m + \text{Occ})$-time queries; here $\text{Occ}$ denotes the number of occurrences reported. They also proposed a data structure for the Generalised Weighted Indexing problem with $O(nz^2 \log z)$ space and $O(m + m \cdot \text{Occ})$ query time. The construction time is not mentioned, but a direct construction of their index works in $\Omega(n^2z^2)$ time. Moreover, they also consider the problem of document listing for weighted sequences.

### 1.2 Our Contribution

We present a new $O(nz)$-time construction of an $O(nz)$-sized data structure for the Weighted Indexing problem that answers queries in optimal $O(m + \text{Occ})$ time. Our index is based on a novel observation that one can always construct a family of $\lceil z \rceil$ special 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 [3] preserving all of its applications. As a by product, we obtain an optimal solution to
the Property Indexing problem that avoids complex tools used in the previous solutions [2, 13, 14, 16]. We
provide a proof-of-concept implementation of our index that was validated for correctness and efficiency. We
also discuss an even simpler randomised construction with worse space complexity and construction time of
the index.

Our approach lets us significantly improve upon the variants of the weighted index proposed in [6]. In
the Approximate Weighted Indexing problem, we obtain $O\left(\frac{n}{\epsilon^2} z \log \frac{n}{\epsilon^2}ight)$ space and $O\left(\frac{n}{\epsilon^2} z \log \frac{n}{\epsilon^2}\right)$ construction time,
preserving the query time. We also improve the space usage in the Generalised Weighted Indexing problem
to $O(nz)$, also in the document listing variant.

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 $\lfloor z \rfloor$ special 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 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 property is used in the construction of
a weighted index and the latter in the construction of an approximate weighted index. The existence of this
family is not immediate. However, its proof not involved and we design a $O(nz)$-time elementary construction
algorithm 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 sequence 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 previous, exponential-size index of Iliopoulos et al. [12],
previous work includes the $O(n z^2 \log z)$-space index of Amir et al. [2] and $O(n z)$-space index by Barton et
al. [3]. Amir et al. [2] show that, after a small modification of the weighted sequence, the set of maximal
string patterns that occur in it has a total length $O(n z^2 \log z)$. Barton et al. [3] show a representation of
this set as a trie and apply Shibuya’s algorithm for suffix tree of a trie construction [20].

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 [21]. Very similar data structures were constructed
by Amir et al. [2] and Kopelowitz [16]. Amir et al. [2] use a heavy machinery of weighted ancestor queries and
a fancy algorithm to mark the properties on edges of the suffix tree. Kopelowitz [16] 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. [13] and clarified by Juan et al. [14] 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.

Our techniques enable us immediately to 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 by Barton et al. [3]. 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 were used in the previous solution by Biswas et
al. [6], however, not as a black box. Moreover, they also use the less efficient reduction of [2] which caused
their data structure to use $O\left(\frac{1}{\epsilon} n z^2\right)$ space, assuming that $z' \leq z$ in each query. Finally, we improve the
space complexity of the generalised weighted index of Biswas et al. [6] by plugging in our construction of $\lfloor z \rfloor$
special strings.
1.4 Structure of the Paper

In Section 3 we present a combinatorial construction of the special family of $|z|$ strings. An efficient implementation of the construction of this family based on tries is proposed in Section 4. In Section 5 the 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 and, with the aid of an auxiliary tool, an approximate weighted index in Section 7. Alternative randomised constructions of the two indexes with worse parameters are discussed in Section 8. Our improvement to the Generalised Weighted Index and our C++ implementation are briefly discussed Section 9.

2 Preliminaries

2.1 Strings and Property Indexing

A string $S$ over an alphabet $\Sigma$ is a finite sequence of letters from $\Sigma$. By $n = |S|$ we denote the length of $S$ and by $S[i]$, for $1 \leq i \leq n$, we denote the $i$-th letter of $S$. By $S[i..j]$ we denote the string $S[i]...S[j]$ called a factor of $S$ (if $i > j$, then the factor is an empty string). 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 $\Pi$ of $S$ is a hereditary collection of integer intervals contained in $\{1, \ldots, n\}$. For simplicity, we represent every property $\Pi$ with an array $\pi[1..|S|]$ such that the longest interval $I \in \Pi$ starting at position $i$ is $\{i, \ldots, \pi[i]\}$. Observe that $\pi$ can be an arbitrary array satisfying $\pi[i] \in \{i - 1, \ldots, n\}$ and $\pi[1] \leq \pi[2] \leq \cdots \leq \pi[n]$. For a string $P$, by $Occ_{\pi}(P, S)$ we denote the set of occurrences $i$ of $P$ in $S$ such that $i + |P| - 1 \leq \pi[i]$. These notions lead us to the statement of the following problem.

**Problem 1** (Property Indexing).

**Input:** A string $S$ of length $n$ over an alphabet $\Sigma$ and an array $\pi$ representing a property $\Pi$.

**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 $\pi_j$. 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 the position $i$ in the strings $S_1, \ldots, S_k$ that respect the properties.

2.2 Weighted Sequences and Weighted Indexing

A weighted sequence $X = x_1x_2\ldots x_n$ of length $|X| = n$ over an alphabet $\Sigma$ 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 $i \in \{1, \ldots, n\}$. 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.

|   | 1  | 2  | 3  | 4  | 5  | 6  |
|---|----|----|----|----|----|----|
| $p_1^{(X)}(A)$ | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ |
| $p_1^{(X)}(B)$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ |

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

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

$$Prob_X(P, i) = \prod_{j=1}^{i} p_{i+j-1}^{(X)}(P[j]).$$
We say that a string $P$ occurs in $X$ at position $i$ if $\text{Prob}_X(P,i) \geq \frac{1}{2}$. We also say that $P$ is a solid factor of $X$ (starting, occurring) at position $i$. By $\text{Occ}_\frac{1}{2}(P,X)$ we denote the set of all positions where $P$ occurs in $X$. The main problem in scope can be formulated as follows.

**Problem 2 (Weighted Indexing).**

**Input:** A weighted sequence $X$ of length $n$ over an alphabet $\Sigma$ and a threshold $\frac{1}{2}$.

**Queries:** For a given pattern string $P$ of length $m$, check if $\text{Occ}_\frac{1}{2}(P,X) \neq \emptyset$ (decision query), compute $|\text{Occ}_\frac{1}{2}(P,X)|$ (counting query), or report all elements of $\text{Occ}_\frac{1}{2}(P,X)$ (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 $O(1)$ time. We further assume that $|\Sigma| = O(1)$; under this assumption a weighted sequence of length $n$ has a representation using $O(n)$ space.

### 3 Existence of an Equivalent Family of Strings

In the definition below, we formalise the property of a string family that we aim to construct.

**Definition 1.** We say that an indexed family $S = (S_j, \pi_j)_{j=1}^{|z|}$ 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 $i \in \{1, \ldots, n\}$, $\text{Count}_S(P,i) = \lfloor \text{Prob}_X(P,i)z \rfloor$.

Note that a $z$-estimation $S$ 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 $\pi_j$. This observation will be used in the construction of our weighted index. Moreover, the value $\text{Count}_S(P,i)$ provides a good estimation of the probability $\text{Prob}_X(P,i)$:

$$\frac{1}{2} \text{Count}_S(P,i) \leq \text{Prob}_X(P,i) < \frac{1}{2} \text{Count}_S(P,i) + \frac{1}{2}.$$

This will let us design an approximate weighted index. An example of a $z$-estimation is shown in Table 2.

| $i$ | 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 $P$ | $\text{Prob}_X(P,3)$ | $\{j : 3 \in \text{Occ}_{\pi_j}(P,S_j)\}$ |
|------------|----------------------|---------------------------------|
| $\varepsilon$ | 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 $S$ 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 $S$.

Below, we prove 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:

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

for $i = 1, \ldots, n$. We start with an equivalent characterisation of $z$-estimations of $X$. 

5
Observation 1. A family \( S = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor} \) 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 \( S_j[i \ldots \pi_j[i]] \).

Next, we prove that this condition uniquely defines the multiset \( \{S_j[i \ldots \pi_j[i]] : 1 \leq j \leq \lfloor z \rfloor \} \).

Lemma 1. 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 character \( c \). In the remaining \( t_i(P) - \sum_{c \in \Sigma} t_i(Pc) \) strings in \( \mathcal{M}_i \), the prefix \( P \) is not followed by any letter. Thus, the multiplicity of \( P \) in \( \mathcal{M}_i \) must be \( m_i(P) \). This implies uniqueness of \( \mathcal{M}_i \).

Observe that \( t_i(P) \geq \sum_{c \in \Sigma} t_i(Pc) \), because \( \text{Prob}_X(P,i) \geq \sum_{c \in \Sigma} \text{Prob}_X(Pc,i) \) and the function \( x \mapsto \lfloor xz \rfloor \) is superadditive. 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 \( \text{Prob}_X(P,i) = 0 \). 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.

Observe that in a \( z \)-estimation, \( S_j[i \ldots \pi_j[i]] \) can be obtained from \( S_j[i+1 \ldots \pi_j[i+1]] \) by inserting a leading character and dropping some number of trailing characters. This statement holds if only \( \pi_j[i] \geq i \); otherwise \( S_j[i \ldots \pi_j[i]] = \varepsilon \). 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 character \( c \in \Sigma \) 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.

| \( \mathcal{M}_1 \) | \( \mathcal{M}_2 \) | \( \mathcal{M}_3 \) | \( \mathcal{M}_4 \) | \( \mathcal{M}_5 \) | \( \mathcal{M}_6 \) |
|-----------|-----------|-----------|-----------|-----------|-----------|
| AAA       | A         | A         | A         | A         | A         |
| AAAAA     | AAA       | AAA       | AAB       | AB        | B         |
| ABAA      | BAA       | AAB       | ABB       | BB        | B         |
| AP        | B         | B         | \varepsilon | 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 2. For every \( 1 \leq i \leq n - 1 \), there is 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 \( R \) the multiset containing all strings \( Q \in \mathcal{M}_{i+1} \) compatible with \( P \), i.e., starting with \( Q' \). We further define \( L \) as the multiset containing all strings \( P' \in \mathcal{M}_i \) that start with \( cQ' \) for some \( c \in \Sigma \). The construction procedure guarantees that each \( Q \in R \) has been matched to a compatible \( P' \) satisfying \( |P'| \geq |P| \); such \( P' \) must belong to the multiset \( L \).

Observe that \( |L| = \sum_{c' \in \Sigma} t_i(c'Q') \leq t_{i+1}(Q') = |R| \) because \( \text{Prob}_X(Q',i+1) \geq \sum_{c' \in \Sigma} \text{Prob}_X(c'Q',i) \) and the function \( x \mapsto \lfloor xz \rfloor \) is superadditive. Consequently, each \( P' \in L \) must be matched to some \( Q \in R \). Since \( P \in L \) is unmatched, we obtain a contradiction. □
Due to Lemma 2 we can index the strings \( \mathcal{M}_i = \{P_{j,i} : 1 \leq j \leq |z|\} \) so that we have \(|z|\) 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 \( \pi_j \) so that \( S_j[i..\pi_j[i]] = P_{j,i} \). The value \( \pi_j[i] \) is not specified if \( P_{j,i} = \varepsilon \); in this case, we may set \( S_j[i] \) to an arbitrary character. The resulting family \( S = (S_j, \pi_j)_{j=1}^{|z|} \) clearly satisfies the characterisation of [??] 1, which completes the proof of the following result.

**Theorem 1.** Each weighted sequence \( X \) has a \( z \)-estimation.

### 4 Efficient Implementation

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

At a high level, we follow the existential construction of Section 3. We start with \( \mathcal{M}_{n+1} \), which consists of \(|z|\) copies of \( \varepsilon \), 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 \( S = (S_j, \pi_j)_{j=1}^{|z|} \). More precisely, we set \( \pi_j[i] \) to \( i + |P_{j,i}| - 1 \) and \( S_j[i] \) to the leading character of \( P_{j,i} \), or an arbitrary character if \( P_{j,i} = \varepsilon \).

Each transformation simulates the procedure provided in the proof of Lemma 2. However, our implementation uses solid factor tries in order to achieve \( O(z) \) amortised running time.

#### 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 \( L(u) \). The root has label \( \varepsilon \), and the parent of a node \( u \) with \( L(u) = Pc \) for \( c \in \Sigma \) is the node \( v \) with \( L(v) = P \); 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 \( L(u) = P_{j,i} \). For each token \( j \), we store the node \( u \in T_i \) with \( L(u) = P_{j,i} \) and the probability \( \text{Prob}_X(P_{j,i}, i) \). Observe that the number of tokens at the node \( u \) is \( m_i(L(u)) \) and the number of tokens in the subtree rooted at \( u \) is \( t_i(L(u)) \). To simplify notation, we denote \( m_i(u) = m_i(L(u)) \) and \( t_i(u) = t_i(L(u)) \). We have the following simple observation; see also Figure 1.

**Observation 2.** The trie \( T_i \) contains \(|z|\) tokens in total and every leaf contains tokens.

![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](image)
4.2 Transformation Algorithm

For each index $i$, we transform the solid factor trie $T_{i+1}$ to $T_i$ and move the tokens so that $M_{i+1}$ is transformed to $M_i$.

Before we describe the implementation, let us formulate a relation between $T_i$ and $T_{i+1}$.

**Observation 3.** If $u \in T_i$ has a non-empty label, $L(u) = cP$, for some $c \in \Sigma$, then $T_{i+1}$ contains a node $v$ with label $L(v) = P$.

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 $u$ as $v$; otherwise, we create $v$ as a copy of $u$. More precisely, we distinguish a heavy letter $h \in \Sigma$ maximising probability $p_{i}^{\langle X \rangle}(c)$ over $c \in \Sigma$. We reuse $u$ if $L(u)$ 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 character $c$. 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 $L(u) = cP$, we maintain the probability $\text{Prob}_X(cP,i)$ and a pointer to the corresponding node $v \in T_{i,h}$ such that $L(v) = hP$. 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 $u$.

**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 $T_{3,h}$, whereas $T_{3,b}$ 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.

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 $L(v) = hQ$ (the token used to represent $Q \in M_{i+1}$). We traverse the path from $v$ towards the root of $T_i$ maintaining the probability $\text{Prob}_X(L(v'),i)$ 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. Figure 2 illustrates this procedure on an example.

### 4.2.1 Correctness

We shall prove that the procedure described above correctly computes $T_i$ and $M_i$. Due to ?? 3, the trie $T_i$ 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 $m_i(v) \leq t_i(v) = 0$. 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 $u$. However, in that case we have $t_i(v) \geq t_i(u) \geq m_i(u) > 0$, because $h$ is the heavy character. This contradiction concludes the proof.
Hence, we proceed to proving that the final configuration of tokens represents \( M_i \). For this, we observe that our algorithm simulates the greedy procedure in the proof of Lemma 2. In other words, we shall prove that we transformed \( P_{j,i+1} \in M_{i+1} \) to the longest compatible element of \( M_i \) which was still unmatched when we processed token \( j \). Suppose that there was an unmatched string \( P' \in M_i \) longer than \( P_{j,i} \). Let \( P' = cQ' \) and observe that, when processing token \( j \), we visited the node \( v' \) with \( L(v') = hQ' \). 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 \( v' \). For 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 \( O(nz) \). 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 \( O(n) \) per token and \( O(nz) \) in total. The remaining operations (construction of subtrees \( T_{i,c} \)) take \( O(1) \) time per created node. The final tree \( T_1 \) has \( O(nz) \) nodes and the overall number of deleted nodes is \( O(nz) \). Hence, the total number of created nodes is also \( O(nz) \).

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

**Theorem 2.** For a weighted sequence \( X \) of length \( n \) over a constant-sized alphabet, one can construct a \( z \)-estimation in \( O(nz) \) time.

5 Property Indexing Made Simple

Every known solution to the Property Indexing problem makes use of suffix trees; ours is no exception. 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 labelled 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 \( L(v) \) to denote the path-label of a node \( v \), i.e., the concatenation of the edge labels along the path from the root to \( v \). The terminal node corresponding to suffix \( S[i \ldots n] \) 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 \( L(v) = cP \) 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 \( v \) leads to an explicit node of \( T \).

The suffix tree of a string of length \( n \) even over an integer alphabet can be constructed in \( O(n) \) time [9].

5.2 Property Suffix Tree Construction

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

For a given string \( S \), we construct the property suffix tree with respect to property \( \Pi \) 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
Since \( u \) is an ancestor of \( u v \) the suffix tree from \( S \) to \( S \) prefix table and of all covers of a weighted sequence. Our weighted index can be used in both.

Our approach to the first phase is similar to Ukkonen’s suffix tree construction \[21\]. We are to determine the locus \( v_i \) of \( S[1 \ldots \pi[i]] \). 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 read only the first letters 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 a node \( w \) 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 \( 2 + |L(w')| - |L(u_i)| \leq 3 + |L(u_{i+1})| - |L(u_i)| \), which gives \( O(n) \) 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. This lets us appropriately subdivide each edge and compute the lists \( L \) 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 \( O(n) \) time.

**Theorem 3.** For a string \( S \) and property \( \Pi \) represented with a table \( \pi \), the property suffix tree can be computed in \( O(n) \) time. Moreover, this data structure can answer property indexing queries in \( O(|P|) \) time (counting) or \( O(|P| + |Occ_\pi(P, S)|) \) 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 \( \mathcal{S} = (S_j, \pi_j)_{j=1}^{n} \) of \( X \), concatenate all the strings and shift the properties so that a single string \( S \) with property \( \pi \) is obtained. Our weighted index is the property suffix tree of \( S \) and \( \pi \). 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 \( S_j \).

The space complexity of the index is obviously \( O(nz) \), where \( n \) is the length of \( X \). Theorems \[2\] and \[8\] show that the data structure can be constructed in \( O(nz) \) time. The resulting weighted index is very similar to the one constructed in \[3\], 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 \( Occ_{\pi}(P, X) \neq \emptyset \), it suffices to traverse down the property suffix tree and check if it contains a node \( v \) corresponding to \( P \). This search takes \( O(m) \) time, where \( m = |P| \). The two remaining types of operations—counting and reporting—require finding distinct positions in the labels of the terminals in the subtree of \( v \). They can be implemented after additional preprocessing for the colour set size \[11\] and coloured range listing problem \[17\]; details can be found in \[3\]. We obtain the same complexities as in Theorem 16 from \[3\].

**Theorem 4.** For a weighted sequence \( X \) of length \( n \) over a constant-sized alphabet and a threshold \( z \), there is a weighted index of \( O(nz) \) size that can be constructed in \( O(nz) \) time and answers decision and counting queries in \( O(m) \) time and reporting queries in \( O(m + |Occ_{\pi}(P, X)|) \) time.

Other applications of the weighted index mentioned in \[3\] include \( O(nz) \)-time computation of the weighted prefix table and of all covers of a weighted sequence. Our weighted index can be used in both.
7 Approximate Weighted Index

Now 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 \( z \) and are allowed to report occurrences with probability \( \geq \frac{1}{z} - \epsilon \), for a given value of \( \epsilon > 0 \). Let us recall that \([6] \) solve this problem in \( \mathcal{O}(\frac{1}{\epsilon}nz^2) \) space (with \( \Omega(\frac{1}{\epsilon}n^2z^2) \) construction time) with \( \mathcal{O}(m + |\text{Occ}_{\frac{1}{\epsilon}}(P, X)|) \)-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 \( \frac{1}{z} < \epsilon \), then the query is trivial as all the positions in \( X \) can be reported. Henceforth, we assume that \( \frac{1}{z} \geq \epsilon \).

Let us consider a \( z \)-estimation \( S \) for the weighted sequence with \( z = \frac{1}{z} \). Let \( \ell = \left\lfloor \frac{1}{\epsilon} \right\rfloor \). By Definition \([11] \) we can return position \( i \) as an occurrence of \( P \) based on whether \( \text{Count}_S(P, i) \geq \ell \); this is shown in the following lemma.

**Lemma 3.** If \( \text{Count}_S(P, i) \geq \ell \), then \( \text{Prob}_X(P, i) \geq \frac{1}{z} - \epsilon \). If \( \text{Count}_S(P, i) < \ell \), then \( \text{Prob}_X(P, i) < \frac{1}{z} \).

**Proof.** Assume that \( \text{Count}_S(P, i) \geq \ell \). Then

\[
\text{Prob}_X(P, i) \geq \frac{1}{z} \text{Count}_S(P, i) \geq \frac{1}{z} \left\lfloor \frac{1}{\epsilon} \right\rfloor \geq \frac{1}{z} (\frac{1}{\epsilon} - 1) = \frac{1}{z} - \epsilon.
\]

Now assume that \( \text{Count}_S(P, i) < \ell \). As \( \text{Count}_S(P, i) = |\text{Prob}_X(P, i)z| \), this concludes that \( \text{Prob}_X(P, i)z < \ell \), which is equivalent to \( \text{Prob}_X(P, i) < \frac{\ell}{z} = \frac{1}{z} \left\lfloor \frac{1}{\epsilon} \right\rfloor \leq \frac{1}{z} \).

Thus our approximate weighted index for \( X \) is the weighted index for \( X \) constructed for \( z = \frac{1}{z} \). 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 \( \left\lfloor \frac{1}{\epsilon} \right\rfloor \) 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 Section 4.1 in \([18] \)) 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 \([18] \) 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 \( N = \mathcal{O}(nz) = \mathcal{O}(\frac{1}{\epsilon}z) \). To find the documents with at least \( \ell \) occurrences, we compute by doubling the smallest \( k \) such that the last of the top \( k \) documents reported has less than \( \ell \) occurrences. The number of documents reported in the last step of the doubling search will be at most \( 2|\text{Occ}_{\frac{1}{\epsilon}}(P, X)| \) and the total number will not exceed \( 4|\text{Occ}_{\frac{1}{\epsilon}}(P, X)| \).

**Theorem 5.** For a weighted sequence of length \( n \) over a constant-sized alphabet and parameter \( \epsilon > 0 \), the Approximate Weighted Indexing problem can be solved in \( \mathcal{O}(\frac{1}{\epsilon}z) \) space with \( \mathcal{O}(m + |\text{Occ}_{\frac{1}{\epsilon}}(P, X)|) \)-time queries. The construction time is \( \mathcal{O}(\frac{1}{\epsilon}z \log \frac{1}{\epsilon}) \).

8 Randomised Construction with Greater Space Usage

A symbol \( X[i] \) of a weighted sequence \( X \) can be interpreted as a probability distribution on \( \Sigma \), and the whole sequence \( X \) can be interpreted as a product distribution on strings of length \( n \) over \( \Sigma \). 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 \( \mathbb{P}[S[i .. i + |P| - 1] = P] = \text{Prob}_X(P, i) \). This interpretation can be used to provide a randomised construction of families \( S \) of strings with properties equivalent to the weighted sequence \( X \) in a certain sense, weaker than the one used in Definition \([11] \).
Lemma 4. There is a randomised algorithm which, given a weighted sequence $X$ of length $n$ and a threshold parameter $z$, in $O(nz \log(nz))$ time constructs a family $S$ of $k = O(z \log(nz))$ strings $S_j$ with properties $\pi_j$ such that $\text{Count}_S(P, i) > 0$ if and only if $\text{Prob}_X(P, i) \geq \frac{1}{z}$. It succeeds with high probability $(1 - \frac{1}{(nz)^c})$ for arbitrarily large constant $c$.

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

This way, $\text{Count}_S(P, i) > 0$ implies $\text{Prob}_X(P, i) \geq \frac{1}{z}$. On the other hand, if $\text{Prob}_X(P, i) \geq \frac{1}{z}$, then, since $\mathbb{P}[S_j[i \ldots i + |P| - 1] \neq P] = 1 - \text{Prob}_X(P, i)$, we have:

$$\mathbb{P}[	ext{Count}_S(P, i) = 0] = (1 - \text{Prob}_X(P, i))^k \leq e^{-k \text{Prob}_X(P, i)} \leq e^{-(c+2) \log(nz)} = \frac{1}{(nz)^c}.$$

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

We can directly use the same methods as in Section 4 to construct a weighted index from the family of strings constructed in Lemma 4. The space complexity of the resulting index is worse than the one in Theorem 4 by a factor of $\log(nz)$ and the construction is randomised.

Corollary 5. There is a data structure of size $O(nz \log(nz))$ for the Weighted Indexing problem which answers queries in optimal time. It can be constructed using a randomised $O(nz \log(nz))$-time algorithm which returns a valid weighted index with high probability.

The same type of construction can be used 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 5. There is a randomised algorithm which, given a weighted sequence $X$ of length $n$ and a parameter $\epsilon$, in $O(\frac{n}{\epsilon^2} \log(\frac{n}{\epsilon}))$ time constructs a family $S$ of $k = O(\frac{n}{\epsilon^2} \log(\frac{n}{\epsilon}))$ strings $S_j$ with properties $\pi_j$ such that $|\text{Prob}_X(P, i) - \frac{1}{k} \text{Count}_S(P, i)| < \epsilon$ 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 $k = \lceil (c + 2)z \log(nz) \rceil$ strings $S_1, \ldots, S_k$. The properties $\pi_j$ satisfy that $S_j[i \ldots \pi_j[i]]$ is the longest prefix of $S_j[i \ldots n]$ such that $\text{Prob}_X(S_j[i \ldots \pi_j[i]], i) \geq \epsilon$.

Observe that if $\text{Prob}_X(P, i) < \epsilon$, then $\text{Count}_S(P, i) = 0$. On the other hand, if $\text{Prob}_X(P, i) \geq \epsilon$, then $\text{Count}_S(P, i) \sim \text{Bin}(k, \text{Prob}_X(P, i))$. Consequently, Hoeffding’s inequality implies

$$\mathbb{P} [|\text{Prob}_X(P, i) - \frac{1}{k} \text{Count}_S(P, i)| > \epsilon] \leq 2e^{-\epsilon^2 k} = 2e^{-(c+2) \log \frac{n}{\epsilon}} \leq 2(\frac{\epsilon}{n})^c + 2.$$

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

We can use this family of strings to construct an approximate weighted index using top-$k$ document retrieval just as in Section 7. We arrive at the following construction with space complexity greater than the one from Theorem 5 by a factor of $\frac{1}{c} \log(\frac{n}{\epsilon})$ (and has a randomised construction).

Corollary 7. There is a data structure of size $O(\frac{n}{\epsilon^2} \log(\frac{n}{\epsilon}))$ which solves the Approximate Weighted Indexing problem with $O(m + \text{Occ}_{\frac{1}{\epsilon}}(P, X))$-time queries. It can be constructed using a randomised $O(\frac{n}{\epsilon^2} \log^2 \frac{n}{\epsilon})$-time algorithm which returns a valid approximate weighted index with high probability.
9 Conclusions

In this article we present an efficient index for Weighted Pattern Matching along with new combinatorial insights into the nature of weighted sequences. We have produced an implementation of the index (see https://bitbucket.org/kociumaka/weighted_index) that we have validated for correctness and efficiency against known weighted pattern matching algorithms [15, 4, 5]. Our implementation supports decision, counting, and reporting variants of queries; however, only decision operations were implemented in worst-case optimal time.

Let us mention that our results can be extended to integer alphabets $\Sigma$, i.e., $\Sigma \subseteq \{1, \ldots, n^{O(1)}\}$, without influencing the space and construction time. We have omitted the description of this extension and preferred to focus on the basic case of a constant-sized alphabet that is also most relevant in practice.

Finally, our ideas can be used to improve the solution for the Generalised Weighted Indexing problem from [6]. They 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 [6] the input weighted sequence is transformed using the reduction of [2] into a special weighted sequence of length $O(nz^2 \log z)$ 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 $O(m + m \cdot |\text{Occ}_P(P, X)|)$ time.

Our $z$-estimation $\mathcal{S}$ can be transformed into a special weighted sequence of length $O(nz)$ 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 [6].

Corollary 8. For a weighted sequence of length $n$ over an integer alphabet, the Generalised Weighted Indexing problem can be solved with $O(m + m \cdot |\text{Occ}_P(P, X)|)$-time queries with an index of size $O(nz)$.

Acknowledgement

We thank an anonymous referee of the previous version of the paper 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.

References

[1] Charu C. Aggarwal and Philip S. Yu. A survey of uncertain data algorithms and applications. IEEE Transactions on Knowledge and Data Engineering, 21(5):609–623, 2009.

[2] Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. Theoretical Computer Science, 395(2-3):298–310, April 2008.

[3] Carl Barton, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Efficient index for weighted sequences. In Roberto Grossi and Moshe Lewenstein, editors, Combinatorial Pattern Matching, CPM 2016, volume 54 of LIPIcs, pages 4:1–4:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[4] Carl Barton, Chang Liu, and Solon P. Pissis. Fast average-case pattern matching on weighted sequences, 2015.

[5] Carl Barton, Chang Liu, and Solon P. Pissis. On-line pattern matching on uncertain sequences and applications. In T.-H. H. Hubert Chan, Minming Li, and Lusheng Wang, editors, Combinatorial Optimization and Applications, COCOA 2016, volume 10043 of LNCS, pages 547–562. Springer, 2016.
[6] 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, 19th International Conference on Extending Database Technology, EDBT 2016, pages 401–412. OpenProceedings.org, 2016.

[7] Manolis Christodoulakis, Costas S. Iliopoulos, Laurent Mouchard, and Kostas Tsichlas. Pattern matching on weighted sequences. In Algorithms and Computational Methods for Biochemical and Evolutionary Networks, CompBioNets 2004, KCL publications, 2004.

[8] Nilesh N. Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. Communications of the ACM, 52(7):86–94, 2009.

[9] Martin Farach. Optimal suffix tree construction with large alphabets. In 38th IEEE Annual Symposium on Foundations of Computer Science, FOCS 1997, pages 137–143. IEEE Computer Society, 1997.

[10] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association, 58(301):13–30, 1963.

[11] Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, Combinatorial Pattern Matching, CPM 1992, volume 644 of LNCS, pages 230–243. Springer, 1992.

[12] 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. Fundamenta Informaticae, 71(2-3):259–277, 2006.

[13] Costas S. Iliopoulos and Mohammad Sohel Rahman. Faster index for property matching. Information Processing Letters, 105(6):218–223, 2008.

[14] M. T. Juan, J. J. Liu, and Y. L. Wang. Errata for “faster index for property matching”. Information Processing Letters, 109(18):1027–1029, 2009.

[15] Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewska. Pattern matching and consensus problems on weighted sequences and profiles. In Seok-Hee Hong, editor, Algorithms and Computation, ISAAC 2016, volume 64 of LIPIcs, pages 46:1–46:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[16] Tsvi Kopelowitz. The property suffix tree with dynamic properties. Theor. Comput. Sci., 638:44–51, 2016.

[17] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, 13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002, pages 657–666. ACM/SIAM, 2002.

[18] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In Yuval Rabani, editor, 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, pages 1066–1077. SIAM, 2012.

[19] Sanguthevar Rajasekaran, X. Jin, and John L. Spouge. The efficient computation of position-specific match scores with the fast Fourier transform. Journal of Computational Biology, 9(1):23–33, 2002.

[20] Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. In Alok Aggarwal and C. Pandu Rangan, editors, Algorithms and Computation, ISAAC 1999, volume 1741 of LNCS, pages 225–236. Springer, 1999.

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